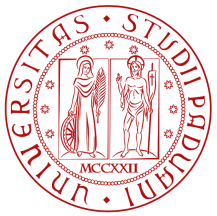


Università
degli Studi
di Padova

DIPARTIMENTO
DI INGEGNERIA
DELL'INFORMAZIONE

Tesi di Laurea Triennale in Ingegneria dell'Informazione

Simulazione Di Cuffie High-Fidelity utilizzando Virtual Studio Technology

Studente: **Dario Benvegnù matr. 2014556**
Relatore: **Prof. Sergio Canazza Targon**
Correlatore: **Dott.ssa Anna Zuccante**

Anno Accademico 2023/2024
Data di Laurea : 14 Marzo 2024

A Marta, alla mia famiglia e ai miei amici

Sommario

Questa tesi espande la tesi magistrale della Dott.ssa Anna Zuccante, proponendo una versione del suo software di simulazione di cuffie Hi-fi compatibile con l'utilizzo su Digital Audio Workstations (DAW).
L'idea alla base del software è di poter replicare il suono di un paio di cuffie, definite Target, attraverso un diverso paio di cuffie, definite Monitor.
La possibilità di riprodurre l'esperienza di ascolto data da un certo modello di cuffie attraverso delle altre cuffie è interessante per diversi motivi, tanto nel campo della ricerca quanto per applicazioni commerciali pensate per aiutare il consumatore nell'acquisto di una cuffia.
Ciò che il software qui proposto aggiunge all'applicazione sviluppata in precedenza è la compatibilità con l'ambiente di produzione audio tipico delle DAW, fornendo una versione dell'applicazione sviluppata dalla dottoressa Zuccante che sia compatibile con la maggior parte dei software di elaborazione audio disponibili in commercio.
L'obbiettivo principale della versione plugin è infatti di fornire a produttori musicali e ingegneri di mix e master la possibilità di simulare la risposta di diversi modelli di cuffie Hi-Fi, per poter valutare il proprio lavoro su diversi sistemi di riproduzione senza dover necessariamente acquistarne molteplici. Il principale vantaggio della traduzione di questo software su interfaccia audio plugin VST è la possibilità di eseguire una comparazione pressoché instantanea di cuffie diverse, eludendo così i problemi dati dalla brevità della memoria ecogena.
Sono analizzate e discusse tre possibili versioni di questo software, realizzate con diversi approcci implementativi.

Contenuti

1 Introduzione

1.1 Digital Audio Workstations - Plugin Audio

In generale il termine DAW si riferisce a sistemi informatici compresi di un convertitore analogico digitale per segnali sonori interfacciato ad un computer, provvisto di un software per la registrazione, l'elaborazione e il montaggio audio. Tuttavia nell'uso comune parlando di DAW ci si riferisce solamente al software in questione. La Virtual Studio Technology [1] é invece uno standard di interfaccia nato nel 1996, e pensato per facilitare la comunicazione tra DAWs e componenti software aggiuntivi detti audio *plug-ins*. Questi ultimi sono software progettati per integrare funzionalità aggiuntive ed ampliare le capacitá delle applicazioni di elaborazione audio. Tali software offrono a terze parti la possibilità di sviluppare estensioni di qualsiasi tipo, dai più semplici effetti a complessi strumenti di analisi. Attualmente, essi costituiscono una parte integrante nel settore dell'audio professionale; infatti esistono migliaia di plugin VST in commercio che, salvo rare eccezioni, possono essere generalmente suddivisi in due categorie:

- ***VSTfx:*** È lo standard utilizzato per gli effetti audio. I plugin che ricadono in questa categoria infatti non sono in grado di generare autonomamente segnali sonori ma sono processori di segnale pensati per modificare le caratteristiche del suono, come equalizzatori, compressori, e vari altri tipi di effetti come riverberi, distorsioni, modulazioni. In questa questa categoria rientrano anche gli stumenti di analisi, i quali, senza apportare modifiche al segnale fornito, consentono lo studio di diverse caratteristiche di interesse, come spettrogramma, immagine stereofonica, inviluppo temporale o forma d'onda su brevi periodi.
- ***VSTi:*** È lo standard utilizzato per sintetizzatori audio. In questa categoria ricadono tutti i plugin in grado di generare segnali audio, tipicamente tutti i tipi di sintetizzatori software (sottrattivi, additivi, FM...), insieme a campionatori (intesi come software in grado di manipolare in diversi modi segnali audio preregistrati) e drum machines.

Ogni plugin sviluppato in questa tesi è del primo tipo, infatti esso è pensato per modificare il segnale audio in modo da simulare il più accuratamente possibile l'esperienza di ascolto data dalle cuffie Target selezionate, attraverso le cuffie Monitor in utilizzo.

1.2 Motivazione - Memoria Ecogena

La memoria Ecogena, nota anche come memoria fonologica a breve termine, è una componente della nostra memoria sensoriale, responsabile della conservazione delle informazioni sonore provenienti dall'ambiente. Questa memoria è in grado di registrare un grande quantitativo di informazioni, tuttavia, come il suo nome implica, la sua durata è relativamente breve (3-4 secondi in media [2]), caratteristica che la rende particolarmente difficile da gestire.
Come accennato in precedenza, un notevole vantaggio dato dall'utilizzo di plugin audio all'interno di una DAW è la possibilità di lavorare in real time. Ciò consente di eseguire comparazioni pressochè istantanee tra un segnale processato con un plugin e il segnale originale. Questo approccio, conosciuto nel linguaggio tecnico come *"Confronto A/B"*, è usato in tutti i settori dell'ingegneria del suono poichè consente di eludere le brevi tempistiche della memoria ecogena, fornendo un modo affidabile e corretto di paragonare segnali audio simili. Nel caso specifico grazie a questa capacità siamo in grado di confrontare la risposta di due modelli di cuffie diversi pressochè istantaneamente.

1.3 Ambiente di Sviluppo Utilizzato

Generalmente i plugin VST sono sviluppati usando il linguaggio `C++` e ambienti di sviluppo integrati come il framework JUCE, molto utilizzato nell'ambito dell'audio professionale.
Tuttavia l'ambiente di sviluppo Matlab, dalla versione R2016b in poi, fornisce svariate classi e strumenti che permettono di programmare e prototipare applicazioni e plugin audio utilizzando il linguaggio di programmazione integrato, per poi tradurre automaticamente il codice sviluppato in formato .vst. [3]
Nello specifico, si fa utilizzo della superclasse `audioPlugin` [4], una classe *handle* dalla quale vengono ereditati tutti gli attributi necessari per generare plugin audio e per accedere alle funzionalità di `Audio Toolbox` [5]. In concomitanza a questa superclasse sono di comune utilizzo funzioni fornite da `DSP System Toolbox` [6] e da `Signal Processing Toolbox` [7], che mettono a disposizione diverse funzioni di elaborazione e analisi del segnale.
È fondamentale sottolineare che per poter generare un plugin Audio a partire da codice sorgente Matlab, questo dovrà seguire regole più stringenti e specifiche rispetto ad un qualunque script, infatti come già detto dovrà essere una sottoclasse di `audioPlugin` e non tutte le funzioni native di Matlab saranno disponibili. Una volta scritto e reso conforme alle specifiche, il codice sorgente potrà essere testato e da questo eventualmente si potrà generare il plugin desiderato.

2 Generazione di Plugin

Questa sezione amplia quanto accennato nel paragrafo 1.3, approfondendo sui vari paradigmi di implementazione e sul workflow alla base dello sviluppo di un generico plugin audio su Matlab.

2.1 Trasformazione di script Matlab in una classe Plugin Audio

Si illustra con un semplice esempio il processo per trasformare uno script Matlab che esegue operazioni su segnali audio in una sottoclasse di `audioPlugin` dalla quale possa essere generato un plugin VST. Supponiamo di voler trasformare il seguente programma, che applica in real time un semplice effetto Tremolo (Modulazione d'Ampiezza) ad un segnale in entrata, in un plugin audio:

```
%% 1 - Inizializzazione delle variabili
Rate = 1;

fileInfo = audioinfo('AmenBreak.wav');
sampleRate = fileInfo.SampleRate;
frameSize = 256;

%% 2 - Creazione degli Oggetti
Sine = audioOscillator('DCOffset',1,'SamplesPerFrame',frameSize,...
                       'Frequency',Rate,'SampleRate',sampleRate);
fileReader = dsp.AudioFileReader('Filename',fileInfo.Filename,'SamplesPerFrame',frameSize);
deviceWriter = audioDeviceWriter('SampleRate',fileReader.SampleRate);

%% 3 - Streaming Audio e elaborazione in tempo reale
while ~isDone(fileReader)
    in = fileReader(); % Lettura di un frame del segnale audio
    gain = Sine(); % Estrazione del frame del segnale modulante
    out = in .* gain; % elaborazione del segnale
    deviceWriter(out); % Scrittura del singolo frame di segnale
end
```

Il codice è suddiviso in tre sezioni, le quali saranno fondamentali nel processo di trasformazione in una classe:

1. Inizializzazione delle variabili necessarie
2. Costruzione degli Oggetti usati per l'elaborazione dell'input
3. elaborazione del segnale audio in tempo reale.

Si inizia creando la struttura del plugin:

```
classdef SimpleTremolo < audioPlugin
    %% 1 - Inizializzazione delle variabili
    properties
        % Variabili Glogali
    end
    properties(Access = private)
        % Variabili Private
    end
    properties (Constant)
        % Interfaccia Utente
    end
    methods
        %% 2 - Creazione degli Oggetti
        function plugin = SimpleTremolo()
            % Costruttore, in questa sezione saranno
            % istanziati tutti gli oggetti usati dal software
        end
        function reset(plugin)
            % funzione di Reset
        end
        %% 3 - Streaming Audio e elaborazione in tempo reale
        function output = process(plugin,input)
            % Funzione di elaborazione
        end
    end
end
```

2.1.1 Inizializzazione delle variabili

Si é visto che un plugin valido dev'essere sottoclasse di `audioPlugin`, come primo passo si definirá la classe come tale, utilizzando la keyword `classdef` e l'operatore `<` per ereditare attributi e metodi necessari.
La sezione di Inizializzazione delle variabili è stata suddivisa in tre blocchi di codice distinti, ognuno dei quali delimitato dalla keyword `properties`, che definisce l'inizio di un blocco di codice contenente variabili e dati appartenenti ad una singola istanza della classe.

1. Il primo blocco conterrà variabili e oggetti globali, cioè dati accessibili e modificabili dall'utente finale. Si deduce facilmente che nell' esempio questo blocco conterrà la sola variabile `Rate`, che quindi sarà l'unico parametro del nostro plugin modificabile dall'utente.

2. Dall'intestazione è evidente che il secondo blocco di `properties` è dedicato ad oggetti e variabili non accessibili direttamente. Infatti essi sono destinati all'uso interno, e ciò che viene dichiarato qui non sarà visibile all'utente. Nell' esempio, qui sará dichiarato l'oscillatore audio usato per modulare il segnale di ingresso.

3. Questo blocco non ha un corrispondente nello script di partenza, esso infatti è il blocco nel quale andremo a specificare l'aspetto dell'interfaccia utente e alcuni parametri non modificabili ma d'interesse per l'utente, come ad esempio il nome del plugin e il numero di canali di input e output. L'Attributo `(Constant)` è infatti utilizzato per specificare le proprietà di classe che saranno uguali per qualsiasi istanza della classe.

Il codice ottenuto per la prima sezione di codice è il seguente

```
properties % Variabili Glogali
    Rate = 1;
end
properties(Access = private) % Variabili Private
    Sine
end
properties (Constant) % Interfaccia Utente
    PluginInterface = audioPluginInterface(...
        audioPluginParameter('Rate','DisplayName','Rate',...
                             'Label','Hz','Mapping',{'lin',0.01,15}))
end
```

2.1.2 Costruzione degli Oggetti

La seconda sezione del codice è dedicata all'istanziazione di tutti gli oggetti necessari al funzionamento del software.
La prima keyword incontrata è `methods`. Questa, come suggerito dal nome, è usata per delimitare le sezioni di codice contenenti le funzioni e i metodi di cui il plugin ha bisogno per funzionare.
Le funzioni di maggiore interesse che si trovano in questa sezione sono due, anche se all'occorrenza se ne potranno aggiungere altre.

- **Costruttore :** La prima funzione da definire è il costruttore stesso.
  Questa funzione ha appunto il compito di costruire tutti gli oggetti necessari, che andranno istanziati nello stesso momento in cui viene istanziato il plugin stesso. In questa funzione saranno anche impostati i valori delle variabili non accessibili all'utente nel caso ne ce ne fosse bisogno. Nell'esempio il costruttore dovrà istanziare l'oscillatore audio che modulerà l'ampiezza del segnale in ingresso.

- **Funzione di Reset :** Serve a riportare alla condizioni iniziali lo stato interno del plugin.
  Questa funzione è chiamata ogni volta che viene avviata una nuova sessione, oppure ogni volta che viene cambiata la frequenza di campionamento dell'ambiente. Nell'esempio la funzione `reset` aggiornerà la frequenza di campionamento dell'oscillatore usando il metodo `getSampleRate(..)` ereditato dalla superclasse.

Segue il codice descritto sopra:

```
methods
    %% 2 - Creazione degli Oggetti
    function plugin = SimpleTremolo()
        plugin.Sine = audioOscillator('DCOffset',1);
    end
    function reset(plugin)
        plugin.Sine.SampleRate = getSampleRate(plugin);
    end
    % ... altre funzioni ...
end
```

2.1.3 Funzioni get e set

Benchè non necessarie per il funzionamento del plugin esaminato in questo esempio, è bene discutere la natura delle funzioni `get` e `set`, che tornano spesso utili nell'implementazione di plugin più complessi. Queste funzioni saranno chiamate automaticamente ogni volta che avviene un accesso alla proprietà ad esse associata, dando la possibilità di eseguire elaborazioni aggiuntive oltre al semplice aggiornamento di una variabile.

- Le funzioni `get` sono utilizzate per recuperare informazioni o dati contenuti in un oggetto del plugin senza volerle modificare. Nell'esempio si potrebbe utilizzare una funzione `get` per accedere alla frequenza di oscillazione del modulatore.

```
    %funzione get
    function value = get.Rate(plugin)
            value = obj.Sine.Frequency;
            % ...elaborazioni aggiuntive
    end
```

- Le funzioni `set` sono invece utilizzate quando si desidera modificare uno specifico parametro del plugin. Nell'esempio si utillizza una funzione `set` per modificare il valore della frequenza di oscillazione del modulatore quando questa viene modificata dall'utente tramite l'interfaccia grafica.

```
    % funzione set
    function set.Rate(plugin,value)
            plugin.Sine.Frequency = value;
            % ...elaborazioni aggiuntive
    end
```

Va sottolineato che nel caso in cui non siano necessarie operazioni aggiuntive durante ogni accesso ad una specifica variabile, l'aggiornamento avverrebbe automaticamente. Pertanto, implementare tali funzioni potrebbe risultare ridondante, ma comunque corretto.
Conciliando quanto è stato sviluppato nei due paragrafi precedenti si ottiene:

```
% ...
methods
    %% 2 - Creazione e gestione degli Oggetti
    function plugin = SimpleTremolo()
        plugin.Sine = audioOscillator('DCOffset',1);
    end
    function reset(plugin)
        plugin.Sine.SampleRate = getSampleRate(plugin);
    end
    function value = get.Rate(plugin)
        value = obj.Sine.Frequency;
    end
    function set.Rate(plugin,value)
        plugin.Sine.Frequency = value;
    end
    % ...
end
```

2.1.4 Funzione di elaborazione

Le funzione di elaborazione contiene l'algoritmo principale di elaborazione del segnale, basato su frame. Questo metodo è infatti invocato continuamente all'interno di un ciclo, e ad ogni iterazione elabora una porzione lunga `frameSize` del segnale. L'informazione racchiusa in questa variabile è di fondamentale importanza per il corretto funzionamento del nostro sistema, infatti essa indica la dimensione in numero di campioni del segnale da elaborare iterazione per iterazione. Ovviamente il `frameSize` deve essere lo stesso per tutti gli oggetti che contribuiscono all'elaborazione del segnale audio e quindi va gestito con gran cura, altrimenti non sarà possibile generare il plugin VST a partire dal codice.
Il contenuto della funzione di elaborazione può variare molto a seconda del tipo di plugin in analisi, può infatti contenere solo qualche semplice comando oppure essere molto complessa. È comunque di buona norma rendere questa funzione il più concisa ed efficiente possibile, in modo che sia veloce e che non introduca effetti di latenza, che potrebbero aggiungere al suono distorsioni indesiderate.
Questa funzione viene chiamata `process` e deve essere definita in un blocco di metodi pubblico.
La funzione di elaborazione ha generalmente due input, il primo dei quali è riservato per l'oggetto `audioPlugin`, mentre il segnale da elaborare solitamente è passato alla funzione tramite il secondo argomento, ed è una matrice ad una o due colonne[1] e `frameSize` righe.
Continuando con l'esempio proposto all'inizio del capitolo, procediamo trasformando la parte di codice che si trova all'interno del ciclo `while` nella funzione `process` del nostro plugin, creando quindi la sezione dedicata allo streaming audio e alla elaborazione in real time del segnale.
La prima differenza inconrata è nella gestione dell'input, infatti nelle classi `audioPlugin` i segnali in entrata e i segnali in uscita sono gestiti automaticamente, perciò non ci sará bisogno di utilizzare le funzioni `dsp.AudioFileReader` [8] e `audioDeviceWriter` [9] .
È giusto sottolineare che un plugin audio valido deve essere in grado di lavorare su frame di lunghezza variabile, quindi `frameSize` dovrà essere aggiornata ad ogni chiamata della funzione `process`, e di conseguenza dovranno essere aggiornati anche tutti gli oggetti che ne fanno uso.
Alla luce di questi accorgimenti, la funzione di elaborazione del plugin di tremolo sarà la seguente:

```
methods
    % ...
    %% 3 - Streaming Audio e elaborazione in Real time
    function output = process(plugin,input)
        frameSize = size(input,1);
        plugin.Sine.Frequency = plugin.Rate;
        plugin.Sine.SamplesPerFrame = frameSize;
        gain = step(plugin.Sine);
        output = input.*gain;
    end
end
```

Esaminiamo la funzione pezzo per pezzo:

- Si aggiorna la variabile `frameSize`, salvandoci l'attuale lunghezza del segnale in input.
- Si aggiorna la frequenza di oscillazione alla quale lavora il modulatore secondo il valore del parametro `Rate` scelto dall'utente.
- Viene generato il segnale modulante, salvato nel vettore chiamato `gain`, che indicherà il guadagno di ampiezza da applicare al segnale di ingresso campione per campione per ottenere l'effetto desiderato. Per ottenere questo vettore si utilizza il metodo `step`[2]
- La funzione conclude moltiplicando elemento per elemento segnale di ingresso e vettore modulante.

Con questa funzione la classe SimpleTremolo è completa, e si potrà passare alla fase di debug e test, per infine convertire il prodotto finale in formato VST da compatibile con la nostra DAW.

[1] una per plugin Mono e due per plugin Stereo, ma questo parametro è flessibile, si pensi ad esempio a plugin per l'elaborazione audio in Standard Dolby 5.1

[2] La funzione `step` è un metodo nativo di Matlab molto versatile, infatti il suo comportamento cambia a seconda dell'oggetto sul quale viene chiamato. Se ad esempio il metodo `step` venisse chiamato su un oggetto che rappresenta un sistema dinamico, questa ne restituirebbe la risposta al gradino. [10]
In questo contesto `step` viene chiamato su un oggetto che rappresenta un oscillatore sinusoidale, perciò restituirà un segnale sinusoidale le cui caratteristiche sono specificate nei parametri dell'oggetto `audioOscillator`.

Segue il codice completo della classe `audioPlugin` sviluppata per l'esempio, chiamata `SimpleTremolo`.

```
classdef SimpleTremolo < audioPlugin
    %% 1 - Inizializzazione delle variabili
    properties
        Rate = 1;
    end
    properties(Access = private)
        Sine
    end
    properties (Constant)
        PluginInterface = audioPluginInterface(...
            'PluginName','SimpleTremolo',...
            'InputChannels',2,'OutputChannels',2,...
            audioPluginParameter('Rate','DisplayName','Rate',...
                                 'Label','Hz','Mapping',{'lin',0.01,15}))
    end
    methods
        %% 2 - Creazione degli Oggetti
        function plugin = SimpleTremolo()
            plugin.Sine = audioOscillator('DCOffset',1);
        end
        function reset(plugin)
            plugin.Sine.SampleRate = getSampleRate(plugin);
        end
        %% 3 - Streaming Audio e elaborazione in Real time
        function output = process(plugin,input)
            frameSize = size(input,1);
            plugin.Sine.Frequency = plugin.Rate;
            plugin.Sine.SamplesPerFrame = frameSize;
            gain = step(plugin.Sine);
            output = input.*gain;
        end
    end
end
```

Si noti che le funzioni `get` e `set` sono state omesse, in quanto non necessarie, e nella dichiarazione dell'interfaccia utente è stato aggiunto il nome del plugin che sarà visualizzato dalla DAW, il numero di canali di input e output, il nome della casa distributrice e la versione del plugin.

