**UNIVERSITÀ DI PADOVA** **FACOLTÀ DI INGEGNERIA**

TESI DI LAUREA

# INSEGUIMENTO DEL TEMPO MUSICALE MEDIANTE ALGORITMI REAL-TIME DI TIME STRETCHING SU FLUSSI AUDIO MULTIPLI

**Laureando**: Stefano Merlin

**Relatore**: Federico Avanzini
**Correlatore**: Matteo Romanin

## Corso di Laurea Magistrale in Ingegneria Informatica

18 aprile 2011

Anno Accademico 2009 - 2010

# Prefazione

In questi anni stiamo assistendo ad un ulteriore passo nell'evoluzione che interessa il mercato *consumer* per la riproduzione di contenuti musicali. Fino a qualche decennio fa erano gli strumenti analogici, come vinili o cassette, a dominare il mercato; oggi oggi invece la situazione è cambiata a tal punto che Sony ha deciso di dismettere la catena di produzione dei suoi storici walkman. La rivoluzione sta colpendo anche il mercato dei contenuti in cui si assiste alla nascita di portali e-commerce che trattano direttamente prodotti digitali privi di supporto fisico o alla creazione di portali a tariffa fissa che stanno proponendo le prime versioni di *musica liquida* in una rete globale sempre più votata al clouding. Se da una parte i consumatori cominciano soltanto ora ad utilizzare al meglio la tecnologia digitale (l'introduzione del lettore mp3 sta superando perfino il lettore CD per facilità ed esperienza di impiego), dall'altra, già da tempo, gli artisti e le case di produzione avevano iniziato a saggiarne le potenzialità. I primi esperimenti di composizione musicale attraverso monitor e tastiera sono stati compiuti verso la fine degli anni '70, ma allora si trattava di tentativi che erano noti ad ambienti ristretti e richiedevano risorse e conoscenze non alla portata di tutti. È stato l'aumento di potenza di calcolo dei *personal computer* che ha esteso l'uso di questi nuovi strumenti ad un maggior numero di utenti. Ai giorni nostri esistono un numero non trascurabile di applicazioni software che permettono di produrre musica senza l'intervento di strumenti fisici anzi, con l'avvento dei *Virtual inSTrument*, si può disporre di una orchestra virtuale. Non solo, una delle più grandi risorse dei software musicali è quella di rendere fruibili all'artista librerie molto ampie di registrazioni digitali che possono essere usate in ogni momento durante la composizione di nuovi brani. Il mercato propone raccolte di suoni digitalizzati che riproducono sia singole note di vari strumenti che interi fraseggi musicali. Non di rado queste ritmiche, o quelle che comunemente vengono indicate coma *basi musicali*, sono presenti nelle librerie campionate a diverse velocità di esecuzione per cercare di coprire una casistica di utilizzo il più estesa possibile. Questo scenario è estremamente positivo per lo sviluppo della creatività, tuttavia esistono dei limiti nell'utilizzo di questo tipo di teconologie: spesso le basi fornite presentano problemi di sincronicità così marcati che il loro utilizzo non risulta soddisfacente. Il compositore si ritrova così a non poter utilizzare direttamente il materiale offerto anche se lo ritiene valido. Il presente lavoro nasce proprio dall'esigenza di risolvere questo problema. In altre parole si è cercato di definire uno strumento che renda fruibile il materiale musicale a disposizione a prescindere dalla velocità di esecuzione con cui è stato acquisito. Il software realizzato in questa sede permette la sincronia automatica delle registrazioni che si hanno in libreria consentendo anche la variazione della velocità del brano durante la riproduzione stessa senza perdere la sincronia tra le diverse parti. In questo modo non si è costretti, come

spesso accade, ad avere più copie dello stesso campione musicale suonato con tempi musicali diversi e allo stesso tempo non si è limitati dalle velocità di esecuzione predefinite nelle librerie dei software musicali. Tutto questo ha il vantaggio di essere trasparente all'utente che si deve solamente preoccupare di definire le variazioni di velocità del brano prodotto.

# Sommario

I software musicali moderni utilizzano il formato Wave PCM come rappresentazione di segnali audio non compressi. Questo formato risulta molto elastico nella definizione della qualità rappresentativa della forma d'onda che descrive, ma non contiene alcuna informazione riguardo elementi basilari della teoria musicale come *tempo musicale* e *battuta*. Scopo del presente lavoro è quello di estendere il formato Wave PCM per dotarlo della possibilità di contenere informazioni aggiuntive di natura temporale, cercando in questo modo di ottenere un processo di sincronia tra rappresentazioni informatiche di brani musicali anche molto eterogenei tra loro.

Il lavoro parte dalla definizione di un metadato aggiuntivo, che viene chiamato *Tag di battuta*, utile per la sincronia in diversi scenari ipotetici. Il documento prosegue analizzando una casistica semplice, come quella di due brani la cui riproduzione non è stata avviata contemporaneamente, fino ad arrivare alla risoluzione completa: avere un *flusso target* dotato di un valore di tempo musicale variabile nel tempo che permetta la sincronia di qualsiasi *flusso secondario* caricato in memoria. Per flusso target si intende proprio l'estensione del Wave PCM che può essere creato in modalità real-time dall'utente o può essere letto da disco: questo nuovo formato contiene il riferimento temporale che viene impostato dall'utente. I flussi secondari sono invece semplici file Wave PCM, di cui si conosce il valore di *battiti per minuto*, che vengono trattati dall'applicazione in modo da seguire le variazioni descritte dal flusso target. Si desume quindi che i *Tag temporali* debbano essere un'indicazione temporale assoluta che non dipenda né dal tempo con cui il brano è stato eseguito e registrato, né dalle proprietà intrinseche del file Wave PCM come *sample rate*, *numero di canali* o valore di *bits per sample*. Seguendo questo approccio si definisce quindi una *sincronia musicale* che viene ottenuta confrontando non più la posizione temporale dei segnali audio, ma comprendendo a quale battuta di spartito appartengono i campioni audio correnti. Tutto il procedimento viene reso trasparente all'utente attraverso un sistema di controllo che, utilizzando il nuovo formato descritto nel documento, permette di modificare, in maniera interattiva o programmata, la velocità di esecuzione del flusso target senza perdere la sincronia dei flussi secondari. Una volta ottenuta la sincronia, ci si accorge di come il modello possa essere utilizzato anche per una ragionata "compressione" dei dati passando dalla logica di segnale alla logica di spartito: vengono introdotte funzionalità di *pausa* e di *ripetizione* che attestano la versatilià del nuovo formato.

Il punto fondamentale di questo lavoro è la scelta di uno strumento che permetta di modificare il tempo musicale dei brani coinvolti senza alterarne il contenuto. Variare semplicemente la velocità di riproduzione porta ad una inevitabile modificazione del tono del suono e questo è chiaramente un effetto altamente indesiderato nel nostro ambito. Vista la natura real-time dell'applicazione si è cercato di applicare funzioni di *time stretching* in maniera molto performante dal punto di vista del tempo di esecuzione, la scelta è quindi ricaduta su uno strumento che lavora nello spazio delle frequenze: il *Phase Vocoder*. Nella seconda parte di questo lavoro si cerca di definire in maniera dettagliata il funzionamento del Phase Vocoder e se ne analizzano i punti deboli: infatti, pur essendo uno strumento velocissimo dato l'uso di algoritmi come *Fast Fourier Transform*, esso introduce degli artefatti nel suono di uscita che possono essere descritti come riverberi o, nei casi peggiori, ci si può trovare in presenza di una vera e propria "destrutturazione". Durante l'analisi si definiscono due paramentri fondamentali per comprendere i possibili affinamenti da applicare agli algoritmi: la *coerenza di fase orizzontale* e la *coerenza di fase verticale*. Queste osservazioni portano ad identificare versioni migliorate del Phase Vocoder che propongono il *Lock di fase*. Sono presenti anche altre propooste che cercano di interpretare in maniera più corretta la *Short-time Fourier Transform* del segnale di ingresso oppure propongono soluzioni per specifiche condizioni come il *reset di fase* quando il segnale risulta essere particolarmente "silenzioso", o il tracciamento di traiettorie euristiche per i picchi dei moduli dell'analisi in frequenza.

Infine vengono illustrate le scelte tecniche e di sviluppo che caratterizzano la fase di implementazione dell'applicazione. Prima di tutto si modifica il Phase Vocoder base al fine in modo che possa creare e utilizzare i *Tag di battuta* visti nella in precedenza; in seguito si cerca di implementare una sezione di controllo che renda possibile la sincronia ricercata. Particolare attenzione è stata posta su alcuni aspetti chiave: velocizzazione della fase di sintesi, configurazione e gestione della memoria e suddivisione degli *eventi* che possono interessare il funzionamento dell'applicazione in thread specifici. In questo modo è possibile creare uno strumento che risponda effettivamente in maniera interattiva e nei tempi previsti. Al termine si presentano risultati che in merito all'effettivo funzionamento dell'applicazione multithread stand alone implementata.

# Indice

# Elenco delle figure

# Elenco delle tabelle

# Capitolo 1

# Sincronizzazione

*"Il più necessario e il più difficile ed essenziale nella musica è il tempo."*
(Wolfgang Amadeus Mozart)

Definire in modo univoco e preciso il tempo è un elemento di primaria importanza nell'ambito musicale. Il dibattito sul tema ha attraversato tutta la storia della musica, basti pensare che sino al XV secolo il sistema per scandire il tempo di un brano musicale da parte del compositore era il *tactus*. Gli autori si affidavano al proprio battito cardiaco per informare gli interpreti del ritmo specifico di un'opera. Ben presto ci si accorse di quanto questo metodo fosse impreciso e variasse da artista ad artista andando ad inficiare notevolmente la precisione dell'informazione temporale trasmessa. La soluzione arrivò nei primi anni del 1800 quando, partendo da un lavoro del 1696 di Étienne Lulié, Dietrich Nikolaus Winkel costruì un nuovo strumento musicale che contenesse in esso la legge (*nomos*) di misura (*metron*) del tempo musicale. Il metronomo è [1] uno strumento racchiuso in una scatola piramidale e comprende un meccanismo ad orologeria che mette in movimento un pendolo con un peso fisso nella parte bassa e un peso mobile nella parte alta, il quale è in grado di variare largamente il periodo di oscillazione. Questa variazione permette allo strumento di eseguire dai 40 ai 208 periodi in un minuto primo. Da allora la tecnologia è molto cambiata e la misurazione del tempo si è notevolmente affinata, ma l'idea del metronomo è ancora insita nel valore di *battiti per minuto*[1] (BPM) che tutti i software musicali adoperano per misurare la velocità a cui un brano musicale deve essere eseguito.

La versatilità degli elaboratori ben presto è stata vista come una nuova frontiera per la produzione artistica e in questo la musica non ha fatto alcuna eccezione: la potenza di calcolo si è riversata nell'enorme possibilità di creare, modificare e far interagire tra loro diversi segnali musicali. Inoltre la progressiva espansione delle memorie digitali ha offerto l'opportunità di aumentare sia la quantità che la qualità delle registrazioni del mondo reale in digitale.

Un qualsiasi suono è in realtà una variazione di pressione, digitalizzarlo è ormai diventata un'azione alla portata di tutti e consiste nel trasformarlo in un segnale elettrico che verrà poi trattato dall'elaboratore. Il formato più utilizzato per il salvataggio ad alta ferdeltà delle infor-

[1] A volte viene erroneamente utilizzata l'espressione *battute per minuto* che si rivela in realtà una definizione ambigua dato che il termine battuta indica la serie metrica compresa tra le stanghette del rigo musicale e non un singolo battito.

mazioni ricavate durante la digitalizzazioni è il Wave (A.1) PCM[2] che quantizza e campiona il segnale elettrico con definizione scelta dall'utente. In questo modo abbiamo informazioni molto precise sulla forma d'onda del segnale e possiamo riprodurlo fedelmente o in alternativa utilizzarlo come input per successivi processi numerici. Tuttavia non siamo in grado di rispondere a due domande fondamentali: a quale velocità è stato eseguito il brano? L'interprete ha modificato in corso d'opera la velocità di esecuzione o l'ha mantenuta sempre costante?

In questo capitolo si propone di dotare il file PCM di metadati aggiuntivi che possano descrivere le informazioni mancanti sulla velocità permettendo in questo modo la sincronizzazione musicale tra file. A questo scopo è necessario per prima cosa definire tre concetti fondamentali: il concetto di flusso target, quello di flussi secondari e il significato di $\alpha$. Il **flusso target** è il flusso audio con valore di battiti per minuto variabile a cui i flussi secondari devono essere sincronizzati mentre i **flussi secondari** sono normali brani musicali salvati in formato Wave PCM di cui si conosce a priori il valore di battiti per minuto. Con $\alpha$ si indica invece il rapporto di compressione ($\alpha < 1$) o di dilatazione ($\alpha > 1$) che subisce la scala dei tempi del flusso secondario durante la sincronizzazione

Successivamente il lavoro approfondirà i metodi per creare flussi audio che, pur vedendo variata la propria velocità dall'intervento dell'utente, sappiano contenere le informazioni sufficienti per permettere comunque la sincronizzazione di altri flussi che hanno velocità costante e spesso diversa da quella che indicata dall'utente.

## 1.1 Un primo approccio

In questo contesto verranno proposte soluzioni di sincronia musicale in vari scenari ordinati per difficoltà crescenti fino alla descrizione del modello completo.

Per *sincronia musicale*, si intende qualcosa di totalmente diverso dalla semplice sincronia che può essere attuata tra file PCM: sincronizzare due brani uguagliandone il tempo trascorso dall'inizio del flusso può avere senso dal punto di vista musicale se e solo se i due segnali rappresentati sono descritti dallo stesso valore di BPM. In altri casi la sincronia fallirebbe. Per meglio comprendere la questione pensiamo a due maestri che si ritrovano per provare l'interpretazione di un brano musicale a due voci e supponiamo che esista un passaggio che ha bisogno di particolari attenzioni rispetto ad altri. Hanno quindi bisogno di attuare sin da subito una sincronizzazione: ciò non avviene attraverso l'indicazione del numero di pagina dello spartito dalla quale intendono iniziare perché quel valore è collegato al tipo di carattere utilizzato, né con il conteggio delle note perché le due parti possono essere anche notevolmente diverse, bensì con l'utilizzo di una misura fondamentale della teoria musicale: la battuta. Per sincronia musicale si intende quindi la possibilità di attuare un allineamento tra i due brani distinti che tenga conto della teoria musicale.

[2] *Pulse Code Modulation* è un metodo di rappresentazione digitale di un segnale analogico. Il metodo utilizza un campionamento nel tempo e una quantizzazione del segnale. La PCM è ampiamente utilizzata non solo nell'ambito audio, ma anche in quello video.

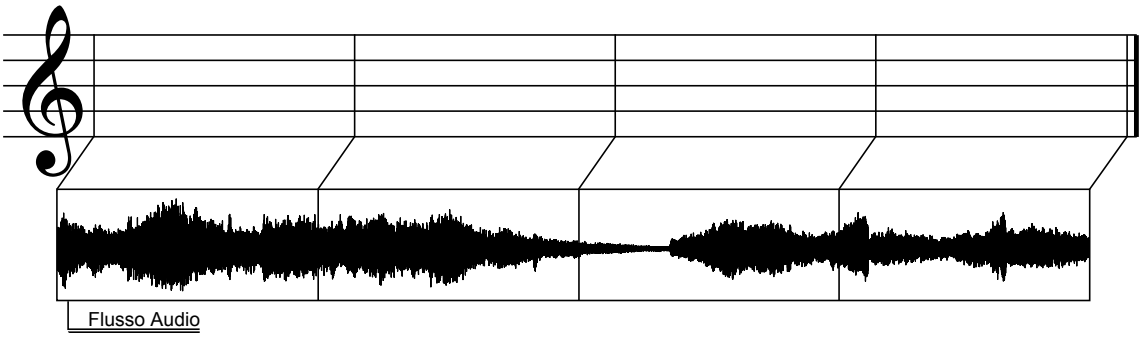


**Figura 1.1:** *Tag di battuta*

Per ottenere la sincronia voluta dobbiamo riuscire ad inserire all'interno di un flusso audio PCM delle informazioni che sappiano comunicare le posizioni delle battute riuscendo così ad avere riferimenti temporali validi.

### 1.1.1 Ottenere i Tag temporali

Al fine di illustrare le metodologie di inserimento di metadati aggiuntivi nel flusso audio target cerchiamo di procedere per esempi di complessità crescente.

#### Partenza non sincrona

Iniziamo quindi dal caso più semplice: l'esecuzione del flusso target è già in corso e abbiamo l'intenzione di sincronizzare il flusso secondario che risiede già in memoria, ma che è ancora alla battuta zero. I due flussi sono caratterizzati dallo stesso valore di BPM che inoltre si mantiene costante durante tutto il brano.

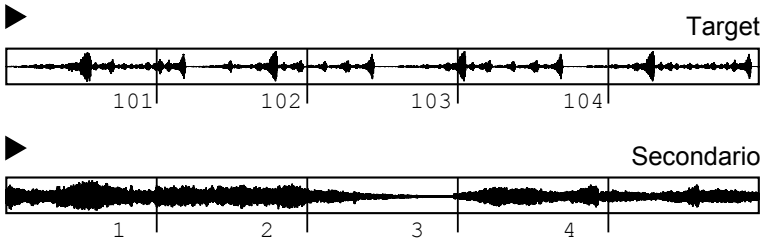


**Figura 1.2:** *Differenza nei tempi di partenza*

Il risultato che vogliamo ottenere è quello di identificare a quale battuta si trova il primo brano e decidere di conseguenza come intervenire sul secondo.

Per fare questo dobbiamo riuscire a ricavare la valutazione della battuta in cui ci si trova nel tempo $t$ e questo valore deve ovviamente valere per qualsiasi tipo di flusso indipendentemente dalla caratterizzazione dello stesso. Tale calcolo va quindi slegato dal *sample rate* e dal numero dei *canali* (mono, stereo ...) andando ad utilizzare l'informazione aggiuntiva della velocità che avevamo ipotizzato essere anche costante.

$$\frac{X_{s_{tg}}(t)}{N_{ch} \times SR} \times \frac{BPM}{60} \tag{1.1}$$

dove $X_{s_{tg}}(t)$ é la posizione misurata in sample del flusso audio target nell'istante $t$, $N_{ch}$ é il numero di canali, $SR$ é il sample rate, $BPM$ é la velocità del brano che in questo caso viene dichiarata fissa. Dobbiamo inoltre porre attenzione al fatto che per renderizzare un flusso audio abbiamo bisogno di caricarne un frammento (che verrà chiamato *frame di rendering*) nella memoria dell'elaboratore. Se ad ogni frame di rendering noi riuscissimo a conoscere il numero di battuta su cui stiamo agendo, potremmo decidere come intervenire per la sincronia musicale.

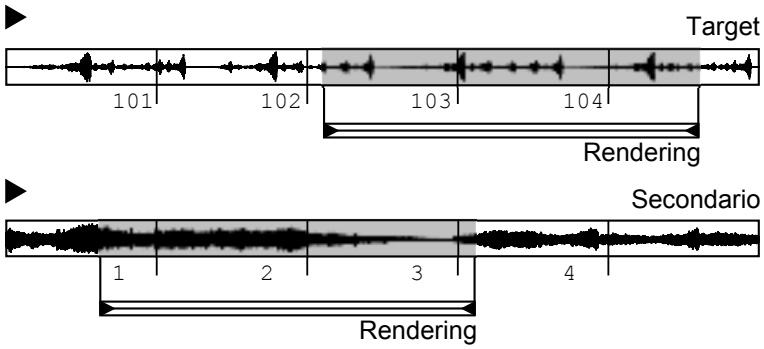


**Figura 1.3:** *Partenze sfalsate con buffer di rendering*

Supponiamo ora di utilizzare un qualsiasi algoritmo di time stretching per poter variare la velocità del brano secondario in modo da poter raggiungere il flusso target: caricheremo in memoria un frame dal file che vogliamo variare e lo modificheremo in modo da poter raggiungere la sincronia.

$$X_{in}(t, \Delta t) \longrightarrow Y_{in}(t', \Delta t') \tag{1.2}$$

dove $X_{in}(t, \Delta t)$ è il frame del file secondario in ingresso, $Y_{in}(t', \Delta t')$ è il frame modificato in velocità e $\Delta t$ è la lunghezza temporale dei frame presi in considerazione. Dalla scrittura si evince che la misura corretta della battuta è data dal valore della funzione dipendente da $t$ e non da $t'$, deduciamo quindi che è fondamentale annotare il tempo di analisi e non quello di sintesi per avere sincronia. Per ottenere la sincronia dobbiamo quindi azzerare la differenza tra i valori di battuta del flusso target (che è già in rendering) e del flusso secondario (che si trova a battuta *Zero*). Tale calcolo dovrebbe essere affrontato per ogni frame di rendering che si crea.

Sebbene lo scenario ipotizzato permetta anche altre soluzioni più semplici, questo caso risulta importante per illustrare la metodologia usata nella risoluzione di problemi più complessi come ad esempio la creazione di flussi audio la cui velocità varia nel tempo, ma che continuano a permettere la sincronizzazione musicale. Il nodo principale è infatti quello di creare un flusso audio ottimizzato che permetta la sincronia anche se il suo valore di BPM varia durante tutta la sua durata. L'equazione (1.1) viene riscritta come:

$$\frac{X_{s_{tg}}(t)}{N_{ch} \times SR} \times \frac{BPM(t)}{60} \tag{1.3}$$

si desume quindi che il valore di BPM vari al variare del tempo. Si potrebbe pensare di appuntare direttamente il valore di BPM per ogni frame successivo di rendering, ma ciò non è possibile perché in questo modo il sistema si rivelerebbe molto sensibile a possibili errori.

È infatti non raccomandabile salvare un valore di velocità per ottenere una sincronizzazione nel tempo: prima di tutto si perderebbe la generalizzazione che si può ottenere nel modello e, non meno importante, si potrebbe creare un modello che pur ottenendo l'uguaglianza dal punto di vista della velocità potrebbe non azzerare la distanza musicale tra i due brani.

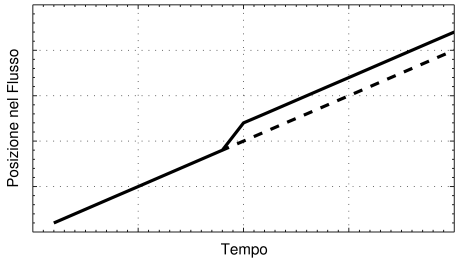


**Figura 1.4:** *Grafico velocità*

Come illustrato in figura sebbene i valori di velocità siano uguali (le due rette procedono parallelamente) i dati non permettono di evidenziare la differenza temporale costante e questo sarebbe inaccettabile.

### Posizionamento dei Tag

Nella musica le battute sono sempre indicate da numeri interi. Nella corrente descrizione questo passaggio è rimasto implicito, ma ha un forte impatto dal punto di vista implementativo. Ipotizziamo infatti che la velocità nel flusso target sia fatta variare:

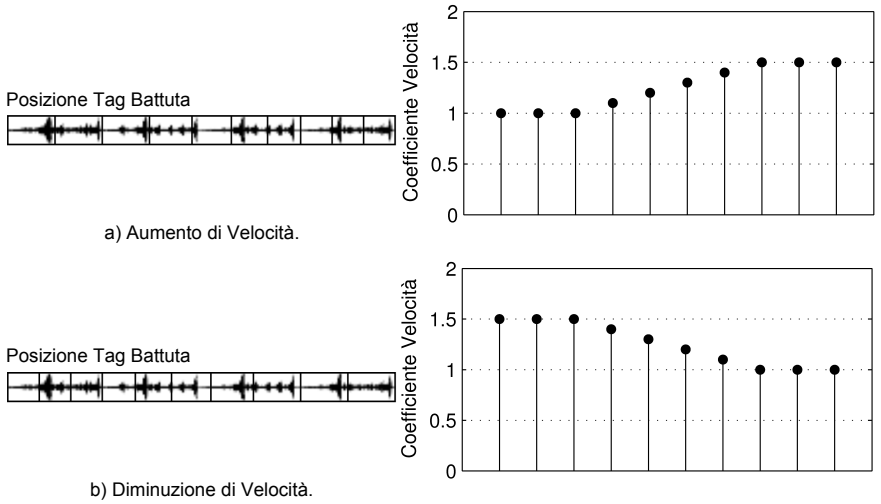


**Figura 1.5:** *Tag battuta interi a velocità variabile*

come si vede dalla figura, le posizioni di riferimento delle battute si presentano a diverse distanze che sono fortemente correlate con la modifica della velocità attuata dall'utente. Questo schema però è di implementazione impossibile dato che non saremmo più in grado di rilevare i Tag e distinguerli dai campioni audio dato che questi ultimi assumono quasiasi valore nel range esistente. Serve quindi definire una distanza standard a cui posizionare questi Tag utilizzando numeri reali per indicare frazioni di battuta del brano.

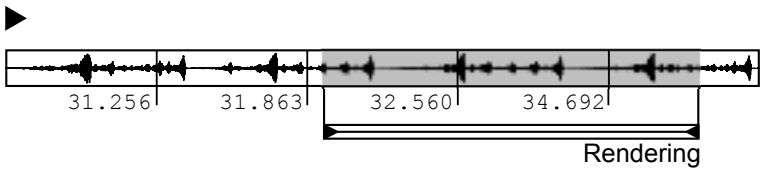


**Figura 1.6:** *Tag battuta reali a distanza fissa*

Si noti che il posizionamento dei Tag è fondamentale perché in questo modo, si decide la massima frequenza di aggiornamento dei valori di BPM e, di fatto, la latenza con cui le modifiche dell'utente vengono applicate al segnale stesso. Si è quindi deciso di far coincidere il valore di latenza con quello di lunghezza del frame di rendering in previsione dell'effettivo utilizzo di questo nuovo flusso.

### Flusso Audio Target

Supponiamo ora di costruire il flusso audio target partendo da un file audio a BPM costante ed utilizzando algoritmi di time stretching per variare la velocità del brano[3] nel tempo. Qualsiasi sia l'algoritmo scelto per la modifica, si dovrà caricare un frame di input e se ne otterrà uno nuovo con velocità modificata. Data la (1.2) e combinata alla (1.3) si vede che bisogna creare un valore di battuta derivato dalla variabile $t$ e che quest'ultimo venga inserito in un flusso audio dipendente dalla variabile $t'$. Si creano così due scale temporali: una di rendering che risulterà solidale a quella dell'ascoltatore e una di analisi che risulterà solidale con quella della registrazione di partenza.

