ALMA MATER STUDIORUM · UNIVERSITÀ DI BOLOGNA

SCUOLA DI SCIENZE
Corso di Laurea in Informatica

WaveTrack, implementazione ottimizzata di un algoritmo di pitch tracking basato su wavelet

Relatore:
Chiar.mo Prof.
Renzo Davoli

Presentata da:
Antonio Cardace

Sessione II
Anno Accademico 2013/2014

No matter what people tell you,
words and ideas can change the world.

Robin Williams

Indice

Introduzione **3**

1 Metodi di risoluzione **5**

1.1 Metodi time-domain 5

1.1.1 Zero-crossing 5

1.1.2 Autocorrelazione 6

1.2 Metodi frequency-domain 7

1.2.1 HPS 8

1.2.2 Altri 8

2 Wavelet **9**

2.1 Trasformate 9

2.2 La trasformata Wavelet 13

2.3 Fast Lifting Wavelet Transform 15

3 Algoritmo di pitch-tracking **19**

3.1 Schema di lavoro 19

3.2 Campionamento e formati 20

3.3 Algoritmo 21

3.3.1 FLWT phase 22

3.3.2 Peak detection phase 25

3.3.3 Averaging phase 28

3.3.4 Period phase 28

4 WaveTrack **31**

4.1 Include e define 31
4.2 Funzioni setter 34
4.3 La funzione compute_pitch() 37
4.4 Core 39
4.4.1 Global max/min phase 39
4.4.2 Silence filter phase 40
4.4.3 FLWT phase 40
4.4.4 Peak detection phase 41
4.4.5 Averaging phase 46
4.4.6 Period phase 47
4.4.7 One-shot period method 48
4.4.8 Ring buffer 50

5 Test e risultati **53**

Sviluppi futuri **61**

Conclusioni **63**

Bibliografia **65**

Introduzione

Il pitch, o frequenza fondamentale, è probabilmente l' informazione più significativa presente in un suono, poiché ad esempio nel mondo della musica, in congiunzione con il timing, permette di saper riconoscere la differenza tra una canzone di Mozart o degli AC/DC o ancora, nell' analisi di una voce umana fornisce le basi per i metodi di speech analysis, per poi arrivare a strumenti di analisi vocali sempre più avanzati, come Google Now o Siri.

Per le ragioni appena elencate, l' importanza di uno strumento real-time che permetta, in modo accurato e veloce, di rilevare il pitch in un segnale sonoro diviene fondamentale qualora si voglia sottoporre la musica o la voce umana alle potenzialità di elaborazione di un calcolatore, con l' obiettivo di estrapolarne una qualsiasi informazione significativa.

Da qui nasce l' interesse per il funzionamento e lo sviluppo di un algoritmo di pitch tracking, in seguito verranno presentate le diverse soluzioni adottabili per risolvere un problema appartenente a tale classe.

Capitolo 1

Metodi di risoluzione

Nello sviluppo di algoritmi di pitch detection si presentano diversi possibili approcci, i quali si dividono in 2 categorie:

- **Time-Domain** : approcci nel dominio del tempo.

- **Frequency-Domain** : approcci nel dominio della frequenza.

1.1 Metodi time-domain

1.1.1 Zero-crossing

Uno degli approcci più semplici è il metodo zero-crossing, nel quale, come si può capire intuitivamente dal nome, si contano le volte che il segnale in entrata si interseca con l'asse delle ascisse, il conteggio finale risulterà come una stima della frequenza fondamentale. Sebbene il metodo dello zero-crossing sia computazionalmente leggero e veloce presenta alcuni svantaggi, infatti essendo un metodo che opera direttamente sul segnale in entrata senza applicare nessuna trasformazione o analisi preliminare, risulta essere molto sensibile a disturbi.

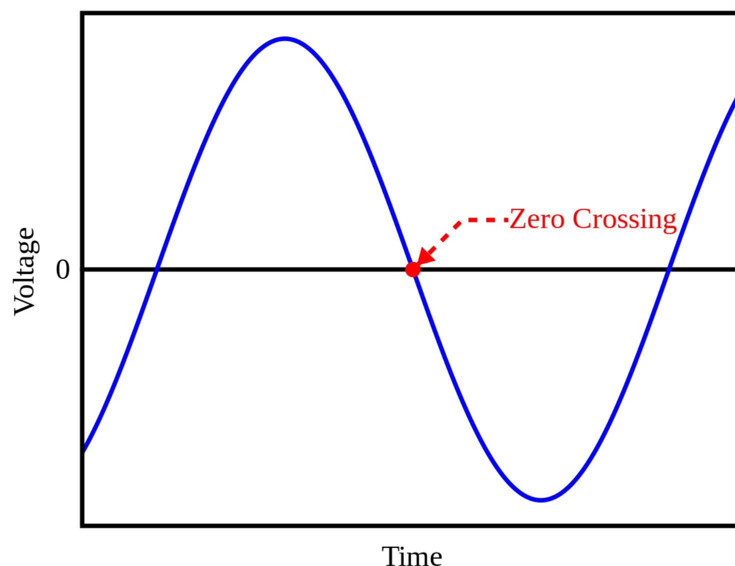


Figura 1.1: esempio zero-crossing

Un algoritmo di pitch tracking real-time riceve in input piccole porzioni di segnale, le conserva in buffer di dimensione arbitraria,[1] e le processa con lo scopo di trovarne la frequenza fondamentale, da qui emerge un altro grande difetto dello zero-crossing, infatti nella scelta di questo approccio risulta fondamentale la dimensione del buffer per l'analisi dei dati, cosicché un buffer troppo piccolo porterebbe ad una perdita di informazione[2] ed un buffer troppo grande inciderebbe sulla velocità dell'applicativo e sulla risoluzione temporale dell'algoritmo.

1.1.2 Autocorrelazione

Alcuni tra gli algoritmi più usati nello sviluppo di metodi per calcolare una stima del pitch in un segnale fanno uso dell'autocorrelazione.

[1]solitamente di 1024/2048MB

[2]le frequenze basse sarebbero più difficili da riconoscere, a causa della loro meno frequente oscillazione

L'autocorrelazione è uno strumento matematico che permette di confrontare un segnale con una versione modificata del segnale stesso[3], mediante questo approccio è possibile trovare periodicità in funzione.

Grazie alle sue peculiarità l'autocorrelazione viene usato in diversi algoritmi di pitch detection, ad esempio:

- Average Squared Mean Difference Function
- YIN algorithm
- MPM algorithm

ciò nonostante presenta alcuni difetti.

In questo metodo la scelta della dimensione della finestra[4] di analisi è di fondamentale importanza, infatti per un riconoscimento corretto del pitch la finestra ideale dovrebbe contenere da 2 a 3 periodi completi della frequenza fondamentale[2], questa caratteristica del segnale non può essere garantita in alcun modo, e dipende unicamente dall'ambiente di impiego dell'algoritmo, inoltre questo approccio funziona meglio su basso-medie frequenze, il chè non lo rende adatto per un algoritmo di pitch tracking general purpose.

1.2 Metodi frequency-domain

I metodi frequency-domain si distinguono dai metodi precedentemente descritti poiché prima di analizzare i dati in input applicano una trasformata matematica[5] per ottenere lo spettrogramma del segnale su cui poi andranno ad effettuare specifiche computazioni.

[3] spesso viene applicata un operazione di shift

[4] porzione di segnale da analizzare

[5] solitamente la transformata di Fourier

1.2.1 HPS

Il metodo Harmonic Product Spectrum, abbreviato come HPS, applica la FFT[6] al segnale in input per trovare le frequenze delle armoniche[7] presenti, una volta ottenute calcola il massimo comune divisore di queste armoniche che rappresenta la frequenza fondamentale.

Tale metodo anche se leggero e veloce da calcolare presenta anch'esso alcuni difetti, infatti risulta poco efficiente a lavorare sulle basse frequenze, inoltre basandosi sulla FFT lo spettro delle frequenze riconoscibili dipende dalla dimensione della finestra utilizzata e finestre di dimensioni maggiori richiedono maggior tempo per il calcolo della FFT stessa.

1.2.2 Altri

Esistono una varietà di metodi che si collocano nella seconda delle due classi analizzate, ad esempio il Cepstral, di cui però non parleremo in questo elaborato nonostante siano diversi dal metodo HPS data la loro complessità e l'intenzione di introdurre solamente le due classi di algoritmi per pitch-tracking.

[6]Fast Fourier Transform

[7]multipli interi della frequenza fondamentale

Capitolo 2

Wavelet

Prima di introdurre il concetto di Wavelet, occorre illustrare alcune nozioni fondamentali riguardanti la teoria dei segnali.

2.1 Trasformate

Le trasformate sono operazioni matematiche che, applicate ad un segnale, consentono di ottenere un' informazione aggiuntiva non ottenibile dal segnale originale.
Procederemo nell'introdurre il concetto di trasformata con un esempio pratico, la trasformata di Fourier[5] .

Premettendo che, un segnale è una funzione nel dominio del tempo, ovvero è una rappresentazione dell'ampiezza del segnale stesso in diversi istanti di tempo,e la sua frequenza è misurata in Hertz[1],la trasformata di Fourier, se applicata ad un segnale, consente di ottenere le frequenze[2] presenti in esso.

Negli esempi seguenti l'asse delle ascisse rappresenta il tempo mentre l'asse delle ordinate rappresenta l'ampiezza.