2.2 Testing

L'ambiente di sviluppo Matlab mette a disposizione uno strumento per testare il funzionamento dei plugin durante il processo di sviluppo, chiamato `audioTestBench` [11]. Questo è uno strumento estremamente efficace che permette di interagire con l'interfaccia utente del plugin senza il bisogno di generare il corrispondente file `.vst`, e quindi dà allo sviluppatore la possibilità di testare tutte le funzionalità del plugin e di correggere eventuali errori o bug velocemente.
`audioTestBench`, oltre a fornire un ambiente di debug, permette di visualizzare l'elaborazione eseguita sul segnale audio sia nel dominio della frequenza che nel dominio del tempo, permette la sincronizzazione interattiva di controllori MIDI con i parametri del plugin, e permette infine di eseguire controlli di validazione del plugin e di generare i file binari, nel nostro caso i file `.vst`.
Si può accedere all'ambiente di testing tramite la `Command Window` di Matlab utilizzando il comando

```
>> audioTestBench <nome del plugin da testare>
```

che aprirà un'istanza del testbench contentente il plugin in esame. (vedi Figura 1)
Scrivendo solo `audioTestBench` si aprirà un'istanza vuota dalla quale sarà possibile scegliere il plugin o i plugin da testare attraverso l'interfaccia grafica.
Eseguendo il comando `audioTestBench SimpleTremolo` ci troveremo davanti alla seguente schermata:

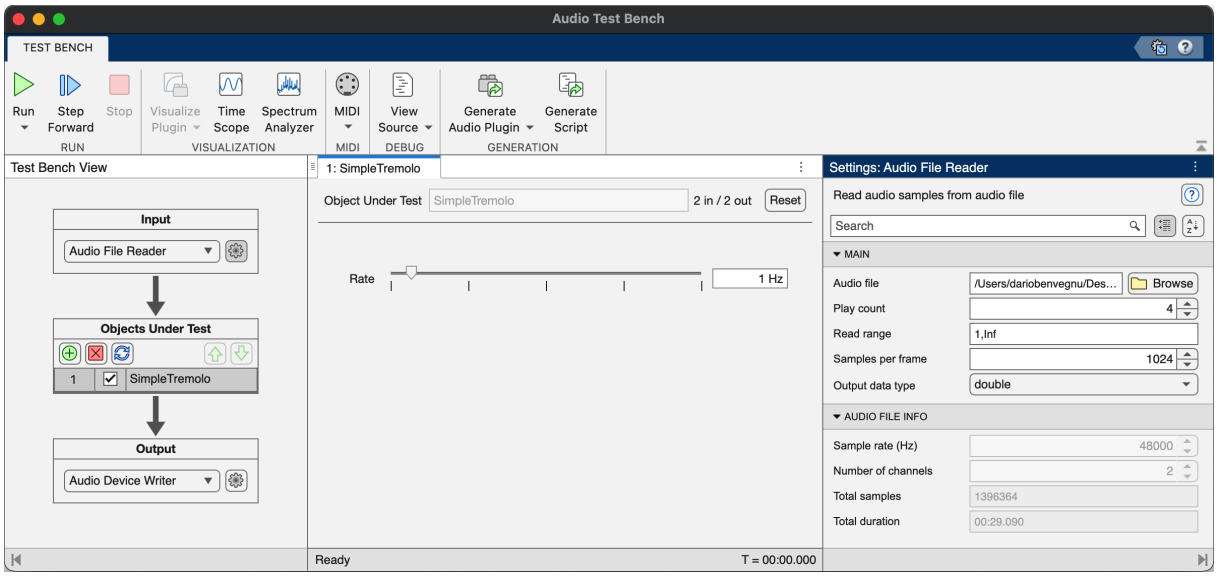


Figure 1: Audio Test Bench

Nell'area centrale della finestra si troverà l'interfaccia grafica del plugin, e da questa sarà possibile interagire con tutti i parametri, che si presentano sottoforma di sliders e knobs se controllano parametri numerici oppure sottoforma di menù a tendina e interruttori se controllano parametri logici. L'unico parametro modificabile dall'utente in `SimpleTremolo` è la frequenza del'oscillatore, perciò la sua interfaccia sarà composta da un unico slider collegato a questo parametro. Nella parte alla sinistra dell'interfaccia del plugin si trova un menù a schema a blocchi dal quale è possibile modificare le opzioni di input e output e aggiungere alla catena eventuali plugin aggiuntivi da testare. Le opzioni di input sono di particolare interesse in quanto permettono di modificare la sorgente di input e di generare segnali utili per testare il funzionamento del nostro processore audio, come impulsi di rumore o sweep in frequenza, ma permettono anche di cambiare importanti variabili interne come il numero di sample per frame e il tipo di dato generato in output.
La sezione superiore della finestra, chiamata *Toolstrip*, permette di avviare la simulazione premendo il tasto `Run`, ma anche di verificare il comportamento del plugin un frame alla volta premendo il tasto `Step Forward`. La riproduzione audio potrà essere bloccata con il tasto `Stop`.
Più a destra troviamo gli strumenti di analisi nel dominio del tempo e nel dominio della frequenza, seguiti dalla sezione usata per testare i plugin con compatibilità MIDI, la sezione di debug ed infine la sezione dedicata alla generazione del plugin stesso, che comprende un menù per scegliere le opzioni di generazione e dà la possibilità di generare uno script per testare automaticamente il plugin prima di convertirlo in un formato compatibile con le Digital Audio Workstations.

2.3 Validazione e Generazione di Plugin

Le operazioni di Validazione e Generazione sono gli ultimi due passi nello sviluppo di un plugin audio con Matlab. Entrambe le operazioni posso essere eseguite all'interno di `audioTestBench`, ma possono anche essere invocate dalla `Command Window` con due appositi comandi.

2.3.1 Validazione

Il processo di validazione serve a testare automaticamente la classe plugin in esame, ed è pensato per trovare gli errori di programmazione che più comunemente vengono commessi durante lo sviluppo. La funzione designata a questo compito è chiamata `validateAudioPlugin` [12], segue la sintassi:

```
>> validateAudioPlugin -<opzioni...> <nome del plugin da validare>
```

Di seguito, vengono illustrate passo dopo passo le operazioni eseguite da questa funzione:

- Esegue un sottoinsieme dei controlli che saranno poi eseguiti nella fase di generazione
- Crea e ed esegue un test bench[3] Matlab per mettere alla prova la classe
- Crea ed esegue una versione MEX[4] del testbench.
- Se la validazione si conclude con successo, elimina i test bench generati.

Il processo di validazione è molto utile durante lo sviluppo di un plugin perchè spesso permette di trovare e correggere errori o problemi che potrebbero sfuggire utilizzando solamente `audioTestBench`. Infatti nel caso in cui una validazione non riesca con successo, i file generati da `validateAudioPlugin` non vengono eliminati, e possono quindi essere utilizzati per il debugging approfondito della classe.
Il `testbench` assicura che il plugin risponda correttamente al cambiamento delle variabili di ambiente, fondamentali al funzionamento, perciò esegue molti test consecutivi, variando la frequenza di campionamento utilizzata e la lunghezza del frame audio. Inoltre, itera controllando tutte le possibili combinazioni assumibili dai parametri modificabili dall'utente, verificando che nessuna di esse porti il plugin in stati problematici o che vada ad intaccare le variabili d'ambiente.
Quando la validazione termina correttamente, il plugin è pronto per la generazione e sulla `Command Window` apparirà un messaggio simile al seguente, che illustra la procedura descritta in precedenza

```
>> validateAudioPlugin SimpleTremolo

Checking plugin class 'SimpleTremolo'... passed.
Generating testbench file 'testbench_SimpleTremolo.m'... done.
Running testbench... passed.
Generating mex file 'testbench_SimpleTremolo_mex.mexmaci64'... done.
Running mex testbench... passed.
Deleting testbench.
Ready to generate audio plugin.
```

2.3.2 Generazione

Completato lo sviluppo del plugin, è ora possibile generarlo utilizzando la funzione `generateAudioPlugin` [13] mediante il seguente comando:

```
>> generateAudioPlugin -<opzioni...> <nome del plugin da generare>
```

Questa funzione, dopo aver eseguito dei controlli simili a quelli visti nel processo di validazione, crea il file binario contenente il plugin, che di default sarà in formato `.vst`.
`generateAudioPlugin` è in grado di esportare i plugin in diversi formati, quali `.vst3` (virtual studio technology di terza generazione), `.AU`, `.AUv3`, `.exe` o come progetto `C++` compatibile con `JUCE`.
Aprendo la finestra di generazione dalla sezione apposita in `audioTestBench` si potrà comodamente invocare la validazione e la generazione di un plugin tramite la semplice interfaccia grafica, dalla quale si potranno scegliere tutte le opzioni di output delle due funzioni appena viste.
Si rimanda alla documentazione di Matlab per conoscere tutte le diverse opzioni di validazione e generazione di plugin audio. [12] [13]

Nota : ogni plugin audio presentato in questo studio è stato sviluppato con la versione `R2024_a` di Matlab. È stata usata questa versione in prerelease poichè le precedenti `R2024_a` e `R2024_b` non permettevano la generazione del plugin a causa di un bug.

[3] un test bench Matlab è unp script progettato per verificare il corretto funzionamento di una funzione o un algoritmo sviluppato in Matlab, eseguendo più volte la funzione variando l'input e simulando condizioni reali.

[4] MEX è l'acronimo di Matlab EXecutable. Questi file permettono di integrare il codice Matlab con codice scritto in C/C++, migliorando le prestazioni delle parti di codice computazionalmente intensive di un plugin audio

2.4 Importazione e Utilizzo su DAW, Risultati

Una volta generato il plugin `SimpleTremolo` potrà essere installato su una qualsiasi DAW che supporti plugin in formato `.vst`, ed utilizzato come effetto audio.
Nel blocco di codice e nelle figure 2 e 3 si mostrano validazione, generazione, installazione e utilizzo di `SimpleTremolo` nella Digital Audio Workstation FL Studio

```
>> validateAudioPlugin SimpleTremolo

Checking plugin class 'SimpleTremolo'... passed.
Generating testbench file 'testbench_SimpleTremolo.m'... done.
Running testbench... passed.
Generating mex file 'testbench_SimpleTremolo_mex.mexmaci64'... done.
Running mex testbench... passed.
Deleting testbench.
Ready to generate audio plugin.

>> generateAudioPlugin SimpleTremolo;
........
>> generateAudioPlugin -vst3 SimpleTremolo;
........
>>
```

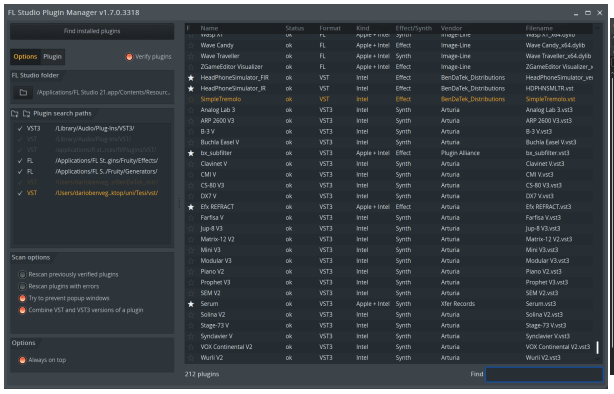


Figure 2: Installazione di SimpleTremolo

Figure 3: Utilizzo di SimpleTremolo su Fl Studio

Si mostra nella figura 4 l'effetto di `SimpleTremolo` su un segnale audio nel dominio del tempo:

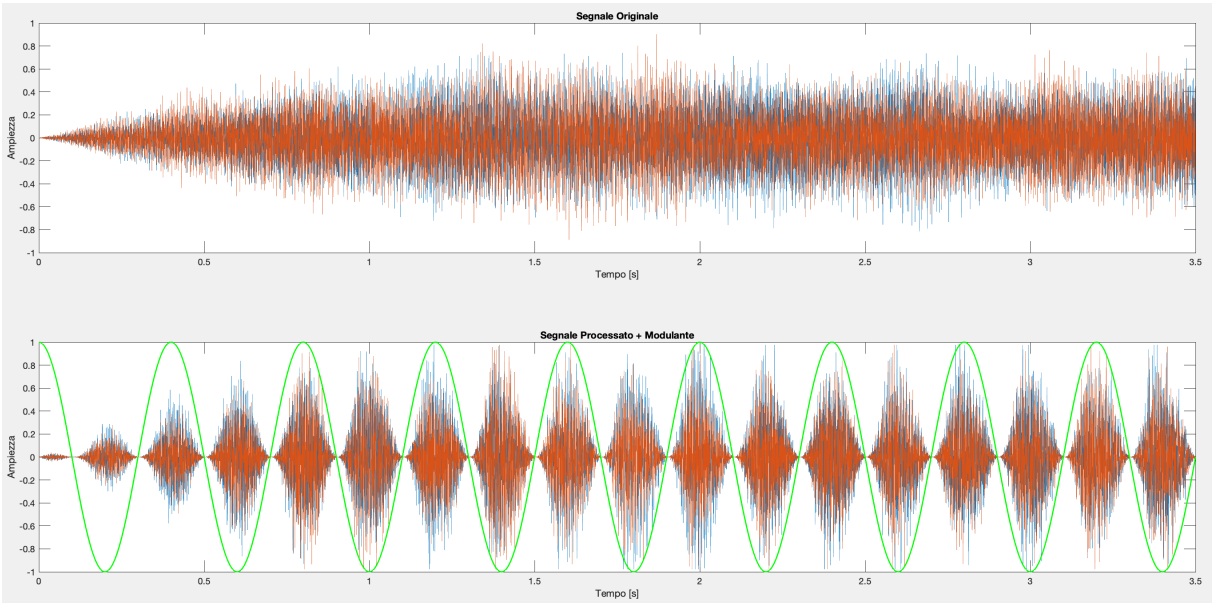


Figure 4: Effetto di SimpleTremolo su un segnale audio

3 Simulazione di Cuffie Hi-Fi

L'idea alla base dello studio della Dottoressa Zuccante svolto nella sua Tesi di Laurea Magistrale [14] è di riuscire a replicare il suono di un determinato modello di cuffie, definite Target, attraverso un altro paio di cuffie, definite Monitor, in modo tale da ottenere nel complesso la stessa esperienza di ascolto che sarebbe data dalle sole cuffie Target.
Questo progetto è portato avanti da studenti, dottorandi, ricercatori e professori attivi nel laboratorio CSC[5], parte del Dipartimento di Ingegneria dell'Informazione dell'Università di Padova, in collaborazione con l'azienda Spin Off *Audio Innova*, fondata nel 2013. Il progetto è nato dalla richiesta di un cliente di alto profilo appartenente alla comunità degli audiofili, che ha idealizzato la creazione di un'applicazione commerciale per la simulazione di cuffie ad alta fedeltà.
Con *simulazione* si intende la capacità di replicare il suono di un paio di cuffie attraverso un altro paio di cuffie, in modo tale che l'utente finale possa confrontare il suono di modelli di cuffie diversi da quelli indossati. Ovviamente non sarà possibile simulare le caratteristiche fisiche delle cuffie in questione, come peso e comodità, ma con questo progetto si vuole fornire ai professionisti dell'audio uno strumento che gli permetta di eseguire veloci comparazioni del proprio lavoro su dispositivi con caratteristiche diverse, e di dare ad ascoltatori esperti un software capace di guidarli nell'acquisto di un nuovo modello di cuffie. Idealmente si desidera creare un ambiente nel quale l'ascoltatore è in grado di riprodurre la stessa esperienza di ascolto che avrebbe indossando un paio di cuffie diverse da quelle utilizzate, ma l'utente finale dovrà essere conscio del fatto che questo prodotto non offrirà un sostituto perfetto delle cuffie desiderate ma bensì una buona approssimazione dell'esperienza di ascolto data da queste ultime.

La simulazione è basata sulla teoria del *Digital Signal Processing* [15], in particolare si fa uso delle risposte impulsive (e di conseguenza delle risposte in frequenza) delle cuffie monitor e delle cuffie target, si sfruttano le proprietà dei sistemi LTI e la teoria della DFT (*Discrete Fourier Transform*).
Il software sviluppato in precedenza permette di usare risposte in frequenza estratte da due diversi database, *Crinacle* e *Oratory1990*. Per semplicità e convenienza, il plugin che si andrà a sviluppare in questa tesi farà utilizzo unicamente del secondo, dato che, come sottolineato dalla Dott.sa Anna in [14], le misurazioni che si trovano in questo database oltre a essere tra le migliori in circolazione hanno bisogno di meno preprocessing rispetto alle misurazioni trovate in *Crinacle*.
Le misurazioni di *Oratory1990* sono state effettuate usando un manichino, o *dummy head*, standard del settore con simulatori di orecchio e un cuscinetto antropometrico. Un altro fondamentale vantaggio dato da questo database è l'approccio utilizzato per ottenere le misurazioni: la risposta impulsiva è ricavata usando un segnale di tipo *Sine Sweep*[6] esponenziale [16][17], e questo ci permette di separare la componente lineare del sistema dalla componente non lineare, dando la possibilità di modellizzare il sistema come *Lineare e Tempo Invariante (LTI)*. Nella figura 5 si possono osservare lo spettrogramma e la rappresentazione nel dominio del tempo di un Sine Sweep.

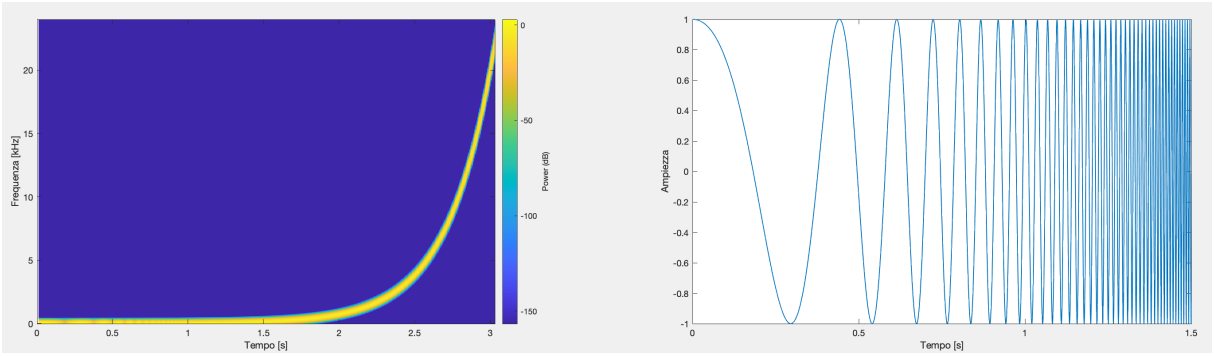
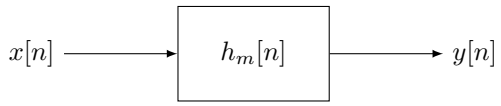


Figure 5: Spettrogramma e Rappresentazione Temporale di un Sine Sweep esponenziale

[5] Il Centro di Sonologia Computazionale, fondato nel 1979 da Giovanni Battista Debiasi, è un laboratorio di ricerca dell'Università di Padova, che si occupa di applicazioni informatico/computazionali nel campo della musica e dell'audio.
[6] Un *Sine Sweep* è un segnale sinusoidale a frequenza variabile, che per applicazioni audio parte da 20 Hz e arriva a 20'000 Hz. Questi vengono usati come toni di riferimento per verificare la risposta in frequenza di un sistema

3.1 Approccio Matematico

L'idea alla base del funzionamento del simulatore prevede l'applicazione di un filtro che contemporaneamente elimina il contributo della cuffia monitor e aggiunge quello della cuffia target, in modo da ottenere la stessa esperienza di ascolto che si avrebbe semplicemente indossando le cuffie target.
Avendo verificato la validità di un modello LTI, è teoricamente possibile simulare le caratteristiche di ascolto di una qualsiasi cuffia semplicemente replicandone la risposta in frequenza.
Generalmente per le cuffie Monitor con IR $h_m[n]$ si ha il seguente schema a blocchi:



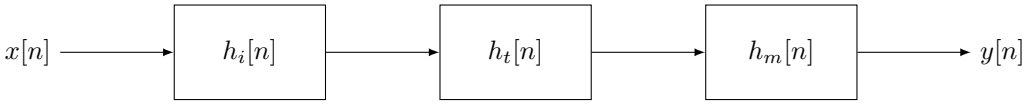
Per ricreare l'esperienza di ascolto data dalla cuffie Target vanno definite alcune variabili:

- $X(f)$: trasformata di Fourier Discreta (DFT) del segnale d'ingresso $x[n]$
- $Y(f)$: DFT del segnale d'uscita $y[n]$
- $H_m(f) = \mathcal{F}[h_m[n]](f)$: risposta in frequenza delle cuffie Monitor, $H_i(f) = \frac{1}{H_m(f)}$ la sua inversa
- $H_t(f) = \mathcal{F}[h_t[n]](f)$: risposta in frequenza delle cuffie Target
- $H_s(f) = \frac{H_t(f)}{H_m(f)}$: combinazione tra $H_t(f)$ e $H_m(f)$

Il segnale d'uscita del nostro sistema verrà calcolato come segue:

$$Y(f) = H_s(f) \cdot H_m(f) \cdot X(f) = \frac{H_t(f)}{H_m(f)} \cdot H_m(f) \cdot X(f) = H_t(f) \cdot X(f) \tag{1}$$

Lo schema a blocchi finale sarà il seguente



Si implementerà quest'idea in 4 step:

- Estrazione della risposta impulsiva di cuffie Monitor e Cuffie Target.
- Estrazione di $h_i[n]$ dalla IR delle cuffie Monitor
- Creazione di un filtro combinando le risposte Impulsive
- Filtraggio del segnale d'ingresso

3.2 Implementazione

Per dare un'idea generale della elaborazione alla base del simulatore, in questa sezione si presenta l'approccio con il quale sono state implementate le precedenti versioni del software. Il dataset *Oratory1990* contiene le misurazioni delle risposte in frequenza sottoforma di vettori: essi contengono i valori in scala SPL[7] e i corrispondenti valori in frequenza.
Inoltre, le risposte in frequenza di ogni cuffia sono ottenute mediando su più misurazioni, in modo da ridurre il più possibile differenze dovute a posizionamenti diversi della cuffia tra misurazione e misurazione, e l'eventuale influenza del rumore ambientale.
Di seguito il vettore viene interpolato per aggiungere dei valori resi necessari dalla teoria delle Trasformate di Fourier che nelle risposte in frequenza fornite dal dataset non sono presenti. Questa operazione non influenza il risultato sonoro finale dato che l'interpolazione aggiunge valori al di fuori del range uditivo dell'orecchio umano (che va da 20 Hz e 20 kHz). Dato che i dati forniti in *Oratory1990* sono di tipo *raw*[8] è necessario applicare delle curve di compensazione. Queste curve sono progettate per modificare il segnale audio al fine di adattarsi alle preferenze degli ascoltatori, correggendo le variazioni in frequenza che possono essere introdotte dal dispositivo di riproduzione in uso e considerando come l'orecchio umano percepisce in modo differente le varie frequenze a diversi livelli di volume d'ascolto.
Per i plugin audio sviluppati in questa tesi è stata scelta la curva di compensazione Harman più recente, cioè la *Harman OE 2018*, rappresentata di colore verde chiaro nella Figura 6.