Questa relazione permette di annotare i Tag di battuta effettivi del brano senza che essi vengano modificati dallo stretching del tempo e rende il valore calcolato valido per il confronto tra ogni flusso.

Come si vede dalla figura quando il tempo viene compresso, i dati di input scorrono molto più velocemente e quindi i Tag di battuta creati avranno distanza assoluta maggiore tra loro, d'altro canto se il tempo venisse dilatato il file di input scorrerebbe molto più lentamente e i valori sarebbero molto più "*vicini*" l'un l'altro. Abbiamo tecnicamente reso assoluto il tempo

[3]Non è una limitazione, se si vuole campionare dal vivo un segnale che varia nel tempo si potrà procedere in modo analogo avendo cura di appuntare i Tag di battuta correttamente attraverso, per esempio, l'utilizzo di un metronomo digitale.

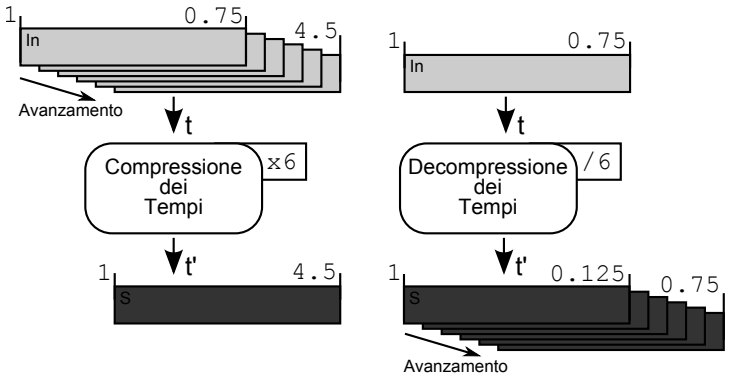


**Figura 1.7:** *Relazione tra frame di ingresso e di sintensi*

slegandoci da quello di rendering: è come se all'interno del flusso audio esistessero riferimenti alle posizioni delle battute originali servite per la creazione del frame di rendering corrente[4].

Quando trattiamo i flussi secondari (che per ora sono campionati allo stesso BPM base del file input per la creazione del flusso target) cerchiamo di adattare la velocità a quella del flusso tenendo nota dei Tag di battuta creati, specificatamente per loro, a livello di run time durante la fase di sincronia.

I due flussi quando hanno marcatori uguali non possono che essere sincronizzati[5]. Per rendere più chiara la procedura possiamo prendere l'esempio in cui il file di input per la creazione del flusso con velocità variabile nel tempo sia identico al flusso secondario che deve essere sincronizzato. Ogni qual volta che i Tag creati coincidono vuol dire che i due algoritmi di time stretching (quello usato per creare il flusso target e quello usato per sincronizzare il flusso secondario) hanno in input gli stessi dati e quindi hanno uscite identiche. Da questo si desume che, se sovrapposti, non darebbero origine a nessuna differenza di sincronia.

### Flussi secondari con diversi valori di BPM

Ora complichiamo ulteriormente il modello pensando di aggiungere flussi secondari dotati di BPM del tutto diversi tra di loro. Per l'algoritmo indicato in percedenza nulla varia e la sincronia viene mantenuta valida anche per questo scenario: supponiamo che un file audio rappresenti la campionatura di un brano audio $A$ suonato a 100 BPM e che un altro $B$ lo sia di uno suonato a 50 BPM. I Tag di battuta vengono creati tenendo conto di questa variazione e le distanze calcolate durante il runtime atte alla sincronia avranno un peso diverso a seconda di quale brano viene preso in considerazione. Una battuta del file $A$ sarà costituita dalla metà dei campioni rispetto a quella del file $B$[6] e quindi la progressione dei valori di battuta procederanno in maniera da

[4] In questo modo possiamo dire da quale battuta dello spartito è stato generato il frame N-esimo di rendering del flusso target a prescindere dal tempo di rendering che non ha alcun legame con lo spartito dato che i battiti per minuto variano nel tempo.

[5] Avere marcatori uguali vuol dire essere nella stessa battua di spartito.

[6] Si ipotizza che entrambi i file siano campionati con lo stesso valore di Sample Rate. Anche questa però è una semplificazione esclusivamente espositiva perché i Tag di battuta non variano al variare del Sample Rate.

ottenere valida l'uguaglianza $\alpha_B = 2\alpha_A$ a regime.

Si comprende quindi come il modello di controllo non debba tener conto dei valori di BPM di tutti i flussi secondari, di questo si preoccupa infatti la funzione che crea i Tag riassuntivi delle posizioni assolute per la sincronia. Quello che risulta sufficiente è che la distanza dal flusso target sia calcolata per ogni flusso secondario e che ognuno di questi sia dotato del proprio coefficiente $\alpha$. Ricordiamo che il flusso target viene creato da un file Wave PCM con valore di BPM costante quindi tutti i valori dei Tag temporali sono relativizzati ai BPM dei sorgenti, siano essi usati per la creazione di un flusso target o come input agli algoritmi di time stretching che devono modificare i flussi secondari per la sincronia.

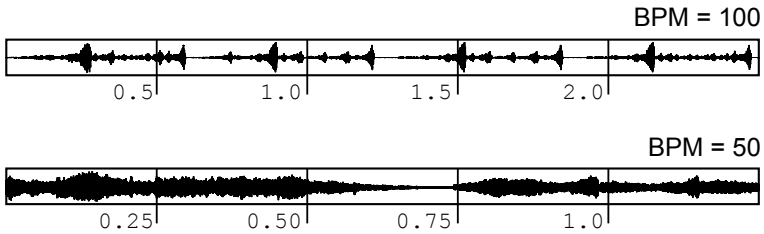


**Figura 1.8:** *Diversi Tag per diversi BPM*

### 1.1.2 Sistema di controllo

Una volta definiti i Tag temporali e sottolineate le loro proprietà, si procede utilizzandoli per creare un sistema di controllo che cerchi di minimizzare la distanza tra il flusso target e quelli secondari in maniera automatica. Sia $\Delta_b(n)$ la distanza in battute che sussiste tra il flusso target e un flusso secondario nell'istante $t$ e sia essa di valore negativo se il flusso secondario è in ritardo rispetto al target o di valore positivo se siamo in presenza di anticipo. Con il nostro controllo vogliamo aumentare il valore di $\alpha(n)$ se $\Delta_b(n)$ è maggiore di zero o diminuirlo in caso contrario. Visto che i valori di Tag di battuta sono stati costruiti in modo che essi possano essere comparati tra di loro per tutti i casi possibili, anche $\Delta_b(n)$ godrà della proprietà di descrivere correttamente lo stato di sincronia in qualsiasi scenario. Questo porta ad un vantaggio notevole perché, in questo modo, il valore di $\alpha$ si adatta a valori differenti in maniera automatica come abbiamo visto in precedenza parlando di flussi secondari dotati di diversi valori di BPM. Ricordiamo infatti che i valori di $\Delta_b(n)$ contengono al loro interno dati che tengono conto già della differenza relativa dei BPM dei vari flussi: i valori dei Tag temporali sono del tutto coerenti tra loro sollevando così il sistema di controllo da eventuali conversioni nel momento della sincronia.

Il controllo possiede in memoria il valore precedente di $\alpha$ e lo deve modificare in relazione al $\Delta_b(n)$ che viene rilevato ad ogni passo di rendering. Si comprende ora che la scelta di uguagliare la distanza dei Tag temporali con la dimensione del buffer di renderizzazione risulta la migliore da un punto di vista di controllo: in questo modo ogni volta che il programma prepara un frame di rendering, esso sa poi analizzare le conseguenze dell'$\alpha$ scelto per la sintesi e aggiustare di conseguenza la velocià per la sintesi del prossimo frame.

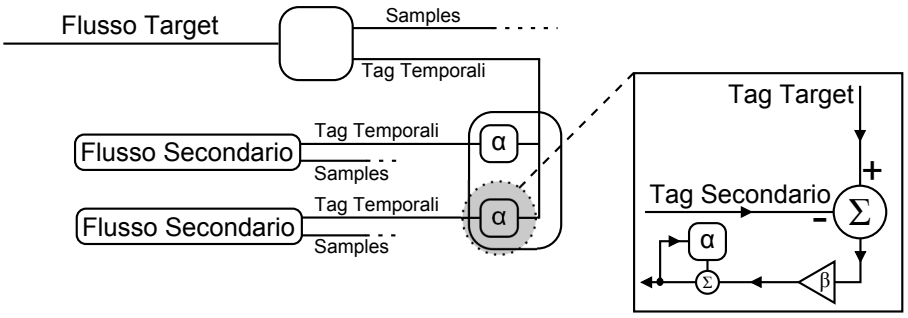


**Figura 1.9:** *Schema del controllo*

Un possibile calcolo del nuovo valore di $\alpha$ è:

$$\alpha(n) = \alpha(n-1) + \frac{\Delta_b(n-1)}{\beta} \tag{1.4}$$

un'istanza di questa formula viene creata ed aggiornata per ogni singolo flusso secondario e permette al sitema di controllo di accelerare se in ritardo o di rallentare se in anticipo. Tale formula richiama la teoria di controllo di sistemi retroazionati negativamente [2], ma non è di facile formalizzazione in quanto la reotrazione agisce sulla variabile di stato $\alpha$ del sistema e non direttamente sull'input. Si è quindi trovato più utile, in fase implementativa, utilizzare il *Metodo Monte Carlo* per la determinazione di $\beta$ che incide su tre fattori qualitativi del modello:

- **Il tempo di risposta del sistema.** Il ritardo o nel caso contrario l'anticipo che il flusso secondario deve accumulare rispetto al target aumenta all'aumentare di $\beta$.
- **Il valore assoluto di desincronizzazione.** Un piccolo valore di $\beta$ rende la variazione di $\alpha$ troppo repentina e permette scostamenti notevoli tra le velocità del flusso target e dei flussi secondari. Questo provoca il superamento di uno dei limiti imposti dal problema: ritardi sempre inferiori a 18 millisecondi (1/32 di battuta a 100 BPM).
- **La desincronizzazione a regime.** Un grande valore di $\beta$ non permette la rilevazione di una desincronizzazione minore dello stesso da parte del sistema. A regime quindi se rimane una distanza che non ha alcun effetto di variazione su $\alpha$, tale distanza non verrà mai azzerata dato che $\alpha(n) = \alpha(n-1)$.

Uno studio empirico riguardo il valore di $\beta$ verrà presentato nel terzo capitolo.

### 1.1.3 Ulteriori affinamenti

Una volta definito il modello base, è possibile l'attuazione di modifiche al fine di abbassare sia le distanze massime gestite, sia il tempo massimo per raggiungere la sincronia.

Il modello di base risponde in maniera consona per variazioni non troppo decise e continue nel tempo. Queste variazioni sono tipiche quando si ha a che fare con interventi dell'utente attraverso un'interfaccia. È possibile introdurre però qualche idea per la gestione di variazioni

istantanee della velocità che portano a discontinuità anche importanti andando ad attivare una **variazione di $\beta$ nel tempo**. Si può ricorrere, per un breve intervallo temporale, ad una variazione del coefficiente in modo che la risposta sia più decisa in presenza di aumenti molto repentini della desincronizzazione. Questa scelta risulta ancora più consona quando la si combina con l'**individuazione diretta di $\alpha$**. Il modello può infatti conoscere la velocità effettiva del target con un passo di ritardo utilizzando la seguente accortezza:

$$v(n-1) = \frac{T_{tg}(n-1) - T_{tg}(n-2)}{\Delta_{b_r}} \tag{1.5}$$

dove $v$ è appunto la velocità istantanea precedente, $T_{tg}(n)$ il Tag temporale del file target all'istante $n$[7] e $\Delta_{b_r}$ il numero di battute che dovrebbero essere trascorse in caso di velocitá unitaria[8].

In questo caso noi potremmo accelerare in maniera consistente quando si trova una variazione di distanza pesante e, quando la distanza rientra entro una soglia prestabilita, forzare la velocita al valore di $v(n-1)$. In questo modo si cerca di recuperare velocemente la discrepanza temporale non eccedendo in senso contrario (ossia accelerando troppo in caso di ritardo o rallentando troppo in caso di anticipo).

Un ultimo elemento, non certo di minore importanza, che può essere soggetto a miglioramento riguarda l'entità della **distanza a regime**, essa infatti si può addirittura rendere nulla se si accetta una diminuzione controllata della qualità del segnale di uscita. Dalla (1.4) si desume che un ritardo $\Delta_b$ che renda vera la disequazione $\frac{\Delta_b}{\beta} < 1$ non ha più alcun effetto sulla velocità, una variazione forzata della stessa in questa situazione non risolverebbe il problema[9]. Guardando bene il modello ci si accorge però che $\Delta_b(n)$ non può essere diminuito solamente da una variazione di velocità.

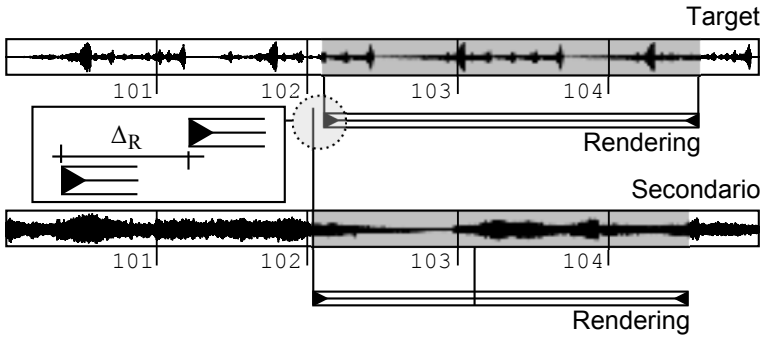


**Figura 1.10:** *Sincronia attraverso spostamento del buffer di rendering*

[7]Notare che possiamo avere accesso a quello di $n-1$ ed $n-2$, ma non ovviamente a quello di $n$.

[8]Il valore è a tutti gli effetti la lunghezza in battute del frame di rendering.

[9]Azzerare la distanza intervenendo solo sulla velocità è molto improbabile perchè una modifica anche minima porterebbe quasi sicuramente ad una compensazione di posizione maggiore rispetto alla stessa distanza. In questo modo si incorrerebbe in una situazione oscillatoria che si risolverebbe solo dopo un tempo troppo lungo.

Guardando la figura (1.10) ci si accorge che spostando in avanti il buffer che renderizza i file secondari[10] si può diminuire la distanza senza intervenire sulla velocità. Ecco che quando le distanze rimangono fisse per un certo periodo (supponiamo 5 o 6 cicli di rendering) si può ritenere che le velocità abbiano lo stesso valore del target e sia quindi possibile portare a zero la distanza attraverso questo metodo. A regime quindi la distanza può venir considerata nulla.

## 1.2 Un possibile nuovo formato

Il flusso target può essere creato in real time: in questo caso l'utente varia nel tempo il valore dei BPM del brano e tutti i flussi secondari seguono questa variazione. È però possibile salvare e trasmettere questo flusso in modo che altri flussi secondari si sincronizzino. Sebbene la creazione delle variazioni non sia più in tempo reale, tutti i flussi secondari continuano a variare man mano che vengono caricati in memoria. Per fare questo viene proposto un nuovo formato che possa salvare le informazioni necessarie al modello di sincronia.

### 1.2.1 Struttura del file

Come abbiamo visto i Tag temporali di battuta sono salvati a distanza regolare, tale distanza coincide con il tempo minimo di aggiornamento della variazione di velocità del flusso audio equivalente quindi alla latenza che esiste tra il controllo manuale di variazione di $\alpha$ e l'effettiva applicazione di questo parametro nell'algoritmo di time stretching.

Esistono due fondamentali modifiche che si devono applicare alla struttura standard di un file PCM[11].

**Header**.

Nell'header va contenuto il valore di latenza con cui il flusso é stato costruito. Questa operazione permette di identificare l'esatta posizione dei Tag temporali che abbiamo definito in precedenza attraverso il semplice calcolo $x_t(n) = \frac{l \times n \times SR \times N_{ch}}{1000}$ con il valore $l$ rappresenta la latenza in millisecondi. A questo punto è bene sottolineare che, sebbene la latenza influisca sulla struttura del formato, nulla ci impedisce di poter utilizzare flussi a diversa latenza. L'idea è quella di interpolare linearmente in valori dei Tag temporali per poter avere valori intermedi del tutto validi: ricordiamo infatti che la velocità in fase di creazione del flusso può variare solamente in corrispondenza dell'annotazione degli stessi Tag e quindi tra un Tag e l'altro essa è del tutto costante e i Tag intermedi sono quindi facilmente ricalcolabili. Il valore di latenza non è quindi una limitazione di interoperabilità, ma un semplice metadato. Va notato che in un flusso target non viene salvato nessun valore rappresentante i BPM, nemmeno quello del flusso originale che

[10] L'entità dello spostamento deve essere tenuta bassa per impedire la presenza di discontinuità troppo pesanti nel tempo che produrebbero dei *click*. Il valore di *salto* potrebbe essere addirittura valutato in relazione alla costanza del segnale in quel punto o ricercando zone simili del segnale nel tempo.

[11] Si veda appendice A.1

è stato utilizzato per la sua creazione. Questo valore deve essere invece presente nei flussi secondari.


**Figura 1.11:** *Header nuovo formato*

**Flusso**.

Il flusso di dati che rappresentano i dati audio sarà quindi una collezione seriale di pacchetti costituiti da insiemi di campioni audio il cui numero varia al variare di valori scritti nell'header: latenza, Sample Rate e numero di canali.

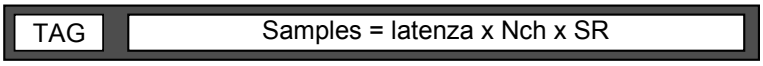


**Figura 1.12:** *Struttura del flusso target*

Ogni volta che un frame di rendering viene creato nel flusso target, il programma dovrà annotare un Tag temporale all'inizio dello stesso. Nelle prove di implementazione l'impatto di overheading è molto contenuto: per un flusso generato da un file PCM stereo campionato a 48kHz con latenza 10 millisecondi risulta dello $0.2\%$.

### 1.2.2 Metadati aggiuntivi e nuove funzionalità

Risulta interessante notare che vi sono ulteriori potenzialità del modello che si possono utilizzare aumentando l'overhead.

**Periodo di pausa**.

Supponiamo che una parte dello spartito sia costituita da una lunga serie di pause, negli spartiti musicali questo viene indicato con una sola battuta dove viene indicato il numero di battute

di pausa. Questo compatta la notazione musicale facilitandone la lettura. Il flusso appena descritto permette di comportarsi in modo del tutto analogo con un chiaro risparmio di spazio in memoria del programma: l'accorgimento prevede di segnare un Tag *pausa*[12] nel punto in cui iniziano le battute di pausa seguito dal numero esatto di frame che si intende rendere impliciti. Da quel punto in poi si scrivono tanti Tag temporali quanti sono i frame che conterrebbero il silenzio.


**Figura 1.13:** *Tag speciale di pausa*

Grazie all'informazione di quanti frame sono dati per impliciti il sistema riesce poi a ritornare al suo funzionamento classico rimanendo sincronizzato. I Tag temporali al contrario dei dati di rendering non possono essere resi impliciti: la variazione temporale potrebbe benissimo continuare anche se i dati di rendering del flusso target sono momentaneamente in pausa.

**Ripetizioni**. È anche possibile, attraverso l'introduzione del nuovo Tag *ripetizione* seguito da un Tag *posizione* e da uno *durata*, poter riutilizzare dei pezzi di brano che si ripetono durante l'esecuzione. In questo caso però i Tag temporali andrebbero modificati in memoria con l'aggiunta di un *offset* che tenga conto della traslazione nel tempo che va effettuata per l'utilizzo di dati precedenti. Quando si utilizzano i Tag temporali successivi alla ripetizione si possono avere due distinti casi:

- i Tag tengono conto della ripetizione precedente e possono essere utilizzati in maniera diretta.
- i Tag non tengono conto della ripetizione precedente e devono essere utilizzati con l'aggiunta di un offset pari alla lunghezza di tutte le ripetizioni giá utilizzate.

Bisogna porre l'attenzione sul fatto che, per quanto concerne questo modello, riutilizzare pezzi di brano precedenti implica di utilizzarne anche le variazioni temporali. In altre parole le distanze tra i Tag temporali non possono essere modificate e, nei riutilizzi, le variazioni di velocità devono essere tutte uguali. Pur essendo presente questa limitazione, l'accorgimento permette di risparmiare notevolmente sulla dimensione del flusso audio target.


**Figura 1.14:** *Tag speciale di ripetizione*

[12]Nei test è stato utilizzato il valore massimo rappresentabile. Utilizzare il valore nullo potrebbe portare problemi nel caso in cui si sia implementato anche l'utilizzo delle *ripetizioni* ed esista, a posizione nulla, un pezzo di brano riutilizzabile.

# Capitolo 2

# Time Stretching

Sincronizzare due flussi audio in questa sede significa variare la velocità del flusso secondario al fine di posizionionarlo sulla stessa battuta del flusso target. Per modificare il tempo di un brano non basta però variare l'andatura di riproduzione dello stesso. Ne ha dato dimostrazione Ross Bagdasarian, pianista armeno-statunitense, che nell'estate del 1958 con il brano "*Witch Doctor*" presentava il suo primo esperimento commerciale, caratterizzato da un cantato stridulo e ad alta frequenza, ottenuto velocizzando tracce audio. Bagdasarian aveva dato voce a *Theodore*, *Simon* e *Alvin*: i Chipmunks[1]. È evidente che l'obiettivo del presente lavoro è proprio quello di evitare il risultato che ricercava Bagdasarian; ovvero si intende variare la velocità e di conseguenza la durata di una registrazione in modo che l'ascoltatore non abbia la sensazione che il brano sia trasposto di tonalità, bensì che sia semplicemente suonato con un tempo musicale diverso. Questo è un tema ben conosciuto nell'elaborazione numerica del segnale poiché interessa molti campi contemporaneamente:

- Ricerca veloce all'interno del materiale audio contenuto nelle sempre più fornite librerie digitali.
- Conversione tra standard video a diverso frame rate.
- Variazione della velocità del parlato per permettere anche ai non vedenti di poter rapidamente cercare elementi all'interno di un testo.
- Modifiche del tracciato elettro cardiaco per diagnosi accurate: modificare la struttura del segnale in questo caso renderebbe inutile lo strumento stesso dato che molte diagnosi si basano sulla forma del segnale rilevato.

Dato l'ampio raggio di applicazione esistono ovviamente più soluzioni che si adattano a scenari diversi tra loro. Questi diversi approcci si possono suddividere in due grandi famiglie: una riguarda le soluzioni nel dominio del tempo, mentre l'altra nel dominio della frequenza.

[1] È interessante vedere che il modello matematico per la generazione di parlato descriva l'aumento di velocità della registrazione con una modifica del filtro del tratto vocale. Questa modifica viene istintivamente percepita dall'ascoltatore come un *rimpicciolimento* del cantante.

**Nel tempo**.

Questi algoritmi funzionano ritagliando il segnale di ingresso in frame di brevissima durata e cercano di ottenere un segnale di uscita ad alta qualità attraverso il calcolo di correlazione tra i frame (SOLA, WSOLA ...), identificando il tono principale presente in ingresso e cercando di rispettarlo in uscita (PSOLA) o con altre metodologie altrettanto costose dal punto di vista computazionale. Tuttavia questi procedimenti, pur dando risultati ad alta qualità[2] non si adattano alla natura interattiva del progetto qui illustrato.

**Nella frequenza**.

Uno dei principali strumenti nel dominio della frequenza è il *Phase Vocoder*. Questo strumento [3], come vedremo in seguito in maniera più approfondita, produce il segnale di uscita partendo dalla *Short Time Fourier Transform* (STFT) del segnale di ingresso, modificandola e creandone una di sintesi. La quantità di tempo che occorre per la sintesi del segnale voluto in questo modo si abbassa notevolmente grazie all'utilizzo di algoritmi veloci come la Fast Fourier Transform[4], rendendo ottimale questa soluzione per la modifica temporale real time. Sfortunatamente questo approccio introduce degli artefatti nel segnale di uscita che possono essere descritti come la presenza di un forte riverbero o, nei casi più critici, come una vera e propria "destrutturazione".

Questo capitolo descrive il Phase Vocoder nelle sue varie versioni: pur essendo infatti uno strumento abbastanza datato nella sua formulazione originaria, è ancora oggetto di attenzioni nella ricerca attuale che ne propone continui affinamenti.

## 2.1 Phase Vocoder: il concetto

Nel 1928 Homer Dudley lavorava nei laboratori Bell ad un progetto che cercava un metodo per poter generare elettromeccanicamente il parlato umano in modo da poterlo trasmettere a grandi distanze con maggiore chiarezza. Il fulcro del suo lavoro si tradusse in un sistema di filtri passabanda in parallelo e alla nascita del *Channel Vocoder*. Channel deriva dal fatto che ogni filtro passabanda si occupa di una certa frequenza del suono da trasmettere, Vocoder sta invece per *codificatore di voce* e ricorda quale fosse l'intento principale. Il Phase Vocoder [5] è molto simile al Channel Vocoder di Dudley e permette la modifica dei segnali audio attraverso tre diversi passaggi che riguardano principalmente la modifica della fase nel dominio delle frequenze.

Vediamo ora di analizzare i diversi passaggi così da illustrare in modo completo il funzionamento dello strumento.

[2]Le tecniche nel tempo sono solitamente utilizzate per segnali audio semplici come possono essere il parlato o strumenti monofonici. Basti pensare all'idea insita nel PSOLA dove si cerca di individuare il tono principale del segnale di ingresso. Anche per questo queste soluzioni non possono essere utili al progetto.

