

Introduzione

Il **segnale audio** è un *segnale analogico* che varia in modo continuo nel tempo. La sua ampiezza può assumere un'infinità di valori passando da un minimo ad un massimo.

Nei dispositivi non è possibile riprodurre tale onda poichè, per loro natura, essi gestiscono **informazioni digitali** mentre il segnale audio è una grandezza continua.

Un segnale audio, per essere rappresentato nei moderni dispositivi deve esser **campionato**.

Il **campionamento** è quel processo che consiste nel prelevare, ad intervalli regolari, il valore del segnale che può presentarsi come segnale elettrico che varia nel tempo.

Il risultato del campionamento è un segnale rappresentato mediante valori numerici discreti che presenteranno dei salti tra un valore all'altro poichè è stato assegnato un singolo valore ad un intervallo di tempo che presentava infiniti valori.

Ovviamente il campionamento produce del **rumore** dovuto al fatto che, approssimando la curva analogica originaria mediante una spezzata, non si riesce ad ottenere una rappresentazione perfetta.

Nella compressione dati le **ridondanze** hanno un ruolo fondamentale: sfruttano le curve analogiche come segnale *cover* per nascondere messaggi segreti.

MP3 (per esteso *Moving Picture Expert Group-1/2 Audio Layer 3*, noto anche come **MPEG-1 Audio Layer III** o **MPEG-2 Audio Layer III**) è un algoritmo di compressione audio di tipo lossy, sviluppato dal gruppo MPEG, in grado di ridurre drasticamente la quantità di dati richiesti per memorizzare un suono, rimanendo comunque una riproduzione accettabilmente fedele del file originale non compresso.

La steganografia in file audio assume un ruolo fondamentale in questa epoca poiché vi è la necessità di nuove tecniche che permettano di rendere segrete le comunicazioni in modo che nessuno (a parte il destinatario legittimo) possa acquisire le informazioni in esse presenti.

La steganografia si interpone tra:

- **Crittografia:** insieme di tecniche che permettono di rendere segreti i messaggi mediante la loro codifica. La codifica avviene utilizzando degli algoritmi di cifratura che rendono il messaggio incomprensibile a chi non è a conoscenza dei relativi sistemi di decodifica;
- **Watermarking:** tecnologia che permette di inserire delle informazioni in un segnale. Tali informazioni di solito fanno riferimento all'originalità del file, al titolare o ad altre informazioni relative ai diritti di proprietà.

Le tecniche di steganografia permettono di nascondere il messaggio inserendolo nel mezzo stesso e garantendo la segretezza della comunicazione.

Nel nostro progetto abbiamo voluto studiare e implementare delle tecniche di steganografia per nascondere dei messaggi in file audio. Nello specifico abbiamo studiato la fattibilità di utilizzare tali tecniche in segnali audio compressi tramite encoder MP3.

Le tecniche implementate sono state:

- **echo data hiding:** che consiste nel nascondere il messaggio sfruttando l'aggiunta di eco al file audio;
- **amplitude data hiding:** che consiste nel nascondere il messaggio modificando l'ampiezza del segnale audio.

Contents

1	Introduzione alla compressione dati	5
1.1	Sorgente	7
1.2	Entropia e teorema di Shannon	7
1.2.1	Teorema fondamentale della codifica sorgente	8
2	Compressione audio & steganografia audio	9
2.1	Background	9
2.2	PCM	11
2.3	Lo standard MPEG-1 Layer III	11
2.4	Riduzione di un fattore 12	12
2.5	Bitrate	12
2.6	Frequenza di campionamento	13
2.7	Channel Modes	13
2.7.1	Joint Stereo	14
2.8	Steganografia	14
3	MP3	16
3.1	Anatomia di un file	16
3.2	The Frame Layout	16
3.2.1	Header	16
3.2.2	Sync	17
3.2.3	ID	17
3.2.4	Layer	18
3.2.5	Bitrate	18
3.2.6	Frequency	19
3.2.7	Padding	19
3.2.8	Mode	19
3.2.9	Mode extension	19
3.2.10	Copyright bit	20
3.2.11	Home	20

CONTENTS	3
3.2.12 Emphasis	20
3.2.13 Private	20
3.3 CRC	20
3.4 Side information	20
3.4.1 main_data_begin	21
3.4.2 scfsi (4 bits, 8 bits)	21
3.4.3 side_info_gr0(1)	21
3.4.4 table_select(10-15 bits, 15-30 bits)	24
3.4.5 subblock_gain(9 bits, 18 bits)	24
3.4.6 region0_count (4 bits, 8 bit), region1_count (3 bits, 6 bits)	24
3.4.7 preflag (1 bit, 2 bits)	24
3.4.8 scalfac_scale(1 bit, 2 bits)	24
3.4.9 Main Data	25
3.4.10 scalefactors	25
3.4.11 ID3	26
3.5 Codifica	27
3.5.1 Analysis Polyphase Filterbank	27
3.5.2 Modified discrete cosine transform (MDCT)	28
3.5.3 Windowing	28
3.5.4 FFT	29
3.5.5 Psychoacoustic Model	29
3.5.6 Nonuniform Quantization	30
3.5.7 Codifica di Huffman	32
3.5.8 Codifica di informazioni secondarie	32
3.5.9 Bitstream Formatting CRC word generation	33
3.6 Decodifica	34
3.6.1 Sync and Error Checking	35
3.6.2 Huffman Decoding e Huffman info decoding	35
3.6.3 Scalefactor decoding	35
3.6.4 Requantizer	35
3.6.5 Reordering	36
3.6.6 Stereo Decoding	36
3.6.7 Alias Reduction	36
3.6.8 Inverse Modified Discrete Cosine Transform (IMDCT) .	37
3.6.9 Frequency Inversion	37
3.6.10 Synthesis Polyphase Filterbank	37
4 Data Audio Hiding: Tools	38
4.1 Overview	38
4.2 DeepSound	38

CONTENTS	4
4.2.1 Overview	38
4.2.2 Interfaccia - Barra dei Menù	39
4.2.3 Interfaccia - Carrier Files	40
4.2.4 Interfaccia - Secret Files	41
4.2.5 Interfaccia - Qualità	42
4.2.6 Conversione	43
4.2.7 Svantaggi	43
4.3 MP3Stego	44
4.3.1 Overview	44
4.3.2 Tool nel dettaglio	44
4.3.3 Codifica	45
4.3.4 Decodifica	45
4.4 Steghide	46
4.4.1 Overview	46
4.4.2 Componenti	46
4.4.3 Algoritmo di Embedding	47
4.5 QuickStego	47
4.5.1 Overview	47
4.6 AudioStego	48
4.6.1 Overview	48
4.6.2 Algoritmo	48
5 Implementazione	50
5.1 Echo hiding in Python	50
5.1.1 Dipendenze	51
5.1.2 Codificatore	51
5.1.3 Decodificatore	54
5.2 Amplitude data hiding in Lame	55
5.2.1 LAME	55
5.2.2 Risultati	56
5.3 Amplitude data hiding in Python	56
5.3.1 Dipendenze	56
5.3.2 Codificatore	56
5.3.3 Decodificatore	59
6 Conclusioni	61

Chapter 1

Introduzione alla compressione dati

La **compressione dati** è l'insieme di tecniche che permettono la *codifica* di una *sequenza di dati digitali*, denotata con la lettera D , in una *sequenza di dati più corta*, denotata con $\Delta(D)$.

Dopo aver compresso una sequenza però, deve esser sempre possibile effettuare l'operazione inversa, ovvero ottenere D o un'ottima approssimazione di D partendo dalla sequenza $\Delta(D)$.

Un concetto fondamentale è la **ridondanza**; la **compressione** di una sequenza di dati è possibile solo se nella sequenza è presente **ridondanza**; ad esempio per le *sequenze di dati casuali* non esiste alcuna tecnica di compressione poichè esse non presentano ridondanza.

I **campi di applicazione** della compressione dati sono :

1. **data storage**: al fine di memorizzare quanti più dati possibili, essi sono dapprima compressi e poi memorizzati;
2. **data communication**: poichè i canali di comunicazione hanno *capacità limitate*, i dati sono compressi e poi trasmessi.

Le *tecniche* di compressione dati sono suddivise in **due categorie**:

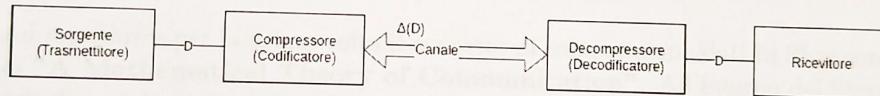
1. **Lossless**: a partire dalla sequenza $\Delta(D)$ si ottiene esattamente la sequenza originale D ;

2. **lossy**: a partire dalla sequenza $\Delta(D)$ si ottiene un' **approssimazione accettabile** di D .

Per la scelta della tecnica di compressione dati da utilizzare è buona norma ottenere un **trade-off** tra la *compressione* stessa e i **vantaggi** che si vogliono ottenere dalla compressione, a seconda delle risorse hardware che si andranno ad utilizzare; ad esempio se abbiamo a disposizione un grande spazio di archiviazione è inutile utilizzare una tecnica di compressione costosa (in termini di tempo) che è in grado di comprimere molto la sequenza di dati.

Un **sistema di compressione dati** può essere scomposto in 5 elementi:

1. **Sorgente** anche detta **trasmettitore**: sistema che prende in input la sequenza di dati da comprimere e/o trasmettere;
2. **compressore** anche detto **codificatore**: sistema che prende in input la sequenza di dati D e la comprime ottenendo la sequenza $\Delta(D)$;
3. **canale**: mezzo fisico attraverso il quale la sequenza di dati è trasmessa dalla sorgente al ricevitore;
4. **decompressore** anche detto **decodificatore**: sistema che prende in input la sequenza di dati $\Delta(D)$, la decomprime ottenendo o D o una buona approssimazione di D ;
5. **ricevitore**: sistema che riceve il segnale.



Le tecniche di compressione spesso sono suddivise anche in:

- **tecniche online**: tecniche in cui la **sequenza di dati da comprimere** è elaborata in *real-time* cioè non conoscono a priori la sequenza di dati che andranno ad elaborare;
- **tecniche offline**: tecniche in cui la sequenza di dati da comprimere è conosciuta a priori (*molto più efficienti*).

1.1 Sorgente

Prima di dare la definizione di sorgente, vi è il bisogno di definire i seguenti concetti:

- **alfabeto**: insieme finito di simboli che contiene almeno un simbolo;
- **carattere**: simbolo di un alfabeto;
- **stringa**: sequenza di caratteri di un alfabeto;
- **sorgente**: processo che genera una sequenza di caratteri di un alfabeto.

Dato un **alfabeto** $\Sigma = s_1, s_2, \dots, s_k$ diremo che una **sorgente** è di **primo ordine** se le **probabilità** dei caratteri di Σ sommano ad 1.

Vi sono due casi particolari di sorgenti di primo ordine per cui le tecniche di compressione dati sono inutili e sono:

- **sorgenti costanti**: sorgenti che trasmettono **informazione nulla**; sono quelle sorgenti in cui la *probabilità* che il carattere i sia trasmesso è uguale a $p_i = 1$, ciò implica che il carattere trasmesso sarà sempre i e per tale motivo è inutile comprimere poiché anche il ricevitore saprà che ogni carattere che riceve è i ;
- **sorgenti casuali**: sorgenti per le quali i **caratteri sono equiprobabili**; per esse la compressione è inutile poiché non è presente ridondanza.

1.2 Entropia e teorema di Shannon

Ad introdurre per la prima volta il concetto di compressione dati fu Shannon in "A Mathematical Theory of Communication". All'interno del libro egli dimostrò che esiste un limite teorico per la compressione di dati senza avere alcuna perdita di informazione, ovvero il segnale originale ed il segnale compresso contengono le stesse informazioni.

Questo limite prende il nome di **entropy rate** e dipende dalla probabilità che una data sequenza di caratteri sia presente all'interno dei dati.

L'enunciato del teorema di Shannon è il seguente:

$k \geq 1$, sia S una sorgente del primo ordine che genera caratteri dell'alfabeto $\Sigma = s_1, s_2, \dots, s_k$ con **probabilità indipendenti** p_1, p_2, \dots, p_k .

L'**entropia** è data dalla seguente formula:

$$H(S) = \sum_{i=1}^k p_i * \log_2(p_i)$$

Informalmente l'entropia è l'informazione media contenuta nella sequenza di dati generata da una sorgente.

Ogni compressione possibile, di tipo lossless, si può avvicinare il più possibile all'entropy rate ma non può mai superarlo.

Diversamente accade per le compressioni di tipo *lossy*. Infatti in questi tipi di compressione è concessa la perdita di informazioni poiché le informazioni perse non sono percepibili dalla platea di utilizzo, quindi il limite teorico può essere ampiamente superato. Questo tipo di compressione è ampiamente utilizzato in molteplici campi quali la compressione di immagini, video ed audio.

1.2.1 Teorema fondamentale della codifica sorgente

L'**entropia** ci permette di determinare quando si ottiene un *compressore ottimale*, basta considerare il seguente teorema:

Sia S una sorgente del primo ordine su un alfabeto $\Sigma = s_1, s_2, \dots, s_k$ e sia $\Gamma = 0, 1$.

Codificare S con caratteri di Γ richiede in media $H(S)$ caratteri di Γ per ogni carattere di Σ .

Bisogna tener presente che i dati che sono stati compressi da un algoritmo di compressione ottimale *non possono esser compressi ulteriormente* poiché non vi è più ridondanza.

