Università Politecnica delle Marche

Multirate Digital Signal Processing And Adaptive Filter Banks

Confronto fra algoritmo LMS e Fast Deconvolution per la cancellazione del crosstalk

Matteo Orlandini e Jacopo Pagliuca

Prof.ssa Stefania Cecchi
Dott.ssa Valeria Bruschi

2 settembre 2021

Indice

1 Introduzione

La presente tesina ha lo scopo di riprodurre i risultati ottenuti nel paper [8], il quale mostra un confronto tra l'algoritmo Least Mean Square e Fast Deconvolution per la cancellazione della diafonia che svolge un ruolo importante nell'ascolto di segnali binaurali tramite altoparlanti.

A differenza dei sistemi stereo classici, un sistema audio 3D permette di posizionare i suoni intorno ad un ascoltatore in modo che questi siano percepiti come provenienti da punti arbitrari nello spazio. In questo modo, l'audio 3D può aumentare il senso di realismo nella musica o nei film e può essere di grande beneficio nella realtà virtuale, nella realtà aumentata, nelle videoconferenze da remoto o per l'intrattenimento casalingo.

La percezione del suono virtuale è ottenuta sintetizzando una coppia di segnali binaurali da un segnale di ingresso monoaurale con le informazioni acustiche 3D fornite: la distanza e la direzione della sorgente sonora rispetto all'ascoltatore. In particolare, il senso dell'orientamento può essere dato dalle funzioni di trasferimento relative alla testa (HRTF), le quali si possono ottenere in modo sperimentale o teorico. Per fornire segnali binaurali, il modo più semplice è attraverso le cuffie. Tuttavia, in molti scenari, ad esempio, a casa o in teleconferenza, molti ascoltatori preferiscono non indossare le cuffie. Se vengono utilizzati altoparlanti, la riproduzione di segnali binaurali all'orecchio dell'ascoltatore non è semplice. Ogni orecchio riceve una cosiddetta componente di diafonia, inoltre, i segnali diretti sono distorti dal riverbero della stanza. Per superare i problemi descritti sopra, è necessario un filtro inverso prima di riprodurre il segnale binaurale attraverso gli altoparlanti.

Poiché occorre invertire i percorsi dell'onda sonora, la cancellazione del crosstalk è un tipico problema di inversione del sistema, che può essere realizzato direttamente o in modo adattivo. Nel primo caso, si presuppone che i percorsi di trasferimento acustico dagli altoparlanti alle orecchie siano noti. Questi percorsi sono spesso descritti dalle HRTF.

In questo progetto vengono analizzati, implementati e confrontati due algoritmi di cancellazione del crosstalk: LMS (Least Mean Square) e Fast Deconvolution. Questo viene fatto dapprima tramite degli script in Matlab che poi saranno replicati in C++ utilizzando il software NuTech che permette il processamento dei segnali audio in tempo reale.

La sezione 2 descrive gli algoritmi più comuni in letteratura, mentre la 3 illustra le funzioni di trasferimento della testa che sono state utilizzate: la loro origine e il significato. Nel capitolo 4 è presente la teoria dietro i due algoritmi confrontati. Le sezioni 5 e 6 presentano e spiegano il codice che è stato sviluppato. Infine i capitoli 7 e 8 mostrano i risultati e li confrontano con quelli ottenuti nel paper di riferimento per verificare l'efficacia del codice.

2 Stato dell'arte

Il concetto di cancellazione ed equalizzazione della diafonia è stato introdotto da Atal e Schroeder e Bauer nei primi anni Sessanta. Da allora sono stati presentati diversi algoritmi sofisticati per la cancellazione della diafonia, utilizzando due o più altoparlanti, in modo diretto o adattivo.

Supponendo che i percorsi di trasferimento acustico dagli altoparlanti alle orecchie siano noti, il metodo di implementazione diretto calcola il filtro di cancellazione della diafonia invertendo direttamente le funzioni di trasferimento HRTF. Solitamente viene utilizzato un manichino di una testa con dei microfoni all'interno per stimare queste funzioni.

Il metodo della stima diretta può essere implementato nel dominio del tempo o della frequenza. I primi sono generalmente dispendiosi dal punto di vista computazionale, mentre i secondi hanno una complessità inferiore. D'altra parte, gli algoritmi nel dominio del tempo hanno prestazioni migliori di quelli nel dominio della frequenza, data la stessa lunghezza del filtro di cancellazione.

Ad esempio, il metodo nel dominio della frequenza chiamato Fast Deconvolution, che è stato dimostrato essere molto utile e facile da usare in diversi casi pratici, può subire un effetto di convoluzione circolare quando i filtri inversi non sono sufficientemente lunghi rispetto alla durata del percorso acustico.

Nei metodi di implementazione adattivi, il filtro di cancellazione della diafonia è calcolato adattando i relativi coefficienti usando i segnali di feedback ricevuti da microfoni in miniatura collocati nelle orecchie dell'utente. Molti metodi adattativi impiegano tipicamente qualche variazione degli algoritmi LMS o RLS (Recursive Least Squares). L'algoritmo LMS cerca di ridurre una funzione costo J pari all'errore quadratico medio tra il segnale desiderato e l'uscita:

$$J = E[e[n]^2] = E[(d[n] - y[n])^2],$$

dove E indica il valore medio, $d[n]$ è il segnale desiderato e $y[n]$ è l'uscita del filtro di cancellazione del crosstalk. Questo algoritmo è noto per la sua semplicità e robustezza ed è ampiamente utilizzato, anche se la sua velocità di convergenza è lenta.

Il metodo LMS non è molto adatto per dati real-time, in quanto per stimare i coefficienti del filtro di cancellazione del crosstalk h usando dati acquisiti in modo continuo si preferiscono metodi ricorsivi [3]. L'algoritmo RLS si ottiene pesando in modo esponenziale i dati in modo da rimuovere gradualmente gli effetti dei vecchi dati sui coefficienti del filtro e permettere il tracciamento di segnali che variano lentamente. La funzione costo J in questo caso è data da

$$J(\mathbf{h}_n) = \sum_{i=0}^{n} \lambda^{n-i} e^2[i],$$

dove e è l'errore definito come la differenza tra il segnale desiderato e l'uscita del filtro di cancellazione del crosstalk e $\mathbf{h}_n$ è il vettore contenente i coefficienti del filtro all'istante n. La funzione costo si riduce a quella dell'LMS quando $\lambda = 1$. Il parametro λ è chiamato "forgetting factor" e viene tipicamente scelto nell'intervallo tra 0.98 e 1.

La funzione costo viene minimizzata facendo la derivata parziale rispetto a tutti i coefficienti k del filtro di cancellazione del crosstalk h e ponendo il risultato pari a zero.

$$\frac{\partial J(\mathbf{h}_n)}{\partial h_n[k]} = \sum_{i=0}^{n} 2\lambda^{n-i} e[i] \frac{\partial e[i]}{\partial h_n[k]} = -\sum_{i=0}^{n} 2\lambda^{n-i} e[i] x[i-k] = 0 \quad k = 0, 1, \ldots, p$$

dove p è il numero di coefficienti del filtro h. Successivamente, sostituendo $e[i]$ con la definizione di errore si ha

$$\frac{\partial J(\mathbf{h}_n)}{\partial h_n[k]} = \sum_{i=0}^{n} \lambda^{n-i} \left[d[i] - \sum_{l=0}^{p} h_n[l] x[i-l] \right] x[i-k] = 0 \quad k = 0, 1, \ldots, p \tag{1}$$

dove $\sum_{l=0}^{p} h_n[l] x[i-l]$ rappresenta l'uscita del filtro h con ingresso x. Manipolando l'equazione (1) si ottiene

$$\sum_{l=0}^{p} h_n[l] \left[\sum_{i=0}^{n} \lambda^{n-i} x[i-l] x[i-k] \right] = \sum_{i=0}^{n} \lambda^{n-i} d[i] x[i-k] \quad k = 0, 1, \ldots, p. \tag{2}$$

L'equazione (2) si può esprimere in forma matriciale come

$$\mathbf{R}_x[n]\mathbf{h}_n = \mathbf{r}_{dx}[n] \tag{3}$$

dove $\mathbf{R}_x[n]$ è la matrice di covarianza di $x[n]$ e $\mathbf{r}_{dx}[n]$ è la cross-covarianza, o covarianza incrociata, tra $d[n]$ e $x[n]$. Basandoci sull'espressione (2) possiamo trovare i coefficienti che minimizzano la funzione costo come

$$\mathbf{h}_n = \mathbf{R}_x^{-1}[n]\mathbf{r}_{dx}[n].$$

Occorre ora trovare una forma di soluzione ricorsiva nella forma

$$\mathbf{h}_n = \mathbf{h}_{n-1} + \Delta\mathbf{h}_{n-1}$$

dove $\Delta\mathbf{h}_{n-1}$ indica il fattore di correzione all'istante $n-1$. Si può dimostrare che
$$\Delta\mathbf{h}_{n-1} = \mathbf{g}[n]\alpha[n]$$
dove
$$\mathbf{g}[n] = \mathbf{P}[n-1]\mathbf{x}[n]\left\{\lambda + \mathbf{x}^T[n]\mathbf{P}[n-1]\mathbf{x}[n]\right\}^{-1}$$
in cui
$$\mathbf{P}[n] = \mathbf{R}_x^{-1}[n] = \lambda^{-1}\mathbf{P}[n-1] - \mathbf{g}[n]\mathbf{x}^T[n]\lambda^{-1}\mathbf{P}[n-1]$$
e
$$\alpha[n] = d[n] - \mathbf{x}^T[n]\mathbf{h}_{n-1}$$
dove $\alpha[n]$ è l'*errore a priori*, differente dall'*errore a posteriori* mostrato nella formula seguente
$$e[n] = d[n] - \mathbf{x}^T[n]\mathbf{h}_n. \tag{4}$$

Nella derivazione dell'RLS, gli input sono considerati deterministici, mentre per LMS sono stocastici. L'algoritmo RLS può arrivare a convergenza in modo più veloce di LMS, aumentando però il carico computazionale.

Un ulteriore algoritmo derivato da LMS è FLMS (Fast Least Mean Square), descritto in [1], viene implementato nel dominio della frequenza e richiede uno sforzo computazionale minore rispetto alla sua controparte nel dominio del tempo. Se l'input del filtro adattativo è un segnale discreto x_j, allora l'algoritmo LMS è descritto dalla seguente equazione
$$\mathbf{h}_{j+1} = \mathbf{h}_j + 2\mu\epsilon_j\mathbf{x}_j \tag{5}$$
dove $\mathbf{h}_j$ e $\mathbf{x}_j$ sono vettori che rappresentano, rispettivamente, i pesi del filtro e le uscite nella linea di ritardo all'istante j.
$$\mathbf{h}_j^T = [h_{1,j}, h_{2,j}, \ldots, h_{n,j}]$$
$$\mathbf{x}_j^T = [x_j, x_{j-1}, \ldots, x_{j-n+1}]$$
L'errore ϵ_j è la differenza tra il segnale desiderato d_j e l'uscita y_j, dove $y_j = \mathbf{x}_j^T\mathbf{h}_j$. Per implementare l'equivalente dell'algoritmo LMS in frequenza usando un filtro con n tappi è conveniente dividere il segnale in blocchi da n valori. Durante il k-esimo blocco, l'uscita del filtro è data da
$$y_{kn+i} = \mathbf{x}_{kn+i}^T\mathbf{h}_k \quad 0 \leq i \leq n-1 \tag{6}$$
dove $\mathbf{h}_k$ è il vettore contenente i coefficienti del filtro nel dominio del tempo durante il k-esimo blocco. Dopo aver processato il k-esimo blocco, il vettore $\mathbf{h}_k$ viene aggiornato iterando l'equazione (5) su n punti come di seguito
$$\mathbf{h}_{k+1} = \mathbf{h}_k + 2\mu\sum_{i=0}^{n-1}\epsilon_{kn+i}x_{kn+i} = \mathbf{h}_k + 2\mu\nabla_k. \tag{7}$$

Le equazioni (6) e (7) possono essere implementate nel dominio della frequenza. Si inizia riscrivendo (6) come convoluzione dei coefficienti del filtro con l'ingresso

$$y_{kn+j} = \sum_{i=0}^{n-1} h_{i,k} x_{kn+j-i}. \tag{8}$$

Per implementare (8) con il metodo *overlap and add* usando la FFT occorre fare un padding di n zeri e usare una FFT a $2n$ punti. Sia $\mathbf{H}_k$ un vettore di lunghezza $2n$ i cui elementi sono i coefficienti nel dominio della frequenza di $\mathbf{h}$ paddati con n zeri

$$\mathbf{H}_k^T = \mathrm{FFT}[\mathbf{h}_k^T, 0, \ldots, 0].$$

Sia $\mathbf{X}_k$ la FFT a $2n$ punti del $(k-1)$-esimo e e del k-esimo blocco:

$$\mathbf{X}_k^T = \mathrm{FFT}[x_{(k-1)n}, \ldots, x_{kn-1}, x_{kn}, \ldots, x_{kn+n-1}].$$

La convoluzione in (8) si realizza calcolando

$$[y_{kn}, \ldots, y_{kn+n-1}]^T = \text{ultimi } n \text{ termini della IFFT di } \{\mathbf{H}_k \otimes \mathbf{X}_k\} \tag{9}$$

dove $\otimes$ indica la moltiplicazione elemento per elemento dei vettori $\mathbf{H}_k$ e $\mathbf{X}_k$. L'equazione (9) permette di calcolare i valori dell'uscita del k-esimo blocco. Si può osservare che un filtro a n tappi nel dominio del tempo richiede un filtro a $2n$ coefficienti nel dominio della frequenza.

Per implementare l'aggiornamento dei tappi del filtro come mostrato nell'equazione (7) si può scrivere

$$\nabla_{j,k} = \sum_{i=0}^{n-1} \epsilon_{kn+i} x_{kn+i-(j-1)} \quad 1 \le j \le n$$

Il valore ∇_k può essere calcolato tramite la FFT se prima viene determinato l'errore $\mathbf{E}_k$ preceduto da n zeri

$$\mathbf{E}_k = \mathrm{FFT}[0, \ldots, 0, (d_{kn} - y_{kn}), \ldots, (d_{kn+n-1} - y_{kn+n-1})]^T.$$

Successivamente, si calcola

$$\nabla_k = \text{primi } n \text{ termini della IFFT di } \{\mathbf{E}_k \otimes \mathbf{X}_k^*\}$$

dove * indica il complesso coniugato. Infine, vengono calcolati i coefficienti del filtro in frequenza come di seguito

$$\mathbf{H}_{k+1} = \mathbf{H}_k + 2\mu \ \mathrm{FFT} \begin{bmatrix} \nabla_k \\ 0 \\ \vdots \\ 0 \end{bmatrix} \tag{10}$$

con n zeri sotto il vettore ∇_k. Se gli ultimi n valori della IFFT dei coefficienti iniziali di $\mathbf{H}_0$ sono forzati a zero, allora l'equazione (10) è l'implementazione esatta di (7) nel dominio della frequenza.

Per ognuno degli n valori del blocco, l'algoritmo FLMS richiede cinque FFT a $2n$ punti e due moltiplicazioni complesse a $2n$ punti. Per ingressi reali, le trasformate sono simmetriche, quindi richiedono il calcolo solamente dei primi $n + 1$ punti. Assumendo che una moltiplicazione complessa sia equivalente a quattro moltiplicazioni reali, si può dimostrare il seguente rapporto

$$\frac{\text{Moltiplicazioni reali FLMS}}{\text{Moltiplicazioni reali LMS}} = \frac{5\left(\log\frac{n}{2}\right) + 14}{n}. \tag{11}$$

Per $n \geq 64$ il rapporto (11) diventa minore di 1, quindi le moltiplicazioni reali dell'algoritmo LMS sono maggiori di quelle di FLMS.

Sebbene molti algoritmi siano stati proposti, il metodo adattativo viene usato più in ambito accademico che nella vita di tutti i giorni. Il motivo è che le persone che non vogliono usare le cuffie probabilmente non vorrebbero utilizzare neanche un paio di microfoni nelle orecchie per ottimizzare la riproduzione degli altoparlanti. Una limitazione di un sistema di cancellazione della diafonia nasce dal fatto che ogni movimento dell'ascoltatore che supera i $75 - 100$ mm può cambiare completamente l'effetto desiderato. Questo problema può essere risolto seguendo in tempo reale la testa dell'ascoltatore nello spazio 3D. La posizione della testa viene determinata da un localizzatore magnetico o basato su fotocamera, quindi i filtri HRTF e quelli di cancellazione della diafonia sono aggiornati in tempo reale in base alla posizione dell'ascoltatore. In questo caso, il filtro di cancellazione della diafonia richiede un'elevato sforzo computazionale per essere aggiornato in tempo reale in base alla posizione della testa [10].

In acustica, la deconvoluzione a singolo canale è particolarmente utile poiché può compensare la risposta di trasduttori imperfetti come cuffie, altoparlanti e amplificatori. La deconvoluzione multicanale è necessaria nella progettazione di sistemi di cancellazione della diafonia e sistemi di imaging di sorgenti virtuali.

La Fast Deconvolution è un metodo molto veloce per calcolare una matrice di filtri digitali che può essere utilizzata per controllare le uscite di un impianto multicanale. Questo metodo è tipicamente più veloce di diversi ordini di grandezza rispetto ai metodi nel dominio del tempo e combina i principi dell'inversione dei minimi quadrati nel dominio della frequenza e il metodo di regolarizzazione di ordine zero. I metodi di regolarizzazione rinunciano a trovare la soluzione esatta del problema, calcolando invece la soluzione di un problema leggermente diverso ma meglio condizionato.

L'algoritmo presuppone che sia possibile utilizzare filtri ottimali lunghi, e funziona bene solo quando il parametro di regolarizzazione è impostato in modo appropriato. Nella pratica questo parametro viene determinato per tentativi.