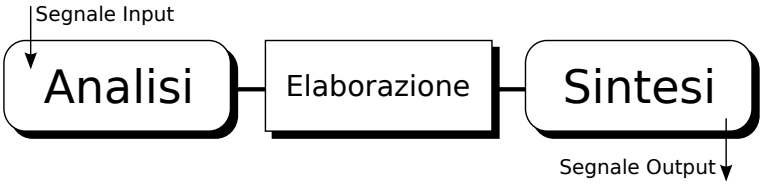


**Figura 2.1:** *Struttura concettuale del Phase Vocoder*

**Analisi**.

Se si calcolasse la Fast Fourier Transform (FFT) di tutto il segnale audio in ingresso, otterremmo un segnale complesso, non dipendente dal tempo, il cui modulo rappresenta solamente la presenza delle componenti armoniche e delle loro ampiezze. Le informazioni relative ai momenti in cui queste componenti siano o meno presenti sono tutte contenute nella fase e risultano di lettura quasi impossibile. Per questo, durante lo stadio di analisi, si calcola la Short-Time Fourier Transform (STFT) [6].

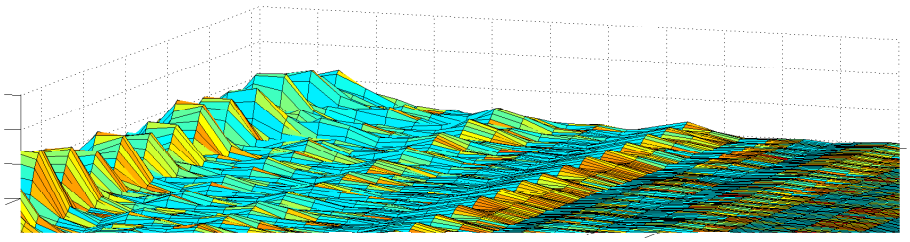

**Figura 2.2:** *Rappresentazione grafica del modulo della STFT*

$$X(t_a^u, \Omega_k) = \sum_{n=-\infty}^{\infty} h(n)x(t_a^u + n)e^{-j\Omega_k n} \tag{2.1}$$

con $\Omega_k = \frac{2k\pi}{N}$.
Nella scrittura $x$ è il segnale di ingresso, $h(n)$ è la finestra di analisi e $\Omega_k$ è la frequenza centrale del canale k-esimo. È opportuno sottolineare alcuni aspetti dell'equazione: prima di tutto, sebbene la sommatoria sia tra $-\infty$ e $+\infty$, la stessa viene limitata nel numero dei termini dall'estensione della finestra $h(n)$ che solitamente è di lunghezza $N$. In altre parole si divide il segnale in ingresso in frame[3] di lunghezza fissa presi ad intervalli regolari $t_a^u = uR_a, u \in \mathbb{N}$, se ne calcola di ognuno la FFT e si ottiene una rappresentazione complessa $X$ dipendente sia dal tempo che dalla frequenza $\Omega_k$ rappresentante l'evoluzione temporale di ogni singolo canale della FFT dotato di fase e modulo. $R_a$ viene definito passo di analisi. In altre parole la STFT è una FFT che viene aggiornata ogni $R_a$ sample e viene calcolata in un intorno centrato su $t_a^u$.

[3]Come vedremo in seguito questi frame sono sottoinsiemi non disgiunti della totalitá dei campioni del segnale di ingresso.

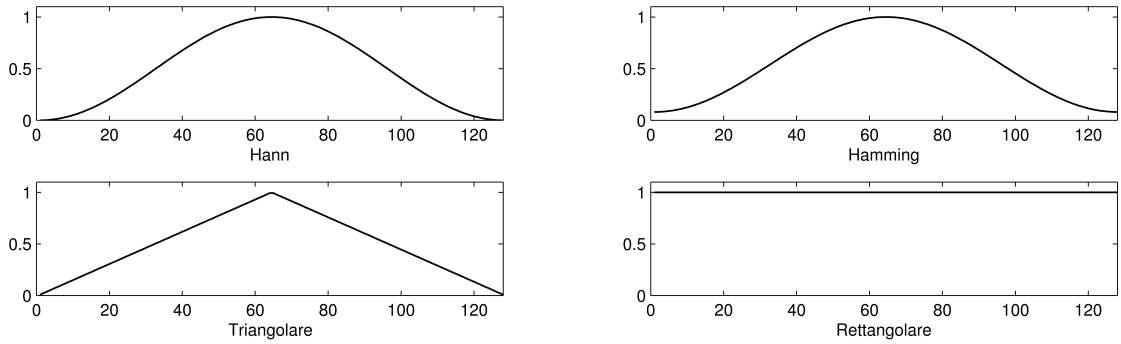


**Figura 2.3:** *Finestrature nella fase di analisi*

**Modifica**.

Nel secondo passo della struttura si attuano gli algoritmi specifici per la modifica del segnale. Il Phase Vocoder deve il suo nome al procedimento che modifica le informazioni di fase lasciando invariato il modulo: in questo modo non modifica *quanto* sia presente una componente spettrale, ma piuttosto *come* essa intervenga. L'effetto più semplice da applicare con un Phase Vocoder è ad esempio chiamato *Robotizer*, in questo caso la fase viene semplicemente azzerata. Il risultato è un suono che sembra essere costituito da impulsi[4] con tono dato dal valore di $R_a$.

**Sintesi**.

In questo ultimo step, viene utilizzata la STFT che abbiamo ottenuto dalle modifiche applicate nel passo di elaborazione $Y(t_s^u, \Omega_k)$. Ogni elemento della STFT dovrà essere utilizzato per costruire un segnale di uscita nel dominio del tempo. Ogni piccolo frame temporale di sintesi andrà a costruire il segnale di uscita dipendendo da un passo di sintesi che indicato con $R_s$ che permette di equispaziare i frame di STFT nell'*Overlap Add* (OLA) finale.

$$y(n) = \sum_{u=-\infty}^{+\infty} w(n - t_s^u) y_u(n - t_s^u) \tag{2.2}$$

con

$$y_u(n) = \frac{1}{N} \sum_{k=0}^{N-1} Y(t_s^u, \Omega_k) e^{j\Omega_k n} \tag{2.3}$$

Il procedimento OLA permette di sovrapporre i frame temporali ottenuti dall'antitrasformazione al fine di produrre un segnale di uscita nel dominio del tempo. È importante notare che non è certo

[4]In effetti il Phase Vocoder crea sempre suoni partendo da impulsi, ma azzerando la fase questo risulta esplicito anche all'ascoltatore dato che si creano discontinuità ad intervalli costanti.

che il segnale prodotto abbia come STFT proprio $Y(t_s^u, \Omega_k)$: questo è dovuto all'intersezione non nulla che solitamente si applica anche nell'OLA[5]. La scrittura $w(n - t_s^u)$ indica un'ulteriore finestratura applicabile in sintesi che solitamente viene mantenuta opzionale.

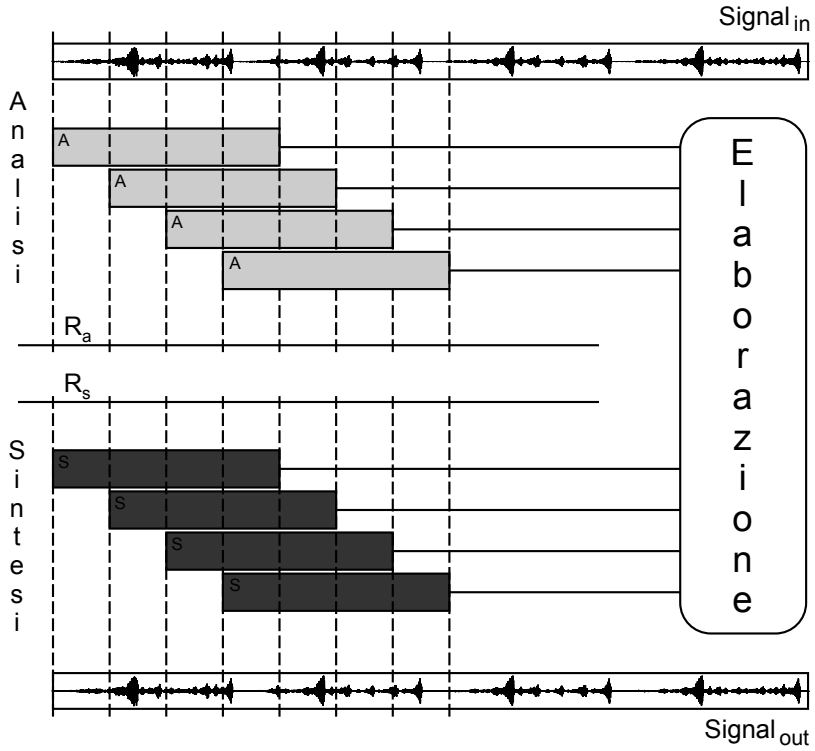


**Figura 2.4:** *Le tre fasi: Analisi, Elaborazione e Sintesi.* $(R_s = R_a)$

[5]Per equivalere infatti dovremmo creare il segnale nel dominio del tempo con $R_s = N$, solo in questo modo ricalcolando la STFT del segnale di uscita otterremmo $Y(t_s^u, \Omega_k)$, ma questo solitamente non accade dato che produrrebbe una pulsazione regolare data dalla forma della finestra di analisi scelta.

## 2.2 Phase Vocoder: time stretching

Una volta illustrato il funzionamento di base del Phase Vocoder, si comprende come il segnale in uscita sia ottenuto attraverso la somma di brevi suoni nel dominio del tempo. Se noi non modificassimo la STFT di analisi e applicassimo l'OLA definito nel terzo passo forzando l'uguaglianza $R_a = R_s$ otterremmo la funzione identità. Il nostro intento è invece quello di cambiare la durata e quindi il tempo musicale del segnale di ingresso [7]. La funzione identità deve quindi essere modificata in due punti:

1) il passo di analisi $R_a$ deve essere differente dal passo di sintesi $R_s$

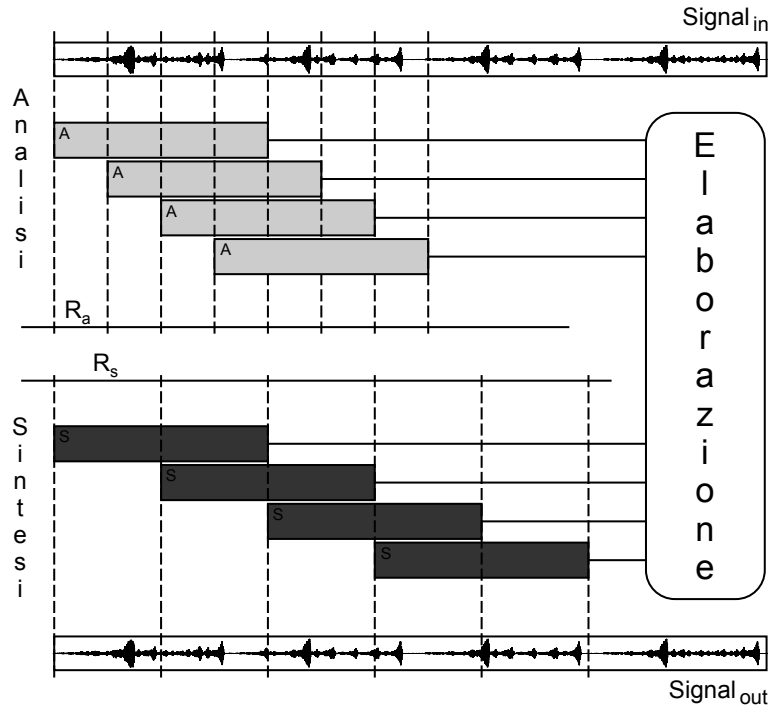


**Figura 2.5:** *Lo stretching temporale.* $(R_s \neq R_a)$

2) vanno eliminate le discontinuitá di fase create dalla diseguaglianza tra il passo di analisi e quello di sintesi.

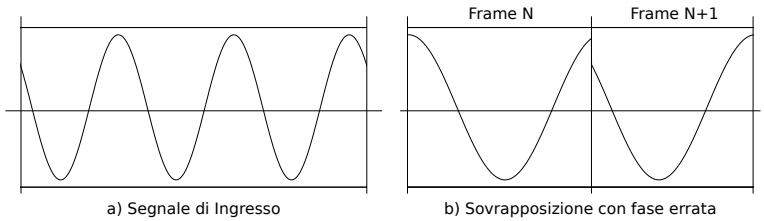


**Figura 2.6:** *Visualizzazione errore di fase nel time stretching*

Per illustrare efficacemente il problema bisogna far riferimento al modello che prevede l'uso di sinusoidi per la sintesi di qualsiasi segnale numerico. Il segnale di ingresso può infatti essere scritto come somma di un numero $I(t)$ di sinusoidi caratterizzate dalle ampiezze $A_i(t)$ e frequenza istantanea $\omega_i(t)$.

$$x(t) = \sum_{i=1}^{I(t)} A_i(t) e^{j\phi_i(t)} \tag{2.4}$$

con $\phi_i(t)$ la fase istantanea della i-esima sinusoide. Il valore di $\phi_i(t)$ è direttamente derivabile dai valori precedenti della frequenza istantanea:

$$\phi_i(t) = \phi_i(0) + \int_0^t \omega_i(\tau) d\tau \tag{2.5}$$

con $\phi_i(0)$ fase iniziale della sinusoide i-esima.

## 2.3 Calcolo della fase

Risulta fondamentale impedire le discontinuità di fase viste nelle figure del punto precedente, per far ciò dobbiamo cercare di sintetizzare delle nuove fasi di sintesi che le eliminino. Come si è visto in precedenza, il primo passo per ottenere una modifica della scala dei tempi è quello di usare un salto di sintesi diverso da quello di analisi, il rapporto tra i due salti $\alpha = \frac{R_s}{R_a}$ indica la modifica temporale ottenuta: se $\alpha > 1$ allora si avrà una contrazione dei tempi e quindi un aumento della velocità del brano, se $\alpha < 1$ si avrà invece una dilatazione dei tempi e quindi una diminuzione della velocità del brano. La fase di sintesi in accordo con $\alpha$ sarà:

$$\phi_s(t_s^u) = \phi_s(0) + \int_0^{t_s^u} \omega_i(\tau/\alpha) d\tau = \phi_s(0) + \alpha \int_0^{t_a^u} \omega_i(\tau) d\tau \tag{2.6}$$

e quindi

$$\phi_s(t_s^u) = \phi_s(0) + \alpha[\phi_i(t_a^u) - \phi_i(0)] \tag{2.7}$$

dove $\phi_s(0)$ è la fase iniziale di sintesi la cui natura è squisitamente arbitraria.

Come si è detto il segnale STFT è di tipo complesso e quindi può essere studiato nel suo modulo $|X(t_a^u, \Omega_k)|$ e nella sua fase $\angle X(t_a^u, \Omega_k)$. Anche se $\alpha \neq 1$, l'uso del Phase Vocoder prevede la modifica della sola fase. Tale caratteristica è concorde con la teoria: non stiamo variando la presenza delle sinusoidi che compongono il segnale, ma stiamo modificando le modalità con cui interagiscono. Per questo si può semplicemente porre $|X(t_a^u, \Omega_k)| = |Y(t_s^u, \Omega_k)|$. Tutto il lavoro che verrà presentato da qui in avanti si concentrerà quindi sui valori di sintesi della fase.

L'idea è quella di utilizzare l'incremento di fase tra due frame consecutivi della STFT di analisi per stimare la frequenza istantanea delle sinusoidi costituenti il segnale. Indicando con $\hat{\omega}_k(t_a^u)$ la frequenza istantanea del canale k-esimo nell'istante $u$ di analisi abbiamo:

$$\Delta\Phi_k^u = \angle X(t_a^u, \Omega_k) - \angle X(t_a^{u-1}, \Omega_k) - R_a\Omega_k \tag{2.8}$$

$$\Delta_p\Phi_k^u = \tan_2(\cos\Phi_k^u, \sin\Phi_k^u) \tag{2.9}$$

$$\hat{\omega}_k(t_a^u) = \Omega_k + \frac{1}{R_a}\Delta_p\Phi_k^u \tag{2.10}$$

Per prima cosa con la (2.8) si calcola l'incremento di fase, con la (2.9) se ne prende la determinazione principale e con la (2.10) si deriva la frequenza della sinusoide più vicina. Con questo procedimento si ottiene il valore attuale di frequenza istantanea della sinusoide che si scosta del valore $\Delta\Phi_k^u$ da $\Omega_k$. In questo modo si riesce così a stimare la frequenza istantanea all'istante $t_a^u$ e, attraverso la *Phase-Propagation Formula*[8], è possibile predire la fase istantanea in caso di modifica dei tempi:

$$\angle Y(t_s^u, \Omega_k) = \angle Y(t_s^{u-1}, \Omega_k) + R_s\hat{\omega}_k(t_a^u) \tag{2.11}$$

Questo calcolo è l'accorgimento base del Phase Vocoder nel caso si voglia attuare un time stretching di un segnale. La formula permette la *conservazione della coerenza di fase orizzontale*, termine che deriva dalle proprietà del sonogramma standard. Scorrere orizzontalmente questa rappresentazione grafica vuol dire spostarsi nel tempo mantenendo costante il canale FFT di riferimento, in questo modo la coerenza orizzontale di fase certifica la possibilità di sovrapporre correttamente pezzi di sinusoide la cui frequenza rimane costante nel tempo

### 2.3.1 Limitazioni d'uso

La (2.11) non funziona in qualsiasi contesto, ma solo nel caso in cui il canale k-esimo venga influenzato da una sola sinusoide. Il tutto si traduce nella ricerca di una buona risoluzione in frequenza e quindi, a meno di non ricorrere allo zero padding, in un corretto dimensionamento della finestra di analisi. Un altro limite importante deriva dal fatto che la frequenza di taglio $\omega_h$ della finestra di analisi deve assumere valori tali da verificare $R_a\omega_h < \pi$. Le finestre standard utilizzate in questi modelli (Hanning, Hamming, ...) assicurano questo quando le stesse si sovrappongono almeno al 75% nella fase di analisi.

### 2.3.2 Problematiche di resintesi

La conservazione di fase orizzontale non è l'unica proprietà che si deve garantire al fine di ottenere una variazione temporale di buona qualità. Un altro elemento che riguarda la generazione di artefatti sonori è quello della non *conservazione della coerenza di fase verticale*. La coerenza di fase verticale assicura il corretto time stretching di segnali composti da sinusoidi le cui frequenze rimangono costanti durante il tempo. Difficilmente un segnale che descrive un brano musicale gode di questa proprietà, quindi è verosimile pensare che esistano componenti che si spostano da un canale FFT all'altro. Nel modello matematico del Phase Vocoder vi saranno allora delle componenti che migrano da una frequenza all'altra tra un frame STFT e il successivo. La Phase-Propagation Formula non tiene conto di questo fattore e sono quindi possibili presenze di discontinuità di fase che potrebbero portare a veri e propri battimenti delle componenti spettrali[6].

[6]È proprio questo che si descrive con *destrutturamento* del suono: le componenti sinusoidali si spostano una rispetto all'altra quando variano di frequenza e quindi, perdendo simultaneità, *spalmano* l'energia del segnale nel tempo.

Sfortunatamente, per segnali che descrivono brani musicali, non vi sono relazioni facilmente deducibili tra fasi di canali adiacenti quindi non c'è possibilità di verificare in modo semplice la coerenza verticale.

Al fine di comprendere più a fondo il problema delle coerenze di fase è importante cercare una relazione tra la fase di sintesi e la fase di analisi. Per fare questo applichiamo iterativamente la (2.11) e cerchiamo di comprendere come gli errori si propagano. Il rapporto di compressione sia sempre definito come $\alpha = \frac{R_s}{R_a}$ e la fase iniziale $\phi_s(0,k)$:

$$\angle Y(t_s^u, \Omega_k) = \phi_s(0,k) + \sum_{i=1}^{u} R_s \hat{\omega}_k(t_a^i) \tag{2.12}$$

usando la (2.9) si ottiene:

$$\angle Y(t_s^u, \Omega_k) = \phi_s(0,k) + \sum_{i=1}^{u} \left[ R_s \Omega_k + \frac{R_s}{R_a} \Delta_P \Phi_k^i \right] \tag{2.13}$$

ricordando che $\Delta_P \Phi_k^i$ è la determinazione principale

$$\angle Y(t_s^u, \Omega_k) = \phi_s(0,k) + \alpha \sum_{i=1}^{u} \left[ \angle X(t_a^i, \Omega_k) - \angle X(t_a^{i-1}, \Omega_k) + 2m_k^i \pi \right] \tag{2.14}$$

dove $2m_k^i\pi$ è il *fattore di unwrapping* che va aggiunto nell'istante $t_a^i$ per poter avere la determinazione principale di $\Phi_k^i$. Si nota subito che i termini intermedi della sommatoria si annullano e rimangono solo gli estremi, mentre i fattori di unwrapping invece rimangono:

$$\angle Y(t_s^u, \Omega_k) = \phi_s(0,k) + \alpha \left[ \angle X(t_a^u, \Omega_k) - \angle X(0, \Omega_k) \right] + \alpha \sum_{i=1}^{u} 2m_k^i \pi \tag{2.15}$$

siamo giunti così all'espressione della fase di sintesi nell'istante $t_s^u$ in funzione della fase iniziale di sintesi $\phi_s(0,k)$, della fase di analisi nell'istante $t_a^u$, del fattore di compressione $\alpha$ e della serie di fattori interi $m_k^i$. Attraverso questa scrittura possiamo comprendere meglio gli errori che vanno ad incidere ad ogni passo sulla valutazione della fase di sintesi.

Innanzitutto si nota che la fase di sintesi dipende solamente dalla fase di analisi iniziale e da quella corrente, ciò significa che se vi sono errori nella valutazione delle fasi di analisi questi non si propagano, ma si annullano nell'iterazione della formula. Un errore invece che viene propagato ad ogni passo riguarda i coefficienti interi $m_k^i$, ne deriva che se viene fatto un errore nel calcolo nella determinazione principale dell'angolo, tutti i frame di sintesi successivi saranno ugualmente errati. Questi errori valgono $2\alpha\pi$ e sono quindi correlati al tipo di stretching sviluppato: se $\alpha$ dovesse essere intero (ad esempio quando la dilatazione temporale viene raddoppiata) avremmo errori multipli di $2\pi$ dando origine ad un errore nullo. Se $\alpha$ è intero non si deve quindi calcolare l'argomento principale dato che il fattore di unwrapping non introduce alcun errore.

Se invece $\alpha \in \mathbb{R}^+ \setminus \mathbb{N}$ questi errori deteriorano gravemente la coerenza verticale di fase: basti pensare agli errori generati da canali FFT influenzati da rumore bianco (fase e modulo casuale) per per una porzione limitata nel tempo.

### 2.3.3 Valore iniziale della fase di sintesi

Abbiamo visto che il calcolo della fase di sintesi corrente viene influenzato in maniera importante dalla fase iniziale di sintesi $\phi_s(0,k)$. Sfortunatamente è possibile valutarne il valore ottimale solo nel caso in cui $\alpha$ sia un intero, al contrario se questo coefficiente è reale non è possibile scegliere un $\phi_s(0,k)$ che possa minimizzare l'errore di propagazione sulla coerenza di fase. L'inizializzazione standard prevede $\phi_s(0,k) = \angle X(0,\Omega_k)$ in questo modo risulta possibile partire da un segnale non modificato ($\alpha = 1$) e giungere a uno modificato senza introdurre discontinuità di fase[7].

## 2.4 Phase Vocoder Avanzati

Una volta compresi i limiti insiti nel modello matematico del Vocoder, si inseriscono modifiche che ne possano migliorare il funzionamento. In questa sezione verranno illustrate tecniche avanzate per aumentare la coerenza di fase verticale che rappresenta il problema cruciale per l'implementazione standard.

È corretto sottolineare che non esistono solo algoritmi di time stretching che si occupano unicamente di modificare la fase, ma ne esistono altri che prendono in considerazione solo i valori del modulo. Questi algoritmi sono indicati come *a solo modulo* e procedono in modo iterativo al fine di determinare una insieme di fasi valide partendo da una rappresentazione STFT a solo modulo [9]. In generale questi metodi sono molto più onerosi dal punto di vista computazionale di quelli proposti in questo capitolo dato che la loro qualità è direttamente proporzionale al numero di iterazioni fatte.

### 2.4.1 Loose Phase Locking

Come abbiamo visto la coerenza di fase verticale è fondamentale e consiste nel non creare discontinuità in una sinusoide che migra da un canale FFT all'altro tra un frame STFT e il successivo. Diventa importante quindi controllare la relazione che esiste tra le fasi di canali FFT adiacenti, se infatti non viene tenuta in considerazione nel modello si presentano errori nell'output soprattutto riguardanti l'ampiezza[8] e si comprende come i riverberi artefatti del vocoder siano generati da battimenti creati dai canali FFT adiacenti che hanno discontinuità di fase. L'approccio del Loose Phase Locking [10] è quello di utilizzare per la sintesi solo quei canali FFT che contengono i picchi dello spettro e di collegare ad essi tutte le fasi dei canali adiacenti. In questo modo si cerca di aumentare sensibilmente la coerenza di fase nelle transizioni di una sinusoide da un canale FFT e un altro. Il procedimento è alquanto semplice e prevede di calcolare le fasi di sintesi come nel modello base del phase vocoder, ma di non usarle direttamente cercando di pesare i valori

[7] Uguagliare la fase iniziale di sintesi a quella iniziale di analisi è fondamentale in questo progetto perché il Phase Vocoder dovrà lavorare il real-time. Questo vuol dire che la variazione di $\alpha$ avviene tramite intervento umano e quindi molto spesso il brano musicale viene fatto partire proprio con $\alpha = 1$ e solo in seguito modificato dall'utente.

[8] Se vengono fatti test utilizzando due diversi toni puri si ottengono errori in ampiezza diversi provando come questi ultimi siano intimamente collegati alla natura del segnale.

ottenuti in modo che esista la coerenza di fase voluta tra canali adiacenti con la seguente formula:

$$\angle Y(t_s^u, \Omega_k) = \angle \left[ Y(t_s^u, \Omega_{k-1}) + Y(t_s^u, \Omega_k) + Y(t_s^u, \Omega_{k+1}) \right] \tag{2.16}$$

Supponiamo ora che nel canale $k$ sia presente un picco. Il modulo del numero complesso associato sarà quindi maggiore di quello dei due canali adiacenti e, in questo modo, la somma complessa avrà fase molto vicina a quella del picco. Viceversa se in $k$ non è presente un picco, la fase che verrà assegnata cambierà e si allineerà con quella del canale FFT più eccitato. In altre parole l'idea è quella di collegare la fase dei canali limitrofi a quella del picco adiacente[9]. I pesi dati ai canali adiacenti sono solitamente unitari, sperimentalmente si osserva che il risultato finale non sembra dipendere troppo dalla pesatura dato che pare coincidere anche se viene cambiata la proporzione da 1:1 a 1:2 o 2:1.