Chapter 2

Compressione audio & steganografia audio

I segnali audio digitali non compressi consumano una grande quantità di dati e non sono adatti alla trasmissione e al salvataggio.

L'**ISO** e il **Moving Pictures Expert Group (MPEG)** hanno sviluppato uno standard che contiene tecniche per la compressione di segnali audio e video.

La parte audio dello standard comprende 3 modalità con performance e complessità crescenti.

La terza modalità, detta anche **Layer III**, è in grado di comprimere CD audio da 1.4 Mb/s a 128 Kb/s senza causare una degradazione percepibile. Attualmente è lo standard di compressione audio più utilizzato.

2.1 Background

La *psicoacustica* è la scienza che si occupa di studiare come il cervello e le orecchie interagiscono quando l'apparato uditivo viene raggiunto da segnali audio.

Il nostro corpo infatti è colpito da radiazioni appartenenti ad uno spettro di frequenze molto ampio, ma è in grado di percepire solo alcuni (luce ed audio ne sono l'esempio). In particolare per il segnale audio, il nostro corpo è in grado di percepire solo radiazioni che variano da un minimo di 20 Hz ad un massimo di 20 KHz. La sensibilità massima si ottiene nella fascia compresa tra 2 KHz e 4 KHz. Muovendoci verso l'estremo delle fasce è più difficile per il corpo umano percepire i rumori provenienti da quelle frequenze ed è

CHAPTER 2. COMPRESIONE AUDIO & STEGANOGRAFIA AUDIO10

necessario aumentare il volume di ascolto per renderli percepibili. Inoltre con l'avanzare dell'età la sensibilità a determinate frequenze risulta diminuire ed è più difficile percepire i suoni.

Fondamentalmente il nostro cervello si come un potente filtro naturale, infatti esso non è in grado di elaborare informazioni per tutti i sensi in un dato tempo. Allo stesso modo è possibile pensare ai codec audio come dei potenti filtri in grado di tagliare via tutte le informazioni che il nostro apparato non sarebbe in grado di riconoscere, e che quindi occuperebbero spazio inutile sui supporti di memorizzazione.

Sperimentalmente si è mostrato che il nostro orecchio lavora con bande di 24 frequenze. Le frequenze presenti in queste, così chiamate, *critical bands* sono difficili da essere distinte. E' possibile inoltre avere che alcune frequenze le mascherino rendendo il suono inudibile. I mascheramenti possono essere di due tipi:

- Simultaneous;
- Temporal.

Simultaneous masking Ipotizziamo di avere un segnale tonale dominante nel segnale. Il rumore dominante all'interno del segnale introdurrà una soglia di mascheramento che taglierà fuori le frequenze che andranno oltre questa soglia.

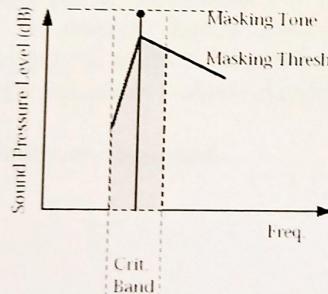


Figure 2.1: Esempio di simultaneous masking.

Temporal masking Questa tipologia di mascheramento avviene nel dominio del tempo. Se una componente tonale più forte (*masker*) ed una più

debole (*maskee*) si sovrappongono per un lasso di tempo quest'ultima risultà offuscata dalla prima.

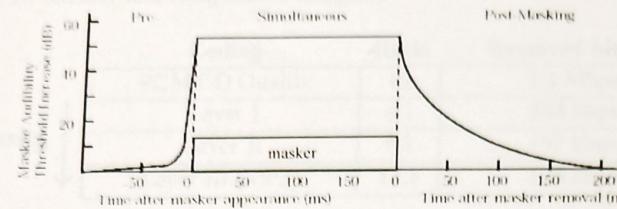


Figure 2.2: Esempio di temporal masking.

2.2 PCM

PCM, Pulse Code modulation, è la modulazione utilizzata per salvare e trasmettere audio digitali non compressi. Per PCM è necessario tenere in considerazione due variabili: **sample rate (Hz)** e **bit rate (bit)**. Un sample rate più alto significa portare più frequenze, mentre un bit rate più alto significa avere una qualità audio migliore, ma ovviamente più peso.

2.3 Lo standard MPEG-1 Layer III

L'ISO e MPEG hanno collaborato per la creazione di uno standard di compressione/decompressione per la trasmissione e lo storage di flussi video e i rispettivi flussi audio. Lo standard doveva essere generico, ovvero ogni decoder utilizzato doveva essere in grado di decodificare ogni bitstream generato da un qualsiasi coder che utilizzasse lo stesso standard.

Lo standard nato era composto da 3 parti:

- Una parte audio;
- Una parte video;
- Una parte di sistema.

La parte di sistema è quella che si occupa della descrizione di come i flussi di audio e video possono essere trasferiti contemporaneamente sullo stesso canale di distribuzione. Utilizzando questo standard è possibile far viaggiare flussi audio e video al bitrate di 1-2 Mbit/s.

La parte audio è composta da 3 layer ognuno con un rapporto di compressione, quindi un bitrate, una complessità maggiore.

	Coding	Ratio	Required bitrate
Complexity ↓	PCM CD Quality	1:1	1.4 Mbps
	Layer I	4:1	384 kbps
	Layer II	8:1	192 kbps
	Layer III (MP3)	12:1	128 kbps

Figure 2.3: MPEG-1 Audio part.

Il layer III comprime il file audio PCM originale con un fattore di compressione di 12. Successivamente con l'avvento di MPEG-2 è stato introdotto:

- Multichannel audio encoding, incluso la configurazione per 5.1;
- Codifica a *sample frequencies* più basse.

2.4 Riduzione di un fattore 12

MP3 è un *perceptual codec* e si avvantaggia sulle "debolezze" dell'essere umano. Infatti, esso è un processo di compressione lossy, quindi con perdita di informazioni in modo irrecuperabile. La perdita di queste informazioni però è accettabile in quanto si va a filtrare audio che comunque non sarebbe percettibile all'apparato uditivo umano.

Ogni essere umano ha una banda critica approssimata da una *scalefactor band*. Per ogni scalefactor band è calcolata una *soglia di mascheramento*. In base a questa soglia le bande sono scalate con uno scalefactor adeguato con lo scopo di ridurre il rumore di quantizzazione causato da una successiva quantizzazione delle linee di frequenza per ciascuna banda.

Per migliorare ancor più il fattore di compressione viene applicato l'Huffman Coding.

2.5 Bitrate

E' un'opzione da impostare prima di effettuare la codifica. Esso rappresenta la quantità di dati necessaria per ogni secondo di riproduzione. La qualità

dell'audio riprodotto è strettamente correlata al bitrate. Il Layer III supporta bitrate da **8 kbit/s** a **320 kbit/s**; tipicamente il bitrate utilizzato è pari a **128 kbit/s**.

Lo standard inoltre prevede la possibilità di utilizzare due tipologie diverse di bitrates:

- *Constant Bitrate (CB)*;
- *Variable Bitrate (VB)*.

Constant Bitrate È l'opzione utilizzata di default. Essa prevede l'utilizzo dello stesso bitrate per tutta la codifica del file audio. Alcune tracce audio però sono più complesse e in alcuni tratti potrebbero aver necessità di più informazioni a causa dell'utilizzo di più strumenti o di effetti particolari. In ambiti come quello dello streaming è necessario avere un bitrate costante.

Variable Bitrate È l'opzione che risulta essere la soluzione ai problemi che si possono avere durante la codifica di tracce complesse. Essa prevede la possibilità di utilizzare bitrate variabili all'interno del file audio, a seconda della necessità. Questa tecnica consente di specificare il bitrate massimo che si desidera raggiungere e sarà poi compito del coder adattare il bitrate alle caratteristiche del segnale. Questo però porta ad avere alcune problematiche nel timing della traccia: i player potranno non essere in grado di mostrare informazioni corrette sul timing e, nel caso peggiore, potrebbero non riuscire a mostrarle del tutto.

2.6 Frequenza di campionamento

La risoluzione dell'audio è fortemente dipendente dalla frequenza di campionamento del segnale audio, ovvero la quantità di volte che il segnale è registrato ogni secondo. Ovviamente una frequenza di campionamento più alta unita ad un bitrate alto offre una qualità migliore nel file audio prodotto. MPEG-1 definisce la compressione audio a 32kHz, 44.1 kHz e 48 kHz.

2.7 Channel Modes

Vi sono quattro diverse channel mode:

- Single Channel;

CHAPTER 2. COMPRESSIONE AUDIO & STEGANOGRADIA AUDIO14

- Dual Channel (2 Single Channel);
- Stereo;
- Joint Stereo.

2.7.1 Joint Stereo

La *Joint Stereo Mode* sfrutta la ridondanza tra i canali *left* e *right* per ottimizzare la codifica. Esistono due tecniche per questo tipo di approccio:

- Middle/Side stereo (MS stereo);
- Intensity Stereo.

MS stereo è utile quando i canali sono fortemente correlati. I canali *left* e *right* sono sommati e sottratti e trasmessi come la somma e la differenza tra i due canali. Dato che i canali tipicamente sono uguali per la maggior parte del tempo la somma dei segnali porta con sé molte più informazioni rispetto alla differenza. Ciò permette la trasmissione più efficientemente rispetto al trattare separatamente i due canali. Inoltre *MS stereo* è una codifica lossless.

Diversamente, nell' *Intensity Stereo Mode* la sottobande di frequenza più alta sono codificate in un unico segnale sommato con le corrispondenti posizioni di intensità per uno scalefactor delle bande codificate. Dato che solo un canale risulta essere trasmesso, le informazioni stereo del segnale sono contenute all'interno dell' *intensity position*. Ciò però può portare ad alcune inconsistenze dato che l'audio viene "strozzato" in un solo canale, anche se questa inconsistenza non viene facilmente udita dall'apparato umano.

2.8 Steganografia

Le tecniche di steganografia nei file audio sono strettamente legate a come avviene la codifica dei file audio.

Esistono differenti tecniche:

- **codifica dei bit bassi:** tale procedura sostituisce i bit di informazione meno significativi di ogni punto campione del segnale con una stringa binaria codificata. Può però introdurre un fastidioso rumore di fondo. Il maggiore inconveniente di tale metodo è la scarsa resistenza ad eventuali manipolazioni, quali l'introduzione di segnali di disturbo nel canale di trasmissione o la ricampionatura del segnale, a meno dell'utilizzo di tecniche di ridondanza;

- **codifica delle fasi:** tale procedura sostituisce la fase di un segmento iniziale del file audio con una fase di riferimento che rappresenta i dati;
- **spread spectrum:** tale procedura diffondono i dati codificati attraverso lo spettro delle frequenze, rendendo difficile scovare le informazioni, a meno di non conoscere la chiave pseudocasuale. Il ricevente dovrà conoscere oltre alla chiave anche i punti di inizio e di fine del messaggio nascosto nel file audio. Lo spread spectrum introduce un rumore di fondo casuale al suono al di sotto della soglia di percezione; inoltre, opportuni codici di correzione dell'errore garantiscono l'integrità dei dati. La quantità di informazioni che si possono celare mediante questo metodo è di circa 4bps;
- **echo hiding:** tale procedura inserisce le informazioni in un segnale ospite introducendo un'eco. I dati vengono celati modificando tre parametri dell'eco: l'ampiezza iniziale, il tasso di indebolimento del suono ed il ritardo. Mentre il ritardo tra il suono originale e l'eco diminuisce, i due segnali si mescolano fino a che l'orecchio umano non può più distinguerli. Usando due diversi ritardi possiamo codificare le cifre 1 o 0. Per codificare più di un bit, il segnale originale viene diviso in parti, su ognuna delle quali viene introdotta un'eco.

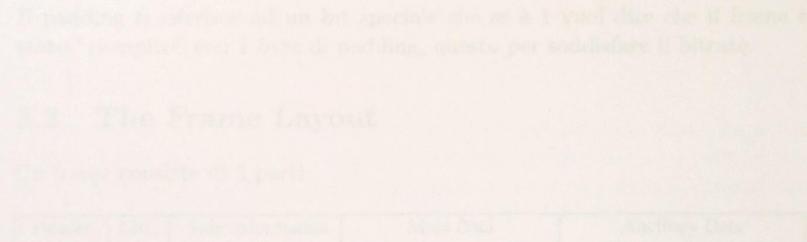


Figure 2.3: Audacity Frame Layout.

Chapter 3

MP3

3.1 Anatomia di un file

Ogni file mp3 è diviso in frammenti più piccoli detti *frames*, ognuno dei quali contiene 1152 audio sample per un totale di *26 ms*, quindi un framerate di circa *38 fps*. Il frame è a sua volta diviso in *granules*, contenente 576 audio samples. La grandezza, in termini di bit, del sample è determinata dal bitrate. La taglia è anche determinata dalla frequenza di campionamento:

$$\frac{144 \times \text{bitrate}}{\text{sample frequency}} + \text{Padding}$$

Il padding si riferisce ad un bit speciale che se è 1 vuol dire che il frame è stato "riempito" con 1 byte di padding, questo per soddisfare il bitrate.

3.2 The Frame Layout

Un frame consiste di 5 parti:

Header	CRC	Side Information	Main Data	Ancillary Data
--------	-----	------------------	-----------	----------------

Figure 3.1: mp3 Frame Layout.

3.2.1 Header

L'header è lungo 32 bit e contiene una word di sincronizzazione ed alcune caratteristiche del frame.

