

Multimedia Data Processing

Teoria basata sulle lezioni
dell'A.A. 2020/2021

Bernardi Mattia

Indice

1 Data Compression	1
1.1 Terminologia	1
1.2 Teoria dell'Informazione	1
1.2.1 Entropia e Compressione	2
1.3 Unary Coding	3
1.4 Elias γ Coding	3
1.5 Huffman Encoding	4
1.6 Run Length Encoding	5
1.6.1 Packbits	5
Algoritmo	5
1.7 Lempel-Ziv	6
1.7.1 LZ77	6
LZSS: Variante di LZ77	7
1.7.2 LZ78	7
LZW	8
Caso Speciale	9
2 Image Capture	11
2.1 Light Formulas	11
2.2 Color Representation	12
2.2.1 CIEXYZ Color Space	12
2.2.2 CIELAB Color Space	12
2.2.3 Color Reproduction	12
2.2.4 CRT Monitor	12
Gamma Correction	13
2.2.5 RGB Color Space al PC	13
2.2.6 Transmissive Color Spaces	13
YC _B C _R Standard	13
2.2.7 Ordered Space	14
HSV Color Space	14
HLS/HSL Color Space	15
2.2.8 Chromatic Quantization	15
Palette Selection	15
2.3 Hardware per l'Acquisizione	15
2.3.1 Dichroic & Trichroic Prism	16
2.3.2 Bayer Pattern	16
Demosaicing	16
Nearest Neighbor	16
Bilinear Interpolation	17
Laplacian Correction	17
3 JPEG Compression	19
3.1 Baseline JPEG	19
3.1.1 DCT: Discrete Cosine Transform	20
3.1.2 Quantization	21
3.1.3 DC Encoding	21
3.1.4 AC Encoding	23
3.1.5 Color Management	23

4 Video Compression	24
4.1 INTRA Frame Encoding	24
4.2 INTER Frame Encoding	24
4.2.1 Time Difference	24
4.2.2 The Motion	24
Motion Vector	25
SAD: Sum of Absolute Difference	25
Lossy Compression	25
5 H.261 Standard	28
5.1 Bitstream Structure	29
5.1.1 Picture Layer	29
5.1.2 Group of Blocks Layer	29
5.1.3 Macroblock Layer	30
5.1.4 Block Layer	33
5.1.5 Quantization	34
6 Other Video Standards	35
6.1 MPEG Standard	35
6.1.1 Differences with H.261	35
6.1.2 MPEG Novelties	36
Video Layer and B-Frames	36
Bidirectional Prediction in B-Frames	37
6.1.3 More Differences With H.261	37
6.2 H.264/AVC	37
6.2.1 Prediction fro INTRA Pictures	38
6.2.2 Variable Block Size	38
6.2.3 Different DCT for Smaller Size	39
7 The Sound	40
7.1 Sound Waves	40
7.1.1 Sound Properties	41
Frequency	41
Wave Length	41
Phase	42
7.2 Sound Pressure	42
7.2.1 Microphones	42
Electrodynamic Microphone	42
Condenser Microphone	43
Polar Patterns	43
Microphones Parameters	44
7.2.2 Sound Reproduction	44
7.3 Digital Audio	44
8 Sound Perception	46
8.1 The Human Ear	46
8.1.1 Outer Ear	47
8.1.2 Middle Ear	47
8.1.3 Inner Ear	47
Critical Band	47
8.2 Anechoic Chamber	47
8.2.1 Isophonic Curves	47
8.3 Masking	48
8.3.1 Temporal Masking	49
9 AAC	50
9.1 Perceptual Audio Coding	50
9.1.1 Sub-Band Processing	51
9.1.2 Perceptual Model	52
Absolute Threshold of Hearing	52
Tone-Masking-Noise / Noise-Masking-Tone	53

10 Containers	54
10.1 Extensible Binary Meta Language	54
10.1.1 EBML Element	54
10.2 Matroska	55
10.2.1 Segment	55
SeekHead	55
Tracks	56
Block	56

Capitolo 1

Data Compression

La quantità di dati con cui interagiamo ogni giorno è in continua crescita e al contempo i nostri sistemi di comunicazione non riescono a stare al passo con questo aumento esponenziale; abbiamo bisogno di modi per ridurre tale mole di dati: abbiamo bisogno di effettuare una compressione. Ai giorni d'oggi qualsiasi cosa è compressa, sarebbe impossibile trasferire tutti i dati che produciamo senza alcuna forma di compressione.

1.1 Terminologia

Per poter parlare di compressione, iniziamo dando qualche definizione che ci sarà utile in seguito:

- **Messaggio:** qualsiasi tipo di informazione che vogliamo recapitare, trasferire, a qualcun altro (una canzone, un suono, un'immagine o un video). Lo stesso messaggio può essere trasferito in molte maniere.
- **Simboli:** è l'insieme di elementi base che compongono il messaggio (pixel, lettere, bit, campioni audio, etc...). È importante notare che diverse sequenze di simboli possono rappresentare lo stesso messaggio.
- **Encoding:** Consideriamo un "Encoding" la rappresentazione tramite bit, sequenze di 0 e 1, di un qualsiasi messaggio avente dimensioni inferiori a quella iniziale.
- **Compressione:** Con compressione ci riferiamo a tutte le operazioni necessarie per ottenere l'encoding di un messaggio di dimensioni possibilmente inferiori all'originale.
- **Decoding:** È l'operazione inversa dell'encoding; possiamo avere due tipi di decoding, *lossless*, ossia senza perdite, oppure *lossy*, ossia un'approssimazione con perdita.

Detto questo è importante capire che la compressione perfetta non esiste, ossia non è possibile trovare un algoritmo che sia in grado di comprimere qualsiasi tipo di dato perfettamente, altrimenti arriveremmo all'assurdo di poter applicare lo stesso algoritmo in maniera iterativa fino ad ottenere un encoding di 0 bit. Come viene effettuata dunque la compressione? L'intero processo si basa su un modello che descrive la probabilità di un messaggio di essere o meno presente, per poi effettuare l'encoding di messaggi meno probabili con più bit e quelli più probabili con meno.

L'**Encoder** è dunque l'algoritmo che sceglie come rappresentare i messaggi.

1.2 Teoria dell'Informazione

Tutto ha inizio dalla teoria dell'informazione, uno dei concetti fondamentali di questo campo è che non può esistere informazione senza incertezza, siccome se sappiamo già tutto allora niente può essere considerato come informazione; da qui dunque diciamo che la quantità di informazione è direttamente collegata alla sua probabilità di accadere: più è probabile e meno informazione ho in modo da ottimizzare il linguaggio per esprimere meno energia per ciò che vogliamo.

Claude Elwood Shannon è stato uno dei pionieri in questo campo, formalizzando una misura capace di rappresentare la quantità di incertezza di un dato set di simboli S :

$$H(S) = \sum_{x \in S} p(x) \log_2 \frac{1}{p(x)} = -\sum_{x \in S} p(x) \log_2 p(x)$$

(Il meno nella formula è per compensare quello che viene fuori dal logaritmo del rapporto).

In cui $H(S)$ è l'incertezza di un set di simboli ed è la somma dell'incertezza di ogni simbolo (trattato separatamente); più alta la probabilità e più importante è ma siccome accade spesso voglio usare meno bit dato che la quantità di informazione è inferiore.

Esempio 1.2.1. Consideriamo l'insieme di simboli $S = \{a, b, c, d\}$, in cui ogni simbolo è caratterizzato dalle seguenti probabilità:

$$p(a) = 0.4 \quad p(b) = 0.3 \quad p(c) = 0.2 \quad p(d) = 0.1$$

Consideriamo inoltre che siccome abbiamo 4 simboli ci servono almeno 2 bit per simbolo, considerando un encoding binario, nella seguente maniera:

$$a = 00, b = 01, c = 10, d = 11$$

Per ogni simbolo andiamo ora a calcolare $H(x)$, sommando tutto insieme otteniamo:

$$H(S) = 1.846439345$$

Ora, siccome stiamo utilizzando 2 bit per simbolo, diciamo che in media stiamo spendendo 2 bit per simbolo, la quantità di informazione calcolata è un numero leggermente inferiore a 2, questo ci dice che in questo caso, con un encoding binario, stiamo sprecando $2 - H(S)$ bit.

1.2.1 Entropia e Compressione

A seguito del suggerimento di Von Neumann, e data la sua analogia con l'entropia statistica, Shannon è arrivato a chiamare la quantità $H(S)$ proprio **Entropia**.

Facciamo un passo indietro e cerchiamo di capire come rappresentare un encoding:

$$C_s = \{(a, 00); (b, 01); (c, 10); (d, 11)\}$$

Ecco questo è un encoding, associare un codice ad ogni simbolo, in questo caso le coppie formate sono del tipo *simbolo - codice binario*. (Questo tipo di encoding viene chiamato encoding binario)

Dato un encoding possiamo dunque calcolarne la lunghezza media come:

$$\bar{L}(C_s) = \sum_{(x,w) \in C_s} p(x)L(w)$$

In cui $L(w)$ indica la lunghezza di ogni parola, nel nostro caso binario pari a 2. Notiamo che se cambiamo l'encoding nel seguente modo:

$$C_s = \{(a, 0); (b, 10); (c, 110); (d, 111)\}$$

E considerando sempre le probabilità dell'Esempio 1.2.1, vediamo che il simbolo a è il più probabile, otteniamo quindi una lunghezza media di 1,9; la domanda che ci poniamo ora è: fino a che punto possiamo manipolare la lunghezza medio del nostro encoding? Shannon ha dimostrato che:

$$H(S) \leq \bar{L}(C_s), \forall C_s$$

Non è dunque possibile ottenere una lunghezza media che sia inferiore dell'entropia, o meglio: l'unico modo per andare al di sotto della misura $H(S)$ è cambiare $H(S)$ stesso, ossia dobbiamo cercare di esprimere l'informazione con un diverso insieme di simboli.

1.3 Unary Coding

Lo *Unary Coding* è un primo sistema di encoding che prevede di inserire tanti 1 quanto è il numero da codificare meno uno e terminare tale numero con uno 0.

Esempio 1.3.1. Nella seguente tabella è riportato un esempio:

Simbolo	Coding	Duale
1	0	1
2	10	01
3	110	001
4	1110	0001
5	11110	00001
...