Nell'applicazione implementata seguendo questa teoria, i test sono stati eseguiti applicando la modifica in modo che essa potesse essere attivata o meno a livello di runtime, è dimostrato che poteva essere aumentata la fedeltà di sintesi rispetto al modello base comprendendo che il suo punto debole risiede proprio nel non controllare la coerenza verticale di fase del segnale di uscita. Questi miglioramenti sono però molto più pronunciati se vengono utilizzati con segnali vocali e questo mal si addice al nostro scopo: la semplicità computazionale è sicuramente una forte attrattiva, ma l'avanzamento qualitativo quando si utilizzano segnali che rappresentano strumenti polifonici, è fortemente legata al tipo di segnale e quindi al tipo di brano che si sta modificando.

### 2.4.2 Phase Locking

L'idea di base del Loose Phase Locking è sicuramente interessante, ma non tiene conto della struttura mutevole di un segnale di ingresso. La semplicità dell'algoritmo, oltre ad essere il suo principale vantaggio, risulta essere anche il suo punto debole proprio perché il modello non tiene conto di una diversa pesatura dei valori rappresentati nei canali FFT dell'analisi, ma al contrario tratta tutti i canali nello stesso modo ottenendo così una qualità di output aleatoria. Altri lavori [11] hanno quindi proposto analisi più accurate del segnale di ingresso [12] [13] in modo da poter adattare meglio il lock di fase al segnale. Questa scelta viene implementata in due nuovi algoritmi l'*Identity Phase Locking* e lo *Scaled Phase Locking* che si occupano di ristabilire la coerenza di fase verticale non in modo asintotico come il Loose Phase Locking, ma in maniera più precisa e veloce.

#### 2.4.2.1 Identity

Al fine di adattarsi allo spettro del segnale in ingresso, l'Identity Phase Locking fa precedere al passo di sintesi una ricerca dei massimi relativi nel modulo del frame di STFT corrente. L'algoritmo quindi, prima di procedere alla sintesi del nuovo frame STFT di uscita, procede ad un

[9] In realtà questo avviene in modo asintotico: mentre l'algoritmo funziona esso mette mano alla fase del frame precedente e del successivo che andranno ad influire i passaggi seguenti. In questo modo la fase viene variata lentamente verso quella dei picchi e non in modo istantaneo.

rilevamento[10] dei picchi e da essi definisce delle zone di influenza associati ai picchi stessi. Il confine delle zone solitamente è posto o alla frequenza mediana tra due picchi consecutivi oppure alla frequenza associata al minimo valore del modulo della FFT tra i due picchi.

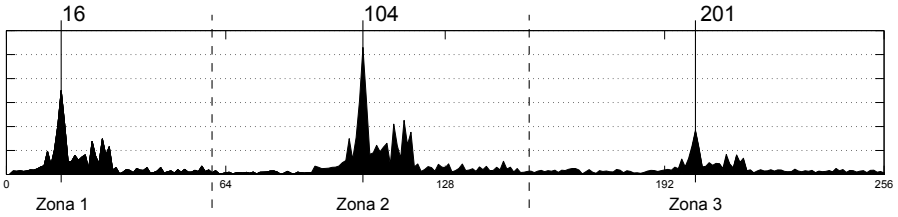


**Figura 2.7:** *Suddivisione dello spettro in zone di influenza*

Una volta ottenuta questa mappatura, si può procedere con la (2.11) con l'accortezza di applicarla solo ai picchi e non a tutti i valori del frame STFT. Già da questo si comprende che oltre ad essere migliore l'algoritmo che stiamo descrivendo sia anche meno costoso dal punto di vista computazionale rispetto al Loose Phase Locking. Dal calcolo della fase di sintesi dei singoli picchi si possono dedurre tutte quelle relative alle fasi di tutti i valori della STFT del segnale senza passare più attraverso la Phase-Propagation Formula. Supponiamo infatti che $\Omega_k$ sia la frequenza su cui stiamo lavorando e che questa frequenza appartenga alla zona di influenza riferita al picco $\Omega_{k_l}$, si avrà che:

$$\angle Y(t_s^u, \Omega_k) - \angle Y(t_s^u, \Omega_{k_l}) = \angle X(t_a^u, \Omega_k) - \angle X(t_a^u, \Omega_{k_l}) \tag{2.17}$$

Questa formula indica che la distanza tra la fase di analisi del picco e quella di un qualsiasi canale facente parte della sua zona di influenza rimane costante anche in sintesi. In questo modo riscrivendo esplicitamente il calcolo della fase di sintesi del canale k-esimo abbiamo:

$$\angle Y(t_s^u, \Omega_k) = \angle Y(t_s^u, \Omega_{k_l}) + \angle X(t_a^u, \Omega_k) - \angle X(t_a^u, \Omega_{k_l}) \tag{2.18}$$

L'idea permette di preservare la coerenza verticale di fase perché i canali adiacenti ad un picco rimangono ad esso solidali (da cui appunto *locked*). Come vedremo la quantità di picchi creati avrà un impatto fondamentale sulla qualità del segnale di uscita, la loro densità è infatti legata solamente alla presenza di massimi relativi più che a nozioni di psicoacustica.

Un ulteriore vantaggio di questo modo di procedere è la possibilità di rilassare la sovrapposizione delle finestre di analisi[11] (da un 75% al 50% della propria lunghezza) perché, essendo il *phase unwrapping* calcolato solamente sui picchi, la frequenza istantanea della sinusoide è sicuramente vicina al canale in questione. In questo caso la limitazione indicata precedentemente come $R_a\omega_h \ < \pi$ ammette valori di $R_a$ doppi rispetto a prima.

[10] Il metodo più semplice per identificare i picchi è quello di identificare i canali FFT dove il modulo della trasformata è maggiore dei suoi quattro vicini (i due precedenti e i due successivi) in questo modo i picchi saranno al massimo $\frac{N}{5}$ dove $N$ è la lunghezza del frame STFT corrente.

[11] Come si vedrà nel terzo capitolo, diminuire la sovrapposizione delle finestre di analisi significa diminuire sia il costo computazionale (esistono infatti meno finestre di analisi a paritá di lunghezza del flusso in input), ma anche il costo in memoria. Supponiamo infatti di dover assicurare un 75% di sovrapposizione: se volessimo caricare tutte le finestre in memoria equivarrebbe caricare 4 volte i dati contenuti nel flusso; con un 50% invece la memoria si dimezzerebbe.

**Tabella 2.1:** *Passaggi algoritmo Identity Phase Locking*

| Passo | Descrizione |
|---|---|
| 1 | Identificare i picchi per ogni frame STFT. |
| 2 | Per ogni picco calcolare la frequenza istantanea e da questa ricavare la fase di sintesi. |
| 3 | Calcolare l'angolo di rotazione $\theta = \angle Y(t_s^u, \Omega_{k_l}) - \angle X(t_a^u, \Omega_{k_l})$ e il fasore $Z = e^{j\theta}$. |
| 4 | Applicare la rotazione a tutti i canali della zona di influenza del picco $Y(t_s^u, \Omega_k) = ZX(t_a^u, \Omega_k)$. |
| 5 | Ripetere il procedimento per tutti i picchi presenti. |
| 6 | Procedere al prossimo frame. |

#### 2.4.2.2 Scaled

Un successivo miglioramento dell'algoritmo si riesce ad ottenere osservando che se un picco migra dal canale $k_0$ nel frame $u-1$ al canale $k_1$ nel frame $u$ l'equazione di *unwrapping* (2.10) dovrebbe essere basata più su $\angle X(t_a^u, \Omega_{k_1}) - \angle X(t_a^{u-1}, \Omega_{k_0})$ piuttosto che su $\angle X(t_a^u, \Omega_{k_1}) - \angle X(t_a^u, \Omega_{k_1})$ e la stessa Phase-Propagation Formula (2.11) dovrebbe essere riscritta come:

$$\angle Y(t_s^u, \Omega_{k_1}) = \angle Y(t_s^{u-1}, \Omega_{k_0}) + R_s \hat{\omega}_{k_1}(t_a^u) \tag{2.19}$$

Infatti l'accumulo di fase va calcolato a partire dal punto di partenza e non da quello di arrivo. L'affinamento porta quindi al problema di determinare quale picco nel frame $u-1$ corrisponda al picco $\Omega_{k_1}$ nel frame $u$. Per risolvere questo passaggio si cerca di individuare la zona di influenza che avrebbe contenuto $\Omega_{k_1}$ nel frame $u-1$. Data la struttura delle zone di influenza, risulta infatti improbabile che un picco possa spostarsi molto velocemente tra di esse tra frame e frame; la fase di sintesi di un canale appartenente alla zona di influenza attuale può quindi essere calcolata come:

$$\angle Y(t_s^u, \Omega_k) = \angle Y(t_s^u, \Omega_{k_l}) + \beta \left[\angle X(t_a^u, \Omega_k) - \angle X(t_a^u, \Omega_{k_l})\right] \tag{2.20}$$

dove $\Omega_{k_l}$ si riferisce all'attuale canale di picco e $\beta$ è il *fattore di scala di fase*. Come è facilmente comprensibile questo aumenta sensibilmente il peso computazionale dell'algoritmo sebbene si riesca ad ottenere una qualitá maggiore.

Il valore di $\beta$ eguaglia quello di $\alpha$ se quest'ultimo è un intero, per valori reali di $\alpha$ invece si propende per $\beta \approx \frac{2}{3} + \frac{\alpha}{3}$. In tutti i casi comunque $\beta \in [0, \alpha]$.

### 2.4.3 Multirisoluzione e traiettorie euristiche

#### Multirisoluzione

Sebbene il calcolo delle fasi di sintesi sia migliorato, questi procedimenti tendono a creare artefatti aggiuntivi sul suono come *filtrature* dello stesso o una perdita di definizione tra le diverse frequenze che può essere percepito come una perdita di definizione delle stesse. Questo non

**Tabella 2.2:** *Passaggi algoritmo Scaled Phase Locking*

| Passo | Descrizione |
|---|---|
| 1 | Identificare i picchi per ogni frame STFT. |
| 2 | Per ogni picco localizzare il corrispondente picco nel frame precedente, calcolare la frequenza istantanea e derivarne la nuova fase di sintesi attraverso la (2.19). |
| 3 | Usare la funzione di unwrapping sulle fasi di analisi dei canali FFT che appartengono alla zona di influenza. |
| 4 | Per ogni canale, appartenente alla zona di influenza, calcolare la differenza di fase di analisi con il picco e usare la (2.20 per calcolare la fase di sintesi corrente. |
| 5 | Ripetere il procedimento per tutti i picchi presenti. |
| 6 | Procedere al prossimo frame. |

deve stupire, se si analizza bene l'algoritmo in effetti si vede che questo non fa altro che rendere *rigido* lo spettro, è come se si perdessero delle informazioni e di conseguenza se ne diminuisse la definizione[12]. È a tutti gli effetti una compressione. Quando si trattano algoritmi di compressione si deve procedere con nozioni di psicoacustica in modo che i dati eliminati non creino effetti innaturali che siano identificabili dall'ascoltatore. Questo fa comprendere come l'aspetto più problematico dell'Identity e dello Scaled Phase Vocoder sia la determinazione dei picchi e delle zone di influenza. È ben documentato [14] che l'orecchio umano nella sua parte interna pratichi una conversione frequenza-posizionamento e che la risoluzione in frequenza non sia lineare, ma quasi-logaritmica. Questo viene del tutto ignorato dagli algoritmi precedenti e crea quindi una perdita di alcune informazioni sensibili trattenendo magari altre riguardo picchi di frequenze che sono percepite come identiche dall'ascoltatore. Oltre che la psicoacustica, si deve tener conto della diversa concentrazione media di frequenze nei brani musicali: le basse frequenze sono normalmente molto più ricche delle alte. Gli algortmi visti potrebbero legare tra loro due frequenze fondamentali. Non basta quindi definire picco quel canale FFT che risulta più eccitato dei suoi quattro vicini, ma si deve dividere lo spettro in zone in cui ci si aspettano quantità diverse di picchi [15]. Per una STFT di dimensione 4096 si possono assumere valide le seguenti considerazioni:

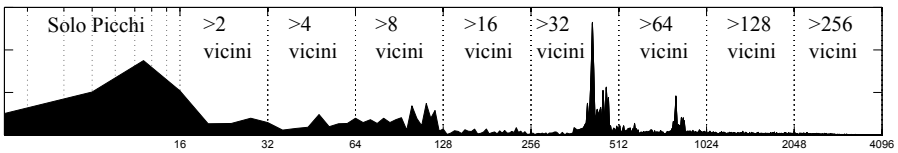


**Figura 2.8:** *Rilevamento dei picchi in multirisoluzione*

[12]Rendere *rigida* la fase di una STFT equivale a quantizzare le modalità di intervendo delle sinusoidi che compongono il suono. Questo potrebbe incidere sul *quando* una certa sinusoide interviene.

**Tabella 2.3:** *Regole per il rilevamento di picchi a multirisoluzione (STFT Frame 4096)*

| Canali | Definizione di picco |
|---|---|
| 1-16 | Sono tutti picchi udibili. |
| 17-32 | Picco se modulo maggiore dei due vicini. |
| 33-64 | Picco se modulo maggiore dei suoi quattro vicini. |
| 65-128 | Picco se modulo maggiore dei suoi otto vicini. |
| ... | ... |

Questo problema era già stato individuato da Bonada [16] anche se con considerazioni diverse: lo studioso infatti aveva notato che se una bassa frequenza, magari molto importante per la caratterizzazione di un suono, era presente, sarebbe servita una finestratura di analisi più lunga per poterne valutare l'ampiezza correttamente. Questo però portava a livelli di riverbero non sopportabili e l'utilizzo di finestre di analisi di lunghezza variabile. Il problema poteva quindi essere risolto variando la definizione del frame STFT: maggiore per l basse frequenze, normale per le alte.

**Traiettorie euristiche**.

Un altro possibile problema riguarda invece lo Scaled Phase Locking Vocoder. L'algoritmo permette anche grandi spostamenti di canale da parte dei picchi tra un frame STFT e il successivo. Questa facilità di identificare picchi padri nei frame precedenti porta a traiettorie spesso prive di senso. Va anche notato che i comportamenti dei picchi non sono uniformi lungo lo spettro delle frequenze: mentre a bassa frequenza il segnale è spesso molto regolare, ad alta frequenza lo spettro è spesso meno stabile[13]. Questo comporta la creazione di troppi picchi dalla vita media piuttosto breve. Per rimediare a questo bisogna definire un valore di salto massimo in funzione della frequenza del picco stesso. Lo spettro viene suddiviso in zone che permettono, al loro interno, la continuazione della traiettoria dei picchi solo per distanze non maggiori di quelle indicate[15]:

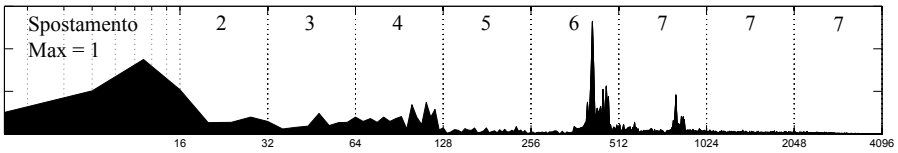


**Figura 2.9:** *Traiettorie euristiche dei picchi*

[13]La presenza di rumore ad alta frequenza gioca un ruolo determinante nella creazione di picchi improvvisi che non fanno parte del segnale utile.

**Tabella 2.4:** *Distanza massima per il tracciamento delle traiettorie*

| Distanza Max | Sottoinsieme canali |
|---|---|
| 1 | 0 - 16 |
| 2 | 17 - 32 |
| 3 | 33 - 64 |
| 4 | 65 - 128 |
| 5 | 192 - 256 |
| 6 | 257 - 512 |
| 7 | Tutti i rimanenti. |
| ... | ... |

### 2.4.4 Silent Phase Reset

Abbiamo visto come sia di fondamentale importanza concentrarsi sul contenimento degli errori di unwrapping delle fasi per poter mantenere una buona coerenza verticale di fase ed evitare i fastidiosi riverberi tipici del Phase Vocoder. Dai calcoli inoltre si evince che siamo in presenza di una deriva della qualità che porta ad una *degradazione* del segnale direttamente proporzionale alla sua lunghezza, le imprecisioni infatti sono di tipo cumulativo e crescono al crescere del numero di frame STFT utilizzate. Sebbene la situazione possa essere migliorata con gli accorgimenti visti, questo problema non può essere eliminato matematicamente se $\alpha \notin \mathbb{N}$. Solitamente però se il segnale di ingresso è notevolmente lungo, è anche statisticamente probabile che vi siano degli intervalli dove forzare la coerenza verticale di fase in modo poco doloroso[14]: se ad esempio l'energia totale di un frame STFT dovesse scendere sotto una soglia[15] prestabilita potremmo pensare che quasi tutto il segnale è rumore di fondo.

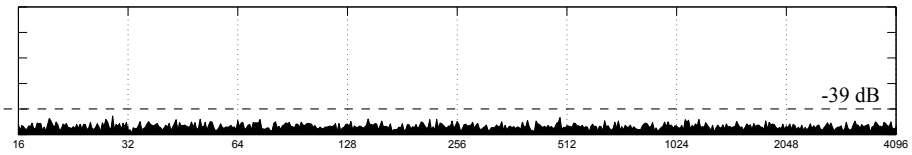


**Figura 2.10:** *Zona di applicazione Silent Phase Reset*

Saperlo ci permette di attendere che il segnale torni a crescere dal punto di vista energetico[16] per porre le fasi di sintesi allo stesso valore delle fasi di analisi e azzerare di colpo l'errore di deriva che causa la perdita di coerenza verticale. Questo intervento genera una discontinuità di fase lungo tutto lo spettro del segnale, quello che viene comunemente chiamato *click*, ma esso avrà un'energia che potremmo controllare attraverso le soglie rendendola così inudibile.

[14]Gli intervalli di cui stiamo parlando sono molto brevi. Un frame di analisi composto da 2048 campioni di un segnale acquisito a 48kHz equivale a circa 43 millisecondi di brano. Trovare un intervallo superiore ad un dodicesimo di battuta in un brano a 100 BPM che soddisfi le caratteristiche ricercate è abbastanza probabile.

[15]Solitamente tra i -20dB e i -30dB.

[16]Questa seconda soglia viene posta ad un decibel di distanza da quella precedente quindi -19dB o -29dB.

### 2.4.5 Altre possibili idee

Prendendo spunto dalle osservazioni precedenti si propongono due possibili ulteriori miglioramenti.

**Reset di traiettoria**.

Con i miglioramenti indicati si riescono a formulare delle regole che riescono a interpretare più intimamente la natura di ogni picco. Infatti se nel frame precedente non si riesce ad individuare un padre, questo ci aiuta a rilevare una sorta di momento di *nascita*, al contrario se nel frame successivo non si riesce a determinare un figlio siamo in presenza della *morte* della traiettoria. L'informazione temporale della nascita può essere fatta coincidere con un reset della fase di sintesi così da cercare di limitare la deriva di coerenza verticale di fase alla sola vita della traiettoria individuata.

**Parziale reset silenzioso di fase**.

Invece di attendere che tutta l'energia di un singolo frame sia sotto una certa soglia, si può estendere l'idea gestendo in maniera del tutto individuale intere bande del segnale. Nel caso in cui un picco nasca in una zona che prima era contrassegnata come *resettabile*, questa tecnica conferisce un reset di fase non solo al picco (come accade teoricamente nel *reset di traiettoria*), ma anche a tutti i canali che fanno parte di quella zona.

### 2.4.6 Qualità e tempi di esecuzione

Data l'abbondanza di modifiche che si possono fare partendo dall'algoritmo base del Phase Vocoder, viene naturale pensare alla creazione di uno strumento matematico per controllare l'effettivo miglioramento o peggioramento della qualità di output. In letteratura sono state già state proposte delle misurazioni che potessero mettere in relazione il segnale in uscita con quello in ingresso ricercando la sicura oggettività di valutazione. In molte altre occasioni invece si fa uso di gruppi di ascolto che decidono a votazione la qualità di ogni modifica. A volte gli ascoltatori sono a conoscenza anche delle stesse modifiche, ma questo non permette di esseri sicuri che le loro decisioni siano affidabili. Ad esempio:

$$D_M = \frac{\sum_{u=P}^{U-P-1} \sum_{k=0}^{N-1} \left[ |Z(t_s^u, \Omega_k)| - |Y(t_s^u, \Omega_k)| \right]^2}{\sum_{u=P}^{U-P-1} \sum_{k=0}^{N-1} |Y(t_s^u, \Omega_k)|^2} \tag{2.21}$$

dove $Z(t_s^u, \Omega_k)$ é la STFT del segnale di uscita[17] e $P$ indica il primo e l'ultimo tratto del segnale che non viene considerato per i transitori dovuti alla mancanza di frame in sovrapposizione.

Questa funzione, proposta la prima volta nel 1984 [9], dovrebbe indicare, per $D_M$ decrescenti, l'aumento della qualità del segnale di uscita. È facile dimostrare che questo non si verifica nemmeno per registrazioni di parlato accostando i valori ottenuti con il Phase Vocoder a quelli

[17] Ricordiamo che il segnale di uscita non ha mai in $Y(t_s^u, \Omega_k)$ la sua STFT.

dello PSOLA[18]. Visti i risultati ci si accorge che la consistenza dello PSOLA risulta essere mi-

**Tabella 2.5:** *Valori di consistenza su parlato*

| Algoritmo | $D_M$ |
|---|---|
| Nessuno | 0db |
| Loose Phase Locking | -11dB |
| Identity Phase Locking | -15dB |
| Scaled Phase Locking | -14dB |
| PSOLA | -9dB |

nore di quella data dall'Identity Phase Locking che risulterebbe addirittura migliore dello Scaled Phase Locking. Risulta quindi abbastanza chiaro che tale valore non dia una buona lettura della qualità di uscita. Si potrebbe comunque provare a utilizzare dei segnali di test (ad esempio lo stesso brano di uno strumento musicale suonato con due BPM diversi) e calcolare $D_M$ non sull'errore energetico rispetto la STFT del segnale di uscita, ma rispetto al segnale di riferimento dilatato nel mondo reale. Questa ovviamente non è una soluzione sempre percorribile.

Va fatta anche qualche considerazione nei confronti dei tempi di esecuzione degli algoritmi. Andare a valutare le prestazioni assolute non ha più molto senso dato che con l'attuale hardware (anche consumer) tutti gli algoritmi visti sono chiaramente eseguibili entro la latenza di 10 millisecondi. Si vuole invece sottolineare come la qualità del segnale di uscita non sia per forza legata ad una complessità computazionale maggiore: il Loose Phase Locking Vocoder si è dimostrato essere molto più esigente dell'Identity Phase Locking Vocoder. Questo fa pensare a quanto sia importante la conoscenza della struttura interna del segnale di input. Probabilmente in futuro, disponendo di modelli sempre più precisi riguardo la generazione dei suoni nel mondo reale, potremmo avere dei Phase Vocoder che scelgono autonomamente l'algoritmo migliore in relazione al tipo di strumento campionato. Informazione che potrebbe essere contenuta in un semplice metadato.

## 2.5 $\alpha$ variabile

Una caratteristica aggiuntiva del Phase Vocoder che utilizzeremo è quella di permettere la variazione di $\alpha$ in tempo reale. Tale possibilità va a complicare la creazione di quello che nel terzo capitolo verrà definito come buffer di rendering. Ora basti sapere che l'OLA finale non viene ovviamente calcolato per ottenere tutto il flusso di uscita, ma solo per la parte che serve. Quindi avremo:

[18]PSOLA sta per Pitch Synchronous Overlap-Add. Viene preso in questo come punto di riferimento perché è un algoritmo nel dominio del tempo molto costoso dal punto di vista computazionale, ma altrettanto accurato nella qualità di output. Si basa sulla divisione dei frame di analisi in relazione a dei marker di tono al fine di mantenerlo invariato nel file di output. [17]

$$X_r(n) = \sum_{u=k}^{k+N} w(n - t_s^u) y_u(n - t_s^u) \tag{2.22}$$

con $y_u(n)$ antitrasformata del frame STFT di sintesi, $N$ il numero di frame di sintesi nel dominio del tempo che occorrono per applicare l'OLA e ottenere il frame di rendering $X_r(n)$ e $k$ l'indice del primo frame di sintesi contenente dati relativi al rendering. La determinazione di $k$ viene demandata al processo di rendering che vedremo in seguito, rimane invece il problema teorico di non creare discontinuità di fase durante la variazione di $\alpha$ in run-time da parte dell'utente.

Il ruolo fondamentale viene giocato proprio dal frame di sintesi $k$-esimo che si rivela il frame di connessione tra il valore di $\alpha$ precedente e quello corrente. Tutti gli algoritmi presentati lavorano al massimo sul frame di sintesi precedente per poter calcolare l'accumulazione della fase che varia al variare di $\alpha$. Ponendo come premessa che la velocitá di esecuzione non venga variata mentre si sta renderizzando $X_r(n)$, ma solo nel calcolo del successivo $X_r(n+1)$[19], si deve individuare il valore di $k$ e mantenere quel frame di sintesi invariato.