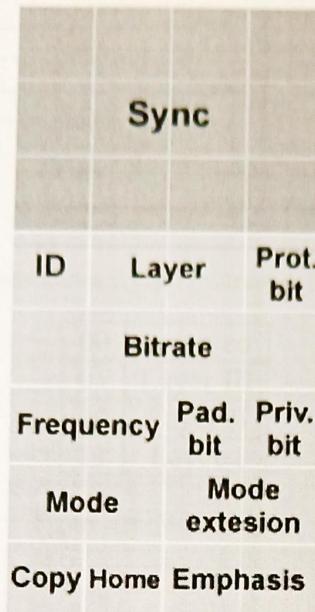


Figure 3.2: mp3 Frame Header.

3.2.2 Sync

La word di sincronizzazione è posta all'inizio di ogni frame in modo da poter sempre agganciare il segnale in ogni punto dello stream. L'unico problema è che anche in altre parti del frame è possibile trovare la parola utilizzata per la sincronizzazione.

3.2.3 ID

Specifica la versione di MPEG utilizzata. Se il bit è uno è utilizzato MPEG-1, MPEG-2 altrimenti.

Alcuni add-on utilizzano solamente 11 bits per la word di sincronizzazione e 2 per l'ID. In tal caso l'ID viene letto seguendo la tabella seguente:

00	MPEG-2.5 (Later extension of MPEG-2)
01	Reserved
10	MPEG-2
11	MPEG-1

Figure 3.3: ID values.

3.2.4 Layer

E' un campo di 2 bits che indica il layer utilizzato:

00	reserved
01	Layer III
10	Layer II
11	Layer I

Figure 3.4: MPEG layer.

3.2.5 Bitrate

E' un campo di 4 bits che indica al decoder con quanti bit è codificato il frame. Questo valore sarà o stesso per tutti i frame se si utilizza CBR.

bits	MPEG-1, layer I	MPEG-1, layer II	MPEG-1, layer III	MPEG-2, layer I	MPEG-2, layer II	MPEG-2, layer III
0 0 0 0						
0 0 0 1 32	32	32	32	32	8	
0 0 1 0 64	48	40	64	48	16	
0 0 1 1 96	56	48	96	56	24	
0 1 0 0 128	64	56	128	64	32	
0 1 0 1 160	80	64	160	80	64	
0 1 1 0 192	96	80	192	96	80	
0 1 1 1 224	112	96	224	112	56	
1 0 0 0 256	128	112	256	128	64	
1 0 0 1 288	160	128	288	160	128	
1 0 1 0 320	192	160	320	192	160	
1 0 1 1 352	224	192	352	224	112	
1 1 0 0 384	256	224	384	256	128	
1 1 0 1 416	320	256	416	320	256	
1 1 1 0 448	384	320	448	384	320	
1 1 1 1						

Figure 3.5: Bitrate definitions.

3.2.6 Frequency

E' un campo di 2 bits che indica la frequenza di campionamento:

Bits	MPEG1	MPEG2	MPEG2.5
00	44100 Hz	22050 Hz	11025 Hz
01	48000 Hz	24000 Hz	12000 Hz
10	32000 Hz	16000 Hz	8000 Hz
11	reserv.	reserv.	reserv.

Figure 3.6: Samplings frequencies.

3.2.7 Padding

E' il padding bit sopra citato.

3.2.8 Mode

Sono 2 bits che rappresentano il Channel Mode utilizzato.

00	Stereo
01	Joint Stereo
10	Dual Channel
11	Single Channel

Figure 3.7: Channel Modes.

3.2.9 Mode extension

Sono 2 bits utilizzati in modalità joint stereo ed indicano i metodi da utilizzare. La modalità può variare anche da frame a frame.

Bits	Intensity stereo	MS stereo
00	Off	Off
01	On	Off
10	Off	On
11	On	On

Figure 3.8: Mode extension bits.

3.2.10 Copyright bit

Indica se è legale copiare il contenuto.

3.2.11 Home

Indica se il frame è posto nel media originale.

3.2.12 Emphasis

Indica al decodificatore se è necessario effettuare la de-enfatizzazione, ovvero se è necessario re-equalizzare il suono dopo una Dolby-like noise suppression.

00	None
01	50/15 ms
10	Reserved
11	CCITT J.17

Figure 3.9: Noise suppression model.

3.2.13 Private

Se settato ad 1 verrà utilizzato CRC.

3.3 CRC

Secondo lo standard i dati sensibili sono da 16 a 31 bits sia nell'header che nel side information. Se questi valori sono errati corromperanno l'intero frame mentre un'errore nei main data corrompe solo parte del frame.

3.4 Side information

La side information è la parte del frame che contiene le informazioni per decodificare i main data. La sua taglia dipende dal channel mode utilizzato per la codifica, 17 o 32 bytes. Le diverse parti sono rappresentate dalla seguente tabella.

main_data_begin	private_bits	scfsi	Side_info gr. 0	Side_info gr. 1
-----------------	--------------	-------	-----------------	-----------------

Figure 3.10: Side information.

3.4.1 main_data_begin

Utilizzando il formato layer III è utilizzata una tecnica chiamata *bit reservoir* la quale consente di sfruttare lo spazio libero alla sinistra del frame per frame consecutivi. Per essere in grado di sapere dove i dati di un certo frame iniziano il decodificatore deve leggere il main_data_begin il quale consiste in un offset negativo dal primo byte della word di sincronizzazione. Dato che il campo è di 9 bits esso può puntare a 4088 bits precedenti, ovvero diversi frame precedenti.

3.4.2 scfsi (4 bits, 8 bits)

ScaleFactor Selection Information determina se gli stessi scalefactors sono utilizzati per entrambi i granules, o meno.

group	scalefactor bands
0	0,1,2,3,4,5
1	6,7,8,9,10
2	11,12,13,14,15
3	16,17,18,19,20

Figure 3.11: Scalefactor groups.

Qui gli scalefactors sono divisi in 4 gruppi. Sono trasmessi 4 bits per canale, uno per ogni scalefactorband. Se un bit appartenente ad uno scalefactor band è 0 allora gli scalefactors per quella determinata banda sono trasmessi per ogni granule. Se lo scalefactor per il granule0 è valido anche per il granule1 allora esso sarà trasmesso una sola volta.

3.4.3 side.info_gr0(1)

Le ultime due sezioni sono identiche strutturalmente e rappresentano i due granules.

part2_3_length	big_values	global_gain	scalefac_compress
windows_switching_flag	block_type	mixed_block_flag	table_select
subblock_gain	region0_count	region1_count	preflag
scalefac_scale	count1table_select		

Figure 3.12: Side info structure.

par2_3_length (12 bits, 24 bits)

Rappresenta il numero di bits allocati nella *main part* per lo scalefactor (part 2) e l'Huffman Coding (part 3).

big_values (9 bits, 18 bits)

Le 576 linee di frequenza non sono codificate utilizzando la stessa tabella di Huffman, bensì sono divise in 5 regioni e codificate singolarmente in modo da ottimizzare la codifica di Huffman su diverse parti dello spettro.

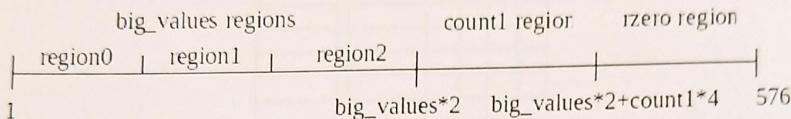


Figure 3.13: Region of the frequency spectrum.

Le regioni sono scelte considerando il massimo valore quantizzato, assumendo che alla frequenza più alta avremo la più bassa ampiezza o la necessità di non codificarla al pieno. Utilizzando questi criteri avremo che nella *rzero region* rappresenta le frequenze più alte quantizzate con valore 0. In *count1 region* sono contenute tutte le quadruple contigue quantizzate con valori 1, 0, o -1. Infine la *big_values* contiene tutte le coppie di valori quantizzate con valori che si estendono fino allo zero.

global_gain (8 bits, 16 bits)

Specifica la grandezza del passo di quantizzazione, necessaria per la dequantizzazione.

scalefac_compress (4 bits, 8 bits)

Determina il numero di bits utilizzati per la trasmissione degli scalefactors. Ogni granule può essere diviso in 12 o 21 scalefactors, in base alla finestra utilizzata. Gli scalefactors sono divisi in 2 gruppi, 0-10 11-20 per le long windows e 0-6, 7-11 per le short windows.

Lo *scalefac_compress* è l'indice di una tabella che indica quanti bit devono essere assegnati al primo o al secondo gruppo di scalefactor.

scalefac_compress	slen1	slen2
0	0	0
1	0	1
2	0	2
3	0	3
4	3	0
5	1	1
6	1	2
7	1	3
8	2	1
9	2	2
10	2	3
11	3	1
12	3	2
13	3	3
14	4	2
15	4	3

Figure 3.14: scalefac_compress table.

windows_switching_flag (1 bit, 2 bits)

Indica se si sta utilizzando una finestra diversa dalla normale.

block_type

Questo campo è utilizzato solo se la windows_switching_flag è settato ed indica il tipo di finestra utilizzato per quel particolare granule.

block_type	window type
00	forbidden
01	start
10	3 short windows
11	end

Figure 3.15: block_type values.

mixed_blockflag

Questo campo è utilizzato solo se la windows_switching_flag è settato ed indica che differenti tipi di finestre sono utilizzate nelle frequenze più alte e più basse. Se questo è impostato le due sottobande più basse sono trasformate utilizzando finestre normali mentre le rimanenti 30 sono trasformate utilizzando le finestre secondo il block_type dichiarato.

3.4.4 table_select(10-15 bits, 15-30 bits)

Nello standard sono definite 32 possibili tabelle di Huffman da utilizzare. Grazie a questo valore è possibile decidere quale di esse utilizzare per ogni regione, granule e canale. La tabella scelta è fortemente dipendente dalle caratteristiche del segnale e dal valore massimo di quantizzazione delle 576 linee di frequenza utilizzate dei granule.

3.4.5 subblock_gain(9 bits, 18 bits)

Questi 3 bit indicano l'offset del guadagno del global_gain per ogni sottoblocco.

3.4.6 region0_count (4 bits, 8 bit), region1_count (3 bits, 6 bits)

region0_count e region1_count contengono rispettivamente un numero inferiore di scalefactor bands rispetto a region0 e region1. I confini della regione sono definiti rispetto al partizionamento dello spettro delle frequenze negli scalefactor bands.

3.4.7 preflag (1 bit, 2 bits)

E' uno shortcut per l'amplificazione addizionale ad alta frequenza dei valori quantizzati. Se il flag è impostato vengono sommati agli scalefactor dei valori definiti all'interno di una tabella.

3.4.8 scalfac_scale(1 bit, 2 bits)

Lo scalefactor è quantizzato logaritmicamente con una quantità con uno step size in accordo con la ta seguente tabella:

scalfac_scale	step size
0	$\sqrt{2}$
1	2

Figure 3.16: Quantization step size applied to scalefactor.

count1table_select (1 bit, 2bits)

Per il count1 region sono disponibili due tabelle di Huffman. Questo campo specifica quello da applicare.

3.4.9 Main Data

La parte dei main data consiste negli scalefactors, tabelle di Huffman e gli Ancillary Data.

3.4.10 scalefactors

Scalefactors Lo scopo degli scalefactors è quello di ridurre il rumore di quantizzazione. Se i campioni in un particolare scalefactor band sono scalati nel modo corretto il rumore di quantizzazione verrà completamente mascherato. Uno scalefactor è inviato per ogni scalefactor band. Il campo scfsi determina se uno scalefactor è condiviso tra più granule.

La suddivisione dello spettro in scalefactor band è fissato indipendentemente dalla lunghezza della finestra e dalla frequenza di campionamento ed è salvata in una tabella nel coder e nel decoder.

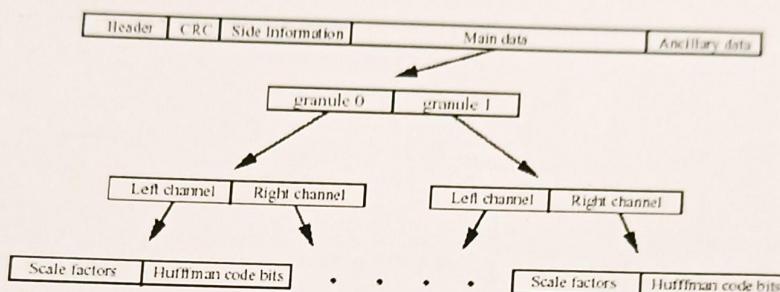


Figure 3.17: Organization of scalefactors.

Huffman code bits Questa parte del frame contiene i bits del codice di Huffman. Le informazioni su come decodificare questi bit sono contenute nelle side information. Le tre subregions nella big_values region sono codificate sempre a coppia.

A seconda se vengono utilizzati blocchi piccoli o grandi, le informazioni contenute nei codici di Huffman variano nell'ordine.

Ancillary Data E' un campo opzionale di non definita lunghezza. I dati in esso contenuti iniziano subito dopo i codici di Huffman e terminano dove main_data_begin punta.

3.4.11 ID3

Il formato MP3 tratta in maniera molto efficace la compressione dei file audio ma non consente la possibilità di memorizzare informazioni testuali. Per ovviare a questo problema è stato introdotto un campo di 128 bytes alla fine del file, chiamato *ID3* e la cui struttura è mostrata dalla seguente figura:

'TAG'	Title (30)	Artist (30)	Album (30)	Year (4)	Comment (28)	'0'	Track (1)	Genre (1)
-------	---------------	----------------	---------------	-------------	-----------------	-----	--------------	--------------

Figure 3.18: ID3 format.