Questo sistema non è però, ovviamente, molto intelligente ed efficace, siccome avendo ogni codice che inizia con 1, non riusciamo ad ottenere alcuna informazione dal primo bit. Possiamo dunque chiederci se è un tipo di encoding ottimo? Per dire ciò dobbiamo per prima cosa correggere la domanda: lo unary coding è ottimo dato uno specifico set di simboli con una specifica distribuzione di probabilità?

$$\bar{L}(C_s) = H(S)$$

Se questa uguaglianza viene soddisfatta allora possiamo confermare l'ottimalità dell'encoding per il set di simboli S . Infatti in base alla distribuzione di probabilità dei simboli la qualità dell'encoding può cambiare: qual'è dunque la distribuzione che rende il coding unario ottimo? Partendo dalla relazione scritta sopra ed esplicitando ambo i termini:

$$\sum_{i=1}^{+\infty} p_i L_i = \sum_{i=1}^{+\infty} p_i \log_2 \frac{1}{p_i}$$

Nel caso unario:

$$L_i = i$$

E duqnue:

$$\sum_{i=1}^{+\infty} p_i \cdot i = \sum_{i=1}^{+\infty} p_i \log_2 \frac{1}{p_i}$$

Da qui, tramite semplici passaggi, possiamo ricavare p_i :

$$i = \log_2 \frac{1}{p_i} \rightarrow 2^i = \frac{1}{p_i} \rightarrow p_i = \frac{1}{2^i}$$

Abbiamo quindi trovato la distribuzione dei simboli secondo la quale il nostro encoding risulterà ottimo.

1.4 Elias γ Coding

È un altro tipo di encoding nel quale continuiamo a supporre che i numeri più piccoli siano più probabili di quelli grandi e basandoci su questa assunzione possiamo dividerli in classi che indicano il numero di bit necessari a rappresentare un numero in binario, da qui aggiungiamo un numero di 0 in testa nel seguente modo, dato un naturale da codificare:

$$x \in \mathbb{N} = \{1, 2, 3, \dots\}$$

Scriviamo la sua rappresentazione binaria preceduta da: $\lfloor \log_2 x \rfloor$ zeri. (Notiamo che $\lfloor \log_2 x + 1 \rfloor$ è il numero di cifre necessarie per scrivere il numero in binario)

Esempio 1.4.1. Nella seguente tabella è riportato un esempio:

Numero	Coding	Duale
1	1	0
2	010	101
3	011	100
4	00100	11011
5	00101	11010
6	00110	11001
7	00111	11000
8	0001000	1110111
9	0001001	1110110
...

In pratica nel codice il numero di 0 corrisponde alla classe e ciò che segue è quello che in binario identifica il numero stesso. Ma quindi quando usiamo questo tipo di encoding? Quando numeri con un basso valore assoluto hanno più probabilità di verificarsi di numeri con un grande valore assoluto. E per i numeri negativi? Come possiamo gestire questa rappresentazione? Ad esempio:

$$map(x) = \begin{cases} 1 - 2x & x \leq 0 \\ 2x & x > 0 \end{cases}$$

1.5 Huffman Encoding

L'algoritmo di Huffman permette di creare un encoding a lunghezza variabile ottimo, ossia non esistono altri algoritmi che consentono di ottenere una lunghezza media inferiore; con lunghezza variabile si intende che la lunghezza del codice per ogni simbolo cambia in base alla probabilità del simbolo: simboli più probabili saranno più corti e simboli meno probabili saranno più lunghi. (Teniamo in considerazione anche che i codici devono poter essere distinti in sequenza senza alcun problema)

Vediamo i passaggi che compongono tale algoritmo:

1. Per prima cosa dobbiamo conoscere la prior di ogni simbolo ed ordinarli in maniera crescente.
2. Da qui prendiamo gli ultimi due della lista e li uniamo, sommando anche la loro probabilità.
3. Ripartiamo dal punto 1 e continuiamo così fino ad ottenere l'insieme di simboli con probabilità 1.
4. Una volta che ho l'intero set di simboli raggruppato inizio con la codifica all'indietro.

Arriviamo ad avere dunque una tabella contenente tutte le informazioni a noi necessarie per effettuare sia encoding che decoding: simbolo, lunghezza, codice; infatti anche nel processo inverso abbiamo bisogno della tabella creata in precedenza in modo da poter associare ad ogni sequenza di bit il corrispondente simbolo.

Notiamo però che, dato lo stesso albero, esistono diversi tipi di encoding possibili, abbiamo la possibilità di ottenerne uno canonico? Certo! Tra tutti quelli a nostra disposizione ne possiamo estrarre due, primale e duale, che possono essere ottenuti solamente dalla lunghezza del codice associato al simbolo dato che con questa informazione possiamo individuare a quale livello dell'albero siamo per poi ricostruirlo.

Vediamo i passaggi:

1. Partiamo con 0 o 1, in base a quale delle due forme stiamo usando.
2. Teniamo conto della lunghezza e del valore attuale, 0 e 0 inizialmente ad esempio.
3. Vediamo il primo simbolo che ha un codice di una certa lunghezza, per esempio 2.
4. Siccome $0 < 2$ in questo caso, dobbiamo aumentare la nostra lunghezza attuale, siccome 1 non basta andiamo direttamente a 2.

5. Da qui dobbiamo scrivere il valore attuale, 0, con 2 bit \rightarrow 00.
6. Ora dobbiamo aumentare il valore attuale, andiamo ad 1.
7. La lunghezza del prossimo simbolo è $3 > 2$, aumentiamo la lunghezza attuale e moltiplichiamo il valore per 2 dopo aver somato 1 $\rightarrow (0+1) \cdot 2$, scrivendo la sua rappresentazione binaria con tanti bit quanto è la lunghezza.

Esempio 1.5.1. Nella seguente tabella è riportato un esempio:

Lunghezza Codice	Lunghezza Attuale	Valore Attuale	Encoding
3	3	0	000
3	3	1	001
3	3	2	010
4	4	6	0110
4	4	7	0111
4	4	8	1000
4	4	9	1001
4	4	10	1010
4	4	11	1011
5	5	24	11000
...

1.6 Run Length Encoding

Il Run Length Encoding, o RLE, è un sistema di compressione che si basa sul contesto, si occupa di effettuare l'encoding di sequenze di simboli uguali, chiamate **run**; non funziona molto bene con testi, in quanto la probabilità di trovare lunghe sequenze di caratteri uguali uno dopo l'altro è pressoché nulla, se si parla invece di immagini in bianco e nero questa eventualità è notevolmente più probabile. Tuttavia ci sono decisioni da prendere: effettuiamo l'encoding in qualsiasi caso? Come distinguiamo il numero dei simboli dai simboli stessi e così via...

Esempio 1.6.1. Consideriamo la sequenza:

*aaaabbccccc**aaaa*

Tramite questo tipo di compressione possiamo tradurla in:

$(4, a)$ $(3, b)$ $(5, c)$ $(4, a)$.

1.6.1 Packbits

Questo algoritmo si basa sulla codifica di **run**, in cui viene effettuata una codifica, e **copy**, in cui nessuna codifica viene effettuata.

Algoritmo. Il sistema è costruito intorno a un byte di comando L detto *lunghezza*:

if: $0 \leq L \leq 127$

Siamo di fronte ad una **copy**, ossia sto dicendo di copiare i successivi $L + 1$ byte così come sono, senza cambiare nulla.

if: $129 \leq L \leq 255$

È l'inizio di una **run**, cioè ripeterò il prossimo byte per $257 - L$ volte.

C'è poi il caso in cui $L = 128$ che indica *EOD*.

Esempio 1.6.2. Consideriamo la sequenza:

aaaabbbbbbcdefgaaabbbabcd

Con Packbits otteniamo il seguente risultato

253, $a - 251$, $b - 4$, $cdefg - 254$, $a - 254$, $b - 3$, $abcd - 128$

Ovviamente scriveremo tutto in byte, di seguito senza alcun separatore.

1.7 Lempel-Ziv

È una famiglia di algoritmi che ha come obiettivo non quello di creare un encoding dei dati, ma di cambiare il set di simboli, ossia ci darà in output un nuovo set di simboli. Partiamo dalla prima variante:

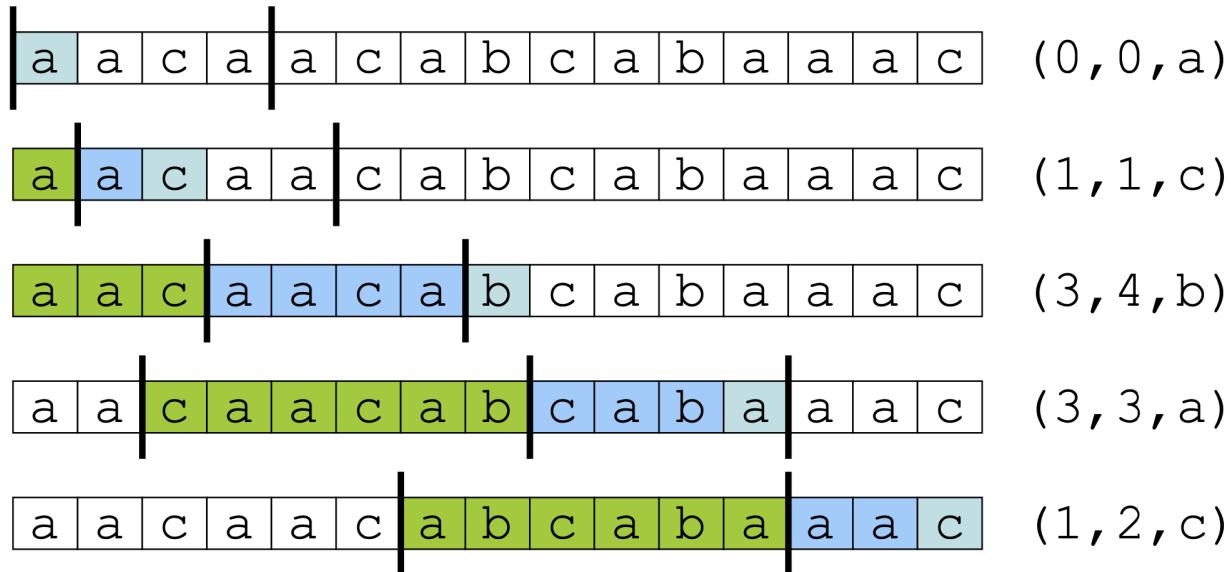
1.7.1 LZ77

Questo algoritmo è costruito sul concetto di *sliding window*: abbiamo un cursore che scorre sui nostri dati, la parte che è già stata codificata viene usata come dizionario per effettuare la codifica dei dati successivi; questo viene effettuato creando una tripletta (**P**, **L**, **C**) ad ogni passo:

- **P**: È la distanza all'indietro rispetto alla corrispondenza più lunga.
- **L**: È la lunghezza effettiva della corrispondenza.
- **C**: È il carattere da mandare una volta che la corrispondenza è finita.