[7] Sound Pressure Level: il livello di pressione sonora è la misura in dB della deviazione dalla pressione ambientale dell'aria provocata da un'onda sonora
[8] ottenute tramite semplici misurazioni

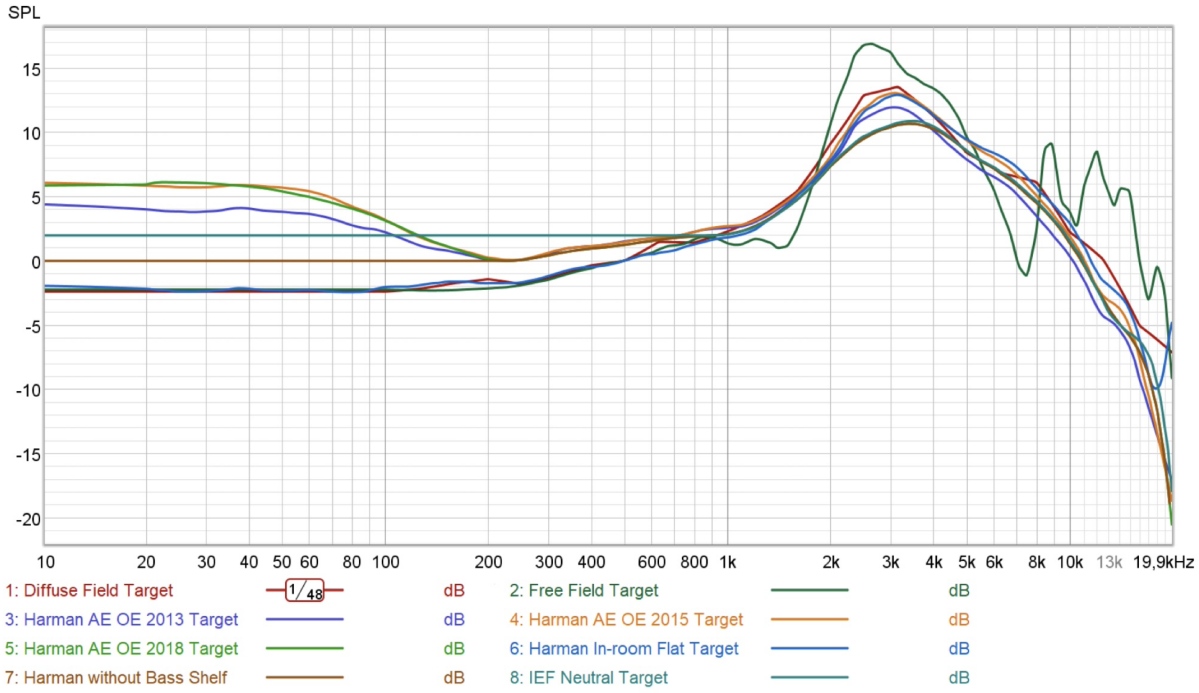


Figure 6: Curve di compensazione (OE: curve per cuffie 'Over Ear', IE: In Ear)

Prima di applicare questa curva è necessario allineare la risposta in frequenza con la curva di compensazione, perciò anche quest'ultima sarà interpolata.
Una volta compensata la risposta in frequenza si procede sottocampionandola a 48kHz e si applica una conversione a 24 bit, per poi applicare diversi tipi di *smoothing*. L'obbiettivo dello smoothing è ridurre gli effetti del rumore sulla risposta impulsiva, preservando le caratteristiche temporali rilevanti dal punto di vista percettivo della risposta originale. Si usano due tipi diversi di smoothing :

- **Smoothing Medio** : L'idea è semplicemente di fare la media di un insieme di valori in modo ricorsivo. Nonostante la sua semplicità, è ottimo per ridurre il rumore casuale.
- **Smoothing frazionario su bande d'ottava** : Questo è un tipo di smoothing complesso spesso applicato nell'elaborazione audio. Uno dei motivi principali per il quale si utilizza è perché è conforme dal punto di vista percettivo poiché segue la risoluzione in frequenza dell'orecchio, con una risoluzione fine a basse frequenze e una risoluzione più grossolana ad alte frequenze.

Infine, dopo aver convertito i valori dalla scala SPL alla scala lineare, l'ultimo passaggio prevede la derivazione dell'IR calcolando la trasformata inversa tramite l'algoritmo di iFFT. Inoltre, è stata implementata anche un'operazione di normalizzazione per recuperare l'IR dal suo valore massimo.
Nel software di partenza la risposta impulsiva generata viene infine salvata su disco in formato *.wav*.
Per quanto concerne la trattazione in questa tesi, quelle appena descritte sono le uniche operazioni d'interesse eseguite nel software `hpSimulOratory.py`, in quanto una volta ottenuta la risposta impulsiva desiderata, il plugin audio potrá semplicemente eseguire una convoluzione tra essa ed il segnale di ingresso. La figura 7 mostra lo schema a blocchi usato per l'estrazione della risposta impulsiva desiderata.

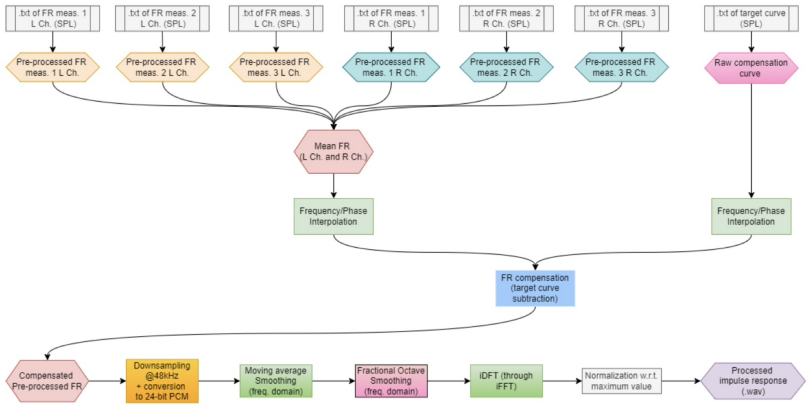


Figure 7: Schema generale per l'estrazione delle risposte impulsive

3.3 Dispositivi Simulati

Durante lo sviluppo sono stati scelti 10 diversi modelli di cuffie da simulare, accomunati da due fondamentali caratteristiche: tutte le cuffie utilizzate sono di tipo circumaurale[9] e con driver dinamici[10] [18]. Nelle figure 8 e 9 sono illustrati un esempio di cuffie circumaurali (*Beyerdynamic DT770*) e uno schema generale di cuffie circumaurali a driver dinamici. La scelta di questa tipologia di cuffie è stata influenzata dalle preferenze degli audiofili [19], dato che questo tipo di cuffie imita più fedelmente l'esperienza di ascolto offerta da una coppia di altoparlanti. Il nostro cervello infatti è abituato a percepire un suono come naturale quando esso, proveniente da un altoparlante, viene riflesso sul busto, sulla testa e sulle orecchie prima di raggiungere il timpano, causando un'alterazione. Le cuffie, invece, attraverso il loro accoppiamento con l'orecchio, eludono questo fenomeno rendendo il suono percepito meno naturale.
Tuttavia le cuffie circumaurali, grazie al fatto che avvolgono l'orecchio senza toccarlo, sono il sistema di cuffie più simile agli altoparlanti.
Inoltre le cuffie dotate di driver dinamici hanno una maggiore e più efficiente copertura delle frequenze udibili, e oltretutto sono tra le più economiche da realizzare.
Nella tabella 1 si elencano tutte le cuffie utilizzate, indicando quali modelli possono essere usati come Monitor e la categoria di ogni modello. Queste categorie sono:

- **Cuffie Aperte**: Le cuffie *Open-Back* sono dotate di padiglioni aperti, che permettono al suono dell'ambiente circostante di entrare. Questo risulta in un suono più naturale, ma impedisce alle cuffie di avere buone proprietà di isolamento acustico. Sono ideali per l'ascolto in ambienti poco rumorosi, e sono le più utilizzate per l'ascolto critico.
- **Cuffie Chiuse**: Le cuffie *Closed-Back* sono invece dotate di padiglioni completamente sigillati. Offrono quindi un migliore isolamento acustico ma intaccano leggermente la qualità del suono per via delle risonanze che si creano all'interno della camera acustica che viene creata, tipicamente aumentando la potenza delle basse frequenze. Sono ideali per ambienti più rumorosi.
- **Cuffie Semi-Aperte**: Sono un compromesso tra le due categorie precedenti, offrendo un migliore isolamento acustico rispetto a cuffie aperte, e sono migliori delle cuffie chiuse per sessioni di ascolto prolungate in quanto la capacità di scambiare un po' di aria con l'esterno le rende meno propense a creare affaticamento da ascolto.

Modello di Cuffie	Monitor	Target	Tipologia
AKG K240 MKII	✓	✓	Semi Aperte
AKG K701	x	✓	Aperte
Beyerdynamic DT770 M 80Ω	x	✓	Chiuse
Beyerdynamic DT990 M 250Ω	✓	✓	Aperte
Beyerdynamic T1	✓	✓	Aperte
Focal Stellia	x	✓	Chiuse
Sennheiser HD600	✓	✓	Chiuse
Sennheiser HD650	x	✓	Aperte
Sennheiser HD800s	x	✓	Aperte
Shure SRH1540	x	✓	Chiuse

Table 1: Modelli di cuffie Monitor e Target

Figure 8: Beyerdynamic DT770

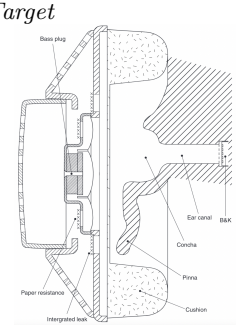


Figure 9: Sezione schematizzata di cuffie Circumaurali con driver dinamici

[9] Le cuffie di questo tipo circondano e coprono completamente il padiglione auricolare
[10] In questo contesto *driver* si può tradurre come *elemento motore*, e si riferisce al trasduttore elettroacustico all'interno delle cuffie che trasforma segnali elettrici in suono. I driver dinamici sono costituiti da tre elementi principali: un magnete permanente, una bobina avvolta attorna ad esso ed un diaframma collegato meccanicamente al magnete.

4 Prima Implementazione

Il primo approccio proposto per tradurre il software di simulazione di cuffie in plugin audio impiega diverse catene di filtri digitali FIR , o *Finite Impulse Response*. Questi sono filtri strettamente digitali caratterizzati da una risposta implusiva che va a zero in un intervallo finito di tempo.
In questa versione del plugin le risposte in frequenza sono state ricavate direttamente dai dati forniti da *Oratory1990*, usando le curve di equalizzazione e le tabelle trovate nei file messi a disposizione in [20]

4.1 Filtri

Si possono costruire filtri con la risposta in frequenza desiderata combinando le risposte in frequenza di filtri più semplici, che sono di tre tipologie : **Low Shelf**, **High Shelf** e **Peaking**. I singoli filtri vengono implementati digitalmente come *filtri biquadratici* [21] [22], filtri lineari del secondo ordine caratterizzati da una funzione di trasferimento esprimibile come rapporto tra due polinomi del secondo ordine.
Generalmente si avrà :

$$H(z) = \frac{b_0 + b_1 z^{-1} + b_2 z^{-2}}{a_0 + a_1 z^{-1} + a_2 z^{-2}} \tag{2}$$

dove:

- b_0, b_1 e b_2 sono i coefficienti del numeratore, che determinano la risposta in frequenza del filtro.
- a_0, a_1 e a_2 sono i coefficienti del denominatore, che determinano la stabilità e la natura del filtro.
- z è la variabile complessa della trasformata Z.

Per studiare più nel dettaglio le funzioni di trasferimento dei filtri usati conviene definire alcuni parametri:

- $\mathrm{A} = 10^{\frac{dBgain}{40}}$ dove *dBgain* è il guadagno del filtro espresso in scala logaritmica.
- $\omega_0 = 2\pi \cdot \frac{f_0}{Fs}$ dove f_0 è la frequenza di taglio e Fs la frequenza di campionamento in uso.
- $\alpha = \frac{\sin\omega_0}{2Q}$ dove Q (*Quality Factor*) è il parametro che indica la larghezza della banda di frequenze che il filtro influenza. Filtri con valori alti di Q agiscono su una gamma di frequenze più stretta, filtri con valori bassi agiscono su una gamma di frequenze più ampia.

4.1.1 Filtri Low Shelf

I Filtri Low Shelf sono progettati per attenuare o amplificare l'ampiezza del segnale in ingresso solamente al di sotto di una determinata frequenza. L'equazione (3) mostra la funzione di trasferimento di un filtro Low Shelf, la figura 10 mostra la sua risposta in frequenza.

$$H(s) = A\frac{s^2 + \frac{\sqrt{A}}{Q}s + A}{As^2 + \frac{\sqrt{A}}{Q}s + 1} \tag{3}$$

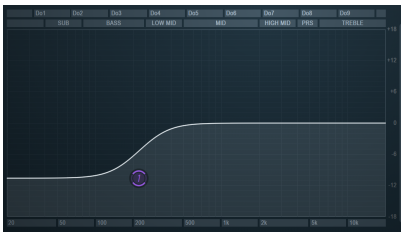


Figure 10: Low Shelf

Dove:

- $b_0 = A\left((A+1) - (A-1)\cos\omega_0 + 2\sqrt{A}\alpha\right)$
- $b_1 = 2A\left((A-1) - (A+1)\cos\omega_0\right)$
- $b_2 = A\left((A+1) - (A-1)\cos\omega_0 - 2\sqrt{A}\alpha\right)$
- $a_0 = (A+1) + (A-1)\cos\omega_0 + 2\sqrt{A}\alpha$
- $a_1 = -2\left((A-1) + (A+1)\cos\omega_0\right)$
- $a_2 = (A+1) + (A-1)\cos\omega_0 - 2\sqrt{A}\alpha$

Nel codice i filtri Low shelf vengono creati tramite i coefficienti usando la seguente funzione :

```
% Creazione di filtri Low Shelf
function [a,b] = biquad_LS_filterCoefficients(Fs,f0,dBGain,Q)
A = 10^(dBGain/40);
w0 = (2*pi*f0) /Fs;
alpha = sin(w0) / (2*Q);
a = [ (A+1) + (A-1)*cos(w0) + 2*alpha*sqrt(A),...
      -2*( (A-1) + (A+1)*cos(w0)),...
      (A+1) + (A-1)*cos(w0) - 2*alpha*sqrt(A)];
b =[ A*( (A+1) - (A-1)*cos(w0) + 2*alpha*sqrt(A)),...
     2*A*( (A-1) - (A+1)*cos(w0) ),...
     A*( (A+1) - (A-1)*cos(w0) - 2*alpha*sqrt(A))];
end
```

4.1.2 Filtri High Shelf

I filtri High Shelf possono essere visti come i complementari dei precedenti, infatti essi amplificano o attenuano l'ampiezza di un segnale solo al di sopra di una determinata frequenza. L'equazione (4) mostra la tipica funzione di trasferimento di un filtro High Shelf, la figura 11 mostra la sua tipica risposta in frequenza.

$$H(s) = A\frac{s^2 + \frac{\sqrt{A}}{Q}s + 1}{s^2 + \frac{\sqrt{A}}{Q}s + A} \tag{4}$$

Figure 11: High Shelf

Dove :

- $b_0 = A\left((A+1) + (A-1)\cos\omega_0 + 2\sqrt{A}\alpha\right)$
- $b_1 = -2A\left((A-1) + (A+1)\cos\omega_0\right)$
- $b_2 = A\left((A+1) + (A-1)\cos\omega_0 - 2\sqrt{A}\alpha\right)$
- $a_0 = A\left((A+1) + (A-1)\cos\omega_0 - 2\sqrt{A}\alpha\right)$
- $a_1 = 2\left((A-1) - (A+1)\cos\omega_0\right)$
- $a_2 = (A+1) - (A-1)\cos\omega_0 - 2\sqrt{A}\alpha$

Analogamente a quanto visto sopra, i filtri High Shelf vengono creati tramite un'apposita funzione :

```
% Creazione di Filtri High Shelf
function [a,b] = biquad_HS_filterCoefficients(Fs,f0,dBGain,Q)
A = 10^(dBGain/40);
w0 = (2*pi*f0) / Fs;
alpha = sin(w0) / (2*Q);
a = [ (A+1) - (A-1)*cos(w0) + 2*alpha*sqrt(A),...
      2*( (A-1) - (A+1)*cos(w0)),...
      (A+1) - (A-1)*cos(w0) - 2*alpha*sqrt(A)];
b = [ A*((A+1)+(A-1)*cos(w0)+2*alpha*sqrt(A)),...
      -2*A*((A-1)+(A+1)*cos(w0)),...
      A*( (A+1)+ (A-1)*cos(w0) - 2*alpha*sqrt(A))];
end
```

4.1.3 Filtri Peaking

A differenza dei filtri definiti in precedenza, questi amplificano o attenuano l'ampiezza di un segnale solo attorno ad una determinata frequenza centrale. L'equazione (5) mostra la tipica funzione di trasferimento di un filtro Peaking, la figura 12 mostra la sua tipica risposta in frequenza.

$$H(s) = \frac{s^2 + s\frac{A}{Q} + 1}{s^2 + \frac{s}{AQ} + 1} \tag{5}$$

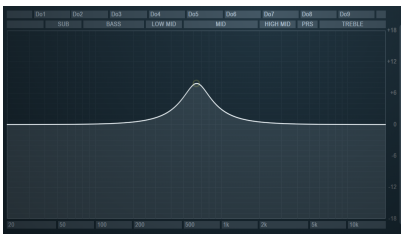


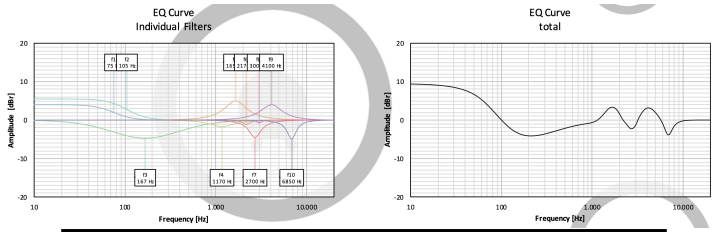
Figure 12: Peaking

- $b_0 = 1 + \alpha A$
- $b_1 = -2\cos(\omega_0)$
- $b_2 = 1 - \alpha A$
- $a_0 = 1 + \frac{\alpha}{A}$
- $a_1 = -2\cos(\omega_0)$
- $a_2 = 1 - \frac{\alpha}{A}$

Nel codice si creano i filtri necessari come segue :

```
% Creazione di filtri Peaking
function [a,b] = biquad_PK_filterCoefficients(Fs,f0,dBGain,Q)
A = 10^(dBGain/40);
w0 = (2*pi*f0)/Fs;
alpha = sin(w0)/(2*Q);
b = [ 1 + alpha*A, -2*cos(w0), 1 - alpha*A];
a = [ 1 + alpha/A, -2*cos(w0), 1 - alpha/A] ;
end
```

Nella figura 13 si mostrano a scopo esemplificativo curve e tabelle relative alle cuffie AKG K240 MKII.



Filter Settings

	Filter Type	Frequency	Gain	Q-Factor	BW
Band 1	LOW_SHELF	75 Hz	4,0 dB	0,71	1,89
Band 2	LOW_SHELF	105 Hz	5,5 dB	0,71	1,89
Band 3	PEAK	167 Hz	-4,7 dB	0,4	3,02
Band 4	PEAK	1170 Hz	-1,7 dB	1,3	1,08
Band 5	PEAK	1650 Hz	5,0 dB	1,4	1,01
Band 6	PEAK	2170 Hz	-1,1 dB	4,0	0,36
Band 7	PEAK	2700 Hz	-4,5 dB	2,5	0,57
Band 8	PEAK	3000 Hz	-0,7 dB	6,0	0,24
Band 9	PEAK	4100 Hz	4,0 dB	1,4	1,01
Band 10	PEAK	6850 Hz	-4,9 dB	3,0	0,48

Figure 13: Impostazioni di Equalizzazione per le cuffie AKG K240 MKII.

Nel codice ogni filtro è rappresentato con un tipo di dato strutturato composto da tre variabili:
La matrice 2x2 `w` indica lo stato del filtro (ovvero le sue *condizioni iniziali*), mentre i vettori `a` e `b` contengono, rispettivamente, i coefficienti del denominatore e del numeratore del filtro.

```
%inizializzazione di un generico filtro
filt = struct('w',[ 0,0 ; 0,0 ],'a',[1,0,0],'b',[1,0,0]);
%...
% Esempio di creazione di filtri Low Shelf, Peaking, High Shelf
[filtLS.a,filtLS.b] = biquad_LS_filterCoefficients(Fs, 69, 4.95 , 1.02);
[filtPK.a,filtPK.b] = biquad_PK_filterCoefficients(Fs, 420, 2.00 , 0.98);
[filtHS.a,filtHS.b] = biquad_HS_filterCoefficients(Fs, 2503, 1 , 9.98 );
```

Nella parte di codice dedicata all'elaborazione audio, ovvero nella funzione `process` del nostro plugin, c'è bisogno di un metodo che dato in input un segnale ed un filtro restituisca in output il segnale d'ingresso filtrato. Questo metodo risulta molto semplice se si sfrutta la funzione `filter` di Matlab:

```
function [y,w] = processBiquad(x, filt, ch)
    [y,w] = filter(filt.a, filt.b, x, filt.w(:,ch));
end
```

4.2 Struttura del codice

Nel plugin `HeadPhoneSimulator` i parametri fondamentali accessibili all'utente sono: la risposta delle cuffie Monitor, che sarà impostata secondo il modello di cuffie indossato al momento dell'ascolto, e la risposta delle cuffie Target, che sarà scelta a seconda di quale modello di cuffie si desidera simulare. Si potranno modificare questi parametri dall'interfaccia utente interagendo con due diversi menù a tendina.
In questa implementazione si è scelto di dare un'ulteriore possibilità all'utente, aggiungendo l'opzione di applicare o meno la curva di compensazione *Harman OE 2018*. In questo caso si potrà accedere a questo parametro tramite un semplice pulsante on-off.
Si è deciso inoltre di aggiungere al plugin un ulteriore parametro, chiamato *bypass*, anch'esso modificabile tramite un pulsante. Questo dà all'utente la possibilità di *"accendere e spegnere"* la simulazione dinamicamente, in modo da poter comparare velocemente il segnale filtrato con l'originale.
In aggiunta ai metodi fondamentali per un plugin audio presentati nel capitolo 2, `HeadPhoneSimulator` utilizza tre funzioni aggiuntive per aggiornare dinamicamente la catena di filtri ogni volta che l'utente interagisce con uno dei parametri elencati sopra. Queste sono:

- `updateMonitor`: chiamata dalla funzione `set` associata al parametro `Monitor`, questa funzione aggiorna i filtri usati per compensare la risposta in frequenza delle cuffie indossate dall'ascoltatore.
- `updateTarget`: chiamata dalla funzione `set` associata al parametro `Target`, questa funzione aggiorna i filtri usati per applicare al segnale audio la risposta in frequenza delle cuffie che l'utente desidera simulare.
- `updateCompensationCurve`: chiamata dalla funzione `set` associata al parametro `Compensation`, questa funzione aggiorna i filtri usati per applicare al segnale audio la curva di compensazione.