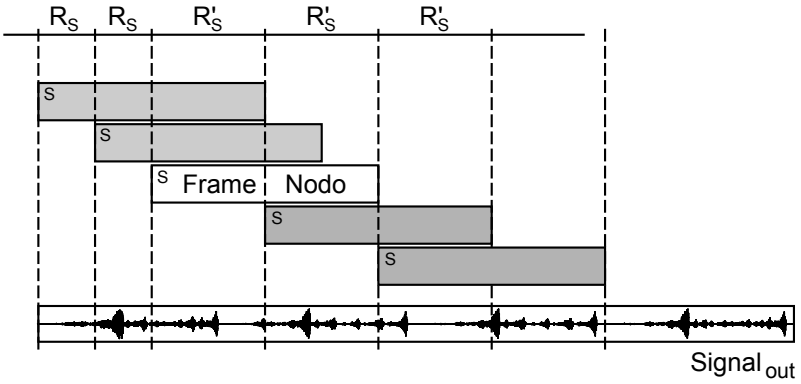


**Figura 2.11:** *Modifica di $\alpha$ e relativo frame nodo*

Sebbene il frame $k$-esimo sia effettivamente calcolato con un $\alpha$ non più valido, esso assicura la continuità di fase con i precedenti. Non rimane che calcolare gli altri $N-1$ frame di sintesi che, per costruzione, assicureranno la continuità di fase con il $k$-esimo. In questo modo si evita di inserire nella formula di propagazione di fase la variazione di $\alpha$ per poter assicurare la continuità di fase andando ad utilizzare il primo frame di rendering dopo la variazione come un viraggio tra il vecchio valore di $\alpha$ e il nuovo. A ben guardare il frame in oggetto non ha un coefficiente di dilatazione temporale ben definito e questa è la direzione verso la quale si è deciso di procedere. È infatti possibile gestire repentine variazioni di $\alpha$ con l'aumento di un grado di libertà dell'algoritmo andando a decidere quanti frame di sintesi dopo il $k$-esimo devono essere ricalcolati o addirittura creare un' interpolazione tra il valore iniziale di $\alpha$ e quello finale. Questo accorgimento permette di lavorare con qualsiasi dimensione di $X_r$ diminuendo di fatto la latenza percepita dall'utente[20].

[19] Questo evidenzia il forte legame tra latenza e lunghezza di $X_r$.

[20] Sebbene la latenza percepita venga migliorata, usando questo metodo con finestre molto grandi e quindi con una

latenza reale molto alta, l'utente percepisce una sorta di inerzia nel cambiamento di velocità del brano. Questo può risultare anche come una possibile personalizzazione nel rendere più dolci le traiettorie di $\alpha$ descritte dall'utente.

# Capitolo 3

# Implementazione

## 3.1 Utilizzo del Phase Vocoder nel progetto

Come abbiamo visto, l'operato del Phase Vocoder è riassumibile nella sintesi di un segnale di uscita come somma di brevi segnali temporali dedotti dal segnale di ingresso e modificati nel dominio della frequenza. Ora vedremo come questo modo di procedere interagisca in maniera quasi naturale con il modello per la creazione e l'utilizzo di Tag temporali descritto nel primo capitolo. Si parte quindi da un'implementazione diretta del Phase Vocoder per poi applicare le modifiche necessarie permettendo così la possibilità sia di creare flussi target che di gestire sincronizzazioni in real time. Questo capitolo, essendo fortemente legato all'implementazione effettiva degli algoritmi visti, porrà attenzione anche al funzionamento degli strumenti scelti per lo sviluppo finale con la descrizione del funzionamento delle librerie WASAPI di Windows.
Il Phase Vocoder sarà dotato di due nuove funzionalità principali:

**Creazione flusso Target**.

Il flusso target può essere generato in real time dall'utente o caricato sotto forma di file utilizzando il formato descritto al termine del primo capitolo. La latenza massima nell'applicazione della modifica temporale, ottenuta con la variazione di $\alpha$, non deve superare i 10 millisecondi.

**Sincronia**.

La sincronia si ottiene modificando in real time la struttura temporale dei flussi secondari utilizzando i Tag temporali come punti di riferimento del flusso target (sia esso generato in real time o caricato da file). Il ritardo massimo permesso è di 18 millisecondi, equivalenti a $\frac{1}{32}$ di battuta a tempo di 100 BPM.

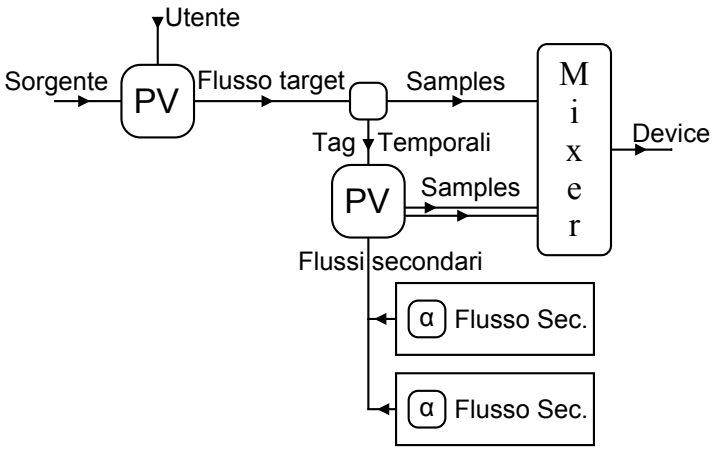


**Figura 3.1:** *Struttura semplificata del programma finale*

I due blocchi sopra descritti possono essere facilmente visualizzati anche nel lavoro che andremo a descrivere. Come si vede in figura esiste un Phase Vocoder adibito alla creazione del flusso target seguendo le modifiche apportate dall'utente[1] e un secondo Phase Vocoder che si occupa della sincronia di ogni flusso secondario rispetto al flusso target.

## 3.2 Implementazione Phase Vocoder

L'implementazione finale è stata scritta in linguaggio *C++* utilizzando l'IDE *Microsoft Visual Studio*. Le operazioni di debug e di testing sono state eseguite su un elaboratore dotato di 4 gigabyte di memoria RAM e di un processore a 64 bit *i7 720-QM*. Per calcolare in modo performante le trasformate di Fourier e le operazioni su segnali complessi sono state utilizzate le soluzioni fornite dalle librerie *Intel Integrated Performance Primitives 6.1.2* specifiche per l'architettura in uso. Un elemento fondamentale per l'interattività è dotare il programma di un'interfaccia grafica che permetta all'utente di modificare i parametri in maniera intuitiva e, per permettere questo, la stessa stesura del codice deve tenerne conto sin dall'inizio. In questa sezione di testo verranno esposte le scelte di sviluppo che sono state attuate nell'implementazione reale dei modelli teorici illustrati durante entrambi i capitoli precedenti. Si cercherà di presentare ogni blocco logico del programma in modo separato anche se, come sarà subito ben visibile, esiste una forte interazione tra tutti loro.

### 3.2.1 La gestione della memoria

Nell'esposizione del modello matematico che definisce l'uso del Phase Vocoder nell'ambito del time stretching, erano state presentate tre sezioni logiche (analisi, modifica e sintesi) che potevano sembrare del tutto slegate tra di loro. Nella realtà invece il discorso è diverso: il primo

[1] Ovviamente tale Phase Vocoder risulta inoperante se il flusso target viene caricato da file o giunge trasmesso da un'altra sorgente.

passo per ottenere una dilatazione o una compressione temporale del segnale di ingresso è quello di porre $R_a \neq R_s$[2] e questo porta a due possibili conseguenze:

1. Variare $R_a$ e mantenere fisso $R_s$.
2. Variare $R_s$ e mantenere fisso $R_a$.

Da un punto di vista matematico, le due scelte sono del tutto equivalenti. Lo stesso non si però dire da un punto di vista implementativo perché esiste un rapporto implicito tra analisi e sintesi che va analizzato con attenzione: la Short Time Fourier Transform (STFT). Supponiamo di scegliere la prima possibilità di sviluppo, presto ci accorgeremmo che ad ogni variazione di $\alpha$ varierebbe il valore di $R_a$ e, dato che la STFT è fortemente legata a questo valore, ad ogni passo ci si ritroverebbe con una STFT di analisi obsoleta. Con la seconda possibilità invece non viene mai modificata la STFT di analisi che può quindi essere computata una ed una sola volta all'inizio dell'applicazione delle modifiche.

Va sempre tenuto conto che questa applicazione deve essere capace di lavorare in real-time e quindi la prima attenzione nello sviluppo deve essere quella di minimizzare i calcoli che vengono fatti in run-time: in questo caso c'è la concreta possibilità di alleggerire il peso computazionale dovuto al time stretching e alla variazione di $\alpha$, attuando il calcolo della STFT di analisi in modalità *offline* e quindi slegandone la validità dall'intervento dell'utente. Come sia stata applicata questa osservazione è riportato nella definizione di preload.

**Preload**.

Le osservazioni fatte sino a questo punto possono essere sviluppate creando una funzione che gestisca i file in ingresso. I tipi di file che possono essere trattati sono fondamentalmente due:

1. Un flusso target caricato da file.
2. Un file Wave PCM candidato a flusso target o secondario.

La funzione voluta entra in gioco solo nella seconda ipotesi quando, oltre a caricare i sample dal file, analizza il segnale che essi definiscono costruendo e caricando in memoria la STFT sin da subito. Sebbene questo rallenti l'operazione di caricamento da disco, si calcolano dati che saranno validi per tutta l'esecuzione del programma. Così facendo una delle tre sezioni logiche del Phase Vocoder è stata risolta mentre l'utente sceglie i brani da utilizzare e non durante l'effettiva modifica di $\alpha$. Ricordiamo infatti che tutto il programma è basato sul valore di latenza: se si superano i 10 millisecondi nella creazione della porzione del segnale di uscita voluto, il processo risentirebbe di una grossa caduta di qualità[3] al punto di divenire inutilizzabile. Liberare risorse nel momento di applicazione delle modifiche al segnale di ingresso permette quindi di poter utilizzare anche gli algoritmi più complessi senza avere effetti collaterali.

[2] Si ricorda che per $R_a$ si intende il salto di analisi ed $R_s$ il salto di sintesi.

[3] Il passo di rendering è di 10 millisecondi, ritardarlo vorrebbe dire lasciare il buffer di memoria identico e quindi creare ripetizioni di segnale molto brevi nel tempo. Questo effetto è chiaramente fastidioso e crea oltretutto una variazione consistente di pitch che va contro l'idea stessa di time stretching con l'uso del Phase Vocoder.

#### 3.2.1.1 Il nodo di memoria

Utilizzare software musicali, che permettono di ottenere risultati ad alta qualità, richiede quasi sempre l'utilizzo di molta memoria primaria da parte dell'elaboratore. È quindi opportuno gestire la memoria in modo da rendere comprensibile quale siano i buffer che devono risiedere in memoria per il corretto funzionamento dell'applicazione e quelli che si possono invece rendere nuovamente disponibili al sistema operativo. Il primo passo per poter ottenere questo risultato è quello di scegliere una struttura dati che si sposi il più possibile con il processo che la andrà ad utilizzare. Il Phase Vocoder gestisce un numero notevole di frame: brevi segnali nel dominio del tempo o della frequenza che rappresentano sia il segnale di ingresso che quello di uscita. Una struttura che ben si presta a questo modello e al rendering di flussi audio è una *lista bidirezionale di nodi*.

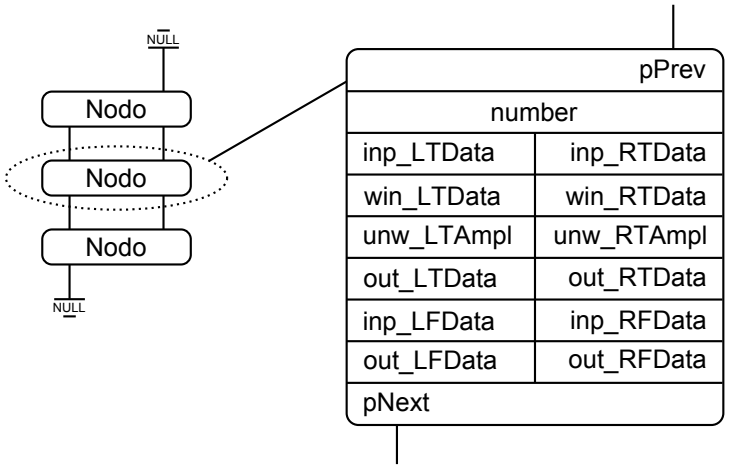


**Figura 3.2:** *Struttura della memoria e nodo di memoria*

In figura viene mostrata la struttura di memoria e la descrizione dettagliata di un nodo per segnali stereo. Il fatto che ogni nodo contenga sia dati di input che dati di output sta nell'interpretazione di una peculiarità del Phase Vocoder che crea una relazione biunivoca tra frame di analisi e frame di sintesi. Ogni frame di sintesi trae informazioni da uno ed un solo frame di analisi.

Al fine di comprendere meglio la struttura interna di un nodo ne riportiamo la descrizione in Tabella 3.1:

**Tabella 3.1:** *Struttura del nodo di memoria* $X \in \{R, L\}$

| Buffer | Descrizione |
|---|---|
| **inp_XTData** | In questo buffer sono salvati i dati del file di ingresso senza alcuna modifica. Essi vengono utilizzati dal programma quando $\alpha$ è unitario e permettono il rendering diretto del file di ingresso. |
| **win_XTData** | Della stessa dimensione del precedente contiene le finestre di analisi del file di ingresso. |
| **unw_XTAmpl** | Questo buffer viene utilizzato per migliorare l'amplificazione in uscita. |
| **out_XTData** | In questa memoria vengono salvati i campioni di uscita nel dominio del tempo. È il buffer dove viene scritta l'antitrasformata del frame di sintesi. |
| **inp_XFData** | I dati contenuti in questo buffer equivalgono al frame di STFT nell'istante corrente. |
| **out_XFData** | Buffer in cui vengono salvate le modifiche al segnale di ingresso nel dominio della frequenza. |
| **number** | Numero progressivo del nodo. |
| **pNext** | Puntatore al nodo successivo. |
| **pPrev** | Puntatore al nodo precedente. |

dove X sta per R nel caso di canale destro e L per canale sinistro[4][5]. La struttura permette di utilizzare gli algoritmi di time stretching implementati fornendo un solo oggetto come parametro: in esso è già pronta la memoria che rappresenta l'analisi e il luogo dove annotare la sintesi. Questo, come vedremo in seguito, snellisce e velocizza il codice finale. Una volta che il nodo è utilizzato[6], tutti i dati contenuti sono deallocabili in blocco proprio perché si è creato un oggetto in sintonia con il modello matematico. Tuttavia si è scelto di lasciarli ugualmente residenti in memoria anche in seguito, per permettere la creazione di strumenti di debug che si sono rivelati molto utili sin dall'inizio.

Un'ultima osservazione va fatta sull'aggettivo **bidirezionale** della lista di nodi: se da una parte il flusso audio porta a chiedere quale sia il frame successivo, la Phase Propagation Formula vista nel secondo capitolo ha bisogno del frame di sintesi che precede quello che si sta sintetizzando al fine di accumulare la fase. È quindi comprensibile come risulti necessario poter scorrere la lista in entrambe le direzioni[7].

[4] Per unw_XTAmpl si veda la sezione titolata "Il problema amplificazione".

[5] Ovviamente non esiste alcuna limitazione rispetto segnali multicanale, si è scelta la rappresentazione stereo a solo scopo espositivo.

[6] In realtà può essere deallocato quando il suo successivo non serve più: la Phase Propagation Formula accumula la fase rispetto al frame di analisi precedente. Tale dato deve chiaramente ancora risiedere in memoria.

[7] Questo osservazione dà uno spunto interessante perché potrebbe essere utilizzato per gestire valori di $\alpha$ anche negativi procedendo a ritroso rispetto la normale direzione di rendering.

### 3.2.2 Dall'analisi alla sintesi

Avendo notato che esiste una forte relazione tra i frame di analisi, che costituiscono la STFT di partenza, e quelli di sintesi, la cui antitrasformata andrà a creare il segnale di uscita, si può procedere con la definizione di una sola funzione[8] per l'applicazione dello stretching temporale su un nodo dato. Questa funzione si presta facilmente alla parametrizzazione del metodo utilizzato nelle varie versioni di Phase Vocoder: viene reso notevolmente semplice modificare, anche in run-time, l'algoritmo utilizzato, cambiando solamente la variabile globale ***TSMode*** e gli switch per le opzioni aggiuntive:

**Tabella 3.2:** *Valori di TSMode e switch di opzione*

| **Valore di TSMode** | **Effetto** |
|---|---|
| TS_NOTHING | Nessun algoritmo time stretching |
| TS_SIMPLE | Algoritmo standard time stretching |
| TS_PHLK_LOOSE | Loose Phase Locking |
| TS_PHLK_ID | Identity Phase Locking |
| TS_PHLK_SC | Scaled Phase Locking |
| **Switch** | **Effetto** |
| SPR | Silent Phase Reset |
| PSPR | Partial Silent Phase Reset |
| MRE | Multirisoluzione |
| ETR | Traiettorie Euristiche |

I valori che possono essere assunti direttamente dalla variabile *TSMode* sono costanti che dichiarano esplicitamente quale algoritmo utilizzare. A questi si aggiungono dei valori booleani che possono attivare e disattivare gli affinamenti descritti alla fine del secondo capitolo. Va inoltre chiarito che il valore iniziale di $R_s$[9] e il tipo di finestra di analisi da utilizzare[10] sono decisi in fase di inizializzazione e preload. Questi due valori non sono quindi generalmente modificabili durante il run-time.

Come si vede in figura (3.3), il metodo prende i dati dal buffer denominato **inp_XFData**, li modifica in accordo con la teoria e crea il buffer **out_XFData** che in precedenza aveva il valore *NULL*. Questa procedura crea il frame nel dominio della frequenza se e solo se il buffer selezionato per la memorizzazione del risultato riporta il valore *NULL*, con questo accorgimento il programma non ricalcola mai un frame di sintesi di uscita se esso risulta già valutato.

In seguito verrà mostrato come la creazione di un buffer di rendering prevede di coinvolgere un numero preciso di frame di sintesi di cui una parte potrebbe essere già stato processata dalla

[8] B.6 Funzione stretchNode.

[9] Risulta molto più comodo variare $R_s$ piuttosto che $\alpha$ pur essendo i due eventi del tutto equivalenti.

[10] Il programma può utilizzare diverse finestre di analisi, ma i risultati migliori si sono ottenuti con la finestra di Hann $w(n) = \frac{1}{2}\left(1 - \cos\left(\frac{2\pi n}{N-1}\right)\right)$

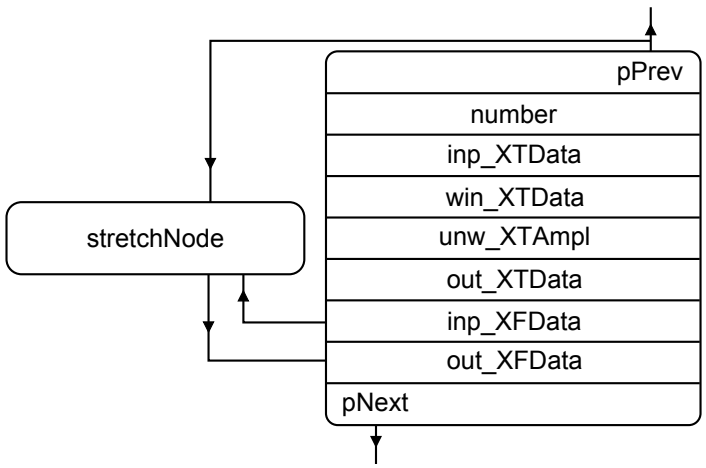


**Figura 3.3:** *Principio di funzionamento del metodo di sintesi di un nodo*

funzione apposita. Se il frame è stato già calcolato è sicuramente diverso da *NULL* e quindi viene lasciato invariato. L'unico caso in cui questa regola non vale è che sia stato variato $\alpha$ e quindi qualsiasi frame di uscita contiene, per un solo passo, dati del tutto obsoleti.
Il fatto che sia la funzione di stretching a decidere se un frame vada ricalcolato o no rende questo evento del tutto trasparente nella successiva implementazione del codice e alleggerisce in automatico il peso computazionale di ogni singolo passo di sintesi, non ricalcolando dati di cui si è già in possesso.

Una volta calcolati i frame di sintesi essi vengono antitrasformati e riportati nel dominio del tempo, tali dati vengono salvati nel buffer **out_XTData** contenuto sempre nello stesso nodo.

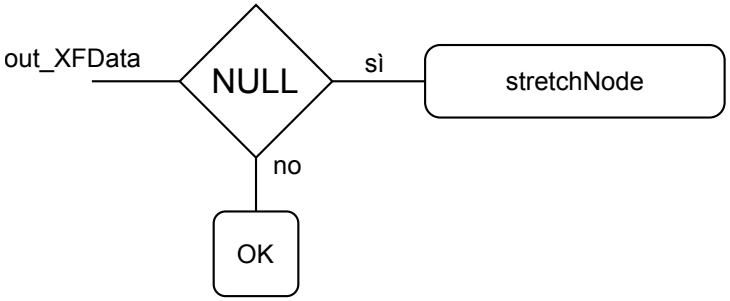


**Figura 3.4:** *Riutilizzo frame di sintesi*

#### 3.2.2.1 Overlap Add

Dopo aver ricavato i frame di sintesi nel dominio del tempo, dobbiamo applicare un *Overlap Add* per creare il frame di rendering. Essendo le distanze tra i frame di sintesi modificabili nel tempo (ricordiamo infatti che mentre $R_a$ è costante, $R_s$ varia nel tempo), come la dimensione

del buffer di rendering[11], varia anche il numero di finestre di sintesi occorrenti per completare un passo di OLA. Per risolvere il problema è stato creato un buffer di puntatori a puntatori di nodi di memoria. La funzione che calcola l'OLA[12] si appropria di questi puntatori (la cui quantità è nota e salvata nella variabile globale *nOLA*) e crea il buffer di rendering basandosi sul valore di $R_s$.

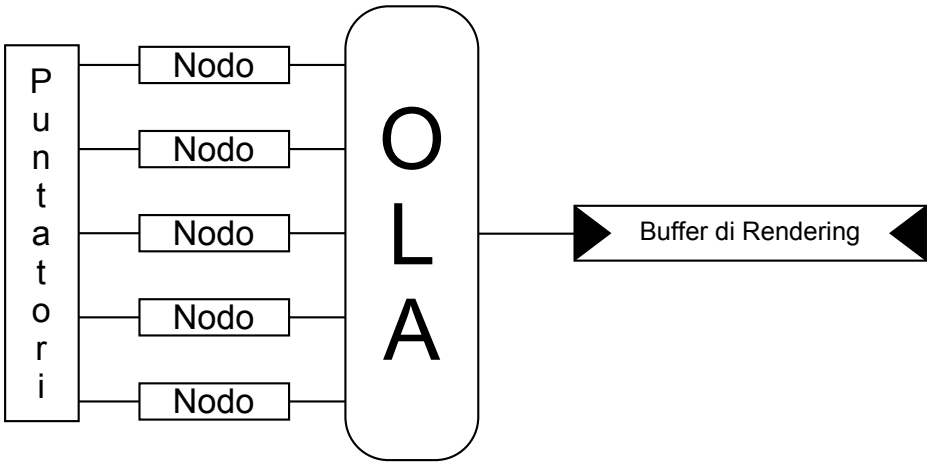


**Figura 3.5:** *Procedimento di OLA per la creazione del buffer di rendering*

Il processo lavora dipendendo solamente dal valore di $R_s$ e dai puntatori ai nodi che sono ritenuti coinvolti nella creazione del buffer di rendering attuale. Nel caso in cui il primo puntatore abbia valore *NULL*, la funzione dichiara *NULL* anche il buffer di rendering, questa scelta verrà utilizzata per fermare il rendering nel caso in cui il flusso audio sia finito[13].

### 3.2.3 Buffer di rendering

È giunto il momento di descrivere come il programma si sposta tra i frame di analisi ottenendo quelli di sintesi dai quali ricavare a sua volta il frame di rendering. Al frame di rendering viene associato un buffer di rendering che deve contenere i dati da renderizzare ad ogni passo. Come abbiamo appena visto, il buffer di rendering viene ottenuto da *nOLA* frame di sintesi nel dominio del tempo i cui puntatori sono sempre disponibili al programma.

Come si vede dalla figura (3.6), il buffer di rendering *scorre* sulle finestre di sintesi e non su quelle di analisi. La nostra implementazione però è stata sviluppata in modo che non esista alcuna differenza di posizione di memoria tra l'analisi e la sintesi e possiamo quindi recuperare il posizionamento del buffer di rendering sia rispetto al segnale in ingresso sia rispetto al segnale in uscita[14].

[11] La sua variazione è contemplata nel caso di variazione del valore di latenza.

[12] B.7 Funzione olaNode.

[13] Se il primo nodo indicato è *NULL* significa che siamo arrivati alla fine della lista bidirezionale di nodi che costituisce la struttura dei campioni in memoria e quindi si deduce che il flusso audio è giunto al termine.

[14] Questo rende molto semplice il conseguimento dei Tag temporali di cui si è parlato nel primo capito essendo resa esplicita a livello di codice la relazione tra flusso di ingresso e flusso modificato.

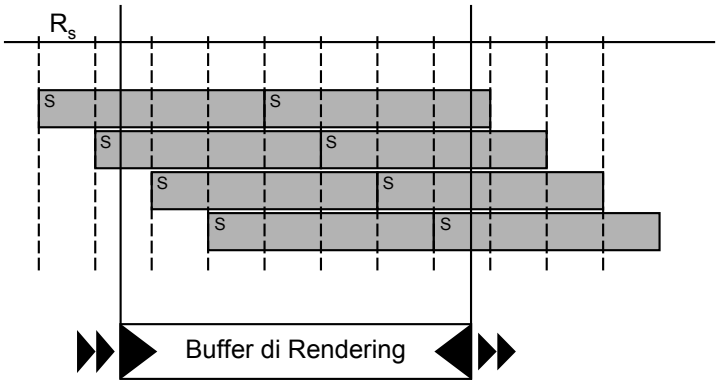


**Figura 3.6:** *Ottenimento del buffer di rendering dai frame di sintesi*

Passiamo ora alla definizione di alcuni termini che serviranno ad esporre lo scorrimento del buffer di rendering sui dati di analisi e di sintesi:

- **Posizione di sintesi di un nodo ($P_{N_S}$).** Viene definita posizione di sintesi di un nodo la posizione che il buffer di sintesi del nodo in questione assume rispetto al flusso audio in uscita. Da notare che, essendo $R_s$ variabile, essa non è ottenibile da nessuna formula diretta.

- **Posizione di analisi di un nodo ($P_{N_A}$).** Viene definita posizione di analisi di un nodo la posizione che il buffer di analisi del nodo in questione assume rispetto al flusso audio in ingresso. Questo valore è ottenibile dal numero del nodo corrente moltiplicato per il valore di $R_a$ che rimane sempre costante.

- **Dimensione di nodo ($D_N$).** Viene definita dimensione di nodo la dimensione dei buffer out_XTData, rappresentante la lunghezza del frame di sintesi nel dominio del tempo.

- **Dimensione di rendering ($D_R$).** Viene definita dimensione di rendering la dimensione del buffer di rendering.

- **Posizione di rendering ($P_R$).** La posizione di rendering è la posizione del frame di rendering rispetto al flusso di output.