[1] cicli per secondo
[2] con i relativi valori di ampiezza

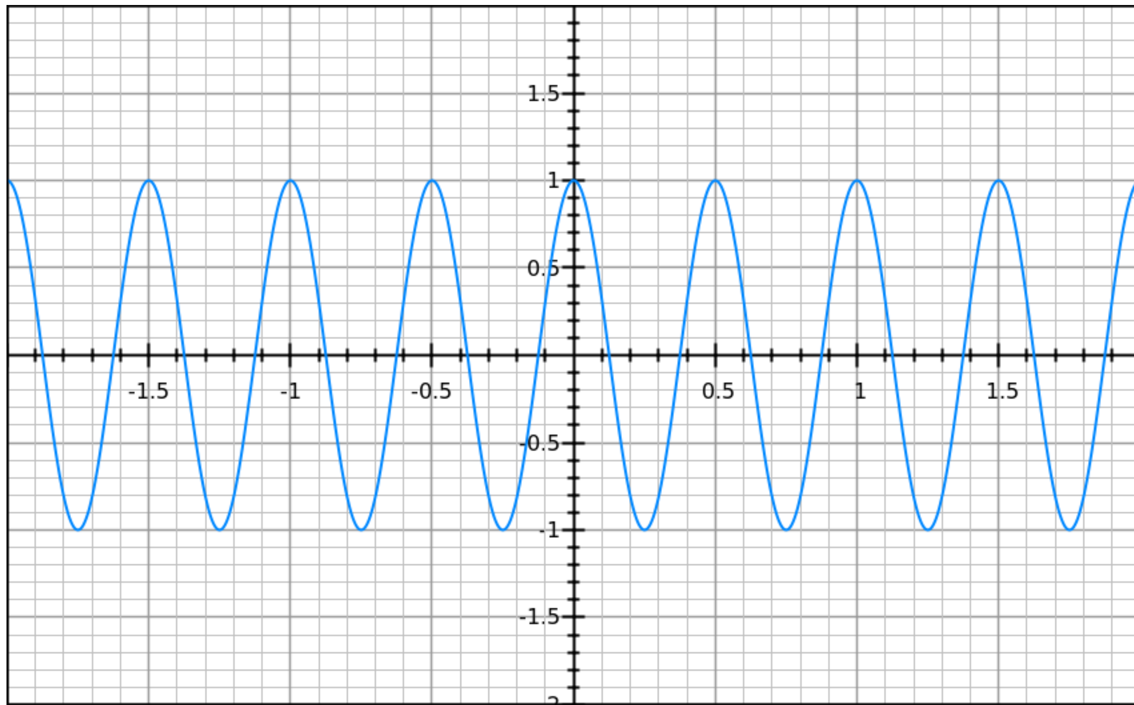


Figura 2.1: Esempio segnale con frequenza 2 Hz

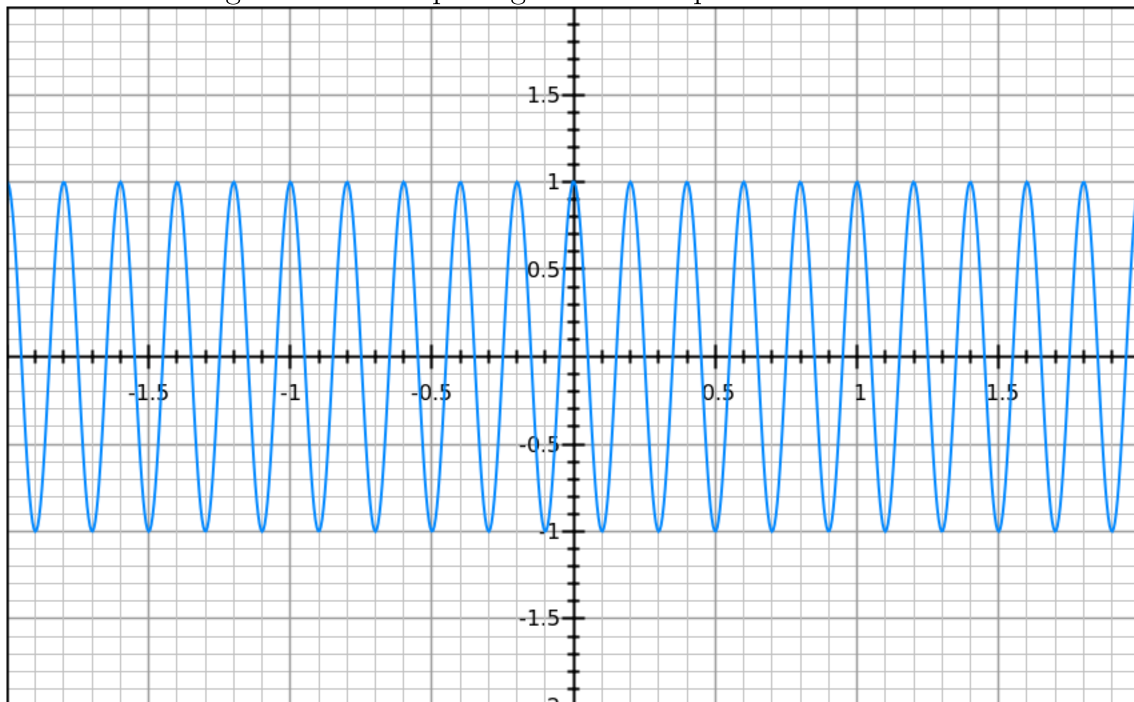


Figura 2.2: Esempio segnale con frequenza 5 Hz

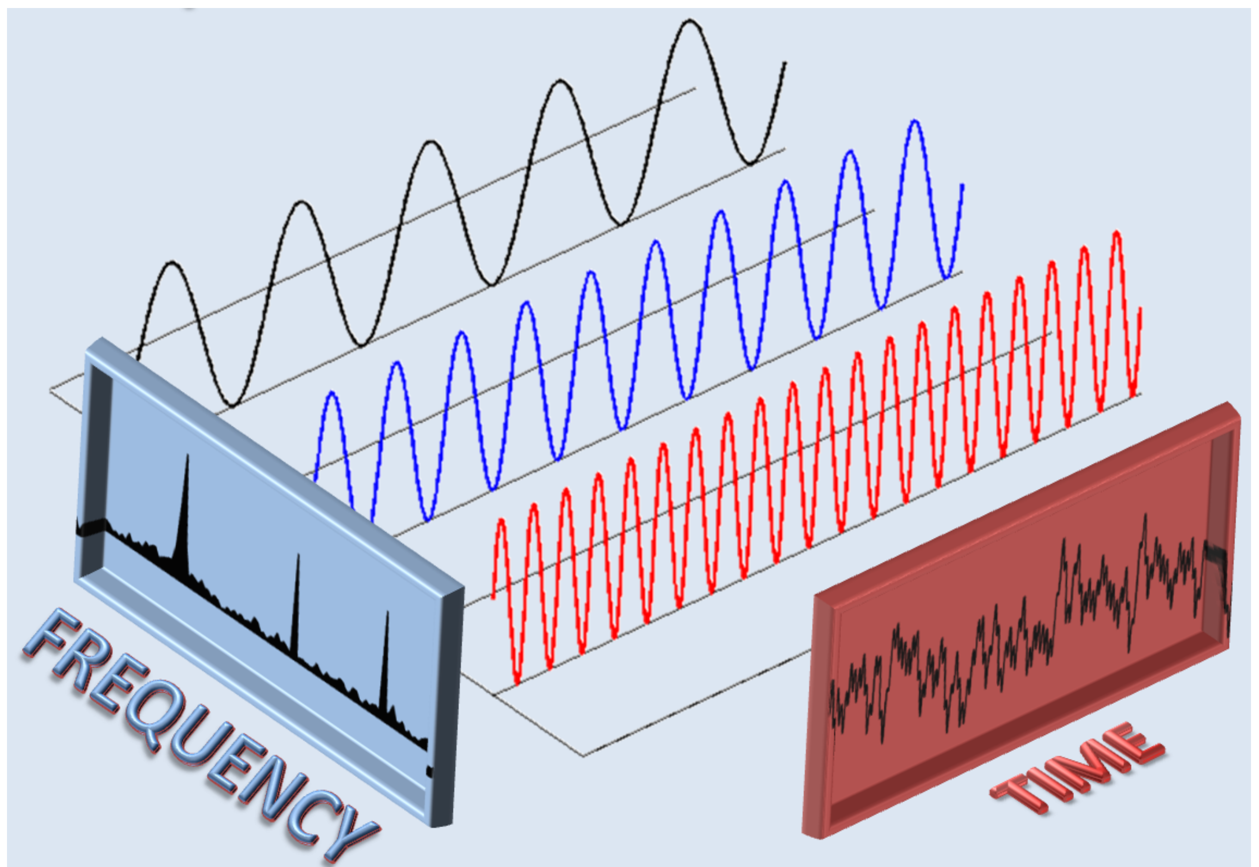


Figura 2.3: rappresentazione traformata di Fourier

Da una prima descrizione la trasformata di Fourier sembrerebbe la soluzione ideale per progettare un algoritmo di pitch tracking, poiché applicandola ad un segnale in entrata riusciamo ad ottenere i valori delle frequenze in esso contenute, tuttavia non è così.

Per meglio comprendere l'ultima affermazione introduciamo ora le nozioni di segnale stazionario e non stazionario.

- **segnale non stazionario** : un segnale le cui proprietà[3] non variano col variare del tempo.
- **segnale stazionario** : un segnale le cui proprietà variano col variare del tempo.

Compresi questi semplici concetti si può intuire che la FT non si presenta tra i metodi più efficaci per risolvere il problema del pitch tracking poiché, dato che i suoni sono ovviamente segnali non stazionari (infatti in caso contrario non saremmo in grado di produrre suoni diversi con la nostra voce o suonare note diverse con una chitarra o un pianoforte), sarebbe in grado di dirci quali frequenze esistono in un segnale ma non quando.

Per ovviare a questa mancanza si potrebbe pensare di suddividere il segnale in entrata in piccoli segmenti ed analizzare segmento per segmento, ottenendo così una buona risoluzione sia nel dominio della frequenza che nel dominio del tempo.

$$Complessita\ computazionale\ FFT^4\ :\ O(nlog(n)) \tag{2.1}$$

Tuttavia anche in questo caso sorge un problema: infatti il calcolo di questa trasformata risulta pesante in termini di risorse computazionali per la costruzione di un algoritmo real-time, dove i requisiti di velocità per garantire una bassa latenza sono fondamentali.

[3]ad esempio, la frequenza in esso contenuta

[4]Fast Fourier Transform, implementazione veloce della FT

2.2 La trasformata Wavelet

Dopo aver brevemente illustrato il concetto di trasformata matematica e il funzionamento della più famosa e utilizzata di esse, ovvero la trasformata di Fourier, introduciamo ora la trasformata Wavelet[6] .

La trasformata Wavelet si distingue dalla trasformata di Fourier poiché può essere usata per diversi scopi, infatti il suo comportamento viene guidato da una funzione chiamata Wavelet madre, la quale può essere scelta arbitrariamente entro alcune condizioni di ammissibilità e viene utilizzata per analizzare il segnale.

Ad esempio, la trasformata Wavelet con la Morlet Wavelet come funzione madre, fornisce una rappresentazione tempo-frequenza di un segnale (a differenza della FT che ci fornisce solo informazioni sulle frequenze), ovvero è in grado di dirci quali frequenze sono presenti e quando lo sono(in realtà ci permette di sapere quali frequenze sono presenti in un intervallo di tempo a causa del principio di indeterminazione di Heisenberg[4]).

Intuitivamente essa sottopone il segnale in entrata a vari filtri high-pass e low-pass, ovvero strumenti che filtrano solo la componente ad alta o bassa frequenza di un segnale, in questo modo una porzione di segnale corrispondete ad una certa banda di frequenze viene rimossa.

Il processo appena descritto viene applicato ripetutamente, durante il quale ad ogni iterazione viene scelta una porzione di segnale[5] su cui riapplicare i due filtri high-pass e low-pass, fino ad un certa soglia predefinita.
Questo procedimento prende il nome di decomposizione.

[5]solitamente la porzione low-pass

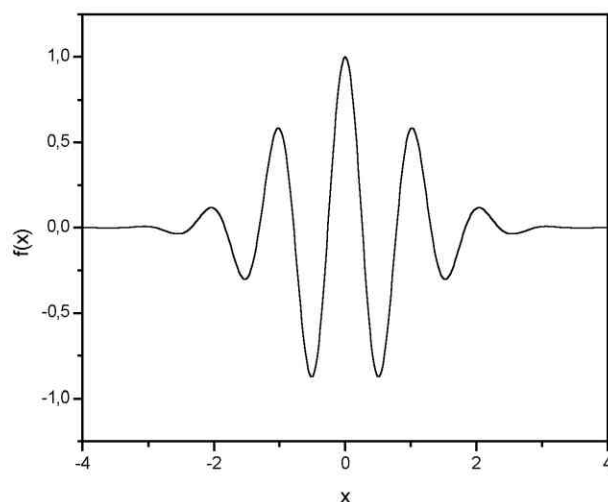


Figura 2.4: rappresentazione grafica della Morlet Wavelet

Una volta effettuata la decomposizione del segnale fino ad una soglia prestabilita avremo diverse porzioni dello stesso segnale che corrispondono però a bande di frequenze diverse (a causa dei filtri applicati), se adesso si ricongiungono insieme tutte queste informazioni su un grafico 3-D otterremo un grafico che sul primo asse ci fornisce una rappresentazione del tempo, sul secondo una rappresentazione delle bande di frequenze e sul terzo dei valori di ampiezza.
Applicando quindi questo procedimento avremo una descrizione di quali frequenze sono presenti nel segnale, i loro intervalli temporali e i relativi valori di ampiezza.

Questo è uno degli esempi d'uso della trasformata Wavelet, che si presenta come uno strumento potente e flessibile grazie alla possibilità di scegliere funzioni Wavelet diverse.
Tuttavia l'esempio appena descritto è un uso delle Wavelet nell'ambito del continuo, dato che lavoriamo con calcolatori abbiamo bisogno di un'applicazione discreta della trasformata.

2.3 Fast Lifting Wavelet Transform

Una delle applicazioni discrete più comuni delle Wavelet è quella di essere usate per la compressione dei dati, infatti la FLWT[6][7] è una particolare implementazione della trasformata Wavelet la quale divide un segnale in entrata in due componenti: approssimazione e dettaglio.

L'approssimazione è il componente più importante poiché contiene informazioni rilevanti come eventuali proprietà periodiche, altrimenti difficili da estrarre dal segnale originale.
Il dettaglio contiene informazioni utili alla ricostruzione del segnale originale, nel nostro caso può quindi anche essere scartato.

La FLWT può essere usata con diverse Wavelet madre, nell'implementazione dell'algoritmo che stiamo per descrivere è stata usata la Haar wavelet.

La FLWT usata con la Wavelet di Haar matematicamente equivale a sottoporre il segnale ad un filtro low-pass ed effettuare un'operazione di downsampling per produrre la componente di approssimazione e sottoporre lo stesso segnale ad un filtro high-pass ed effettuare un operazione di downsampling per produrre la componente di dettaglio.

Le seguenti sono le equazioni derivate da De-bauchies and Sweldens per la FLWT usata con la Wavelet di Haar:

$$d_0(n) = x(2n+1) \tag{2.2}$$

$$a_0(n) = x(2n) \tag{2.3}$$

$$d_1(n) = d_0(n) - a_0(n) \tag{2.4}$$

$$a_1(n) = a_0(n) + d_1(n) \tag{2.5}$$

[6]Fast Lifting Wavelet Transform

dove $x(n)$ è il segnale originale e $a_1(n)$ e $d_1(n)$ sono rispettivamente la prima approssimazione e il primo dettaglio.

Le ultime due equazioni possono essere semplificate nel seguente modo:

$$d_1(n) = x(2n+1) - x(2n) \tag{2.6}$$

$$a_1(n) = \frac{x(2n+1) + x(2n)}{2} \tag{2.7}$$

Applicando ripetutamente le ultime due equazioni sulla componente di approssimazione ottenuta dall'iterazione precedente, la Haar Wavelet FLWT fornisce un metodo di calcolo semplice e veloce per estrarre le informazioni significative contenute in un segnale, così da riuscire ad individuare eventuali periodicità e riuscire ad ottenere, ad ogni iterazione, una mole di dati su cui lavorare sempre minore[7].