(Dopo ogni step muoviamo il cursore $L + 1$ spazi in avanti)

Esempio 1.7.1. Vediamo un esempio di encoding:



In questo esempio abbiamo un buffer di 4 elementi ed un dizionario di dimensione massima 6.

Procediamo per righe:

1. Il primo carattere individuato è una *a*, guardiamo indietro ma per ora il dizionario è vuoto siccome siamo all'inizio, quindi non abbiamo alcuna corrispondenza. Di conseguenza la tripletta sarà (**P**: 0, **L**: 0, **C**: *a*). A questo punto il carattere *a* viene aggiunto al dizionario e muoviamo il cursore(buffer) di $0 + 1$ posti in avanti.
2. Nel buffer ora abbiamo *a|c|a|a* e nel dizionario abbiamo *a*. A questo punto abbiamo una corrispondenza tra il primo carattere del buffer ed il dizionario, di conseguenza indicherò di quanto devo andare indietro per trovare la corrispondenza e la lunghezza di quest'ultima; dopodiché scriviamo il prossimo carattere, ossia *c* → (**P**: 1, **L**: 1, **C**: *c*).
3. Quando raggiungiamo la lunghezza massima del dizionario, durante lo step di update quello che facciamo è scartare i primi valori per aggiungere quelli nuovi.

Durante il decoding, non abbiamo nessuna necessità di avere il dizionario siccome esso viene ricreato in modo dinamico dalle triplettie inviate.

Come mai questo algoritmo funziona? È stato dimostrato che LZ77 è asintoticamente ottimo, ossia che con stringhe lunghe abbastanza riusciamo a raggiungere il limite posto dall'entropia del messaggio quando la finestra di ricerca tende all'infinito:

$$H_n = \sum_{X \in A^n} p(X) \log \frac{1}{p(X)} \quad \rightarrow \quad H = \lim_{n \rightarrow \infty} H_n$$

Il problema è che quando abbiamo detto "abbastanza lunghe", intendiamo veramente tanto lunghe, computazionalmente non trattabile.

LZSS: Variante di LZ77

In questa variante, adottata da *gzip*, abbiamo due possibilità:

- (**0, posizione, lunghezza**) quando abbiamo una lunghezza superiore a 3.
- (**1, carattere**) quando abbiamo una lunghezza inferiore a 3.

In questa versione, posizione, lunghezza e carattere sono codificati tramite Huffman con l'introduzione di alcune regole speciali per migliorare la compressione.

1.7.2 LZ78

Nuova versione rispetto a LZ77, siccome richiedeva di cercare una sequenza all'interno di un'altra sequenza risultava lento; per questo motivo LZ78 non usa una sliding window ma un vero e proprio dizionario:

- Cerchiamo la stringa **S** più lunga all'interno del nostro dizionario.
- Mandiamo la corrispondenza seguita dal prossimo carattere **c**.
- Aggiungiamo la nuova stringa **Sc** al dizionario.

Esempio 1.7.2. Vediamo un esempio di encoding:

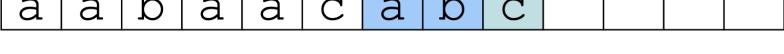
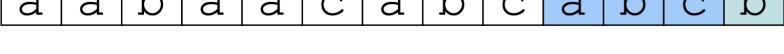
	Output	Dictionary
a a b a a c a b c a b c b	(0, a)	1 = a
a a b a a c a b c a b c b	(1, b)	2 = ab
a a b a a c a b c a b c b	(1, a)	3 = aa
a a b a a c a b c a b c b	(0, c)	4 = c
a a b a a c a b c a b c b	(2, c)	5 = abc
a a b a a c a b c a b c b	(5, b)	6 = abcb

Procediamo per righe:

1. Partiamo all'inizio leggendo il carattere **a**, ovviamente il dizionario per ora è vuoto e quindi il primo carattere sarà 0, mentre in questo caso **c: a** che verrà aggiunto poi al dizionario.
2. Il secondo carattere è ancora **a**, bene stavolta ce lo abbiamo, guardiamo avanti e abbiamo poi una **b**, questo non lo abbiamo quindi per la **a** usiamo il dizionario alla posizione 1, **S: a**, e **c: b** e duqne aggiungeremo **Sc: ab** al dizionario.

Per quando riguarda il decoding si utilizza lo stesso metodo per costruire il dizionario, per cui non c'è alcun bisogno di mandarlo siccome può essere ricostruito dalle tuple

Esempio 1.7.3. Vediamo un esempio di decoding:

Input	Dictionary
(0, a) 	1 = a
(1, b) 	2 = ab
(1, a) 	3 = aa
(0, c) 	4 = c
(2, c) 	5 = abc
(5, b) 	6 = abcb

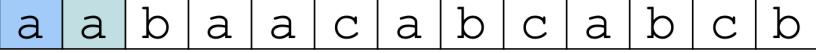
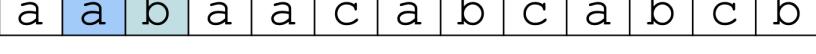
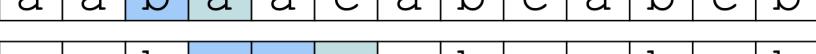
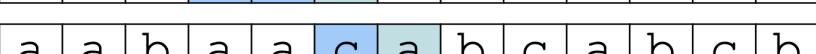
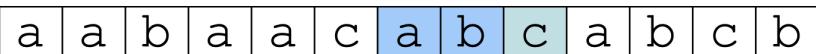
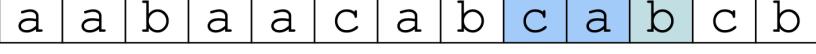
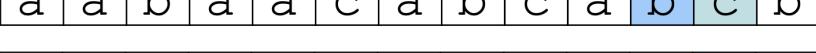
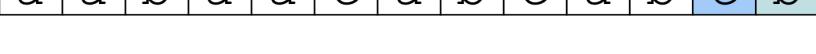
Procediamo per righe:

1. Riceviamo la coppia **(0, a)**, 0 indica una pagina vuota del dizionario, scriviamo il carattere *a* e lo aggiungiamo.
2. Abbiamo ora ricevuto **(1, b)**, ossia ci sta dicendo di prendere il carattere dal dizionario alla posizione 1 e aggiungere una *b*, per poi andare ad inserire la nuova sequenza.

LZW

È un'evoluzione di LZ78 nella quale possiamo evitare di mandare il carattere *c* extra, questo viene fatto tramite l'inizializzazione di un dizionario contenente tutti i valori da 0 a 255 e siccome questa inizializzazione è di default non abbiamo bisogno di trasmettere alcuna informazione aggiuntiva. Dopodiché andiamo a creare le sequenze alle posizioni > 255:

Esempio 1.7.4. Vediamo un esempio di encoding:

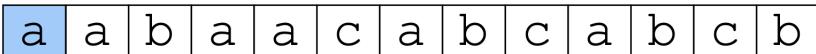
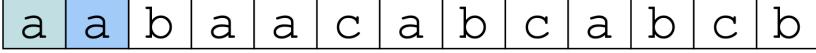
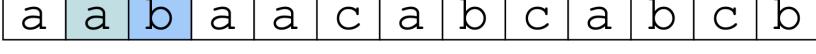
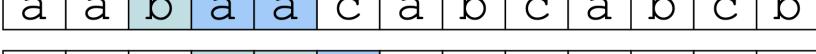
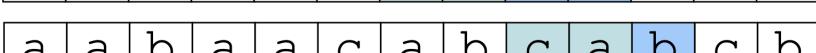
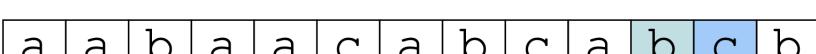
Output	Dictionary
	97 256=aa
	97 257=ab
	98 258=ba
	256 259=aac
	99 260=ca
	257 261=abc
	260 262=cab
	98 263=bc
	99 264=cb
	98

Procediamo per righe:

1. All'inizio guardiamo il primo carattere, *a*, lo abbiamo nel dizionario → guardiamo dunque il secondo, *a*, non abbiano la sequenza *aa* stavolta → scriviamo in output il numero del dizionario corrispondente al carattere *a* e aggiungiamo alla prima posizione vuota, ossia 256, la sequenza *aa* formata dalla sequenza che già abbiamo nel dizionario a cui aggiungiamo il prossimo carattere. Ci spostiamo al secondo posto.
2. Il secondo è ancora una *a* che abbiamo, guardiamo avanti e vediamo una *b*, la sequenza *ab* non la abbiamo → output 97 e aggiungiamo *ab* alla posizione 257.
3. Leggiamo ora la *b* che abbiamo, guardiamo avanti e vediamo una *a*, la sequenza *ba* non l'abbiamo → output 98 e aggiungiamo *ba* alla posizione 258.

Nella fase di decoding ricostruiamo il codice ed il dizionario basandoci solamente sui codici ricevuti:

Esempio 1.7.5. Vediamo un esempio di decoding:

Input	Dictionary
97	 256=aa
97	 257=ab
256	 258=ba
99	 259=aac
257	 260=ca
260	 261=abc
98	 262=cab
99	 263=bc

La fase di decoding prosegue in modo analogo all'encoding, con l'unica differenza che invece di guardare i caratteri successivi per l'aggiunta al dizionario, facciamo un check su quelli precedenti.

Procediamo per righe:

1. Riceviamo 97 che corrisponde ad una *a* nel dizionario di default, prima non abbiamo niente e quindi passiamo al successivo.
2. Ancora un 97, controlliamo prima e avevamo una *a*, siccome non abbiamo la sequenza *aa* nel dizionario, la aggiungiamo nella posizione 256.
3. Così via come nell'encoding...