Sfortunatamente la grandezza dei campi è molto limitata ed il fatto che i dati siano posti alla fine del file non ne rende possibile la lettura durante lo streaming. Per ovviare a questo problema è stata introdotta una versione molto più complessa, rinominata ID3v2.

3.5 Codifica

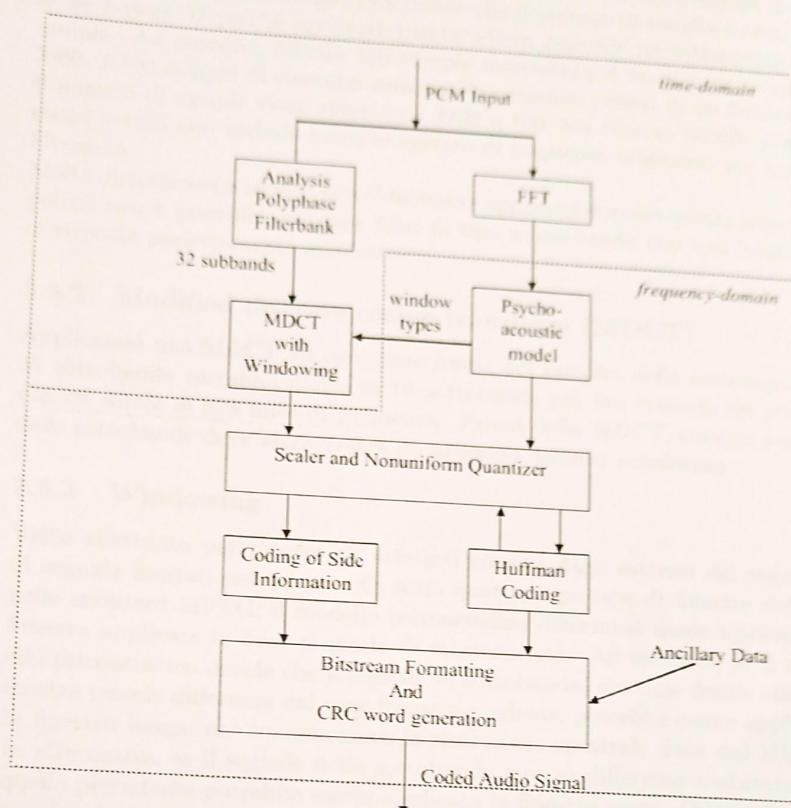


Figure 3.19: Schema di codifica

3.5.1 Analysis Polyphase Filterbank

1152 PCM sample vengono filtrati attraverso 32 sottobande di frequenza equispaziate in base alla frequenza Nyquist del segnale PCM. Se la frequenza del segnale PCM è 44,1kHz, la Frequenza di Nyquist sarà 22.05kHz. Ciascuna sottobanda sarà ampia circa 689Hz (calcolata dividendo $Freq.Nyquist/32$). La sottobanda più bassa possiederà un range compreso tra i 0 e i 689Hz, la sottobanda successiva tra i 689 e i 1378Hz e così via. Ogni sample potrebbe

contenere componenti di segnale tra 0 e 22.05kHz che verranno filtrate in una sottobanda appropriata. Questo significa che il numero di sample è cresciuto di un fattore 32 poiché ogni sottobanda adesso possiede un sottospettro del sample. Ad esempio, filtrare 100 sample incrementa il numero di sample a 3200. I 100 sample di ciascuna sottobanda verranno ridotti di un fattore 32. Il numero di sample viene ridotto da 3200 a 100, ma ciascun sample pesa di meno perché non include tutto lo spettro di frequenze originario per via del filtraggio.

Molta distorsione (*aliasing*) potrebbe essere applicata durante questa riduzione poiché non è possibile costruire filtri di tipo passa-banda con una frequenza di risposta perfettamente quadrata.

3.5.2 Modified discrete cosine transform (MDCT)

Applicando una MDCT ad ogni *time frame* dei samples della sottobanda, le 32 sottobande verranno divise in 18 sottobande più fini creando un *granule* con un totale di 576 linee di frequenza. Prima della *MDCT*, ciascun segnale delle sottobande deve attraversare un processo definito *windowing*.

3.5.3 Windowing

Viene effettuato per ridurre gli artefatti causati dagli estremi dei segmenti di segnale limitati nel tempo. Ci sono quattro tipologie di finestre definite nello standard MPEG; il modello psicoacustico determina quale tipologia di finestra applicare in base al grado di stazionarietà. Ad esempio, se il modello psicoacustico decide che il segnale di sottobanda, nel time frame attuale, mostra piccole differenze dal time frame precedente, potrebbe essere applicata la finestra lunga, che incrementerà la risoluzione spettrale data dal MDCT. In alternativa, se il segnale della sottobanda mostra differenze sostanziali da quello precedente potrebbe essere applicata la finestra corta. Questa finestra consiste in tre finestre corte sovrapposte ed incrementerà la risoluzione temporale fornita dal MDCT. Una risoluzione temporale più alta è necessaria per controllare gli artefatti temporali, ad esempio *pre-echi*.

Per ottenere un miglior adattamento quando le finestre temporali sono richieste, vengono definite due finestre chiamate *start* e *stop*.

Una finestra lunga diventa una finestra di *start* se è immediatamente seguita da una finestra corta.

In modo simile, una finestra lunga diventa una finestra di *stop* se è immediatamente preceduta da una finestra breve. Le finestre di *start* e *stop* vengono distorte per adattarsi ai lati più ripidi della finestra breve più vicina.

La distorsione introdotta nella *Analysis Polyphase Filterbank* è rimossa per

ridurre l'ammontare di informazione che deve essere trasmessa. Vengono utilizzate una serie di computazioni a farfalla che aggiungono versioni specchiate o ripesate di sottobande adiacenti una all'altra (vedi 3.6.7)

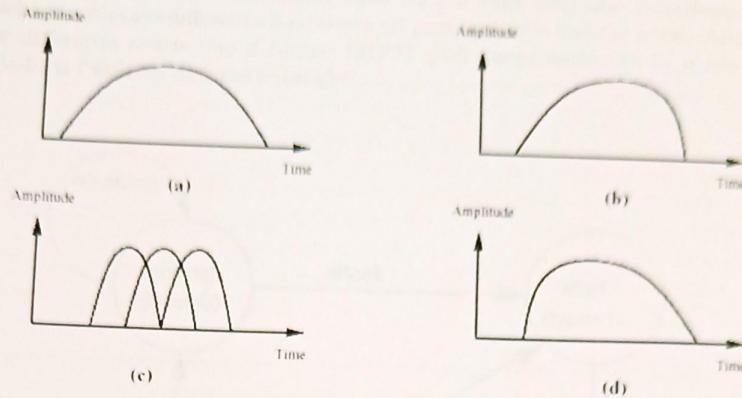


Figure 6.2: Window types

(a) normal window, (b) start window, (c) short windows, (d) stop window

Figure 3.20: Tipologie di finestre applicabili

3.5.4 FFT

Mentre il segnale è processato dalla *Analysis Polyphase Filterbank* viene anche spostato nel dominio delle frequenze da una trasformata veloce di Fourier. Vengono applicate contemporaneamente due FFT a 1024 e 256 punti sui 1152 Sample PCM per dare una maggiore risoluzione di frequenze e informazioni sui cambiamenti allo spettro nel tempo.

3.5.5 Psychoacoustic Model

Questa componente riceve l'input dall'output della componente FFT. Dato che i sample sono nel dominio delle frequenze è possibile applicare un insieme di algoritmi. Questi algoritmi possiederanno informazioni sulla percezione del suono da parte dell'uomo, pertanto possono dare informazioni su quali parti sono udibili dall'uomo e quali no. Queste informazioni sono fondamentali per determinare quali tipologie di finestre devono essere applicate dall'MDCT e per dare informazioni al *Nonuniform Quantization block* su come quantizzare

le linee di frequenze.

Per capire quale tipologia di finestra deve essere inviata al MDCT è necessario comparare i due spettri ottenuti dalla FFT e dagli altri due precedenti. Se vengono rilevate differenze è richiesto un passaggio alle finestre brevi. Appena le differenze spariscono il blocco MDCT può essere notificato in modo che riadotti l'utilizzo di finestre lunghe.

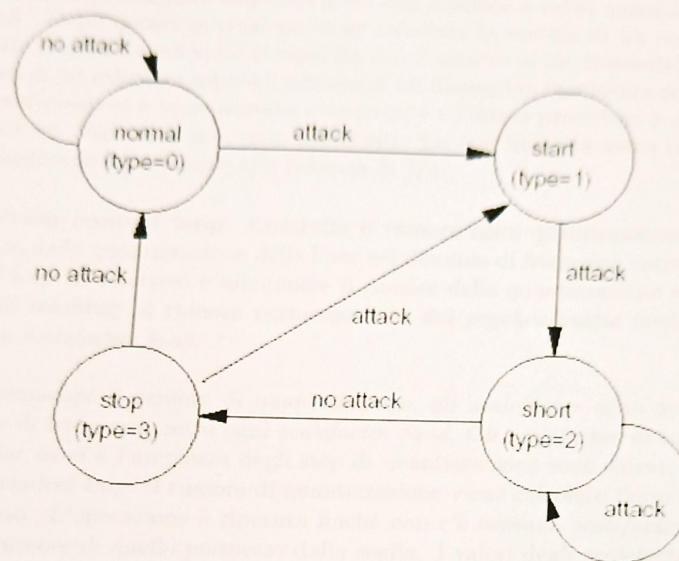


Figure 3.21: Scelta finestra da applicare

Lo *Psychoacoustic Model* analizza lo spettro della FFT per individuare componenti tonali dominanti e, per ciascuna *critical band*, vengono calcolate soglie per il masking. Componenti di frequenza sotto questa soglia vengono nascoste.

3.5.6 Nonuniform Quantization

In queste due componenti lo scaling, la quantizzazione e la codifica di Huffman vengono applicati a 576 valori spettrali. L'operazione viene compiuta in modo iterativo in due loop innestati, un *distortion control loop* (più esterno) e un *rate*

control loop(più interno).

Rate control loop Effettua la quantizzazione dei sample sul dominio delle frequenze e determina l'ampiezza dello step di quantizzazione. In aggiunta si occupa di suddividere i *big_values* in regioni, di selezionare le tavole di Huffman per ciascuna regione e di calcolare i confini tra le regioni.

I smaple sono quantizzati incrementando l'ampiezza dello step finché i valori quantizzati non possono essere codificati utilizzando una delle tavole di Huffman. Una maggiore ampiezza dello step conduce a valori quantizzati più piccoli. Dopo questa operazione viene calcolata la somma di bit codificata tramite Huffman e poi viene comparata con il numero di bit disponibile. Se la somma di bit calcolata supera il numero di bit disponibili l'ampiezza dello step di quantizzazione è incrementato nuovamente e l'intera procedura è ripetuta finché i bit disponibili non sono sufficienti. La non linearità viene ottenuta incrementando ogni sample alla potenza di 3/4.

Distortion control loop Controlla il rumore nella quantizzazione che è prodotto dalla quantizzazione delle linee nel dominio di frequenze entro il *rate control loop*. L'obiettivo è mantenere il rumore della quantizzazione sotto la soglia di masking (il rumore permesso dato dal *psychoacoustic model*) per ciascuna *scalefactor band*.

Per determinare il rumore di quantizzazione, gli *scalefactor* sono applicati alle linee di frequenza entro ogni *scalefactor band*. Gli *scalefactor* di tutte gli *scalefactor band* e l'ampiezza degli step di quantizzazione sono salvati prima del *rate control loop*. Il rumore di quantizzazione viene calcolato dopo il loop più interno. L'operazione è ripetuta finché non c'è nessuna *scalefactor band* con più rumore di quello permesso dalla soglia. I valori degli *scalefactor* che appartengono alle bande che sono troppo rumorose vengono incrementati per ogni iterazione del loop. Il risultato è che il rumore causato dalla quantizzazione non sarà udibile da un essere umano e il loop verrà interrotto.

Possono esserci ancora situazioni dove entrambi i loop potrebbero continuare e questo dipende dalla soglia calcolata. Per evitare queste situazioni ci sono diverse condizioni nel *distortion control loop* che possono essere controllate per interrompere preventivamente queste iterazioni.

Input dei loop:

- Vettore delle magnitudini dei 576 valori spettrali;

- Distorsione consentita per le *scalefactor bands*;
- Il numero di *scalefactor bands*;
- Bits disponibili per la codifica di Huffman e per la codifica degli *scalefactors*;
- Il numero di bit da aggiungere al numero medio di bit, richiesti dal valore dell'entropia psicoacustica per il *granule*.

Output del loop:

- Vettore di 576 valori quantizzati;
- Gli *scalefactors*;
- Informazione sull'ampiezza degli step per la quantizzazione;
- Numero di bit non utilizzati per utilizzi futuri;
- Preflag (pre-enfasi dei loop attiva/disattiva);
- Informazioni per la codifica di Huffman:
 - *big_values* (numero delle coppie dei valori codificati con Huffman, escludendo *count_1*);
 - *count1tableselect* (tavola di Huffman di valori assoluti $j = 1$ alla fine superiore dello spettro)
 - *table_select* (tavola delle regioni di Huffman);
 - *region0_count* e *region1_count* (usati per calcolare i confini tra le regioni);
 - *part2_3_length*.

3.5.7 Codifica di Huffman

I valori quantizzati sono codificati tramite Huffman. Ogni divisione dello spettro delle frequenze può essere codificata utilizzando tavole differenti. La codifica di Huffman costituisce uno dei maggiori punti di forza per MPEG-1 Layer III, garantendo ottima qualità a bassi bitrates.