- **Nodo in rendering**. Un nodo viene definito come "in rendering" se $P_{N_S} \in [P_R, P_R + D_R] \vee P_{N_S} + D_N \in [P_R, P_R + D_R]$.

- **Nodo fuori rendering**. Un nodo viene definito come "fuori rendering" se $P_{N_S} > P_R + D_R \vee P_{N_S} + D_N < P_R$.

- **Nodo in entrata**. Un nodo viene definito "in entrata" se al passo corrente è "in rendering" mentre al passo precedente era "fuori rendering".

- **Nodo in uscita**. Un nodo viene definito "in uscita" se al passo corrente è "fuori rendering" mentre al passo precedente era "in rendering".

- **Primo Nodo in rendering** ($N_1$). Viene definito *primo nodo in rendering* il nodo che è "in rendering" e ha il numero più basso.

L'unico valore a cui bisogna fare attenzione è $P_{N_S}$. La sua valutazione è ottenibile sommando al $P_{N_S}$ del *primo nodo in rendering* il valore di $R_s$ moltiplicato per $D_{N_S}$, dove $D_{N_S}$ è la distanza in nodi tra il frame corrente e il *primo nodo di rendering*. Tutto questo poichè, variando $R_s$ nel tempo, la posizione di sintesi del frame di rendering nei confronti dell'output non aumenta in maniera regolare. In memoria verrà quindi tenuta come variabile globale il $P_{N_S}$ del *primo nodo in rendering* dal quale possono essere dedotti tutti gli altri.

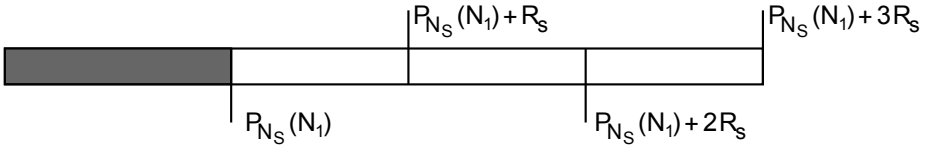


**Figura 3.7:** *Recupero posizione di sintesi dei nodi*

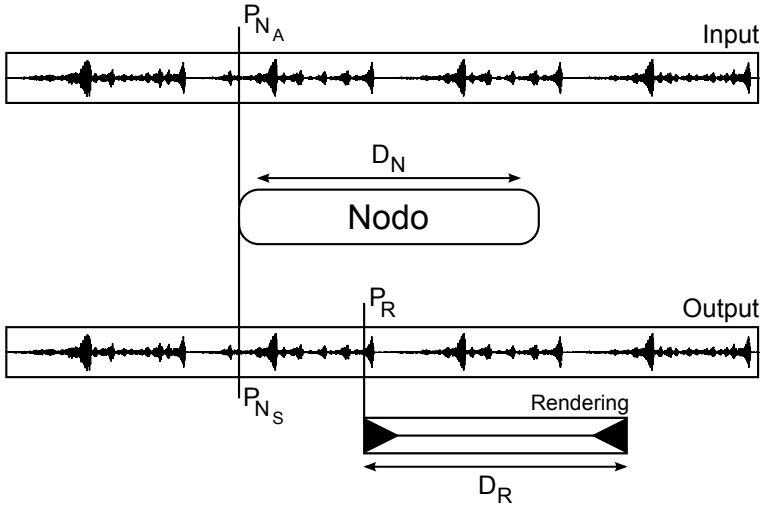


**Figura 3.8:** *Rappresentazione grafica delle definizioni sui nodi*

Quando il buffer di rendering scorre, il valore di $P_R$ viene incrementato esattamente della latenza impostata; vengono controllati quali nodi risultano rispettivamente *in uscita* e *in entrata* e vengono aggiornati i puntatori utilizzati dall'OLA. Viene quindi aggiornato il valore di $P_{N_S}$ del *primo nodo in rendering* tenendo conto del valore di $R_s$ corrente. È l'avanzamento del buffer di rendering che decide quali nodi siano da calcolare e come vadano posizionati per la corretta parametrizzazione dell'OLA.

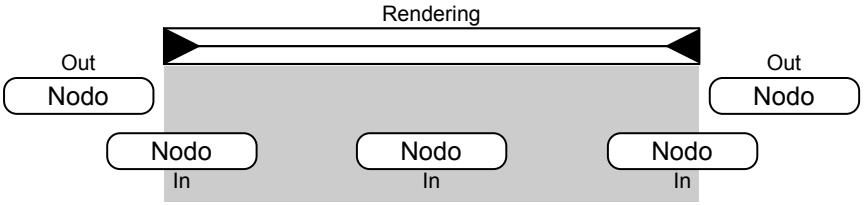


**Figura 3.9:** *Nodi in entrata e in uscita dal buffer di rendering*

#### 3.2.3.1 Impatto di $\alpha$ variabile sul buffer

Il ruolo giocato da $R_s$ e quindi di $\alpha$, in fase implementativa, diviene ancora più importante se si osserva che da esso dipende direttamente il valore di *nOla*. Se diminuisce il salto di sintesi, vorrà dire che servono più frame di sintesi per poter creare un frame di rendering:

$$nOLA(t) = \frac{D_R}{R_s(t)} \tag{3.1}$$

Oltre a notare che *nOLA* varia nel tempo, si vede che per valori molto piccoli di $R_s$ esso cresce in maniera drammatica per una applicazione real time. Un valore elevato di *nOLA* porta infatti a due problemi molto importanti per l'ottenimento di un singolo buffer di rendering:

- **Computazionale**. Sebbene il programma calcoli i soli frame di sintesi che effettivamente servono all'ottenimento dell'intput, un valore di $R_s$ piccolo porta ad un innalzamento del numero di *nodi in entrata* nell'intersezione con il buffer di rendering e quindi aumenta sensibilmente il numero di calcoli da eseguire per ottenere l'output.

- **Risorse**. Anche se la memoria utilizzata può essere drasticamente diminuita liberando i nodi che sono stati utilizzati, i frame di sintesi che servono per l'ottenimento del buffer di rendering devono essere disponibili all'applicazione. Quindi la memoria utilizzata è inversamente proporzionale al valore di $R_s$.

Poiché l'obiettivo principale dell'applicazione è quello di funzionare in real time, questi due problemi si traducono inevitabilmente in un limite inferiore sui valori di $R_s$ e quindi un limite superiore al valore di $\alpha$.

Esiste anche un altro risvolto della questione che è bene analizzare a fondo: il momento della variazione di $R_s$ è un punto di picco nell'utilizzo della potenza di calcolo. Come abbiamo detto nella sezione 2.5 il *primo nodo di rendering* è l'unico nodo che non viene risintetizzato, questo vuol dire che tutti i buffer di sintesi nel tempo degli altri nodi utilizzati per l'OLA vengono resi nulli dall'algoritmo di variazione. In questo caso troviamo che l'elaboratore è tenuto a sintetizzare $(nOla - 1)$ buffer contro un numero minore in caso di $R_s$ costante. Questo non è altro che il *worst case* da analizzare per poter dimensionare correttamente i valori minimi e massimi di $\alpha$[15]. Ovviamente questo valore varia a seconda sia della potenza di calcolo di cui siamo a disposizione, sia dell'algoritmo scelto tramite il valore *TSMode*.

[15]Al fine di decidere i requisiti minimi si deve quindi dedurre $\alpha$ in modo che il numero massimo di volte che viene applicata la funzione di time stretching non necessiti di un tempo di esecuzione sueriore al valore di latenza.

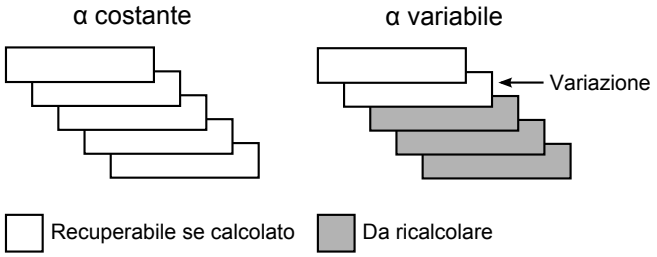


**Figura 3.10:** *Complessità computazionale diversa tra $\alpha$ costante e variabile*

#### 3.2.3.2 Il problema amplificazione

Nella struttura del nodo di memoria è presente anche **unw_XTAmpl** che sta per *unwanted Time Amplification*. Tale buffer serve per contenere i problemi generati dal comportamento che il Phase Vocoder assume nei confronti dell'amplificazione. In figura (3.11) si considera un segnale costante di ingresso $x = 1$ di 768 campioni. In fase di analisi vengono create 20 finestre di Hann lunghe 128 campioni con $R_s = 32$ come viene mostrato in (3.11a). Essendo il segnale costante, la fase di modifica del Phase Vocoder non apporta nessun cambiamento ai frame di sintesi a prescindere dal valore di $\alpha$. Vediamo quindi cosa succede al segnale di uscita con la variazione di $\alpha$.

$\alpha$ **= 1**.
Le finestre di analisi si sovrappongono per il 75% e la lunghezza del segnale di uscita è composto sempre da 768 campioni. Sovrapponendo queste finestre, con il processo di Overlap Add, si ottiene l'amplificazione del segnale di uscita mostrato in (3.11b). Si nota che dopo un primo transitorio, dovuto alla mancanza di frame di analisi precedenti al primo, si ha un'amplificazione costante del 200%. Il finale è caratterizzato da un secondo transitorio creato anche in questo caso da una sovrapposizione mancante di frame successivi all'ultimo.

$\alpha$ **= 1.5**.
Come si è detto in precedenza la STFT non dipende da $\alpha$, quindi la sua antitrasformata coincide ancora con la successione di finestre mostrate in (3.11a). In questo caso però $R_s = 21 \neq R_a$ e, come mostrato in (3.11c), questo comporta un accorciamento del segnale a 512 campioni. In (3.11d) viene mostrato l'andamento dell'amplificazione che, seppur mantenendo la stessa struttura della (3.11b), attesta l'amplificazione a regime al 300%.

$\alpha$ **= 0.4**.
Anche in questo caso la (3.11a) rimane valida. Questa volta però $R_s = 80$ e il segnale di uscita, che è composto da 1920 campioni, ha un'amplificazione descritta dalla figura (3.11f). L'amplificazione presenta un tremolo molto accentuato la cui frequenza è direttamente collegata al valore di $R_s$.

Com'è facilmente comprensibile, questo comportamento non è auspicabile. Se la velocità del

brano viene aumentata, l'ascoltatore percepisce anche un aumento di volume o, nei casi peggiori, anche una saturazione del segnale di uscita. Quando invece la velocità del brano viene diminuita il segnale di uscita viene addizionato di un tremolo che dipende sia in frequenza che in ampiezza dal valore di $\alpha$. Il buffer **unw_XTAmpl** salva al suo interno la forma della finestratura di analisi e permette, nella fase di OLA, di ricostruire l'amplificazione che il Phase Vocoder ha applicato al frame di rendering. Si hanno quindi tutti i dati necessari per rendere l'amplificazione in uscita il più stabile possibile anche per diverse variazioni temporali. Nel caso in cui la finestra di analisi non possa essere modificata in fase di run-time, basta che il programma abbia in memoria una sola di queste. Nella funzione che espleta l'operazione di Overlap Add infatti il programma conosce gli indici di ogni campione nella finestra e può risalire all'amplificazione associata in analisi. L'utilizzo della memoria da parte di unw_XTAmpl è marginale rispetto benefici che questo buffer può invece portare.

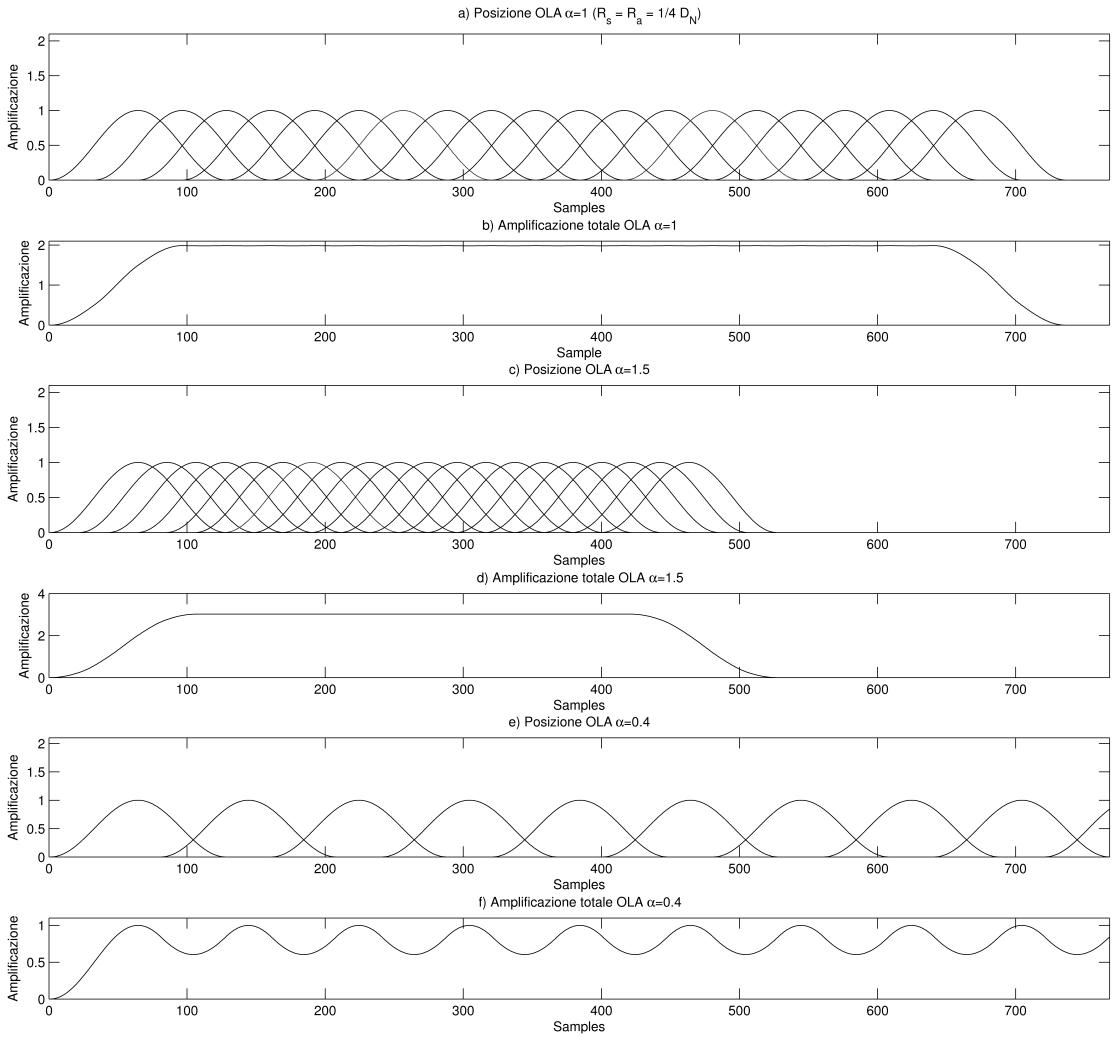


**Figura 3.11:** *Variazione dell'amplificazione in uscita al variare di* $\alpha$

### 3.2.4 Creazione dei Tag temporali

Nel primo capitolo abbiamo visto come nella (1.2) esistano due diverse variabili temporali: $t$ che è collegata all'analisi e $t'$ che è collegata alla sintesi e al rendering. Il Phase Vocoder che stiamo implementando può facilmente applicare queste due diverse scale temporali ai nodi che compongono il buffer di rendering: di ognuno infatti conosce la *posizione di analisi* e la *posizione di sintesi*. La gestione della memoria a nodi facilita dunque la creazione dei Tag temporali descritti in precedenza.

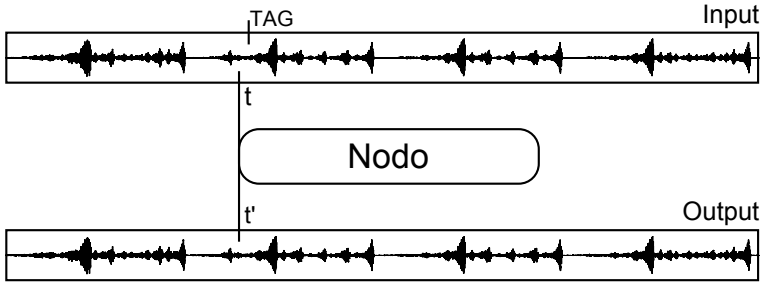


**Figura 3.12:** *Ottenere Tag temporale da Phase Vocoder modificato*

Come si vede in figura, la modifica da apportare per gestire i Tag temporali voluti è resa modesta dalla natura stessa del Phase Vocoder. Come abbiamo più volte detto il Phase Vocoder ritaglia il file di ingresso in piccoli frame per analizzarli e modificarli: numerare questi frame[16], come si è fatto in precedenza, consiste proprio nel creare i riferimenti temporali che stiamo cercando. Da quella che è stata definita nella sezione riguardante il buffer di rendering come *posizione di analisi di un nodo* può essere facilmente derivato un Tag temporale che tenga conto di quanto detto nel primo capitolo.

$$\frac{P_{N_A}(N_1)}{SR} \times \frac{BPM}{60} \tag{3.2}$$

dove $BPM$ non varia nel tempo perché è il valore con cui è stato taggato il file Wave PCM di input[17]. Va notato anche che al contrario della (1.1) non compare il valore $N_{ch}$ che viene direttamente gestito dai nodi di memoria.

Dalla figura (3.13) ci si accorge che la (3.2) non è del tutto corretta dato che non è detto che il buffer di rendering sia sempre allineato con i nodi di memoria:

esiste quindi un *offset* che, oltre ad intervenire all'interno del calcolo del buffer di rendering, può essere utilzzato per creare un Tag temporale perfettamente sequenziato al passo di rendering. Osserviamo infatti che l'offset tra buffer di rendering e frame di sintesi può essere facilmente

[16]Campo number del nodo di memoria.

[17]Ricordiamo che in questo momento stiamo creando dei Tag temporali da file Wave PCM standard che devono solamente avere un Tag BPM fisso. Sono solo i flussi target che permettono di conservare un BPM variabile nel tempo.

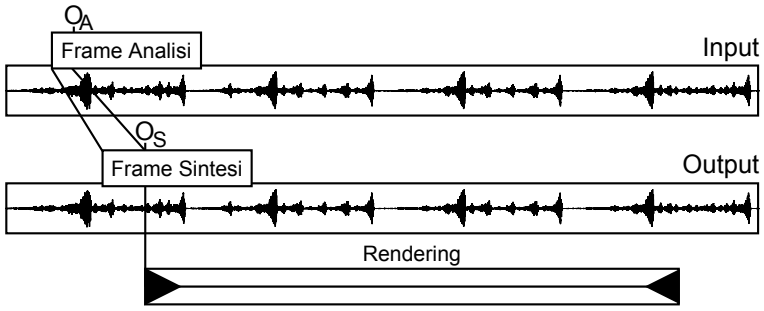


**Figura 3.13:** *Offset tra buffer di rendering e frame di sintesi*

convertito, attraverso il valore corrente di $R_s$, nell'offset tra l'inizio dello stesso buffer di rendering con il frame di analisi: $O_A = O_S/R_s$ con $O_A$ *offset di analisi* e $O_S$ *offset di sintesi*. Il Tag temporale corrente sarà quindi calcolabile come:

$$\frac{P_{N_A}(N_1) + O_S/R_s}{SR} \times \frac{BPM}{60} \tag{3.3}$$

Questo valore dovrà essere calcolato ad ogni passo del buffer di rendering e inserito nel flusso come metadato. Notiamo come la definizione di nodo di memoria contenga al suo interno una relazione implicita tra il flusso di analisi e il flusso di sintesi che rende possibile ed immediata la conversione tra il tempo $t$ e tempo $t'$. In questo modo il Phase Vocoder non sa solamente da quanto sta renderizzando verso il device, ma sa anche l'esatto punto da cui sta attingendo dati dal flusso di input: questo era proprio quello che volevamo ottenere per poter annotare e gestire il valore di battuta del brano.

Il programma quindi utilizzerà gli algoritmi di time stretching e, ad ogni riempimento del buffer di rendering, annoterà la posizione di analisi calcolata rispettando la struttura del file descritto nel primo capitolo. Questo procedimento verrà attuato anche nei *flussi secondari* e i Tag temporali ottenuti serviranno per aggiornare e rilevare la distanza presente tra i diversi flussi in gioco. In altre parole i due Phase Vocoder utilizzati[18] si comunicano le posizioni di analisi in battute e fanno in modo che tutti i brani rispettino tale posizione. Il tempo $t'$ è invece gestito dalla posizione di sintesi e regola la corretta sovrapposizione dei frame di sintesi.

#### 3.2.4.1 Effettuare la sincronia

Ora che abbiamo correttamente creato i Tag temporali voluti, li utilizziamo per ottenere la *sincronia musicale* che ci eravamo proposti sin dall'inizio.

Al fine di ottenere la sincronia ci si deve immedesimare sul secondo Phase Vocoder che gestisce i flussi secondari. Quest'ultimo è del tutto ignaro del lavoro compiuto dalla prima istanza adibita alla creazione del flusso target: quello a cui possiamo accedere è solamente una serie di pacchetti che contengono un frame di rendering, la cui lunghezza è definita dal metadato

[18]Ricordiamo che il primo serve a creare il flusso target, mentre il secondo per gestire la sincronizzazione dei flussi secondari.

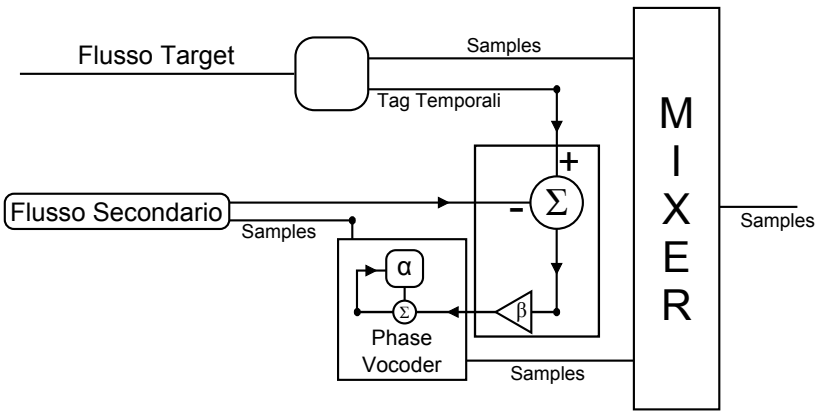


**Figura 3.14:** *Struttura controllo sincronia*

*latenza* contenuto nell'header, e il suo valore di partenza in battute.

La sincronia si ottiene facendo in modo che anche il Phase Vocoder adibito alla sincronia calcoli, ad ogni passo di rendering, la (3.3) per ogni flusso secondario che deve trattare e ne confronti il risultato con quello che proviene dal flusso target. Ad ogni successivo calcolo, il valore $P_{N_A}(N_1) + O_S/R_s$ aumenterà in funzione dell'$R_s$ utilizzato non solo perchè presente in modo esplicito nella formula, ma anche perché una variazione del salto di sintesi crea una diversa quantità di *nodi in uscita* e *nodi in entrata* andando a modificare il valore di $P_{N_A}(N_1)$. Risulta quindi comprensibile come, anche dal punto di vista matematico, sia possibile ottenere la sincronia variando il valore di $R_s$:

- **Aumentando** $R_s$ (diminuire $\alpha$) se il valore calcolato risulta maggiore di quello in arrivo dal flusso target.
- **Diminuendo** $R_s$ (aumentare $\alpha$) se il valore calcolato risulta minore di quello in arrivo dal flusso target.

In altre parole rallentando il flusso che insegue se questo è in anticipo o velocizzandolo in caso contrario. La (1.4) viene quindi tradotta nella:

$$R_s(n) = R_s(n-1) + \frac{T_t - T_s}{\beta} \tag{3.4}$$

dove $T_t$ è il Tag in arrivo dal flusso target e $T_s$ il Tag calcolato mentre si elabora il flusso secondario.

#### 3.2.4.2 La questione $\beta$

Il parametro $\beta$ è importante nel funzionamento del programma. In un primo momento si è cercato un suo valore ottimale, in seguito ci si è accorti che non era opportuno procedere in questo modo. Per uno strumento musicale $\beta$ può essere un paramentro da lasciare all'utente.

**Tabella 3.3:** *Impatto di $\beta$ sul funzionamento Rampa*

| $\beta$ **Battute** $10^{-3}$ | **Ritardo Variazione [ms]** | **Scostamento MAX [ms]** | **Recupero [s]** | **Forma +/-** |
|---|---|---|---|---|
| 0.125 | 20 | 8.56 | >3 | - |
| 0.250 | 20 | 9.06 | >3 | - |
| 0.375 | 20 | 6.31 | >3 | - |
| 0.500 | 20 | 5.10 | >3 | - |
| 0.625 | 20 | 2.65 | 2.22 | - |
| 0.750 | 20 | 2.65 | 2.22 | - |
| 0.875 | 20 | 2.35 | 0.81 | - |
| 1.000 | 20 | 2.35 | 0.81 | - |
| 1.125 | 20 | 2.35 | 0.81 | - |
| 1.250 | 20 | 2.35 | 0.81 | - |
| 1.625 | 130 | 5.90 | 0.41 | + |
| 5.000 | 220 | 11.09 | 0.36 | - |
| 6.250 | 250 | 15.44 | 0.36 | - |

Il rapporto tra $\Delta_b$ e $\beta$ deve essere un numero puro. Questo implica che $\beta$ ha la stessa unità di misura di $\Delta_b$. Per cercare l'influenza di $\beta$ sul programma, variamo il coefficiente lungo l'intervallo deciso e annotiamo quattro paramentri qualitativi:

**Ritardo Valutazione [ms]** Tempo intercorso tra la prima variazione di velocità del file di target e quella del secondario. In altre parole quanto ci mette il Phase Vocoder di sincronia a rispondere alla variazione del target. Tale valore è sempre un multiplo della latenza[19].
**Scostamento MAX [ms]** La differenza massima in millisecondi tra il Tag temporale del flusso target e quello del secondario. Indica di quanto sono desincronizzati i flussi.
**Recupero [s]** Quanto ci mette il Phase Vocoder a stabilizzare il flusso secondario dall'ultima variazione di velocità del flusso target.
**Forma [+/-]** Abbiamo visto che sia il flusso target che i secondari sono descritti da una traiettoria di velocità propria. Analizzando i dati ci si è resi conto che non esiste un'effettiva correlazione tra il rispetto della traiettoria e lo scostamento massimo. Si possono ottenere così buoni valori di sincronia pur non facendo seguire la traiettoria target al flusso secondario. Il risultato è comprensibile pensando che il sistema di controllo, retroazionato negativamente, risponda con grandi variazioni di $\alpha$ se $\beta$ ha valore molto contenuto. Il non rispetto della traiettoria di velocità del flusso target non è desiderabile perché si traduce in variazioni frequenti e repentine della velocità dei flussi secondari. Se $|\alpha_T - \alpha_S| = |\Delta_\alpha| < 0.06$ si dice che la traiettoria di velocità del flusso target è stata rispettata e viene indicata in tabella con il simbolo **+**. Nella figura (3.15) viene proposto un caso limite dove $|\alpha_T - \alpha_S| = |\Delta_\alpha| > 0.5$ che fa comprendere come la traiettoria target non vegna rispettata pur contenendo lo scostamento massimo entro valori accettabili: 20 millisecondi per la figura proposta.