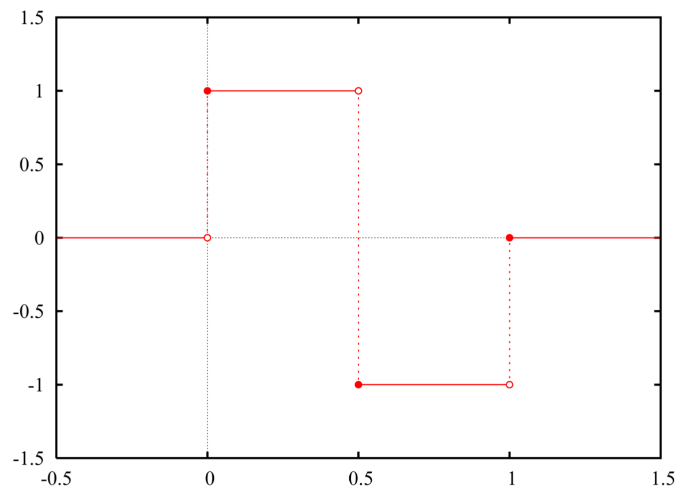


Figura 2.5: rappresentazione grafica della Wavelet di Haar

[7] precisamente la dimensione dei dati da analizzare viene dimezzata ad ogni passo

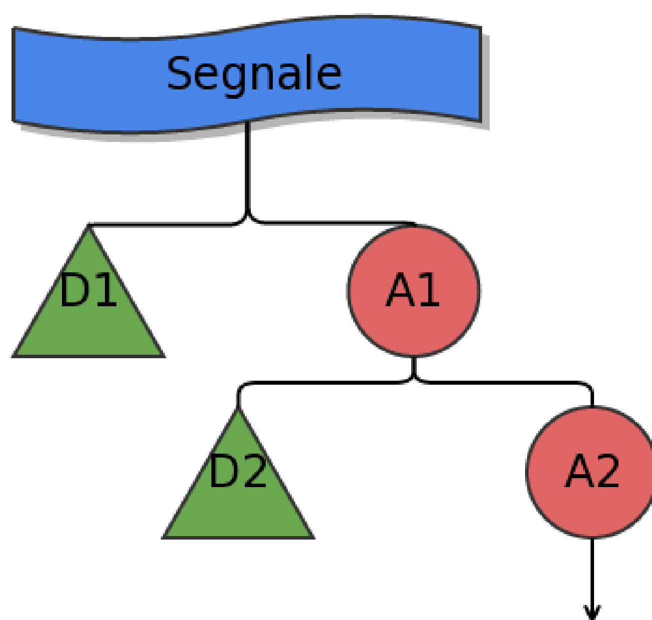


Figura 2.6: Schema FLWT, dove D rappresenta la componente di dettaglio e A la componente di approssimazione.

Inoltre dal momento che il frequency-range desiderato è compreso nell'intervallo di 80-3000 Hz, la ripetuta applicazione di filtri low-pass sulla componente di approssimazione non comporta alcuna perdità di informazione ai fini della rilevazione del pitch.

Capitolo 3

Algoritmo di pitch-tracking

3.1 Schema di lavoro

Una qualsiasi implementazione di un algoritmo di pitch tracking si sviluppa su uno schema generale condiviso.

Solitamente un applicazione richiede dati in input[1] da una scheda audio, li immagazzina in buffer di una dimensione arbitraria[2] e chiama la libreria (che implementa il vero algoritmo), la quale analizza il buffer con lo scopo di ricavarne il valore della frequenza fondamentale e restituisce un risultato, il tutto si sviluppa in un ciclo dove l'applicativo chiama continuamente le funzionalità della libreria.

[1] nel nostro caso un segnale sonoro

[2] solitamente di 1024/2048 MB

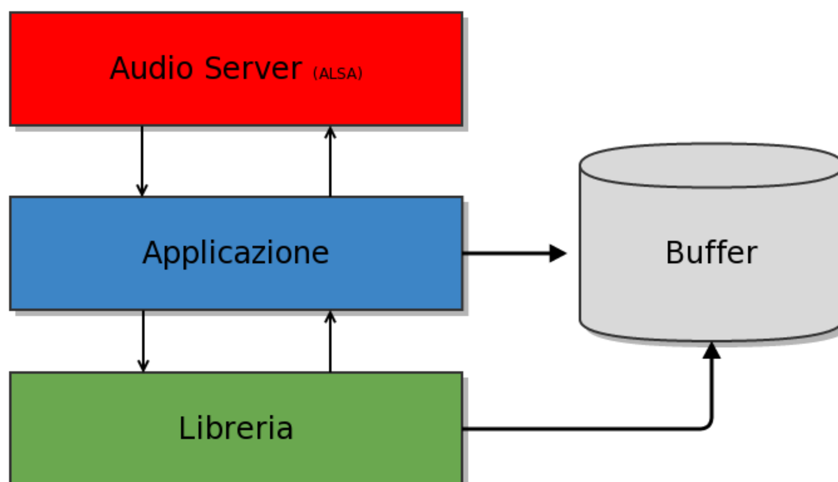


Figura 3.1: Modello di funzionamento di una libreria che implementa un algoritmo di pitch tracking

3.2 Campionamento e formati

I dati provenienti da una scheda audio sono nella maggior parte dei casi una rappresentazione del segnale analogico in entrata in codifica PCM[3], campionati con una certa frequenza e disponibili in un formato predefinito. Ad esempio, con il seguente comando:

```
arecord -r 44100 -f S16_LE
```

si istruisce il software ALSA[4] ,in un sistema basato su kernel Linux, a raccogliere dati dalla scheda audio principale in codifica PCM, con frequenza di campionamento pari a 44100 Hz e renderli disponibili come interi a 16 bit in formato little endian.

[3]pulse-code modulation

[4]Advanced Linux Sound Architecture

La scelta del formato e della frequenza di campionamento è fondamentale, poiché utilizzando un tipo di formato non adatto o una frequenza di campionamento troppo bassa si può incorrere in problemi con l'aliasing o il clipping[8] .

Ecco un esempio di campionamento di un segnale analogico, dove il tratto continuo indica il segnale originale e i punti indicano i campioni acquisiti.

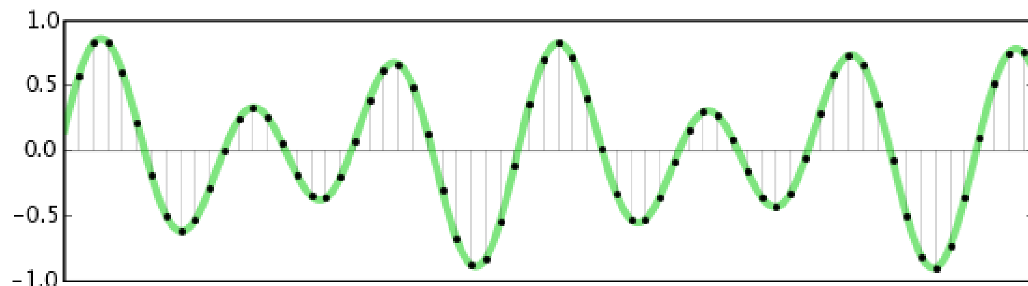


Figura 3.2: Esempio grafico del processo di campionamento

3.3 Algoritmo

Dopo aver introdotto alcuni concetti di base possiamo finalmente introdurre una descrizione dell'algoritmo che è stato implementato nella libreria, oggetto di questo elaborato.
Il seguente algoritmo è stato descritto nell'articolo di Eric Larson e Ross Maddox[1].

L'algoritmo si sviluppa su più fasi e più iterazioni per ogni finestra di segnale, ecco elencate le diverse fasi:

1. FLWT step
2. peak detection step
3. central mode distance step
4. averaging step
5. period step

3.3.1 FLWT phase

Durante questa fase vengono calcolate le equazioni 2.6, 2.7 riportate a pagina 16 per il segnale in entrata, il che equivale ad applicare la FLWT per ridurre la dimensione dei dati in input, evidenziare possibili proprietà periodiche e ridurre la quantità di rumore presente[5].

Ecco un esempio grafico per far comprendere quanto l'applicazione di questa operazione sia importante prima di effettuare qualsiasi tipo di analisi sulle proprietà del segnale, infatti si può notare come le successive applicazioni della trasformata dimezzino il numero di campioni ed eliminino il rumore presente nel segnale, così da rendere la sua periodicità maggiormente visibile dal calcolatore.

Le seguenti immagini sono state estratte dall'articolo di Ross Maddox e Eric Larson [1].

[5] se presente

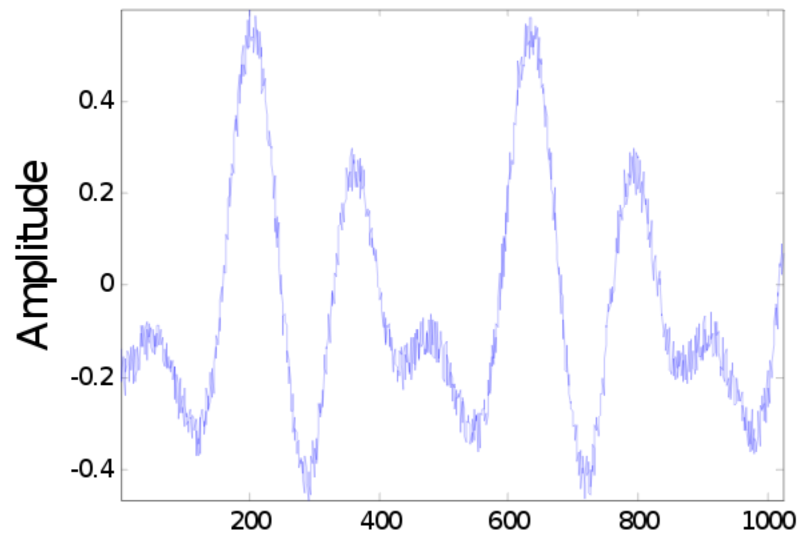


Figura 3.3: Segnale originale

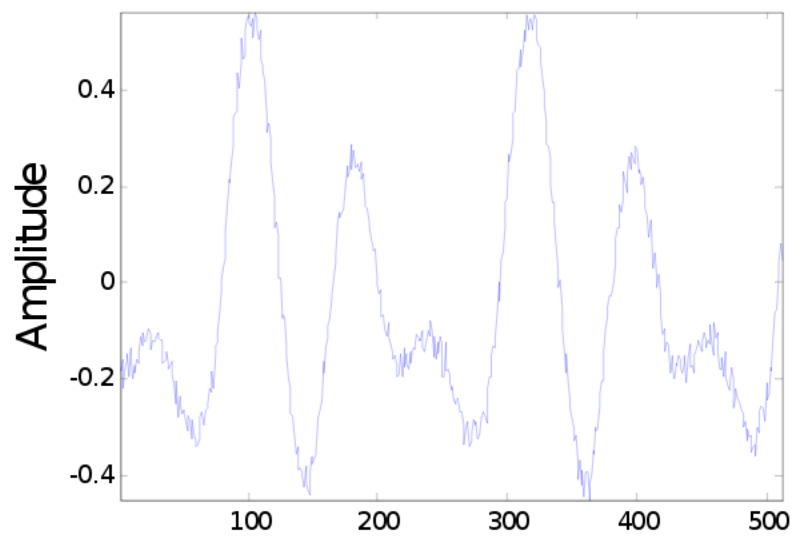


Figura 3.4: Prima approssimazione

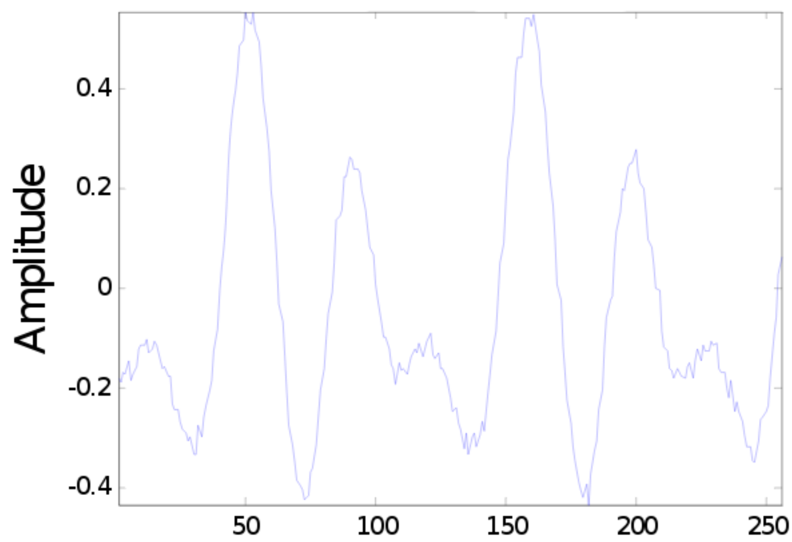


Figura 3.5: Seconda approssimazione

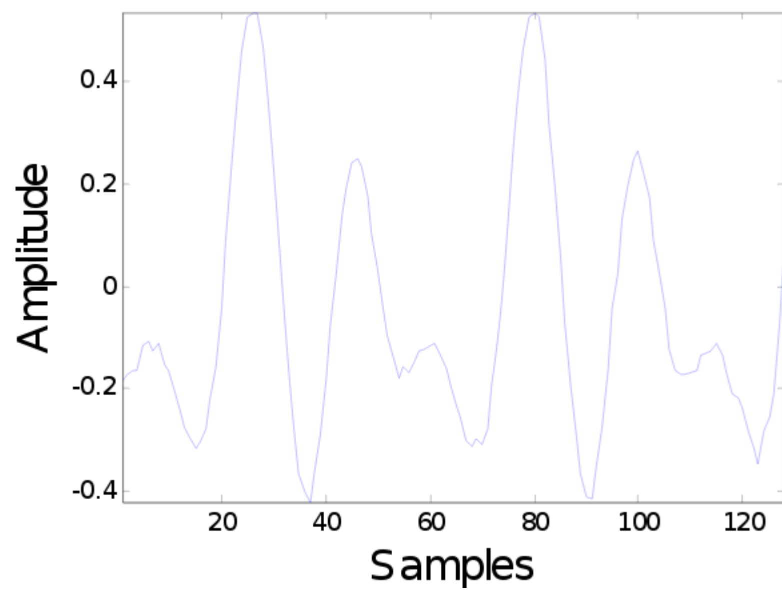


Figura 3.6: Terza approssimazione

3.3.2 Peak detection phase

Durante questa fase si analizza la finestra di segnale appena acquisita per trovarne i picchi di massimo/minimo.