3.5.8 Codifica di informazioni secondarie

Tutti i parametri generati dall'encoder sono raccolti così che il decoder possa riprodurre il segnale audio. Questi parametri risiedono nelle informazioni secondarie del frame.

3.5.9 Bitstream Formatting CRC word generation

Nel componente finale il bitstream viene finalmente generato. L'header del frame, le informazioni secondarie, CRC e le linee di frequenze codificate con Huffman sono combinate per ottenere i frames. Ciascuno di questi frame rappresenta 1152 sample PCM codificati.

3.6 Decodifica

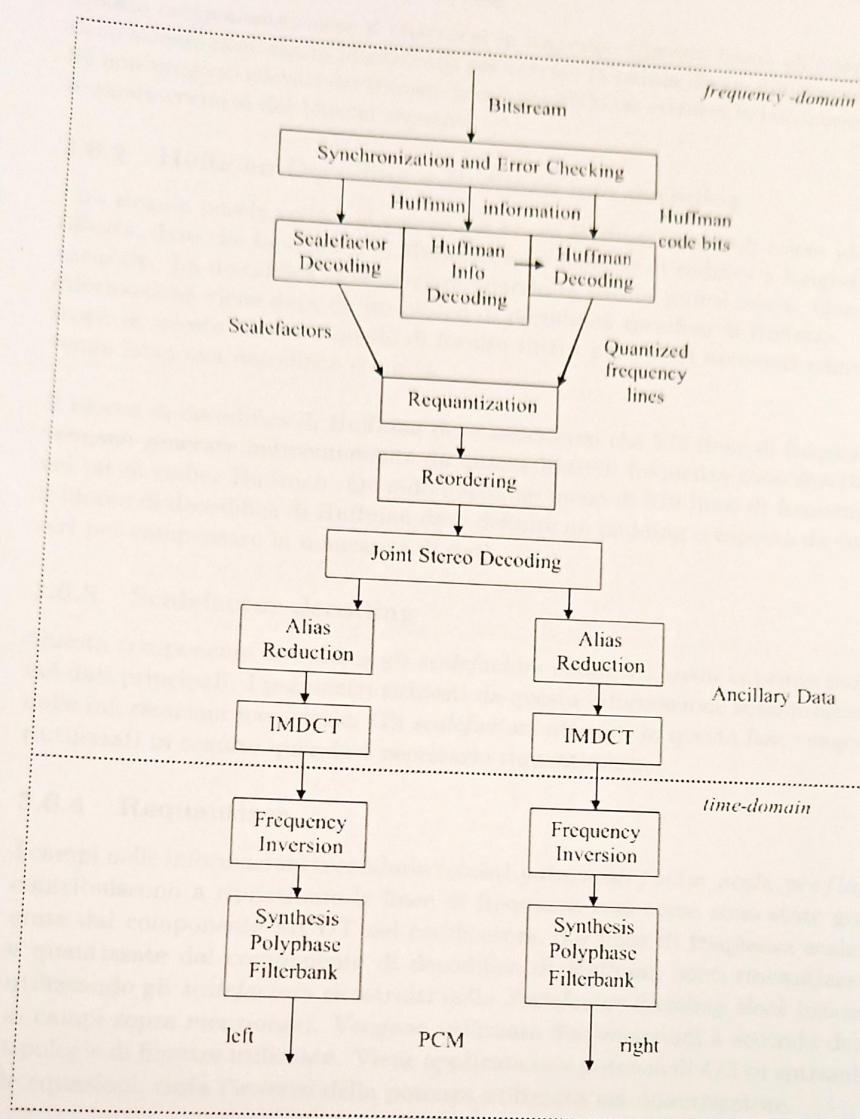


Figure 3.22: Schema di decodifica

3.6.1 Sync and Error Checking

Questo componente riceve il bitstream in ingresso. Ciascun frame all'interno dello stream deve essere identificato per cercare la parola di sincronizzazione. Se non vengono rilevati dei frames, non è possibile far estrarre le informazioni in modo corretto dai blocchi seguenti.

3.6.2 Huffman Decoding e Huffman info decoding

Una singola parola codice al centro dei bit di Huffman non può essere identificata, dato che la codifica di Huffman è un metodo di codifica a lunghezza variabile. La decodifica deve iniziare quando la parola codice inizia. Questa informazione viene data da un blocco di decodifica specifico di Huffman. Lo scopo di questo blocco è quello di fornire tutti i parametri necessari affinché venga fatta una decodifica corretta.

Il blocco di decodifica di Huffman deve assicurarsi che 576 linee di frequenza vengano generate indipendentemente da quante linee di frequenza sono descritte nei bit di codice Huffman. Quando appaiono meno di 576 linee di frequenza, il blocco di decodifica di Huffman deve definire un padding composto da tutti zeri per compensare la mancanza di dati.

3.6.3 Scalefactor decoding

Questa componente decodifica gli *scalefactors* codificati, ossia la prima parte dei dati principali. I parametri richiesti da questa informazione sono prelevati dalle informazioni secondarie. Gli *scalefactors* ottenuti in questa fase vengono riutilizzati in seguito quando è necessario riquantizzare.

3.6.4 Requantizer

I campi nelle informazioni secondarie (*global_gain*, *scalefactor_scale*, *preflag*) contribuiscono a ripristinare le linee di frequenza così come sono state generate dal componente MCDT nel codificatore. Le linee di frequenza scalate e quantizzate dal componente di decodifica di Huffman sono riquantizzate utilizzando gli *scalefactors* ricostruiti nello *Scalefactor decoding block* insieme ai campi sopra menzionati. Vengono utilizzate due equazioni a seconda delle tipologie di finestre utilizzate. Viene applicata una potenza di 4/3 su entrambe le equazioni, ossia l'inverso della potenza utilizzata nel quantizzatore.

3.6.5 Reordering

Le linee di frequenza generate dal *Requantizer block* non sono sempre ordinate allo stesso modo. Nel MDCT l'uso di lunghe finestre prima delle trasformazioni genererebbe linee di frequenza ordinate prima per sottobanda e poi per frequenza. Utilizzando finestre brevi verrebbero generate linee di frequenza ordinate prima per sottobande, poi per finestra e poi per frequenza. Al fine di incrementare l'efficienza della codifica di Huffman le linee di frequenza, nel caso di finestre brevi, vengono prima ordinate per sottobande, poi per frequenza e infine per finestre, dato che i samples con frequenze simili hanno probabilmente valori più simili tra loro.

Il componente di *reordering* cercherà finestre brevi in ciascuna delle 36 sottobande e le riordinerà nel caso che vengano trovate.

3.6.6 Stereo Decoding

Questo componente effettua le operazioni necessarie per convertire il segnale stereo codificato in segnali separati (uno a destra e uno a sinistra). Il metodo utilizzato per codificare il segnale stereo può essere retto dal parametro *mode* e *mode_extension* nell'header di ciascun frame.

3.6.7 Alias Reduction

Al fine di ricostruire il segnale audio dobbiamo necessariamente riaggiungere gli artefatti di aliasing nel segnale (rimossi nella fase di Analysis Polyphase Filterbank, 3.5.1). Questa ricostruzione consiste nell'effettuare computazioni a farfalla per ciascuna sottobanda, come illustrato in figura.

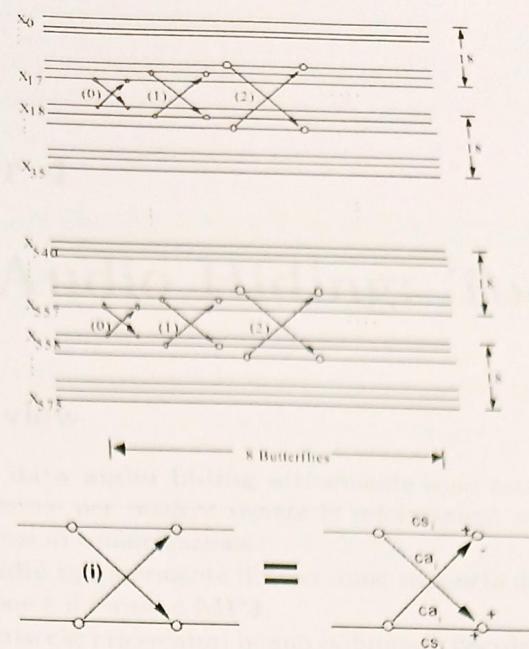


Figure 3.23: Riduzione alias

3.6.8 Inverse Modified Discrete Cosine Transform (IMDCT)

Le linee di frequenza dal componente di *Alias Reduction* sono mappati in 32 sottobande di *Polyphase filter*. Questo componente emetterà 18 sample nel dominio del tempo per ciascuna delle 32 sottobande.

3.6.9 Frequency Inversion

Ciascun time sample dispari di ogni sottobanda dispari è moltiplicato per -1 al fine di compensare le inversioni di frequenza della *Synthesis Polyphase Filterbank*.

3.6.10 Synthesis Polyphase Filterbank

Questo componente trasforma le 32 sottobande dei 18 sample nel dominio del tempo di ogni *granule* in 18 blocchi dei 32 sample PCM, fornendo il risultato finale.

Chapter 4

Data Audio Hiding: Tools

4.1 Overview

Le tecniche di **data audio hiding** attualmente sono considerate uno strumento potentissimo per *rendere segrete* le informazioni scambiate tramite i più comuni mezzi di comunicazione.

Il **formato audio** maggiormente diffuso come supporto di memorizzazione e di comunicazione è il formato **MP3**.

E' per tale motivo che i ricercatori hanno sviluppato tecniche di steganografia basate sull'utilizzo di tale formato.

La **compressione MP3** sfrutta delle *debolezze del sistema uditivo umano* per comprimere le informazioni presenti nel file audio. I più conosciuti *tools* per il **data audio hiding** sono :

1. DeepSound [];
2. MP3Stego [];
3. Steghide [];
4. QuickStego [];
5. AudioStego [].

4.2 DeepSound

4.2.1 Overview

DeepSound è un tool per il **data audio hiding** utilizzabile sui *sistemi operativi Windows* (XP SP3/Vista/7/8/10).

E' possibile scaricarlo gratuitamente all'indirizzo <http://jpinsoft.net/deepsound/>, inoltre bisogna tener presente che durante l'installazione è necessario interrompere l'esecuzione di *antivirus* o *firewall* poichè il software è riconosciuto come potenzialmente dannoso.

4.2.2 Interfaccia - Barra dei Menù

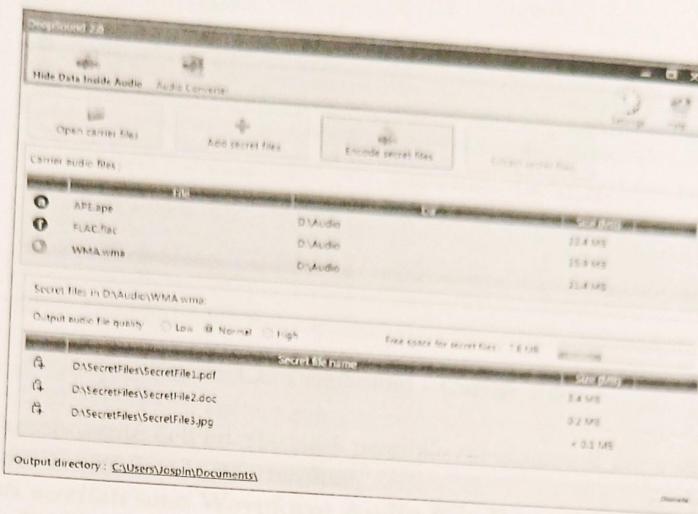


Figure 4.1: DeepSound - Barra dei menù

La barra dei menù è composta da:

- **Hide Data Inside Audio:** pulsante che permette di scegliere la modalità **Steganografica** del tool, ovvero permette di nascondere le *informazioni* in un *segnale audio* detto **carrier**;
- **Audio Converter:** pulsante che permette di scegliere la modalità **Conversione** che permette di convertire il file audio da a FLAC, MP3, WMA, WAV, APE a FLAC, MP3, WAV, APE;
- **Settings:** pulsante che permette di **configurare** il tool, ad esempio permette di *scegliere la lingua* (Inglese o Slovacco) o la **chiave di cifratura** (AES a 256 bit) da utilizzare per rendere segreti i dati;
- **Help:** pulsante che fornisce la **documentazione ufficiale** del tool.

4.2.3 Interfaccia - Carrier Files

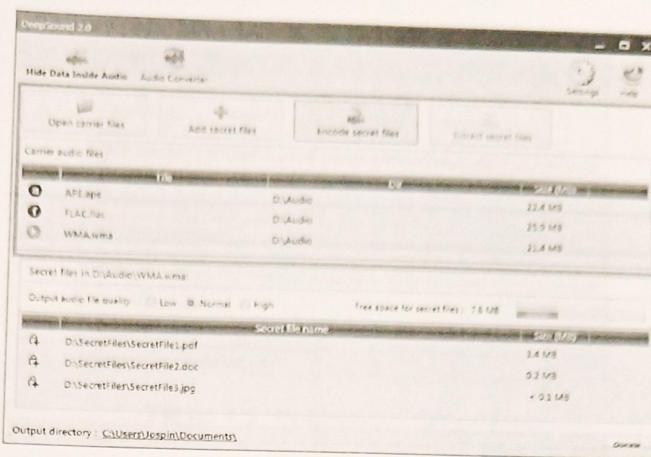


Figure 4.2: DeepSound - Carrier Files

In tale componente dell'interfaccia è possibile selezionare il *file audio* da utilizzare per nascondere le informazioni.