Una tipica situazione di ascolto con due altoparlanti con cancellazione del closstalk è rappresentata in figura 1. La rappresentazione in frequenza dei segnali desiderati binaurali è indicata con X_1 e X_2 per i suoni che raggiungono rispettivamente orecchio destro e sinistro. Con Y_1 e Y_2 invece, si indicano i suoni che effettivamente raggiungono l'ascoltatore, attraversando il sistema. $C_{i,j}$ sono le HRTF dagli altoparlanti alle orecchie dell'ascoltatore, $H_{i,j}$ sono i filtri di cancellazione della diafonia, per $i, j = 1, 2$.

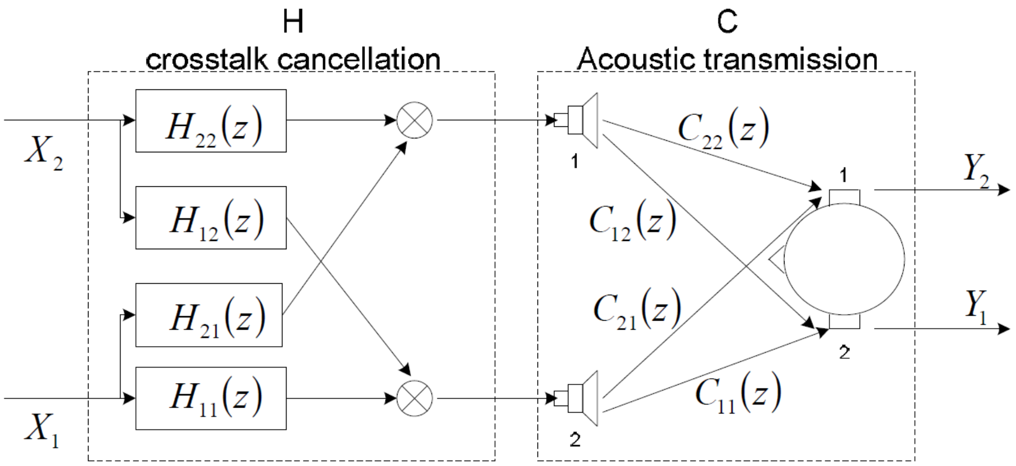


Figura 1: Diagramma a blocchi del sistema sonoro usato

Il sistema in 1 può essere rappresentato nel dominio della frequenza come:

$$\mathbf{Y} = \mathbf{CHX} \tag{12}$$

ovvero

$$\begin{bmatrix} Y_1 \\ Y_2 \end{bmatrix} = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} \begin{bmatrix} H_{11} & H_{12} \\ H_{21} & H_{22} \end{bmatrix} \begin{bmatrix} X_1 \\ X_2 \end{bmatrix} \tag{13}$$

L'equazione (13) può essere riscritta come:

$$\begin{bmatrix} Y_1 \\ Y_2 \end{bmatrix} = \begin{bmatrix} C_{11}X_1 & C_{12}X_1 & C_{11}X_2 & C_{12}X_2 \\ C_{21}X_1 & C_{22}X_1 & C_{21}X_2 & C_{22}X_2 \end{bmatrix} \begin{bmatrix} H_{11} \\ H_{21} \\ H_{12} \\ H_{22} \end{bmatrix} \tag{14}$$

Per avere il segnale in uscita uguale a quello desiderato H dovrebbe essere l'inversa di C. La diretta inversa di C però non garantisce la cancellazione del crosstalk in quanto gli elementi potrebbero non soddisfare la condizione di fase minima.

3 Dataset

Il dataset usato è composto ad un ampio set di misurazioni della funzione di trasferimento relativa alla testa (HRTF) di un microfono dummy-head KEMAR. Le misurazioni consistono nelle risposte impulsive dell'orecchio sinistro e destro di un altoparlante Realistic Optimus Pro 7 montato a 1,4 metri dal KEMAR. Sono state utilizzate sequenze binarie pseudo-casuali per ottenere le risposte impulsive a una frequenza di campionamento di 44.1 kHz. [2]

Le misurazioni sono state effettuate nella camera anecoica del MIT. Il KEMAR è stato montato in posizione verticale su un giradischi motorizzato che può essere ruotato con precisione sotto il controllo del computer. L'altoparlante è stato montato su un supporto a braccio che ha consentito il posizionamento accurato dell'altoparlante a qualsiasi elevazione rispetto al KEMAR. Pertanto, le misurazioni sono state effettuate un'elevazione alla volta, impostando l'altoparlante all'altezza corretta e quindi ruotando il KEMAR su ciascun azimut.

I dati HRTF vengono archiviati nelle directory per elevazione. Ogni nome di directory ha il formato "elevEE", dove EE è l'angolo di elevazione. All'interno di ogni directory, il nome di ogni file ha il formato "XEEeAAAa.wav" dove X può essere "L" o "R" rispettivamente per la risposta dell'orecchio sinistro e destro, "EE" è l'angolo di elevazione della sorgente in gradi, da -40° a 90°, e AAA è l'azimut della sorgente in gradi, da 0° a 355°. Gli angoli di elevazione e azimut indicano la posizione della sorgente rispetto al KEMAR, in modo che, ad esempio, in corrispondenza dell'elevazione 0 e azimut 0 sia di fronte al KEMAR , l'elevazione 90 è direttamente sopra il KEMAR, elevazione 0 e azimut 90 è a destra del KEMAR. Ad esempio, il file "R-20e270a.wav" è la risposta dell'orecchio destro, con la sorgente 20 gradi sotto il piano orizzontale e 90 gradi a sinistra della testa.

4 Descrizione teorica degli algoritmi

4.1 LMS

Per la cancellazione del crosstalk l'algoritmo più comune è quello dell'LMS che nonostante sia semplice e accurato, ha una veloce convergenza [9], [4]. La figura 2 mostra il diagramma di cancellazione del crosstalk usando LMS, x indica l'ingresso, y l'uscita, d il segnale desiderato ed e l'errore.

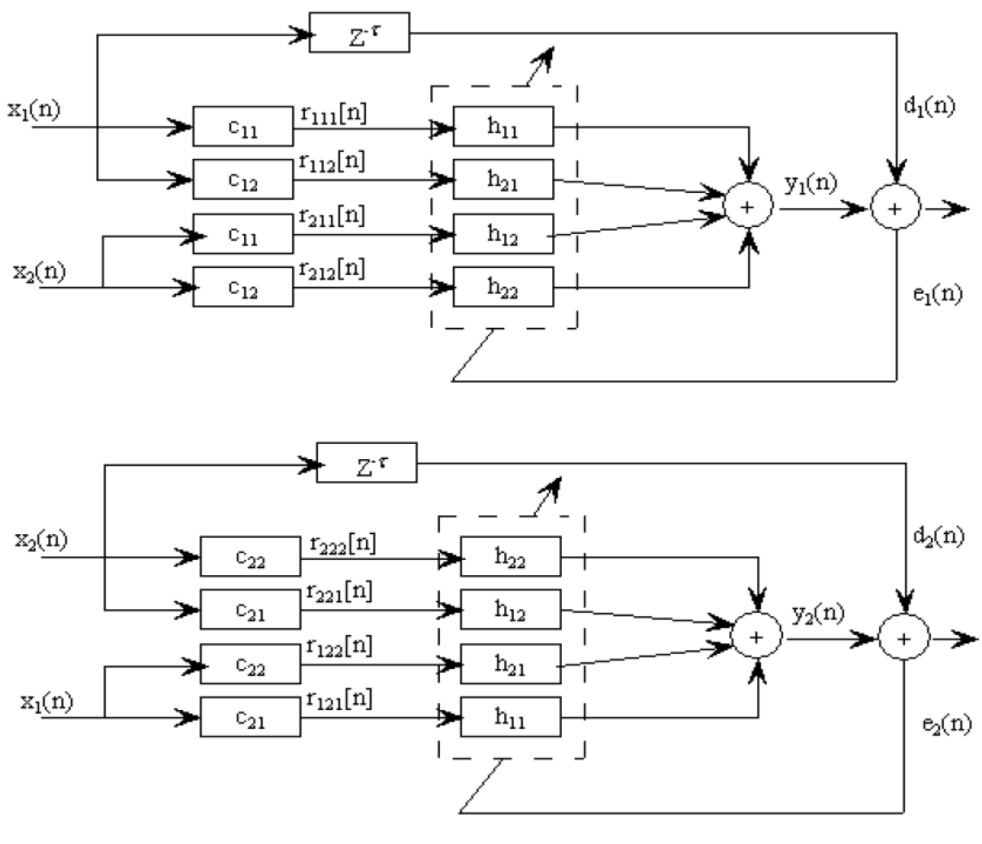


Figura 2: Diagramma a blocchi della cancellazione del crosstalk usando LMS

Le variabili con lettera minuscola rappresentano il segnale nel dominio del tempo e τ rappresenta il ritardo del sistema. Per sistemi causali, l'uscita $y_i[n]$ dovrebbe essere idealmente una versione ritardata dell'ingresso $x_i[n]$. Le sequenze r_{ilm} sono i segnali $x_i[n]$ di ingresso filtrati dalle funzioni c_{lm} che hanno M tappi nel dominio del tempo.

In generale, si può scrivere il filtraggio nell'n-esimo istante di un segnale $\mathbf{a}[\mathbf{n}] = [a[n], a[n-1], \ldots, a[n-M+1]]^T$ con un filtro $\mathbf{b}[\mathbf{n}] = [b_0[n], b_1[n], \ldots, b_{M-1}[n]]^T$

di lunghezza M come

$$y[n] = \sum_{j=0}^{M-1} b_j[n]a[n-j] = \mathbf{b}^T[\mathbf{n}] \cdot \mathbf{a}[\mathbf{n}]. \tag{15}$$

Il filtraggio dei segnali di riferimento $x_i[n]$ con le hrir c_{lm} è dato dalla seguente equazione

$$r_{ilm}[n] = \sum_{j=0}^{M-1} c_{lm}[j]x_i[n-j]. \tag{16}$$

Si può applicare la formula (15) all'equazione (16) per calcolare le $r_{ilm}[n]$ nel seguente modo:

$$r_{ilm}[n] = \sum_{j=0}^{M-1} c_{lm,j}[n]x_i[n-j] = \mathbf{c_{lm}}^T[\mathbf{n}] \cdot \mathbf{x_i}[\mathbf{n}]. \tag{17}$$

Il segnale ricevuto ad ogni orecchio è dato da:

$$y_i[n] = r_{1i1}[n] \circledast h_{11}[n] + r_{1i2}[n] \circledast h_{21}[n] + r_{2i1}[n] \circledast h_{12}[n] + r_{2i2}[n] \circledast h_{22}[n] \tag{18}$$

dove i, l, m assumono i valori 1 o 2 e $\circledast$ rappresenta la convoluzione. Il criterio dell'algoritmo LMS è la minimizzazione della funzione costo

$$J = E[e[n]^2] = E[(d[n] - y[n])^2] \tag{19}$$

dove $e[n]$, $d[n]$ e $y[n]$ sono definiti come

$$e[n] = \begin{bmatrix} e_1[n] \\ e_2[n] \end{bmatrix}, \quad d[n] = \begin{bmatrix} d_1[n] \\ d_2[n] \end{bmatrix}, \quad y[n] = \begin{bmatrix} y_1[n] \\ y_2[n] \end{bmatrix} \tag{20}$$

La minimizzazione di J avviene con il metodo steepest descend. Nel dominio del tempo discreto il sistema si può scrivere come

$$\begin{bmatrix} y_1[n] \\ y_2[n] \end{bmatrix} = \begin{bmatrix} c_{11}[n] \circledast x_1[n] & c_{12}[n] \circledast x_1[n] & c_{11}[n] \circledast x_2[n] & c_{12}[n] \circledast x_2[n] \\ c_{21}[n] \circledast x_1[n] & c_{22}[n] \circledast x_1[n] & c_{21}[n] \circledast x_2[n] & c_{22}[n] \circledast x_2[n] \end{bmatrix} \circledast \begin{bmatrix} h_{11}[n] \\ h_{21}[n] \\ h_{12}[n] \\ h_{22}[n] \end{bmatrix} \tag{21}$$

Usando (16), l'equazione (21) diventa

$$\begin{bmatrix} y_1[n] \\ y_2[n] \end{bmatrix} = \begin{bmatrix} r_{111}[n] & r_{112}[n] & r_{211}[n] & r_{212}[n] \\ r_{121}[n] & r_{122}[n] & r_{221}[n] & r_{222}[n] \end{bmatrix} \circledast \begin{bmatrix} h_{11}[n] \\ h_{21}[n] \\ h_{12}[n] \\ h_{22}[n] \end{bmatrix} \tag{22}$$

Per ottenere l'errore si calcola la differenza fra il segnale desiderato è l'uscita effettiva.

$$e_1^{(1)} = d_1 - (r_{111} \circledast h_{11} + r_{112} \circledast h_{21} + r_{211} \circledast h_{12} + r_{212} \circledast h_{22})$$

l'aggiornamento avviene come di seguito:

$$h^{(k+1)} = h^{(k)} - \mu e_i^{(k)} \cdot \mathbf{r}_i^T \tag{23}$$

dove $\mathbf{r}_i$ è l'i-esima riga della matrice che contiene le r_{ilm} nell'equazione (22) e $e_i^{(k)}$ è l'errore k-esimo step e all'i-esima riga. Esplicitando l'equazione (23), si ottiene

$$\begin{aligned}
h_{11}^{(k+1)}[n] &\leftarrow h_{11}^{(k)}[n] - \mu e_1^{(1)} \cdot r_{111}[n] \\
h_{21}^{(k+1)}[n] &\leftarrow h_{21}^{(k)}[n] - \mu e_1^{(1)} \cdot r_{112}[n] \\
h_{12}^{(k+1)}[n] &\leftarrow h_{12}^{(k)}[n] - \mu e_1^{(1)} \cdot r_{211}[n] \\
h_{22}^{(k+1)}[n] &\leftarrow h_{22}^{(k)}[n] - \mu e_1^{(1)} \cdot r_{212}[n]
\end{aligned}$$

Allo stesso modo, per il secondo canale l'errore è definito come:

$$e_2^{(2)} = d_2 - (r_{121} \circledast h_{11} + r_{122} \circledast h_{21} + r_{221} \circledast h_{12} + r_{222} \circledast h_{22})$$

$$\begin{aligned}
h_{11}^{(k+1)}[n] &\leftarrow h_{11}^{(k)}[n] - \mu e_2^{(2)} \cdot r_{121}[n] \\
h_{21}^{(k+1)}[n] &\leftarrow h_{21}^{(k)}[n] - \mu e_2^{(2)} \cdot r_{122}[n] \\
h_{12}^{(k+1)}[n] &\leftarrow h_{12}^{(k)}[n] - \mu e_2^{(2)} \cdot r_{221}[n] \\
h_{22}^{(k+1)}[n] &\leftarrow h_{22}^{(k)}[n] - \mu e_2^{(2)} \cdot r_{222}[n]
\end{aligned}$$

4.2 Fast Deconvolution

La deconvoluzione [5] [6] [7], nella sua forma più elementare, può essere descritta come il compito di calcolare l'input di un sistema a tempo discreto conoscendo il suo output. Di solito si presume che il sistema sia lineare e che la relazione input output sia nota con precisione. Consideriamo una funzione costo del tipo

$$J = E + \beta V(f) \tag{24}$$

dove E è una misura dell'errore della pressione sonora

$$E = |Y_1 - X_1|^2 + |Y_2 - X_2|^2 \tag{25}$$

e V è una funzione della frequenza che indica il costo computazionale. Il numero $\beta \geq 0$ è un parametro di regolarizzazione che determina quanto peso assegnare alla funzione V. La regolarizzazione dipendente dalla frequenza viene utilizzata per prevenire picchi elevati nella risposta in ampiezza dei filtri ottimali. Poiché non sappiamo a priori se la matrice C è non singolare per determinate frequenze, la matrice H di cancellazione del crosstalk può contenere valori molto alti. All'aumentare di β da zero a infinito, J cambia gradualmente dalla minimizzazione della sola funzione di errore E alla minimizzazione dello sforzo computazionale V.

Siano S i segnali trasferiti agli altoparlanti facendo passare il segnale X attraverso la matrice di cancellazione del crosstalk H. Otteniamo

$$V(f) = S_b^+ S_b \tag{26}$$

con

$$S_b = BS = BHX \tag{27}$$

dove B è una matrice 2×2 e il simbolo $^+$ indica l'inversa generalizzata della matrice S_b. La soluzione approssimata della funzione J è definita da

$$H(z) = \left[C^T(z^{-1})C(z) + \beta B^T B)\right]^{-1} C^T(z^{-1}) \tag{28}$$

dove l'apice ^{T} denota la trasposta della matrice. Se la matrice B è uguale alla matrice identità I, si ottiene $S_b = S$, dunque l'equazione (28) diventa

$$H(z) = \left[C^T(z^{-1})C(z) + \beta I)\right]^{-1} C^T(z^{-1})z^{-m} \tag{29}$$

dove la componente z^{-m} implementa un ritardo di m campioni. Un ritardo di modellazione viene utilizzato per garantire che la rete di cancellazione del cross-talk funzioni bene non solo in termini di ampiezza, ma anche in termini di fase.

Le equazioni (28) e (29) rappresentano una espressione di $H(z)$ nel dominio continuo della frequenza. Se, come nel nostro caso, il dominio della frequenza è discreto, occorre usare la formula (30), in cui il valore di $H[k]$ è dato da

$$H[k] = \left[C^H[k]C[k] + \beta I)\right]^{-1} C^H[k] \tag{30}$$

dove k indica la k-esima frequenza corrispondente a $\exp(j2\pi k/N)$ e l'apice ^{H} denota l'operatore Hermitiano che fa la trasposta coniugata del suo argomento. Dall'equazione si può osservare che ponendo $\beta = 0$ si ottiene $H = C^{-1}$. In questo caso, poiché $Y = CHX = CC^{-1}X = IX = X$, si ottiene in uscita il segnale d'ingresso.

L'equazione (30) si ricava dalla (29) notando che la trasformata di Fourier a tempo discreto è la valutazione della trasformata zeta sul cerchio unitario, quindi ponendo $z = e^{j\omega}$ si ottiene la DTFT. Si noti che un generico segnale $X(z^{-1}) = X(e^{-j\omega}) = X^*(e^{j\omega})$ corrisponde nel dominio discreto della frequenza a $X^*[k]$ e nel tempo discreto all'inversione temporale $x[-n]$. Come precedentemente detto, l'operatore Hermitiano fa la trasposta coniugata del suo argomento, quindi si può capire perché $C^T(z^{-1})$ diventa $C^H[k]$ nell'equazione (30).