[19]Nel nostro caso 10 millisecondi.

**Tabella 3.4:** *Impatto di $\beta$ sul funzionamento Sinusoide*

| $\beta$ **Battute** $10^{-3}$ | **Ritardo Variazione [ms]** | **Scostamento MAX [ms]** | **Forma +/-** |
|---|---|---|---|
| 0.125 | 40 | 12.00 | - |
| 0.250 | 40 | 14.85 | - |
| 0.375 | 40 | 12.40 | - |
| 0.500 | 40 | 5.46 | - |
| 0.625 | 40 | 6.02 | - |
| 0.750 | 40 | 4.06 | - |
| 0.875 | 50 | 4.67 | - |
| 1.000 | 50 | 4.67 | - |
| 1.125 | 50 | 4.35 | - |
| 1.250 | 50 | 4.35 | - |
| 1.625 | 130 | 7.96 | + |
| 5.000 | 170 | 12.63 | - |
| 6.250 | 210 | 16.35 | - |

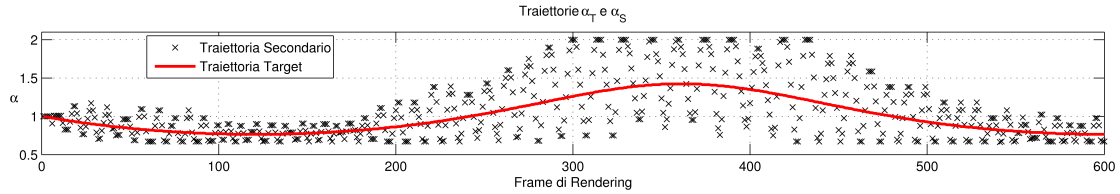


**Figura 3.15:** *Traiettoria target non rispettata*

1.625 millesimi di battuta è il valore che è stato scelto per il default: ha una buona risposta nella traiettoria e la distanza massima è al di sotto dei 18 millisecondi voluti. È però interessante vedere che, seppur a discapito della forma della traiettoria, 1 millesimo di battuta è un valore che minimizza lo scostamento massimo. Si potrebbe quindi dotare lo strumento di un tasto che, se premuto, porti il valore di $\beta$ a un millesimo di battuta. In questo modo l'utente, quando vorrà fare modifiche repentine alla velocità, terrà premuto il tasto e agirà come desidera sul controllo. Alla fine rilascerà il pulsante facendo tornare lo strumento al funzionamento normale.

Il paramentro $\beta$ verrebbe così vissuto come una sorta di valore di inerzia del sistema di cui disporre a proprio piacimento.

#### 3.2.4.3 Distanza zero a regime

Nella sezione 1.1.3 si è visto che se risulta valida la disequazione $(T_t - T_s)/\beta < 1$, non si ottiene nessuna variazione di $R_s$ anche se la distanza tra i flussi non è nulla. Si è quindi proposto un affinamento per poter portare a zero la distanza a regime.

Per fare questo basta incrementare di poco (uno o due unità al massimo) il valore di $P_R$. Nel nostro caso si è scelto di tradurre la distanza tra i flussi in sample[20] e incrementare tale valore di una unità se la distanza fosse dispari o di due unità se la distanza fosse pari. Tale procedimento, che nell'implementazione viene chiamato *hard-sync*, non comporta la comparsa di distorsioni (la perdita di campioni sintetizzati è davvero irrisoria e spalmata nel tempo) e viene avviata solo quando la disequazione sopra riportata è valida e la distanza è costante almeno da 10 passi di rendering. Questo perchè si cerca di ridurre al minimo l'utilizzo dell'*hard-sync* proprio per la perdita di qualità che, seppur minima, viene introdotta. È facile comprendere dalla figura che la

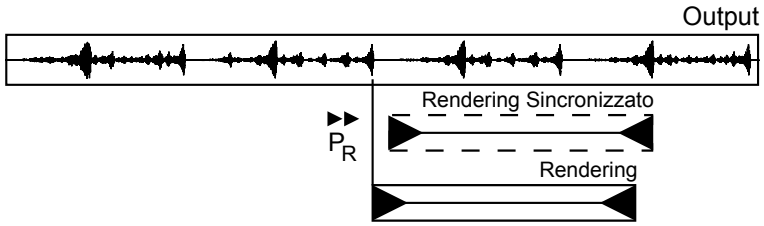


**Figura 3.16:** *Operazione di spostamento forzato del buffer di rendering*

variazione di $P_R$ comporta la variazione di un parametro fondamentale nell'operazione di OLA finale e quindi sposta anche l'*offset* tra il buffer di rendering e il *primo nodo in rendering*. Il Tag che ne risulterà sarà quindi variato del minimo valore possibile per tornare alla sincronia.

Si era presa in considerazione anche la possibilità di spostare il buffer di rendering anche di grandi valori cercando punti adatti allo scopo[21], tale proposta non può essere portata a termine dato che, solitamente, il gap che viene trattato dall'*hard-sync* è molto breve e quindi le possibilità di trovare altri valori di spostamento spesso fallisce.

## 3.3 Il flusso audio finale

In questa sezione si vede come il flusso audio finale viene creato e come viene inviato al device di rendering. Per fare questo verranno introdotti tutti gli strumenti utilizzati non solo per far interagire tra di loro i diversi blocchi di codice, ma anche per sincronizzare il lavoro in modo che venga sempre calcolato solo ciò che si ritiene indispensabile ad ogni passo, mantenendo più basso possibile il peso computazionale.

### 3.3.1 Mixer

I blocchi di codice che sono stati descritti in precedenza devono essere sincronizzati in maniera ordinata in modo da non creare eccessiva confusione implementativa. Poiché le parti di codice

[20] La distanza viene convertita in samples tenendo conto dei soli valori conosciuti sul flusso secondario, nulla serve del flusso target.

[21] Si poteva cercare una grande correlazione tra il punto di partenza e quello di arrivo.

che devono coesistere sono diverse, si è deciso di costruire una funzione unica che si occupi di tutto il lavoro in modo che non vi siano mai chiamate incomplete o non correttamente ordinate. Tale funzione, chiamata mixer[22], si occupa della sincronizzazione dei flussi, del mixing audio e della corretta sequenza di attivazione dei Phase Vocoder. Inoltre attraverso questa funzione vengono dettati i tempi corretti del Phase Vocoder implementato che diventano modifica, sintesi, rendering. Chi decide questi tempi è lo stesso sistema operativo.

#### 3.3.1.1 WASAPI

Per ottenere quanto scritto in precedenza, è stato fatto ampio uso delle procedure fornite da Microsoft raccolte nelle **WASAPI** (Windows Audio Session Application Program Interface). Sono principalmente due le interfacce che permettono un meccanismo di sincronizzazione tra gli algoritmi software e il rendering hardware: **IAudioClient** e **IAudioRenderClient**. La prima rende possibile la configurazione e l'inizializzazione di tutto l'hardware che può acquisire o renderizzare dati audio, nel nostro caso è stata utilizzata in *exclusive-mode* per poter avere accesso diretto al buffer hardware della scheda audio installata nel sistema[23]. IAudioRenderClient si occupa invece del passaggio fisico della memoria tra il programma e la scheda.
È proprio IAudioClient che, grazie alla gestione ad eventi, riesce a scandire i tempi di lavoro e a dirigere le varie sezioni del programma in modo proficuo. Vediamo come:

prima che il programma cominci a renderizzare creando il primo frame e consegnandolo a IAudioRenderClient, l'applicazione deve configurare IAudioClient attraverso il metodo **SetEventHandle**. In questo modo si comunica a IAudioClient quale evento debba essere lanciato nel momento in cui il device é pronto a renderizzare il frame che é stato scritto sull'hardware in precedenza. Qui avviene la sincronia: ogni volta che IAudioClient segnala che é pronto a renderizzare i dati attraverso l'evento specificato, il programma comincia a sintetizzare il frame successivo. Non vi é alcun frame che divida il frame in rendering da quello che si sta renderizzando, in questo modo la latenza con cui il sistema risponde é esattamente uguale alla durata in millisecondi del frame di rendering.

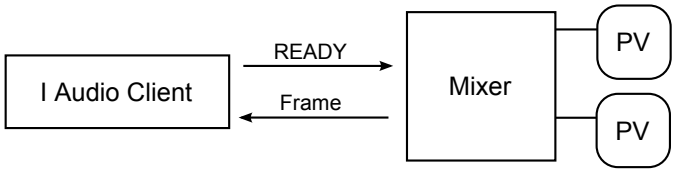


**Figura 3.17:** *Mixer e IAudioClient*

Ecco che entra in gioco la funzione di mixing, che si preoccupa di lanciare il processo per determinare quale $\alpha$ si debba utilizzare per rimanere sincronizzati, di iniziare la sintesi del segnale

[22] B.4 mixer.

[23] Sebbene questo metodo dia numerose limitazioni che danneggerebbero un software di produzione, (come ad esempio il non poter usare altri software musicali contemporaneamente a quello che stiamo sviluppando) esso ci permette di non avere niente di gestito dal sistema operativo e di poter avere dati temporali *puliti* e non contaminati ad esempio dal motore audio del sistema operativo.

con i parametri scelti, di miscelare i file secondari con il file target e di passare tutti i campioni ottenuti a IAudioRenderClient.

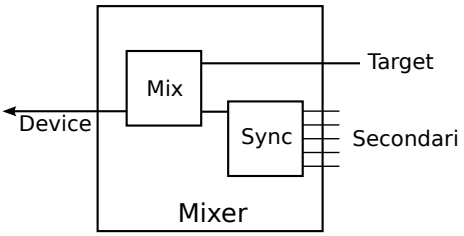


**Figura 3.18:** *Mixer in dettaglio*

Si comprende quindi che IAudioClient richiede ogni 10 millisecondi la creazione di un nuovo frame di rendering a `char*`Mixer() che restituisce proprio un puntatore ad un buffer di memoria. Richiedere il risultato della funzione mixer equivale a chiedere tutto quello che serve per la creazione del nuovo frame aggiornando tutti i parametri di sincronia che sono stati costruiti.

Questo approccio modulare permette di poter modificare gli algoritmi in zone ben delimitate del codice. La chiamata a cascata inoltre mantiene pulito il codice all'interno dei thread utilizzati in modo da mantenerli leggibili e di semplice interpretazione[24].

## 3.4 Threading

Una volta completato il lavoro ci si accorge come sia necessario gestire in modo differente gli eventi che vengono generati durante il run-time. Se questo non venisse fatto, tutta il lavoro impiegato per costruire uno strumento real-time verrebbe reso vano. Si identificano tre tipologie di eventi molto diversi tra loro:

**Eventi utente**.

Sono tutte le interazioni che l'utente ha con l'interfaccia grafica. Di particolare importanza sono quelle che riguardano la variazione di parametri di sintesi del segnale di uscita ($\alpha$, volume master, volume di un flusso secondario ...) che devono rilevare le modifiche apportate in un tempo molto limitato: sono direttamente collegate al valore di latenza che abbiamo definito all'inizio del progetto non poter superare il valore di 10 millesimi di secondo.

**Eventi applicazione**.

Sono quegli eventi che permettono il corretto funzionamento dell'applicazione. Sono creati direttamente dalle librerie del sistema operativo[25] e devono avere precedenza su tutti gli altri, se così non fosse l'applicazione diventerebbe subito instabile.

[24] Si vedano B.1 B.2 B.3.
[25] Li gestisce IAudioClient.


**Figura 3.19:** *Controlli per gli eventi Utente*

**Eventi debug**.

Sono le informazioni che vengono date all'utente o allo sviluppatore per poter comprendere lo stato di funzionamento del programma. Possono qualitativamente essere raggruppate in *Informazioni*, *Avvertenze* ed *Errori*. Sebbene le ultime siano utili solamente allo sviluppatore (che ne trae ggrande vantaggio in fase di debug), gli altri due possono indicare all'utente informazioni che sottolineano eventi generalmente negativi, ma non per questo fatali all'esecuzione del programma come il *clipping* (molto facile da incontrare quando si ha la miscelazione di più flussi audio) o addirittura informazioni che descrivono solo i parametri di funzionamento del sistema (il terzo flusso secondario sta procedendo con un valore di $\alpha$ pari a $1.236$). Gestire tutto questo in un unico thread rende il codice molto complesso da leggere e il programma nè di uso semplice, nè produttivo. La prima sensazione che si ottiene è quella di una interfaccia lenta nel rispondere che azzera tutti gli sforzi fatti per permettere una latenza molto bassa nella rilevazione dei parametri di sintesi. Sono stati creati quindi tre thread: uno per l'interfaccia, uno per il rendering e uno per il debug. Mentre quelli di interfaccia e di rendering creano e gestiscono eventi del sistema operativo, quello di debug poggia su librerie di delegati in modo da poter essere disabilitato più facilmente durante il run-time.

## 3.5 Risultati

Nelle tabelle 3.3 e 3.4 sono riassunti i valori di funzionamento del programma per diversi $\beta$. In questa sezione si cerca di illustrare, in modo particolareggiato, l'andamento di due valori importanti nel funzionamento del programma per il valore di default di $\beta$: le traiettorie di velocità e lo

scostamento temporale dal flusso target. Le traiettorie scelte per il flusso target sono: una rampa, che permette di comprendere in quanto tempo il flusso secondario viene portato alla sincronia, e una sinusoide, che valuta il funzionamento in una variazione continuata nel tempo.

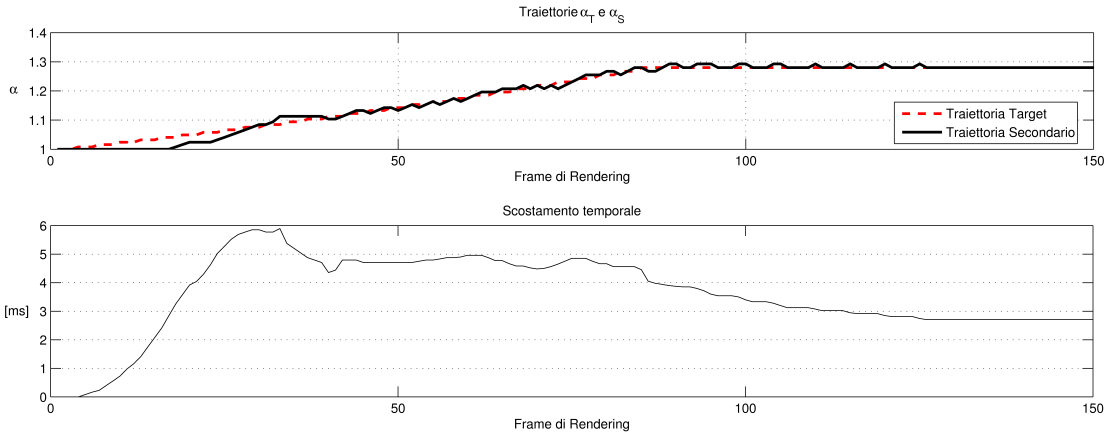


**Figura 3.20:** *Traiettorie di velocità e scostamento Rampa*

In questo primo esempio, si prende in considerazione una velocità target che varia descrivendo una rampa, tale variazione è indicata con linea tratteggiata. La linea continua invece rappresenta la variazione di velocità sul flusso secondario. Tralasciando i valori già visti nella tabella 3.3 (sono riportati a $\beta = 1.625 \times 10^{-3}$ battute) poniamo l'attenzione sulle caratteristiche che ben descrivono il funzionamento del programma. Il flusso secondario parte da un coefficiente di velocità pari a uno proprio come il flusso target che però inizia lentamente ad accelerare. A questo punto il programma comincia a verificare lo scostamento dei Tag sino a quando reagisce modificando la velocità del flusso secondario. Come si vede nel grafico di scostamento, dato che la rampa è positiva e quindi va verso un'accelerazione del brano, il flusso target anticipa (scostamento positivo) il secondario sino a quando la distanza arriva a 6 millisecondi. Dopo il raggiungimento di questo valore, la risposta della sincronizzazione mantiene lo scostamento sotto i 5 millisecondi.

In seguito alla conclusione della modifica di velocità da parte del flusso target, l'algoritmo si avvicina progressivamente al valore finale. Dal grafico delle traiettorie si vede come l'applicazione compia brevi aumenti di velocità che coincidono ad una piccola diminuzione dello scostamento temporale sino a quando si giunge alla soglia di non rilevamento definita dal valore di $\beta$ che in questo caso equivale a poco meno di 3 millisecondi. Se fosse presente l'*hard sync*, dopo 10-15 frame di rendering si vedrebbe diminuire lo scostamento fino al valore zero. Tale diminuzione descriverebbe una linea retta descritta da una pendenza molto bassa: basti pensare che il metodo utilizza spostamenti di uno o due sample per ogni frame di rendering.

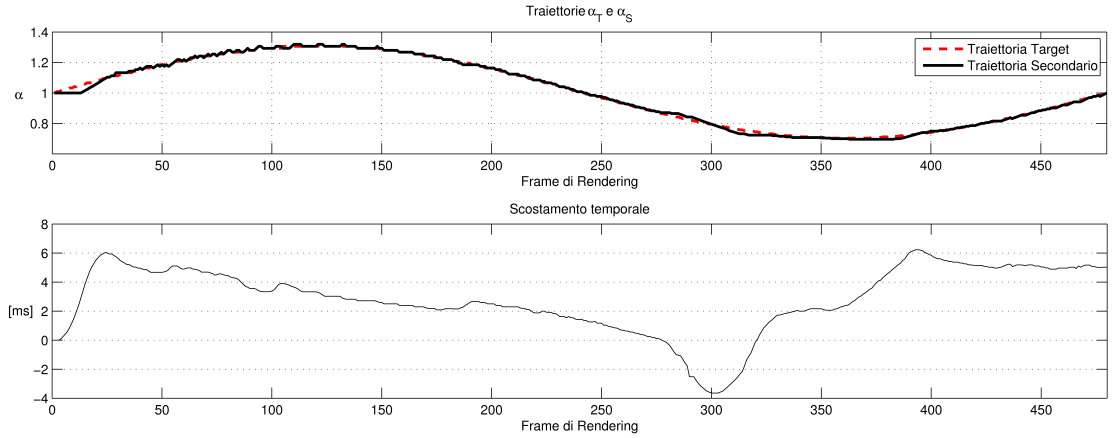


**Figura 3.21:** *Traiettorie di velocità e scostamento Seno*

Anche in questo caso lo scostamento massimo si attesta sui 6 millisecondi. In questo caso non è possibile apprezzare lo scostamento residuo a regime, dato che la variazione di $\alpha$ è continuata nel tempo. Il comportamento visto con il segnale a rampa si intuisce anche nei punti dove il seno ha pendenza costante, nei punti dove invece la pendenza varia si visualizza lo scostamento massimo.

Per traiettorie che non descrivono variazioni repentine della velocità si è quindi abbondantemente sotto il livello massimo di scostamento previsto essere 18 millesimi di secondo.

# Appendici

# Appendice A

# Formato file Wave PCM canonico

Il formato Wave è un sottoinsieme delle specifiche RIFF per il salvataggio di dati multimediali. Un file RIFF inizia con un *header* seguito da una sequenza di *chunk*. Un Wave è spesso un RIFF con un chunk singolo composto da due subchunk:

- **fmt** che specifica il formato.
- **data** che contiene i raw dell'audio.

| Offset Byte | Description | Dimension Byte | Endian |
|---|---|---|---|
| 00 | Chunk ID | 4 | Big |
| 04 | Chunk Size | 4 | Little |
| 08 | Format | 4 | Big |
| 12 | Sub Chunk 1 ID | 4 | Big |
| 16 | Sub Chunk 1 Size | 4 | Little |
| 20 | Audio Format | 2 | Little |
| 22 | Num Channels | 2 | Little |
| 24 | Sample Rate | 4 | Little |
| 28 | Byte Rate | 4 | Little |
| 32 | Block Align | 2 | Little |
| 34 | Bits per Sample | 2 | Little |
| 36 | Sub Chunk 2 ID | 4 | Big |
| 40 | Sub Chunk 2 Size | 4 | Little |
| 44 | DATA | | Little |

**Figura A.1:** *Header RIFF Wave PCM*

- **ChunkID** Contiene le lettere RIFF in codifica ASCII (0x52494646 formato big-endian).
- **ChunkSize** 36 + SubChunk2Size, o più precisamente: 4 + (8 + SubChunk1Size) + (8 + SubChunk2Size). È a tutti gli effetti la dimensione dell'intero file in byte meno gli 8 di *ChunkID* e di *ChunkSize*
- **Format** Contiene le lettere WAVE in codifica ASCII (0x57415645 formato big-endian).
- **Subchunk1ID** Contiene le lettere fmt in codifica ASCII (0x666d7420 formato big-endian).
- **Subchunk1Size** 16 per i file PCM. Questa è la dimensione del subchunk rimanente.
- **AudioFormat** 1 per i file PCM. Quantizzazione lineare.
- **NumChannels** Mono = 1, Stereo = 2, ...
- **SampleRate** Numero di sample al secondo.
- **ByteRate** SampleRate $\times$ NumChannels $\times$ BitsPerSample / 8. Numero di Byte al secondo.
- **BlockAlign** NumChannels $\times$ BitsPerSample / 8. Numero di Byte per ogni singolo campione audio.
- **BitsPerSample** Numero di Bit per ogni singolo campione audio mono.
- **Subchunk2ID** Contiene le lettere data in codifica ASCII (0x64617461 formato big-endian).
- **Subchunk2Size** NumSamples $\times$ NumChannels $\times$ BitsPerSample / 8. Dimensione in Byte della regione che contiene i campioni audio.
- **Data** I campioni audio.

# Appendice B

# Pseudocodice

In questa sezione del documento si illustrano, per mezzo di pseudocodice, le principali funzioni del programma al fine di rendere completa la descrizione di funzionamento. Seppur notevolmente semplificate, non sono stati inseriti tutti i controlli avanzati, in queste funzioni è possibile rintracciare i metodi di sincronia delle diverse sezioni e l'idea che sta alla base dell'implementazione.

## B.1 Start Rendering

```
01 - creo il thread di rendering;
02 - alloco il buffer di rendering;
03 - invio il puntatore del buffer di rendering
     al device hardware;
04 - precarico il primo buffer di rendering richiamando
     la funzione mixer;
05 - copio la memoria sul buffer di rendering inviandolo
     al device hardware;
06 - rilascio il buffer di rendering;
07 - lancio il thread di rendering;
```

Prima di far partire il rendering dei dati che si sono ottenuti dal programma, si riempe preventivamente il buffer dell'hardware con il primo frame di rendering. Questo è importante al fine di evitare la formazione di *click* in avvio: il buffer hardware non si svuota in maniera automatica e, se fatto partire accidentalmente su di un buffer appena dichiarato, ci si troverebbe a renderizzare rumore. Si sottolinea inoltre che l'invio e il rilascio del puntatore del buffer di rendering viene ottenuto attraverso le WASAPI. Quindi per farlo si deve sia aver inizializzato opportunamente il device sia sapere che tipi di dato il device è pronto a ricevere. La funzione ritorna un valore booleano che attesta l'effettivo lancio del thread.

## B.2 Stop Rendering

```
01 - lancio evento di STOP;
02 - dealloco il buffer di rendering;
03 - fermo il thread di rendering;
04 - lancio tutti i post processi;
```

Come si vede il trhead di rendering è sempre in attesa di eventi che dicano se fermarsi o che lo informino se il device ha accettato i dati precedentemente inviati. Per fermare il thread basta quindi lanciare da qualsiasi punto del codice l'evento di stop. Sebbene la funzione possa sembrare scarna, nell'implementazione reale essa è invece resa complessa ed articolata proprio nell'espletamento della linea ***2***. Se si vogliono salvare su disco i dati del funzionamento dell'applicazione è impensabile farlo mentre si sta renderizzando: un minimo ritardo dovuto a scrittura andrebbe ad inficiare la qualità del suono in uscita. Per questo motivo all'interno del programma esistono dei buffer che trattengono i dati di debug sino alla fine del rendering e la scrittura di tali informazioni vengono posticipate dopo lo stop del thread e, di fatto, proprio in questa funzione. Si sottolinea che i dati di debug trattati in questo punto non hanno nulla a che fare con le informazioni di debug che l'applicazione invia tramite interfaccia grafica all'utente, ma si sta parlando di dati come tutti i dati temporali o in frequenza contenuti dai nodi di memoria, la registrazione audio di cosa è stato renderizzato e altri dati che in real-time non avrebbe senso controllare.