I *formati accettati* sono **Waveform Audio File Format (wav)**, **Free Lossless Audio Codec (flac)** o **Mokey's Audio (ape)**.

4.2.3 Interfaccia - Carrier Files

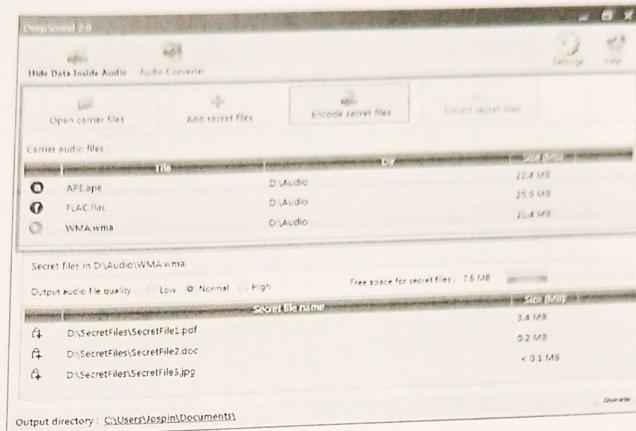


Figure 4.2: DeepSound - Carrier Files

In tale componente dell'interfaccia è possibile selezionare il *file audio* da utilizzare per nascondere le informazioni. I formati accettati sono **Waveform Audio File Format (wav)**, **Free Lossless Audio Codec (flac)** o **Mokey's Audio (ape)**.

4.2.4 Interfaccia - Secret Files

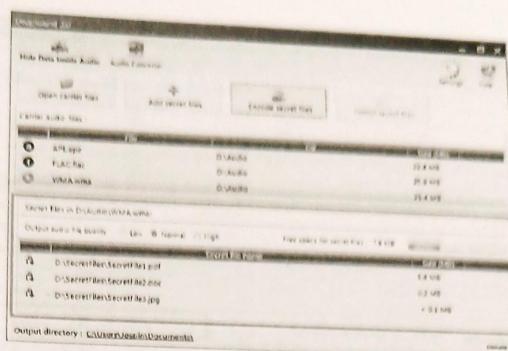


Figure 4.3: DeepSound - Secret Files

In tale sezione è possibile selezionare i *file* da nascondere nel segnale audio. È possibile notare che il tool calcola lo *spazio libero* e utilizzabile per nascondere le informazioni in modo progressivo. Lo spazio da utilizzare per nascondere i file dipende dalla **qualità dell'audio** scelta: *low*, *normal* e *high*.

È possibile scegliere la qualità della compressione audio secondo diversi criteri, secondo cui sarà scelta durante il processo di compressione.

1. Low: 50 MB di informazioni segrete inserite nel file audio originale.
2. Normal: 90 MB di informazioni segrete inserite nel file audio originale.
3. High: 120 MB di informazioni segrete inserite nel file audio originale.

Indicando la qualità *low* il file in output risparmierà più spazio di memoria rispetto a *high*.

4.2.5 Interfaccia - Qualità

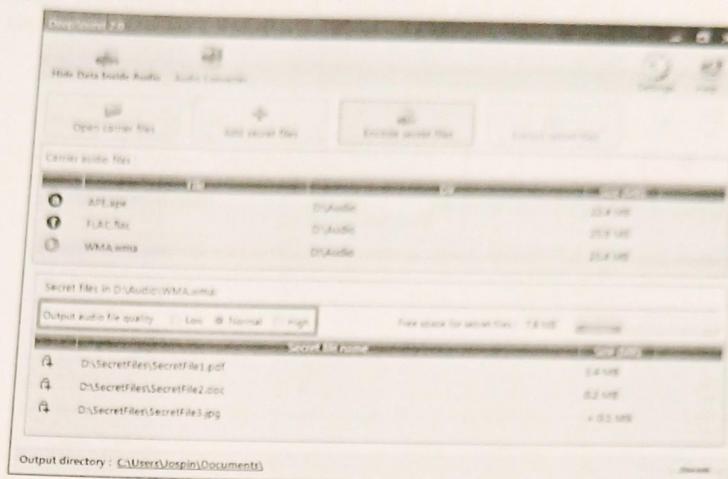


Figure 4.4: DeepSound - Qualità

La quantità di informazione nascosta nel segnale dipende dalla qualità del segnale output scelta durante il processo di steganografia.

Le quantità sono determinate dalle seguenti regole:

1. **Low:** 50 MB di informazioni segrete nascoste in 100 MB del file audio originale;
2. **Normal:** 25 MB di informazioni segrete nascoste in 100 MB del file audio originale;
3. **High:** 12.5 MB di informazioni segrete nascoste in 100 MB del file audio originale.

Scegliendo la **qualità low** il file in output contiene un rumore di sottofondo percepibile.

4.2.6 Conversione

Come accennato in precedenza, DeepSound può essere utilizzato anche come tool di conversione.

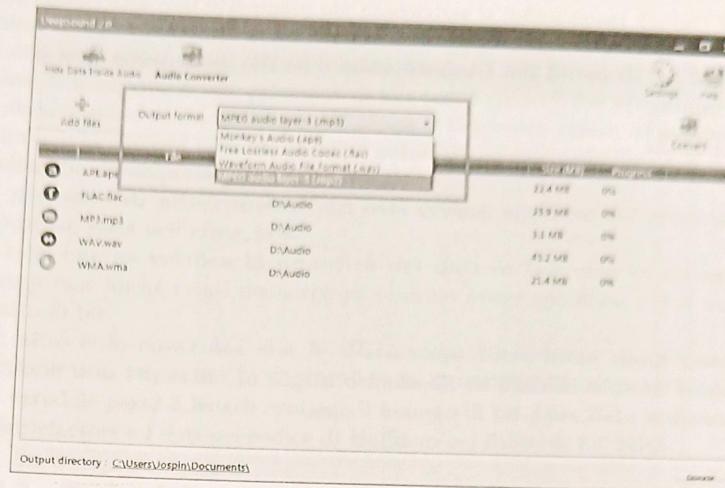


Figure 4.5: DeepSound - Conversione

I formati in output sono i seguenti:

- **Waveform Audio File Format** (.wav)
- **Free Lossless Audio Codec** (.flac)
- **Windows media audio lossless** (.wma)
- **MPEG audio layer-3** (.mp3)
- **Monkey's Audio** (.ape)

4.2.7 Svantaggi

Pur essendo uno dei tool di audio data hiding maggiormente conosciuto e utilizzato, DeepSound è un tool utilizzabile solo su dispositivi *Windows*; inoltre non permette di nascondere i dati in file *MP3* in modo diretto.

4.3 MP3Stego

4.3.1 Overview

MP3Stego è un tool sviluppato per nascondere le informazioni durante il processo di **compressione** di file MP3.

I dati sono **compresi**, **cifrati** e infine **nascosti nel flusso di bit MP3**. Una delle caratteristiche principali di tale tool è che se il file steganografico è decompresso *senza utilizzare MP3Stego* e l'apposita *chiave di cifratura* allora le informazioni nascoste saranno automaticamente cancellate e la qualità dell'audio decompresso sarà molto bassa.

Il **processo di offuscamento** dei dati avviene al centro del **processo di codifica**, ossia nell'*inner-loop*.

Il ciclo interno **verifica la quantità** dei dati in ingresso ed aumenta la dimensione finché i dati memorizzati possono essere codificati con il numero esatto di bit.

Un altro ciclo **controlla** che le **distorsioni introdotte dalla quantizzazione non superino la soglia** definita da un modello acustico standard. La variabile `part2_3_length` contiene il numero di bit `main_data` utilizzati per lo `scalefactors` e i dati nel codice di Huffman nel flusso di bit MP3.

4.3.2 Tool nel dettaglio

Il **sorgente** (da compilare per l'utilizzo) è disponibile all'indirizzo http://www.petitcolas.net/fabien/software/MP3Stego_1_1_18.zip.

Il tool fu progettato per l'utilizzo da *riga di comando* ma nel 2015 Frans Vyncke sviluppò un'**interfaccia grafica** disponibile all'indirizzo http://www.petitcolas.net/fabien/software/MP3Stego_GUI.zip.

E' basato sui seguenti componenti:

- **Encoder:** 8HZ-mp3 0.2b;
- **Decoder:** MP3Decoder (dist10) ISO MPEG Audio Subgroup Software Simulation Group;
- **Liberia di compressione:** ZLib 1.1.4 di Jean-Loup Gailly;
- **Libreria di cifratura:** 3DES di Eric Young;
- **Funzione hash:** SHA-1 di James J. Gillogly.

CHAPTER 4. DATA AUDIO HIDING: TOOLS

45

4.3.3 Codifica

- Il comando da lanciare per nascondere un file *hidden.txt* in una traccia *svega.wav* usando la password *pass* e ottenendo il file *svega_stego.mp3* è il seguente:

```
encode -E hidden\$\_stext.txt -P pass svega.wav svega\$\_stego.mp3
```

```
Z:\Development\MP3Stego>encode -E hidden_text.txt -P pass svega.wav svega_stego.mp3
See README file for copyright info
MP3StegoEncoder 1.1.15
Input file = 'svega_stego.mp3'  Output file = 'svega_stego.mp3.pcm'
Will attempt to extract hidden information. Output: svega_stego.mp3.txt
HDR: s=FFFF, id=1, l=3, ep=off, br=9, sf=0, pd=1, pr=0, n=3, js=0, c=0, a=0, e=0
MPEG-1, layer=III, tot bitrate=128, sfrq=44.1
mode=single-ch, shln=32, jshd=32, ch=1
[Frame 791]Avg slots/frame = 417.434; b/snp = 2.98; br = 127.839 kbps
Encoding "svega.wav" to "svega_stego.mp3"
Hiding "hidden_text.txt"
[Frame 791 of 791] <100.00%> Finished in 0: 0: 6
```

Figure 4.6: MP3Stego - codifica.

4.3.4 Decodifica

- Il comando da lanciare per ottenere le informazioni nascoste nel file *svega_stego.mp3* è il seguente:

```
decode -X P pass svega\$\_stego.mp3
```

```
Z:\Development\MP3Stego>decode -X -P pass svega_stego.mp3
See README file for copyright info
Input file = 'svega_stego.mp3'  Output file = 'svega_stego.mp3.pcm'
Will attempt to extract hidden information. Output: svega_stego.mp3.txt
HDR: s=FFFF, id=1, l=3, ep=off, br=9, sf=0, pd=1, pr=0, n=3, js=0, c=0, a=0, e=0
MPEG-1, layer=III, tot bitrate=128, sfrq=44.1
mode=single-ch, shln=32, jshd=32, ch=1
[Frame 791]Avg slots/frame = 417.434; b/snp = 2.98; br = 127.839 kbps
Decoding of "svega_stego.mp3" is finished
The decoded PCM output file name is "svega_stego.mp3.pcm"
```

Figure 4.7: MP3Stego - decodifica.

4.4 Steghide

4.4.1 Overview

Steghide è uno dei tool più *conosciuti e potenti* per la steganografia sia in **immagini** che in **file audio**.

Esso è sotto *licenza GNU* ed è disponibile sia per **Windows** che per **Linux**. La *versione attuale* è la 0.5.1.

Il *codice sorgente* è disponibile all'indirizzo <http://prdownloads.sourceforge.net/steghide/steghide-0.5.1.tar.gz?download>.

Il tool permette di:

- comprimere i dati nascosti;
- cifrare i dati nascosti;
- controllare l'integrità mediante meccanismi di **checksum**.

I **formati in input** supportati e utilizzati come **cover file** sono *JPEG, BMP, WAV, AU*.

Non vi sono restrizioni sul formato dei dati da nascondere.

4.4.2 Componenti

Il tool è basato sulle seguenti *librerie*:

- **libmhash**: insieme di *algoritmi hash* e *algoritmi di generazione di chiavi di cifratura*; tale libreria permette al tool di convertire una password in una chiave di cifratura da utilizzare per nascondere i file;
- **libmcrypt**: insieme di *algoritmi di cifratura simmetrici* che permettono di rendere i file segreti;
- **libjpeg**: libreria che permette la compressione di immagini jpeg; essa permette al tool di nascondere dei dati in immagini jpeg;
- **zlib**: libreria per la compressione di dati lossless.

Nella versione per **Windows** le *librerie sono già incluse*, mentre in **Linux** bisogna *scaricarle manualmente* per garantire il corretto funzionamento del tool.

4.4.3 Algoritmo di Embedding

L'algoritmo di embedding (su file audio) lavora come segue:

- I dati segreti sono compressi e cifrati;
- A partire dalla password è generato un PRN (*numero pseudocasuale*) utilizzato per creare una sequenza di campioni audio da utilizzare per nascondere i dati;
- Si utilizzano degli algoritmi basati sulla teoria dei grafi per trovare delle coppie di campioni (*campione originale, campione ad hoc*) tali che scambiando i loro valori si ottiene l'effetto di nascondere i dati;

L'algoritmo permette di nascondere i dati in un segnale audio, facendo sì che il sistema uditivo umano lo percepisca, basandosi sul fatto che il numero di volte in cui il campione audio selezionato e utilizzato per nascondere i dati non cambi tra l'audio originale e l'audio modificato. I meccanismi di steganografia sono attuabili lanciando da riga di comando i seguenti comandi:

- Il comando da lanciare per nascondere un file *secret.txt* in una traccia *audio.wav* usando la password *pass* e ottenendo il file *audioStego.wav* è il seguente:

```
steghide embed -cf audio.wav -ef secret.txt pass
```

- Il comando da lanciare per recuperare un file *secret.txt* in una traccia *audioStego.wav* usando la password *pass* è il seguente:

```
steghide info audioStego.wav pass
```

4.5 QuickStego

4.5.1 Overview

QuickStego è un tool che permette di nascondere dati in file immagini o audio.