Il ritardo di m campioni z^{-m} viene esplicitato nella risposta impulsiva che viene calcolata implementando i seguenti passi:

1. si calcola la matrice 2×2 $C[k]$ tramite una FFT delle risposte impulsive $c_{ij}[n]$ con $i = 1, 2$ e $j = 1, 2$. Ad esempio, $C_{11}[k]$ contiene l'ampiezza della k-esima armonica della FFT di c_{11},
2. si calcola $H[k]$ usando la formula (30),
3. si calcola $h[n]$ facendo una FFT inversa a N punti,
4. si implementa uno shift ciclico di m campioni per ogni elemento di $h_{ij}[n]$ con $i = 1, 2$ e $j = 1, 2$.

Dato che l'uscita Y è data da

$$Y = CHX = \begin{bmatrix} C_{11}X_1 & C_{12}X_1 & C_{11}X_2 & C_{12}X_2 \\ C_{21}X_1 & C_{22}X_1 & C_{21}X_2 & C_{22}X_2 \end{bmatrix} \begin{bmatrix} H_{11} \\ H_{21} \\ H_{12} \\ H_{22} \end{bmatrix} \tag{31}$$

nell'implementazione pratica non si può calcolare tutta l'uscita con la sola operazione matriciale (31), occorre usare la tecnica dell'overlap and save per filtrare l'ingresso X con i filtri C e H. L'overlap and save è utile per eseguire un filtraggio in real time con un filtro a risposta impulsiva finita. Questa tecnica viene usata per fare la convoluzione a blocchi tra un segnale di ingresso $x[n]$ molto lungo e un filtro FIR $h[n]$:

$$y[n] = x[n] * h[n] = \sum_{m=-\infty}^{+\infty} h[m] \cdot x[n-m] = \sum_{m=1}^{M} h[m] \cdot x[n-m] \tag{32}$$

poiché $h[m] \neq 0$ solo per $m \in [1, M]$.

L'overlap and save permette di calcolare dei blocchi di $y[n]$ di lunghezza L e concatenarli insieme per formare il segnale di uscita completo. Si definisce il k-esimo blocco d'ingresso come

$$x_k[n] = \begin{cases} x[n+kL], & 1 \leq n \leq L+M-1 \\ 0 & \text{altrove} \end{cases} \tag{33}$$

quindi il k-esimo blocco di uscita è dato da

$$y_k[n] = x_k[n] * h[n] = \sum_{m=1}^{M} h[m] \cdot x_k[n-m]. \tag{34}$$

Si può definire l'uscita per $kL + M \leq n \leq KL + L + M - 1$, o in modo equivalente per $M \leq n - kL \leq L + M - 1$, nel seguente modo:

$$y[n] = \sum_{m=1}^{M} h[m] \cdot x_k[n - kL - m] = y_k[n - kL]. \tag{35}$$

L'implementazione dell'overlap and save con un filtro composto da $(L+M-1)$ tappi consiste nei seguenti passi:

1. si divide il segnale di ingresso in blocchi di lunghezza L
2. nel caso del primo blocco si aggiungono $M-1$ zeri all'inizio, altrimenti si aggiungono all'inizio del blocco gli ultimi $M-1$ campioni del blocco precedente, come mostrato in figura 3,
3. si fa la FFT a $L+M-1$ punti del k-esimo blocco di ingresso,
4. si fa la FFT a $L+M-1$ punti del filtro FIR $h[n]$
5. si moltiplicano le FFT calcolate per trovare la risposta in frequenza del k-esimo blocco di uscita,
6. si fa la FFT inversa a $L+M-1$ punti del k-esimo blocco di uscita,
7. si scartano i primi $M-1$ punti, ottenendo il k-esimo blocco in uscita di lunghezza L, come mostrato nei blocchi di uscita y_k di figura 3.

Un'implementazione più efficiente è invece quella descritta di seguito e mostrata nella figura 4, in cui i primi due punti precedentemente elencati rimangono uguali:

3. si fa la FFT a $2 \cdot (L+M-1)$ punti del k-esimo e del $(k-1)$-esimo blocco di ingresso lunghi rispettivamente $(L+M-1)$ campioni,

4. si fa la FFT a $2 \cdot (L + M - 1)$ punti del filtro FIR $h[n]$ con un padding di $(L + M - 1)$ zeri

5. si moltiplicano le FFT calcolate per trovare la FFT del blocco di uscita,

6. si fa la FFT inversa del blocco di uscita,

7. si scartano i primi $L + M - 1$ punti nel tempo dell'uscita.

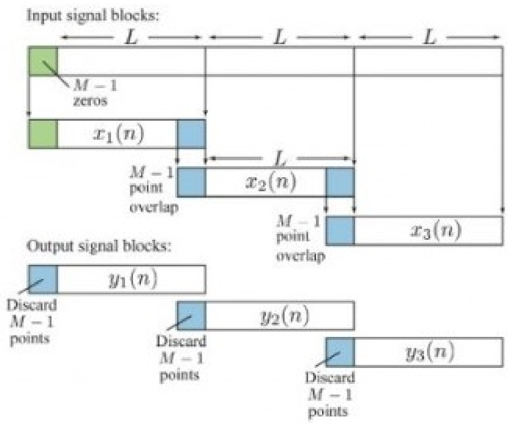


Figura 3: Metodo Overlap and Save

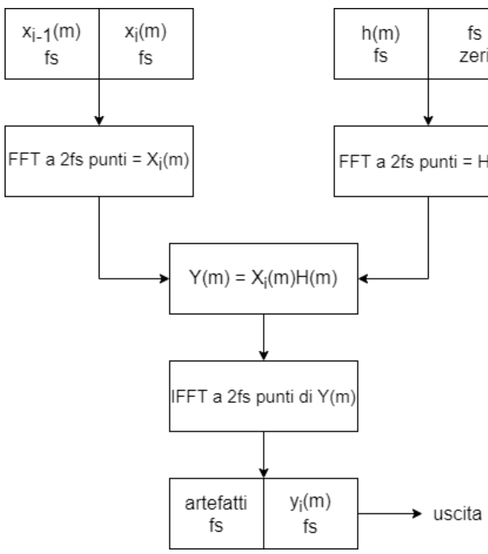


Figura 4: Diagramma dell'implementazione dell'Overlap and Save

5 Codice Matlab

5.1 LMS

Nel codice dell'algoritmo LMS in Matlab si carica il file audio in formato wav "Daft Punk - Get Lucky_cut.wav". Poiché questo formato contiene i campioni del canale sinistro e di quello destro, si dividono i due canali salvando il primo nella variabile `x1` e il secondo in `x2`.

```
[x, Fsample] = audioread('Daft Punk - Get Lucky_cut.wav');
x1 = x(:, 1);
x2 = x(:, 2);
```

Codice 1: Caricamento del file audio

Allo stesso modo vendono caricate le quattro HRIR `c11`, `c12`, `c21` e `c22`.

```
% c11: HRIR left loudspeaker - left ear
[c11,Fs] = audioread('HRTF_measurements/elev0/L0e330a.wav');
% c12: HRIR right loudspeaker - left ear
[c12,~] = audioread('HRTF_measurements/elev0/L0e030a.wav');
% c21: HRIR left loudspeaker - right ear
[c21,~] = audioread('HRTF_measurements/elev0/R0e330a.wav');
% c22: HRIR right loudspeaker - righ ear
[c22,~] = audioread('HRTF_measurements/elev0/R0e030a.wav');
```

Codice 2: Caricamento delle HRIR

Viene salvata la lunghezza dei filtri c_{11}, c_{12}, c_{21} e c_{22} nella variabile `M`. Si ha che $M = 512$ e si imposta il ritardo temporale `tau` pari a `M`. Conoscendo il valore del ritardo temporale, è possibile costruire i due segnali desiderati `d1` e `d2` mettendo `tau` zeri all'inizio del vettore d'ingresso.

```
M = length(c11);       % lunghezza dei filtri c11, c12, c21, c22
% il segnale desiderato e' x ritardato di tau campioni
tau = M;    % ritardo temporale
d1 = [zeros(tau,1); x1(1:end-tau)]; % segnale desiderato 1 ...
    (left)
d2 = [zeros(tau,1); x2(1:end-tau)]; % segnale desiderato 2 ...
    (right)
```

Codice 3: Costruzione del segnale desiderato

Vengono inizializzati i vettori `y1` e `y2` di uscita, i vettori `e1` e `e2` che contengono l'errore per i due canali, i vettori $\mathtt{r}_{\mathtt{ilm}}$ che contengono il filtraggio di `x1` e `x2` con `c11`, `c12`, `c21` e `c22`. Questi vettori sono tutti di lunghezza pari a quella del segnale x, cioè `L`.

```
L = length(x1);
y1 = zeros(L,1);  % output 1 (left)
y2 = zeros(L,1);  % output 2 (right)

e1 = zeros(L,1);  % errore 1 (left)
e2 = zeros(L,1);  % errore 2 (right)

r111 = zeros(L,1);    % uscita di x1 filtrato da c11 per ...
   output y1
r112 = zeros(L,1);    % uscita di x1 filtrato da c12 per ...
   output y1
r211 = zeros(L,1);    % uscita di x2 filtrato da c11 per ...
   output y1
r212 = zeros(L,1);    % uscita di x2 filtrato da c12 per ...
   output y1
r222 = zeros(L,1);    % uscita di x2 filtrato da c22 per ...
   output y2
r221 = zeros(L,1);    % uscita di x2 filtrato da c21 per ...
   output y2
r122 = zeros(L,1);    % uscita di x1 filtrato da c22 per ...
   output y2
r121 = zeros(L,1);    % uscita di x1 filtrato da c21 per ...
   output y2
```

Codice 4: Inizializzazione dei vettori di lunghezza L

Successivamente, vengono inizializzati i vettori di lunghezza `M`, le variabili `h11`, `h12`, `h21` e `h22` contengono i filtri di ricostruzione, i buffer `x1buff` e `x2buff` necessari per il filtraggio con i c e i buffer delle uscite r_{ilm} dove $i, l, m = 0, 1$.

```
x1buff = zeros(M, 1);
x2buff = zeros(M, 1);

r111buff = zeros(M,1);
r112buff = zeros(M,1);
r211buff = zeros(M,1);
r212buff = zeros(M,1);
r222buff = zeros(M,1);
r221buff = zeros(M,1);
r122buff = zeros(M,1);
r121buff = zeros(M,1);
```

Codice 5: Inizializzazione dei vettori di lunghezza M

Nel codice 6 viene usato un $\mu = 10^{-3}$ e viene implementata l'equazione (17) che effettua il filtraggio tra le hrir e il segnale di ingresso. Per fare questo, si utilizza un buffer che aggiunge in testa il campione entrante invertendo il segnale di ingresso. Facendo il prodotto scalare dei buffer di ingresso con le rispettive c_{lm}, si ottiene il filtraggio desiderato.

```
mu = 1e-3;
for n = 1:L
    x1buff = [x1(n); x1buff(1:end-1)];
    x2buff = [x2(n); x2buff(1:end-1)];

    r111(n) = c11'*x1buff;
    r112(n) = c12'*x1buff;
    r211(n) = c11'*x2buff;
    r212(n) = c12'*x2buff;
    r222(n) = c22'*x2buff;
    r221(n) = c21'*x2buff;
    r122(n) = c22'*x1buff;
    r121(n) = c21'*x1buff;
```

Codice 6: Filtraggio del segnale di ingresso con le hrir

L'equazione (17) viene nuovamente applicata nel codice 7 per calcolare le uscite. I segnali intermedi r_{ilm} vanno a costituire un buffer e attraversano i filtri h. Le quattro uscite dei filtri h vengono infine sommate per ottenere `y1` e `y2`, come mostrato nell'equazione (18).

```
    r111buff = [r111(n); r111buff(1:end-1)];
    r112buff = [r112(n); r112buff(1:end-1)];
    r211buff = [r211(n); r211buff(1:end-1)];
    r212buff = [r212(n); r212buff(1:end-1)];
    r222buff = [r222(n); r222buff(1:end-1)];
    r221buff = [r221(n); r221buff(1:end-1)];
    r122buff = [r122(n); r122buff(1:end-1)];
    r121buff = [r121(n); r121buff(1:end-1)];

    y1(n) = ...
        h11'*r111buff+h21'*r112buff+h12'*r211buff+h22'*r212buff;
    y2(n) = ...
        h11'*r121buff+h21'*r122buff+h12'*r221buff+h22'*r222buff;
```

Codice 7: Calcolo delle uscite

Viene quindi calcolato l'errore come la differenza fra il segnale desiderato e le uscite.

```
    e1(n) = d1(n)-y1(n);
    e2(n) = d2(n)-y2(n);
```

Codice 8: Calcolo dell'errore

I valori dell'errore vengono poi utilizzati per aggiornare i tappi dei filtri `h11`, `h12`, `h21` e `h22` tramite un altro ciclo for, utilizzando la formula (23).

```
    for k = 1:M
        h11(k) = ...
            h11(k)+mu*(e1(n)*r111buff(k)+e2(n)*r121buff(k));
        h12(k) = ...
            h12(k)+mu*(e1(n)*r211buff(k)+e2(n)*r221buff(k));
        h21(k) = ...
            h21(k)+mu*(e1(n)*r112buff(k)+e2(n)*r122buff(k));
        h22(k) = ...
            h22(k)+mu*(e1(n)*r212buff(k)+e2(n)*r222buff(k));
    end
end
```

Codice 9: Aggiornamento del filtro

5.2 Fast Deconvolution

Nella Fast Deconvolution, come per il codice della LMS, si carica il file audio e le risposte impulsive `c11`, `c12`, `c21` e `c22` come mostrato nei codici 1 e 2. A differenza dell'algoritmo LMS, nella Fast Deconvolution si usa un overlap and save quindi occorre scegliere come dividere in blocchi il segnale di ingresso. Si implementa l'overlap and save come descritto nel capitolo 4.2, con dei blocchi di lunghezza $L = 4096$, pari alla lunghezza di default del frame di Nu-Tech, e un overlap al 50%, quindi $M = L/2 = 2048$. La lunghezza totale del frame con overlap sarà quindi $\text{fs} = L + M - 1 = 4096 + 2048 - 1 = 6143$, per una FFT efficiente si sceglie la potenza di 2 più vicina a 6143, quindi $\text{fftLen} = 2^{13} = 8192$.

```
L = 4096;
M = L/2;
fs = L + M - 1; % frame size
fftLen = 2.^nextpow2(fs);
```

Codice 10: Parametri overlap and save

Per eseguire l'overlap and save è necessaria la FFT di lunghezza pari a **fftLen** dei filtri C e H. Poiché i filtri H non sono ancora stati calcolati, vengono inizializzati a zero.

```
C11 = fft(c11, fftLen);     % HRTF left loudspeaker - left ear
C12 = fft(c12, fftLen);     % HRTF right loudspeaker - left ear
C21 = fft(c21, fftLen);     % HRTF left loudspeaker - right ear
C22 = fft(c22, fftLen);     % HRTF right loudspeaker - right ear

H11 = zeros(fftLen,1);     % filtro di cancellazione del ...
   crosstalk input 1 output 1
H12 = zeros(fftLen,1);     % filtro di cancellazione del ...
   crosstalk input 2 output 1
H21 = zeros(fftLen,1);     % filtro di cancellazione del ...
   crosstalk input 1 output 2
H22 = zeros(fftLen,1);     % filtro di cancellazione del ...
   crosstalk input 2 output 2
```

Codice 11: FFT delle risposte impulsive del canale e inizializzazione dei filtri di cancellazione del crosstalk

L'implementazione dell'equazione (30) è mostrata nel codice 12 con $\beta = 0.1$ e $B = I$, dove I è la matrice identità. Per ogni frequenza k dei filtri C_{11}, C_{12}, C_{21} e C_{22}, composti da **fftLen** punti in frequenza, viene costruita una matrice $C[k]$ nel seguente modo:

$$C[k] = \begin{bmatrix} C_{11}[k] & C_{12}[k] \\ C_{21}[k] & C_{22}[k] \end{bmatrix}.$$

Per trovare i filtri H_{11}, H_{12}, H_{21} e H_{22} si calcola la matrice H come mostrato nell'equazione (30) dove H è una matrice 2×2 costruita nel seguente modo:

$$H[k] = \begin{bmatrix} H_{11}[k] & H_{12}[k] \\ H_{21}[k] & H_{22}[k] \end{bmatrix}.$$

I filtri H_{11}, H_{12}, H_{21} e H_{22} sono presi dagli elementi della matrice H.

```
beta = 0.1;
B = [1 0; 0 1];
for k = 1:length(C11)
    C = [C11(k) C12(k); C21(k) C22(k)];
    H = (C'*C+beta*(B'*B))^(-1)*C';
    H11(k) = H(1, 1);
    H12(k) = H(1, 2);
    H21(k) = H(2, 1);
    H22(k) = H(2, 2);
```

```
end
```

Codice 12: Calcolo dei filtri di cancellazione del crosstalk

Dopo aver calcolato i filtri H, si procede con l'implementazione dell'overlap and save inizializzando due buffer `x1Buff` e `x2Buff` con `fs` zeri.

```
x1Buff = zeros(fs, 1);
x2Buff = zeros(fs, 1);
```

Codice 13: Inizializzazione dei buffer per l'overlap and save

Successivamente si divide il segnale di ingresso, lungo `nPoints`, in blocchi di lunghezza `L`, risultando in un numero di blocchi pari a `nPoints/L`. Poiché non sappiamo a priori se questo valore è intero, lo si approssima per difetto usando la funzione `floor` di Matlab. Per l'`i`-esimo blocco si copiano `L` campioni del segnale di ingresso nel buffer dall'indice M in poi, come mostrato in figura 3.

```
nPoints = length(x1);
for i = 1 : floor(nPoints/L)
    copy L values of x vector into the buffer from M onwards
    x1Buff(fs - L + 1 : fs) = x1((i - 1) * L + 1 : i * L);
    x2Buff(fs - L + 1 : fs) = x2((i - 1) * L + 1 : i * L);
```

Codice 14: Copia di L campioni del segnale di ingresso nel buffer

Dopo aver riempito il buffer, si fa la FFT a `fftLen` punti del buffer di ingresso.

```
    X1BUFF = fft(x1Buff, fftLen);
    X2BUFF = fft(x2Buff, fftLen);
```

Codice 15: FFT del buffer di ingresso

Si calcola il buffer di uscita in frequenza implementando l'equazione (31) come mostrato nel codice 16.

```
    Y1BUFF = ...
        (C11.*H11+C12.*H21).*X1BUFF+(C11.*H12+C12.*H22).*X2BUFF;
    Y2BUFF = ...
        (C21.*H11+C22.*H21).*X1BUFF+(C21.*H12+C22.*H22).*X2BUFF;
```

Codice 16: Calcolo del buffer di uscita in frequenza

Usando il buffer di uscita in frequenza, si calcola la IFFT e si prendono solamente i valori reali.

```
    y1Buff = real(ifft(Y1BUFF));
    y2Buff = real(ifft(Y2BUFF));
```

Codice 17: IFFT del buffer di uscita

L'aggiornamento dell'uscita avviene scartando i primi $M - 1$ campioni e dunque prendendo gli ultimi L campioni del buffer precedentemente calcolato.

```
    % discard first M - 1 values
    y1((i-1) * L + 1 : i * L) = y1Buff(fs - L + 1 : fs);
    y2((i-1) * L + 1 : i * L) = y2Buff(fs - L + 1 : fs);
```

Codice 18: Aggiornamento dell'uscita

Infine, si aggiornano i buffer di ingresso prendendo gli ultimi $M - 1$ punti del buffer di ingresso precedente e mettendoli all'inizio del buffer di ingresso successivo.

```
    % first M - 1 values of the new vector must be the last ...
        ones of the previous array
    x1Buff(1 : M - 1) = x1Buff(fs - M + 2 : fs);
    x2Buff(1 : M - 1) = x2Buff(fs - M + 2 : fs);
end
```

Codice 19: Aggiornamento del buffer di ingresso

6 Codice C

6.1 LMS

Nella libreria `Plugin.h` viene inizialmente definito il nome del Nuts come “LMS Filter” e l’indice `ID_MU` dell’RTWatch che permette di cambiare il parametro μ.

```
#define NUTS_NAME   "LMS Filter"
#define ID_MU 0
```

Codice 20: Dichiarazione delle costanti in `Plugin.h`

Nella classe `class PlugIn : public LEEffect` vengono definite le variabili che saranno utilizzate nell’algoritmo.