La fase di peak detection si scompone in più step.
La finestra corrente viene analizzata per trovare il valore medio di ampiezza del segnale il quale, rappresentando la componente continua, dovrà essere sottratto ad ogni campione acquisito.
Successivamente vengono memorizzate le posizioni dei primi massimi/minimi dopo ogni zero-crossing se rispettano le due condizioni di ammissibilità :

$$c_i > m_a * p \tag{3.1}$$

$$i > m_{i-1} + \delta \tag{3.2}$$

dove:

- c_i : campione analizzato,
- i : indice del campione analizzato,
- m_a : massimo assoluto della finestra corrente,
- p : soglia arbitraria espressa in percentuale,
- m_{i-1} : indice dell'$i-1$-esimo massimo/minimo locale.

La prima condizione impone al campione c_i di essere maggiore di una certa quantità, calcolata come percentuale sul massimo globale dell'attuale finestra. Tale condizione definisce un limite per numero di massimi/minimi locali memorizzabili uguale ad il numero di zero-crossing meno uno.

La seconda condizione garantisce che nessun campione possa essere considerato un massimo/minimo locale se non rispetta la distanza di δ campioni

dall'ultimo massimo/minimo locale registrato.

La quantità δ è un parametro controllabile in base alla massima frequenza riconoscibile F ed al livello corrente di approssimazione del segnale, definito come il numero di volte i che la FLWT è stata applicata (F_s è la frequenza di campionamento e F_m è la massima frequenza riconoscibile).

$$\delta = max\left(\left\lfloor \frac{F_s}{2^i * F_m} \right\rfloor, 1\right). \tag{3.3}$$

Una volta riconosciuti i massimi e minimi locali presenti nella finestra corrente (secondo le condizioni prima illustrate) si procede a calcolare le distanze tra essi.

Per ogni indice m_i (corrispondente al massimo/minimo i) viene calcolata la distanza tra tale indice e i successivi indici m_{i+1}, m_{i+2}, m_{i+3}, $\ldots m_{i+N}$ dove N è un parametro arbitrario.

Lo schema di calcolo per le distanze appena illustrato, serve a garantire che sinusoidi con più di un massimo/minimo locale per periodo siano analizzate correttamente, inoltre questo meccanismo non causa dimezzamenti della frequenza poiché la finitezza della finestra di segnale garantisce che le distanze corrette siano in maggior numero delle distanze scorrette le quali possono essere causate da armoniche.

Una volte calcolate le distanze, per ognuna di queste viene calcolato il numero di distanze vicine ad essa (entro un certo parametro δ definito come prima).

La distanza con più valori vicino ad essa verrà presa come distanza centrale.

Ecco un esempio:

L'insieme delle distanze calcolate D contiene

$$D = \{24, 26, 47, 48, 49\}$$

con i relativi valori (quante volte queste distanze sono state trovate nella finestra corrente):

$$V = \{2, 1, 1, 2, 1\}$$

Ora calcoliamo per ogni distanza la quantità α, ovvero quella che possiamo definire come distanza modale, che ci indica il numero di distanze vicino ad essa, dove il parametro di tolleranza $\delta = 1$.

$$\begin{aligned}
\alpha_1 &= 23 * 0 + 24 * 2 + 25 * 0 = 48 \\
\alpha_2 &= 25 * 0 + 26 * 1 + 27 * 0 = 26 \\
\alpha_3 &= 46 * 0 + 47 * 1 + 48 * 2 = 143 \\
\alpha_4 &= 47 * 1 + 48 * 2 + 49 * 1 = 192 \\
\alpha_5 &= 48 * 2 + 49 * 1 + 50 * 0 = 145
\end{aligned}$$

In questo esempio δ_4 verrà presa come distanza modale centrale della finestra corrente.

Esiste inoltre un meccanismo per gestire i casi di pareggio tra distanze modali.

Se la distanza modale α_i appena calcolata possiede un valore uguale a quello della distanza modale centrale γ finora riconosciuta, la nuova distanza modale diventa centrale se e solo se rispetta la seguente condizione:

$$D_i = 2 * D_\gamma \tag{3.4}$$

ovvero se la nuova distanza è esattamente il doppio della distanza centrale corrente, questo meccanismo garantisce che non vi siano dimezzamenti della frequenza.

3.3.3 Averaging phase

Una volta che la distanza modale centrale è stata selezionata, viene applicato uno schema di averaging[6] per incrementare la precisione dell'algoritmo. Durante questo processo viene calcolata la media aritmetica delle distanze vicine a quella centrale (entro il parametro δ), e viene presa come periodo della finestra di segnale corrente.

Il meccanismo appena illustrato aumenta la risoluzione nel dominio della frequenza, in particolare risulta molto efficiente nel riconoscimento della frequenze alte dove anche la differenza di un solo intero nel valore del periodo può portare ad un grosso errore nel valore finale della frequenza.

3.3.4 Period phase

Nell'ultima fase dell'algoritmo si controlla se il periodo T_i riconosciuto nell'iterazione corrente i dell'algoritmo è simile al periodo T_{i-1} riconosciuto nell'iterazione precedente.

[6] media aritmetica

La similarità è un test di uguaglianza tra i due periodi con una tolleranza indicata dal parametro δ e viene espressa dalla seguente condizione:

$$|T_i * 2 - T_{i-1}| <= \delta \tag{3.5}$$

Il valore T_i viene moltiplicato per due per ovviare al downsampling effettuato durante l'applicazione della FLWT.

Se la condizione 3.5 è verificata, si assume che T_{i-1} sia il periodo della finestra di segnale corrente e il relativo valore di frequenza F viene calcolato come segue:

$$F = \frac{F_s}{2^{i-1} * T_{i-1}} \tag{3.6}$$

dove F_s è la frequenza di campionamento.

Nel caso la condizione 3.5 non sia verificata si itera nuovamente l'algoritmo appena descritto applicando un nuovo livello di FLWT e tutte le fasi successive.

Capitolo 4

WaveTrack

WaveTrack è l'implementazione ottimizzata dell'algoritmo appena descritto prodotta durante lo sviluppo di questa tesi.
La libreria è scritta nel linguaggio C e presenta alcune nuove funzionalità e caratteristiche non presenti nell'algoritmo iniziale.

4.1 Include e define

Nella libreria vengono inclusi i seguenti file .h :

```
#include<math.h>
#include<stdlib.h>
#include<unistd.h>
#include"wavetrack.h"
#include<string.h>
```

Come si può notare tutte le librerie incluse nel codice aderiscono allo standard POSIX.

Inoltre nel codice sono presenti le seguenti istruzioni preprocessore, le quali definiscono alcuni valori standard per i parametri configurabili della libreria:

```
#define SAMPLE_RATE 44100.0
#define FLWT_LEVELS 6
#define DIFFS_LEVELS 3
#define MAX_FREQ 3000.0
#define THRESHOLD_RATIO 0.75

#define SILENCE_THRESHOLD 0.7

/* ring buffer operation macros */
#define RINGSIZE  3 //dimension of the ring buffer, ↩
    a bigger buffer has more precision in returned ↩
    values, but less reactive over short (duration) ↩
    notes
                    // a smaller one gives more ↩
                       reactivity  but less ↩
                       accuracy, when operating in ↩
                       SPEED mode, the suggested ↩
                       size is 3, when in ACCURACY ↩
                       mode is 2
#define PRECISION 5 //precision in Hz

/* Algorithm operating mode parameter*/
#define ACCURACY  0
#define SPEED     1
```

- SAMPLE_RATE : definisce il valore per la frequenza di campionamento usata e deve essere usata coerentemente con il metodo di acquisizione dei dati in input;
- FLWT_LEVELS : definisce il numero massimo di applicazioni della FLWT ad una finestra di segnale, di conseguenza definisce anche il numero massimo di iterazioni dell'algoritmo;
- DIFFS_LEVELS : definisce il numero di massimi/minimi successivi da prendere in considerazione per il calcolo delle distanze;
- MAX_FREQ : definisce la massima frequenza riconoscibile dall'algoritmo, importante per il calcolo del δ, 3.3;
- THRESHOLD_RATIO : definisce il parametro p nel calcolo della condizione 3.1;
- SILENCE_THRESHOLD : definisce una soglia, espressa in percentuale, che permette di filtrare il segnale in entrata al fine di riconoscere il silenzio;
- RINGSIZE : definisce la dimensione di uno specifico buffer chiamato ring, il cui funzionamento verrà illustrato successivamente;
- PRECISION : definisce il livello desiderato di precisione del risultato, espresso in Hz, quando la libreria viene chiamata in modalitá ACCURACY, se il livello di precisione non viene raggiunto non viene restituito nessun valore di ritorno;
- ACCURACY : se la libreria viene chiamata con il flag ACCURACY restituisce un valore di ritorno non nullo se e solo se viene raggiunto il livello di precisione desiderato;
- SPEED : se la libreria viene chiamata con il flag SPEED restituisce un valore di ritorno anche se il livello di precisione desiderato non viene raggiunto;

4.2 Funzioni setter

Per la gestione dei parametri configurabili della libreria è stata dichiarata la seguente struttura nel file wavetrack.h:

```
/* structure containing the algorithm parameters */
struct pitch_tracker_params;
```

la struct *pitch_tracker_params*, come si può notare, viene solo dichiarata, ma la sua struttura interna non viene descritta nel file wavetrack.h, in questo modo il compilatore viene informato solo sull'esistenza della struttura ma non sulla sua composizione, cosicché programmi scritti per funzionare con vecchie versioni della libreria riescano a funzionare con future versioni della stessa senza dover apportare modifiche.

Prima di poter usare le funzionalità offerte dalla libreria bisogna chiamare la seguente funzione :

```
/* init function which sets the algorithm standard ↩
   parameters, to be called before compute_pich */
struct pitch_tracker_params* open_pitch_tracker();
```

All'interno di questa funzione viene allocata una struttura *pitch_tracker_params* e viene configurata con i parametri standard dell'algoritmo definiti nelle macro prima descritte.

Quando invece non si ha più bisogno delle funzionalità offerte dalla libreria il programmatore dovrà chiamare la seguente funzione:

```
/* to be called when the library is no longer to be ↩
   used */
void close_pitch_tracker(struct pitch_tracker_params ↩
   **settings);
```

altrimenti si causerà un memory leak poiché non è stata rilasciata una porzione di memoria precedentemente allocata.

Per impostare i diversi parametri della libreria sono state implementate le seguenti funzioni:

```
/* setter functions */
void tracker_set_maxfreq(struct pitch_tracker_params ↩
   *settings, double freq);
void tracker_set_flwtlevels(struct ↩
   pitch_tracker_params *settings, unsigned int ↩
   levels);
void tracker_set_difflevels(struct ↩
   pitch_tracker_params *settings, unsigned int ↩
   levels);
void tracker_set_thresholdratio(struct ↩
   pitch_tracker_params *settings, double ratio);
```

- *tracker_set_maxfreq*() : imposta la massima frequenza riconoscibile dalla libreria, prende come parametri un puntatore alla struttura *pitch_tracker_params* ed un numero reale di tipo *double* che rappresenta l'impostazione desiderata espressa in Hz;

- *tracker_set_flwtlevels*() : imposta il numero massimo di iterazioni dell'algoritmo e conseguentemente il numero massimo di applicazioni della FLWT ad una finestra di segnale.
  Solitamente per il riconoscimento del pitch in una finestra bastano 2-3 iterazioni, la libreria può quindi trarre benefici da un alto valore di questo parametro solo nel caso in cui la finestra di segnale analizzata contenga un livello di rumore elevato.

- *tracker_set_difflevels*() : imposta il numero di massimi/minimi successivi da prendere in considerazione durante il calcolo delle distanze.
  Il valore di questo parametro risulta fondamentale per la corretta gestione di sinusoidi con più di un massimo/minimo per periodo, una configurazione errata potrebbe indurre l'algoritmo a commettere errori nel risultato a causa di armoniche presenti nel segnale, le quali compaiono come multipli interi della frequenza fondamentale.

- *tracker_set_thresholdratio*() : imposta la percentuale per il calcolo della prima condizione 3.1 , tale condizione è utile per stabilire una soglia sotto il quale massimi e minimi locali non verranno memorizzati per essere inclusi nel calcolo delle distanze durante la fase di peak detection.

Le funzioni finora illustrate operano sulla struttura *pitch_tracker_params* che viene definita nel file wavetrack.c nel seguente modo:

```
/* definition of the struct containing the algorithm ↩
   parameters */
struct pitch_tracker_params{
   unsigned int diff_levels;
   unsigned int flwt_levels;
   unsigned int max_freq;
   double max_threshold_ratio;
};
```

4.3 La funzione compute_pitch()

La funzione *compute_pitch*, definita nel file wavetrack.h come segue :

```
/* call to compute the pitch of the current window */
double compute_pitch(double *sample_vector, unsigned ↩
   int samplecount, unsigned int sample_format, ↩
   struct pitch_tracker_params *settings, int op_mode);
```

essa rappresenta il punto di incontro tra un qualsiasi programma che utilizza la libreria WaveTrack e la libreria stessa, infatti se chiamata fornendo i parametri richiesti in modo corretto, restituisce il valore riconosciuto di frequenza fondamentale della finestra di segnale che si vuole analizzare.

Ecco una descrizione dei parametri richiesti dalla funzione:

- *sample_vector* : puntatore al buffer che contiene la finestra di segnale che si vuole analizzare. È fortemente consigliato usare buffer la cui dimensione è una potenza di due.

  La scelta della dimensione del buffer risulta fondamentale, un buffer troppo piccolo (512 campioni) migliorerà la risoluzione in tempo dellàlgoritmo ma peggiorerà sensibilmente la risoluzione in frequenza[1]. Un buffer troppo grande (4096 campioni) migliorerebbe di molto la risoluzione in frequenza ma peggiorerebbe di molto la risoluzione in tempo[2].