4.3 Valutazione delle Prestazioni

Dopo aver generato il plugin audio se ne possono valutare le prestazioni in relazione al software sviluppato dalla Dott.ssa Zuccante a partire da un segnale di test. Si andrà a processare questo segnale separatamente con `HeadPhoneSimulator` e con `hpSimulatorOratory.py`, per poi confrontare i rispettivi *power spectrum*[11] dei diversi segnali di output.
Per avere un paragone coerente andranno, ovviamente, usate impostazioni identiche in entrambe le versioni del simulatore, perciò cuffie Monitor, cuffie Target e curve di compensazione usate dovranno essere scelte uguali in entrambi i casi. Per tutti i test di questo tipo si sono usate le cuffie *AKG K240 MKII* come Monitor e le cuffie *AKG K701* come Target, e la curva di compensazione *Harman OE 2018*.
Una volta generati i segnali, si potranno analizzare con un semplice script Matlab:

```
[FIR_Audio, fs ]= audioread('LAmaj7_Noised_FIR.wav');
ReferenceAudio = audioread('
    LAmaj7_Noised_Processed_Audio_Monitor_Beyerdynamic_T1_Target_AKG_K240withFIR.wav');

% Conversione da Stereo a Mono per migliorare la visualizzazione
FIR_Audio = (FIR_Audio(:,1) + FIR_Audio(:,2))/2;
ReferenceAudio = (ReferenceAudio(:,1) + ReferenceAudio(:,2))/2;

figure(1)
hold on, grid on, grid minor
pspectrum(FIR_Audio,fs)
pspectrum(ReferenceAudio,fs)
title('Power␣Spectrum␣of␣FIR␣processed␣Signal␣vs␣Reference␣Signal')
legend('FIR␣Processed','Reference')
```

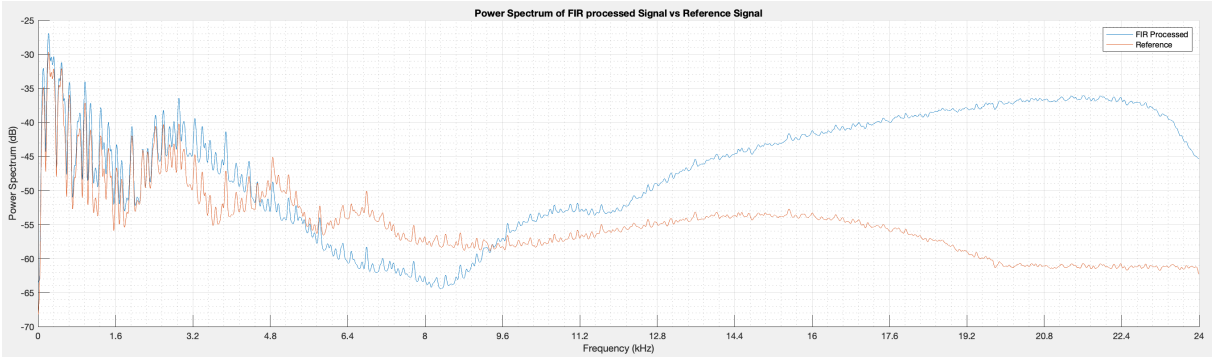


Figure 14: Spettro di potenza del segnale elaborato con il plugin audio rispetto al segnale di riferimento

Nella figura 14 sono illustrati gli spettri di potenza del segnale elaborato col plugin e col software di Anna. Da questo confronto, è possibile notare come i due spettri non coincidano, ed in particolar modo, da circa 4.8kHz questa differenza aumenta notevolmente.
Questa divergenza può essere dovuta al fatto che i filtri non sono costruiti rigorosamente secondo i dati forniti dai file `.txt` del database Oratory1990 ma sono costruiti per approssimazione, cercando cioè di creare una catena di filtri che abbia una risposta in frequenza complessiva il più simile possibile a quella desiderata. Per ottimizzare le prestazioni del plugin sviluppato, è possibile considerare un significativo incremento nel numero di filtri impiegati per approssimare le diverse risposte in frequenza. Tuttavia, l'uso di filtri di ordine più elevato potrebbe rallentare e rendere meno efficiente l'algoritmo di elaborazione finale. Tale risultato ha portato allo sviluppo di una versione ottimale del plugin.

Il codice della prima versione del plugin, chiamato `HeadPhoneSimulatorFIR`, è consultabile nell' appendice A.

[11] Il Power Spectrum di un generico segnale, o *Spettro di Potenza*, descrive la distribuzione della potenza nelle componento di frequenza che compongono il segnale stesso

5 Seconda Implementazione

Nella seconda versione del plugin è stato usato un approccio totalmente diverso: al posto di usare una catena di filtri per approssimare la risposta in frequenza complessiva desiderata, sono state estratte tutte le risposte impulsive necessarie dal software `hpSimulator.py`, che vengono poi usate nel plugin, per elaborare il segnale di ingresso usando la convoluzione discreta.
In questo modo l'elaborazione eseguita sul segnale in ingresso produrrà sicuramente risultati coerenti con quelli ottenuti elaborando lo stesso segnale con il software di partenza, ed inoltre la funzione `process` di questa implementazione risulterà più semplice e concisa della precedente.
Per generare una classe adatta alla generazione di plugin il codice dovrà essere strutturato in modo leggermente più complesso rispetto quanto visto fin'ora.

5.1 Matlab System Objects

I `System Objects` di Matlab sono oggetti specializzati per implementare e simulare sistemi dinamici ai quali sono applicati segnali d'ingresso variabili nel tempo. Essi utilizzano stati interni per memorizzare il comportamento passato, che poi vengono utilizzati nella computazione successiva.
Questa capacità li rende particolarmente efficienti per elaborare grandi flussi di dati presentati come sequenza di segmenti, metodo con il quale vengono gestiti i segnali audio e i segnali video nelle applicazioni real time. La capacità di elaborare dati in streaming offre anche il vantaggio di non dover conservare grandi moli di dati in memoria.
Nello specifico, nel plugin si fa uso di molteplici istanze di `dsp.FrequencyDomainFIRFilter` che, come suggerito dal nome, implementa tramite filtri FIR il filtraggio nel dominio della frequenza struttando la FFT (*Fast Fourier Transform*), ed è ottimizzata per implementare filtri con risposte impulsive molto lunghe. Questa oggetto sarà usato in concomitanza con `step()`, funzione comune a ogni System Object che ne esegue l'algoritmo. In generale l'output di `step` dipende dall'oggetto di sistema stesso, ed in questo caso sarà il risultato della convoluzione tra segnale in ingresso e risposta impulsiva del filtro.
Per sfruttare questi oggetti per prima cosa vanno caricate in memoria le risposte impulsive:

```
ImpulseResponse1 = audioread('IRs/M_AkgK240_T_AkgK701.wav').';
```

E successivamente, nella fase di costruzione, si istazia l'oggetto :

```
plugin.FIR1 = dsp.FrequencyDomainFIRFilter('Numerator', plugin.ImpulseResponse1 ,'
    PartitionForReducedLatency', true, 'PartitionLength', plugin.frameSize);
```

Generalmente una specifica quantità di dati viene passata all'oggetto in ogni iterazione del ciclo, perció in questa implementazione entrerà in gioco la variabile `frameSize`, presentata nel capitolo 2, che in `HeadPhoneSimulatorFIR` era gestita automaticamente.

5.2 Struttura del Codice

Poichè vengono utilizzati `System Objects`, il plugin dovrà essere una sottoclasse anche di `Matlab.System`, perciò dovremo modificare l'intestazione della classe come segue :

```
classdef HeadPhoneSimulatorIR < audioPlugin & Matlab.System %#codegen
```

Inoltre la struttura generale della classe presenterà alcune differenze rispetto alla struttura vista nel capitolo 2. Nello specifico, si dovranno aggiungere svariate funzioni, alcune tipiche delle sottoclassi di `Matlab.System` e alcune dette `propagators`.
Tra le prime si hanno :

- `stepImpl( )`: Questa funzione viene chiamata ad ogni passo del processo di elaborazione, ed è l'equivalente della funzione `process`. Questa applicherà la convoluzione veloce utilizzando gli oggetti `dsp.FrequencyDomainFIRFilter` del plugin.
- `setupImpl( )`: Viene chiamata all'inizio della sessione per configurare ogni filtro FIR del plugin. Imposta i diversi oggetti `dsp.FrequencyDomainFIRFilter` come filtri FIR nel dominio della frequenza con risposta impulsiva e `frameSize` specificati.
- `resetImpl( )`: Questa funzione resetta gli oggetti `dsp.FrequencyDomainFIRFilter` del plugin ed imposta la latenza in campioni del plugin alla dimensione della partizione, prendendo di fatto il ruolo della funzione `reset()`.

- `isInputSizeMutableImpl( )`: Questa funzione ritorna `true`, indicando che la dimensione dell'input può cambiare da un passo all'altro.
- `saveObjectImpl( )` è utilizzata per salvare lo stato corrente dell'oggetto. Salva tutte le proprietà rilevanti dell'oggetto in modo che possano essere caricate successivamente utilizzando la funzione `loadObjectImpl`.
- `loadObjectImpl( )` è utilizzata per caricare lo stato salvato di un oggetto. Carica tutte le proprietà rilevanti dell'oggetto che sono state salvate in precedenza utilizzando la funzione `saveObjectImpl`. Questo è utile per mantenere consistente lo stato dell'oggetto tra diverse sessioni di utilizzo.

Le funzioni chiamate *Propagators* sono metodi utilizzati per determinare le proprietà dell'uscita in base alle proprietà dell'ingresso, per poi propagare le proprietà dell'input all'output.

- `isOutputComplexImpl( )`: Questa funzione ritorna un booleano che indica se l'output è complesso o meno. In questo caso ritornerà sempre `false`.
- `getOutputSizeImpl( )`: Ritorna la dimensione dell'output, che in questo caso è la stessa dell'input, quindi usa la funzione `propagatedInputSize( )` internamente per ottenere la dimensione dell'input.
- `getOutputDataTypeImpl( )`: Ritorna il tipo di dati dell'output, che in questo caso è lo stesso dell'input, quindi usa internamente la funzione `propagatedInputDataType( )` per ottenere il tipo di dati dell'input.
- `isInputSizeMutableImpl( )`: Questa funzione ritorna `true`, indicando che la dimensione dell'input può cambiare da un passo all'altro.

L'ultima funzione usata nella seconda implementazione è `updateIR`, chiamata dalle funzioni `set.Monitor` e `set.Target` ogni volta che l'utente modifica i valori di una di queste variabili.
Questa aggiorna la risposta impulsiva del filtro utilizzato per elaborare il segnale di ingresso, un oggetto `dsp.FrequencyDomainFIRFilter` aggiuntivo chiamato `currentFIR`.
Rispetto alla versione precedente, la struttura del codice cambia nel raggruppamento di tutte le funzioni appena descritte in un blocco di metodi protetti. Queste infatti sono destinate al solo uso interno e non devono essere accessibili o modificabili dall'esterno.
Un'altra differenza è data dal fatto che le risposte impulsive utilizzate una volta caricate in memoria non potranno più essere modificate, perciò queste andranno racchiuse in un blocco di proprietà non modificabili, dette *Nontunable*.
Il codice della seconda versione del plugin è consultabile nell'appendice B.

5.3 Valutazione delle Prestazioni

Si possono valutare le prestazioni di `HeadPhoneSimulatorIR` con test del tutto analoghi a quelli svolti nella sezione 4.3, calcolando lo spettro di potenza di un segnale processato con il plugin per paragonarlo a quello dello stesso segnale processato col software della Dott.ssa Zuccante.

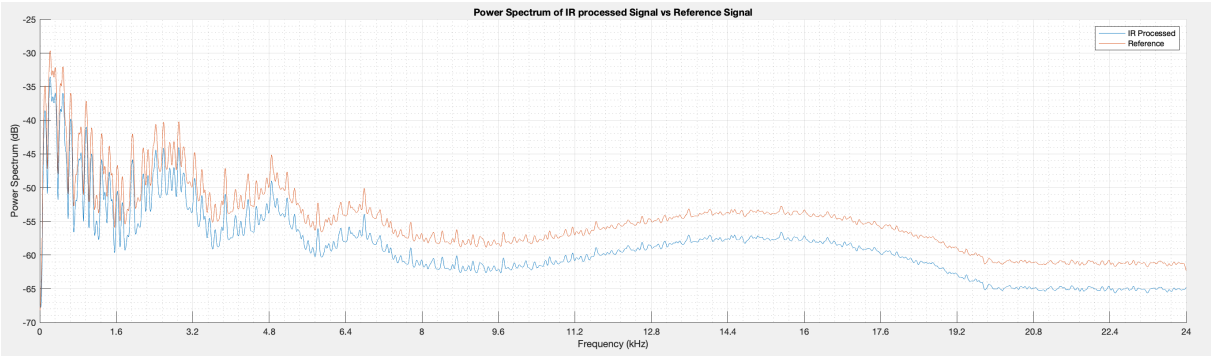


Figure 15: Spettro di potenza del segnale elaborato con il plugin audio rispetto al segnale di riferimento

Nella figura 15 si può osservare il risultato del test eseguito. Si nota immediatamente che le prestazioni di questa implementazione sono di gran lunga migliori di quelle dell'implementazione precedente. L'unica divergenza che emerge osservando questi due spettri di potenza è infatti una differenza di ampiezza di circa 4 dB, facilmente correggibile in una DAW.
Questo risultato positivo era da aspettarsi, dato che per filtrare il segnale di ingresso si sono usate le risposte impulsive che vengono generate e utilizzate da `hpSimulatorOratory.py`.
La seconda versione del plugin offre un altro vantaggio: dato che ogni risposta impulsiva viene caricata in memoria una sola volta all'inizio della sessione, ad ogni interazione col plugin nel quale l'utente cambia le cuffie da simulare o le cuffie indossate basterà aggiornare il filtro salvato nell'oggetto `currentFIR` per avere l'effetto desiderato. Questa operazione da un punto di vista computazionale è molto più semplice e veloce rispetto a dover aggiornare un'intera catena di filtri biquadratici, e permette di usare una sola funzione aggiuntiva piuttosto che due.
Si potrebbe pensare di caricare dinamicamente le varie risposte impulsive necessarie durante l'esecuzione stessa, in modo da alleggerire l'occupazione della memoria e migliorare la leggibilità del codice. Un tentativo è stato fatto, ma vista la natura dei plugin generati da Matlab questo sarebbe complesso da implementare e richiederebbe uno studio più lungo. Una futura implementazione di questo plugin idealmente dovrebbe caricare la risposta in frequenza desiderata direttamente dal database *Oratory1990*, per ricavare da questa la risposta impulsiva necessaria. Questa operazione non è banale e complicherebbe sia l'algoritmo di elaborazione che la struttura del plugin, che probabilmente non potrebbe più essere sviluppato coi tools forniti da Matlab ma dovrebbe essere sviluppato con gli strumenti più avanzati forniti dal linguaggio C++ e dal framework JUCE.
La figura 16 mette a confronto gli spettri di potenza del segnale originale, dello stesso processato con il software di partenza, chiamato *Reference*, e dei due segnali ottenuti utilizzando `HeadPhoneSimulatorFIR` e `HeadPhoneSimulatorIR`. Questa evidenzia meglio la discrepanza tra le prestazioni delle due diverse implementazioni.

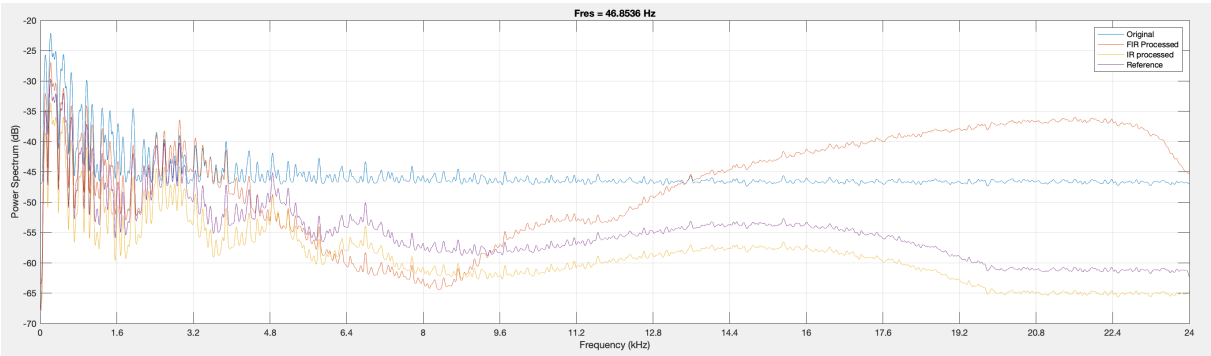


Figure 16: Spettri di potenza: segnale originale (blu), di riferimento(viola) e processato con le due implementazioni descritte (arancio e giallo)

.

6 Terza Implementazione

L'approccio pensato inizialmente per trasformare il software sviluppato dalla Dott.ssa Zuccante in un plugin audio era di tradurre il codice di `hpSimulatorOratory.py` funzione per funzione per poi organizzare il tutto in una classe `audioPlugin` Matlab. Tuttavia sono stati trovati dei limiti che hanno evidenziato come questa soluzione non fosse abbastanza efficiente per essere usata in un'applicazione real time come un plugin audio. Questi limiti hanno portato quindi allo sviluppo delle implementazioni viste nei capitoli precedenti.
Una volta tradotte le funzioni di `hpSimulOratory.py` necessarie in codice Matlab, sono stati eseguiti alcuni test per valutare l'efficienza dell'algoritmo sviluppato, dai quali sono emersi diversi problemi di performance, sia per quanto riguarda la fedeltà nella ricostruzione della risposta impulsiva desiderata che nel tempo di esecuzione dell'algoritmo.

6.1 Limiti

Il limite più vincolante sorge nella fase finale della funzione chiamata `generateIR`, nella quale, a seguito di tutte le elaborazioni sulle risposte in frequenza eseguite dalle precedenti funzioni, viene creata la risposta impulsiva finale che sarà usata per filtrare il segnale dato all'ingresso. Segue la funzione:

```
function genimpresp = generateIR(freqs, smoothedComb)
    PREAMP_HEADROOM = 0.2; F_STEP = 1.01; DEFAULT_FS = 48000;
    N = floor(freqs(floor(length(freqs) / 2)) / F_STEP);
    last_freq_half = freqs(floor(length(freqs) / 2));
    last_freqs = (linspace(last_freq_half + F_STEP, floor(DEFAULT_FS / 2), N+1))';
    half_freqs = [freqs(1:floor(length(freqs) / 2)); last_freqs];
    smoothedCombHalf = smoothedComb(1:floor(length(freqs) / 2));
    smoothedCombHalf = [smoothedCombHalf; ones(length(last_freqs),1) * smoothedComb(floor(length
        (freqs) / 2)) ];
    csspl = csapi(half_freqs, smoothedCombHalf);
    smoothedCombHalf = fnval(csspl, half_freqs);
    smoothedCombHalf = smoothedCombHalf - max(smoothedCombHalf);
    smoothedCombHalf = smoothedCombHalf - PREAMP_HEADROOM;
    smoothedCombHalf = smoothedCombHalf * 2;
    corrected_raw_values = 10 .^ (smoothedCombHalf / 20); % from dB to linear
    corrected_raw_values(end) = 0.0; % NYQUIST freq
    F = half_freqs(:) / (DEFAULT_FS / 2);
    F = F/max(F);
    % calcolo della risposta impulsiva finale
    genimpresp = firminphase( firls( length(half_freqs), F, corrected_raw_values ) );
end
```

Le prestazioni totali del sistema sono fortemente influenzate dall'ultima riga di questo metodo, che calcola la risposta impulsiva finale del sistema. Questa è ottenuta usando due funzioni, `firls()` che è utilizzata per realizzare filtri FIR con fase lineare e successivamente `firminphase()`, che calcola il fattore spettrale di fase minima di un filtro FIR a fase lineare. `firls()` prende al primo argomento l'ordine del filtro da generare, che nel codice di `hpSimulOratory.py` è impostato alla lunghezza del vettore monolatero delle frequenze[12]. Ciò che limita le prestazioni del nostro sistema è proprio la creazione di un filtro di ordine molto alto: questa è un'operazione computazionalmente complessa che richiede un tempo di esecuzione troppo lungo per un'applicazione pensata per elaborazioni real time.
Il sistema è stato testato più volte, creando via via filtri di ordine decrescente e valutando la differenza tra un segnale elaborato con il software originale ed un segnale elaborato con la risposta impulsiva appena generata. Da questi test è sorto il seguente trade-off:
Generare filtri di ordine alto porta ad avere risposte impulsive molto simili alle originali, ma creare filtri complessi aumenta significativamente il tempo di calcolo. Per ridurlo si possono generare filtri di ordine più basso, ovviamente creando risposte impulsive più approssimate che non saranno in grado nè di compensare correttamente la risposta in frequenza delle cuffie Monitor nè di applicare la risposta in frequenza delle Cuffie Target, fallendo quindi nel simulare fedelmente l'esperienza di ascolto di un determinato paio di cuffie attraverso un altro.

Lo script Matlab usato per i test chiama la funzione `extractIR`, che esegue internamente tutte le operazioni di preprocessing sulle risposte in frequenza di monitor e target, e ne misura il tempo di esecuzione. Successivamente elabora lo stesso segnale di riferimento con una risposta impulsiva generata con il software

[12] Contenente solamente frequenze positive

`hpSimulOratory.py` e carica il file generato da extractIR, chiamato *"Risultato.wav"*, per confrontarne l'output nel dominio della frequenza con il metodo usato in precedenza.
Segue il codice usato per testare il sistema.

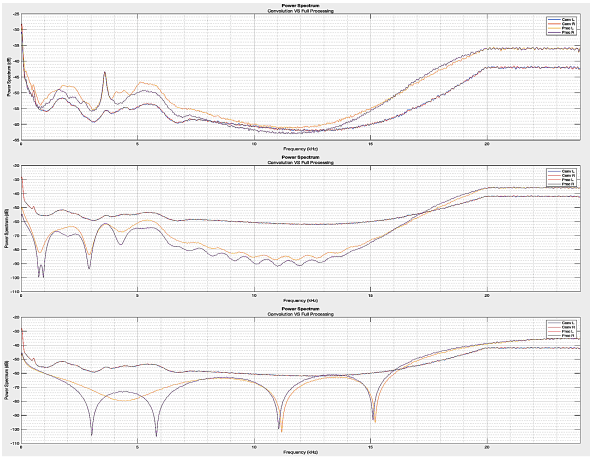
```
monitor = 'AKG K240'; target = 'AKG K701';

tic , extractIR(monitor,target) , toc

[impresp, fs] = audioread('M_AkgK240_T_AkgK701.wav');
noiseSweep = audioread("NoiseSweeper.wav");
conv_result_L = conv(noiseSweep(:,1),impresp(:,1));
conv_result_R = conv(noiseSweep(:,2),impresp(:,2));
conv_result = [conv_result_L,conv_result_R];
result = audioread('Risultato.wav');

figure(1), hold on , grid on, grid minor
pspectrum(conv_result,fs)
pspectrum(result,fs)
title('Power Spectrum','Convolution VS Full Processing');
legend(['Conv L','Conv R','Proc L','Proc R'])
```

6.2 Valutazione delle prestazioni

Si è deciso di non proseguire sviluppando il plugin con questo approccio implementativo perché i risultati di questi test non sono stati abbastanza soddisfacenti: filtri con buone prestazioni risultano troppo lunghi da creare mentre filtri veloci da creare hanno prestazioni troppo scadenti. I risultati dei test sono riportati nella tabella 2, che riporta i dati sperimentali di tempo impiegato da `extractIR` per generare la risposta impulsiva in e l'ordine del relativo filtro. La figura adiacente illustra gli spettri di potenza dei due segnali elaborati con `extractIR` e `hpSimulOratory.py`, per N = 1000, 100, 10.