Caso Speciale

Una cosa della quale dobbiamo tenere conto utilizzando questo algoritmo, è il caso speciale in cui abbiamo delle ripetizioni:

Esempio 1.7.6.

albalalaica

La compressione della seguente stringa sarà:

Output	Dizionario
97	[256] = 'al'
108	[257] = 'lb'
98	[258] = 'ba'
256	[259] = 'ala'
259	[260] = 'alai'
105	[261] = 'ic'
99	[262] = 'ca'
97	

In fase di decoding invece la procedura sarà:

Input	Decoding	Dizionario
97	a	
108	l	[256] = 'al'
98	b	[257] = 'lb'
256	al	[258] = 'ba'
259	???	

In questi casi uò succedere che ci arrivi in input una posizione del dizionario che non è ancora stata ricostruita; nell'esempio vediamo che 259 dovrebbe essere *ala*, ossia l'input allo step successivo con una *a* extra alla fine della sequenza → la regola che ne deriviamo è dunque che, in caso di codice non ancora ricostruito, l'output sarà identico a quello dello step precedente a cui aggiungiamo in coda il primo carattere della sequenza.

Capitolo 2

Image Capture

Con **Image Capture** intendiamo il processo tramite il quale andiamo a "catturare" la realtà che ci circonda in modo da poterla renderla disponibile su di una macchina; questo avviene tramite la digitalizzazione di segnali provenienti da diversi sensori: Fotocamera, TV Camera, Scanner . . .

2.1 Light Formulas

Per comprendere le formule della luce e della quantità luminosa dobbiamo analizzare alcune grandezze radiometriche fondamentali.

- **Energia radiante** Q_e : è l'energia totale emessa da una sorgente.
- **Flusso radiante** (potenza radiante) Φ_e : è l'energia emessa da una sorgente per unità di tempo.
- **Intensità radiante** I_e : è il flusso radiante per unità di angolo solido in una data direzione, considerando la sorgente come origine delle coordinate.
- **Radianza** L_e : È la quantità di energia emessa da una superficie nell'unità di tempo (= Flusso Radiante) per unità di angolo solido in una data direzione (= Intensità radiante) e per unità di superficie.

La luminanza L_v è la parte del flusso di radiazione che è capace di produrre una sensazione luminosa. Si ottiene moltiplicando lo spettro di flusso radiante per la curva di efficacia luminosa spettrale, cioè per la funzione $V(\lambda)$, ed integrando sull'intervallo di lunghezze d'onda utile.

L'intensità luminosa I_v è la grandezza fotometrica corrispondente all'intensità radiante, cioè è il flusso luminoso emesso in un angolo solido pari a 1 sr (steradianti) in una data direzione.

$$Q_e : \text{Energia Radiante} \quad P_e = \frac{dQ_e}{dt} \quad \text{Flusso Energia Radiante} \quad (2.1)$$

$$I_e = \frac{dP_e}{d\Omega} : \text{Intensità Radiante} \quad L_e = \frac{d^2 P_e}{dA_p d\Omega} : \text{Radianza} \quad (2.2)$$

$$I_v = K_m \int_{380}^{780} I_{e,\lambda} V(\lambda) d\lambda \quad \text{Intensità luminosa} \quad (2.3)$$

$$L_v = K_m \int_{380}^{780} L_{e,\lambda} V(\lambda) d\lambda \quad \text{Luminanza} \quad (2.4)$$

2.2 Color Representation

A differenza delle nostre orecchie, gli occhi non riescono ad effettuare un'analisi spettrale ma ci danno solo una sensazione della combinazione di tutte le lunghezze d'onda visibili, ossia non vediamo un'onda verde, una blu ed una rossa ma un insieme delle 3.

Il primo color space fu introdotto nel 1860 composto da 3 diverse lunghezze d'onda che combinate insieme erano in grado di riprodurre qualsiasi colore possibile, il **CIE RGB Color Space**:

$$\lambda_b = 435.8\text{nm} \quad \lambda_g = 546.1\text{nm} \quad \lambda_r = 700.0\text{nm}$$

Alcune combinazioni richiedevano fonti luminose negative per il rosso ad esempio, cosa fisicamente impossibile e quindi non tutti i colori erano riproducibili; da qui è stato creato un nuovo spazio chiamato **XYZ**, in cui la Y rappresenta la luminanza mentre X e Z non sono correlate a nulla di fisico e quindi siamo in grado di riprodurre tutti i colori.

2.2.1 CIEXYZ Color Space

Questo spazio è alla base di qualsiasi altro color space siccome consente di rappresentare qualsiasi colore fisico:

$$\begin{bmatrix} X \\ Y \\ Z \end{bmatrix} = \begin{bmatrix} 0.4900 & 0.3100 & 0.2000 \\ 0.1170 & 0.8124 & 0.0106 \\ 0 & 0.0100 & 0.9900 \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix}$$

Sfortunatamente questo spazio non è adatto a descrivere la distanza tra colori secondo la percezione umana; si è dunque arrivati ad un nuovo color space.

2.2.2 CIELAB Color Space

In questo caso L ha il significato della luminanza mentre A e B sono le coordinate:

$$a^* = 500 \left[f\left(\frac{X}{X_n}\right) - f\left(\frac{Y}{Y_n}\right) \right] \quad b^* = 200 \left[f\left(\frac{Y}{Y_n}\right) - f\left(\frac{Z}{Z_n}\right) \right]$$

$$\text{with } f(x) = \begin{cases} x^{\frac{1}{3}} & x > 0.008856 \\ 7.787x + \frac{16}{116} & x \leq 0.008865 \end{cases}$$

2.2.3 Color Reproduction

Esistono 3 tecniche fondamentali tramite i quali possiamo riprodurre un colore:

- **Additive Synthesis:** Con questa tecnica creiamo un colore tramite il mixing di due o tre colori tra di loro, normalmente: rosso, verde e blu.
- **Subtractive Synthesis:** Può essere considerata la duale della prima tecnica; è ciò che accade quando si dipinge.
- **Spatial Integration:** È una tecnica con la quale miriamo a riprodurre la Sintesi Additiva all'interno dell'occhio di chi osserva, ossia il colore non viene processato alla sorgente ma solo alla destinazione.

2.2.4 CRT Monitor

I monitor CRT, ossia *Cathod Ray Tube*, è una vecchia tecnologia usata originariamente per la riproduzione della luce; è stato alla base dei sistemi televisivi per molto tempo e si basa sulla sintesi dei colori mediante *Spatial Integration* di fosforo RGB.

Gamma Correction

Fino ad'ora abbiamo assunto che la relazione tra la luce e la sua rappresentazione numerica fosse lineare. Nel caso del CRT, la relazione tra l'emissione rispetto alla quantità di tensione applicata non è affatto lineare: mettiamo il caso di applicare una certa tensione ed otteniamo un data quantità di luce, se raddoppiamo la tensione la luce non raddoppia ma cresce comunque; arriviamo a vedere che in qualche modo questa relazione è di tipo esponenziale controllata da un fattore γ .

$$\frac{L_v}{L_{v,max}} = \left(\frac{E}{E_{max}} \right)^\gamma$$

Inizialmente questo valore è stato settato su 2.2 per creare un senso comune del colore; il problema era che ogni televisore doveva poi calcolare questa relazione esponenziale, solo che al tempo questa computazione era molto difficile siccome non esistevano i PC. La soluzione fu quella di acquisire l'immagine, misurare i livelli di voltaggio ed applicare la correzione gamma alla sorgente in modo da non far gravare questo peso su tutte le destinazioni.

Esempio 2.2.1. Supponiamo di avere due valori di luminanza a, b a noi ignoti, conosciamo però a', b' di norma $\in [0, 1]$. Abbiamo quindi che $a' = a^\gamma, b' = b^\gamma$ e vogliamo calcolare il valore corrispondente alla luminanza media tra i due punti. La nostra "macchina" calcolerà:

$$m' = \frac{a' + b'}{2} \quad \text{ma dovrebbe essere} \quad m' = m^\gamma = \left(\frac{a + b}{2} \right)^\gamma$$

Come facciamo a calcolare il corretto valore di m' avendo a disposizione solamente a', b' ?

$$a = (a')^{\frac{1}{\gamma}}, b = (b')^{\frac{1}{\gamma}} \rightarrow m' = m^\gamma = \left(\frac{(a')^{\frac{1}{\gamma}} + (b')^{\frac{1}{\gamma}}}{2} \right)^\gamma$$

2.2.5 RGB Color Space al PC

La rappresentazione dei colori nelle "macchine" è regolata dalle schede video che dovranno utilizzare un monitor CRT; una delle tipiche rappresentazioni è detta *truecolor*, la quale utilizza valori a 8 bit per ogni pixel in modo da controllare i 3 segnali colorati portati al monitor. Possiamo anche utilizzare meno bit per banda come per esempio nel *hicolor* a 15 o 16 bit → 5-5-5, 5-6-5. Se non è specificato altrimenti questi valori sono stati sottoposti alla correzione gamma.

2.2.6 Transmissive Color Spaces

Lo spazio RGB non è l'unico esistente, questo altro tipo di spazi sono stati creati per le trasmissioni televisive: per molto tempo queste trasmissioni sono state solo in scala di grigi (bianco e nero), nelle quali solamente un segnale veniva mandato invece che 3. Siccome chiedere alle persone di cambiare il decoder all'arrivo delle TV a colori era impensabile, dato il costo molto elevato, è stato aggiunto un segnale modulato in fase in modo da poter filtrare via i segnali non utilizzabili dall'apparecchio; soluzione molto più economica.

YC_BC_R Standard

Questo è lo standard che ha assunto un ruolo dominante nello spazio digitale, nel caso della conversione RGB avviene in questo modo:

$$\begin{bmatrix} Y \\ C_B \\ C_R \end{bmatrix} = \begin{bmatrix} 0.299 & 0.587 & 0.114 \\ -0.1687 & -0.3313 & 0.5 \\ 0.5 & -0.4187 & -0.0813 \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix} + \begin{bmatrix} 0 \\ 128 \\ 128 \end{bmatrix}$$

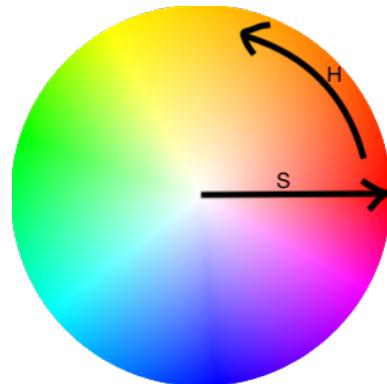
Questa matrice è la stessa utilizzata dal JPEG: nella colonna RGB abbiamo valori gamma-corretti $\in [0, 255]$ prodotti dalla fotocamera.