Le variabili sono le stesse usate nel codice Matlab in 4 e 5.

```
private:

    int FrameSize, SampleRate;
    int M, L, tau;
    double mu;
    double* x1, * x2, * d1, * d2, * y1, * y2, * e1, * e2,
        * x1buff, * x2buff;
    double* c11, * c12, * c21, * c22;
    double* h11, * h12, * h21, * h22;
    double* r111, * r112, * r211, * r212, * r222, * r221,
        * r122, * r121;
    double* r111buff, * r112buff, * r211buff, * r212buff,
        * r222buff, * r221buff, * r122buff, * r121buff;
```

Codice 21: Dichiarazione delle variabili in `Plugin.h`

Nella libreria `myLib.h` vengono dichiarate le funzioni usate per scrivere e leggere i file “.dat”.

```
void read_dat(char *name, double *data, int dim);
void write_dat(char *name, double *data, int dim,
    char*save_dir);
```

Codice 22: Libreria `myLib.h`

Nel codice sorgente `myLib.cpp` vengono definite le funzioni `read_dat` e `write_dat`. La prima apre in input e legge un file, mentre la seconda apre in output un file e ne scrive il dato `data`.

```
#include "StdAfx.h"
#include "myLib.h"

void write_dat(char *name, double *data, int dim,
    char *save_name)
{
    char name_a[MAX_FILE_NAME_LENGTH];
    memset(name_a,0,MAX_FILE_NAME_LENGTH*sizeof(char));
    strcpy(name_a,save_name);
    strcat(name_a,name);

    char *c;
    c=(char *)(void *)data;

    fstream File;
    File.open(name_a,ios::out | ios::binary);
    File.write(c,dim*sizeof(double));
    File.close();
}

void read_dat(char *name, double *data, int dim)
{
    char *c;
    c=(char *)(void *)data;

    ifstream File;
    File.open(name, ios::in | ios::binary);
    File.read(c, dim * sizeof(double));
    File.close();
}
```

Codice 23: File `myLib.cpp`

Nel file `PlugIn.cpp` sono presenti le funzioni:

- Costruttore della classe Plugin : viene chiamata quando il NUTS viene caricato sulla board del NUTECH
- Funzione di inizializzazione: viene chiamata quando si lancia la riproduzione con il tasto START
- Funzione di elaborazione: all'interno di questa funzione va messa la parte di elaborazione real time del segnale
- Funzione di deinizializzazione: viene chiamata quando si seleziona il tasto STOP
- Distruttore della classe Plugin : viene chiamata quando il NUTS viene eliminato dalla board del NUTECH

Nel costruttore vengono inizializzate a zero le variabili precedentemente definite. `Framesize` e `Samplerate` vengono ricavati tramite la `CBFunction` e vengono impostati due pin di input e due pin di output, rispettivamente con la funzione `LESetNumInput(2)` e con `LESetNumOutput(2)`.

```
PlugIn::PlugIn(InterfaceType _CBFunction,void * _PlugRef,
    HWND ParentDlg): LEEffect(_CBFunction,_PlugRef,ParentDlg)
{
    FrameSize = CBFunction(this,NUTS_GET_FS_SR,0,
        (LPVOID)AUDIOPROC);
    SampleRate = CBFunction(this,NUTS_GET_FS_SR,1,
        (LPVOID)AUDIOPROC);

    LESetNumInput(2);
    LESetNumOutput(2);

    M = 512;    // filter length
    tau = M;
    mu = 1e-4;

    y1 = 0;
    y2 = 0;

    x1 = 0;
    x2 = 0;

    x1buff = 0;
    x2buff = 0;

    e1 = 0;
    e2 = 0;

    d1 = 0;
    d2 = 0;

    c11 = 0;
    c12 = 0;
    c21 = 0;
    c22 = 0;

    h11 = 0;
    h12 = 0;
    h21 = 0;
    h22 = 0;

    r111 = 0;
    r112 = 0;
```

```
    r211 = 0;
    r212 = 0;
    r222 = 0;
    r221 = 0;
    r122 = 0;
    r121 = 0;

    r111buff = 0;
    r112buff = 0;
    r121buff = 0;
    r122buff = 0;
    r211buff = 0;
    r212buff = 0;
    r221buff = 0;
    r222buff = 0;

    bufferNumber = 0;
}
```

Codice 24: Costruttore

Nella funzione di inizializzazione vengono allocate le variabili.

```
void __stdcall PlugIn::LEPlugin_Init()
{
    if (y1 == 0)
    {
        y1 = ippsMalloc_64f(FrameSize);
        ippsZero_64f(y1, FrameSize);
    }
```

Codice 25: Esempio allocazione delle variabili in `LEPlugin_Init`

Inoltre vegono caricati i tappi dei filtri HRIR tramite la funzione `read_dat` presente in `myLib.cpp`.

```
    // load filter taps
    read_dat("c11.dat", c11, M);

    read_dat("c12.dat", c12, M);

    read_dat("c21.dat", c21, M);

    read_dat("c22.dat", c22, M);
```

Codice 26: Lettura delle HRIR

Nella funzione Process avviene l'algoritmo vero e proprio. Questa funzione viene chiamata per ogni frame dell'ingresso e produce il corrispondente frame in uscita. Gli ingressi e le uscite vengono quindi caricati in variabili temporanee in modo da poter essere elaborate. Gli ingressi `InputData1` e `InputData2` vengono copiati nelle variabili `x1` e `x2` tramite la funzione `ippsCopy_64f`. Usando la stessa funzione si crea il vettore del segnale desiderato ritardando l'ingresso di `tau` campioni copiando `tau` valori dal puntatore `x1 + FrameSize - tau` in `d1` e allo stesso modo dal puntatore `x2 + FrameSize - tau` in `d2`. Si copiano successivamente i primi `FrameSize - tau` valori di `x1` e `x2` dall'indice `tau` in poi di `d1` e `d2`. I vettori di uscita `y1` e `y2` vengono invece posti pari a zero.

```
int __stdcall PlugIn::LEPlugin_Process(PinType **Input,
    PinType **Output,LPVOID ExtraInfo)
{
    double* InputData1 = ((double*)Input[0]->DataBuffer);
    double* OutputData1 = ((double*)Output[0]->DataBuffer);
    double* InputData2 = ((double*)Input[1]->DataBuffer);
    double* OutputData2 = ((double*)Output[1]->DataBuffer);

    // d1[0:tau] = x1[end - tau:end];
    ippsCopy_64f(x1 + FrameSize - tau, d1, tau);
    // copy InputData1 to x1
    ippsCopy_64f(InputData1, x1, FrameSize);
    // divide each element of the vector x1 by 32768 and
    //   store the result in x1
    ippsDivC_64f_I(32768.0, x1, FrameSize);
    //d1[tau + 1:end] = x1[0:FrameSize - tau];
    ippsCopy_64f(x1, d1 + tau, FrameSize - tau);
    // d2[0:tau] = x2[end - tau:end];
    ippsCopy_64f(x2 + FrameSize - tau, d2, tau);
    // copy InputData2 to x2
    ippsCopy_64f(InputData2, x2, FrameSize);
    // divide each element of the vector x2 by 32768 and
    //  store the result in x2
    ippsDivC_64f_I(32768.0, x2, FrameSize);
    //d2[tau + 1:end] = x2[0:FrameSize - tau];
    ippsCopy_64f(x2, d2 + tau, FrameSize - tau);

    // initialize y1 and y2 with zeros
    ippsZero_64f(y1, FrameSize);
    ippsZero_64f(y2, FrameSize);
```

Codice 27: Process

Il ciclo for corrisponde a quello usato in Matlab: vengono creati dei vettori di buffer temporanei e il segnale di ingresso viene elaborato un campione alla volta. La creazione dei buffer di ingresso `x1buff` e `x2buff` avviene shiftando con `ippsMove_64f` i primi `M - 1` valori di un posto verso destra e mettendo al primo posto l'`i`-esimo valore di `x1` e `x2`. I segnali r_{ijl} intermedi si ottengono facendo il prodotto scalare con la funzione `ippsDotProd_64f` tra i buffer di ingresso e i filtri c_{ij}.

```
    for (int i = 0; i < FrameSize; i++)
    {
        ippsMove_64f(x1buff, x1buff + 1, M - 1);
        x1buff[0] = x1[i];

        ippsMove_64f(x2buff, x2buff + 1, M - 1);
        x2buff[0] = x2[i];

        ippsDotProd_64f(c11, x1buff, M, &(r111[i]));
        ippsDotProd_64f(c12, x1buff, M, &(r112[i]));
        ippsDotProd_64f(c11, x2buff, M, &(r211[i]));
        ippsDotProd_64f(c12, x2buff, M, &(r212[i]));
        ippsDotProd_64f(c22, x2buff, M, &(r222[i]));
        ippsDotProd_64f(c21, x2buff, M, &(r221[i]));
        ippsDotProd_64f(c22, x1buff, M, &(r122[i]));
        ippsDotProd_64f(c21, x1buff, M, &(r121[i]));
```

Codice 28: Calcolo dei segnali intermedi

A questo punto sono i segnali r_{ijl} ad essere inseriti nel buffer. Facendo il prodotto scalare di ognuno di essi per i filtri di ricostruzione h_{ij} si ottengono le quattro componenti dell'uscita uscita. Ogni componente è salvata nella variabile di appoggio `ytmp` e sommata dopo ogni prodotto scalare alle variabili di uscita. In questo modo si ottengono i segnali di uscita `y1` e `y2`.

```
        ippsMove_64f(r111buff, r111buff + 1, M - 1);
        r111buff[0] = r111[i];

        ippsMove_64f(r112buff, r112buff + 1, M - 1);
        r112buff[0] = r112[i];

        ippsMove_64f(r121buff, r121buff + 1, M - 1);
        r121buff[0] = r121[i];

        ippsMove_64f(r122buff, r122buff + 1, M - 1);
        r122buff[0] = r122[i];

        ippsMove_64f(r211buff, r211buff + 1, M - 1);
        r211buff[0] = r211[i];
```

```
        ippsMove_64f(r212buff, r212buff + 1, M - 1);
        r212buff[0] = r212[i];

        ippsMove_64f(r221buff, r221buff + 1, M - 1);
        r221buff[0] = r221[i];

        ippsMove_64f(r222buff, r222buff + 1, M - 1);
        r222buff[0] = r222[i];

        double ytmp = 0.0;
        ippsDotProd_64f(h11, r111buff, M, y1 + i);
        // y1[i] = y1[i] + ytmp;

        ippsDotProd_64f(h21, r112buff, M, &ytmp);
        y1[i] = y1[i] + ytmp;
        //ippsAddC_64f_I(ytmp, y1 + i, 1);

        ippsDotProd_64f(h12, r211buff, M, &ytmp);
        y1[i] = y1[i] + ytmp;
        //ippsAddC_64f_I(ytmp, y1 + i, 1);

        ippsDotProd_64f(h22, r212buff, M, &ytmp);
        y1[i] = y1[i] + ytmp;
        //ippsAddC_64f_I(ytmp, y1 + i, 1);

        ippsDotProd_64f(h11, r121buff, M, y2 + i);
        //y2[i] = y2[i] + ytmp;

        ippsDotProd_64f(h21, r122buff, M, &ytmp);
        y2[i] = y2[i] + ytmp;
        //ippsAddC_64f_I(ytmp, y2 + i, 1);

        ippsDotProd_64f(h12, r221buff, M, &ytmp);
        y2[i] = y2[i] + ytmp;
        //ippsAddC_64f_I(ytmp, y2 + i, 1);

        ippsDotProd_64f(h22, r222buff, M, &ytmp);
        y2[i] = y2[i] + ytmp;
        //ippsAddC_64f_I(ytmp, y2 + i, 1);
```

Codice 29: Calcolo delle uscite

L'errore viene calcolato per ogni campione tramite la differenza fra il segnale desiderato e quello ottenuto.

```
        e1[i] = d1[i] - y1[i];
        e2[i] = d2[i] - y2[i];
```

I filtri h vengono poi aggiornati analogamente a 9 tramite il metodo steepest descend.

```
        for (int j = 0; j < M; j++)
        {
            h11[j] = h11[j] + mu * (e1[i] * r111buff[j] +
                e2[i] * r121buff[j]);
            h12[j] = h12[j] + mu * (e1[i] * r211buff[j] +
                e2[i] * r221buff[j]);
            h21[j] = h21[j] + mu * (e1[i] * r112buff[j] +
                e2[i] * r122buff[j]);
            h22[j] = h22[j] + mu * (e1[i] * r212buff[j] +
                e2[i] * r222buff[j]);
        }
    }
```

Codice 30: Aggiornamento dei filtri h

I vettori in uscita vengono quindi di nuovo riportati in `OutputData1` e `OutputData2` dopo essere stati moltiplicati per 32768.

```
    // multiply each element of the vector y1 by 32768
    //  and store the result in y1
    ippsMulC_64f_I(32768.0, y1, FrameSize);
    // copy y1 to OutputData1
    ippsCopy_64f(y1, OutputData1, FrameSize);
    // multiply each element of the vector y2 by 32768
    //  and store the result in y2
    ippsMulC_64f_I(32768.0, y2, FrameSize);
    // copy y2 to OutputData2
    ippsCopy_64f(y2, OutputData2, FrameSize);
    return COMPLETED;
}
```

Codice 31: Aggiornamento delle uscite e scaling

Nella `LEPlugin_Delete`, la funzione `ippsFree` rilascia la memoria precedentemente allocata in maniera dinamica nella Init. Un esempio di come usare questa funzione è mostrato di seguito.

```
void __stdcall PlugIn::LEPlugin_Delete()
{
    if (y1 != 0)
    {
        ippsFree(y1);
        y1 = 0;
```

```
    }
```

Codice 32: Delete

L'RT Watch per variare il parametro μ è creato nella funzione `LERTWatchInit` come di seguito. Come si può vedere, la variabile che accetta in ingresso può essere solo di tipo `double` e di dimensione `sizeof(double)` byte.

```
void __stdcall PlugIn::LERTWatchInit()
{
    WatchType NewWatch;

    memset(&NewWatch, 0, sizeof(WatchType));
    NewWatch.EnableWrite = true;
    NewWatch.LenByte = sizeof(double);
    NewWatch.TypeVar = WTC_DOUBLE;
    NewWatch.IDVar = ID_MU;
    sprintf(NewWatch.VarName, "mu\0");
    CBFunction(this, NUTS_ADDRTWATCH, 0, &NewWatch);
}
```

Codice 33: Creazione RT Watch

Per cambiare correttamente il parametro dalla finestra del NU-TECH è necessario usare le funzioni `LEGetParameter` e `LESetParameter`, rispettivamente illustrate nei codici 34 e 35. Nella prima funzione, `memcpy` copia `sizeof(double)` byte dalla locazione puntata da `&mu` a quella puntata da `Data`. Nella seconda funzione avviene l'opposto, cioè si copia il valore immesso dall'utente nella variabile `mu`. Questo accade solo se la variabile booleana `isRunning` è falsa, cioè se il programma non è in esecuzione. Il controllo descritto è necessario per non cambiare il valore μ durante l'esecuzione dell'algoritmo adattativo.

```
int  __stdcall PlugIn::LEGetParameter(int Index,void *Data)
{

    CBFunction(this, NUTS_GETSECURETIME, NUTSSECURE, 0);

    if (Index == ID_MU)
    {
        memcpy((double*)Data, &mu, sizeof(double));
    }

    CBFunction(this, NUTS_RELEASESECURETIME, NUTSSECURE, 0);
    return 0;
}
```