Ordine del filtro	Tempo impiegato
`length(half freqs)`	>10 min
`length(half freqs)/4`	>10 min
N = 1000	299.331498 sec
N = 500	79.536514 sec
N = 300	36.745420 sec
N = 200	23.450392 sec
N = 100	12.810245 sec
N = 50	9.836871 sec
N = 30	9.234718 sec
N = 20	9.154286 sec
N = 10	7.903303sec

Table 2: Prestazioni di extractIR



Dalla tabella e dagli spettri di potenza emerge immediatamente il trade-off sopracitato: nonostante i filtri di grado più alto approssimino meglio la risposta impulsiva desiderata, questi per calcolarla impiegano tempi troppo elevati per applicazioni real time, e filtri più veloci da implementare hanno performance troppo scadenti per essere utilizzati. Nelle prime due entrate della tabella il tempo di esecuzione è indicato con un limite inferiore dato che, una volta lanciato l'algoritmo di test, dopo più di 10 minuti questo non era ancora arrivato a compimento. Perciò, si è deciso di bloccare l'esecuzione e di riprovare il test diminuendo l'ordine del filtro.
Si vede inoltre che un filtro di ordine 1000 impiega quasi 5 minuti per essere ottenuto, e già presenta una discrepanza significativa rispetto al filtro di riferimento, soprattutto sulle basse frequenze, differenza chiaramente udibile ascoltando il segnale di riferimento e quello appena generato.
Per proseguire sviluppando il plugin con questo metodo lo si dovrebbe programmare facendo in modo che ogni risposta impulsiva sia calcolata all'istanziazione della classe stessa, ma avendo 36 risposte impulsive diverse, con filtri (*non ottimi!*) di ordine 1000 il caricamento del plugin durerebbe quasi 3 ore. Programmando invece il software in modo che ricalcoli la risposta impulsiva necessaria ogni volta che l'utente modifica il modello di cuffie Target o Monitor si perderebbe il vantaggio dato dall'elusione della memoria ecogena, che era uno dei principali obbiettivi preposti per questa tesi.

7 Conclusioni

Dopo una parte introduttiva sullo sviluppo di software audio tramite Matlab e sulla simulazione di cuffie Hi-fi, in questa tesi sono stati proposti ed esaminati diversi approcci per trasformare il software di simulazione di cuffie `HeadPhoneSimulator` in un plugin audio VST, sviluppato in precedenza dalla Dottoressa Anna Zuccante, per renderlo compatibile con le più comuni Digital Audio Workstations.
Come specificato in precedenza, l'obiettivo principale di questo studio era quello di fornire una versione del software di partenza che desse la possibilità di eseguire confronti in real time tra diversi modelli di cuffie, in modo da eludere le brevi tempistiche della memoria ecogena. Si è cercato un modo efficiente di eseguire questa simulazione in tempo reale, concentrandosi sul rendere immediata la transizione tra la simulazione di due modelli di cuffie differenti.
Questo obbiettivo ha portato allo sviluppo del plugin `HeadPhoneSimulatorIR`, visto nel capitolo 5, che implementa la simulazione a partire dalle risposte impulsive fornite da `hpSimulOraroty.py`. Questa soluzione è quella che ha fornito risultati migliori, dimostrandosi affidabile dal punto di vista della fedeltà della simulazione audio ed efficiente dal punto di vista della complessità computazionale.
Le altre due soluzioni proposte non si sono rilevate abbastanza performanti o efficienti per essere utilizzate. La soluzione che fa utilizzo di una catena di filtri FIR, `HeadPhoneSimulatorFIR`, non dà risultati soddisfacenti dal punto di vista della qualità della simulazione.
La soluzione che invece copia pari pari le funzioni definite in `hpSimulatorOratory.py` si è dimostrata troppo inefficiente dal punto di vista computazionale per essere tradotta in un software real time.

7.1 Miglioramenti Futuri

Questo progetto è comunque ancora allo stato iniziale, e molti miglioramenti potrebbero essere portati con studi futuri:

- Sviluppare nuovamente il plugin audio utilizzando il linguaggio `C++` ed il framework `JUCE`. Le librerie fornite da questi darebbero la possibilità di utilizzare funzionalità più avanzate e di rendere quindi più efficiente e robusto il software sviluppato. In secondo luogo si avrebbe la possibilità di rendere l'aspetto del plugin più coerente con il software della Dott.ssa Zuccante, dando la possibilità di visualizzare i dati delle cuffie simulate/indossate e la risposta in frequenza applicata al segnale.

- Dare all'utente la possibilità, come nel software di partenza, di scegliere tra diversi database dai quali estrarre le risposte in frequenza delle cuffie, e anche di caricare autonomamente risposte impulsive (o in frequenza) a suo piacimento, un po' come nei comuni riverberi a convoluzione.

- Nel caso in cui si riesca ad implementare il plugin di simulazione in modo che calcoli autonomamente la risposta del filtro a partire dai dati forniti nei database, lo si potrebbe collegare direttamente coi database web d'interesse. In questo modo si avrebbe la possibilità di simulare molti più modelli di cuffie e il plugin si aggiornerebbe autonomamente ogni qualvolta venissero aggiornati i database.

La figura 17 mostra i due plugin `HeadPhoneSimulatroFIR` e `HeadPhoneSimulatroIR` all'interno di Fl Studio.

Figure 17: `HeadPhoneSimulatroFIR` e `HeadPhoneSimulatroIR` all'interno di Fl Studio