- **Y** è il valore di illuminazione, qui il verde è dominante e come possiamo vedere non è affatto la media tra le 3 componenti.

- C_B è un color channel a 2 vie, valori positivi verso il blu mentre i negativi verso una tonalità di giallo.
- C_R è un color channel a 2 vie, valori positivi verso il rosso mentre i negativi verso il verde.

2.2.7 Ordered Space

In aggiunta allo spazio RGB, ne esistono altri che permettono di manipolare i colori tramite specifiche numeriche, utile se stiamo processando immagini manualmente, siccome utilizzando lo spazio RGB questa operazione risulta difficile, ad esempio se vogliamo una versione più chiara del colore. Per risolvere questo problema Munsell creò un *color atlas*, una sorta di dizionario dei colori, che ordina i colori in "pagine" basandosi su **H: Hue** ed **S: Saturation**.



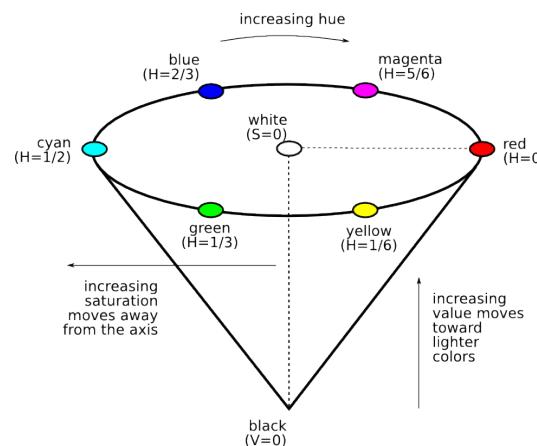
Come possiamo vedere è un cerchio, dobbiamo trovare un modo per approssimare le coordinate polari senza seno e coseno; vediamo come.

HSV Color Space

Lo spazio HSV, Hue Saturation Value, è un'approssimazione lineare in cui H è un angolo $\in [0, 360]$ mentre S e V sono due valori $\in [0, 1]$; per ogni punto dobbiamo definire:

$$\begin{aligned} Max &= \max(R, G, B) \\ Min &= \min(R, G, B) \end{aligned}$$

Consideriamo la seguente struttura:



Come possiamo vedere S indica la differenza tra il minimo e il massimo, se abbiamo lo stesso valore per R, G e B allora ciò che otteniamo è un colore 0-saturato, per cui sono su scala di grigio; per ottenere il valore S facciamo nel seguente modo:

$$S = \frac{(Max - Min)}{Max}$$

Mentre per V utilizziamo il valore massimo:

$$V = Max$$

Ci rimangono ora da modellare gli angoli da 0 a 360, ossia dobbiamo ottenere la linearizzazione di un cerchio, ecco come:

$$H = \frac{\pi}{3} \begin{cases} \frac{G-B}{(Max-Min)} & Max = R \\ 2 + \frac{B-R}{(Max-Min)} & Max = G \\ 4 + \frac{R-G}{(Max-Min)} & Max = B \end{cases}$$

HLS/HSL Color Space

È uno spazio introdotto poco dopo l'HSV, qui L sostituisce V in modo che il valore del bianco non sia lo stesso degli altri colori primari; da qui definiamo L come:

$$L = \frac{Max + Min}{2}$$

Oltre a questo, cambiamo solo la definizione di S, lasciando H invariato:

$$S = \begin{cases} \frac{Max-Min}{(Max+Min)} & L \leq 0.5 \\ \frac{Max-Min}{2-(Max+Min)} & L > 0.5 \end{cases}$$

2.2.8 Chromatic Quantization

Così come la risoluzione spaziale in altezza e larghezza, anche quella dei colori è un aspetto da tenere in considerazione siccome non è sempre necessario avere tutti i colori *truecolor*, per risparmiare spazio e potenza computazionale possiamo adottare due tecniche:

1. È possibile utilizzare **meno bit** per pixel.
2. Utilizzare una **Palette**, ossia una sorta di look-up table, valori predefiniti in memoria a cui i pixel puntano.

Palette Selection

La selezione della palette gioca un ruolo fondamentale nella rappresentazione dei colori; è sempre possibile usare una **palette standard**, siccome trovare i migliori 256 colori che approssimano l'immagine è NP-Completo. Esistono dunque una serie di euristiche che ci permettono di ottimizzare la scelta dei colori della palette, una di queste è stata introdotta da Paul Heckbert, il quale ha ideato il **Median Cut Algorithm** per ottenere una palette ottimizzata di colori.

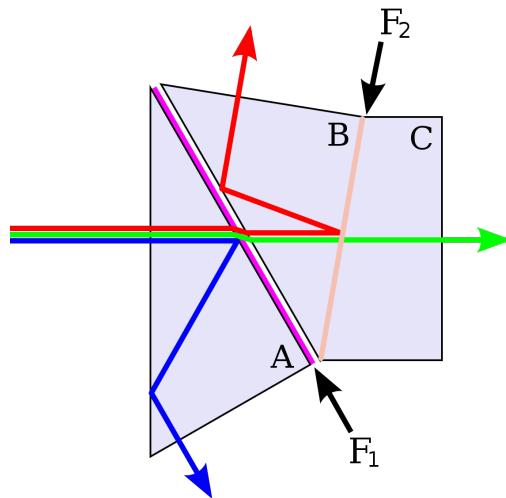
2.3 Hardware per l'Acquisizione

La prima soluzione a questo problema fu quella di utilizzare un prisma per dividere i raggi di luce basandoci sulla lunghezza d'onda; per la precisione si trattava di un **Prisma Dicroico**:

- Sostituisce le precedenti celle fotosensibili le quali non erano sensibili alle diverse lunghezze d'onda.
- Con questo prima possiamo dividere un singolo raggio di luce in due raggi distinti (diversi colori con diverse lunghezze d'onda).
- Una tipica applicazione è quella di combinare più prismi dicroici in modo da dividere la luce nei tre colori R, G e B; (questo da il nome ad alcune fotocamere 3-CCD).

2.3.1 Dichroic & Trichroic Prism

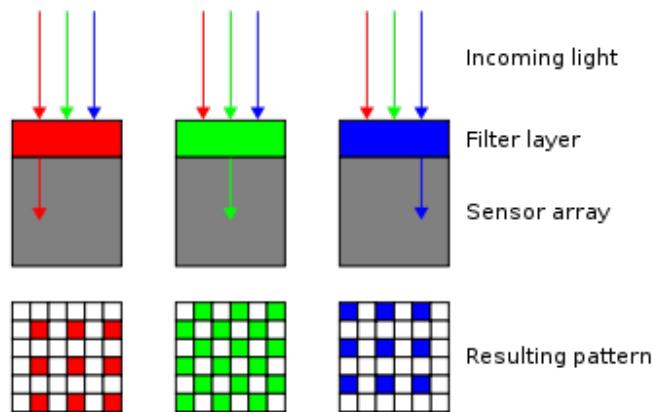
Sotto è mostrato un possibile layout per i prismi:



Come possiamo vedere la luce entra e passa attraverso il primo prisma dicroico (notiamo come in ingresso abbiamo in realtà un solo raggio di luce e non 3 separati), il quale divide le onde blu dalle altre attraverso un bilanciamento tra riflessione e rifrazione ma siccome non vogliamo che il raggio rimbalzi a caso, il secondo deve essere una riflessione totale in modo da poterlo indirizzare verso il sensore. Dopodiché un secondo prisma divide il rosso dal verde mentre l'ultimo è utilizzato solo per assicurarci che il raggio esca alla giusta angolazione.

2.3.2 Bayer Pattern

Le fotocamere 3-CCD sono impiegate solo in ambito professionale, la reale soluzione plausibile fu introdotta da Bayer: il trucco qua è stabilire un compromesso ed invece che avere le 3 misurazioni per ogni pixel abbiamo solamente una misura per pixel e per ottenere gli altri due valori effettuiamo un'interpolazione con i valori vicini. Siccome stiamo effettuando un sub-sampling dobbiamo rispettare il teorema di Nyquist; in ogni caso commetteremo degli errori, per questo motivo si andrà a generare dell'aliasing, creazione di falsi colori che non sono presenti, nei punti in cui c'è una forte variazione lungo una delle due direzioni. Vediamo come sono costruiti questi pattern:



Demosaicing

Il demosaicing è il passo successivo all'acquisizione dei dati con il pattern di Bayer, ci permette di trasformare le informazioni ottenute dai sensori in immagini a colori RGB complete, nello specifico esistono vari modi come ad esempio *Nearest Neighbor*, *Bilinear Interpolation*, *Laplacian Correction*.

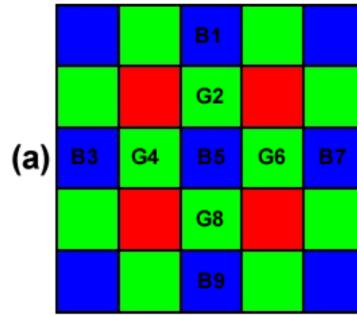
Nearest Neighbor In questo metodo le componenti sono interpolate utilizzando i valori dei pixel in un "vicinato", ossia un gruppo di dati in un delimitato spazio; per i valori mancanti prendiamo quello più vicino in una delle direzioni fondamentali. Questo metodo è molto veloce e semplice ma terribile dal punto di vista visiva.

Bilinear Interpolation È molto simile al metodo NN ma al posto di utilizzare il valore più vicino, calcoliamo ed utilizziamo la media dei valori attorno. È un algoritmo molto efficiente ed è la base per alcuni migliori anche se produce un effetto di blurring nell'immagine.

Laplacian Correction L'idea di questo algoritmo è quella di interpolare in maniera lineare evitando però i casi in cui sappiamo che in una data direzione abbiamo un forte cambiamento; i canali blu e rosso non sono dunque indipendenti l'uno dall'altro e non sono nemmeno indipendenti dal verde, il quale viene utilizzato come "proxy" per cercare di predirre la variazione degli altri colori.