[1] alcune frequenze basse verrebbero completamente tagliate fuori dallo spettro di frequenze riconoscibili dall'algoritmo

[2] i tempi di latenza aumenterebbero sensibilmente

- *sample_count* : indica la dimensione del buffer, espressa come numero di campioni;
- *sample_format* : indica il formato di acquisizione dei dati, i possibili valori sono definiti nel file wavetrack.h e sono i seguenti:
    1. S8 : *signed integer* 8 bit,
    2. U8 : *unsigned integer* 8 bit,
    3. S16 : *signed integer* 16 bit,
    4. U16 : *unsigned integer* 16 bit,
    5. S24 : *signed integer* 24 bit,
    6. U24 : *unsigned integer* 24 bit,
    7. S32 : *signed integer* 32 bit,
    8. U32 : *unsigned integer* 32 bit,
- *settings* : puntatore alla struttura contenente i valori dei parametri configurabili dell'algoritmo;
- *op_mode* : imposta il metodo di lavoro della libreria, i valori possibili sono i seguenti:
    1. ACCURACY,
    2. SPEED.

La routine descritta finora funge solamente da "wrapper" per ragioni che verranno illustrate successivamente.

4.4 Core

Il vero "cuore" della libreria è la funzione *__compute_pitch*(), la quale implementa concretamente l'algoritmo oggetto di questa tesi.
L'implementazione è divisa anch'essa, come l'algoritmo, in più fasi.

4.4.1 Global max/min phase

In questa fase, subito dopo aver configurato i parametri base dell'algoritmo, si scorre il vettore contenente i campione della finestra di segnale corrente per calcolarne il massimo e minimo globale e la componente continua[3].

```
/* compute amplitude Threshold and the DC ↩
   component */
for( i = 0 ; i< samplecount ; i++){
   DC_component += approx[i];

   max_value = max( approx[i], max_value );
   min_value = min( approx[i], min_value );
}
```

```
DC_component = DC_component/samplecount;
max_threshold = (max_value - ↩
   DC_component)*max_threshold_ratio + DC_component;
min_threshold = (min_value - ↩
   DC_component)*max_threshold_ratio + DC_component;
```

Come si può notare nel secondo spezzone di codice, la componente continua del segnale o valore medio, non viene concretamente sottratta da ogni elemento del buffer *approx* ma viene in realtà aggiunta nel calcolo delle quan-

[3] media aritmetica di tutti i valori nel vettore

tità *max_threshold* e *min_threshold*, così da risparmiare tempo in termini computazionali.

4.4.2 Silence filter phase

Dopo aver calcolato il massimo assoluto della finestra di segnale viene effettuato un semplice test per filtrare il silenzio e non sprecare risorse nell'analizzare set di dati inutili da elaborare.

```
if( sample_format && max_value < sample_format ){
   last_mode = UNPITCHED;
   last_iter = NONE;
   return UNPITCHED;
}
```

4.4.3 FLWT phase

In questa fase viene applicata la FLWT e viene calcolato il parametro δ 3.3.

```
   /* perform the FLWT */
   curr_sample_number /= 2;
   for(j=0; j<=curr_sample_number ; j++){
      approx[j] = ( approx[2*j] + approx[2*j + 1] ↩
         )/2;
   }
```

Dallo spezzone di codice riportato dal sorgente si può notare che delle due equazioni 2.6, 2.7 sia stata implementata solo la 2.7, questo perchè per gli scopi della libreria solo la componente di approssimazione risulta utile ai fini del pitch tracking, inoltre non calcolando la componente di dettaglio si risparmiano risorse computazionali.

```
power2<<=1;
```

```
dist_delta = (int) max( floor( SAMPLE_RATE/ ( ↩
    max_freq * power2 ) ), 1);
```

In quest'altro spezzone di codice si vuole inoltre mostrare il calcolo del δ e l'uso che viene fatto della variabile *power2*, infatti per diminuire il numero di chiamate alla libreria *math.h*, tutti i calcoli che coinvolgono potenze di 2 sono stati effettuati applicando uno shift alla variabile *power2*, così da ottenere tempi di computazione minori.

4.4.4 Peak detection phase

Nella peak detection phase è stato implementato il processo di riconoscimento dei massimi e minimi,calcolo delle distanze e scelta della distanza modale centrale.

```
for( j=1 ; j< curr_sample_number ; j++ ){
    sign_test = approx[j] - approx[j-1];

    if( prev_sign_test >= 0 && sign_test < 0 ){
        if( approx[j-1] >= max_threshold && ↩
            zero_crossed && !too_close){
            maxs[maxs_no] = j-1;
            maxs_no++;
            zero_crossed = 0;
            too_close = dist_delta;
        }
    }
```

```
        else if( prev_sign_test <=0 && sign_test>0 ){
            if( approx[j-1] <= min_threshold && ↩
                zero_crossed && !too_close ){
                mins[mins_no] = j-1;
                mins_no++;
                zero_crossed = 0;
                too_close = dist_delta;
            }
        }

        if( ( approx[j] <= DC_component && ↩
            approx[j-1] > DC_component ) || ( ↩
            approx[j] >= DC_component && approx[j-1] ↩
            < DC_component ) )
            zero_crossed = 1;

        prev_sign_test = sign_test;

        if(too_close)
            too_close--;

    }
```

Nella prima parte della porzione di codice appena mostrata viene dapprima eseguito il test della derivata prima, se il test evidenzia l'esistenza di un massimo o minimo locale viene controllato che il minimo/massimo appena trovate sia sufficientemente distante dal precedente, in tal caso viene memorizzato nel vettore appropriato e la variabile *zero_crossed* viene reimpostata a zero, il che impedisce la memorizzazione di un altro massimo/minimo finchè non si verifichi nuovamente uno zero-crossing nel segnale.

Nella seconda parte viene gestito il conteggio della variabile *too_close* che equivale al parametro δ calcolato in precedenza, inoltre è presente un costrutto *if* per testare l'avvenimento di uno zero-crossing.

In quest'ultimo controllo viene testata la variabile che in realtà contiene il valore medio del segnale invece dello zero, poiché precedentemente lo stesso valore non è stato sottratto dai campioni della finestra corrente, il tutto per ottimizzare e velocizzare la libreria.

Dopo aver memorizzato nei vettori *maxs* e *mins* ogni massimo o minimo locale del segnale avvenuto dopo uno zero-crossing, si può procedere al calcolo delle distanze.

```
for( j=0; j< maxs_no ; j++)
    for( i=1 ; i<= diff_levels ; i++ )
        if( i+j < maxs_no ){
            d_index = abs_val( maxs[j] - ↩
                maxs[i+j] );
            distances[d_index]++;
            if( d_index > max_d_index )
                max_d_index = d_index;
        }
```

È stato riportato solo il codice che mostra il calcolo delle distanze tra i massimi locali poiché la porzione riguardante il calcolo tra i minimi locali è esattamente uguale.

Calcolare le distanze e memorizzarle in un vettore sarebbe molto inefficiente, poiché successivamente, nella scelta della distanza modale centrale, bisognerebbe scorrere tutto il vettore ogni volta che si analizza una distanza candidata, questo perchè non si può fare nessuna ipotesi sull'ordinamento del vettore.

Quindi, per ottimizzare il processo di calcolo, la distanza tra due generici elementi $maxs[j]$ e $maxs[i+j]$ viene calcolata e memorizzata nella variabile *d_index* e viene incrementato l'elemento di indice *d_index* del vettore *distances*.

In questo modo, con due sole istruzioni[4] per ogni elemento del vettore, riusciamo a calcolare sia le distanze per l'insieme di massimi locali riconosciuti sia le occorrenze di ogni distanza, che ci tornerà utile nella scelta della distanza modale centrale.

Inoltre la variabile *max_d_index* viene usata per tenere traccia del massimo indice usato nel vettore *distances.* così da risparmiare iterazioni la prossima volta che si andrà ad operare su tale vettore.

Nel codice viene mostrato l'uso della funzione *abs_val*(), definita come segue:

```
static inline int abs_val(double num){ return num >= ↩
   0. ? num : -num; }
```

invece che usare la funzione per calcolare il valore assoluto definita nella libreria *math.h*, si è scelto di implementarne una versione propria e dichiararla come *static inline*, questo elimina gli overhead portati da chiamate ad una libreria esterna e consente di raggiungere velocità di esecuzione (per le chiamate alla funzione) prossime a quelle di una macro.

Una volte calcolate le distanze, sia per i massimi che per i minimi locali della finestra di segnale corrente, si procede a scegliere la distanza modale centrale.

[4] moltiplicate per la variabile *diff_levels*

```
for( j = -dist_delta ; j <= dist_delta ; ↩
    j++){
    if( i+j >= 0 && i+j <= max_d_index )
        count += distances[i+j];
}
```

Durante l'elaborazione di una distanza candidata questa può classificarsi in quattro diversi casi:

1. il punteggio[5] della distanza candidata è uguale al miglior punteggio calcolato finora. In questo caso la distanza candidata viene scelta come migliore se si verifica una delle seguenti condizioni:

    - il valore assoluto della distanza candidata meno quello della distanza modale centrale della finestra precedente rispetta una certa tolleranza;
    - il valore della distanza candidata è esattamente il doppio del valore della distanza con il miglior punteggio finora calcolato;

    ```
    if( count == center_mode_c && count ↩
        > floor(curr_sample_number/i/4) ){
        if( last_mode != UNPITCHED && ↩
            abs_val( i - last_mode/power2 ↩
            ) <= dist_delta ){
            center_mode_i = i;
        }
        else if( i== center_mode_i*2 ){
            center_mode_i = i;
        }
    ```

[5]la somma del numero di occorrenze della distanza stessa e delle distanze entro un certo parametro δ

2. il punteggio della distanza candidata è il maggiore calcolato finora e viene quindi scelta come miglior distanza;

```
         else if( i== center_mode_i*2 ){
            center_mode_i = i;
```

3. il punteggio della distanza candidata è di un intero minore del finora migliore e il valore assoluto della distanza stessa meno quello della distanza modale centrale della finestra precedente rispetta una certa tolleranza.
   Se la condizione appena illustrata si verifica, essa viene scelta come distanza migliore.

```
         else if( count == center_mode_c-1 && ↩
            last_mode > UNPITCHED && abs_val( ↩
            i - last_mode/power2 ) <= ↩
            dist_delta )
            center_mode_i = i;
```

4. Il punteggio della distanza candidata è inferiore al migliore calcolato, in questo caso si continua ad iterare sul vettore *distances* scartando la distanza appena analizzata.

Alla fine del processo appena descritto, la distanza considerata migliore ,dopo aver analizzato tutti gli elementi del vettore *distances*,diviene la distanza modale centrale.

4.4.5 Averaging phase

Dopo aver selezionato la distanza modale centrale si procede ad applicare lo schema di averaging descritto nella sezione 3.3.3.

```
    if( center_mode_i > 0 ){
        for( i= -dist_delta ; i <= dist_delta ; ↩
            i++){
            if( center_mode_i + i >= 0 && ↩
                center_mode_i + i <= max_d_index ){
                if( distances[center_mode_i+i] == ↩
                    0 )
                    continue;

                mean_no += ↩
                    distances[center_mode_i+i];
                mean += (center_mode_i+i) * ↩
                    distances[center_mode_i+i];
            }

        }
        mode = mean / mean_no;
    }
```

A questo punto si può assumere che la variabile *mode* contenga il periodo della finestra di segnale corrente.

4.4.6 Period phase

In questa fase si controlla la similarità del periodo appena calcolato con il periodo trovato durante l'iterazione precedente, se essa si presenta entro una certa tolleranza δ calcolata come mostrato nell'equazione 3.3 la libreria calcola il valore del pitch e lo restituisce come valore di ritorno.

```
    /* if the mode distance is equivalent to that ↩
       of the previous level , then is taken as the ↩
       period , otherwise next level of FLWT */
     if( old_mode_value>0.  && abs_val( 2*mode − ↩
        old_mode_value ) <= dist_delta  ){
       free(maxs);
       free(mins);
       free(distances);

       pitch = SAMPLE_RATE/( ↩
          power2/2*old_mode_value );
       last_iter = curr_level−2;
       last_mode = mode;
       return pitch;
     }
```

4.4.7 One-shot period method

Oltre al classico metodo per verificare il riconoscimento di una frequenza fondamentale nel segnale, il quale necessità almeno di due iterazioni dell'algoritmo, è stato aggiunta una nuova funzionalità per riconoscere più velocemente il pitch in un segnale effettuando una sola iterazione.

Il seguente metodo si basa sull'assunzione che spesso una singola nota duri in misura maggiore dell'arco di tempo necessario per l'acquisizione dei campioni di una finestra di segnale, ovvero la componente di latenza dovuta alla scheda audio,device driver e software che si utilizza per prelevare il segnale.

Il software ALSA ad esempio per acquisire 2048 campioni in formato S16 impiega circa 21 ms[6].

Una volta svolte tutte le fasi precedenti, la libreria confronta il periodo della prima iterazione dell'algoritmo con il periodo della finestra di segnale precedente, se questi risultano essere simili viene subito restituito il valore di frequenza relativo al periodo appena riconosciuto.

Nella maggior parte dei casi quindi, sfruttando questa funzionalità, i primi 20-30 ms di un segnale vengono riconosciuti effettuando 2-3 iterazioni dell'algoritmo, e la restante parte di segnale che presenta una periodicità simile viene riconosciuta in una sola iterazione, cosicché nel caso pessimo si dimezzi il tempo necessario alla libreria per trovare la frequenza fondamentale.