I dati nascosti possono esser ricavati solo utilizzando il tool stesso, per cui non

esiste un altro utilizzo.

I tool è totalmente gratuito e disponibile all'indirizzo <http://quickcrypto.com/products/QS12Setup.zip>.

Bisogna tener presente che si vuol nascondere i dati in un'immagine o file audio e ricavarli mediante l'utilizzo di una password bisogna passare alla versione a pagamento (*QuickCrypto*).

QuickStego ha le seguenti caratteristiche:

- E' utilizzabile sui sistemi operativi Windows (XP, Vista e 7);
- Fornisce una funzionalità per l'invio o la condivisione degli output creati;
- Permette di nascondere le informazioni in audio di formati MP3 o WAV;
- L'algoritmo utilizzato per nascondere i dati altera i campioni audio cercando di non creare un disturbo udibile dal sistema uditivo umano.

4.6 AudioStego

4.6.1 Overview

AudioStego è un tool per nascondere qualsiasi tipologia di file (testuale, immagini o audio) in file audio sia MP3 che WAV; è un software open source e disponibile all'indirizzo <https://github.com/danielcardeenas/AudioStego>. E' sviluppato per sistemi Linux e necessita della libreria boost.

4.6.2 Algoritmo

L'algoritmo utilizzato dal programma procede come segue:

- Il file **cover** è caricato in un **buffer** utilizzato durante il processo di embedding, inoltre è inizializzato un **mp3Header** che contiene le posizioni dei byte modificabili (rappresenta lo spazio disponibile);
- Il file da nascondere è caricato in un **buffer** (*msgBuffer*), inoltre è inizializzato un **my_header** che rappresenta le informazioni che saranno nascoste;
- Vengono aggiunti i caratteri @; per indicare la fine dei file;
- Facendo riferimento al **mp3Header** e al **my_header** i byte del messaggio sono nascosti nel file audio.

Per l'utilizzo possiamo far riferimento ai seguenti comandi:

- Il comando da lanciare per nascondere un file *secret.txt* in un file *audio.mp3* è il seguente:

```
./HideMeIn audio.mp3 secret.txt}
```

- Il comando da lanciare per estrarre i dati da un file *audioStego.mp3* è il seguente:

```
./HideMeIn audioStego.mp3 -f
```

HideMeIn è il nome del tool creato dopo aver compilato **AudioStego**.

Chapter 5

Implementazione

Abbiamo provato sia l'**Echo hiding** che l'**Amplitude hiding**. L'implementazione dell'algoritmo di Echo hiding è basata sul paper "*An Improvement for Hiding Data in Audio Using Echo Modulation*" [] di Huynh Ba Dieu.

Abbiamo realizzato un modulo **codificatore** e un modulo **decodificatore** utilizzando il linguaggio Python.

L'implementazione dell'algoritmo di Amplitude hiding è basata sul paper "*Robust and High-Quality Time-Domain Audio Watermarking Based on Low-Frequency Amplitude Modification*" [] di Wen-Nung Lie, Member e Li-Chun Chang.

Abbiamo modificato il compressore e il decompressore del progetto LAME, implementato in linguaggio C.

Dato il tentativo fallito abbiamo implementato lo stesso algoritmo anche in Python.

5.1 Echo hiding in Python

La nostra implementazione si basa su una libreria per la gestione e la manipolazione di dati audio chiamata *pydub* (disponibile al seguente URL: <http://pydub.com>), estremamente conveniente per il dominio applicativo.

5.1.1 Dipendenze

Le dipendenze necessarie ad eseguire il codice sono le seguenti:

```
sudo apt-get install python-dev python-setuptools python-pip
sudo pip install bitarray
sudo pip install pydub
sudo apt-get install ffmpeg
```

5.1.2 Codificatore

```
# Librerie
from bitarray import bitarray # struttura dati bitarray
import fileinput # read from file
from pydub import AudioSegment # lettura file mp3
import sys #Exit

# Numero di sample per ciascun frame
SAMPLES_PER_FRAME = 5000

# Inserisci il primo a (17): " or "17"
seed = raw_input("Inserisci il seed (123123): " or "123123"
messaggio = raw_input("Inserisci il messaggio da codificare ("
                      "Sternoclidomastoideo): " or "Sternoclidomastoideo"
messaggio_bitarray = bitarray() # Inizializzo array di bit vuoto
messaggio_bitarray.fromstring(messaggio) # Popolo array di bit con
                                         stringa in input

# Init
sample_da_nascondere = len(messaggio_bitarray) # grandezza
sequenza
#print "Pair inserito: (" + a + "," + seed + ")" # Stampiamo pair
ricevuto
a= int(a) # casting
seed = int(seed) # casting
xi = seed # seed(in fase di init)/numero precedente generato
R = bitarray(sample_da_nascondere) # Inizializzo array di bit R

def rng():
    global xi
    xi = (a*xi + 2*a)%sample_da_nascondere
    return xi

for i in range(sample_da_nascondere):
    xi_bis = rng() # Sequenza X generata
    if xi_bis % 6 > 2 :
        R[i] = True
```

CHAPTER 5. IMPLEMENTAZIONE

52

```

34     elif xi_bis % 6 <= 2 :
35         R[i] = False
36     #print "Array R"
37     #print R # Stampa array R
38
39     file_path = raw_input("Inserisci il nome del file MP3 (test.wav):
40                           ") or "test.wav"
41     # Start processing della canzone
42     fake_song = AudioSegment.empty();
43     original_song = AudioSegment.from_wav(file_path)
44
45     """ Costanti
46     Secondo i miei test, sono buoni valori per nascondere l'echo...
47     # -2db per l'echo rispetto all'originale
48     echo_loudness = -10
49     # applichiamo l'echo con delay di 50ms
50     echo_delay = 1
51
52     # Estrazione del numero di frame
53     number_of_frames_total = int(original_song.frame_count())
54     # Suddivisione in frame da 1024 sample
55     available_bits = number_of_frames_total/SAMPLES_PER_FRAME
56
57     # DEBUG PRINT
58     print ("Numero di frame disponibili: "), number_of_frames_total
59     print ("Numero di bit che si possono nascondere: "),
60     available_bits =
61
62     # Verifichiamo se il messaggio da nascondere supera il numero di
63     # frame disponibili
64     if(available_bits < sample_da_nascondere):
65         print ("Il numero di bit da nascondere risulta troppo elevato!")
66         sys.exit(0)
67
68     count = 0
69     for i in range(0,sample_da_nascondere):
70         # Get the slice
71         start_index = i * SAMPLES_PER_FRAME
72         end_index = (i+1) * SAMPLES_PER_FRAME
73         count += SAMPLES_PER_FRAME
74         #print "Start index:",start_index," End index:",end_index
75         porzione_eco = original_song.get_sample_slice(start_index,
76                                         end_index)
77         print ("Durata porzione_eco: "), porzione_eco.__len__(),"ms"
78         if(i < sample_da_nascondere):
79             if R[i]: #Ri contiene 1
80                 if messaggio_bitarray[i]:
81                     #copia
82                     print ("Copia samples")
83                     echoed = porzione_eco.get_array_of_samples();

```

CHAPTER 5. IMPLEMENTAZIONE

53

```
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
639
640
641
642
643
644
645
646
647
648
649
649
650
651
652
653
654
655
656
657
658
659
659
660
661
662
663
664
665
666
667
668
669
669
670
671
672
673
674
675
676
677
678
679
679
680
681
682
683
684
685
686
687
688
689
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
709
710
711
712
713
714
715
716
717
718
719
719
720
721
722
723
724
725
726
727
727
728
729
729
730
731
732
733
733
734
735
735
736
736
737
737
738
738
739
739
740
740
741
741
742
742
743
743
744
744
745
745
746
746
747
747
748
748
749
749
750
750
751
751
752
752
753
753
754
754
755
755
756
756
757
757
758
758
759
759
760
760
761
761
762
762
763
763
764
764
765
765
766
766
767
767
768
768
769
769
770
770
771
771
772
772
773
773
774
774
775
775
776
776
777
777
778
778
779
779
780
780
781
781
782
782
783
783
784
784
785
785
786
786
787
787
788
788
789
789
790
790
791
791
792
792
793
793
794
794
795
795
796
796
797
797
798
798
799
799
800
800
801
801
802
802
803
803
804
804
805
805
806
806
807
807
808
808
809
809
810
810
811
811
812
812
813
813
814
814
815
815
816
816
817
817
818
818
819
819
820
820
821
821
822
822
823
823
824
824
825
825
826
826
827
827
828
828
829
829
830
830
831
831
832
832
833
833
834
834
835
835
836
836
837
837
838
838
839
839
840
840
841
841
842
842
843
843
844
844
845
845
846
846
847
847
848
848
849
849
850
850
851
851
852
852
853
853
854
854
855
855
856
856
857
857
858
858
859
859
860
860
861
861
862
862
863
863
864
864
865
865
866
866
867
867
868
868
869
869
870
870
871
871
872
872
873
873
874
874
875
875
876
876
877
877
878
878
879
879
880
880
881
881
882
882
883
883
884
884
885
885
886
886
887
887
888
888
889
889
890
890
891
891
892
892
893
893
894
894
895
895
896
896
897
897
898
898
899
899
900
900
901
901
902
902
903
903
904
904
905
905
906
906
907
907
908
908
909
909
910
910
911
911
912
912
913
913
914
914
915
915
916
916
917
917
918
918
919
919
920
920
921
921
922
922
923
923
924
924
925
925
926
926
927
927
928
928
929
929
930
930
931
931
932
932
933
933
934
934
935
935
936
936
937
937
938
938
939
939
940
940
941
941
942
942
943
943
944
944
945
945
946
946
947
947
948
948
949
949
950
950
951
951
952
952
953
953
954
954
955
955
956
956
957
957
958
958
959
959
960
960
961
961
962
962
963
963
964
964
965
965
966
966
967
967
968
968
969
969
970
970
971
971
972
972
973
973
974
974
975
975
976
976
977
977
978
978
979
979
980
980
981
981
982
982
983
983
984
984
985
985
986
986
987
987
988
988
989
989
990
990
991
991
992
992
993
993
994
994
995
995
996
996
997
997
998
998
999
999
1000
1000
```

CHAPTER 5. IMPLEMENTAZIONE

54

```
121 # Export fake song  
122 fakesong.export("original.modified.wav", format="wav")  
123 # Export original song  
124 #original_song.export("original.mp3", format="mp3")  
code/main.py
```

5.1.3 Decodificatore

```
# Librerie  
from bitarray import bitarray # struttura dati bitarray  
import numpy  
import fileinput # read from file  
from pydub import AudioSegment # lettura file mp3  
import sys #exit  
  
# Numero di sample per ciascun frame  
SAMPLES_PER_FRAME = 5000  
  
# Inputs  
12 a = raw_input("Inserisci il primo a (17): ") or "17"  
13 seed = raw_input("Inserisci il seed (123123): ") or "123123"  
14 lenght = raw_input("Lunghezza messaggio nascosto (21): ") or 21  
15 source_file_path = raw_input("Inserisci il nome del file MP3  
16 source (original.wav): ") or "test.wav"  
17 encoded_file_path = raw_input("Inserisci il nome del file MP3  
18 source modificato (original_modified.wav): ") or "  
original_modified.wav"  
  
19 # Prendi file MP3 originale  
20 source_file = AudioSegment.from_wav(source_file_path)  
21 # Prendi file codificato  
22 encoded_file = AudioSegment.from_wav(encoded_file_path)  
  
23 source_samples = source_file.get_array_of_samples()  
24 encoded_samples = encoded_file.get_array_of_samples()  
25  
26 print "Source MP3: ", len(source_samples)  
27 print "Encoded MP3: ", len(encoded_samples)  
28  
# TODO: Confrontare SAMPLES_PER_FRAME alla volta  
# for i in range(0, len(source_samples), SAMPLES_PER_FRAME):  
# if(source_samples[i] != encoded_samples[i]):  
# print "ID ", i, " Original sample was ", source_samples[i]  
# print "ID ", i, " New sample is ", encoded_samples[i]
```

code/decoder.py

5.2 Amplitude data hiding in Lame

Abbiamo implementato lo schema esposto nel paper utilizzando il progetto LAME.

Lo schema di embedding è funzionante (se non si effettua compressione) poiché ogni in ogni test effettuato sono stati riestratti i dati nascosti. Effettuando embedding e compressione dei dati (su file MP3) i dati estratti risultano totalmente errati. Il codice del progetto è disponibile al seguente url <https://github.com/ProjectUnisa/AmplitudeHidingTest>.

Per eseguire la **decodifica** basta lanciare i seguenti comandi:

```
mv decode_frontend frontend
sudo ./configure
sudo make install
cd frontend
sudo make
./lame --decode -f test.mp3 or ./lame --decode -f cello.mp3
mv frontend decode_frontend
```

Per eseguire la **codifica** basta lanciare i seguenti comandi:

```
mv encode_frontend frontend
sudo ./configure
sudo make install
cd frontend
sudo make
./lame -s textToHiding -f test -e or ./lame -s textToHiding -f
      cello
mv frontend encode_frontend
```

5.2.1 LAME

LAME è l'acronimo ricorsivo di "LAME Ain't an MP3 Encoder" ed è un *encoder MPEG Audio Layer III (MP3)* sotto licenza **LGPL (open source)**. Il suo sviluppo ebbe inizio nel 1988 da **Mike Cheng**. Inizialmente non era un codificatore bensì una patch dimostrativa GPL che modificava l'originale codificatore *dist10*¹. Essendo una patch non era utilizzabile per produrre file MP3.