A HeadPhoneSimulatroFIR

Segue il codice della classe `HeadPhoneSimulatorFIR` e delle relative funzioni esterne

```
classdef HeadPhoneSimulatorFIR < audioPlugin %#codegen
    properties % variabili globali
        Monitor = MonitorEnum.None;
        Target = TargetEnum.None;
        Bypass = false;
        Compensation = true;
        Fs = 44100;
    end
    properties (Constant) % interfaccia utente
        PluginInterface = audioPluginInterface( ...
            'PluginName','HeadPhoneSimulator_FIR','InputChannels',2,'OutputChannels',2,...
            'VendorName','BenDaTek_Distributions','VendorVersion','0.0.1',...
            ...'BackgroundImage','sfondo.jpg',...
            audioPluginParameter('Monitor','DisplayName','Monitor',...
            'Mapping',{'enum','None','AKG K 240 MKII','Beyerdynamic DT 990 250ohm',...
                       'Beyerdynamic T1','Sennheiser HD600'},'Layout',[1,2]),...
            audioPluginParameter('Target','DisplayName','Target',...
            'Mapping',{'enum','None','AKG K 240 MKII','AKG K 701','Beyerdynamic DT 770 80ohm',...
                       'Beyerdynamic DT 990 250ohm','Beyerdynamic T1','Focal Stellia',...
                       'Sennheiser HD600','Sennheiser HD650','Sennheiser HD800s', ...
                       'Shure SR H1540'},'Layout',[3,2]),...
            audioPluginParameter('Compensation','DisplayName','Compensation',...
                         'Style','vrocker','Layout',[1 1; 3 1]),...
            audioPluginParameter('Bypass','DisplayName','Bypass',...
                         'Style','vrocker','Layout',[1 3; 3 3]),...
            audioPluginGridLayout('RowHeight', [60,30,60,30],...
                          'ColumnWidth',[100 300 60],...
                          'Padding', [10 10 10 10 ]));
    end
    properties (Access = private) % variabili private
        Harman_1 = struct('w',[ 0,0 ; 0,0 ],'a',[1,0,0],'b',[1,0,0]);
        Harman_2 = struct('w',[ 0,0 ; 0,0 ],'a',[1,0,0],'b',[1,0,0]);
        Harman_3 = struct('w',[ 0,0 ; 0,0 ],'a',[1,0,0],'b',[1,0,0]);
        Harman_4 = struct('w',[ 0,0 ; 0,0 ],'a',[1,0,0],'b',[1,0,0]);
        Harman_5 = struct('w',[ 0,0 ; 0,0 ],'a',[1,0,0],'b',[1,0,0]);
        Harman_6 = struct('w',[ 0,0 ; 0,0 ],'a',[1,0,0],'b',[1,0,0]);
        Harman_7 = struct('w',[ 0,0 ; 0,0 ],'a',[1,0,0],'b',[1,0,0]);
        Harman_8 = struct('w',[ 0,0 ; 0,0 ],'a',[1,0,0],'b',[1,0,0]);
        Harman_9 = struct('w',[ 0,0 ; 0,0 ],'a',[1,0,0],'b',[1,0,0]);
        Harman_10 = struct('w',[ 0,0 ; 0,0 ],'a',[1,0,0],'b',[1,0,0]);
        Harman_11 = struct('w',[ 0,0 ; 0,0 ],'a',[1,0,0],'b',[1,0,0]);

        filt0_Monitor = struct('w',[ 0,0 ; 0,0 ],'a',[1,0,0],'b',[1,0,0]);
        filt1_Monitor = struct('w',[ 0,0 ; 0,0 ],'a',[1,0,0],'b',[1,0,0]);
        filt2_Monitor = struct('w',[ 0,0 ; 0,0 ],'a',[1,0,0],'b',[1,0,0]);
        filt3_Monitor = struct('w',[ 0,0 ; 0,0 ],'a',[1,0,0],'b',[1,0,0]);
        filt4_Monitor = struct('w',[ 0,0 ; 0,0 ],'a',[1,0,0],'b',[1,0,0]);
        filt5_Monitor = struct('w',[ 0,0 ; 0,0 ],'a',[1,0,0],'b',[1,0,0]);
        filt6_Monitor = struct('w',[ 0,0 ; 0,0 ],'a',[1,0,0],'b',[1,0,0]);
        filt7_Monitor = struct('w',[ 0,0 ; 0,0 ],'a',[1,0,0],'b',[1,0,0]);
        filt8_Monitor = struct('w',[ 0,0 ; 0,0 ],'a',[1,0,0],'b',[1,0,0]);
        filt9_Monitor = struct('w',[ 0,0 ; 0,0 ],'a',[1,0,0],'b',[1,0,0]);

        filt0_Target = struct('w',[ 0,0 ; 0,0 ],'a',[1,0,0],'b',[1,0,0]);
        filt1_Target = struct('w',[ 0,0 ; 0,0 ],'a',[1,0,0],'b',[1,0,0]);
        filt2_Target = struct('w',[ 0,0 ; 0,0 ],'a',[1,0,0],'b',[1,0,0]);
        filt3_Target = struct('w',[ 0,0 ; 0,0 ],'a',[1,0,0],'b',[1,0,0]);
        filt4_Target = struct('w',[ 0,0 ; 0,0 ],'a',[1,0,0],'b',[1,0,0]);
        filt5_Target = struct('w',[ 0,0 ; 0,0 ],'a',[1,0,0],'b',[1,0,0]);
        filt6_Target = struct('w',[ 0,0 ; 0,0 ],'a',[1,0,0],'b',[1,0,0]);
        filt7_Target = struct('w',[ 0,0 ; 0,0 ],'a',[1,0,0],'b',[1,0,0]);
        filt8_Target = struct('w',[ 0,0 ; 0,0 ],'a',[1,0,0],'b',[1,0,0]);
        filt9_Target = struct('w',[ 0,0 ; 0,0 ],'a',[1,0,0],'b',[1,0,0]);
    end
    methods
        function plugin = HeadPhoneSimulatorFIR() % costruttore
            plugin.Fs = getSampleRate(plugin);

            plugin.Harman_1.w = [ 0 , 0 ; 0 , 0];
            plugin.Harman_2.w = [ 0 , 0 ; 0 , 0];
            plugin.Harman_3.w = [ 0 , 0 ; 0 , 0];
            plugin.Harman_4.w = [ 0 , 0 ; 0 , 0];
            plugin.Harman_5.w = [ 0 , 0 ; 0 , 0];
            plugin.Harman_6.w = [ 0 , 0 ; 0 , 0];
            plugin.Harman_7.w = [ 0 , 0 ; 0 , 0];
            plugin.Harman_8.w = [ 0 , 0 ; 0 , 0];
            plugin.Harman_9.w = [ 0 , 0 ; 0 , 0];
            plugin.Harman_10.w = [ 0 , 0 ; 0 , 0];
            plugin.Harman_11.w = [ 0 , 0 ; 0 , 0];

            [plugin.Harman_1.a,plugin.Harman_1.b] = biquad_LS_filterCoefficients(plugin.Fs, 68, 4.95 , 1.02);
            [plugin.Harman_2.a,plugin.Harman_2.b] = biquad_PK_filterCoefficients(plugin.Fs, 220, -1.50 , 0.64);
            [plugin.Harman_3.a,plugin.Harman_3.b] = biquad_PK_filterCoefficients(plugin.Fs, 1245, -2.70 , 2.38);
            [plugin.Harman_4.a,plugin.Harman_4.b] = biquad_PK_filterCoefficients(plugin.Fs, 2165, 2.40 , 4.12);
            [plugin.Harman_5.a,plugin.Harman_5.b] = biquad_PK_filterCoefficients(plugin.Fs, 4686, 5.90 , 2.41);
            [plugin.Harman_6.a,plugin.Harman_6.b] = biquad_PK_filterCoefficients(plugin.Fs, 5348, 2.00 , 0.98);
```

```
87              [plugin.Harman_7.a,plugin.Harman_7.b] = biquad_PK_filterCoefficients(plugin.Fs, 8276, 5.10 , 1.55);
88              [plugin.Harman_8.a,plugin.Harman_8.b] = biquad_PK_filterCoefficients(plugin.Fs, 11312, 4.52 , 0.71);
89              [plugin.Harman_9.a,plugin.Harman_9.b] = biquad_HS_filterCoefficients(plugin.Fs, 12500, 0.96 , 0.70);
90              [plugin.Harman_10.a,plugin.Harman_10.b] = biquad_PK_filterCoefficients(plugin.Fs, 13181, 3.50 , 0.60);
91              [plugin.Harman_11.a,plugin.Harman_11.b] = biquad_PK_filterCoefficients(plugin.Fs, 18477, -9.50 , 0.09);
92          end
93
94          function reset(plugin) % funzione di reset
95              plugin.Fs = getSampleRate(plugin);
96
97              plugin.Harman_1.w = [ 0 , 0 ; 0 , 0];
98              plugin.Harman_2.w = [ 0 , 0 ; 0 , 0];
99              plugin.Harman_3.w = [ 0 , 0 ; 0 , 0];
100             plugin.Harman_4.w = [ 0 , 0 ; 0 , 0];
101             plugin.Harman_5.w = [ 0 , 0 ; 0 , 0];
102             plugin.Harman_6.w = [ 0 , 0 ; 0 , 0];
103             plugin.Harman_7.w = [ 0 , 0 ; 0 , 0];
104             plugin.Harman_8.w = [ 0 , 0 ; 0 , 0];
105             plugin.Harman_9.w = [ 0 , 0 ; 0 , 0];
106             plugin.Harman_10.w = [ 0 , 0 ; 0 , 0];
107             plugin.Harman_11.w = [ 0 , 0 ; 0 , 0];
108
109             plugin.filt0_Target.w = [ 0 , 0 ; 0 , 0];
110             plugin.filt1_Target.w = [ 0 , 0 ; 0 , 0];
111             plugin.filt2_Target.w = [ 0 , 0 ; 0 , 0];
112             plugin.filt3_Target.w = [ 0 , 0 ; 0 , 0];
113             plugin.filt4_Target.w = [ 0 , 0 ; 0 , 0];
114             plugin.filt5_Target.w = [ 0 , 0 ; 0 , 0];
115             plugin.filt6_Target.w = [ 0 , 0 ; 0 , 0];
116             plugin.filt7_Target.w = [ 0 , 0 ; 0 , 0];
117             plugin.filt8_Target.w = [ 0 , 0 ; 0 , 0];
118             plugin.filt9_Target.w = [ 0 , 0 ; 0 , 0];
119
120             plugin.filt0_Monitor.w = [ 0 , 0 ; 0 , 0];
121             plugin.filt1_Monitor.w = [ 0 , 0 ; 0 , 0];
122             plugin.filt2_Monitor.w = [ 0 , 0 ; 0 , 0];
123             plugin.filt3_Monitor.w = [ 0 , 0 ; 0 , 0];
124             plugin.filt4_Monitor.w = [ 0 , 0 ; 0 , 0];
125             plugin.filt5_Monitor.w = [ 0 , 0 ; 0 , 0];
126             plugin.filt6_Monitor.w = [ 0 , 0 ; 0 , 0];
127             plugin.filt7_Monitor.w = [ 0 , 0 ; 0 , 0];
128             plugin.filt8_Monitor.w = [ 0 , 0 ; 0 , 0];
129             plugin.filt9_Monitor.w = [ 0 , 0 ; 0 , 0];
130         end
131         function output = process(plugin,input) % funzione di elaborazione
132             output = input;
133             if plugin.Bypass == 0
134                 for ch = 1:min(size(input))
135
136                     y = input(:,ch);
137
138                     [yH1 , plugin.Harman_1.w(:,ch)] = processBiquad(y , plugin.Harman_1 , ch);
139                     [yH2 , plugin.Harman_2.w(:,ch)] = processBiquad(yH1 , plugin.Harman_2 , ch);
140                     [yH3 , plugin.Harman_3.w(:,ch)] = processBiquad(yH2 , plugin.Harman_3 , ch);
141                     [yH4 , plugin.Harman_4.w(:,ch)] = processBiquad(yH3 , plugin.Harman_4 , ch);
142                     [yH5 , plugin.Harman_5.w(:,ch)] = processBiquad(yH4 , plugin.Harman_5 , ch);
143                     [yH6 , plugin.Harman_6.w(:,ch)] = processBiquad(yH5 , plugin.Harman_6 , ch);
144                     [yH7 , plugin.Harman_7.w(:,ch)] = processBiquad(yH6 , plugin.Harman_7 , ch);
145                     [yH8 , plugin.Harman_8.w(:,ch)] = processBiquad(yH7 , plugin.Harman_8 , ch);
146                     [yH9 , plugin.Harman_9.w(:,ch)] = processBiquad(yH8 , plugin.Harman_9 , ch);
147                     [yH10 , plugin.Harman_10.w(:,ch)] = processBiquad(yH9 , plugin.Harman_10, ch);
148                     [yH , plugin.Harman_11.w(:,ch)] = processBiquad(yH10 , plugin.Harman_11, ch);
149
150                     [y0 , plugin.filt0_Monitor.w(:,ch)] = processBiquad(yH, plugin.filt0_Monitor, ch);
151                     [y1 , plugin.filt1_Monitor.w(:,ch)] = processBiquad(y0, plugin.filt1_Monitor, ch);
152                     [y2 , plugin.filt2_Monitor.w(:,ch)] = processBiquad(y1, plugin.filt2_Monitor, ch);
153                     [y3 , plugin.filt3_Monitor.w(:,ch)] = processBiquad(y2, plugin.filt3_Monitor, ch);
154                     [y4 , plugin.filt4_Monitor.w(:,ch)] = processBiquad(y3, plugin.filt4_Monitor, ch);
155                     [y5 , plugin.filt5_Monitor.w(:,ch)] = processBiquad(y4, plugin.filt5_Monitor, ch);
156                     [y6 , plugin.filt6_Monitor.w(:,ch)] = processBiquad(y5, plugin.filt6_Monitor, ch);
157                     [y7 , plugin.filt7_Monitor.w(:,ch)] = processBiquad(y6, plugin.filt7_Monitor, ch);
158                     [y8 , plugin.filt8_Monitor.w(:,ch)] = processBiquad(y7, plugin.filt8_Monitor, ch);
159                     [y9 , plugin.filt9_Monitor.w(:,ch)] = processBiquad(y8, plugin.filt9_Monitor, ch);
160
161                     [y10, plugin.filt0_Target.w(:,ch)] = processBiquad(y9 , plugin.filt0_Target, ch);
162                     [y11, plugin.filt1_Target.w(:,ch)] = processBiquad(y10, plugin.filt1_Target, ch);
163                     [y12, plugin.filt2_Target.w(:,ch)] = processBiquad(y11, plugin.filt2_Target, ch);
164                     [y13, plugin.filt3_Target.w(:,ch)] = processBiquad(y12, plugin.filt3_Target, ch);
165                     [y14, plugin.filt4_Target.w(:,ch)] = processBiquad(y13, plugin.filt4_Target, ch);
166                     [y15, plugin.filt5_Target.w(:,ch)] = processBiquad(y14, plugin.filt5_Target, ch);
167                     [y16, plugin.filt6_Target.w(:,ch)] = processBiquad(y15, plugin.filt6_Target, ch);
168                     [y17, plugin.filt7_Target.w(:,ch)] = processBiquad(y16, plugin.filt7_Target, ch);
169                     [y18, plugin.filt8_Target.w(:,ch)] = processBiquad(y17, plugin.filt8_Target, ch);
170                     [y19, plugin.filt9_Target.w(:,ch)] = processBiquad(y18, plugin.filt9_Target, ch);
171
172                     output(:,ch) = y19;
173                 end
174             end
175         end %
176         % ---------------------------- MONITOR ---------------------------------------
177         function set.Monitor( plugin , val )
```

```
178             plugin.Monitor = val;
179             updateMonitor(plugin);
180         end
181         function updateMonitor(plugin) % funzione di aggiornamento per le cuffie monitor
182             plugin.Fs = getSampleRate(plugin);
183             switch plugin.Monitor
184                 case MonitorEnum.None
185                     plugin.filt0_Monitor.a = [1,0,0]; plugin.filt0_Monitor.b = [1,0,0];
186                     plugin.filt1_Monitor.a = [1,0,0]; plugin.filt1_Monitor.b = [1,0,0];
187                     plugin.filt2_Monitor.a = [1,0,0]; plugin.filt2_Monitor.b = [1,0,0];
188                     plugin.filt3_Monitor.a = [1,0,0]; plugin.filt3_Monitor.b = [1,0,0];
189                     plugin.filt4_Monitor.a = [1,0,0]; plugin.filt4_Monitor.b = [1,0,0];
190                     plugin.filt5_Monitor.a = [1,0,0]; plugin.filt5_Monitor.b = [1,0,0];
191                     plugin.filt6_Monitor.a = [1,0,0]; plugin.filt6_Monitor.b = [1,0,0];
192                     plugin.filt7_Monitor.a = [1,0,0]; plugin.filt7_Monitor.b = [1,0,0];
193                     plugin.filt8_Monitor.a = [1,0,0]; plugin.filt8_Monitor.b = [1,0,0];
194                     plugin.filt9_Monitor.a = [1,0,0]; plugin.filt9_Monitor.b = [1,0,0];
195                 case MonitorEnum.AKGK240MKII
196                     [plugin.filt0_Monitor.a,plugin.filt0_Monitor.b] = biquad_LS_filterCoefficients(plugin.Fs, 75, -4 , 0.71)
                            ;
197                     [plugin.filt1_Monitor.a,plugin.filt1_Monitor.b] = biquad_LS_filterCoefficients(plugin.Fs, 105, -5.5,
                            0.71);
198 
199                     [plugin.filt2_Monitor.a,plugin.filt2_Monitor.b] = biquad_PK_filterCoefficients(plugin.Fs, 167, 4.7, 0.4
                            );
200                     [plugin.filt3_Monitor.a,plugin.filt3_Monitor.b] = biquad_PK_filterCoefficients(plugin.Fs, 1170, 1.7, 1.3
                            );
201                     [plugin.filt4_Monitor.a,plugin.filt4_Monitor.b] = biquad_PK_filterCoefficients(plugin.Fs, 1650, -5 , 1.4
                            );
202                     [plugin.filt5_Monitor.a,plugin.filt5_Monitor.b] = biquad_PK_filterCoefficients(plugin.Fs, 2170, 1.1, 4.0
                            );
203                     [plugin.filt6_Monitor.a,plugin.filt6_Monitor.b] = biquad_PK_filterCoefficients(plugin.Fs, 2700, 4.5, 2.5
                            );
204                     [plugin.filt7_Monitor.a,plugin.filt7_Monitor.b] = biquad_PK_filterCoefficients(plugin.Fs, 3000, 0.7, 6.0
                            );
205                     [plugin.filt8_Monitor.a,plugin.filt8_Monitor.b] = biquad_PK_filterCoefficients(plugin.Fs, 4100, -4 , 1.4
                            );
206                     [plugin.filt9_Monitor.a,plugin.filt9_Monitor.b] = biquad_PK_filterCoefficients(plugin.Fs, 6850, 4.9, 3.0
                            );
207                 case MonitorEnum.BeyerDynamicDT990250ohm
208                     [plugin.filt0_Monitor.a,plugin.filt0_Monitor.b] = biquad_LS_filterCoefficients(plugin.Fs, 105, -5 ,
                            0.71);
209 
210                     [plugin.filt1_Monitor.a,plugin.filt1_Monitor.b] = biquad_PK_filterCoefficients(plugin.Fs, 100, 5.2, 0.4
                            );
211                     [plugin.filt2_Monitor.a,plugin.filt2_Monitor.b] = biquad_PK_filterCoefficients(plugin.Fs, 650, -1.2, 1.8
                            );
212                     [plugin.filt3_Monitor.a,plugin.filt3_Monitor.b] = biquad_PK_filterCoefficients(plugin.Fs, 1900, 0.6, 3 )
                            ;
213                     [plugin.filt4_Monitor.a,plugin.filt4_Monitor.b] = biquad_PK_filterCoefficients(plugin.Fs, 2600, -0.9, 5
                            );
214                     [plugin.filt5_Monitor.a,plugin.filt5_Monitor.b] = biquad_PK_filterCoefficients(plugin.Fs, 3150, -2.5,
                            1.41);
215                     [plugin.filt6_Monitor.a,plugin.filt6_Monitor.b] = biquad_PK_filterCoefficients(plugin.Fs, 3800, 0.9, 5 )
                            ;
216                     [plugin.filt7_Monitor.a,plugin.filt7_Monitor.b] = biquad_PK_filterCoefficients(plugin.Fs, 8400, 2.5, 6 )
                            ;
217                     [plugin.filt8_Monitor.a,plugin.filt8_Monitor.b] = biquad_PK_filterCoefficients(plugin.Fs, 9400, 3 , 6 );
218 
219                     [plugin.filt9_Target.a,plugin.filt9_Target.b] = biquad_HS_filterCoefficients(plugin.Fs, 9000, 6 , 0.71 )
                            ;
220                 case MonitorEnum.BeyerDynamicT1
221                     [plugin.filt0_Monitor.a,plugin.filt0_Monitor.b] = biquad_LS_filterCoefficients(plugin.Fs, 105, -5.5,
                            0.71);
222 
223                     [plugin.filt1_Monitor.a,plugin.filt1_Monitor.b] = biquad_PK_filterCoefficients(plugin.Fs, 200, 3.2, 0.4
                            );
224                     [plugin.filt2_Monitor.a,plugin.filt2_Monitor.b] = biquad_PK_filterCoefficients(plugin.Fs, 500, -1 , 0.5
                            );
225 
226                     [plugin.filt3_Monitor.a,plugin.filt3_Monitor.b] = biquad_HS_filterCoefficients(plugin.Fs, 1700, -7 , 0.5
                            );
227 
228                     [plugin.filt4_Monitor.a,plugin.filt4_Monitor.b] = biquad_PK_filterCoefficients(plugin.Fs, 2000, 1 , 4 );
229                     [plugin.filt5_Monitor.a,plugin.filt5_Monitor.b] = biquad_PK_filterCoefficients(plugin.Fs, 3650, 1 , 3 );
230                     [plugin.filt6_Monitor.a,plugin.filt6_Monitor.b] = biquad_PK_filterCoefficients(plugin.Fs, 6080, 5.3, 3 )
                            ;
231                     [plugin.filt7_Monitor.a,plugin.filt7_Monitor.b] = biquad_PK_filterCoefficients(plugin.Fs, 8450, 11.6,
                            2.5 );
232 
233                     [plugin.filt8_Target.a,plugin.filt8_Target.b] = biquad_LS_filterCoefficients(plugin.Fs, 9000, 4 , 0.71);
234 
235                     [plugin.filt9_Target.a,plugin.filt9_Target.b] = biquad_PK_filterCoefficients(plugin.Fs,11900, 5 , 4 );
236                 case MonitorEnum.SennheiserHD600
237                     [plugin.filt0_Target.a,plugin.filt0_Target.b] = biquad_LS_filterCoefficients(plugin.Fs, 25, -6 , 0.71);
238 
239                     [plugin.filt1_Monitor.a,plugin.filt1_Monitor.b] = biquad_PK_filterCoefficients(plugin.Fs, 150, 2.9, 0.5
                            );
240                     [plugin.filt2_Monitor.a,plugin.filt2_Monitor.b] = biquad_PK_filterCoefficients(plugin.Fs, 1330, 2 , 1.4
                            );
241                     [plugin.filt3_Monitor.a,plugin.filt3_Monitor.b] = biquad_PK_filterCoefficients(plugin.Fs, 3150, 3.1,
```

```
                    2.2);
242                 [plugin.filt4_Monitor.a,plugin.filt4_Monitor.b] = biquad_PK_filterCoefficients(plugin.Fs, 3500, 1.0, 4 )
                        ;
243                 [plugin.filt5_Monitor.a,plugin.filt5_Monitor.b] = biquad_PK_filterCoefficients(plugin.Fs, 5000, 1.2, 6 )
                        ;
244                 [plugin.filt6_Monitor.a,plugin.filt6_Monitor.b] = biquad_PK_filterCoefficients(plugin.Fs, 5850, 2.2, 4.5
                         );
245                 [plugin.filt7_Monitor.a,plugin.filt7_Monitor.b] = biquad_PK_filterCoefficients(plugin.Fs, 5600, -2 ,
                        0.71);
246                 [plugin.filt8_Monitor.a,plugin.filt8_Monitor.b] = biquad_PK_filterCoefficients(plugin.Fs, 7710, 2.5, 5 )
                        ;
247
248                 [plugin.filt9_Monitor.a,plugin.filt9_Monitor.b] = biquad_HS_filterCoefficients(plugin.Fs,13000, 3 ,
                        0.71);
249             end
250         end
251         % --------------------------- TARGET ----------------------------------------
252         function set.Target( plugin , val )
253             plugin.Target = val;
254             updateTarget(plugin);
255         end
256         function updateTarget( plugin ) % funzione di aggiornamento per le cuffie target
257             plugin.Fs = getSampleRate(plugin);
258             switch plugin.Target
259                 case TargetEnum.None
260                     plugin.filt0_Target.a = [1,0,0]; plugin.filt0_Target.b = [1,0,0];
261                     plugin.filt1_Target.a = [1,0,0]; plugin.filt1_Target.b = [1,0,0];
262                     plugin.filt2_Target.a = [1,0,0]; plugin.filt2_Target.b = [1,0,0];
263                     plugin.filt3_Target.a = [1,0,0]; plugin.filt3_Target.b = [1,0,0];
264                     plugin.filt4_Target.a = [1,0,0]; plugin.filt4_Target.b = [1,0,0];
265                     plugin.filt5_Target.a = [1,0,0]; plugin.filt5_Target.b = [1,0,0];
266                     plugin.filt6_Target.a = [1,0,0]; plugin.filt6_Target.b = [1,0,0];
267                     plugin.filt7_Target.a = [1,0,0]; plugin.filt7_Target.b = [1,0,0];
268                     plugin.filt8_Target.a = [1,0,0]; plugin.filt8_Target.b = [1,0,0];
269                     plugin.filt9_Target.a = [1,0,0]; plugin.filt9_Target.b = [1,0,0];
270                 case TargetEnum.AKGK240MKII
271                     [plugin.filt0_Target.a,plugin.filt0_Target.b] = biquad_LS_filterCoefficients(plugin.Fs, 75, 4 , 0.71);
272                     [plugin.filt1_Target.a,plugin.filt1_Target.b] = biquad_LS_filterCoefficients(plugin.Fs, 105, 5.5, 0.71);
273
274                     [plugin.filt2_Target.a,plugin.filt2_Target.b] = biquad_PK_filterCoefficients(plugin.Fs, 167, -4.7, 0.4 )
                            ;
275                     [plugin.filt3_Target.a,plugin.filt3_Target.b] = biquad_PK_filterCoefficients(plugin.Fs, 1170, -1.7, 1.3
                            );
276                     [plugin.filt4_Target.a,plugin.filt4_Target.b] = biquad_PK_filterCoefficients(plugin.Fs, 1650, 5 , 1.4 );
277                     [plugin.filt5_Target.a,plugin.filt5_Target.b] = biquad_PK_filterCoefficients(plugin.Fs, 2170, -1.1, 4.0
                            );
278                     [plugin.filt6_Target.a,plugin.filt6_Target.b] = biquad_PK_filterCoefficients(plugin.Fs, 2700, -4.5, 2.5
                            );
279                     [plugin.filt7_Target.a,plugin.filt7_Target.b] = biquad_PK_filterCoefficients(plugin.Fs, 3000, -0.7, 6.0
                            );
280                     [plugin.filt8_Target.a,plugin.filt8_Target.b] = biquad_PK_filterCoefficients(plugin.Fs, 4100, 4 , 1.4 );
281                     [plugin.filt9_Target.a,plugin.filt9_Target.b] = biquad_PK_filterCoefficients(plugin.Fs, 6850, -4.9, 3.0
                            );
282                 case TargetEnum.AKGK701
283                     [plugin.filt0_Target.a,plugin.filt0_Target.b] = biquad_LS_filterCoefficients(plugin.Fs, 105, 5.5, 0.71);
284
285                     [plugin.filt1_Target.a,plugin.filt1_Target.b] = biquad_PK_filterCoefficients(plugin.Fs, 200, -2.7, 0.4 )
                            ;
286                     [plugin.filt2_Target.a,plugin.filt2_Target.b] = biquad_PK_filterCoefficients(plugin.Fs, 730, 3.2, 1.4 );
287                     [plugin.filt3_Target.a,plugin.filt3_Target.b] = biquad_PK_filterCoefficients(plugin.Fs, 1300, 2.9, 1.5 )
                            ;
288
289                     [plugin.filt4_Target.a,plugin.filt4_Target.b] = biquad_HS_filterCoefficients(plugin.Fs, 2000, 4 , 0.71);
290
291                     [plugin.filt5_Target.a,plugin.filt5_Target.b] = biquad_PK_filterCoefficients(plugin.Fs, 2450, -5.1, 2.0
                            );
292                     [plugin.filt6_Target.a,plugin.filt6_Target.b] = biquad_PK_filterCoefficients(plugin.Fs, 3500, 3.0, 1.4 )
                            ;
293                     [plugin.filt7_Target.a,plugin.filt7_Target.b] = biquad_PK_filterCoefficients(plugin.Fs, 5850, -6.5, 2.5
                            );
294                     [plugin.filt8_Target.a,plugin.filt8_Target.b] = biquad_PK_filterCoefficients(plugin.Fs, 7500, -1.2, 6.0
                            );
295
296                     [plugin.filt9_Target.a,plugin.filt9_Target.b] = biquad_HS_filterCoefficients(plugin.Fs,10000, -4 , 0.71)
                            ;
297                 case TargetEnum.BeyerDynamicDT77080ohm
298                     [plugin.filt0_Target.a,plugin.filt0_Target.b] = biquad_LS_filterCoefficients(plugin.Fs, 100, 10.5, 1 );
299
300                     [plugin.filt1_Target.a,plugin.filt1_Target.b] = biquad_PK_filterCoefficients(plugin.Fs, 100, -7.0, 0.4 )
                            ;
301                     [plugin.filt2_Target.a,plugin.filt2_Target.b] = biquad_PK_filterCoefficients(plugin.Fs, 3500, 4.5, 0.7 )
                            ;
302                     [plugin.filt3_Target.a,plugin.filt3_Target.b] = biquad_PK_filterCoefficients(plugin.Fs, 2000, -3.5, 1.4
                            );
303                     [plugin.filt4_Target.a,plugin.filt4_Target.b] = biquad_HS_filterCoefficients(plugin.Fs, 2800, -1.3, 3.0
                            );
304
305                     plugin.filt5_Target.a = [1,0,0]; plugin.filt5_Target.b = [1,0,0];
306                     plugin.filt6_Target.a = [1,0,0]; plugin.filt6_Target.b = [1,0,0];
307                     plugin.filt7_Target.a = [1,0,0]; plugin.filt7_Target.b = [1,0,0];
308                     plugin.filt8_Target.a = [1,0,0]; plugin.filt8_Target.b = [1,0,0];
```

```
309                     plugin.filt9_Target.a = [1,0,0]; plugin.filt9_Target.b = [1,0,0];
310                 case TargetEnum.BeyerDynamicDT990250ohm
311                     [plugin.filt0_Target.a,plugin.filt0_Target.b] = biquad_LS_filterCoefficients(plugin.Fs, 105, 5 , 0.71);
312
313                     [plugin.filt1_Target.a,plugin.filt1_Target.b] = biquad_PK_filterCoefficients(plugin.Fs, 100, -5.2, 0.4 )
                            ;
314                     [plugin.filt2_Target.a,plugin.filt2_Target.b] = biquad_PK_filterCoefficients(plugin.Fs, 650, 1.2, 1.8 );
315                     [plugin.filt3_Target.a,plugin.filt3_Target.b] = biquad_PK_filterCoefficients(plugin.Fs, 1900, -0.6, 3 );
316                     [plugin.filt4_Target.a,plugin.filt4_Target.b] = biquad_PK_filterCoefficients(plugin.Fs, 2600, -0.9, 5 );
317                     [plugin.filt5_Target.a,plugin.filt5_Target.b] = biquad_PK_filterCoefficients(plugin.Fs, 3150, 2.5, 1.41)
                            ;
318                     [plugin.filt6_Target.a,plugin.filt6_Target.b] = biquad_PK_filterCoefficients(plugin.Fs, 3800, -0.9, 5 );
319                     [plugin.filt7_Target.a,plugin.filt7_Target.b] = biquad_PK_filterCoefficients(plugin.Fs, 8400, -2.5, 6 );
320                     [plugin.filt8_Target.a,plugin.filt8_Target.b] = biquad_PK_filterCoefficients(plugin.Fs, 9400, -3 , 6 );
321
322                     [plugin.filt9_Target.a,plugin.filt9_Target.b] = biquad_HS_filterCoefficients(plugin.Fs, 9000, -6 , 0.71
                            );
323                 case TargetEnum.BeyerDynamicT1
324                     [plugin.filt0_Target.a,plugin.filt0_Target.b] = biquad_LS_filterCoefficients(plugin.Fs, 105, 5.5, 0.71);
325
326                     [plugin.filt1_Target.a,plugin.filt1_Target.b] = biquad_PK_filterCoefficients(plugin.Fs, 200, -3.2, 0.4 )
                            ;
327                     [plugin.filt2_Target.a,plugin.filt2_Target.b] = biquad_PK_filterCoefficients(plugin.Fs, 500, 1 , 0.5 );
328
329                     [plugin.filt3_Target.a,plugin.filt3_Target.b] = biquad_HS_filterCoefficients(plugin.Fs, 1700, 7 , 0.5 );
330
331                     [plugin.filt4_Target.a,plugin.filt4_Target.b] = biquad_PK_filterCoefficients(plugin.Fs, 2000, -1 , 4 );
332                     [plugin.filt5_Target.a,plugin.filt5_Target.b] = biquad_PK_filterCoefficients(plugin.Fs, 3650, -1 , 3 );
333                     [plugin.filt6_Target.a,plugin.filt6_Target.b] = biquad_PK_filterCoefficients(plugin.Fs, 6080, -5.3, 3 );
334                     [plugin.filt7_Target.a,plugin.filt7_Target.b] = biquad_PK_filterCoefficients(plugin.Fs, 8450,-11.6, 2.5
                            );
335
336                     [plugin.filt8_Target.a,plugin.filt8_Target.b] = biquad_LS_filterCoefficients(plugin.Fs, 9000, -4 , 0.71)
                            ;
337
338                     [plugin.filt9_Target.a,plugin.filt9_Target.b] = biquad_PK_filterCoefficients(plugin.Fs,11900, -5 , 4 );
339                 case TargetEnum.FocalStellia
340                     [plugin.filt0_Target.a,plugin.filt0_Target.b] = biquad_PK_filterCoefficients(plugin.Fs, 125, -5 , 1.1 );
341                     [plugin.filt1_Target.a,plugin.filt1_Target.b] = biquad_PK_filterCoefficients(plugin.Fs, 280, 1.4, 1.2 );
342                     [plugin.filt2_Target.a,plugin.filt2_Target.b] = biquad_PK_filterCoefficients(plugin.Fs, 1150, -1.7, 1.4
                            );
343                     [plugin.filt3_Target.a,plugin.filt3_Target.b] = biquad_PK_filterCoefficients(plugin.Fs, 1870, -2.9, 3 );
344                     [plugin.filt4_Target.a,plugin.filt4_Target.b] = biquad_PK_filterCoefficients(plugin.Fs, 2800, -2.1, 5 );
345                     [plugin.filt5_Target.a,plugin.filt5_Target.b] = biquad_PK_filterCoefficients(plugin.Fs, 3600, -3.6, 7 );
346                     [plugin.filt6_Target.a,plugin.filt6_Target.b] = biquad_PK_filterCoefficients(plugin.Fs, 4500, 5 , 0.9 );
347                     [plugin.filt7_Target.a,plugin.filt7_Target.b] = biquad_PK_filterCoefficients(plugin.Fs, 6000, -4.6, 3.2
                            );
348                     [plugin.filt8_Target.a,plugin.filt8_Target.b] = biquad_PK_filterCoefficients(plugin.Fs, 8000, -3 , 4 );
349
350                     [plugin.filt9_Target.a,plugin.filt9_Target.b] = biquad_HS_filterCoefficients(plugin.Fs, 9000, 6 , 0.71);
351                 case TargetEnum.SennheiserHD600
352                     [plugin.filt0_Target.a,plugin.filt0_Target.b] = biquad_LS_filterCoefficients(plugin.Fs, 25, 6 , 0.71);
353
354                     [plugin.filt1_Target.a,plugin.filt1_Target.b] = biquad_PK_filterCoefficients(plugin.Fs, 150, -2.9, 0.5 )
                            ;
355                     [plugin.filt2_Target.a,plugin.filt2_Target.b] = biquad_PK_filterCoefficients(plugin.Fs, 1330, -2 , 1.4 )
                            ;
356                     [plugin.filt3_Target.a,plugin.filt3_Target.b] = biquad_PK_filterCoefficients(plugin.Fs, 3150, -3.1, 2.2)
                            ;
357                     [plugin.filt4_Target.a,plugin.filt4_Target.b] = biquad_PK_filterCoefficients(plugin.Fs, 3500, -1.0, 4 );
358                     [plugin.filt5_Target.a,plugin.filt5_Target.b] = biquad_PK_filterCoefficients(plugin.Fs, 5000, -1.2, 6 );
359                     [plugin.filt6_Target.a,plugin.filt6_Target.b] = biquad_PK_filterCoefficients(plugin.Fs, 5850, -2.2, 4.5
                            );
360                     [plugin.filt7_Target.a,plugin.filt7_Target.b] = biquad_PK_filterCoefficients(plugin.Fs, 5600, 2 , 0.71);
361                     [plugin.filt8_Target.a,plugin.filt8_Target.b] = biquad_PK_filterCoefficients(plugin.Fs, 7710, -2.5, 5 );
362
363                     [plugin.filt9_Target.a,plugin.filt9_Target.b] = biquad_HS_filterCoefficients(plugin.Fs,13000, -3 , 0.71)
                            ;
364                 case TargetEnum.SennheiserHD650
365                     [plugin.filt0_Target.a,plugin.filt0_Target.b] = biquad_PK_filterCoefficients(plugin.Fs, 21, 4.5, 1 );
366                     [plugin.filt1_Target.a,plugin.filt1_Target.b] = biquad_PK_filterCoefficients(plugin.Fs, 97, -1.8, 0.7 );
367                     [plugin.filt2_Target.a,plugin.filt2_Target.b] = biquad_PK_filterCoefficients(plugin.Fs, 215, -0.8, 1.1 )
                            ;
368                     [plugin.filt3_Target.a,plugin.filt3_Target.b] = biquad_PK_filterCoefficients(plugin.Fs, 600, 0.6, 1 );
369                     [plugin.filt4_Target.a,plugin.filt4_Target.b] = biquad_PK_filterCoefficients(plugin.Fs, 1300, -1.1, 1.6
                            );
370
371                     [plugin.filt5_Target.a,plugin.filt5_Target.b] = biquad_HS_filterCoefficients(plugin.Fs, 1900, 3 , 0.71);
372
373                     [plugin.filt6_Target.a,plugin.filt6_Target.b] = biquad_PK_filterCoefficients(plugin.Fs, 2900, -2 , 3 );
374                     [plugin.filt7_Target.a,plugin.filt7_Target.b] = biquad_PK_filterCoefficients(plugin.Fs, 3400, -2.1, 3 );
375                     [plugin.filt8_Target.a,plugin.filt8_Target.b] = biquad_PK_filterCoefficients(plugin.Fs, 5500, -1.7, 3 );
376
377                     [plugin.filt9_Target.a,plugin.filt9_Target.b] = biquad_HS_filterCoefficients(plugin.Fs,11000, -3 , 0.71)
                            ;
378                 case TargetEnum.SennheiserHD800s
379                     [plugin.filt0_Target.a,plugin.filt0_Target.b] = biquad_LS_filterCoefficients(plugin.Fs, 105, 5.5, 0.71);
380
381                     [plugin.filt1_Target.a,plugin.filt1_Target.b] = biquad_PK_filterCoefficients(plugin.Fs, 200, -4.7, 0.3 )
                            ;
382                     [plugin.filt2_Target.a,plugin.filt2_Target.b] = biquad_PK_filterCoefficients(plugin.Fs, 1080, -2.6, 1.7
```

```
                    );
                [plugin.filt3_Target.a,plugin.filt3_Target.b] = biquad_PK_filterCoefficients(plugin.Fs, 1380, 2 , 1.4);
                [plugin.filt4_Target.a,plugin.filt4_Target.b] = biquad_PK_filterCoefficients(plugin.Fs, 2750, -2.1, 3 );
                [plugin.filt5_Target.a,plugin.filt5_Target.b] = biquad_PK_filterCoefficients(plugin.Fs, 3200, 1 , 1.4 );
                [plugin.filt6_Target.a,plugin.filt6_Target.b] = biquad_PK_filterCoefficients(plugin.Fs, 4000, -1.3, 6 );
                [plugin.filt7_Target.a,plugin.filt7_Target.b] = biquad_PK_filterCoefficients(plugin.Fs, 5500, -5.3, 3.5)
                    ;
                [plugin.filt8_Target.a,plugin.filt8_Target.b] = biquad_PK_filterCoefficients(plugin.Fs, 6300, -2.1, 5 );

                [plugin.filt9_Target.a,plugin.filt9_Target.b] = biquad_HS_filterCoefficients(plugin.Fs,10000, -5 , 0.71)
                    ;
            case TargetEnum.ShureSRH1540
                [plugin.filt0_Target.a,plugin.filt0_Target.b] = biquad_PK_filterCoefficients(plugin.Fs, 27, -4.2, 0.6 );
                [plugin.filt1_Target.a,plugin.filt1_Target.b] = biquad_PK_filterCoefficients(plugin.Fs, 95, -8.6, 0.5 );

                [plugin.filt2_Target.a,plugin.filt2_Target.b] = biquad_LS_filterCoefficients(plugin.Fs, 105, 5.5, 0.71);

                [plugin.filt3_Target.a,plugin.filt3_Target.b] = biquad_PK_filterCoefficients(plugin.Fs, 2800, -2.5, 1.3
                    );
                [plugin.filt4_Target.a,plugin.filt4_Target.b] = biquad_PK_filterCoefficients(plugin.Fs, 4700, 1.4, 2 );
                [plugin.filt5_Target.a,plugin.filt5_Target.b] = biquad_PK_filterCoefficients(plugin.Fs, 6000, -2.6, 5 );

                [plugin.filt6_Target.a,plugin.filt6_Target.b] = biquad_HS_filterCoefficients(plugin.Fs, 9000, -1 , 0.71)
                    ;

                plugin.filt7_Target.a = [1,0,0]; plugin.filt7_Target.b = [1,0,0];
                plugin.filt8_Target.a = [1,0,0]; plugin.filt8_Target.b = [1,0,0];
                plugin.filt9_Target.a = [1,0,0]; plugin.filt9_Target.b = [1,0,0];
        end
    end
    function set.Bypass(plugin,val)
        plugin.Bypass = val;
    end
    function val = get.Compensation(plugin)
        val = plugin.Compensation;
    end
    function set.Compensation(plugin,val)
        plugin.Compensation = val;
        updateCompensationCurve(plugin);
    end
    function updateCompensationCurve(plugin) % funzione di aggiornamento per la curva di compensazione
        if plugin.Compensation
            [plugin.Harman_1.a, plugin.Harman_1.b] = biquad_LS_filterCoefficients(plugin.Fs, 68, 4.95 , 1.02);
            [plugin.Harman_2.a, plugin.Harman_2.b] = biquad_PK_filterCoefficients(plugin.Fs, 220, -1.50 , 0.64);
            [plugin.Harman_3.a, plugin.Harman_3.b] = biquad_PK_filterCoefficients(plugin.Fs, 1245, -2.70 , 2.38);
            [plugin.Harman_4.a, plugin.Harman_4.b] = biquad_PK_filterCoefficients(plugin.Fs, 2165, 2.40 , 4.12);
            [plugin.Harman_5.a, plugin.Harman_5.b] = biquad_PK_filterCoefficients(plugin.Fs, 4686, 5.90 , 2.41);
            [plugin.Harman_6.a, plugin.Harman_6.b] = biquad_PK_filterCoefficients(plugin.Fs, 5348, 2.00 , 0.98);
            [plugin.Harman_7.a, plugin.Harman_7.b] = biquad_PK_filterCoefficients(plugin.Fs, 8276, 5.10 , 1.55);
            [plugin.Harman_8.a, plugin.Harman_8.b] = biquad_PK_filterCoefficients(plugin.Fs, 11312, 4.52 , 0.71);
            [plugin.Harman_9.a, plugin.Harman_9.b] = biquad_HS_filterCoefficients(plugin.Fs, 12500, 0.96 , 0.70);
            [plugin.Harman_10.a, plugin.Harman_10.b] = biquad_PK_filterCoefficients(plugin.Fs, 13181, 3.50 , 0.60);
            [plugin.Harman_11.a, plugin.Harman_11.b] = biquad_PK_filterCoefficients(plugin.Fs, 18477, -9.50 , 0.09);
        else
            plugin.Harman_1.a = [1,0,0];plugin.Harman_1.b = [1,0,0];
            plugin.Harman_2.a = [1,0,0];plugin.Harman_2.b = [1,0,0];
            plugin.Harman_3.a = [1,0,0];plugin.Harman_3.b = [1,0,0];
            plugin.Harman_4.a = [1,0,0];plugin.Harman_4.b = [1,0,0];
            plugin.Harman_5.a = [1,0,0];plugin.Harman_5.b = [1,0,0];
            plugin.Harman_6.a = [1,0,0];plugin.Harman_6.b = [1,0,0];
            plugin.Harman_7.a = [1,0,0];plugin.Harman_7.b = [1,0,0];
            plugin.Harman_8.a = [1,0,0];plugin.Harman_8.b = [1,0,0];
            plugin.Harman_9.a = [1,0,0];plugin.Harman_9.b = [1,0,0];
            plugin.Harman_10.a = [1,0,0];plugin.Harman_10.b = [1,0,0];
            plugin.Harman_11.a = [1,0,0];plugin.Harman_11.b = [1,0,0];
        end
    end
  end
end
```