Codice 34: Funzione `LEGetParameter`

```
void __stdcall PlugIn::LESetParameter(int Index,void *Data,
    LPVOID bBroadCastInfo)
{
    CBFunction(this, NUTS_GETSECURETIME, NUTSSECURE, 0);

    if (Index == ID_MU)
    {
        if (!isRunning)
        {
            memcpy(&mu, (double*)Data, sizeof(double));
            CBFunction(this, NUTS_UPDATERTWATCH, ID_MU, 0);
        }
    }

    CBFunction(this, NUTS_RELEASESECURETIME, NUTSSECURE, 0);

}
```

Codice 35: Funzione `LESetParameter`

6.2 Fast Deconvolution

Nella libreria `Plugin.h` viene inizialmente definito il nome del Nuts come "Fast Deconvolution" e l'indice `ID_BETA` dell'RT Watch che permette di cambiare il parametro β.

```
#define NUTS_NAME   "Fast Deconvolution"
#define ID_BETA 0
```

Codice 36: Dichiarazione delle costanti in `Plugin.h`

Successivamente vengono dichiarate le variabili intere `FrameSize` e `SampleRate` che contengono rispettivamente la lunghezza del frame e la frequenza di campionamento, `L`, `M`, `fs` e `fftLen` sono le stesse variabili viste nel codice 10, mentre `fftOrd` contiene l'ordine della FFT, cioè l'esponente di 2 usato per trovare i punti su cui viene calcolata la FFT. Il parametro `beta` rappresenta il parametro β dell'equazione (29), `hrir` contiene le risposte impulsive di c_{11}, c_{12}, c_{21} e c_{22}. Vengono poi dichiarati come `Ipp64fc` tutti i puntatori necessari per i buffer di ingresso e di uscita dell'overlap and save, i puntatori delle risposte in frequenza C_{11}, C_{12}, C_{21}, C_{22}, H_{11}, H_{12}, H_{21} e H_{22} e altre variabili temporanee di appoggio usate durante i calcoli. Lo struct `Ipp64fc` è formato da un campo reale e uno immaginario come mostrato di seguito.

```
typedef struct {
    Ipp64f  re;
    Ipp64f  im;
} Ipp64fc;
```

Codice 37: Struct `Ipp64fc`

Le variabili `pSpec`, `pMemSpec`, `pMemInit`, `pMemBuffer`, `sizeSpec`, `sizeInit` e `sizeBuffer` sono usate per inizializzare e calcolare la FFT. La variabile booleana `isRunning` servirà successivamente per conoscere se il programma è in esecuzione e in particolare, in questa

```
int FrameSize,SampleRate;
int L, M, fs, fftLen, fftOrd;
double beta, * hrir;
Ipp64fc* x1buff, * x2buff, * X1BUFF, * X2BUFF, * y1buff,
    * y2buff, * Y1BUFF, * Y2BUFF;
Ipp64fc* c11, * c12, * c21, * c22, * C11, * C12, * C21, * C22;
Ipp64fc* H11, * H12, * H21, * H22;
Ipp64fc det, invtemp, invtemp2, Cconj[2][2], C[2][2],
    B[2][2], Ctemp[2][2], Ctemp2[2][2];
Ipp64fc* temp, * temp2, * temp3;
IppsFFTSpec_C_64fc* pSpec;

Ipp8u* pMemSpec, * pMemInit, * pMemBuffer;

int sizeSpec, sizeInit, sizeBuffer;

int fftLen, fftOrd;

bool isRunning;
```

Codice 38: Dichiarazione delle variabili in `Plugin.h`

Nel costruttore vengono inizializzate le variabili dichiarate nella libreria `Plugin.h` come mostrato nel codice 39. Si ottengono il `FrameSize` e il `SampleRate` usando la funzione `CBFunction` che permette di interagire con il Nu-Tech. Vengono impostati due pin di input e due pin di output, rispettivamente con la funzione `LESetNumInput(2)` e con `LESetNumOutput(2)`. La variabile `L` è pari alla lunghezza del frame, cioè di default 4096, l'overlap `M` è pari alla metà di `L`, quindi la lunghezza di tutto il buffer è `fs = L + M - 1`. La variabile booleana `isRunning` è inizializzata falsa.

```
PlugIn::PlugIn(InterfaceType _CBFunction, void * _PlugRef,
    HWND ParentDlg): LEEffect(_CBFunction,_PlugRef,ParentDlg)
{
    FrameSize = CBFunction(this,NUTS_GET_FS_SR,0,
        (LPVOID)AUDIOPROC);
    SampleRate = CBFunction(this,NUTS_GET_FS_SR,1,
        (LPVOID)AUDIOPROC);

    LESetNumInput(2);
    LESetNumOutput(2);

    isRunning = false;

    L = FrameSize;
    M = int(L / 2);
    fs = L + M - 1;

    beta = 0.1;

    x1buff = 0;
    x2buff = 0;

    X1BUFF = 0;
    X2BUFF = 0;

    y1buff = 0;
    y2buff = 0;

    Y1BUFF = 0;
    Y2BUFF = 0;

    temp = 0;
    temp2 = 0;
    temp3 = 0;

    hrir = 0;
    c11 = 0;
    c12 = 0;
    c21 = 0;
    c22 = 0;

    C11 = 0;
    C12 = 0;
    C21 = 0;
    C22 = 0;

    H11 = 0;
    H12 = 0;
```

```
    H21 = 0;
    H22 = 0;

    fftLen = 8192;
    fftOrd = (int)(log10((double)fftLen) / log10(2.0));
}
```

Codice 39: Costruttore

Nella funzione `LEPlugin_Init` la variabile booleana `isRunning` diventa `true`, questo non permette di modificare il parametro β nell'RT Watch.

```
void __stdcall PlugIn::LEPlugin_Init()
{
    isRunning = true;
```

Codice 40: Variabile `isRunning` nella funzione `LEPlugin_Init`

Successivamente, vengono allocate tutte le variabili dichiarate come puntatori, un esempio di allocazione è mostrata nel codice 41. A differenza del codice 13 in Matlab, in questo caso le variabili `x1buff` e `x2buff` che contengono il buffer nel tempo sono della stessa dimensione di `X1BUFF` e `X2BUFF` che contengono le rispettive trasformate di Fourier, cioè `fftLen`. In particolare i valori diversi da zero sono sempre `fs`, come in Matlab, ma i restanti `fftLen-fs` campioni sono uguali a zeri. Questo è necessario perché per sfruttare la funzione che calcola la FFT dobbiamo avere un segnale nel tempo della stessa lunghezza di quello in frequenza. Allo stesso modo anche i buffer di uscita nel tempo `y1buff` e `y2buff` e in frequenza `Y1BUFF` e `Y2BUFF` sono lunghi `fftLen`. Le variabili di appoggio per i calcoli `temp`, `temp2`, `temp3`, le variabili che contengono le HRIR, le HRTF e i filtri di ricostruzioni sono tutte di dimensione `fftLen`.

```
if (x1buff == 0)
{
        x1buff = ippsMalloc_64fc(fftLen);
        ippsZero_64fc(x1buff, fftLen);
}
```

Codice 41: Allocazione della variabile `x1buff` nella funzione `LEPlugin_Init`

La HRIR vengono lette dai file ".dat" e salvate nell'array `hrir` di dimensione `M`. Successivamente, questo array viene copiato nel campo reale delle variabili `c11`, `c12`, `c21` e `c22` di tipo `Ipp64fc`.

```
if (hrir == 0)
{
    hrir = ippsMalloc_64f(M);
    ippsZero_64f(hrir, M);
}
// load filter taps
read_dat("c11.dat", hrir, M);
for (int i = 0; i < M; i++)
    c11[i].re = hrir[i];

read_dat("c12.dat", hrir, M);
for (int i = 0; i < M; i++)
    c12[i].re = hrir[i];

read_dat("c21.dat", hrir, M);
for (int i = 0; i < M; i++)
    c21[i].re = hrir[i];

read_dat("c22.dat", hrir, M);
for (int i = 0; i < M; i++)
    c22[i].re = hrir[i];
```

Codice 42: Lettura delle HRIR

Nel codice 43 si può vedere come è inizializzata la FFT. La funzione `ippsFFTGetSize_C_64fc` calcola le dimensioni della struttura delle specifiche della FFT e i buffer di lavoro richiesti. I parametri della funzione sono:

- l'ordine `fftOrd` della FFT: sia N la lunghezza del segnale di ingresso, allora $N = 2^{\texttt{fftOrd}}$,
- il metodo di normalizzazione `IPP_FFT_DIV_INV_BY_N` che specifica di calcolare la IFFT con una normalizzazione $1/N$,
- il parametro deprecato `ippAlgHintNone`,
- il puntatore `sizeSpec` al valore della dimensione della struttura che contiene le specifiche della FFT,
- il puntatore `sizeInit` al valore della dimensione del buffer per la funzione di inizializzazione della FFT,
- il puntatore `sizeBuffer` al valore della dimensione del buffer di lavoro della FFT.

La funzione `ippsFFTGetSize_C_64fc` calcola:

- la dimensione della struttura delle specifiche della FFT e la salva in `sizeSpec`,

- la dimensione del buffer di lavoro `sizeInit` necessario per la funzione `ippsFFTInit_C` che inizializza la struttura dalla FFT,
- la dimensione del buffer di lavoro `sizeBuffer` della FFT per le funzioni di `ippsFFTFwd` e `ippsFFTInv` che calcolano rispettivamente la FFT diretta e inversa.

Successivamente, la funzione `ippsFFTInit_C_64fc` inizializza la struttura delle specifiche della FFT. I parametri di questa funzione sono:

- l'ordine `fftOrd` della FFT: sia N la lunghezza del segnale di ingresso, allora $N = 2^{\texttt{fftOrd}}$,
- il metodo di normalizzazione `IPP_FFT_DIV_INV_BY_N` che specifica di calcolare la IFFT con una normalizzazione $1/N$,
- il parametro deprecato `ippAlgHintNone`,
- il puntatore al puntatore alla struttura delle specifiche della FFT `pSpec` che deve essere creata
- il puntatore `pMemSpec` all'area per la struttura delle specifiche della FFT,
- il puntatore al buffer di lavoro `pMemInit`.

La funzione `ippsFFTInit_C_64fc` inizializza la struttura delle specifiche della FFT `pSpec` con i seguenti parametri:

- l'ordine `fftOrd` della FFT,
- il flag `IPP_FFT_DIV_INV_BY_N`,
- il parametro `ippAlgHintNone`.

```
// get sizes for required buffers
ippsFFTGetSize_C_64fc(fftOrd, IPP_FFT_DIV_INV_BY_N,
    ippAlgHintNone, &sizeSpec, &sizeInit, &sizeBuffer);
/// allocate memory for required buffers
pMemSpec = (Ipp8u*)ippMalloc(sizeSpec);
pMemInit = (Ipp8u*)ippMalloc(sizeInit);
pMemBuffer = (Ipp8u*)ippMalloc(sizeBuffer);
ippsFFTInit_C_64fc(&pSpec, fftOrd, IPP_FFT_DIV_INV_BY_N,
    ippAlgHintNone, pMemSpec, pMemInit);
```

Codice 43: Inizializzazione FFT

Dopo aver inizializzato la struttura delle specifiche della FFT, si procede ad eseguire la trasformata di Fourier delle HRIR, usando le variabili `pSpec` e `pMemBuffer` precedentemente definite, come di seguito

```
ippsFFTFwd_CToC_64fc(c11, C11, pSpec, pMemBuffer);
ippsFFTFwd_CToC_64fc(c12, C12, pSpec, pMemBuffer);
ippsFFTFwd_CToC_64fc(c21, C21, pSpec, pMemBuffer);
ippsFFTFwd_CToC_64fc(c22, C22, pSpec, pMemBuffer);
```

Codice 44: FFT delle HRIR

Conoscendo le risposte in frequenza `C11`, `C12`, `C21` e `C22`, è possibile implementare la formula (30) per trovare i filtri `H11`, `H12`, `H21` e `H22`. Poiché non è possibile eseguire il calcolo dell'equazione in una sola riga di codice come nel programma 12 di Matlab, è necessario usare le variabili di appoggio per calcolare le somme e i prodotti in più passaggi. Dobbiamo dunque calcolare le matrici C e C^H, successivamente calcoliamo

$$C_{\text{temp}}[k] = C^H[k]C[k] + \beta \cdot I \tag{36}$$

e ne facciamo l'inversa. Infine, moltiplichiamo l'inversa per $C^H[k]$.

Per iniziare si costruisce la matrice `C` e la sua complessa coniugata chiamata `Cconj`, come nel codice 45. Sia

$$C[k] = \begin{bmatrix} C_{11}[k] & C_{12}[k] \\ C_{21}[k] & C_{22}[k] \end{bmatrix}$$

allora

$$C_{\text{conj}}[k] = \begin{bmatrix} C^*_{11}[k] & C^*_{12}[k] \\ C^*_{21}[k] & C^*_{22}[k] \end{bmatrix} \tag{37}$$

```
for (int n = 0; n < fftLen; n++)
{
    C[0][0] = C11[n];
    C[0][1] = C12[n];
    C[1][0] = C21[n];
    C[1][1] = C22[n];

    ippsConj_64fc(&(C[0][0]), &(Cconj[0][0]), 1);
    ippsConj_64fc(&(C[0][1]), &(Cconj[0][1]), 1);
    ippsConj_64fc(&(C[1][0]), &(Cconj[1][0]), 1);
    ippsConj_64fc(&(C[1][1]), &(Cconj[1][1]), 1);
```

Codice 45: Implementazione della matrice `C` e `Cconj`

Esplicitando l'equazione (36) si può scrivere

$$
\begin{gathered}
C_{\text{temp}}[k] = C^H[k]C[k] + \beta \cdot I = \\
\begin{bmatrix} C_{11}[k] & C_{12}[k] \\ C_{21}[k] & C_{22}[k] \end{bmatrix}^H \cdot \begin{bmatrix} C_{11}[k] & C_{12}[k] \\ C_{21}[k] & C_{22}[k] \end{bmatrix} + \beta \cdot \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}
\end{gathered} \tag{38}
$$

Facendo la trasposta e coniugata di C si ha

$$
C_{\text{temp}}[k] = \begin{bmatrix} C^*_{11}[k] & C^*_{21}[k] \\ C^*_{12}[k] & C^*_{22}[k] \end{bmatrix} \cdot \begin{bmatrix} C_{11}[k] & C_{12}[k] \\ C_{21}[k] & C_{22}[k] \end{bmatrix} + \begin{bmatrix} \beta & 0 \\ 0 & \beta \end{bmatrix} \tag{39}
$$

Usando la definizione di C_{conj} (37) e applicandola alla (39) si ottiene

$$
C_{\text{temp}}[k] = \begin{bmatrix} C_{conj11}[k] & C_{conj21}[k] \\ C_{conj12}[k] & C_{conj22}[k] \end{bmatrix} \cdot \begin{bmatrix} C_{11}[k] & C_{12}[k] \\ C_{21}[k] & C_{22}[k] \end{bmatrix} + \begin{bmatrix} \beta & 0 \\ 0 & \beta \end{bmatrix} \tag{40}
$$

Scrivendo l'equazione (40) come sistema si possono ottenere gli elementi della matrice $C_{\text{temp}}[k]$ come di seguito

$$
\begin{cases}
C_{\text{temp11}}[k] = C_{conj11}[k] \cdot C_{11}[k] + C_{conj21}[k] \cdot C_{21}[k] + \beta \\
C_{\text{temp12}}[k] = C_{conj11}[k] \cdot C_{12}[k] + C_{conj21}[k] \cdot C_{22}[k] \\
C_{\text{temp21}}[k] = C_{conj12}[k] \cdot C_{11}[k] + C_{conj22}[k] \cdot C_{21}[k] \\
C_{\text{temp22}}[k] = C_{conj12}[k] \cdot C_{12}[k] + C_{conj22}[k] \cdot C_{22}[k] + \beta
\end{cases} \tag{41}
$$

L'elemento $C_{\text{temp11}}[k]$ viene calcolato come mostrato nel seguente codice, moltiplicando i due prodotti $C_{conj11}[k] \cdot C_{11}[k]$ e $C_{conj21}[k] \cdot C_{21}[k]$ con la funzione `ippsMul_64fc`, successivamente vengono sommati tra loro con la funzione `ippsAdd_64fc_I` e infine sommata la costante β con `ippsAddC_64fc_I`.

```
//Ctemp[0][0] = Cconj[0][0] * C[0][0] + Cconj[1][0] * C[1][0]
// + beta;
ippsMul_64fc(&(Cconj[0][0]), &(C[0][0]), &(Ctemp[0][0]), 1);
ippsMul_64fc(&(Cconj[1][0]), &(C[1][0]), &(Ctemp2[0][0]), 1);
ippsAdd_64fc_I(&(Ctemp2[0][0]), &(Ctemp[0][0]), 1);
ippsAddC_64fc_I({ beta, 0 }, &(Ctemp[0][0]), 1);
```

Codice 46: Implementazione della prima riga del sistema (41)

In modo analogo vengono calcolati gli elementi $C_{\text{temp12}}[k]$, $C_{\text{temp21}}[k]$ e $C_{\text{temp22}}[k]$.

```
//Ctemp[0][1] = Cconj[0][0] * C[0][1] + Cconj[1][0] * C[1][1];
ippsMul_64fc(&(Cconj[0][0]), &(C[0][1]), &(Ctemp[0][1]), 1);
```

```
ippsMul_64fc(&(Cconj[1][0]), &(C[1][1]), &(Ctemp2[0][1]), 1);
ippsAdd_64fc_I(&(Ctemp2[0][1]), &(Ctemp[0][1]), 1);

//Ctemp[1][0] = Cconj[0][1] * C[0][0] + Cconj[1][1] * C[1][0];
ippsMul_64fc(&(Cconj[0][1]), &(C[0][0]), &(Ctemp[1][0]), 1);
ippsMul_64fc(&(Cconj[1][1]), &(C[1][0]), &(Ctemp2[1][0]), 1);
ippsAdd_64fc_I(&(Ctemp2[1][0]), &(Ctemp[1][0]), 1);

//Ctemp[1][1] = Cconj[0][1] * C[0][1] + Cconj[1][1] * C[1][1]
// + beta;
ippsMul_64fc(&(Cconj[0][1]), &(C[0][1]), &(Ctemp[1][1]), 1);
ippsMul_64fc(&(Cconj[1][1]), &(C[1][1]), &(Ctemp2[1][1]), 1);
ippsAdd_64fc_I(&(Ctemp2[1][1]), &(Ctemp[1][1]), 1);
ippsAddC_64fc_I({ beta, 0 }, &(Ctemp[1][1]), 1);
```