Ecco l'implementazione del metodo appena illustrato:

```
/* if the mode from the previous windows is ↩
   similar to the one computed in the current ↩
   window, then it is the same frequency */
else if( last_mode > 0. && curr_level == 1 && ↩
   last_iter>-1 && ( abs_val( ( last_mode*( ↩
   power2<<last_iter)) - mode ) ) <= ↩
   dist_delta ){
   free(maxs);
   free(mins);
   free(distances);

   pitch = SAMPLE_RATE/( power2*mode );
   last_mode = mode;
   last_iter = 1;
   return pitch;
}
```

[6] millisecondi

4.4.8 Ring buffer

Come descritto in precedenza la funzione *compute_pitch*() è in realtà un "wrapper" per la funzione *__compute_pitch*(), la quale implementa il vero algoritmo di pitch tracking.
La prima routine ha però l'importante compito di implementare il *ring buffer*.

Il ring buffer viene usato dalla libreria per mitigare la presenza di valori errati riconosciuti dall'algoritmo e allo stesso tempo per migliorare la precisione dei valori restituiti.

Il vettore *ring* viene sfruttato nel seguente modo : ogni volta che la funzione *__compute_pitch*() restituisce un risultato, questo viene salvato nel ring buffer, a questo punto viene estratta la coppia di valori $ring[i]$, $ring[i+j]$ che minimizza la seguente quantità:

$$precision = |\ ring[i] - ring[i+j]\ | \tag{4.1}$$

e viene calcolato il valore *pitch* :

$$pitch = \frac{ring[i] + ring[i+j]}{2} \tag{4.2}$$

Ora la libreria ha due comportamenti possibili a seconda del contenuto del parametro *op_mode* :

- se *op_mode* contiene il valore $ACCURACY$ e la quantità *precision* rispetta una certa tolleranza definita nella macro *PRECISION*, il valore della variabile *pitch* viene restituito come valore di ritorno, altrimenti la finestra di segnale corrente verrà considerata come *unpitched*[7].
- se *op_mode* contiene il valore *SPEED* e la quantità *precision* rispetta una certa tolleranza definita nella macro *PRECISION*, il valore della variabile *pitch* viene restituito come valore di ritorno, altrimenti viene restituito l'ultimo valore calcolato dalla routine *__compute_pitch*().

[7]la frequenza fondamentale non è riconoscibile o non presente

```
pitch =  __compute_pitch( approx, ↩
   samplecount, sample_format, settings);
ring[i] = pitch;
i= (i+1) % RINGSIZE;

if( pitch != UNPITCHED ){
   for(j=0 ; j<RINGSIZE ; j++){
      precision = abs_val( ring[j] - ↩
         ring[(j+1) % RINGSIZE]);
      if( precision < best_precision ){
         pitch = ( ring[j] + ring[(j+1) % ↩
            RINGSIZE] )/2;
         best_precision = precision;
      }
   }
}

if( best_precision <= PRECISION )
   return pitch;

if( op_mode == SPEED )
   return pitch;
else
   return UNPITCHED;
```

Il meccanismo del *ring buffer* appena descritto consente di affinare i risultati restituiti dall'algoritmo e discriminare valori che hanno subito uno dei classici problemi nell'ambito del pitch tracking, ovvero il dimezzamento della frequenza, migliorando così la precisione della libreria.

Capitolo 5

Test e risultati

In questo capitolo verranno riportati i test effettuati sulla libreria WaveTrack.
Dato che esiste un' altra implementazione dell'algoritmo descritto nel capitolo 4 chiamata dywapitchtrack[9] sviluppata dall'azienda schmittMachine, la quale però differisce per molti aspetti da WaveTrack, essa verrà usata come confronto per i benchmark che verranno presentati in seguito.

I seguenti test sono stati effettuati con la versione 4.9.1 di *GCC* e la compilazione delle due librerie è avvenuta senza includere nessun flag di ottimizzazione. In tutte e due le librerie sono stati usati i seguenti parametri dell'algoritmo:

- numero massimo di applicazioni della FLWT : 6,
- numero di massimi/minimi successivi presi in considerazione per il calcolo delle distanze : 3,
- massima frequenza riconoscibile : 3000 Hz,
- soglia, espressa in percentuale, (in base al massimo/minimo globale) per il riconoscimento di un massimo/minimo locale : 75%.

Inoltre nella libreria dywapitchtrack per correttezza dei test è stato necessario aggiungere il silence filter descritto nella sezione 4.4.2.

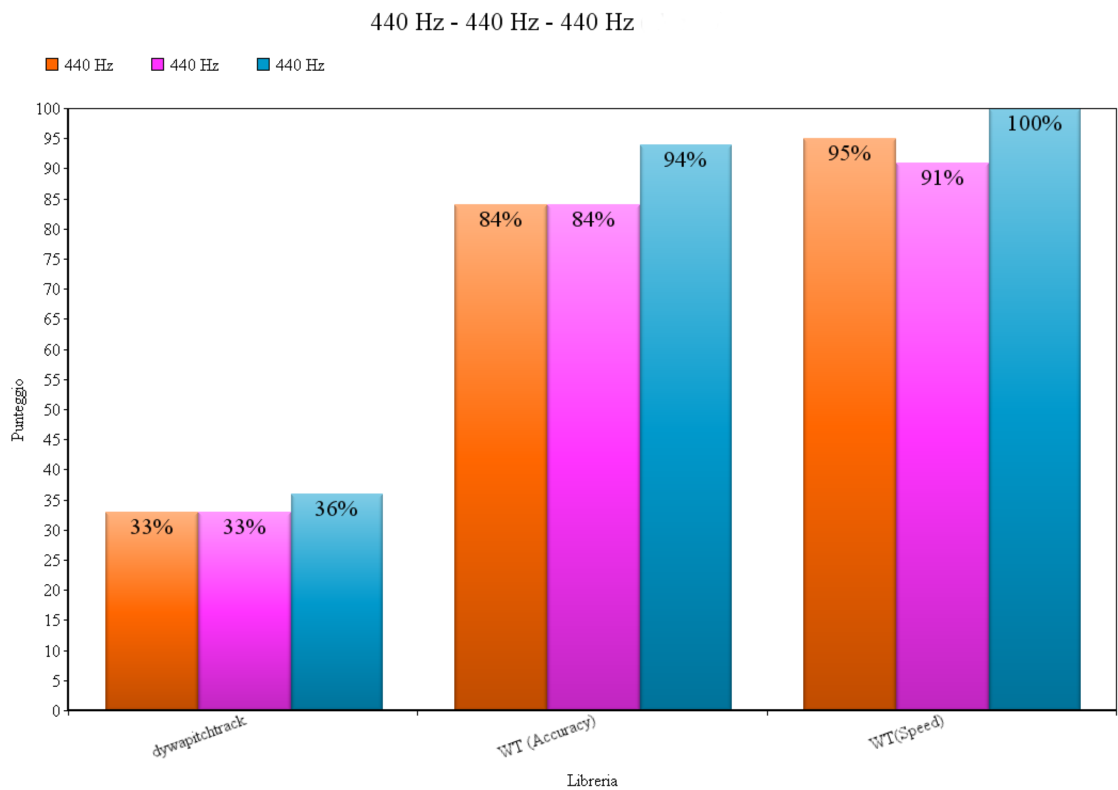
Per l'acquisizione del suono è stato usato il software ALSA su un sistema Arch Linux a 64 bit con kernel Linux versione 3.16.1-1, con i seguenti parametri di acquisizione:

- frequenza di campionamento : 44100 *Hz*,
- formato : *S16*,
- numero di canali : 1,
- latenza : 23.219 ms

la dimensione del buffer usata per i campioni del segnale è di 2048 campioni. Inoltre i test sono stati effettuati su un calcolatore con processore AMD octa core da 5 GHz usando come generatore di suoni un simulatore di pianoforte.

Di seguito viene presentata una serie di grafici dove, il primo di ogni serie indica sull'asse delle x il nome della libreria e sull'asse delle y il relativo punteggio espresso in percentuale, in base a quanti valori restituiti come valore di ritorno sono esatti con una tolleranza di 5 Hz, ed il secondo mostra sull'asse delle y il nome della libreria e sull'asse delle x il tempo medio impiegato per riconoscere ogni nota della sequenza.

Il nome del grafico indica quali frequenze sono state suonate ed in quale ordine, la didascalia mostra invece quali toni sono stati suonati e la relativa durata.



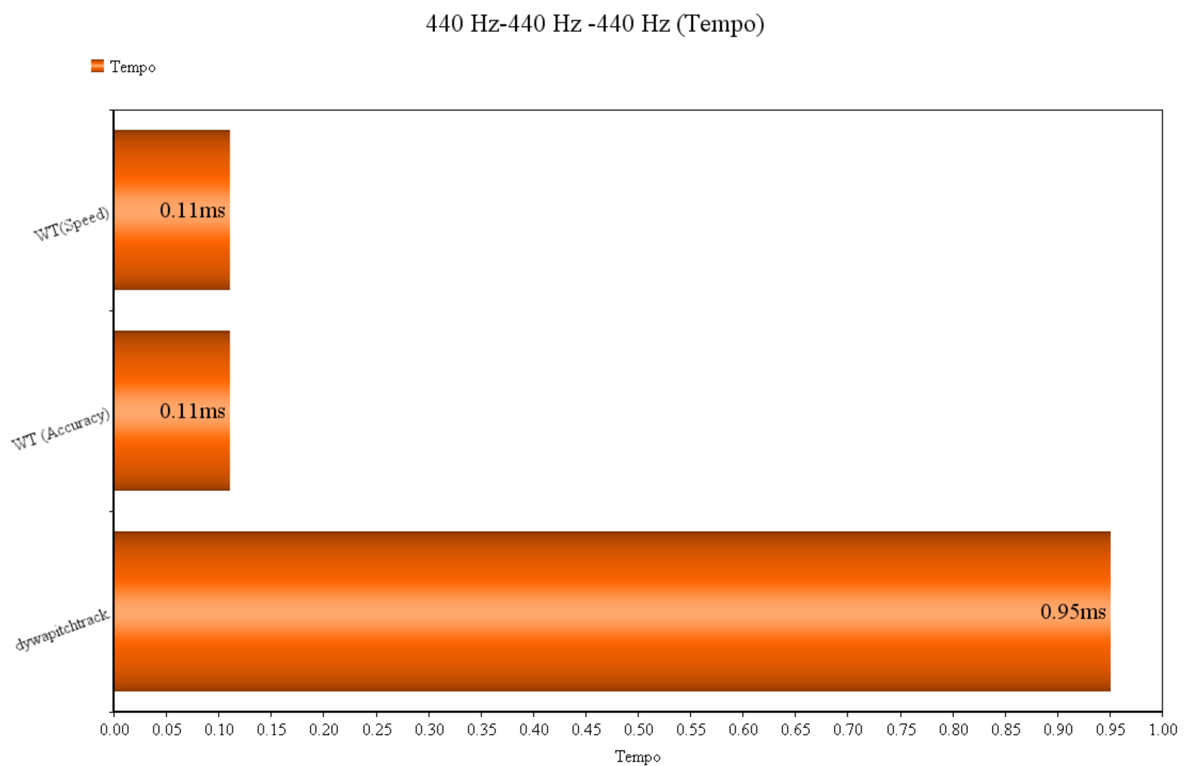
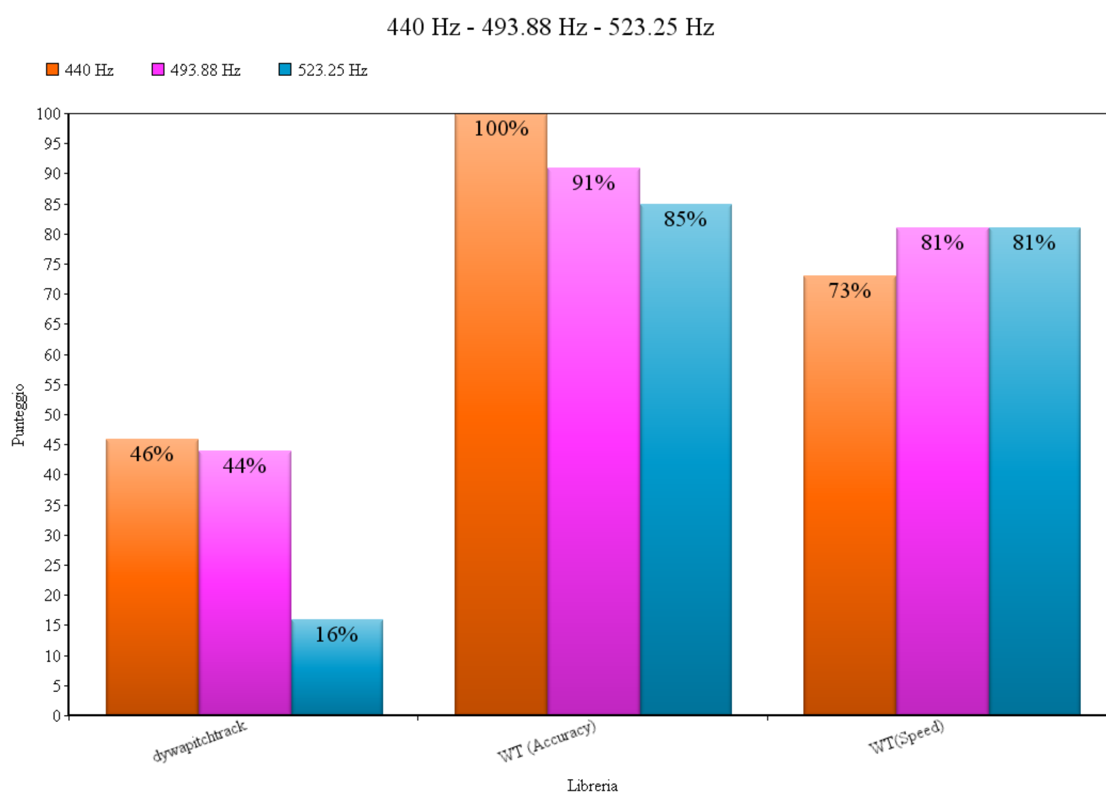