Classi enumeratrici per gestire la scelta di cuffie Target e Monitor

```
    classdef MonitorEnum < int8
    enumeration
        None (0)
        AKGK240MKII (1)
        BeyerDynamicDT990250ohm (2)
        BeyerDynamicT1 (3)
        SennheiserHD600 (4)
    end
    methods (Static)
        function yes = addClassNameToEnumNames
            yes = true;
        end
    end
end
```

```
classdef TargetEnum < int8
    enumeration
        None (0)
        AKGK240MKII (1)
        AKGK701 (2)
        BeyerDynamicDT77080ohm (3)
        BeyerDynamicDT990250ohm (4)
        BeyerDynamicT1 (5)
        FocalStellia (6)
        SennheiserHD600 (7)
        SennheiserHD650 (8)
        SennheiserHD800s (9)
        ShureSRH1540 (10)
    end
    methods (Static)
        function yes = addClassNameToEnumNames
            yes = true;
        end
    end
end
```

Classi per calcolare i coefficienti di filtri Low Shelf, High Shelf e Peaking biquadratici e relativa funzione per applicarli

```
function [a,b] = biquad_LS_filterCoefficients(Fs,f0,dBGain,Q)
    A = 10^(dBGain/40);
    w0 = (2*pi*f0) /Fs;
    alpha = sin(w0) / (2*Q);
    a = [ (A+1) + (A-1)*cos(w0) + 2*alpha*sqrt(A),...
          -2*( (A-1) + (A+1)*cos(w0)),...
          (A+1) + (A-1)*cos(w0) - 2*alpha*sqrt(A)];
    b =[ A*( (A+1) - (A-1)*cos(w0) + 2*alpha*sqrt(A)),...
         2*A*( (A-1) - (A+1)*cos(w0) ),...
         A*( (A+1) - (A-1)*cos(w0) - 2*alpha*sqrt(A))];
end
```

```
function [a,b] = biquad_HS_filterCoefficients(Fs,f0,dBGain,Q)
    A = 10^(dBGain/40);
    w0 = (2*pi*f0) / Fs;
    alpha = sin(w0) / (2*Q);
    a = [ (A+1) - (A-1)*cos(w0) + 2*alpha*sqrt(A),...
          2*( (A-1) - (A+1)*cos(w0)),...
          (A+1) - (A-1)*cos(w0) - 2*alpha*sqrt(A)];
    b = [ A*((A+1)+(A-1)*cos(w0)+2*alpha*sqrt(A)),...
          -2*A*((A-1)+(A+1)*cos(w0)),...
          A*( (A+1)+ (A-1)*cos(w0) - 2*alpha*sqrt(A))];
end
```

```
function [a,b] = biquad_PK_filterCoefficients(Fs,f0,dBGain,Q)
    A = 10^(dBGain/40);
    w0 = (2*pi*f0)/Fs;
    alpha = sin(w0)/(2*Q);
    b = [ 1 + alpha*A, -2*cos(w0), 1 - alpha*A];
    a = [ 1 + alpha/A, -2*cos(w0), 1 - alpha/A] ;
end
```

```
function [y,w] = processBiquad(x, filt, ch)
    [y,w] = filter(filt.a,filt.b,...
                   x,...
                   filt.w(:,ch));
end
```