Vediamo come:

Esempio 2.3.1. Partiamo da questa immagine e consideriamo il verde, ossia guardiamo solo le variazioni del verde:



Di solito usiamo le derivate per controllare le variazioni ma qui non abbiamo nessuna equazione da derivare, dobbiamo perciò approssimarla utilizzando la *central difference* prendendo il valore successivo e quello precedente:

$$\frac{\partial f(x)}{\partial x} \cong f(x+1) - f(x-1)$$

Ora che abbiamo quella di grado 1, come facciamo la derivata seconda?

$$\frac{\partial^2 f(x)}{\partial x^2} \cong f'(x+1) - f'(x-1) =$$

$$f'' \cong f(x+2) - 2f(x) + f(x-2)$$

Sappiamo ora come fare le derivate, vediamo ora di calcolare il valore del verde alla posizione centrale, controlliamo la sua variazione orizzontale che si presenta come:

$$|G4 - G6| + |2B5 - B3 - B7|$$

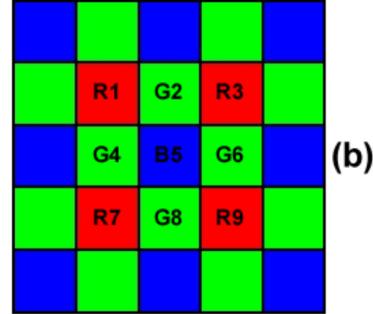
In cui i valori assoluti sono per prendere in considerazione solo l'ampiezza e non le direzioni ed i primi valori corrispondono alla variazione del verde con la derivata prima, i secondi sono la variazione orizzontale del blu con la derivata seconda.

Controlliamo la variazione verticale:

$$|G2 - G8| + |2B5 - B1 - B9|$$

Ora che abbiamo entrambe le variazioni vediamo quale delle due è maggiore e scegliamo di interpolare verso quella in cui abbiamo un cambiamento minore; altrimenti se sono uguali interpoliamo lungo tutte le direzioni.

Vediamo ora il caso in cui dobbiamo interpolare il rosso o il blu:



Ora che abbiamo tutti i valori per il verde, se siamo nel caso in cui abbiamo i dati per i valori del rosso nelle celle sopra e sotto, oppure destra e sinistra, interpoliamo tenendo conto di questi valori aggiungendo la derivata seconda dei pixel verdi. Nel caso delle diagonali invece, quindi con 4 valori da interpolare, facciamo la stessa cosa del verde ma invece che considerare le variazioni verticali ed orizzontali, teniamo in conto quelle diagonali. Per il blu è la stessa cosa del rosso.

Capitolo 3

JPEG Compression

Lo standard JPEG è stato introdotto siccome con le precedenti tecnologie di compressione applicate ad immagini, si riusciva ad ottenere al massimo un rapporto del 50%, il che non è sufficiente per la quantità di figure che produciamo oggi; è stato creato per gestire immagini fotografiche, con una scala di colori continua e non scene realizzate al computer. Questa tecnica impiega al suo interno alcune delle tecnologie già viste come la **DCT**, **RLE** e l'**Huffman Encoding**(entropy coding). Questo standard opera in 4 diverse modalità:

- **Lossless JPEG:** Una versione non più usata ormai.
- **Baseline JPEG:** Versione standard utilizzata. (Sequenziale)
- **Progressive JPEG:** Variazione della Baseline.
- **Hierarchical JPEG:** Variazione della Baseline.

3.1 Baseline JPEG

Questo standard di compressione è detto *color-blind*, ossia non tiene in considerazione i colori per sviluppare le tecniche di compressione, infatti mira a comprimere matrici di n bits per pixel qualunque essi siano; per immagini a colori dobbiamo dunque separare i vari livelli e lavorare su ognuno di essi in maniera separata.

Per prima cosa, ogni pixel viene scalato:

$$x' = x - 2^{n-1}$$

In cui x è il valore originale del pixel, x' rappresenta la scalatura ed n è il numero di bit per pixel che stiamo considerando; il motivo di questa scalatura è quello di avere "*0-mean values*". Dopodiché l'immagine viene suddivisa in blocchi 8×8 non sovrapposti ed ogni blocco viene applicata la **DCT**, Discrete Cosine Transform, per ottenere matrici $\in M_{8 \times 8}$ di coefficienti.

Questi valori vengono dunque quantizzati, ed ecco la principale parte *lossy* del JPEG, per poi essere ordinati secondo uno *zig-zag path*; infine questi coefficienti vengono codificati utilizzando una combinazione di **RLE** ed **Huffman**.

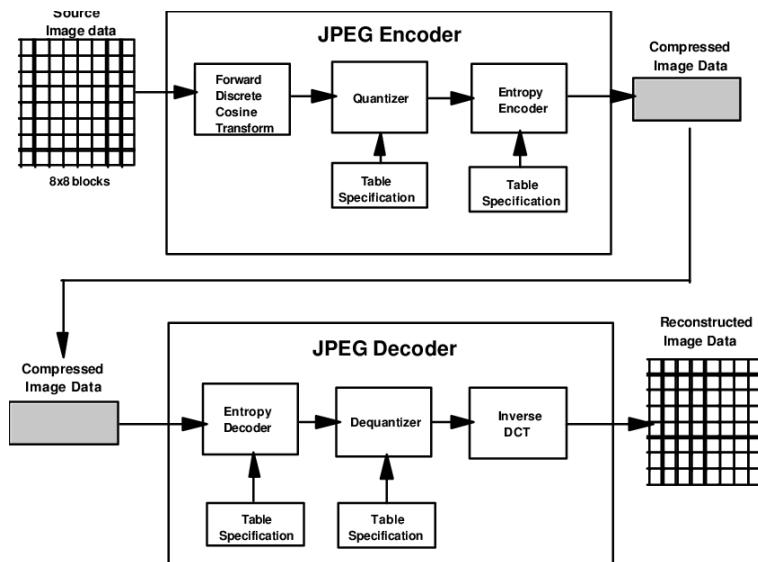


Figura 3.1: Schema di Compressione/Decompressione

3.1.1 DCT: Discrete Cosine Transform

Questa trasformata è una variante della più nota *Trasformata di Fourier*, che lavora solo con coefficienti reali, niente roba immaginaria; questa operazione è come fare un cambio di base, ci permette di osservare i nostri valori da un nuovo punto di vista, un nuovo spazio, rispetto a quello canonico, in modo da poterne analizzare le frequenze.

Esempio 3.1.1. Supponiamo di avere i seguenti valori espressi nella seguente base:

$$v = \begin{bmatrix} 3 & 4 \end{bmatrix} \quad B = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

Gli stessi dati possono essere espressi in modo diverso, per esempio tramite la loro media e la loro differenza fratto 2, la base cambierà di conseguenza:

$$v = \begin{bmatrix} 3.5 & -0.5 \end{bmatrix} \quad B = \begin{bmatrix} 0.5 & 0.5 \\ 0.5 & -0.5 \end{bmatrix}$$

Arriviamo dunque alla DCT vera e propria: si tratta di una trasformata bidirezionale ed è fatta nel seguente modo:

$$S_{uv} = \frac{1}{4} C_u C_v \sum_{y=0}^7 \sum_{x=0}^7 s_{xy} \cos\left(\frac{(2x+1)u\pi}{16}\right) \cos\left(\frac{(2y+1)v\pi}{16}\right) \quad \text{con } 0 \leq u, v \leq 7$$

In cui:

$$S_{uv} = \begin{bmatrix} S_{00} & S_{01} & \cdots & S_{07} \\ S_{10} & S_{11} & \cdots & S_{17} \\ \vdots & \vdots & \ddots & \vdots \\ S_{70} & S_{71} & \cdots & S_{77} \end{bmatrix} \quad s_{xy} = \begin{bmatrix} s_{00} & s_{01} & \cdots & s_{07} \\ s_{10} & s_{11} & \cdots & s_{17} \\ \vdots & \vdots & \ddots & \vdots \\ s_{70} & s_{71} & \cdots & s_{77} \end{bmatrix}$$

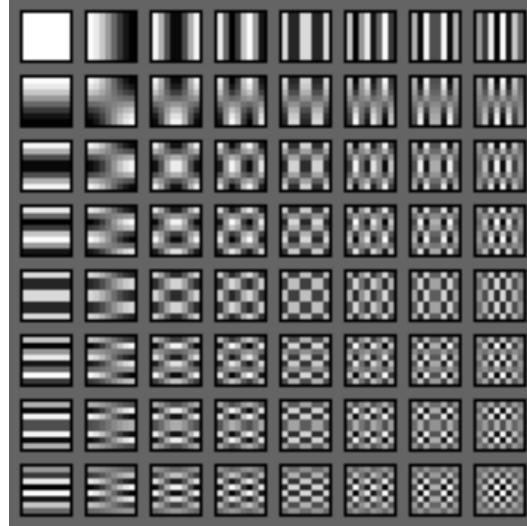
Il valore di C_i è dato dalla seguente regola:

$$C_i = \begin{cases} \frac{1}{\sqrt{2}} & x = 0 \\ 1 & altrimenti \end{cases}$$

Come possiamo vedere dunque ogni valore S_{uv} ha un contributo fornito da tutti i valori s_{xy} contemporaneamente, definizione di prodotto scalare!

$$S_{uv} = \frac{1}{4} C_{uv} s_{xy} \cdot k_{uv}$$

Questa trasformata ci permette, tramite il cambiamento dei valori in u, v , di osservare la frequenza dei nostri valori, ossia quanto rapidamente cambiano. Possiamo vedere qui una visualizzazione 2D dei diversi valori assunti durante la trasformata:



Questa matrice ci da dunque anche la motivazione sul metodo di ordinamento a zig-zag, sulle diagonali abbiamo la stessa frequenza, crescente andando verso destra.

3.1.2 Quantization

Questa porzione è la vera e propria parte *lossy* del JPEG in cui avviene una perdita di precisione in quanto approssimiamo dei valori per risparmiare molti bit; nelle immagini naturali, è molto più facile avere variazioni che si avvicinano più alla parte in alto a sinistra della matrice qui sopra piuttosto che quella in basso a destra, perciò invece di quantizzare ogni "cella" dividendo per lo stesso numero, useremo numeri più piccoli nella parte con meno variazione mentre saranno più grandi quelli della sezione a varianza maggiore:

16	11	10	16	24	40	51	61
12	12	14	19	26	58	60	55
14	13	16	24	40	57	69	56
14	17	22	29	51	87	80	62
18	22	37	56	68	109	103	77
24	35	55	64	81	104	113	92
49	64	78	87	103	121	120	101
72	92	95	98	112	100	103	99

Figura 3.2: JPEG Quantization Matrix

In questo modo avremo errori maggiori ad alte frequenze mentre saranno minori quelli commessi a frequenze più basse.