¹dist: è la qualità peggiore; file MP3 difettoso (tutti i blocchi audio sono marcati come difettosi)

Nel 2000 sono stati pubblicati gli ultimi sorgenti ISO rendendo LAME un vero e proprio codificatore, in grado di competere con i principali codificatori presenti sul mercato. Attualmente è molto utilizzato per le capacità di encoding in real time; in particolare è utilizzato da tools quali iRecordMusic e Audacity.

5.2.2 Risultati

Nell'algoritmo è stato utilizzato il parametro d (la frazione di $AOAA^2$) che muta a seconda di differenti attacchi come la compressione lossy.

5.3 Amplitude data hiding in Python

L'implementazione, come nel caso di Echo Data Hiding, si basa sulla libreria pydub.

5.3.1 Dipendenze

Le dipendenze necessarie per eseguire il codice sono le seguenti:

```
1 sudo apt-get install python-dev python-setuptools python-pip
2 sudo pip install bitarray
3 sudo pip install pydub
4 sudo apt-get install ffmpeg
```

5.3.2 Codificatore

```
# Libs
from pydub import AudioSegment # read file mp3
from utility import *
import time

# message = input("Inserisci il messaggio da codificare (A): ") or "A"
# Array di bits per il message
message_bits = to_bits(message)
# Numero di bit da nascondere
numero_bit_da_nascondere = len(message_bits)
# Stampa array di bit
print("Bit da nascondere: ", message_bits)
```

²Average of Absolute Amplitudes: la caratteristica principale di ogni sezione di sample. Ad ogni iterazione è suddivisa in 3 sezioni denotate con E1,E2,E3.

```
file_path = input("Inserisci il nome del file WW (cello.wav) ")  
# Start song processing  
original_song = AudioSegment.from_wav(file_path)  
# Numero di sample song extratti  
samples_number = original_song.frame_count()  
print("Numero di frames file WW: ", samples_number) # 1000000  
samples_int = split_channels(original_song.get_array_of_samples())  
samples = []  
for sample in samples_int:  
    samples.append(float(sample))  
samples_number = len(samples)  
# Total number of GOS  
number_of_gos = samples_number / SAMPLES_PER_GOS  
print("Numero totale di GOS: ", number_of_gos)  
# Get All samples  
for i in range(0, len(samples)):  
    if samples[i] == 0:  
        print("Indice 0 ", i)  
        exit(-1)  
counter = 0  
  
# Estraiamo esclusivamente i GOS necessari per nascondere il  
messaggio  
for i in range(0, numero_bit_da_nascondere):  
    e_values = extract_e(counter, i, samples)  
    # Riassegnazione variabili (per readability)  
    Emin = e_values[0][0]  
    Emid = e_values[1][0]  
    Emax = e_values[2][0]  
  
    threshold = (Emax + (2 * Emid) + Emin) * d  
    print("***** GOS number ", i, "*****")  
    while(1):  
        # Estraiamo e_values già ordinati  
        # e_values[0]: ( Emin, start_pos, end_pos )
```

```

60      # e_values[1]: ( Emid, start_pos, end_pos )
62      # e_values[2]: ( Emax, start_pos, end_pos )
63      e_values = extract_e(counter, i, samples)
64
65      # Riassegnazione variabili (per readability)
66      Emin = e_values[0][0]
67      Emid = e_values[1][0]
68      Emax = e_values[2][0]
69
70      # Determiniamo A e B
71      A = Emax - Emid
72      B = Emid - Emin
73
74      if message_bits[i] == 1: # Devo nascondere 1
75          subtraction = A - B
76          # Indici per l'incremento/decremento
77          start_increase_index = e_values[2][1]
78          end_increase_index = e_values[2][2]
79          start_decrease_index = e_values[1][1]
80          end_decrease_index = e_values[1][2]
81      else: # Devo nascondere 0
82          subtraction = B - A
83          # Indici per l'incremento/decremento
84          start_increase_index = e_values[1][1]
85          end_increase_index = e_values[1][2]
86          start_decrease_index = e_values[0][1]
87          end_decrease_index = e_values[0][2]
88
89          # Calcola delta
90          delta = (threshold - subtraction) / 3
91          print("Soglia:", threshold, " Subtraction:", subtraction)
92          if (subtraction >= threshold): # esci
93              break
94          else: # continua ad incrementare
95              print("*** RIPETO INCREMENTO ***")
96              if (message_bits[i] == 1):
97                  Emax += delta
98                  Emid -= delta
99                  omega_up = 1 + (delta / Emax)
100                 omega_down = 1 - (delta / Emid)
101                 print("Nasconde 1 emax:", Emax, " emid ", Emid, " "
102                     omega_up, " omega_up ", "omega_down ", omega_down)
103             else:
104                 Emid += delta
105                 Emin -= delta
106                 omega_up = 1 + (delta / Emid)
107                 omega_down = 1 - (delta / Emin)
108                 print("Nasconde 0 emid:", Emid, " emin ", Emin, " "
109                     omega_up, " omega_up ", "omega_down ", omega_down)

```

```

106     # Incremento il valore del sample
107     for to_increase in range(start_increase_index,
108         end_increase_index):
109         # print("Prima ", samples[to_increase])
110         samples[to_increase] = samples[to_increase] *
111             omega_up
112         # print("Dopo ", samples[to_increase])
113         # Decremento il valore del sample
114         for to_decrease in range(start_decrease_index,
115             end_decrease_index):
116             samples[to_decrease] = samples[to_decrease] *
117                 omega_down
118         # Passa al GOS successivo
119         counter += SAMPLES_PER_GOS
120     # Sostituisce i samples modificati
121     original_song = original_song.spawn(mix_channels(samples,
122         split_channels(original_song.get_array_of_samples())[1]))
123     # Compressa
124     original_song.export("output_with_data.mp3", format="mp3")
125     # Non compresso
126     original_song.export("output_with_data.wav", format="wav")

```

code/compress.py

5.3.3 Decodificatore

```

1 # Libs
2 from pydub import AudioSegment
3 from utility import *
4
5 # Path file stego
6 file_path = input("Inserisci il nome del file MP3 ("
7     "output_with_data.mp3): ") or "output_with_data.mp3"
8
9 # Open file stego
10 original_song = AudioSegment.from_mp3(file_path)
11
12 left_right_array = original_song.split_to_mono() # separazione in
13     due flussi diversi
14 left_array = left_right_array[0] # left channel
15
16 left_samples_number = original_song.frame_count()
17 print("Numero di frames file MP3 (LEFT CHANNEL): ",
18     left_samples_number)
19
20 # Total number of GOS
21 number_of_gos = left_samples_number / SAMPLES_PER_GOS
22 print("Numero totale di GOS: ", number_of_gos)

```

```
21 # Get samples of the left channel
22 samples_left = left_array.get_array_of_samples()
23 counter = 0
24 bit_8_counter = 0
25 for i in range(0, int(number_of_gos)):
26     # Estraiamo e-values già ordinati
27     # e-values[0]: Emin
28     # e-values[1]: Emid
29     # e-values[2]: Emax
30     e_values = extract_e(counter, i, samples_left)
31
32     # Riassegnazione variabili (per readability)
33     Emin = e_values[0][0]
34     Emid = e_values[1][0]
35     Emax = e_values[2][0]
36
37     # Determiniamo A e B
38     A = Emax - Emid
39     B = Emid - Emin
40
41     if A >= B:
42         print("1", end="")
43     if B > A:
44         print("0", end="")
45
46     counter += SAMPLES_PER_GOS
47
48     if bit_8_counter == 7:
49         bit_8_counter = 0
50         print("")
51     else:
52         bit_8_counter += 1
```

code/extractionData.py

Chapter 6

Conclusioni

Differenti studi hanno dimostrato che la riproduzione e la copia di dati audio digitali non compromette l'integrità degli stessi; ciò ha comportato l'ideazione di tecniche che nascondono informazioni sensibili proprio nei dati audio.

Il lavoro proposto da W.N.Lie et al [], sul quale ci siamo basati, illustra una tecnica tra le molteplici.

Tale lavoro consiste nella definizione di un metodo di audio watermarking, basato sulla modifica dell'ampiezza, che consente di mantenere un'alta qualità del suono, anche a seguito di attacchi quali ad esempio *low-pass filtering*, *time scaling*, etc. L'informazione è nascosta nel segnale audio nel dominio del tempo. Ogni bit è nascosto modificando le differenze di "average-of-absolute-amplitude" (AOAA) suddivise in 3 sezioni e raggruppate in GOS (gruppi di samples).

Da uno studio del 2008 [], due anni dopo la pubblicazione del lavoro [], è stato dimostrato che utilizzando dei GOS di dimensione piccola, il metodo proposto non è in grado di resistere alla compressione MP3 e che quindi le informazioni nascoste non vengono estratte correttamente dopo la decompressione MP3: allo stesso modo aumentando la dimensione del GOS la quantità di informazione memorizzabile diminuisce drasticamente.

Da quanto descritto nel paper [] sono state effettuate diverse prove con GOS di dimensioni differenti: sono partiti da 200 e, incrementando il numero di 50 GOS alla volta, si sono fermati a 600; hanno infine affermato che la dimensione più accettabile era 300. Nella nostra implementazione abbiamo replicato tali esperimenti: siamo partiti da 200 ma ci siamo fermati a 300 poiché il metodo

non si è rivelato robusto utilizzando la compressione MP3.

L'implementazione utilizzata nel paper [] è Lame 3.93 con bitrate 128kbps, da 30 Settembre 2001 considerato il miglior codificatore per il bitrate indicato. Abbiamo implementato la funzione *amplitude_hiding*.

L'encoder utilizza la funzione *lame_encoder* che carica le informazioni del file da codificare e invoca la funzione *lame_encoder_loop*. Tale funzione apre il file PCM, legge le informazioni relative ai tag *id3v2* e carica i sample in una variabile *ircad* utilizzando la funzione *get_audio*. Tale funzione prende i frame audio dal file PCM e li inserisce in un *buffer* separando i due canali: *right* e *left*. La variabile utilizzata è *Buffer[2][1152]*. La chiamata successiva è al metodo *lame_encode_buffer_int* che prende in input il buffer e lo codifica. Abbiamo inserito il nostro metodo (*amplitude_hiding*) prima di tale chiamata ma non abbiamo ottenuto gli obiettivi prefissati.

In alcune prove abbiamo verificato il contenuto del buffer prima e dopo la chiamata a *lame_encode_buffer_int*, riscontrando delle variazioni nei dati. Abbiamo identificato in tale chiamata la fase di compressione di tale encoder.

Abbiamo infine approfondito le operazioni svolte in *lame_encode_buffer_int*, comprendendo che tale funzione prende in input il buffer di interi e lo trasforma in un buffer del tipo *sample_t*, applicando il seguente filtro:

```
buffer[0][i] = buffer_l[i] * (1.0 / (1L << (8 * sizeof(int) - 16)))
buffer[1][i] = buffer_r[i] * (1.0 / (1L << (8 * sizeof(int) - 16)))
```

Successivamente è chiamata la funzione *lame_encode_mp3_frame* che, prendendo in input il buffer del tipo *sample_t*, si occupa della codifica dei singoli frame del file audio finale.

Tale funzione è scomposta in cinque fasi:

1. Applicazione del **Modello psicoacustico**;
2. Applicazione della **MDCT**;
3. Applicazione della modalità: MS sta per *MID/SIDE stereo*, LR sta per *Left/Right stereo*;
4. Quantization loop;

CHAPTER 6. CONCLUSIONI

63

5. Formattazione del bitstream: i frame vengono scritti nel bitstream e successivamente copiati in un array di *int*, utilizzati per il salvataggio nel file MP3 finale.

Nella nostra implementazione il buffer di sample viene modificato subito dopo averlo letto dal file in PCM, prima della sua trasformazione nel tipo *sample_t*; tale strategia potrebbe essere un motivo del fallimento dell'estrazione dell'informazione nascosta.

Bibliography

- [1] J. Ba'tora, "Deepsound," 2015. Disponibile online: <http://jpinsoft.net/deepsound>.
- [2] F. Petitcolas, "Mp3stego," 2006. Disponibile online: <http://www.petitcolas.net/steganography/mp3stego/>.
- [3] S. Hetzl, "Steghide," 2003. Disponibile online: <http://steghide.sourceforge.net>.
- [4] C. S. Solution, "Quickstego," 2005. Disponibile online: <http://quickcrypto.com/products/QS12Setup.zip>.
- [5] D. Cardeenas, "Audiotstego," 2014. Disponibile online: <https://github.com/danielcardeenas/AudioStego>.
- [6] H. B. Dieu, "An improvement for hiding data in audio using echo modulation," *ICIEIS*, 2013.
- [7] M. Wen-Nung Lie and L.-C. Chang, "Robust and high-quality time-domain audio watermarking based on low-frequency amplitude modification," *IEEE TRANSACTIONS ON MULTIMEDIA*, no. 1, p. 46, 2006.
- [8] A. S. Harumi Murata. Akio Ogihara, Motoi Iwata, "Rmultiple embedding for time-domain audio watermarking based on low- frequency amplitude modification," *ITC-CSCC 2008*, 2008.