Codice 47: Implementazione della matrice `Ctemp`

Dopo aver calcolato la matrice `Ctemp` con la formula (36), occorre trovare la sua inversa. Per fare ciò, si inizia calcolando il determinante di

$$C_{\text{temp}}[k] = \begin{bmatrix} C_{\text{temp11}}[k] & C_{\text{temp12}}[k] \\ C_{\text{temp21}}[k] & C_{\text{temp22}}[k] \end{bmatrix}.$$

La formula per trovare il determinante di C_{temp} è la seguente

$$\det = (C_{\text{temp11}}[k] \cdot C_{\text{temp22}}[k]) - (C_{\text{temp12}}[k] \cdot C_{\text{temp21}}[k]) \tag{42}$$

L'equazione (42) è stata implementata nel codice 48 calcolando i prodotti $C_{\text{temp11}}[k] \cdot C_{\text{temp22}}[k]$ e $C_{\text{temp12}}[k] \cdot C_{\text{temp21}}[k]$ con la funzione `ippsMul_64fc`, infine sottraendo i due prodotti con `ippsSub_64fc_I` e salvando il determinante nella variabile `det`.

```
ippsMul_64fc(&(Ctemp[0][0]), &(Ctemp[1][1]), &det, 1);
ippsMul_64fc(&(Ctemp[0][1]), &(Ctemp[1][0]), &invtemp, 1);
ippsSub_64fc_I(&invtemp, &det, 1);
```

Codice 48: Calcolo del determinante della matrice `Ctemp`

Una volta noto il determinante possiamo implementare la formula dell'inversa di $C_{\text{temp}}[k]$ come nella seguente equazione

$$\begin{gathered} C_{\text{temp}}^{-1}[k] = \begin{bmatrix} C_{\text{temp11}}[k] & C_{\text{temp12}}[k] \\ C_{\text{temp21}}[k] & C_{\text{temp22}}[k] \end{bmatrix}^{-1} = \\ \frac{1}{\det(C_{\text{temp}})} \begin{bmatrix} C_{\text{temp22}}[k] & -C_{\text{temp12}}[k] \\ -C_{\text{temp21}}[k] & C_{\text{temp11}}[k] \end{bmatrix}. \end{gathered} \tag{43}$$

In C la formula (43) viene applicata dividendo ogni elemento di `Ctemp` per il determinante `det` con la funzione `ippsDivC_64fc`, invertendo gli elementi `Ctemp[0][0]` e `Ctemp[1][1]`, e cambiando segno agli elementi `Ctemp[0][1]` e `Ctemp[1][0]` moltiplicandoli per -1 con la funzione `ippsMulC_64fc_I`. L'inversa è calcolata usando il seguente codice

```
invtemp = Ctemp[0][0];
//Ctemp[0][0] = Ctemp[1][1] / det;
ippsDivC_64fc(&(Ctemp[1][1]), det, &(Ctemp[0][0]), 1);
//Ctemp[1][1] = invtemp / det;
ippsDivC_64fc(&invtemp, det, &(Ctemp[1][1]), 1);
//Ctemp[0][1] = -Ctemp[0][1] / det;
ippsMulC_64fc_I({ -1.0, 0.0 }, &det, 1);
ippsDivC_64fc_I(det, &(Ctemp[0][1]), 1);
//Ctemp[1][0] = -Ctemp[1][0] / det;
ippsDivC_64fc_I(det, &(Ctemp[1][0]), 1);
```

Codice 49: Calcolo dell'inversa della matrice `Ctemp`

L'ultimo passo consiste nel moltiplicare l'inversa di $C_{\text{temp}}[k]$ per $C^H[k]$ al fine implementare l'ultima parte della formula (30), cioè

$$H[k] = \left[C^H[k]C[k] + \beta I)\right]^{-1} C^H[k] = C^{-1}_{\text{temp}}[k] \cdot C^H[k]. \tag{44}$$

Sia $C_{\text{inv}}[k] = C^{-1}_{\text{temp}}[k]$, usando (37) ed esplicitando l'equazione (45) si ha

$$\begin{gathered} H[k] = C^{-1}_{\text{temp}}[k] \cdot C^H[k] = C_{\text{inv}}[k] \cdot C^H[k] = \\ \begin{bmatrix} C_{\text{inv}11}[k] & C_{\text{inv}12}[k] \\ C_{\text{inv}21}[k] & C_{\text{inv}22}[k] \end{bmatrix} \cdot \begin{bmatrix} C_{\text{conj}11}[k] & C_{\text{conj}21}[k] \\ C_{\text{conj}12}[k] & C_{\text{conj}22}[k] \end{bmatrix} = \begin{bmatrix} H_{11}[k] & H_{12}[k] \\ H_{21}[k] & H_{22}[k] \end{bmatrix}. \end{gathered} \tag{45}$$

Se si scrive l'equazione (45) come sistema si ottiene

$$\begin{cases} H_{11}[k] = (C_{\text{inv}11}[k] \cdot C_{\text{conj}11}[k]) + (C_{\text{inv}12}[k] \cdot C_{\text{conj}12}[k]) \\ H_{12}[k] = (C_{\text{inv}11}[k] \cdot C_{\text{conj}21}[k]) + (C_{\text{inv}12}[k] \cdot C_{\text{conj}22}[k]) \\ H_{21}[k] = (C_{\text{inv}21}[k] \cdot C_{\text{conj}11}[k]) + (C_{\text{inv}22}[k] \cdot C_{\text{conj}12}[k]) \\ H_{22}[k] = (C_{\text{inv}21}[k] \cdot C_{\text{conj}21}[k]) + (C_{\text{inv}22}[k] \cdot C_{\text{conj}22}[k]) \end{cases} \tag{46}$$

L'implementazione della prima riga del sistema (46), e quindi del filtro H_{11}, è mostrata nel codice 50 in cui si calcolano i prodotti $(C_{\text{inv}11}[k] \cdot C_{\text{conj}11}[k])$ e $(C_{\text{inv}12}[k] \cdot C_{\text{conj}12}[k])$, si sommano tra loro i prodotti e si salva il risultato nel campo reale e immaginario della variabile `H11`. Si noti che nel codice la variabile `Ctemp` contiene la matrice C^{-1}_{temp}.

```
//invtemp = Ctemp[0][0] * Cconj[0][0] + Ctemp[0][1] *
//  Cconj[0][1];
ippsMul_64fc(&(Ctemp[0][0]), &(Cconj[0][0]), &invtemp2, 1);
ippsMul_64fc(&(Ctemp[0][1]), &(Cconj[0][1]), &invtemp, 1);
ippsAdd_64fc_I(&invtemp2, &invtemp, 1);
H11[n].re = invtemp.re;
H11[n].im = invtemp.im;
```

Codice 50: Calcolo dell'n-esima armonica di `H11`

In modo analogo si calcolano le variabili `H12`, `H21` e `H22` come di seguito.

```
//invtemp = Ctemp[0][0] * Cconj[1][0] + Ctemp[0][1] *
    //Cconj[1][1];
ippsMul_64fc(&(Ctemp[0][0]), &(Cconj[1][0]), &invtemp2, 1);
ippsMul_64fc(&(Ctemp[0][1]), &(Cconj[1][1]), &invtemp, 1);
ippsAdd_64fc_I(&invtemp2, &invtemp, 1);
H12[n].re = invtemp.re;
H12[n].im = invtemp.im;
//invtemp = Ctemp[1][0] * Cconj[0][0] + Ctemp[1][1] *
    //Cconj[0][1];
ippsMul_64fc(&(Ctemp[1][0]), &(Cconj[0][0]), &invtemp2, 1);
ippsMul_64fc(&(Ctemp[1][1]), &(Cconj[0][1]), &invtemp, 1);
ippsAdd_64fc_I(&invtemp2, &invtemp, 1);
H21[n].re = invtemp.re;
H21[n].im = invtemp.im;
//invtemp = Ctemp[1][0] * Cconj[1][0] + Ctemp[1][1] *
    //Cconj[1][1];
ippsMul_64fc(&(Ctemp[1][0]), &(Cconj[1][0]), &invtemp2, 1);
ippsMul_64fc(&(Ctemp[1][1]), &(Cconj[1][1]), &invtemp, 1);
ippsAdd_64fc_I(&invtemp2, &invtemp, 1);
H22[n].re = invtemp.re;
H22[n].im = invtemp.im;
}
```

Codice 51: Calcolo dell'n-esima armonica di `H12, H21, H22`

Una volta eseguito il ciclo `for` iniziato nel codice 45 e terminato nel 51, la funzione `LEPlugin_Init` finisce e sono ora noti i filtri H_{11}, H_{12}, H_{11}, H_{21} e H_{22} per effettuare la correzione del crosstalk.

A questo punto la funzione `LEPlugin_Process` effettua il filtraggio con i filtri di ricostruzione usando l'overlap and save in modo analogo a quanto visto con i codici 14, 15, 16, 17, 18 e 19 in Matlab. Inizialmente, per semplicità si copiano i puntatori degli ingressi in `InputData1` e `InputData2` e delle uscite in `OutputData1` e `OutputData2`.

```
int __stdcall PlugIn::LEPlugin_Process(PinType** Input,
    PinType** Output, LPVOID ExtraInfo)
{
    double* InputData1 = ((double*)Input[0]->DataBuffer);
    double* OutputData1 = ((double*)Output[0]->DataBuffer);
    double* InputData2 = ((double*)Input[1]->DataBuffer);
    double* OutputData2 = ((double*)Output[1]->DataBuffer);
```

Codice 52: Copia dei puntatori degli ingressi e delle uscite

Successivamente, vengono riempiti i campi reali dei buffer di ingresso `x1buff` e `x2buff` dall'indice `M - 1` fino a `L + M - 1` copiando `L` valori degli ingressi. I buffer vengono poi normalizzati dividendo per $2^{15} = 32768$ per non andare in overflow e per poter confrontare gli stessi segnali in Matlab.

```
for (int i = 0; i < L; i++)
{
    x1buff[M - 1 + i].re = InputData1[i];
    x2buff[M - 1 + i].re = InputData2[i];
}
// divide L elements of the vector x1buff by 32768 and store
//  the result in x1buff
ippsDivC_64fc_I({ 32768.0, 0.0 }, x1buff + M - 1, L);
// divide L elements of the vector x2buff by 32768 and store
//  the result in x2buff
ippsDivC_64fc_I({ 32768.0, 0.0 }, x2buff + M - 1, L);
```

Codice 53: Creazione dei buffer di ingresso e normalizzazione

Una volta riempiti i buffer di ingresso `x1buff` e `x2buff`, ne viene fatta la FFT e salvato il risultato rispettivamente in `X1BUFF` e `X2BUFF`.

```
ippsFFTFwd_CToC_64fc(x1buff, X1BUFF, pSpec, pMemBuffer);
ippsFFTFwd_CToC_64fc(x2buff, X2BUFF, pSpec, pMemBuffer);
```

Codice 54: FFT dei buffer di ingresso

Poiché non è possibile implementare in una sola riga l'equazione (31) come mostrato nel codice 16 di Matlab, occorre dividere la formula in più passaggi. Scrivendo (31) come sistema lineare si ha

$$\begin{cases} Y_1 = (C_{11} \cdot H_{11} + C_{12} \cdot H_{21}) \cdot X_1 + (C_{11} \cdot H_{12} + C_{12} \cdot H_{22}) \cdot X_2 \\ Y_2 = (C_{21} \cdot H_{11} + C_{22} \cdot H_{21}) \cdot X_1 + (C_{21} \cdot H_{12} + C_{22} \cdot H_{22}) \cdot X_2 \end{cases} \tag{47}$$

Sia temp $= C_{11} \cdot H_{11}$, $\text{temp}_2 = C_{12} \cdot H_{21}$, allora il primo addendo della prima riga del sistema (47) è data da

$$\text{temp}_3 = (\text{temp} + \text{temp}_2) \cdot X_1 = (C_{11} \cdot H_{11} + C_{12} \cdot H_{21}) \cdot X_1.$$

Allo stesso modo si può calcolare il secondo addendo e sommarli per trovare Y_1. Il calcolo di Y_2 è analogo. L'implementazione del sistema (47) è mostrato nel codice 55, in cui vengono calcolati i buffer di uscita in frequenza `Y1BUFF` e `Y2BUFF`.

```
// temp = C11.*H11
ippsMul_64fc(C11, H11, temp, fftLen);
// temp2 = C12.*H21
ippsMul_64fc(C12, H21, temp2, fftLen);
// temp2 = C11.*H11+C12.*H21
ippsAdd_64fc_I(temp, temp2, fftLen);
// temp3 = (C11.*H11 + C12.*H21).*X1BUFF
ippsMul_64fc(temp2, X1BUFF, temp3, fftLen);
// temp = C11.*H12
ippsMul_64fc(C11, H12, temp, fftLen);
// temp2 = C12.*H22
ippsMul_64fc(C12, H22, temp2, fftLen);
// temp2 = C11.*H12+C12.*H22
ippsAdd_64fc_I(temp, temp2, fftLen);
// temp = (C11.*H12+C12.*H22).*X2BUFF
ippsMul_64fc(temp2, X2BUFF, temp, fftLen);
// Y1BUFF = (C11.*H11 + C12.*H21).*X1BUFF +
//  (C11.*H12+C12.*H22).*X2BUFF
ippsAdd_64fc(temp3, temp, Y1BUFF, fftLen);

// temp = C21.*H11
ippsMul_64fc(C21, H11, temp, fftLen);
// temp2 = C22.*H21
ippsMul_64fc(C22, H21, temp2, fftLen);
// temp2 = C21.*H11+C22.*H21
ippsAdd_64fc_I(temp, temp2, fftLen);
// temp3 = (C21.*H11+C22.*H21).*X1BUFF
ippsMul_64fc(temp2, X1BUFF, temp3, fftLen);
// temp = C21.*H12
ippsMul_64fc(C21, H12, temp, fftLen);
// temp2 = C22.*H22
ippsMul_64fc(C22, H22, temp2, fftLen);
// temp2 = C21.*H12+C22.*H22
ippsAdd_64fc_I(temp, temp2, fftLen);
// temp = (C21.*H12+C22.*H22).*X2BUFF
ippsMul_64fc(temp2, X2BUFF, temp, fftLen);
// Y2BUFF = (C21.*H11+C22.*H21).*X1BUFF +
```

```
//  (C21.*H12+C22.*H22).*X2BUFF
ippsAdd_64fc(temp3, temp, Y2BUFF, fftLen);
```

Codice 55: Calcolo dei buffer di uscita in frequenza

Procedendo con l'algoritmo dell'overlap and save occorre ora fare le IFFT dei buffer di uscita e salvarle nelle variabili `y1buff` e `y2buff`.

```
ippsFFTInv_CToC_64fc(Y1BUFF, y1buff, pSpec, pMemBuffer);
ippsFFTInv_CToC_64fc(Y2BUFF, y2buff, pSpec, pMemBuffer);
```

Codice 56: Calcolo dei buffer di uscita nel tempo con la IFFT

L'aggiornamento dell'uscita avviene moltiplicando i buffer di uscita per $2^{15} = 32768$ e copiando `L` elementi dall'indice `M - 1` all'indice `L + M - 1` dei valori reali di `y1buff` e `y2buff`.

```
// output update
// multiply each element of the vector y1buff by 32768 and
//  store the result in y1buff
ippsMulC_64fc_I({ 32768.0, 0.0 }, y1buff, fs);
// multiply each element of the vector y2buff by 32768 and
//  store the result in y2buff
ippsMulC_64fc_I({ 32768.0, 0.0 }, y2buff, fs);
for (int i = 0; i < L; i++)
{
    OutputData1[i] = y1buff[M - 1 + i].re;
    OutputData2[i] = y2buff[M - 1 + i].re;
}
```

Codice 57: Aggiornamento delle uscite

L'ultimo passo consiste nell'aggiornare il buffer di ingresso spostando gli ultimi `M - 1` valori all'inizio del buffer successivo con la funzione `ippsMove_64fc`.

```
ippsMove_64fc(x1buff + L, x1buff, M - 1);
ippsMove_64fc(x2buff + L, x2buff, M - 1);
```

Codice 58: Aggiornamento dei buffer di ingresso

Nella funzione Delete, come precedentemente mostrato nel codice 32, vengono liberate dalla memoria tutte le variabili che erano state allocate nella Init.

```
void __stdcall PlugIn::LEPlugin_Delete()
{
    if (x1buff != 0)
```

```
    {
        ippsFree(x1buff);
        x1buff = 0;
    }
```

Codice 59: Funzione `LEPlugin_Delete`

L'RT Watch per cambiare il parametro β viene dichiarato in modo analogo a quanto visto per l'algoritmo LMS nel codice 33. Come per LMS, La variabile da modificare nel template è di tipo `double` e di dimensione `sizeof(double)` byte.

```
void __stdcall PlugIn::LERTWatchInit()
{
    WatchType NewWatch;

    memset(&NewWatch, 0, sizeof(WatchType));
    NewWatch.EnableWrite = true;
    NewWatch.LenByte = sizeof(double);
    NewWatch.TypeVar = WTC_DOUBLE;
    NewWatch.IDVar = ID_BETA;
    sprintf(NewWatch.VarName, "Beta\0");
    CBFunction(this, NUTS_ADDRTWATCH, 0, &NewWatch);
}
```

Codice 60: Creazione RT Watch

Il parametro β può essere variato dall'utente direttamente dal NU-TECH. Si usa `LEGetParameter` per portare la variabile `beta` nella board del NU-TECH come precedentemente illustrato nel codice 34.

```
int  __stdcall PlugIn::LEGetParameter(int Index,void *Data)
{

    CBFunction(this, NUTS_GETSECURETIME, NUTSSECURE, 0);

    if (Index == ID_BETA)
    {
        memcpy((double*)Data, &beta, sizeof(double));
    }

    CBFunction(this, NUTS_RELEASESECURETIME, NUTSSECURE, 0);
    return 0;
}
```

Codice 61: Funzione `LEGetParameter`

Invece, se il programma non è in esecuzione l'utente può variare il β poiché il calcolo dei filtri di ricostruzione H con la formula (30) avviene a priori nella Init, cioè prima di iniziare qualsiasi operazione sul segnale. Cambiare il β durante la process non porterebbe modifiche all'uscita. La funzione `LESetParameter` è descritta nel codice 62.

```
void __stdcall PlugIn::LESetParameter(int Index,void *Data,
    LPVOID bBroadCastInfo)
{

    CBFunction(this, NUTS_GETSECURETIME, NUTSSECURE, 0);

    if(Index == ID_BETA)
    {
        if (!isRunning)
        {
            memcpy(&beta, (double*)Data, sizeof(double));
            CBFunction(this, NUTS_UPDATERTWATCH, ID_BETA, 0);
        }

    }

    CBFunction(this, NUTS_RELEASESECURETIME, NUTSSECURE, 0);
}
```

Codice 62: Funzione `LESetParameter`

7 Risultati

In questo capitolo vengono presentati tutti i risultati ottenuti dall'implementazione dei due programmi. Abbiamo confrontato l'algoritmo LSM e FD usando il fattore di separazione dei canali (channel separation factor). Questo fattore è definito come il rapporto tra la potenza del percorso ipsilaterale e quello controlaterale, come mostrato nelle equazioni (48) e (49) rispettivamente per il canale sinistro e destro.

$$J_L = E\left\{20 \cdot \log_{10} \frac{C_{11}H_{11} + C_{12}H_{21}}{C_{21}H_{11} + C_{22}H_{21}}\right\} \quad [\mathrm{dB}] \tag{48}$$

$$J_R = E\left\{20 \cdot \log_{10} \frac{C_{22}H_{22} + C_{21}H_{12}}{C_{12}H_{22} + C_{11}H_{12}}\right\} \quad [\mathrm{dB}] \tag{49}$$

dove $E\{\cdot\}$ indica il valore medio su tutte le frequenze.