I coefficienti che otteniamo non hanno tutti lo stesso significato, ad esempio il primo coefficiente, quello in posizione $(0, 0)$, di ogni blocco 8×8 , viene chiamato *DC Coefficient* ed è proporzionale alla media dei valori dei pixel all'interno del blocco; gli altri sono detti *AC Coefficient* e descrivono la variazione all'intero del blocco. Il DC viene compresso in maniera diversa dagli siccome possiamo assumere che avrà un valore simile a quello dei blocchi vicini, la codifica avverrà con una differenza dal coefficiente del blocco precedente.

3.1.3 DC Encoding

Se lavoriamo con *8-bit per pixel*, arriviamo ad avere 11 bit per i DC Coefficient e siccome quello che facciamo è codificare la differenza tra 2 di questi numeri a 11 bit, ci serviranno al massimo 12 bit per la codifica dei primi coefficienti; come effettuiamo l'encoding? Se fossero distribuiti equamente avremmo bisogno di 12 bit, ma di fatto non lo sono, come al solito siamo sotto l'assunzione che differenze basse siano più probabili di quelle alte:

SSSS	DC Difference
0	0
1	-1,1
2	-3,-2,2,3
3	-7,-6,-5,-4,4,5,6,7
4	-15,...,-8,8,...,15
5	-31,...,-16,16,...,31
6	-64,...,-32,32,...,63
7	-127,...,-65,65,...,127
8	-255,...,-128,128,...,255
9	-511,...,-256,256,...,511
10	-1023,...,-512,512,...,1023
11	-2047,...,-1024,1024,...,2047

A sinistra abbiamo le classi, simili all'*Elias γ Coding*, mentre a destra ci sono il numero di valori rappresentabili con i bit di una data classe; man mano che scendo nella tabella si va a costruire una sorta di piramide in cui i valori che stanno in cima sono i più probabili mentre quelli alla base sono i più rari.

Utilizziamo dunque Huffman per effettuare l'encoding della categoria ed un coding binario per i valori effettivi:

Esempio 3.1.2. Prendiamo ad esempio la classe 2 vediamo come effettuiamo la codifica dei valori.

$$-3 = 00 \quad -2 = 01 \quad 2 = 10 \quad 3 = 11$$

In pratica l'encoding viene effettuata in complemento a 1, neghiamo i bit di ogni numero per ottenere la sua versione negativa, siccome i valori che stanno in mezzo sono stati già codificati nelle classi precedenti.

3.1.4 AC Encoding

L'idea è la stessa dei coefficienti DC ma stavolta non andremo a codificare uno scarto tra due numeri ma solamente il valore così com'è, con l'unica differenza che lo 0 non viene codificato in questo modo ma tramite **RLE**:

SSSS	AC Values
1	-1,1
2	-3,-2,2,3
3	-7,-6,-5,-4,4,5,6,7
4	-15,...,-8,8,...,15
5	-31,...,-16,16,...,31
6	-64,...,-32,32,...,63
7	-127,...,-65,65,...,127
8	-255,...,-128,128,...,255
9	-511,...,-256,256,...,511
10	-1023,...,-512,512,...,1023

Questa volta inoltre per ogni valore diverso da 0, associamo il numero di 0 che hanno preceduto questo valore, è stato deciso di utilizzare 4 bit per questo numero perciò avremo (0,15) possibilità; inoltre dobbiamo sempre codificare la classe, in questo caso 10 possibilità, avrò dunque una tabella di Huffman da $16 \times 10 + 2$ combinazioni. Il +2 è dato da due casi speciali: quello in cui ho 16 zeri di fila → ZRL, e quello in cui siamo alla fine del blocco → EOB.

3.1.5 Color Management

Quando lavoriamo con immagini a colori potremmo separare i 3 canali per poi codificarli a parte con le regole che abbiamo appena visto ma nella realtà non è ciò che si fa:

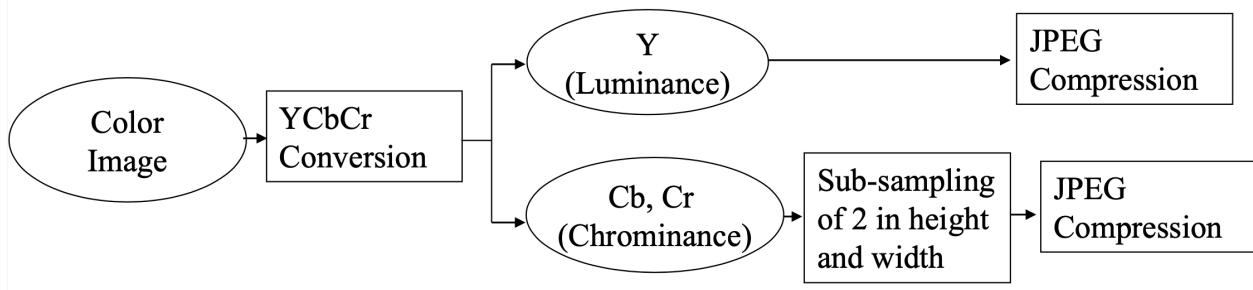


Figura 3.3: JPEG Color Management

Effettuiamo due codifiche diverse, una per i valori di Y così come sono, siccome il nostro occhio è molto più sensibile a questa componente; mentre per i valori di C_b, C_r , prima di effettuare la compressione, viene effettuato un sub-sampling riducendo di 1/4 le immagini originali.

Capitolo 4

Video Compression

Dopo aver visto la compressione delle immagini tramite lo standard JPEG, siamo arrivati ora a dover comprimere interi video che, da un punto di vista tecnico, non sono altro che una sequenza di suddette immagini. Ogni paese ha il proprio standard, in Europa l'obiettivo è quello di mandare 25 frame al secondo.

Siccome l'encoding in un unico stream di un video qualsiasi, anche molto corto e di qualità pessima, richiederebbe quantità di memoria non sostenibili, si sono dovute ideare nuove tecniche per ottimizzare questa procedura.

4.1 INTRA Frame Encoding

In questa tecnica ogni frame viene compresso indipendentemente dagli altri, ossia non andremo ad utilizzare informazioni esterne oltre a quelle interne.

Anche così però, nonostante le dimensioni vengano ridotte notevolmente, abbiamo una richiesta di memoria non sostenibile.

Questo tipo di codifica è nota come **M-JPEG:Motion JPEG**.

4.2 INTER Frame Encoding

Se vogliamo fare meglio però, dobbiamo muoverci al di fuori del singolo frame ed utilizzare informazioni che provengono dagli altri frames. Come mettiamo in pratica questa cosa?

4.2.1 Time Difference

Siccome da un frame al successivo è molto probabile che ci siano piccole variazioni, possiamo predirre quali saranno i valori dei prossimi pixel a partire dai correnti, ovviamente possiamo commettere errori perciò dobbiamo aggiungere alla predizione la differenza tra i due valori; per effettuare la codifica poi sarà necessario avere solamente lo scarto tra i due frame siccome avendo il primo possiamo trovare il secondo tramite la differenza.

Il range di questo valore è più largo di quello dei possibili pixel, siccome se un pixel può essere $\in [0, 255]$, la differenza tra due potrebbe essere $\in [-255, 255]$ quindi siamo passati ad aver bisogno di più bit; dobbiamo perciò riportare questo range in un rango visibile facendo una divisione(intera) per 2 aggiungendo poi 127 \rightarrow lo 0 sarà in corrispondenza del 127.

4.2.2 The Motion

Fin'ora abbiamo visto come due frame acquisiti da una telecamera fissa su oggetti fissi risultino molto simili tra di loro, le cose cambiano parecchio se la telecamera inizia a muoversi: l'introduzione del movimento conduce la nostra predizione ad essere sbagliata; la stessa cosa accade se è l'oggetto a muoversi nella scena invece della camera.

Ci serve un modo per migliorare il nostro pronostico: potremmo inviare di quanto ogni pixel si muove da un frame all'altro, solo che ci servirebbero più dati di prima e quindi

non ha senso.

La prima soluzione fu quella di mandare *motion vectors* per gruppi di pixel: prendiamo un insieme di pixel che si muove nella stessa maniera e inviamo informazioni su dove si trovava in precedenza; la dimensione del gruppo scelta fu 16×16 e venne chiamato *macroblock*. (16×16 per collegarsi al JPEG che utilizza blocchi 8×8)

Motion Vector

Abbiamo utilizzato il termine *motion vector*, ma di cosa si tratta esattamente? È una coppia di numeri (x, y) che indica dove il dato punto si trovava nell'immagine precedente. (In realtà punta alla posizione più simile) Per misurare la similarità dobbiamo confrontare tra loro tutti i blocchi presenti nei due frame, la quale risulta essere un'operazione computazionalmente pesante se usassimo un approccio *Brute Force*.

Fortunatamente esistono approssimazioni ed euristiche per effettuare questa operazione praticamente in *real time*.

Abbiamo detto che avendo a disposizione il frame precedente ed il *motion vector* possiamo costruire l'immagine composta dai macroblocchi di quella prima; ovviamente però anche i vettori che utilizziamo occupano spazio e dobbiamo trovare duqne un compromesso.

SAD: Sum of Absolute Difference Con questa tecnica consideriamo l'informazione che ci fornisce il *motion vector* solo se la **SAD** del blocco in questione è $>$ di una certa soglia: teniamo in considerazione i vettori che sono relativi a grandi movimenti.

Lossy Compression

Il vero guadagno avviene quando buttiamo via parte dell'informazione, non è necessario mandare l'intera differenza tra due frame ma è sufficiente qualcosa di simile e così come nel JPEG, utilizziamo la **DCT** seguita da una quantizzazione per capire quali parti dell'immagine è utile tenere.