Figura 5.1: A4 (∼16ms) - A4 (∼32ms) - A4 (∼49ms)



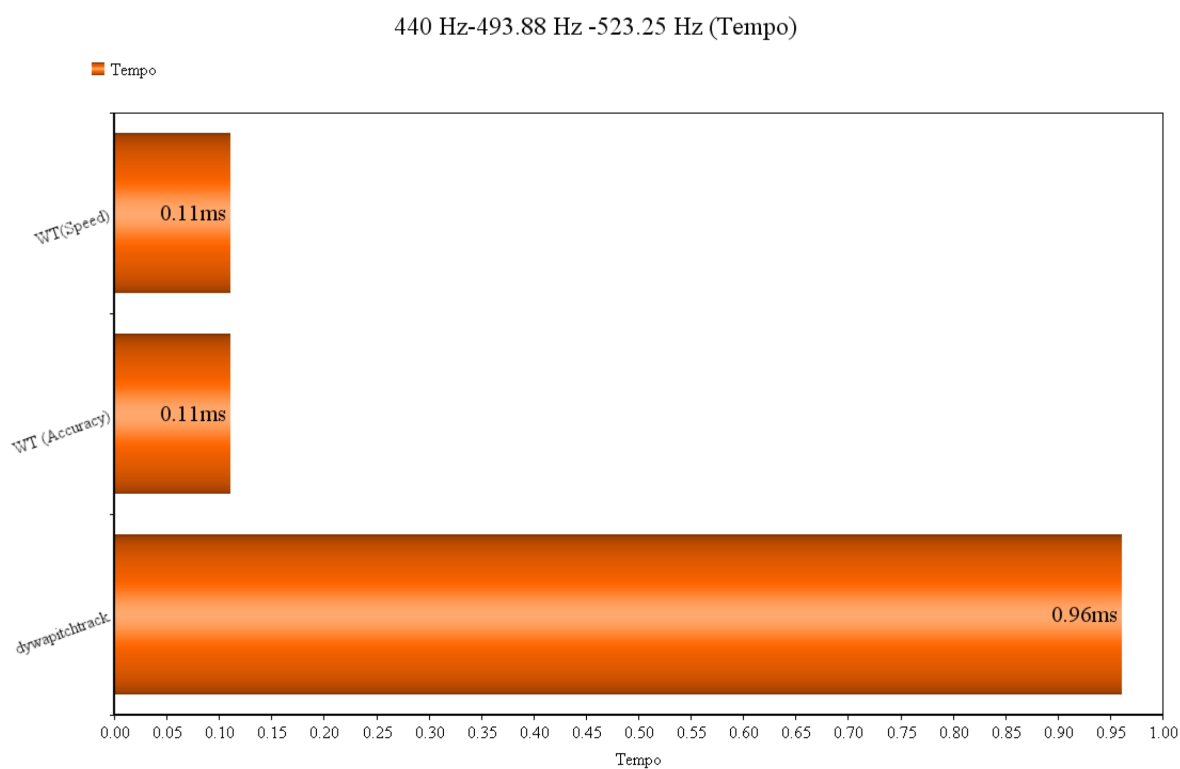
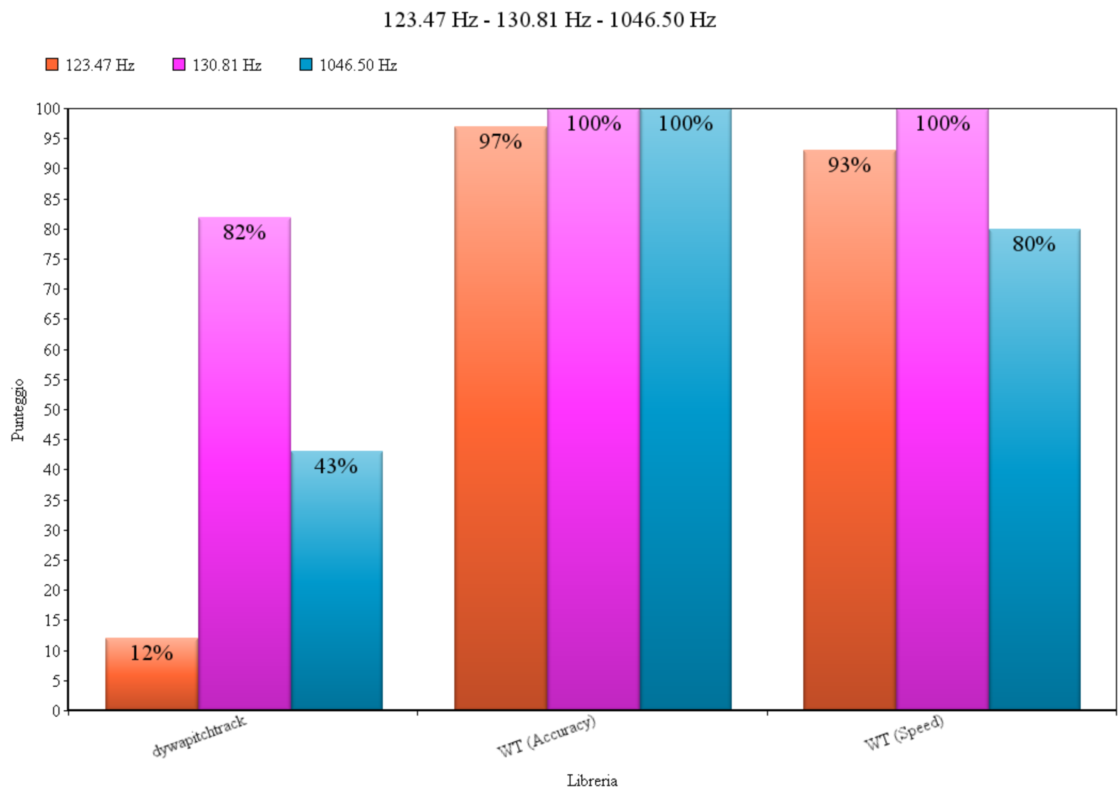


Figura 5.2: A4 (∼18ms) - B4 (∼32ms) - C5 (∼45ms)



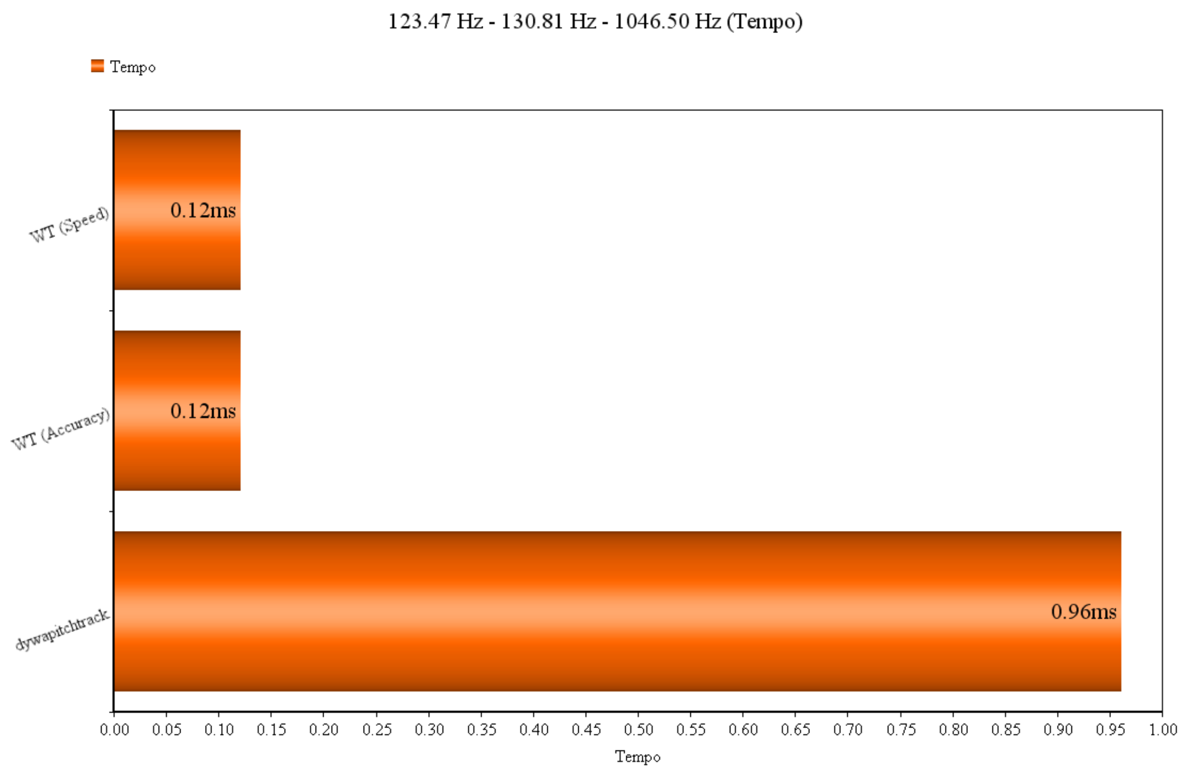
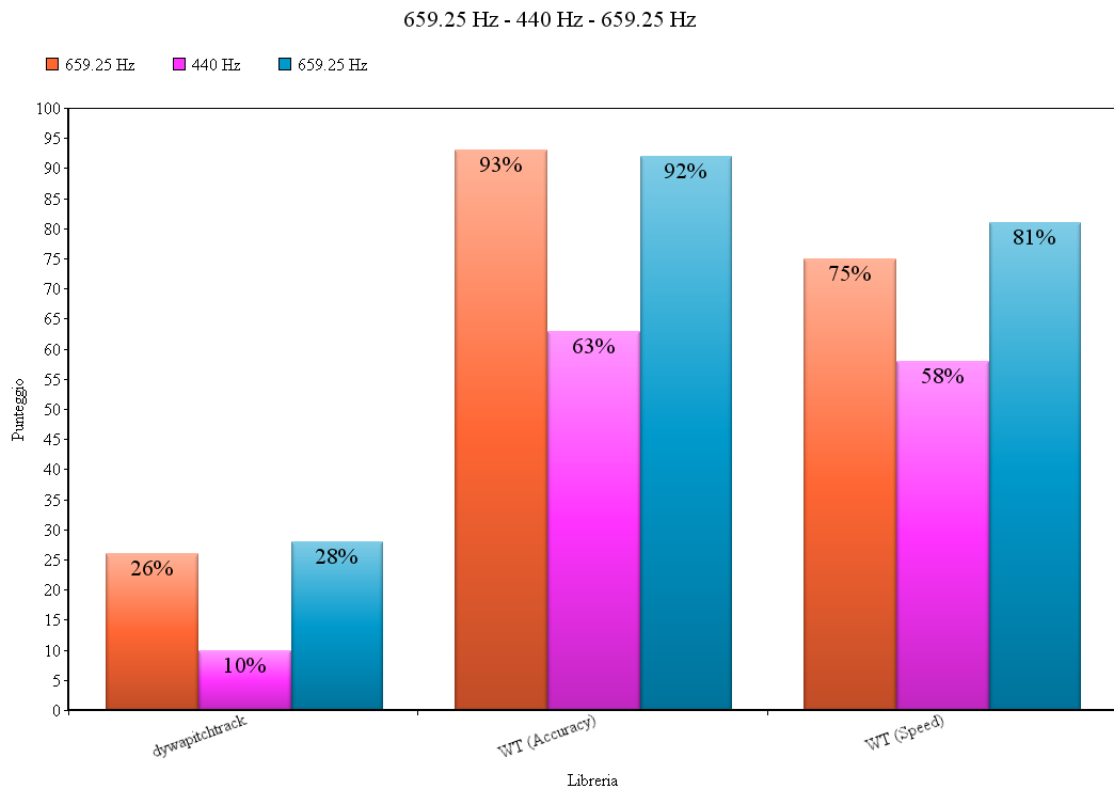


Figura 5.3: B2 (∼71ms) - C3 (∼75ms) - C6 (∼32ms)