Le tabelle 1 e 2 mostrano i valori di channel separation factor rispettivamente per LMS e FD al variare di μ e β.

μ	J_R [dB]	J_L [dB]
10^{-3}	22.032	21.924
$5 \cdot 10^{-4}$	17.610	20.375
10^{-4}	11.063	16.921
$5 \cdot 10^{-5}$	9.742	16.161

Tabella 1: Confronto di J_R e J_L per l'algoritmo LMS per diversi μ.

β	J_R [dB]	J_L [dB]
1	25.062	26.538
0.3	28.829	30.498
0.1	34.908	36.532
0.01	47.657	48.661

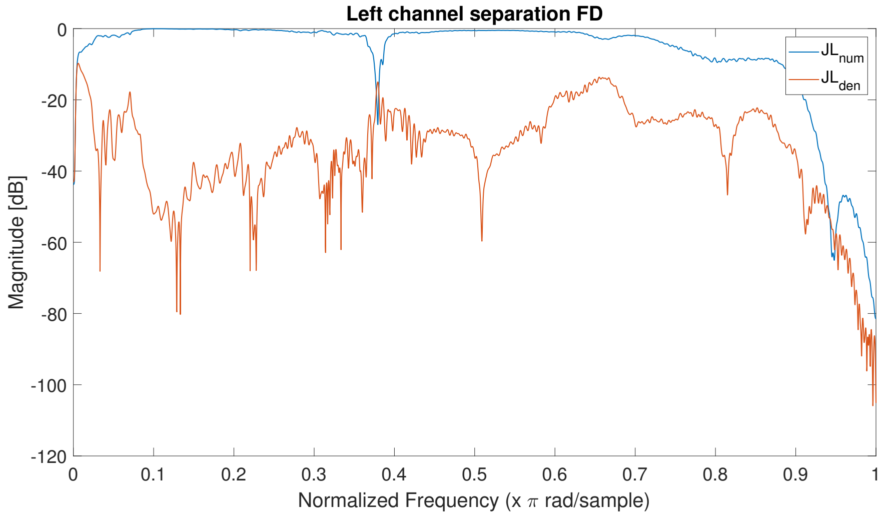
Tabella 2: Confronto di J_R e J_L per l'algoritmo Fast Deconvolution per diversi β.

La figura 5 mostra l'andamento in frequenza del numeratore e denominatore delle formule (48) e (49) per entrambi gli algoritmi implementati. Poiché le figure sono esposte in dB si può fare la differenza tra il numeratore e il denominatore per ottenere l'effettivo valore di J_L e J_R per ogni frequenza.

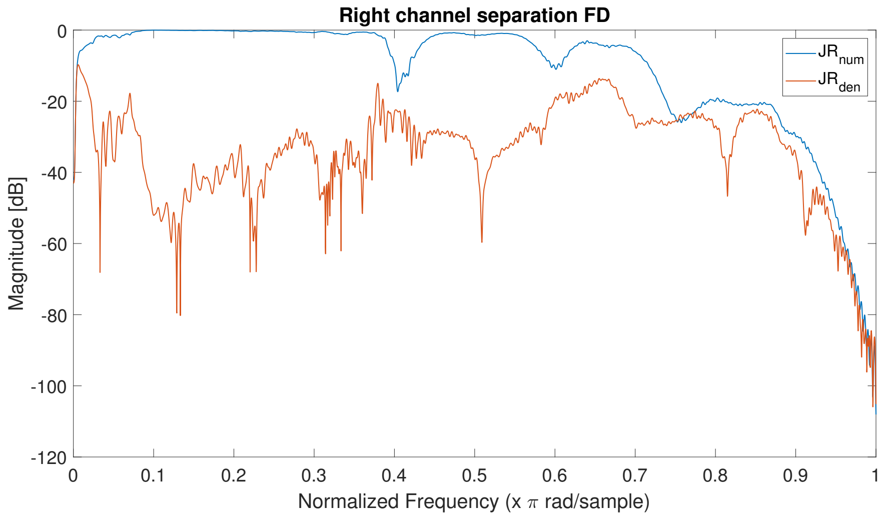
La figura 6 mostra l'andamento nel tempo del mean square error in dB dell'algoritmo LMS. Dalla formula (19) si ha che l'n-esimo valore dell'errore è dato dalla differenza tra l'n-esimo valore del segnale desiderato e dall'n-esimo valore dell'uscita.

La figura 7 mostra un confronto tra i filtri di ricostruzione H_{11}, H_{12}, H_{21} e H_{22} degli algoritmi LMS e FD. Un confronto tra i filtri ottenuti nel progetto, figura 7, e quelli proposti nel paper, figura 8, mostra che per alcune frequenze non si ha una perfetta uguaglianza tra i filtri calcolati con LMS e quelli calcolati con FD.

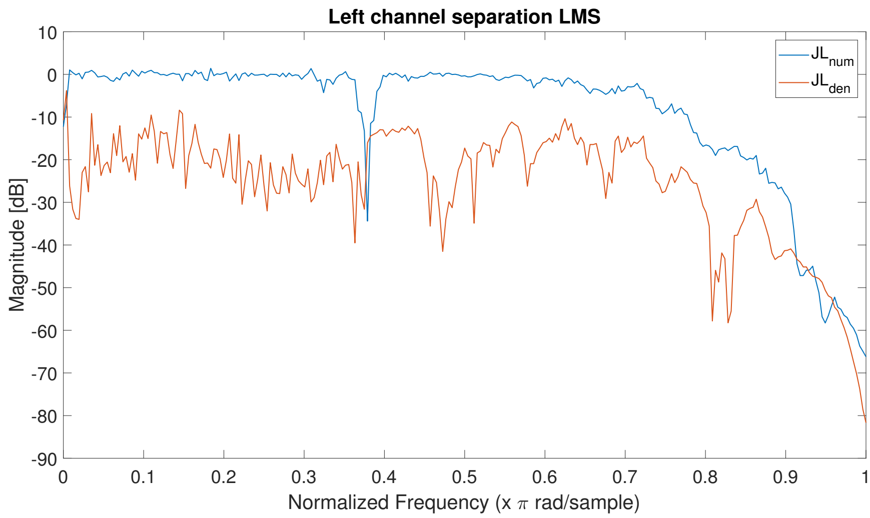
Le figure 9a e 9b mostrano un confronto tra gli ingressi e le uscite nell'algoritmo Fast Deconvolution, mentre le figure 9c e 9d confrontano il segnale desiderato con l'uscita prodotta dall'LMS.


Left channel separation FD
Magnitude [dB]
Normalized Frequency (x π rad/sample)
JL_num
JL_den

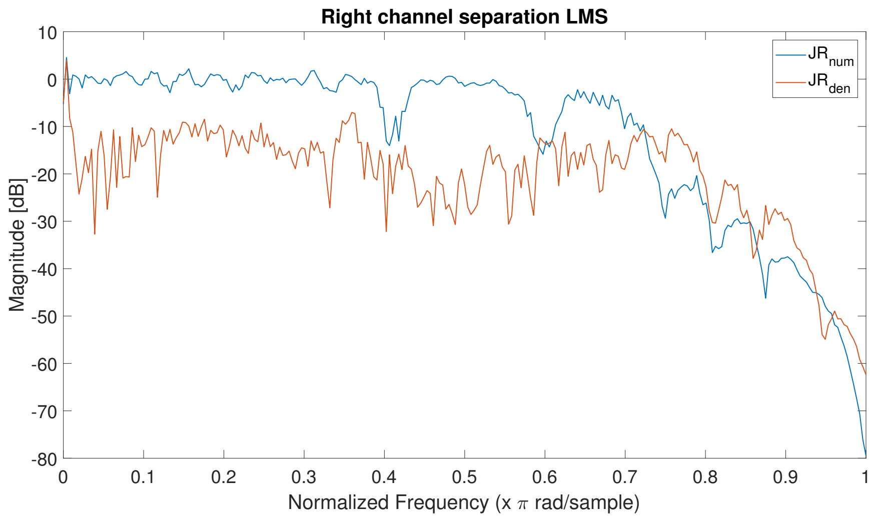
(a)


Right channel separation FD
Magnitude [dB]
Normalized Frequency (x π rad/sample)
JR_num
JR_den

(b)

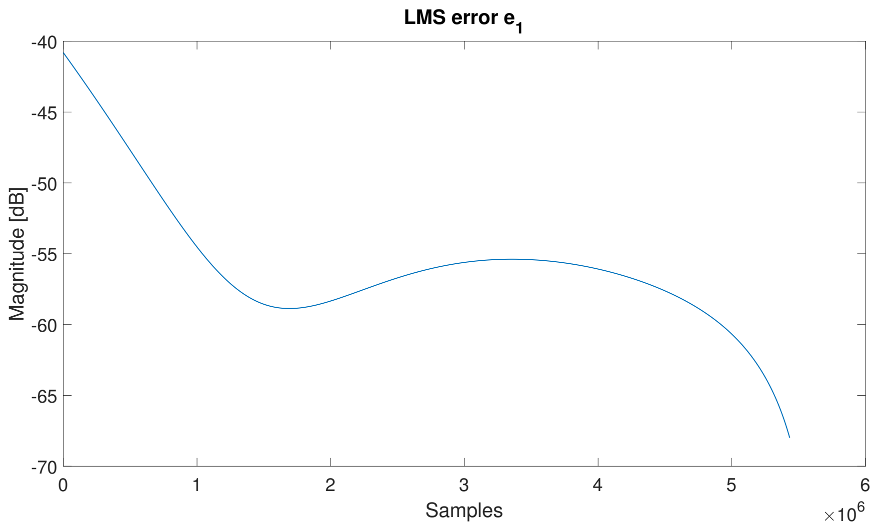


(c)

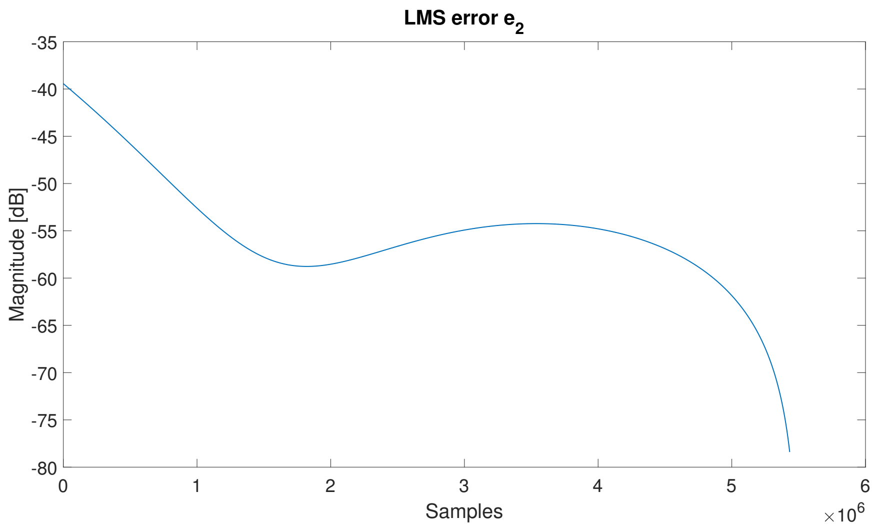


(d)

Figura 5: Confronto del numeratore e del denominatore di (a) J_L e (b) J_R per FD con $\beta = 0.3$, (c) J_L e (d) J_R per LMS con $\mu = 10^{-3}$.

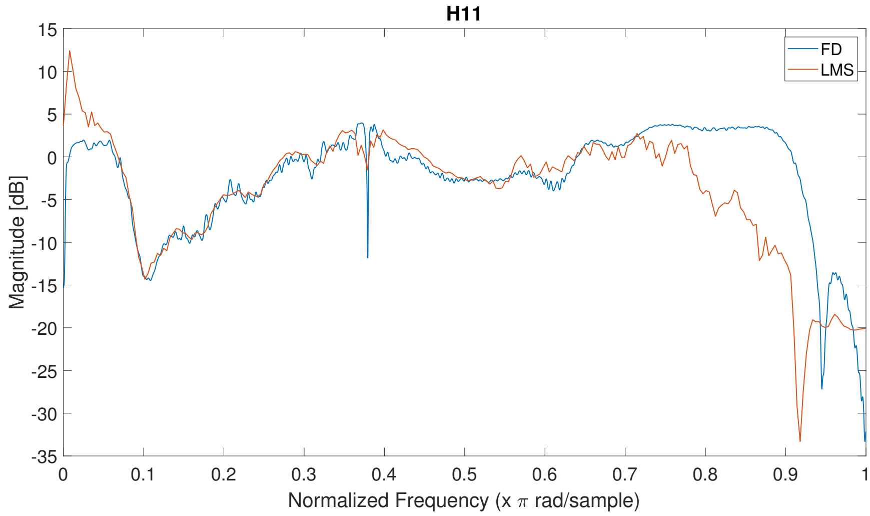


(a)

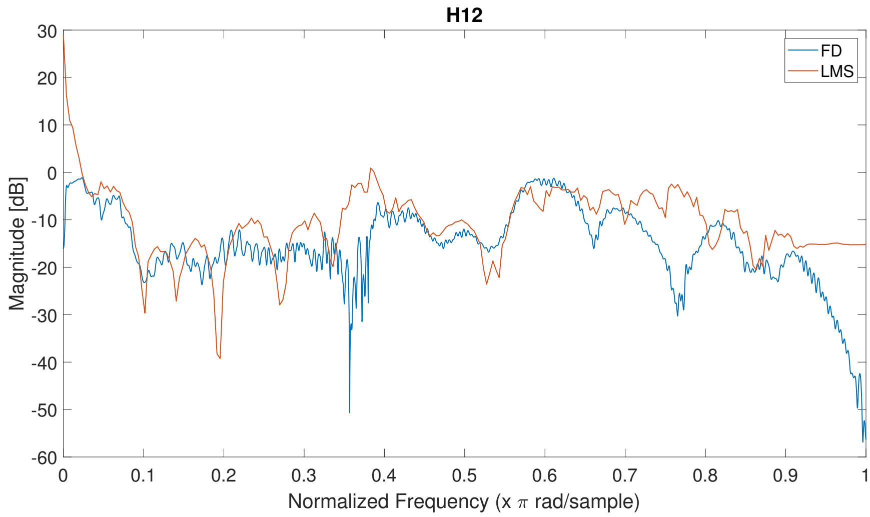


(b)

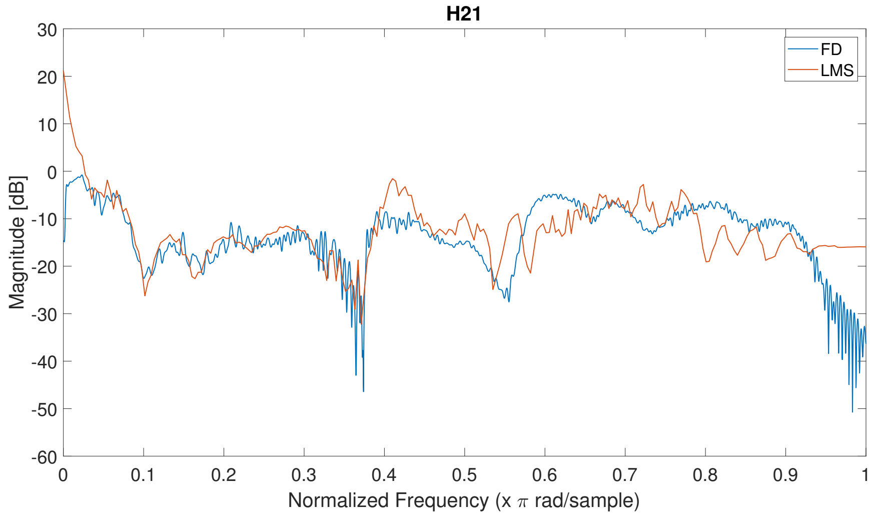
Figura 6: Confronto dell'MSE del canale (a) sinistro e (b) destro dell'algoritmo LMS.