## B.3 Do Render Step

```
01 - configuro le opzioni del thread di rendering;
02 - controllo se siamo in presenza di un flusso target o
     di una modifica manuale;
03 - entro in loop infinito sino a quando:
      - il buffer di rendering ha puntatore NULL;
      - ricevo uno stop;
04 - attendo gli eventi;
05 - se arriva uno STOP esco dal loop;
06 - se arriva un READY
07 -   applico le modifiche ad alpha;
08 -   collego il buffer in uscita dal mizer al device
       hardware;
09 -   se richiesto salvo il rendering in uscita in
       memoria;
10 -   se il buffer di rendering ha puntatore NULL
       fermo il rendering;
11 -   rilascio il buffer e rientro in loop;
```

Appena si crea il thread di rendering questa funzione viene fatta eseguire in automatico. La funzione provvede a configurare il proprio thread e controlla subito che tipo di comportamento dovrà adottare: il suo comportamento infatti varia se è l'utente che modifica il valore di $\alpha$ o se questa variazione è salvata direttamente nel flusso target. È questa la parte di codice che attende lo stop inviato da *stop rendering* o accetta il NULL dato in uscita dalla funzione *mixer* nel momento in cui uno dei flussi è giunto al termine. Un'altro importante ruolo di do render step è quello di sincronizzare il funzionamento dell'applicazione: questa funzione attende un evento *ready* da parte del sistema operativo che attesti l'effettiva ricezione del buffer di rendering da parte del device hardware. Solo dopo questo evento il codice calcolerà la sintesi successiva e questo risulta fondamentale per rispettare il valore di latenza di 10 millisecondi che è stato imposto al progetto. Come si comprende dal codice i NULL vengono utilizzati come marcatori per la fine del file e vengono generati al livello di gestione dela memoria dell'applicazione: *stretchNode* e *olaNode*. Queste funzioni invece li propagano solamente e, in questa funzione particolarmente, si intendono come equivalenti ad un evento di STOP.

## B.4 Mixer

```
01 - aggiorno la distanza tra il flusso target e il
     secondario;
02 - applico il calcolo per il nuovo alpha;
03 - limito il valore di alpha tra 0.5 e 1.5;
04 - sintetizzo un frame di rendering del flusso
     secondario con il valore di alpha corrente
     utilizzando Up Render Window;
05 - se il puntatore restituito ha valore NULL
     vuol dire che il flusso secondario ha
     raggiunto il termine del file. Ritorno
     NULL in modo che Do Render Step fermi il
     rendering;
06 - se il controllo di alpha risulta su manuale
     sintetizzo un frame di rendering del flusso
     target, altrimenti carico un frame di rendering
     del flusso target dal nuovo formato audio;
07 - se il puntatore restituito ha valore NULL
     vuol dire che il flusso target ha raggiunto
     il termine del file. Ritorno NULL in modo che
     Do Render Step fermi il rendering;
08 - miscelo i due segnali ottenuti;
09 - converto in formato hardare;
10 - ritorno il buffer ottenuto dal miscelamento;
```

Questa funzione gestisce completamente la creazione dei frame da renderizzare per ottenere la sincronia. Come si vede monitora la distanza tra i flussi, calcola $\alpha$ in modo da poter mantenere la sincronia e, attraverso la funzione *Up Render Window* ottiene i dati direttamente dai Phase Vocoder presenti nel progetto. Come si vede anche la funzione *mixer* propaga i NULL in arrivo dalla gestione di memoria in modo da poter gestire lo stop del rendering e si comporta in maniera differente a seconda che si sia in presenza di modifica della velocità manuale o si stia utilizzando un flusso target da disco. In questo modo infatti si può spegnere/accendere il Phase Vocoder deputato alla creazione del flusso target. In realtà l'istanza del Phase Vocoder è unica, quindi è proprio la funzione *mixer* a decidere su quali dati vada o no richiamata la funione *Up Render Window* che richiama in cascata tutti gli strumenti implementati nel Phase Vocoder.

## B.5 Up Render Window

```
01 - aggiorno il valore di alpha;
02 - se risulta variato annullo tutti i buffer di uscita
     tranne quello del primo nodo in rendering;
03 - faccio uscire i nodi in uscita;
04 - faccio entrare i nodi in entrata;
05 - se il puntatore al primo nodo risulta NULL ritorno
     NULL ed esco dal procedimento;
06 - annoto la prima posizione di sintesi dato l'alpha
     corrente;
07 - applico lo stretch node ai nodi entrati;
08 - applico olaNode ai nodi interni al rendering;
09 - applico al buffer di rendering la maschera di
     amplificazione;
10 - aggiorno la posizione di analisi;
11 - ritorno il buffer;
```

Questa è la funzione principale per l'utilizzo del Phase Vocoder. È importante far notare che il valore di $\alpha$ viene aggiornato solamente in un punto ben preciso: in questo modo si impedisce di variare un valore importante come $R_s$ mentre si sta sintetizzando un nodo di memoria. Se accadesse andremmo a danneggiare gli algoritmi di stretching che andrebbero a calcolare propagazioni di fase errate. Inoltre la funzione si interessa di scorrere i nodi di memoria per ottenere quelli che effettivamente servono per la sintesi del rendering corrente e di appuntarsi sia la posizione di analisi che quella di sintesi ad ogni passo in modo da poter desumere il valore preciso dei Tag temporali per la sincronia. La funzione utilizza *Stretch Node* e *Ola Node* senza controllare se questi effettivamente servono perchè sono le altre due funzioni ad eseguire i calcoli solo se necessari. Infine, ruolo non di poca importanza, applica la maschera di amplificazione che si è definita nella sezione 3.2.3.2.

## B.6 Stretch Node e Set TS Mode

**Stretch Node**

```
01 - controllo le impostazioni di stretching;
02 - se il nodo ricevuto ha puntatore di dati di output
     diverso da NULL esco;
03 - applico l'algoritmo di time stretching al nodo
     creando un buffer di dati di output;
```

Questo codice riceve in input un nodo di memoria e lo modifica secondo gli algoritmi visti nel secondo capitolo. Viene utilizzato l'artifizio di calcolare la sintesi solo se i puntatori sono nulli: in questo modo non vengono mai calcolati due volte gli stessi stretch. Quando $\alpha$ viene variato il è Set TS Mode che si preoccupa di rendere NULL i nodi che devono essere ricalcolati. Le variabili globali del gestore di memoria intervengono nella scelta dell'algoritmo utilizzato per ottenere i buffer di output.

**Set TS Mode**

```
01 - configuro la finestratura di analisi;
02 - configuro il valore di hop di analisi;
03 - configuro il metodo di time stretching;
04 - configuro la finestratura di sintesi;
05 - configuro il valore di hop di sintesi;
06 - configuro gli switch di time stretching;
```

La funzione configura i metodi di utilizzo di Stretch Node. Non ritorna nulla, ma va a modificare variabili globali del gestore di memoria che servono per la corretta parametrizzazione degli algoritmi da usare. Questo permette di controllare il momento in cui viene variato il valore di $\alpha$ per evitare frame di sintesi ibridi che portano a deriva di fase sia verticale che orizzontale. È l'unica funzione adibita alle modifiche in modo che le variabili globali non possano essere corrotte da codice esterno. Nel caso in cui il valore di hop di sintesi venga variato[1], tale funzione si preoccupa di assegnare il valore NULL a tutti i buffer dei nodi che contengono dati obsoleti.

[1] È il metodo migliore per variare $\alpha$ dato che, da un punto di vista implementativo, bastano $R_a$ ed $R_s$ come parametri.

## B.7 Ola Node

```
01 - se il nodo ricevuto ha puntatore NULL esco;
02 - se il nodo possiede buffer di dati in output vuoti
     il nodo in questione non si trova in rendering;
03 - calcolo l'offset tra nodo e buffer di rendering;
04 - sommo i dati nel buffer di rendering;
05 - aggiorno i dati riguardo la maschera di amplificazione;
```

Questa funzione permette di inserire, all'interno del buffer di rendering, il buffer rappresentante il frame di sintesi per il segnale di output. Nel caso in cui il puntatore che interessa sia null vuol dire che il nodo preso in considerazione non è stato reputato interno al buffer di rendering. Questo permette di lanciare l'OLA su tutti gli nOla puntatori che servono per creare il file di uscita senza preoccuparsi di quali effettivamente servono e quali sono ancora inutili evitando di calcolare le condizioni di intersezione tra frame di output e buffer di rendering più volte durante il run-time. Il frame di sintesi viene posizionato in maniera corretta nei confronti del buffer di rendering corrente attraverso il calcolo dell'offset. I valori utili vengono accumulati in modo che i vari frame si sommino come voluto dall'overlap add. Per finire la funzione crea, attraverso procedimento di overlap add, anche la maschera di amplificazione di cui si è parlato nel terzo capitolo.

# Appendice C

# Interfaccia utente

Riflettendo in maniera più ampia sul lavoro esposto fino a questo punto, si potrebbe pensare che il raggiungimento delle specifiche tecniche desiderate sia totalmente dovuto alla bontà del modello matematico che guida l'implementazione finale dell'applicazione. Tuttavia questo è un errore che potrebbe vanificare tutti gli sforzi fatti in fase di progettazione.

Tutto il documento si basa su due aspetti fondamentali: la **qualità** del segnale di uscita e il rispetto di una **latenza** massima di dieci millisecondi: ovvero l'intervallo temporale tra l'intervento dell'utente e l'effettiva modifica del segnale di uscita. Questi due parametri dipenderebbero solo dal modello matematico se l'applicazione non fosse né interattiva, né in Real-time: in questo caso l'utente potrebbe solamente configurare i parametri iniziali non avendo alcuna possibilità di intervento durante il run-time.

Nel nostro caso invece si propone un'applicazione al servizio dell'utente che può attuare modifiche mentre il programma è in esecuzione e questo significa pensare ad una *Graphic User Interface* (GUI) che preveda una buona sinergia tra il programma e gli strumenti input e di output utilizzati dall'utente (tastiera, mouse, video ...). Se l'interazione utente-software non fosse soddisfacente e desse luogo a problemi di reattività o, nei casi peggiori, generasse errori, tutto il lavoro sulla latenza e la qualità potrebbe andare perso.

Al fine di comprendere questo aspetto, pensiamo ad una semplice applicazione di calcolo che non richieda l'interazione da parte dell'utente durante l'esecuzione. Una volta descritto il problema, il software si interessa di risolverlo e fino a quando non si è ottenuto il risultato, l'interfaccia non ha alcuna utilità: non avrebbe nessun senso modificare i dati del problema mentre l'elaboratore cerca di risolverlo. In questo caso da parte dell'utente è accettabile che la GUI rimanga "bloccata" fino a quando il lavoro non è portato a termine. L'unica opzione che potrebbe risultare interessante risulterebbe essere quella di interrompere il processo senza attendere forzatamente l'arresto programmato.

È banale dimostrare che la nostra applicazione non può accettare uno scenario del genere: come potrebbe l'utente sfruttare una latenza di 10 millisecondi se l'interfaccia venisse aggiornata ogni mezzo secondo? Ecco che si riesce a comprendere come la latenza percepita dall'utente non sia solo una questione di modello matematico, ma anche di interfaccia. Il discorso appena fatto potrebbe far erroneamente pensare che il problema sia un'interfaccia troppo "lenta" e che quindi

una possibile soluzione sia quella di implementare una GUI più reattiva possibile. Questo porta a due conseguenze interessanti:

1. Se la latenza di interfaccia permettesse la variazione di $\alpha$ ogni millisecondo, l'algoritmo *stretchNode* vedrebbe variato il paramentro $R_s$ fino a dieci volte durante la sua esecuzione, danneggiando quasi sicuramente la qualità del segnale di uscita[1].
2. Un'eccessiva velocità di aggiornamento dei dati a schermo, oltre che ad aumentare il peso computazionale dell'applicazione, causa spesso confusione nell'utente.

Se il primo punto è in effetti analogo a quanto detto per la latenza, il secondo richiede di ricorrere al concetto di **usabilità**. In questa fase infatti è necessario più che mai realizzare l'interfaccia come un'attenta traduzione che porta dalla logica di programma alla logica di utilizzo. I punti fondamentali da rispettare per ottenere un prodotto usabile sono otto[18]:

1. Realizzare un dialogo semplice e naturale.
2. Semplificare la struttura dei compiti.
3. Favorire il riconoscimento piuttosto che il ricordo.
4. Rendere visibile lo stato del sistema.
5. Prevenire e limitare gli errori di interazione e facilitarne il recupero.
6. Essere coerenti.
7. Facilitare la flessibilità di utilizzo e l'efficienza dell'utente.
8. Fornire l'help.

Al fine di favorire il riconoscimento piuttosto che il ricordo, l'interfaccia è stata fisicamente divisa in varie sezioni che verranno esposte separatamente.

## C.1 I Controlli Utente

L'utente, durante il run-time, può variare la velocità del flusso target, il volume e l'algoritmo in uso per il time stretching. Da un punto di vista implementativo, questa finestra di dialogo viene eseguita su un thread diverso da quello di rendering e modifica le variabili globali di $\alpha$, amplificazione e *TSMode*. Come si nota, la rappresentazione di TSMode è molto semplificata rispetto a quella descritta nel terzo capitolo in quanto sono visibili solo i valori principali. Il fatto che non tutti i parametri siano trattati allo stesso modo e non venga creata una mera lista ordinata permette di far percepire all'utente la diversa importanza dei parametri: passare da un *Loose* ad

[1] Qui si evince che esistono due tempi diversi di reazione e il valore di $\alpha$ possa essere variato solo quando la logica di programmazione lo permette e non quando viene rilevata una modifica nell'interfaccia.


**Figura C.1:** *Interfaccia di modifica dei parametri*

uno *Scaled* è una scelta dalle conseguenze più marcate rispetto a quella di attivare o meno il *Silent Phase Reset*. I parametri che non sono presenti nella figura C.1 possono essere visualizzati premendo il tasto "Avanzate".

Potrebbe venir da chiedersi come mai una finestra di dialogo, che si occupa solamente della rappresentazione e modifica della variabili globali, abbia bisogno di un thread dedicato. Quando viene utilizzata la slitta per la variazione di $\alpha$, l'elaboratore deve preoccuparsi anche di ridisegnare la parte grafica, se rendering e interfaccia venissero trattate sullo stesso thread questo porterebbe a due possibili scenari: una latenza superiore ai 10 millisecondi o alla fastidiosa sensazione che il controllo (la slitta, la manopola . . . ) risponda a scatti alle sollecitazioni dell'utente. L'indipendenza dei due eventi fa in modo che l'interfaccia risponda in maniera naturale e che gli algoritmi di time stretching controllino il valore di $\alpha$ solo quando effettivamente serve e non ogni qual volta che il controlo segnala una variazione di un parametro. Da un punto di vista più tecnico gli eventi di interfaccia non incidono sul thread di rendering e vengono trattati in maniera del tutto diversa dagli eventi WASAPI.

Un altro aspetto interessante è quello che lega il parametro al suo controllo: questo procedimento permette di comprendere a fondo il lavoro di traduzione che l'interfaccia fa tra lo-gica di applicazione e logica di presentazione. Le librerie del *Framework 3.5* della Microsoft contengono strumenti avanzati per la modifica di immagini durante il run-time:

```
Image^ ctrlAmpl = Image::FromFile("pathAmpl");
Drawing2D::Matrix^ transfMatrixAmpl = gcnew Drawing2D::Matrix;
...
transfMatrixAmpl->RotateAt(rotAngle, rotCentre);
ctrlAmpl->Transform = transfMatrixAmpl;
pboxAmpl->Invalidate();
```

L'esempio di codice semplificato riportato serve a comprendere come il valore di amplificazione passi da un semplice fattore moltiplicativo ad un angolo di rotazione su cui l'utente può intervenire[2]. Come è prevedibile $\alpha$ diventa invece una distanza di traslazione. Sono state quindi costruite funzioni *delegate*[3] che trasformano eventi dell'interfaccia grafica in modifiche dei parametri dei thread di sintesi del suono. Ovviamente a volte può avvenire anche il contrario: supponiamo per esempio che l'istanza di una sincronia risulti impossibile nel contesto corrente (l'$\alpha$ di un flusso secondario dovrebbe essere portato ad un valore troppo elevato per raggiungere il target), questo evento va indicato all'utente attraverso una modifica dell'interfaccia da parte del thread di rendering. Questa operazione viene portata a termine da una funzione *delegate* lanciata da uno dei due Phase Vocoder che può così intervenire su un thread estraneo (in questo caso sia quello della GUI che quello delle informazioni di *debug*).

## C.2 Lo Stato

La regola numero 4 dice: *"Rendere visibile lo stato del sistema"*. Durante l'utilizzo dell'applicazione ci si rende subito conto di quanto sia importante questa regola: cercare di intuire lo stato di sicncronia attraverso l'uscita sonora dà origine ad una sensazione di *insicurezza* in quanto non si dispone di un chiaro feedback di conferma. Poichè l'utente non deve essere tentato di reputare modificabili dei valori che sono in realtà una descrizione, si è ritenuto opportuno creare una finestra di dialogo apposita separata da quella dei controlli utente. Uno strumento semplice, che merita una descrizione aggiuntiva, è il **visualizzatore di sincronia** riportato in figuura C.2: esso consente l'immediata valutazione dello stato di *sincronia musicale* di ogni singola traccia.

La grafica utilizzata, come si vede in figura C.2, segnala, attraverso il posizionamento della barretta rossa, di quanto sia in ritardo o in anticipo il flusso secondario rispetto al flusso target rappresentato dalla barretta verde. Una volta dotati tutti i flussi secondari di questo indicatore, si ha immediatamente la percezione dello stato di sincronia musicale globale. Nella stessa finestra possono essere visualizzati i valori istantanei di BPM per tutti i flussi[4] e il numero di nodi di

[2]Nel codice mostrato l'immagine ruota solamente di *rotAngle* radianti. Questo passo unitario di rotazione deve essere collegato all'evento che deve controllarne l'iterazione: nel nostro caso si deve fare click sul controllo con il tasto sinistro del mouse e trascinare. Per ogni evento di spostamento del mouse è stata associata una rotazione di 0.03 radianti. Anche la rotazione unitaria è importante alla fine dell'usabilità: ruotare troppo vuol dire dare poca definizione allo spostamento del mouse, ruotare troppo poco vuol dire non poter attuare modifiche repentine.

[3]Queste funzioni sono delicate perché devono mettere in collegamento diversi thread che non conoscono lo stato operativo dei colleghi.

[4]Sebbene sia graficamente interessante si può optare per visualizzare solamente il valore di BPM del flusso target. Mostrarli tutti potrebbe creare confusione nell'utente.

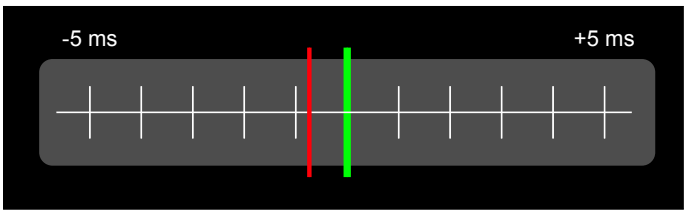


**Figura C.2:** *Il visualizzatore di Sincronia*

memoria utilizzati da ogni singolo flusso. L'utente in questo modo sa quanta memoria RAM ha ancora a disposizione e a che punto è la sincronia musicale: l'esperienza di utilizzo risulta migliore in quanto si hanno valori oggettivi su cui basarsi senza far ricorso a stime sull'output.

## C.3 Strumenti avanzati e possibili espansioni

Occorre offrire strumenti altrettanto efficaci anche per gli utenti più esperti. A questo scopo si è ritenuto necessario rendere accessibili le funzioni di debug opportunatamente modificate in visualizzazioni avanzate dei segnali prodotti. Risulta interessante notare come, nel caso di strumenti più tecnici, la logica di presentazione sia più vicina a quella di programma: questo perchè si può fare affidamento sulla conoscenza di base di chi utilizzerà le funzioni avanzate.

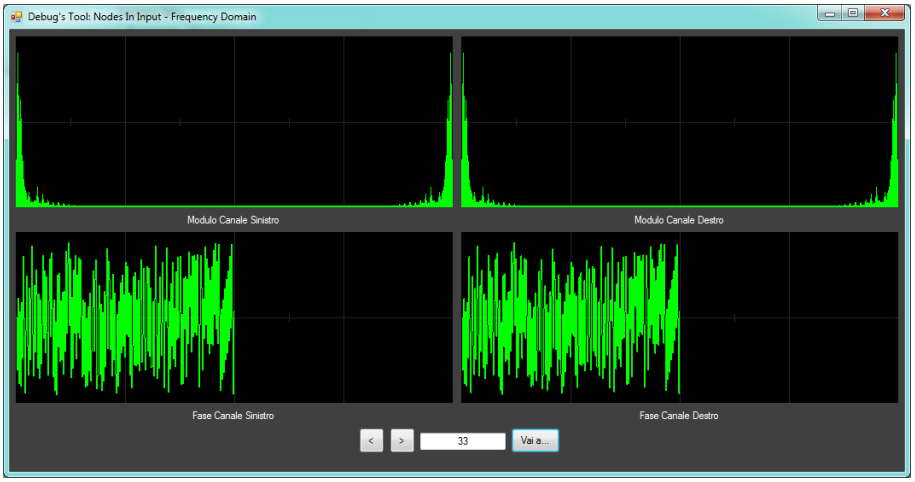


**Figura C.3:** *Nodo in frequenza*

In figura C.3 viene riportata la visualizzazione di un nodo di ingresso o di uscita nel dominio della frequenza. Questo tipo di dati è utile per utenti che vogliono ad esempio controllare lo spettro del segnale di uscita, ma pemette anche di aggiungere ulteriori strumenti come filtri o effetti che potrebbero arricchire le funzionalità dei Phase Vocoder presenti. Si potrebbero esaltare alcune frequenze o attenuarne altre prima o dopo il time stretching.

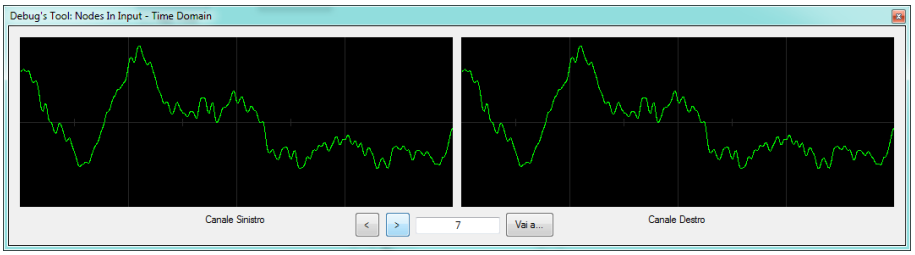


**Figura C.4:** *Nodo nel tempo*

Allo stesso modo sono possibili visualizzazioni nel tempo (figura C.4) dove gli effetti da introdurre potrebbero riguardare la generazione di riverberi o l'applicazione di amplificazioni variabili. Se si accorpano le possibilità con una certa coerenza nel rispetto della regola 6, l'utente saprà sempre ritrovare l'effettistica che gli serve perché gli interventi nel tempo e quelli in frequenza sono divisi non solo a livello concettuale, ma anche a livello della struttura dell'applicazione.

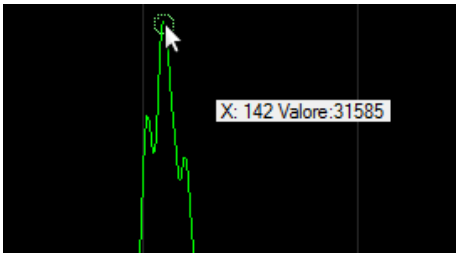


**Figura C.5:** *Zoom nel grafico*

Sebbene i grafici siano uno strumento potente per capire come si sta comportando il programma, essi risultano frustranti nell'utilizzo se manca un grado di libertà importante come lo *zoom*. È importante sottolineare infatti che per per alcuni tipi di segnali, le funzioni di *autoscale* di cui sono stati dotati i grafici del programma rendono inutilizzabili i dati a schermo. Sipensi all'utilizzo di questo strumento quando si è in presenza di un segnale descritto da spettri a linee o affetto da problemi come *click* che si vogliono eliminare.

# Bibliografia


[1] Autori Vari. *Universo*. Istituto Geografico DeAgostini, vol.8 Lemma, Novara, 1977.

[2] Ferrante; Lepschy; Viaro. *Introduzione ai controlli automatici*. Libreria UTET, prima edizione, pp. 348-353, April 2000.

[3] Dolson. *The Phase Vocoder: a tutorial*. Computer Music Journal, vol. 10, pp. 14-27, 1986.

[4] Portnoff. *Implementation of the Digital Phase Vocoder using the Fast Fourier Transform*. IEEE Trasaction on Acoustics, speech and signal processing pp. 243-248, June 1976.

[5] Golden; Flanagan. *Phase Vocoder*. Bell System Technical Journal, pp. 1493-1509, November 1966.

[6] Portnoff. *Time-scale modifications of speech based on short time Fourier analysis*. IEEE Trasaction on Acoustics, speech and signal processing, vol. ASSP-29(3), pp. 374-390, June 1981.

[7] Arfib; Keiler; Zölzer. *DAFX - Digital Audio Effects*. John Wiley & sons, LTD cap. 8, pp. 237-276, 2002.

[8] Arfib; Bernadini; Götzen. *Traditional Implementations of a Phase Vocoder. The tricks of the trade*. DAFX-00 Confrence on Digital Audio Effects, pp. 37-43, Verona, December 2000.

[9] Griffen; Lim. *Signal estimation from modified short-time Fourier Transform*. IEEE Transaction on Acoustics, Speech and Signal Processing, vol. ASSP-32(2), pp. 236-243, April 1984.

[10] Puckette. *Phase-locked Vocoder*. IEEE ASSP Workshop on Applications of Signal Processing to Audio and Acoustics, 1995.

[11] Dolson; Laroche. *Improved Phase Vocoder time-scale modification of Audio*. IEEE Transaction on Speech and Audio Processing, vol. 7, N. 3, pp. 328-330, May 1999.

[12] McAulay; Quatieri. *Audio signal processing based on sinusoidal analisys/synthesis*. Applications of Digital Signal Processing to Audio and Acoustics, pp. 343-416, 1998.

[13] Ferreira. *An odd-DFT based approach to time-scale expansion of audio signals*. IEEE Transaction on speech and audio processing, pp. 441-453, 1999.

[14] Avanzini. *Algorithms for Sound and Music Computing*. http://smc.dei.unipd.it/, cap. 5 Auditory Processing pp. 2-11, 2010.

[15] Borchers; Karrer; Lee. *PhaVoRIT: a Phase Vocoder for Real-Time interactive Time-Stretching*. 2006.

[16] Bonada. *Automatic technique in frequency domain for near-loseless time-scale modification of audio*. Proceedings of International Computer Music Conference, Berlin, 2000.

[17] Laroche; Moulines. *Non-parametric techniques for pitch-scale and time-scale modification of speech*. Speech Communication vol. 16, pp. 175-205, 1995.

[18] Nielsen. *Usability Engineering*. Academic Press, Boston, 1993.