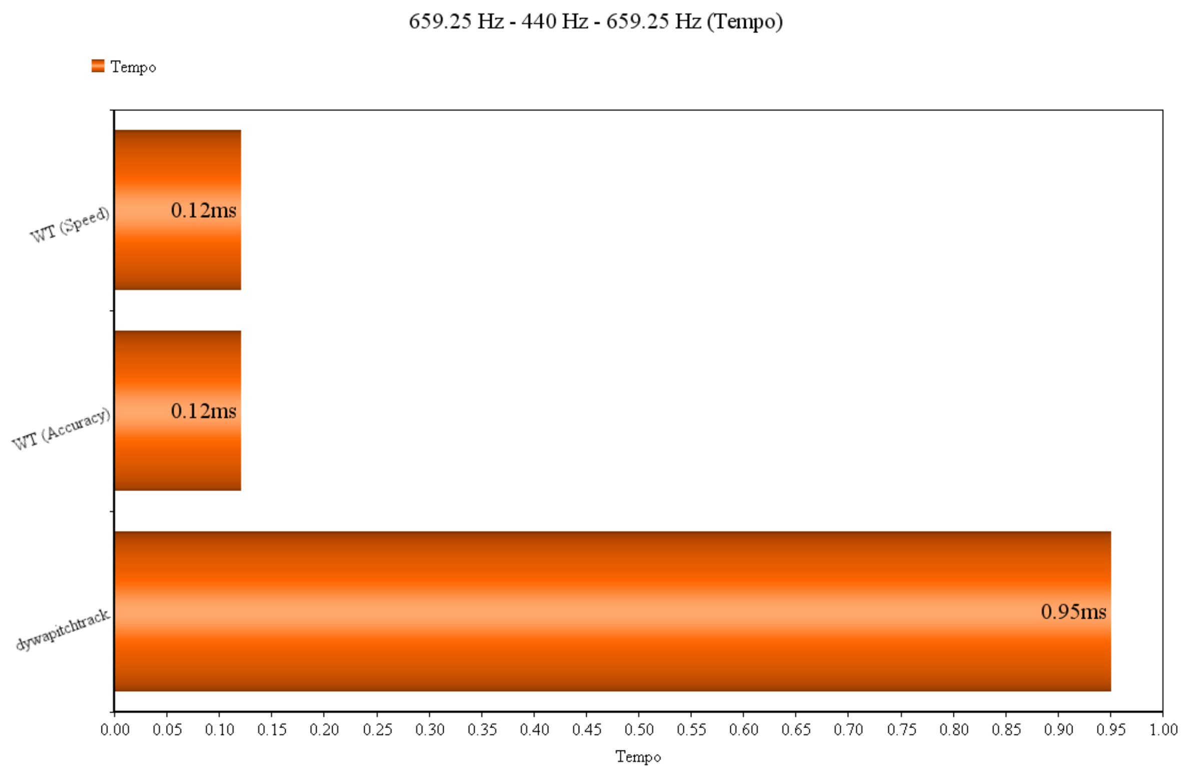
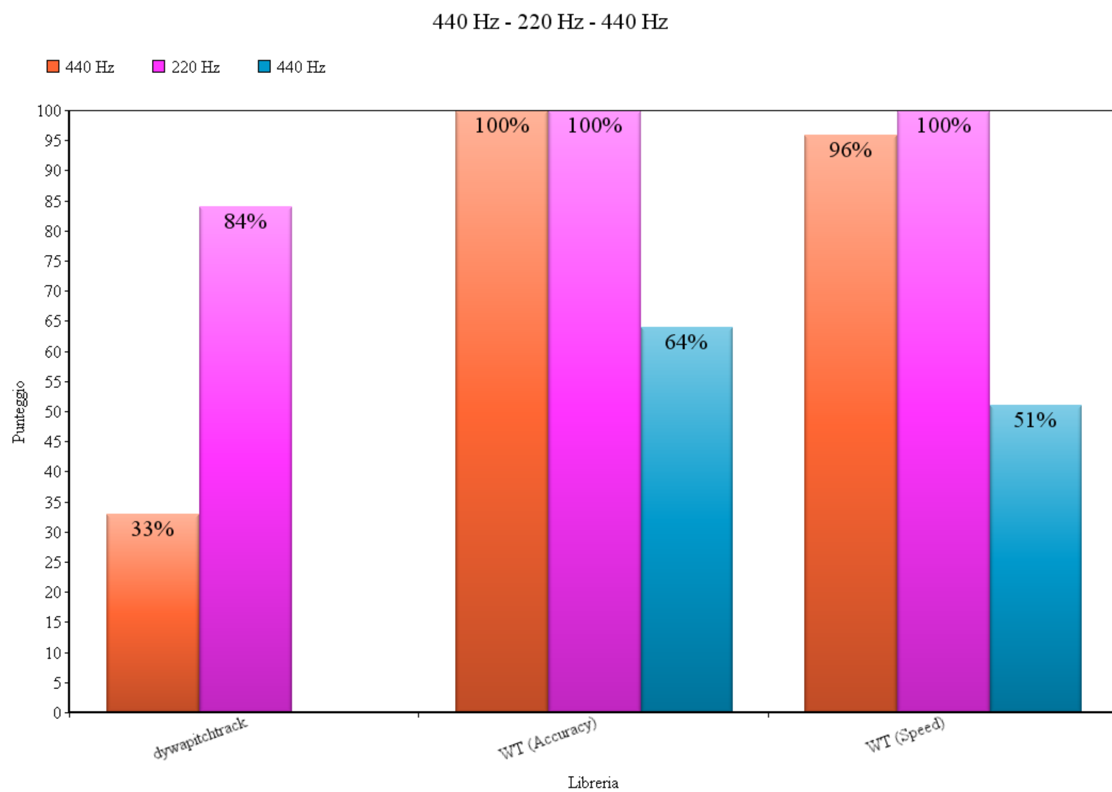


Figura 5.4: E5 (∼31ms) - A4 (∼44ms) - E5 (∼27ms)



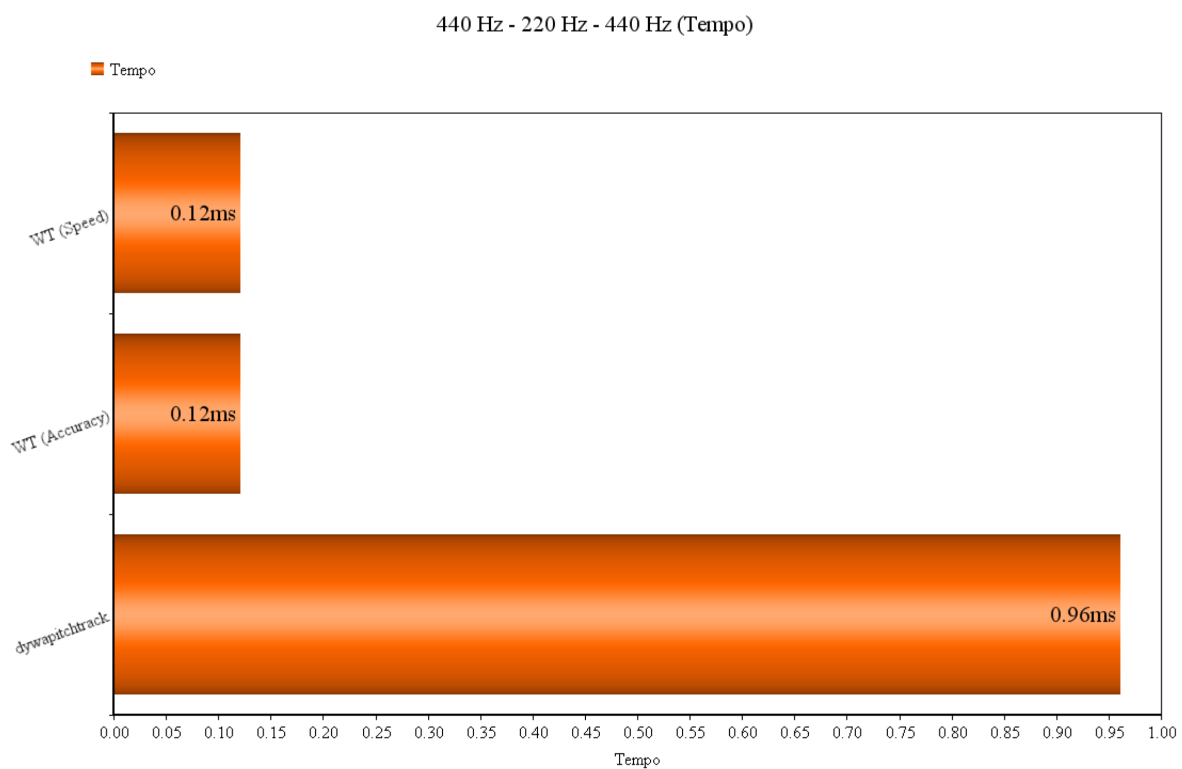


Figura 5.5: A4 (∼25ms) - A3 (∼59ms) - A4 (∼22ms)

Si possono notare alcuni risultati interessanti, innanzitutto WaveTrack ottiene risultati migliori della libreria *dywapitchtrack* in tutti i test effettuati, sia in termini di prestazioni che di precisione, emergono inoltre alcuni difetti di *dywapitchtrack*, infatti come si può notare dal grafico 5.5 il terzo A4 (440 Hz) non viene riconosciuto dalla libreria a causa della sua ottimizzazione per la voce umana.

Nello specifico *dywapitchtrack* effettua una fase di post-processing dove se vi sono dimezzamenti della frequenza improvvisi la libreria scarta il risultato e continua a restituire come valore di ritorno il pitch trovato precedentemente, è evidente come questo meccanismo di ottimizzazione sià dannoso per un uso della libreria che non sia specifico all'analisi della voce umana.

Si può notare inoltre (grafici 5.4, 5.5) come occasionalmente la libreria WaveTrack soffra di problemi relativi al dimezzamento della frequenza, tali problemi sono difficili da gestire a causa della difficoltà nel riconoscere i falsi positivi dai falsi negativi, si è pero riusciti a mitigarlo con il meccanismo del *ring-buffer.*

Sviluppi futuri

L' accuratezza di WaveTrack potrebbe essere migliorata lavorando su un sistema di *ring-buffering* più intelligente o aumentando le dimensioni dei buffer e cambiando il meccanismo usato per la selezione delle frequenze più simili tra loro, implementando magari un metodo più sofisticato del semplice confronto a coppie dei valori contenuti nel buffer.

Vi sono inoltre dei margini di miglioramento per il contenimento del problema dell dimezzamento della frequenza, infatti gli attuali meccanismi seppur funzionanti nella maggior parte dei casi possono fallire quando si analizzano frequenze che appartengono alla banda di frequenze più bassa riconoscibile dall'algoritmo.

Infine sarebbe interessante implementare diverse funzioni Wavelet madre al posto della Wavelet di Haar e studiarne gli effetti sulle prestazioni e l'accuratezza dell'algoritmo.

Conclusioni

Come si può evincere dai test i risultati sono ottimi, anche con segnali che durano pochi millisecondi la libreria riesce a produrre risultati corretti e la latenza generata è bassissima (∼0.11ms in media).

Inoltre dal confronto con la libreria *dywapitchtrack* si può notare come le ottimizzazioni e le funzionalità aggiunte all'algoritmo originale abbiano portato dei miglioramenti sostanziali sia nell'accuratezza dei risultati sia nelle prestazioni generali dell'implementazione.

Durante questo lavoro di tesi si è quindi riusciti a produrre un implementazione ottimizzata dell'algoritmo descritto nel capitolo 3, aggiungendo inoltre funzionalità come il *ring buffer* o il *one-shot period method*, che hanno contribuito ad incrementare precisione e velocità dell'algoritmo.

La libreria WaveTrack, date le sue esigue richieste di risorse computazionali, si presentà come un'ottima scelta su qualsiasi piattaforma per svolgere compiti di pitch-tracking, anche su calcolatori con risorse limitate (ad esempio Raspberry Pi).

Bibliografia

[1] Eric Larson, Ross Maddox, "*Real-Time Time-Domain Pitch Tracking Using Wavelets*", Proceedings of the University of Illinois at Urbana Champaign Research Experience for Undergraduates Program, August 2008.

[2] Lawrence Rabiner, "*On the use of autocorrelation analysis for pitch detection*" IEEE Transactions Acoustics, Speech, And Signal Processing, Vp;. ASSP-25, No. 1, February 1977.

[3] Patricio de la Cuadra, "*PITCH DETECTION METHODS REVIEW*", https://ccrma.stanford.edu/∼pdelac/154/m154paper.htm, 2 Agosto 2014.

[4] Robi Polikar, "*The Wavelet Tutorial*", 1999, http://users.rowan.edu/∼polikar/WAVELETS/WTtutorial.html, 2 Agosto 2014.

[5] Ronald Bracewell, *The Fourier Transform & Its Applications*. 3rd edition, McGraw-Hill, New York, 2000.

[6] Stephane Mallat, *A Wavelet Tour of Signal Processing, Third Edition: The Sparse Way*, Academic Press, 2008.

[7] Ingrid Daubechies , Wim Sweldens, "*Factoring wavelet transforms into lifting steps*", J. Fourier Anal. Appl, 1998.

[8] Li Tan, J. Jiang, "*Fundamentals of Analog and Digital Signal Processing*", AuthorHouse, 2008.

[9] schmittMachine, "*dywapitchtrack*", 2010, http://www.schmittmachine.com/dywapitchtrack.html, 4 Agosto 2014.

Ringraziamenti

Ringrazio i miei genitori, che da sempre mi supportano in ogni mia decisione, credono in me e che fin da piccolo mi hanno costantemente spinto ad interessarmi di qualsiasi cosa stuzzicasse la mia curiosità.

Ringrazio mia sorella, che nonostante mi debba sopportare tutti i giorni non si tira mai indietro a ogni mia richiesta d'aiuto.

Ringrazio Jessica, senza la quale molte virgole di questa tesi sarebbero nei posti sbagliati e che forse non lo sa ma è la persona che ogni giorno mi spinge sempre a puntare più in alto e a fare sempre di più.

Ringrazio il professor. Renzo Davoli per la sua infinita pazienza, per il suo sapere smisurato che in ogni occasione riesce a stimolare la mià curiosità, per avermi fatto appassionare al mondo dei sistemi operativi e per la sua capacità di ispirare le persone con il suo spirito "open source".

Ringrazio infine tutti i miei amici, che sono pronti a sostenermi in ogni momento e senza i quali nulla sarebbe così divertente.

Antonio Cardace.