B HeadPhoneSimulatroIR

Segue il codice della classe `HeadPhoneSimulatorIR`

```
classdef HeadPhoneSimulator_IR < audioPlugin & matlab.System %#codegen
    properties % variabili globali accessibili e modificabili
        Monitor = MonitorEnum.AKGK240MKII;
        Target = TargetEnum.AKGK701;
        bypass = false;
    end
    properties(Constant) % interfaccia utente
        PluginInterface = audioPluginInterface( ...
            'PluginName','HeadPhoneSimulator_IR','InputChannels',2,'OutputChannels',2,...
            'VendorName','BenDaTek_Distributions','VendorVersion','0.0.2',...
            audioPluginParameter('Monitor','DisplayName','Monitor','Mapping',...
                                 {'enum','AKG K240 MKII','Beyerdynamic DT 990250',...
                                     'Beyerdynamic T1','Sennheiser HD600'},'Layout',[1,1]),...
            audioPluginParameter('Target','DisplayName','Target','Mapping', ...
                                 {'enum','AKG K240 MKII','AKG K701','Beyerdynamic DT 770 80 ohm',...
                                     'Beyerdynamic DT 990 250 ohm','Beyerdynamic T1','Focal Stellia',...
                                     'Sennheiser HD 600','Sennheiser HD 650',...
                                     'Sennheiser HD 800s','Shure SR H1540'},'Layout',[3,1]),...
            audioPluginParameter('bypass','DisplayName','bypass',...
                                  'Style','vrocker','Layout',[1 2; 3 2]),...
            audioPluginGridLayout('RowHeight', [60,30,60,30],...
                                  'ColumnWidth',[300 60],...
                                  'Padding', [10 10 10 10]));
    end
    properties(Nontunable) % variabili globali non modificabili
        ImpulseResponse0 = audioread('IRs/BypassResponse.wav').';

        ImpulseResponse1 = audioread('IRs/M_AkgK240_T_AkgK701.wav').';
        ImpulseResponse2 = audioread('IRs/M_AkgK240_T_BeyerdynamicDT770.wav').';
        ImpulseResponse3 = audioread('IRs/M_AkgK240_T_BeyerdynamicDT990.wav').';
        ImpulseResponse4 = audioread('IRs/M_AkgK240_T_BeyerdynamicT1.wav').';
        ImpulseResponse5 = audioread('IRs/M_AkgK240_T_FocalStellia.wav').';
        ImpulseResponse6 = audioread('IRs/M_AkgK240_T_SennheiserHD600.wav').';
        ImpulseResponse7 = audioread('IRs/M_AkgK240_T_SennheiserHD650.wav').';
        ImpulseResponse8 = audioread('IRs/M_AkgK240_T_SennheiserHD800s.wav').';
        ImpulseResponse9 = audioread('IRs/M_AkgK240_T_ShureSRH1540.wav').';
        ImpulseResponse10 = audioread('IRs/M_BeyerdynamicDT990_T_AkgK240.wav').';
        ImpulseResponse11 = audioread('IRs/M_BeyerdynamicDT990_T_AkgK701.wav').';
        ImpulseResponse12 = audioread('IRs/M_BeyerdynamicDT990_T_BeyerdynamicDT770.wav').';
        ImpulseResponse13 = audioread('IRs/M_BeyerdynamicDT990_T_BeyerdynamicT1.wav').';
        ImpulseResponse14 = audioread('IRs/M_BeyerdynamicDT990_T_FocalStellia.wav').';
        ImpulseResponse15 = audioread('IRs/M_BeyerdynamicDT990_T_SennheiserHD600.wav').';
        ImpulseResponse16 = audioread('IRs/M_BeyerdynamicDT990_T_SennheiserHD650.wav').';
        ImpulseResponse17 = audioread('IRs/M_BeyerdynamicDT990_T_SennheiserHD800s.wav').';
        ImpulseResponse18 = audioread('IRs/M_BeyerdynamicDT990_T_ShureSRH1540.wav').';
        ImpulseResponse19 = audioread('IRs/M_BeyerdynamicT1_T_AkgK240.wav').';
        ImpulseResponse20 = audioread('IRs/M_BeyerdynamicT1_T_AkgK701.wav').';
        ImpulseResponse21 = audioread('IRs/M_BeyerdynamicT1_T_BeyerdynamicDT770.wav').';
        ImpulseResponse22 = audioread('IRs/M_BeyerdynamicT1_T_BeyerdynamicDT990.wav').';
        ImpulseResponse23 = audioread('IRs/M_BeyerdynamicT1_T_FocalStellia.wav').';
        ImpulseResponse24 = audioread('IRs/M_BeyerdynamicT1_T_SennheiserHD600.wav').';
        ImpulseResponse25 = audioread('IRs/M_BeyerdynamicT1_T_SennheiserHD650.wav').';
        ImpulseResponse26 = audioread('IRs/M_BeyerdynamicT1_T_SennheiserHD800s.wav').';
        ImpulseResponse27 = audioread('IRs/M_BeyerdynamicT1_T_ShureSRH1540.wav').';
        ImpulseResponse28 = audioread('IRs/M_SennheiserHD600_T_AkgK240.wav').';
        ImpulseResponse29 = audioread('IRs/M_SennheiserHD600_T_AkgK701.wav').';
        ImpulseResponse30 = audioread('IRs/M_SennheiserHD600_T_BeyerdynamicDT770.wav').';
        ImpulseResponse31 = audioread('IRs/M_SennheiserHD600_T_BeyerdynamicDT990.wav').';
        ImpulseResponse32 = audioread('IRs/M_SennheiserHD600_T_BeyerdynamicT1.wav').';
        ImpulseResponse33 = audioread('IRs/M_SennheiserHD600_T_FocalStellia.wav').';
        ImpulseResponse34 = audioread('IRs/M_SennheiserHD600_T_SennheiserHD650.wav').';
        ImpulseResponse35 = audioread('IRs/M_SennheiserHD600_T_SennheiserHD800s.wav').';
        ImpulseResponse36 = audioread('IRs/M_SennheiserHD600_T_ShureSRH1540.wav').';

        frameSize = 1024; %
    end
    properties(Access = private) % variabili globali private
        FIR0;

        FIR1; FIR10; FIR19; FIR28;
        FIR2; FIR11; FIR20; FIR29;
        FIR3; FIR12; FIR21; FIR30;
        FIR4; FIR13; FIR22; FIR31;
        FIR5; FIR14; FIR23; FIR32;
        FIR6; FIR15; FIR24; FIR33;
        FIR7; FIR16; FIR25; FIR34;
        FIR8; FIR17; FIR26; FIR35;
        FIR9; FIR18; FIR27; FIR36;

        currentFIR;
    end
    methods(Access = protected)
        function output = stepImpl(plugin,input) % funzione di elaborazione
            output = input;
            if ~plugin.bypass
                output = step(plugin.currentFIR,input);
```

```
87            end
88        end
89        function setupImpl(plugin, u) % funzione di setup
90            plugin.FIR0 = dsp.FrequencyDomainFIRFilter('Numerator', plugin.ImpulseResponse0 ,'PartitionForReducedLatency',
                  true, 'PartitionLength', plugin.frameSize);
91
92            plugin.FIR1 = dsp.FrequencyDomainFIRFilter('Numerator', plugin.ImpulseResponse1 ,'PartitionForReducedLatency',
                  true, 'PartitionLength', plugin.frameSize);
93            plugin.FIR2 = dsp.FrequencyDomainFIRFilter('Numerator', plugin.ImpulseResponse2 ,'PartitionForReducedLatency',
                  true, 'PartitionLength', plugin.frameSize);
94            plugin.FIR3 = dsp.FrequencyDomainFIRFilter('Numerator', plugin.ImpulseResponse3 ,'PartitionForReducedLatency',
                  true, 'PartitionLength', plugin.frameSize);
95            plugin.FIR4 = dsp.FrequencyDomainFIRFilter('Numerator', plugin.ImpulseResponse4 ,'PartitionForReducedLatency',
                  true, 'PartitionLength', plugin.frameSize);
96            plugin.FIR5 = dsp.FrequencyDomainFIRFilter('Numerator', plugin.ImpulseResponse5 ,'PartitionForReducedLatency',
                  true, 'PartitionLength', plugin.frameSize);
97            plugin.FIR6 = dsp.FrequencyDomainFIRFilter('Numerator', plugin.ImpulseResponse6 ,'PartitionForReducedLatency',
                  true, 'PartitionLength', plugin.frameSize);
98            plugin.FIR7 = dsp.FrequencyDomainFIRFilter('Numerator', plugin.ImpulseResponse7 ,'PartitionForReducedLatency',
                  true, 'PartitionLength', plugin.frameSize);
99            plugin.FIR8 = dsp.FrequencyDomainFIRFilter('Numerator', plugin.ImpulseResponse8 ,'PartitionForReducedLatency',
                  true, 'PartitionLength', plugin.frameSize);
100           plugin.FIR9 = dsp.FrequencyDomainFIRFilter('Numerator', plugin.ImpulseResponse9 ,'PartitionForReducedLatency',
                  true, 'PartitionLength', plugin.frameSize);
101
102           plugin.FIR10 = dsp.FrequencyDomainFIRFilter('Numerator', plugin.ImpulseResponse10 ,'PartitionForReducedLatency'
                  , true, 'PartitionLength', plugin.frameSize);
103           plugin.FIR11 = dsp.FrequencyDomainFIRFilter('Numerator', plugin.ImpulseResponse11 ,'PartitionForReducedLatency'
                  , true, 'PartitionLength', plugin.frameSize);
104           plugin.FIR12 = dsp.FrequencyDomainFIRFilter('Numerator', plugin.ImpulseResponse12 ,'PartitionForReducedLatency'
                  , true, 'PartitionLength', plugin.frameSize);
105           plugin.FIR13 = dsp.FrequencyDomainFIRFilter('Numerator', plugin.ImpulseResponse13 ,'PartitionForReducedLatency'
                  , true, 'PartitionLength', plugin.frameSize);
106           plugin.FIR14 = dsp.FrequencyDomainFIRFilter('Numerator', plugin.ImpulseResponse14 ,'PartitionForReducedLatency'
                  , true, 'PartitionLength', plugin.frameSize);
107           plugin.FIR15 = dsp.FrequencyDomainFIRFilter('Numerator', plugin.ImpulseResponse15 ,'PartitionForReducedLatency'
                  , true, 'PartitionLength', plugin.frameSize);
108           plugin.FIR16 = dsp.FrequencyDomainFIRFilter('Numerator', plugin.ImpulseResponse15 ,'PartitionForReducedLatency'
                  , true, 'PartitionLength', plugin.frameSize);
109           plugin.FIR17 = dsp.FrequencyDomainFIRFilter('Numerator', plugin.ImpulseResponse15 ,'PartitionForReducedLatency'
                  , true, 'PartitionLength', plugin.frameSize);
110           plugin.FIR18 = dsp.FrequencyDomainFIRFilter('Numerator', plugin.ImpulseResponse15 ,'PartitionForReducedLatency'
                  , true, 'PartitionLength', plugin.frameSize);
111
112           plugin.FIR19 = dsp.FrequencyDomainFIRFilter('Numerator', plugin.ImpulseResponse19 ,'PartitionForReducedLatency'
                  , true, 'PartitionLength', plugin.frameSize);
113           plugin.FIR20 = dsp.FrequencyDomainFIRFilter('Numerator', plugin.ImpulseResponse20 ,'PartitionForReducedLatency'
                  , true, 'PartitionLength', plugin.frameSize);
114           plugin.FIR21 = dsp.FrequencyDomainFIRFilter('Numerator', plugin.ImpulseResponse21 ,'PartitionForReducedLatency'
                  , true, 'PartitionLength', plugin.frameSize);
115           plugin.FIR22 = dsp.FrequencyDomainFIRFilter('Numerator', plugin.ImpulseResponse22 ,'PartitionForReducedLatency'
                  , true, 'PartitionLength', plugin.frameSize);
116           plugin.FIR23 = dsp.FrequencyDomainFIRFilter('Numerator', plugin.ImpulseResponse23 ,'PartitionForReducedLatency'
                  , true, 'PartitionLength', plugin.frameSize);
117           plugin.FIR24 = dsp.FrequencyDomainFIRFilter('Numerator', plugin.ImpulseResponse24 ,'PartitionForReducedLatency'
                  , true, 'PartitionLength', plugin.frameSize);
118           plugin.FIR25 = dsp.FrequencyDomainFIRFilter('Numerator', plugin.ImpulseResponse25 ,'PartitionForReducedLatency'
                  , true, 'PartitionLength', plugin.frameSize);
119           plugin.FIR26 = dsp.FrequencyDomainFIRFilter('Numerator', plugin.ImpulseResponse26 ,'PartitionForReducedLatency'
                  , true, 'PartitionLength', plugin.frameSize);
120           plugin.FIR27 = dsp.FrequencyDomainFIRFilter('Numerator', plugin.ImpulseResponse27 ,'PartitionForReducedLatency'
                  , true, 'PartitionLength', plugin.frameSize);
121
122           plugin.FIR28 = dsp.FrequencyDomainFIRFilter('Numerator', plugin.ImpulseResponse28 ,'PartitionForReducedLatency'
                  , true, 'PartitionLength', plugin.frameSize);
123           plugin.FIR29 = dsp.FrequencyDomainFIRFilter('Numerator', plugin.ImpulseResponse29 ,'PartitionForReducedLatency'
                  , true, 'PartitionLength', plugin.frameSize);
124           plugin.FIR30 = dsp.FrequencyDomainFIRFilter('Numerator', plugin.ImpulseResponse30 ,'PartitionForReducedLatency'
                  , true, 'PartitionLength', plugin.frameSize);
125           plugin.FIR31 = dsp.FrequencyDomainFIRFilter('Numerator', plugin.ImpulseResponse31 ,'PartitionForReducedLatency'
                  , true, 'PartitionLength', plugin.frameSize);
126           plugin.FIR32 = dsp.FrequencyDomainFIRFilter('Numerator', plugin.ImpulseResponse32 ,'PartitionForReducedLatency'
                  , true, 'PartitionLength', plugin.frameSize);
127           plugin.FIR33 = dsp.FrequencyDomainFIRFilter('Numerator', plugin.ImpulseResponse33 ,'PartitionForReducedLatency'
                  , true, 'PartitionLength', plugin.frameSize);
128           plugin.FIR34 = dsp.FrequencyDomainFIRFilter('Numerator', plugin.ImpulseResponse34 ,'PartitionForReducedLatency'
                  , true, 'PartitionLength', plugin.frameSize);
129           plugin.FIR35 = dsp.FrequencyDomainFIRFilter('Numerator', plugin.ImpulseResponse35 ,'PartitionForReducedLatency'
                  , true, 'PartitionLength', plugin.frameSize);
130           plugin.FIR36 = dsp.FrequencyDomainFIRFilter('Numerator', plugin.ImpulseResponse36 ,'PartitionForReducedLatency'
                  , true, 'PartitionLength', plugin.frameSize);
131
132           setup(plugin.FIR0, u);
133
134           setup(plugin.FIR1, u); setup(plugin.FIR10, u); setup(plugin.FIR19, u); setup(plugin.FIR28, u);
135           setup(plugin.FIR2, u); setup(plugin.FIR11, u); setup(plugin.FIR20, u); setup(plugin.FIR29, u);
136           setup(plugin.FIR3, u); setup(plugin.FIR12, u); setup(plugin.FIR21, u); setup(plugin.FIR30, u);
137           setup(plugin.FIR4, u); setup(plugin.FIR13, u); setup(plugin.FIR22, u); setup(plugin.FIR31, u);
138           setup(plugin.FIR5, u); setup(plugin.FIR14, u); setup(plugin.FIR23, u); setup(plugin.FIR32, u);
139           setup(plugin.FIR6, u); setup(plugin.FIR15, u); setup(plugin.FIR24, u); setup(plugin.FIR33, u);
140           setup(plugin.FIR7, u); setup(plugin.FIR16, u); setup(plugin.FIR25, u); setup(plugin.FIR34, u);
```

```
141             setup(plugin.FIR8, u); setup(plugin.FIR17, u); setup(plugin.FIR26, u); setup(plugin.FIR35, u);
142             setup(plugin.FIR9, u); setup(plugin.FIR18, u); setup(plugin.FIR27, u); setup(plugin.FIR36, u);
143
144             plugin.currentFIR = plugin.FIR1;
145         end
146         function resetImpl(plugin) % funzione di reset
147             reset(plugin.FIR0);
148
149             reset(plugin.FIR1); reset(plugin.FIR10); reset(plugin.FIR19); reset(plugin.FIR28);
150             reset(plugin.FIR2); reset(plugin.FIR11); reset(plugin.FIR20); reset(plugin.FIR29);
151             reset(plugin.FIR3); reset(plugin.FIR12); reset(plugin.FIR21); reset(plugin.FIR30);
152             reset(plugin.FIR4); reset(plugin.FIR13); reset(plugin.FIR22); reset(plugin.FIR31);
153             reset(plugin.FIR5); reset(plugin.FIR14); reset(plugin.FIR23); reset(plugin.FIR32);
154             reset(plugin.FIR6); reset(plugin.FIR15); reset(plugin.FIR24); reset(plugin.FIR33);
155             reset(plugin.FIR7); reset(plugin.FIR16); reset(plugin.FIR25); reset(plugin.FIR34);
156             reset(plugin.FIR8); reset(plugin.FIR17); reset(plugin.FIR26); reset(plugin.FIR35);
157             reset(plugin.FIR9); reset(plugin.FIR18); reset(plugin.FIR27); reset(plugin.FIR36);
158
159             setLatencyInSamples(plugin, plugin.frameSize);
160         end
161         function flag = isInputSizeMutableImpl(~,~)
162             flag = true;
163         end
164         function s = saveObjectImpl(obj)
165             s = saveObjectImpl@matlab.System(obj);
166             s = savePluginProps(obj,s);
167             if isLocked(obj)
168
169                 s.FIRA = matlab.System.saveObject(obj.FIR0);
170
171                 s.FIR1 = matlab.System.saveObject(obj.FIR1);
172                 s.FIR2 = matlab.System.saveObject(obj.FIR2);
173                 s.FIR3 = matlab.System.saveObject(obj.FIR3);
174                 s.FIR4 = matlab.System.saveObject(obj.FIR4);
175                 s.FIR5 = matlab.System.saveObject(obj.FIR5);
176                 s.FIR6 = matlab.System.saveObject(obj.FIR6);
177                 s.FIR7 = matlab.System.saveObject(obj.FIR7);
178                 s.FIR8 = matlab.System.saveObject(obj.FIR8);
179                 s.FIR9 = matlab.System.saveObject(obj.FIR9);
180
181                 s.FIR10 = matlab.System.saveObject(obj.FIR10);
182                 s.FIR11 = matlab.System.saveObject(obj.FIR11);
183                 s.FIR12 = matlab.System.saveObject(obj.FIR12);
184                 s.FIR13 = matlab.System.saveObject(obj.FIR13);
185                 s.FIR14 = matlab.System.saveObject(obj.FIR14);
186                 s.FIR15 = matlab.System.saveObject(obj.FIR15);
187                 s.FIR16 = matlab.System.saveObject(obj.FIR16);
188                 s.FIR17 = matlab.System.saveObject(obj.FIR17);
189                 s.FIR18 = matlab.System.saveObject(obj.FIR18);
190
191                 s.FIR19 = matlab.System.saveObject(obj.FIR19);
192                 s.FIR20 = matlab.System.saveObject(obj.FIR20);
193                 s.FIR21 = matlab.System.saveObject(obj.FIR21);
194                 s.FIR22 = matlab.System.saveObject(obj.FIR22);
195                 s.FIR23 = matlab.System.saveObject(obj.FIR23);
196                 s.FIR24 = matlab.System.saveObject(obj.FIR24);
197                 s.FIR25 = matlab.System.saveObject(obj.FIR25);
198                 s.FIR26 = matlab.System.saveObject(obj.FIR26);
199                 s.FIR27 = matlab.System.saveObject(obj.FIR27);
200
201                 s.FIR28 = matlab.System.saveObject(obj.FIR28);
202                 s.FIR29 = matlab.System.saveObject(obj.FIR29);
203                 s.FIR30 = matlab.System.saveObject(obj.FIR30);
204                 s.FIR31 = matlab.System.saveObject(obj.FIR31);
205                 s.FIR32 = matlab.System.saveObject(obj.FIR32);
206                 s.FIR33 = matlab.System.saveObject(obj.FIR33);
207                 s.FIR34 = matlab.System.saveObject(obj.FIR34);
208                 s.FIR35 = matlab.System.saveObject(obj.FIR35);
209                 s.FIR36 = matlab.System.saveObject(obj.FIR36);
210             end
211         end
212         function loadObjectImpl(obj, s, wasLocked)
213             if wasLocked
214
215                 obj.FIR0 = matlab.System.loadObject(s.FIRA);
216
217                 obj.FIR1 = matlab.System.loadObject(s.FIR1);
218                 obj.FIR2 = matlab.System.loadObject(s.FIR2);
219                 obj.FIR3 = matlab.System.loadObject(s.FIR3);
220                 obj.FIR4 = matlab.System.loadObject(s.FIR4);
221                 obj.FIR5 = matlab.System.loadObject(s.FIR5);
222                 obj.FIR6 = matlab.System.loadObject(s.FIR6);
223                 obj.FIR7 = matlab.System.loadObject(s.FIR7);
224                 obj.FIR8 = matlab.System.loadObject(s.FIR8);
225                 obj.FIR9 = matlab.System.loadObject(s.FIR9);
226                 obj.FIR10 = matlab.System.loadObject(s.FIR10);
227                 obj.FIR11 = matlab.System.loadObject(s.FIR11);
228                 obj.FIR12 = matlab.System.loadObject(s.FIR12);
229                 obj.FIR13 = matlab.System.loadObject(s.FIR13);
230                 obj.FIR14 = matlab.System.loadObject(s.FIR14);
231                 obj.FIR15 = matlab.System.loadObject(s.FIR15);
```

```
            obj.FIR16 = matlab.System.loadObject(s.FIR16);
            obj.FIR17 = matlab.System.loadObject(s.FIR17);
            obj.FIR18 = matlab.System.loadObject(s.FIR18);
            obj.FIR19 = matlab.System.loadObject(s.FIR19);
            obj.FIR20 = matlab.System.loadObject(s.FIR20);
            obj.FIR21 = matlab.System.loadObject(s.FIR21);
            obj.FIR22 = matlab.System.loadObject(s.FIR22);
            obj.FIR23 = matlab.System.loadObject(s.FIR23);
            obj.FIR24 = matlab.System.loadObject(s.FIR24);
            obj.FIR25 = matlab.System.loadObject(s.FIR25);
            obj.FIR26 = matlab.System.loadObject(s.FIR26);
            obj.FIR27 = matlab.System.loadObject(s.FIR27);
            obj.FIR28 = matlab.System.loadObject(s.FIR28);
            obj.FIR29 = matlab.System.loadObject(s.FIR29);
            obj.FIR30 = matlab.System.loadObject(s.FIR30);
            obj.FIR31 = matlab.System.loadObject(s.FIR31);
            obj.FIR32 = matlab.System.loadObject(s.FIR32);
            obj.FIR33 = matlab.System.loadObject(s.FIR33);
            obj.FIR34 = matlab.System.loadObject(s.FIR34);
            obj.FIR35 = matlab.System.loadObject(s.FIR35);
            obj.FIR36 = matlab.System.loadObject(s.FIR36);
        end
        loadObjectImpl@matlab.System(obj,s,wasLocked);
        reload(obj,s);
    end
    %--------------------------------------------------------------------
    % Propagators
    function varargout = isOutputComplexImpl(~)
        varargout{1} = false;
    end
    function varargout = getOutputSizeImpl(obj)
        varargout{1} = propagatedInputSize(obj, 1);
    end
    function varargout = getOutputDataTypeImpl(obj)
        varargout{1} = propagatedInputDataType(obj, 1);
    end

    function varargout = isOutputFixedSizeImpl(obj)
        varargout{1} = propagatedInputFixedSize(obj,1);
    end
end
methods
    function plugin = HeadPhoneSimulator_IR() % costruttore
        plugin.FIR0 = dsp.FrequencyDomainFIRFilter;

        plugin.FIR1 = dsp.FrequencyDomainFIRFilter;
        plugin.FIR2 = dsp.FrequencyDomainFIRFilter;
        plugin.FIR3 = dsp.FrequencyDomainFIRFilter;
        plugin.FIR4 = dsp.FrequencyDomainFIRFilter;
        plugin.FIR5 = dsp.FrequencyDomainFIRFilter;
        plugin.FIR6 = dsp.FrequencyDomainFIRFilter;
        plugin.FIR7 = dsp.FrequencyDomainFIRFilter;
        plugin.FIR8 = dsp.FrequencyDomainFIRFilter;
        plugin.FIR9 = dsp.FrequencyDomainFIRFilter;

        plugin.FIR10 = dsp.FrequencyDomainFIRFilter;
        plugin.FIR11 = dsp.FrequencyDomainFIRFilter;
        plugin.FIR12 = dsp.FrequencyDomainFIRFilter;
        plugin.FIR13 = dsp.FrequencyDomainFIRFilter;
        plugin.FIR14 = dsp.FrequencyDomainFIRFilter;
        plugin.FIR15 = dsp.FrequencyDomainFIRFilter;
        plugin.FIR16 = dsp.FrequencyDomainFIRFilter;
        plugin.FIR17 = dsp.FrequencyDomainFIRFilter;
        plugin.FIR18 = dsp.FrequencyDomainFIRFilter;

        plugin.FIR19 = dsp.FrequencyDomainFIRFilter;
        plugin.FIR20 = dsp.FrequencyDomainFIRFilter;
        plugin.FIR21 = dsp.FrequencyDomainFIRFilter;
        plugin.FIR22 = dsp.FrequencyDomainFIRFilter;
        plugin.FIR23 = dsp.FrequencyDomainFIRFilter;
        plugin.FIR24 = dsp.FrequencyDomainFIRFilter;
        plugin.FIR25 = dsp.FrequencyDomainFIRFilter;
        plugin.FIR26 = dsp.FrequencyDomainFIRFilter;
        plugin.FIR27 = dsp.FrequencyDomainFIRFilter;

        plugin.FIR28 = dsp.FrequencyDomainFIRFilter;
        plugin.FIR29 = dsp.FrequencyDomainFIRFilter;
        plugin.FIR30 = dsp.FrequencyDomainFIRFilter;
        plugin.FIR31 = dsp.FrequencyDomainFIRFilter;
        plugin.FIR32 = dsp.FrequencyDomainFIRFilter;
        plugin.FIR33 = dsp.FrequencyDomainFIRFilter;
        plugin.FIR34 = dsp.FrequencyDomainFIRFilter;
        plugin.FIR35 = dsp.FrequencyDomainFIRFilter;
        plugin.FIR36 = dsp.FrequencyDomainFIRFilter;

        plugin.currentFIR = dsp.FrequencyDomainFIRFilter;
    end

    function set.Monitor(plugin,val)
        plugin.Monitor = val;
        updateIR(plugin);
```

```
        end
        function set.Target(plugin,val)
            plugin.Target = val;
            updateIR(plugin);
        end
        function updateIR(plugin) % funzione di aggiornamento delle risposte impulsive
            switch plugin.Monitor
                case MonitorEnum.AKGK240MKII
                    switch plugin.Target
                        case TargetEnum.AKGK240MKII, plugin.currentFIR = plugin.FIR0; % Target == Monitor
                        case TargetEnum.AKGK701, plugin.currentFIR = plugin.FIR1;
                        case TargetEnum.BeyerDynamicDT77080, plugin.currentFIR = plugin.FIR2;
                        case TargetEnum.BeyerDynamicDT990250, plugin.currentFIR = plugin.FIR3;
                        case TargetEnum.BeyerDynamicT1, plugin.currentFIR = plugin.FIR4;
                        case TargetEnum.FocalStellia, plugin.currentFIR = plugin.FIR5;
                        case TargetEnum.SennheiserHD600, plugin.currentFIR = plugin.FIR6;
                        case TargetEnum.SennheiserHD650, plugin.currentFIR = plugin.FIR7;
                        case TargetEnum.SennheiserHD800s, plugin.currentFIR = plugin.FIR8;
                        case TargetEnum.ShureSRH1540, plugin.currentFIR = plugin.FIR9;
                    end
                case MonitorEnum.BeyerDynamicDT990250
                    switch plugin.Target
                        case TargetEnum.AKGK240MKII, plugin.currentFIR = plugin.FIR10;
                        case TargetEnum.AKGK701, plugin.currentFIR = plugin.FIR11;
                        case TargetEnum.BeyerDynamicDT77080, plugin.currentFIR = plugin.FIR12;
                        case TargetEnum.BeyerDynamicDT990250, plugin.currentFIR = plugin.FIR0; % Target == Monitor
                        case TargetEnum.BeyerDynamicT1, plugin.currentFIR = plugin.FIR13;
                        case TargetEnum.FocalStellia, plugin.currentFIR = plugin.FIR14;
                        case TargetEnum.SennheiserHD600, plugin.currentFIR = plugin.FIR15;
                        case TargetEnum.SennheiserHD650, plugin.currentFIR = plugin.FIR16;
                        case TargetEnum.SennheiserHD800s, plugin.currentFIR = plugin.FIR17;
                        case TargetEnum.ShureSRH1540, plugin.currentFIR = plugin.FIR18;
                    end
                case MonitorEnum.BeyerDynamicT1
                    switch plugin.Target
                        case TargetEnum.AKGK240MKII, plugin.currentFIR = plugin.FIR19;
                        case TargetEnum.AKGK701, plugin.currentFIR = plugin.FIR20;
                        case TargetEnum.BeyerDynamicDT77080, plugin.currentFIR = plugin.FIR21;
                        case TargetEnum.BeyerDynamicDT990250, plugin.currentFIR = plugin.FIR22;
                        case TargetEnum.BeyerDynamicT1, plugin.currentFIR = plugin.FIR0; % Target == Monitor
                        case TargetEnum.FocalStellia, plugin.currentFIR = plugin.FIR23;
                        case TargetEnum.SennheiserHD600, plugin.currentFIR = plugin.FIR24;
                        case TargetEnum.SennheiserHD650, plugin.currentFIR = plugin.FIR25;
                        case TargetEnum.SennheiserHD800s, plugin.currentFIR = plugin.FIR26;
                        case TargetEnum.ShureSRH1540, plugin.currentFIR = plugin.FIR27;
                    end
                case MonitorEnum.SennheiserHD600
                    switch plugin.Target
                        case TargetEnum.AKGK240MKII, plugin.currentFIR = plugin.FIR28;
                        case TargetEnum.AKGK701, plugin.currentFIR = plugin.FIR29;
                        case TargetEnum.BeyerDynamicDT77080, plugin.currentFIR = plugin.FIR30;
                        case TargetEnum.BeyerDynamicDT990250, plugin.currentFIR = plugin.FIR31;
                        case TargetEnum.BeyerDynamicT1, plugin.currentFIR = plugin.FIR32;
                        case TargetEnum.FocalStellia, plugin.currentFIR = plugin.FIR33;
                        case TargetEnum.SennheiserHD600, plugin.currentFIR = plugin.FIR0; % Target == Monitor
                        case TargetEnum.SennheiserHD650, plugin.currentFIR = plugin.FIR34;
                        case TargetEnum.SennheiserHD800s, plugin.currentFIR = plugin.FIR35;
                        case TargetEnum.ShureSRH1540, plugin.currentFIR = plugin.FIR36;
                    end
            end %switch
        end % updateIR
        function set.bypass(plugin,val)
            plugin.bypass = val;
        end
    end
end
```

Ringraziamenti

Uno speciale ringraziamento va a tutte le persone che hanno sostenuto e reso possibile questo studio, con il loro aiuto, con la loro presenza o solamente con un pensiero.

Bibliografia

[1] *Virtual Studio Technology Wiki*. URL: https://en.wikipedia.org/wiki/Virtual_Studio_Technology.

[2] A.D. Baddeley et al. *Memory*. Cognitive Psychologie. Psychology Press, 2009. ISBN: 9781848720008. URL: https://books.google.it/books?id=3h-BPQAACAAJ.

[3] Charlie DeVane and Gabriele Bunkheila. "Automatically Generating VST Plugins from Matlab Code". In: *Audio Engineering Society Convention 140*. Audio Engineering Society. 2016.

[4] The MathWorks Inc. *audioPlugin class online documentation*. (accessed in 2024). 2022. URL: https://it.mathworks.com/help/audio/ref/audioplugin-class.html?searchHighlight=audio%20plugin&s_tid=srchtitle_support_results_1_audio%20plugin.

[5] The MathWorks Inc. *Audio Toolbox (R2024a)*. (accessed in 2024). 2024. URL: https://it.mathworks.com/help/audio/index.html?searchHighlight=audio%20toolbox&s_tid=srchtitle_support_results_1_audio%20toolbox.

[6] The MathWorks Inc. *DSP System Toolbox (R2024a)*. (accessed in 2024). 2024. URL: https://it.mathworks.com/help/dsp/index.html?searchHighlight=DSP%20system%20toolbox&s_tid=srchtitle_support_results_1_DSP%20system%20toolbox.

[7] The MathWorks Inc. *Signal Processing Toolbox (R2024a)*. (accessed in 2024). 2024. URL: https://it.mathworks.com/help/signal/index.html?searchHighlight=Signal%20processing%20toolbox&s_tid=srchtitle_support_results_1_Signal%20processing%20toolbox.

[8] The MathWorks Inc. *dsp.AudioFileReader online documentation*. (accessed in 2024). 2023. URL: https://it.mathworks.com/help/dsp/ref/dsp.audiofilereader-system-object.html?searchHighlight=dsp.AudioFileReader&s_tid=srchtitle_support_results_1_dsp.AudioFileReader.

[9] The MathWorks Inc. *AudioFileWriter online documentation*. (accessed in 2024). 2023. URL: https://it.mathworks.com/help/audio/ref/audiodevicewriter-system-object.html?searchHighlight=audioDeviceWriter&s_tid=srchtitle_support_results_1_audioDeviceWriter.

[10] The MathWorks Inc. *step function and Matlab System Objects class online documentation*. 2024. URL: https://it.mathworks.com/help/matlab/ref/step.html.

[11] The MathWorks Inc. *Audio Test Bench (R2024a)*. (accessed in 2024). 2024. URL: https://it.mathworks.com/help/audio/ref/audiotestbench-app.html.

[12] The MathWorks Inc. *validateAudioPlugin class online documentation*. (accessed in 2024). 2023. URL: https://it.mathworks.com/help/audio/ref/validateaudioplugin.html.

[13] The MathWorks Inc. *generateAudioPlugin class online documentation*. (accessed in 2024). 2023. URL: https://it.mathworks.com/help/audio/ref/generateaudioplugin.html.

[14] Anna Zuccante. "A Signal-Processing Based simulation system for stereo high-end headsets: optimising and testing". PhD thesis. Università di Padova, 2023.

[15] Bernard Mulgrew, Peter Grant, and John Thompson. "Digital signal processing: concepts and applications". In: (2002).

[16] Angelo Farina. "Simultaneous Measurement of Impulse Response and Distortion With a Swept-Sine Technique". In: (Nov. 2000).

[17] Guy-Bart Stan, Jean-Jacques Embrechts, and Dominique Archambeau. "Comparison of different impulse response measurement techniques". In: *Journal of the Audio Engineering Society* 50.4 (Apr. 2002), pp. 249–262.

[18] John Borwick. *Loudspeaker and headphone handbook*. CRC Press, 2012.

[19] Jiazeng Wang et al. "Research on Potential Market Trend of High-Fidelity Headphones". In: *The Frontiers of Society, Science and Technology* 2 (12 2020), pp. 127–132. DOI: https://doi.org/10.25236/FSST.2020.021219. URL: https://francis-press.com/papers/2564.

[20] *List of Oratory1990 Presets*. 2019. URL: https://www.reddit.com/r/oratory1990/wiki/index/list_of_presets/.

[21] Robert Bristow-Johnson. "Cookbook formulae for audio EQ biquad filter coefficients". In: *http://www.musicdsp. org/files/Audio-EQ-Cookbook. txt* (2016).

[22] *Audio-EQ-Cookbook*. 2019. URL: https://webaudio.github.io/Audio-EQ-Cookbook/audio-eq-cookbook.html.