H11
FD
LMS
Magnitude [dB]
Normalized Frequency (x π rad/sample)

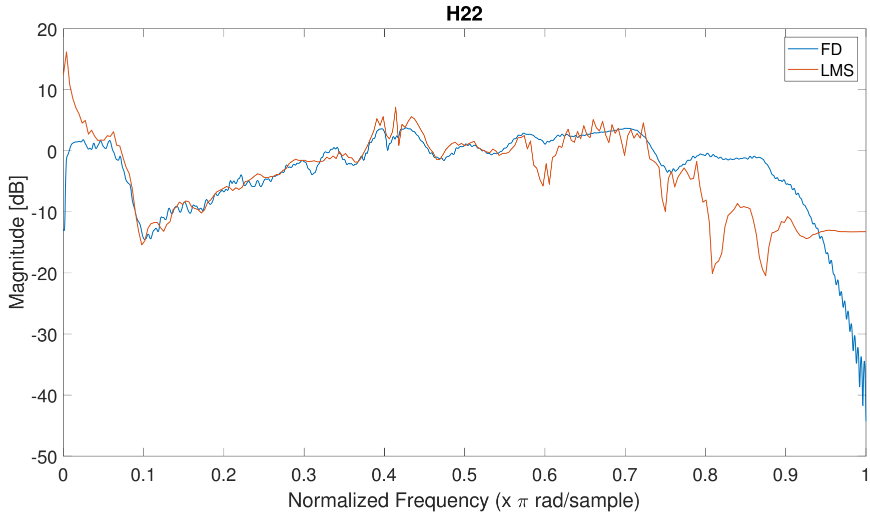
(a)


H12
FD
LMS
Magnitude [dB]
Normalized Frequency (x π rad/sample)

(b)

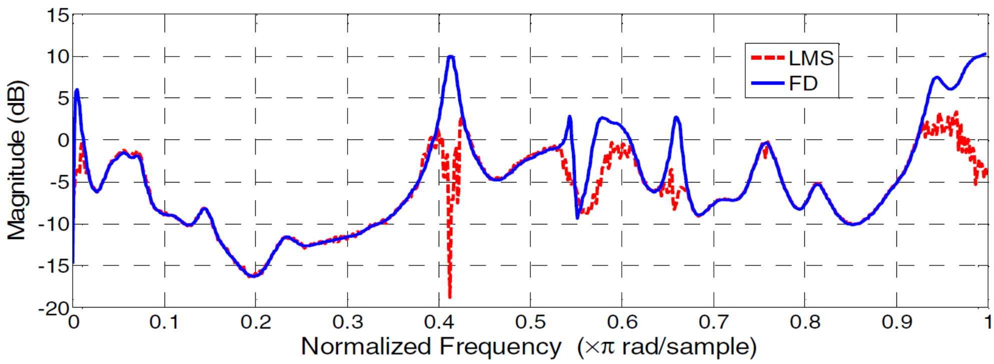


(c)



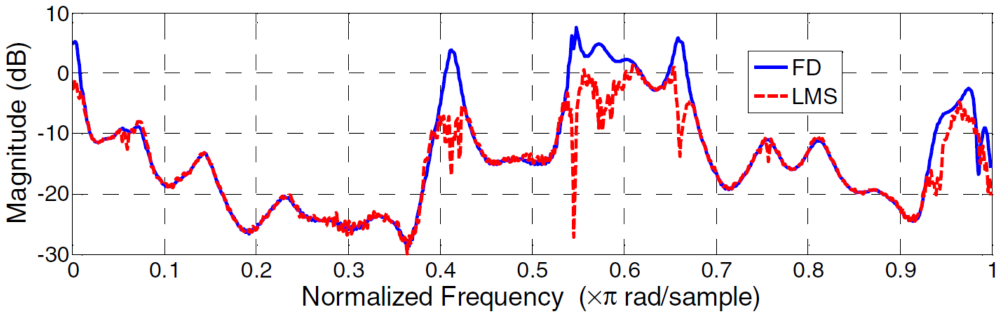
(d)

Figura 7: Confronto dei filtri di cancellazione del crosstalk (a) H_{11}, (b) H_{12}, (c) H_{21}, (d) H_{22} di LMS e Fast Deconvolution.



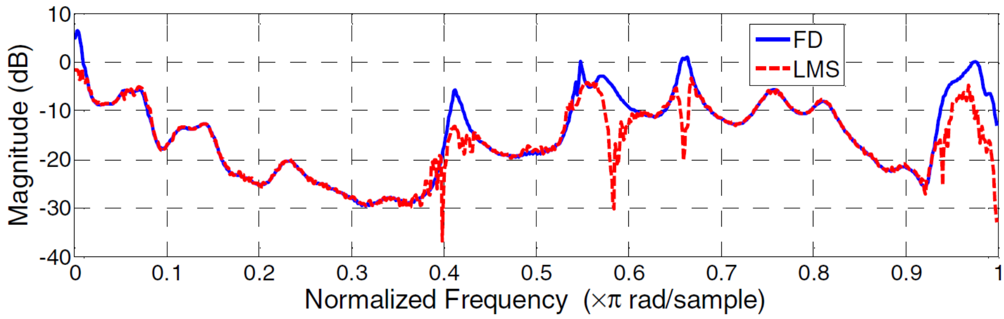
LMS
FD
Magnitude (dB)
Normalized Frequency (×π rad/sample)

(a)

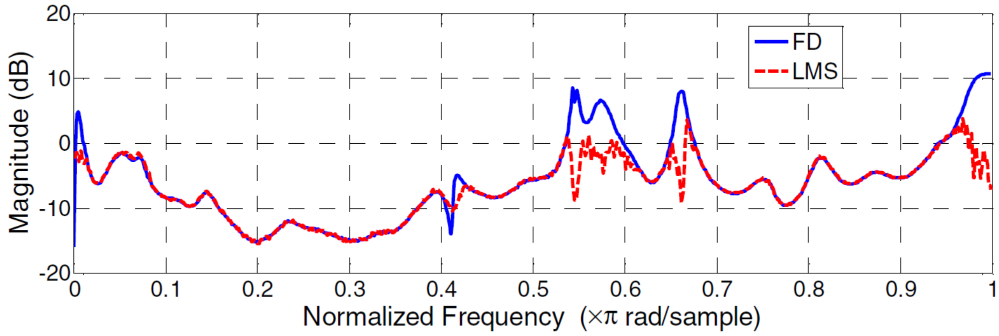


FD
LMS
Magnitude (dB)
Normalized Frequency (×π rad/sample)

(b)

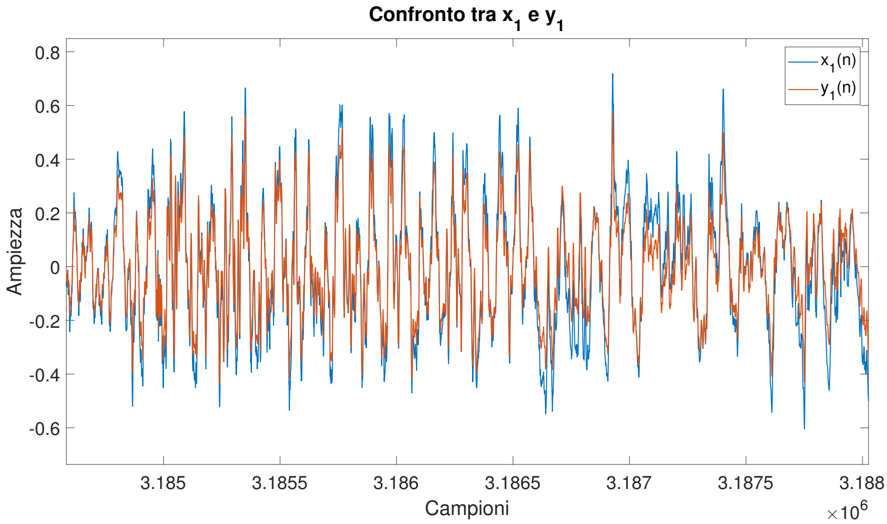


(c)

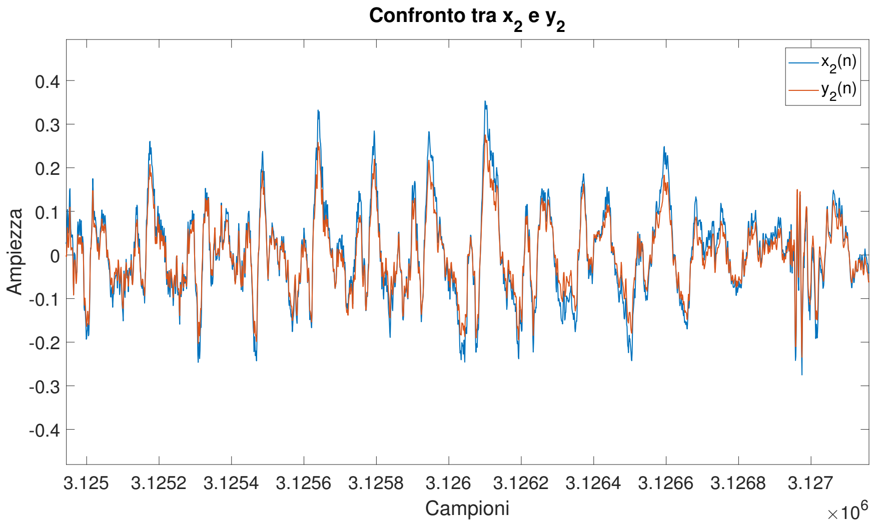


(d)

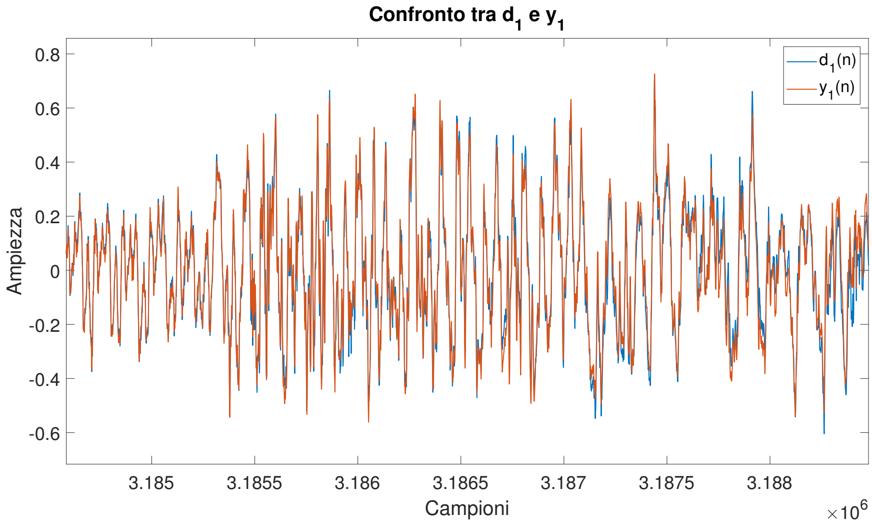
Figura 8: Confronto dei filtri di cancellazione del crosstalk (a) H_{11}, (b) H_{12}, (c) H_{21}, (d) H_{22} di LMS e Fast Deconvolution dell'articolo [8].


Confronto tra x_1 e y_1
$x_1(n)$
$y_1(n)$
Ampiezza
0.8
0.6
0.4
0.2
0
-0.2
-0.4
-0.6
3.185
3.1855
3.186
3.1865
3.187
3.1875
3.188
Campioni
$\times 10^6$

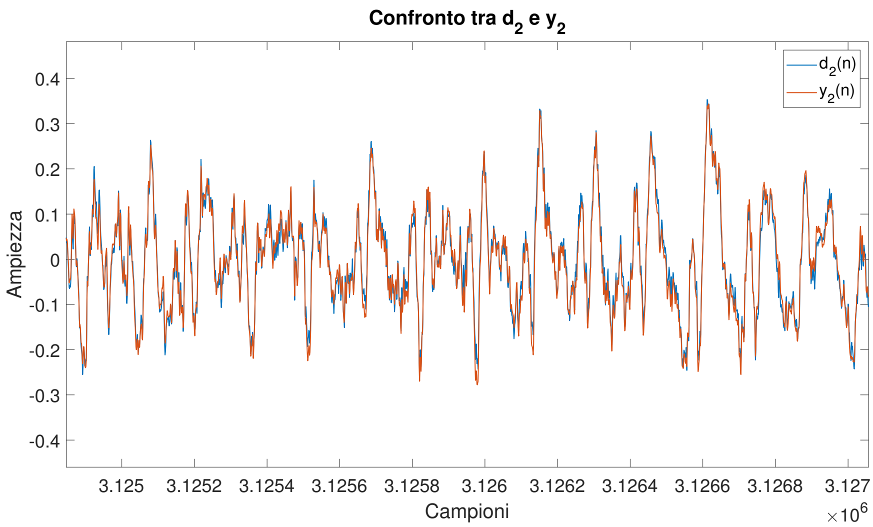
(a)


Confronto tra x_2 e y_2
$x_2(n)$
$y_2(n)$
Ampiezza
0.4
0.3
0.2
0.1
0
-0.1
-0.2
-0.3
-0.4
3.125
3.1252
3.1254
3.1256
3.1258
3.126
3.1262
3.1264
3.1266
3.1268
3.127
Campioni
$\times 10^6$

(b)



(c)



(d)

Figura 9: Confronto degli ingressi e le uscite (a) sinistro e (b) destro della Fast Deconvolution, (c) sinistro e (d) destro di LMS.

8 Conclusioni

Dai risultati mostrati nel capitolo 7, si possono fare diverse osservazioni. Si può notare dalla figura 5 che nel fattore di separazione dei canali il numeratore è quasi sempre maggiore del denominatore: questo indica che la separazione dei canali sta avvenendo in modo corretto.

Si può osservare dalla figura 6 che all'aumentare del tempo l'errore dell'algorimo LMS diminuisce: questo indica che l'algoritmo produrrà un'uscita sempre più vicina a quella desiderata.

Dalla figura 7 si nota che i filtri H_{11} e H_{22} trovati con LMS e FD sono più simili tra loro rispetto ai filtri H_{12} e H_{21}.

Si può notare dalla tabella 1 che il fattore di separazione dei canali aumenta all'aumentare di μ, mentre per la tabella 2 questo fattore aumenta al diminuire di β. Entrambi i parametri vengono aggiustati durante l'implementazione pratica, in particolare, il comportamento del β per la Fast Deconvolution è in linea con quanto ci si aspetta dalla formula (24) spiegata nel capitolo 4.2.

A differenza del paper di riferimento [8], in cui il fattore di separazione dei canali è migliore per il canale destro, nel nostro caso otteniamo valori migliori per il canale sinistro. Questo può essere dovuto ad una diversa scelta delle HRIR tra l'implementazione dell'articolo e la nostra. Nelle simulazioni dell'articolo, infatti, è stato utilizzato il valore medio di 10 set di HRTF per calcolare la matrice di cancellazione della diafonia H, e successivamente sono stati utilizzati i 10 set originali di HRTF per calcolare i rispettivamente fattori di separazione dei canali.

Dalle tabelle 1 e 2 si ottengono dei fattori di separazione dei molto simili a quelli proposti in [8], in cui si ha in media $J_L = 12.6251\,\mathrm{dB}$ e $J_R = 16.0611\,\mathrm{dB}$ per LMS e $J_L = 12.8147\,\mathrm{dB}$ e $J_R = 17.2171\,\mathrm{dB}$ per FD. Possiamo dunque affermare che i filtri di cancellazione del crosstalk lavorano in modo corretto.

Dal confronto tra l'uscita desiderata e l'ingresso in figura 9 si può notare come questi segnali siano sovrapposti: possiamo sostenere dunque che la cancellazione del crosstalk avviene in modo corretto.

I risultati ottenuti per ognuno dei due algoritmi con il Nu-Tech sono uguali a quelli prodotti dal Matlab, questo ci permette di affermare che gli algoritmi sviluppati nei due diversi linguaggi di programmazione sono equivalenti e producono le stesse uscite.

Riferimenti bibliografici

[1] E. Ferrara. «Fast implementations of LMS adaptive filters». In: *IEEE Transactions on Acoustics, Speech, and Signal Processing* 28 (1980), pp. 474–475.

[2] B. Gardner. «HRTF Measurements of a KEMAR Dummy-Head Microphone». In: 1994.

[3] Emmanuel C. Ifeachor e Barrie W. Jervis. *Digital Signal Processing: A Practical Approach*. 2nd. Pearson Education, 2002. ISBN: 0201596199.

[4] Jun Seong Kim, Sang Gyun Kim e Chang D. Yoo. «A Novel Adaptive Crosstalk Cancellation using Psychoacoustic Model for 3D Audio». In: *2007 IEEE International Conference on Acoustics, Speech and Signal Processing - ICASSP '07*. Vol. 1. 2007, pp. I-185-I–188. DOI: 10.1109/ICASSP.2007.366647.

[5] O. Kirkeby, P.A. Nelson, H. Hamada e F. Orduna-Bustamante. «Fast deconvolution of multichannel systems using regularization». In: *IEEE Transactions on Speech and Audio Processing* 6.2 (1998), pp. 189–194. DOI: 10.1109/89.661479.

[6] O. Kirkeby, P. Rubak e Angelo Farina. «Analysis of ill-conditioning of multi-channel deconvolution problems». In: feb. 1999, pp. 155–158. ISBN: 0-7803-5612-8.

[7] Ole Kirkeby, Per Rubak, Philip Nelson e Angelo Farina. «Design of Cross-Talk Cancellation Networks by Using Fast Deconvolution». In: (nov. 2000).

[8] Dan Li, Zhong-Hua Fu, Lei Xie e Yanning Zhang. «Comprehensive comparison of the least mean square algorithm and the fast deconvolution algorithm for crosstalk cancellation». In: *2012 International Conference on Audio, Language and Image Processing*. 2012, pp. 224–229. DOI: 10.1109/ICALIP.2012.6376616.

[9] P.A. Nelson, H. Hamada e S.J. Elliott. «Adaptive inverse filters for stereophonic sound reproduction». In: *IEEE Transactions on Signal Processing* 40.7 (1992), pp. 1621–1632. DOI: 10.1109/78.143434.

[10] Lin Wang e Yin Fuliang. «A Stereo Crosstalk Cancellation System Based on the Common-Acoustical Pole/Zero Model». In: *EURASIP Journal on Advances in Signal Processing* 2010 (gen. 2010). DOI: 10.1155/2010/719197.