In base alla divisione che effettuiamo possiamo ottenere diversi tipi di risoluzione; fin'ora dunque, la compressione dei video avviene nel seguente modo:

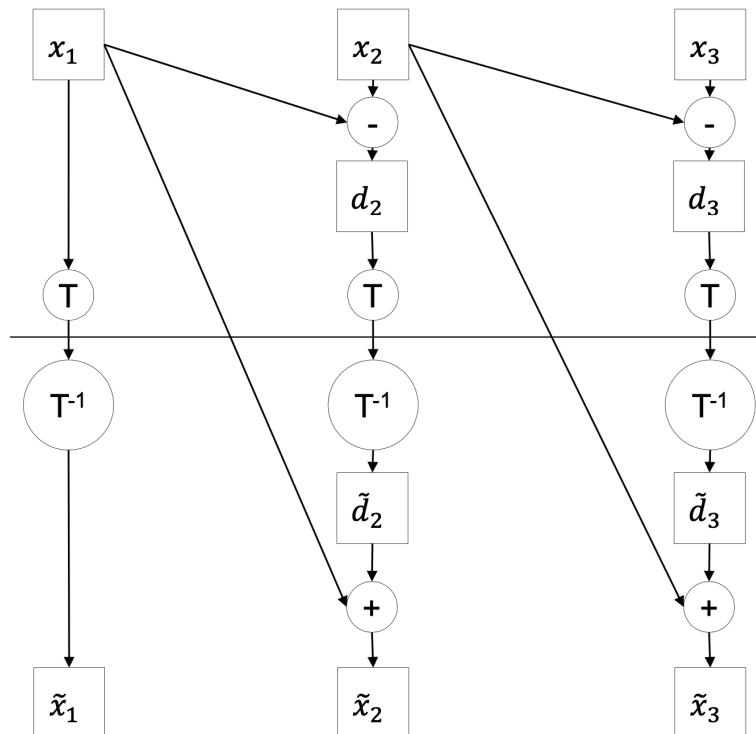


Figura 4.1: Video Compression Initial Schema

Come possiamo vedere c'è qualcosa che non va, infatti abbiamo alcuni collegamenti che nella pratica non possiamo avere, infatti quando siamo alla fine del secondo frame, al

punto \tilde{x}_2 , dovremmo utilizzare informazioni provenienti dal frame originale che però non è disponibile al decoder.

Ciò che accade dunque è che siccome non abbiamo x_1 ma abbiamo la sua ricostruzione, lo schema sarà:

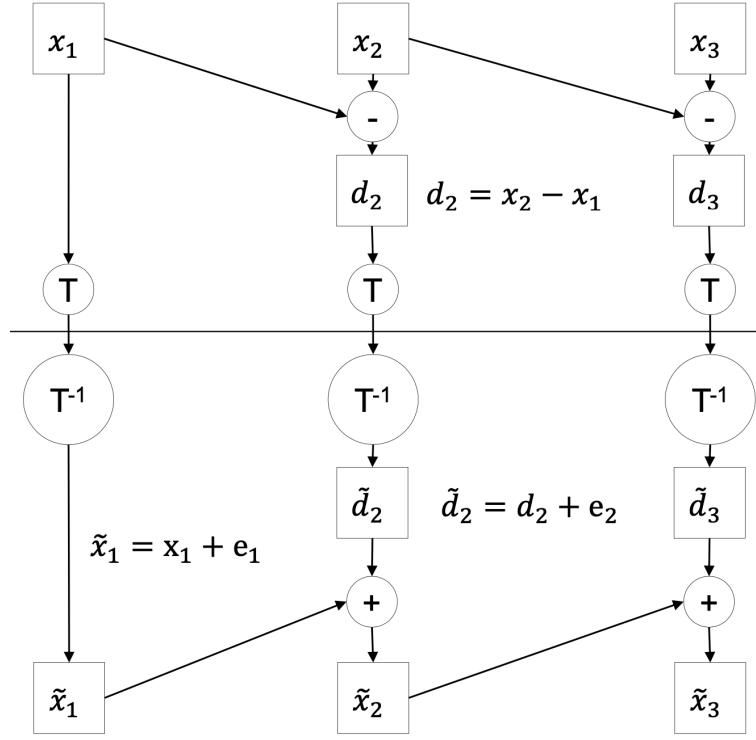


Figura 4.2: Schema Without x_1

È però facile notare che più andiamo avanti e più errori si accumuleranno portando la nostra immagine, dopo pochi frame, ad essere molto rumorosa. Questo fenomeno è detto *error propagation* e dobbiamo dunque trovare una soluzione.

Il problema è che non avendo x_1 , utilizziamo due riferimenti diversi tra la parte alta e la parte bassa dello schema; è questa la cosa da sistemare: dobbiamo usare lo stesso riferimento in entrambi i casi:

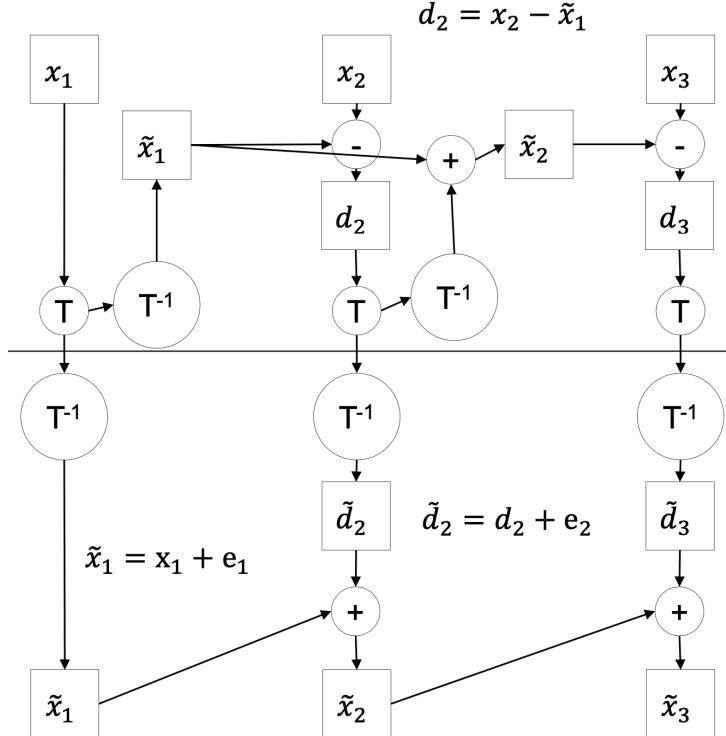


Figura 4.3: Real Compression & Decompression schema

Prima di entrare nello specifico con il primo standard, vediamo uno schema generale di un encoder:

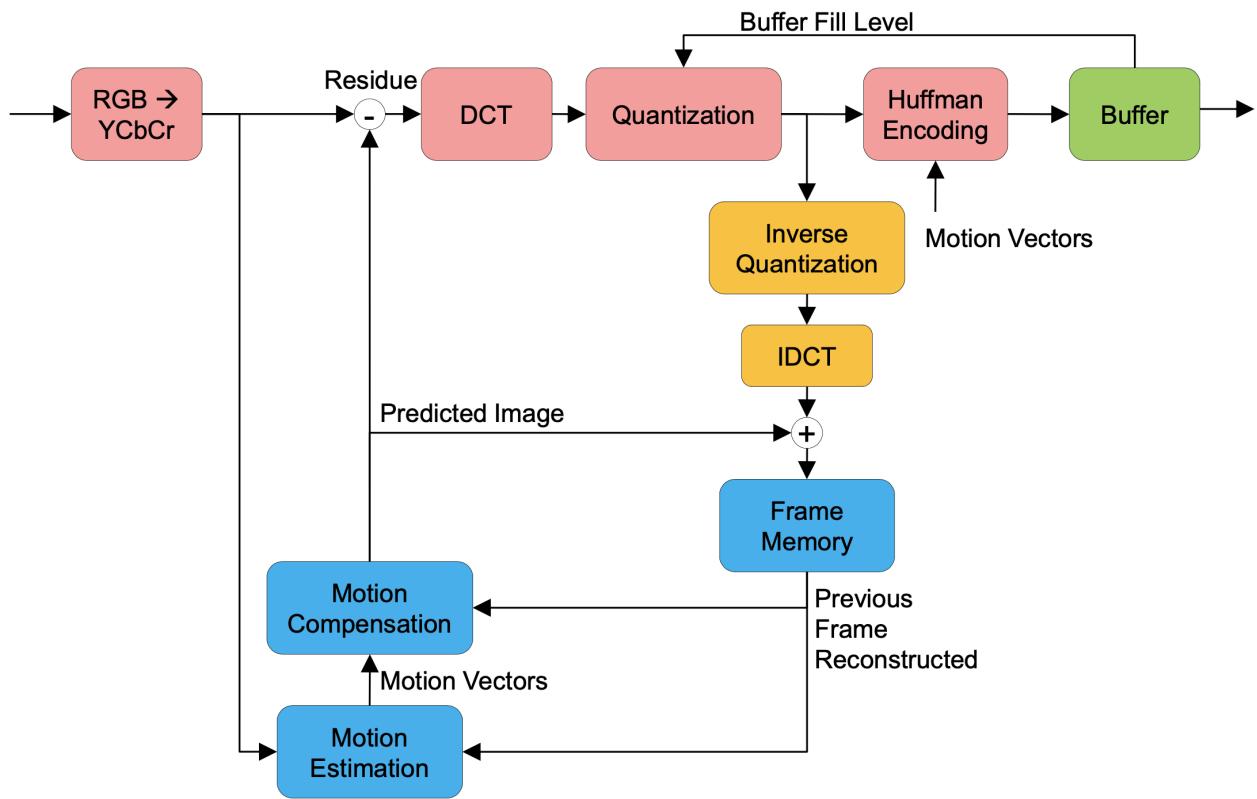


Figura 4.4: General Encoder Schema

Notiamo che non è segnata la parte di INTRA Frame coding ma almeno per il primo frame ne abbiamo bisogno, oppure è utile durante errori di trasmissione o altri casi particolari.

Capitolo 5

H.261 Standard

Dopo aver visto un po' di nozioni generali per quanto riguarda la compressione di file video, iniziamo ora con introdurre uno dei primi standard che fu utilizzato per effettuare *Video Coding*.

Il titolo originale è *Video Codec for Audiovisual Services at $p \times 64$ kbit/s*, in cui " $p \times 64$ kbit/s" deriva dalle linee di trasmissione **ISDN** che arrivavano nelle case con cavi capaci di trasmettere 64 kbps; unendo più linee in parallelo era possibile ottenere la banda sovraccitata. (max 30 Linee)

È uno standard creato per la videoteléfono ed utilizza $Y C_b C_r$ per i colori con le componenti di colore sub-campionate; Vediamo uno schema che raffigura l'encoder per lo standard **H.261**:

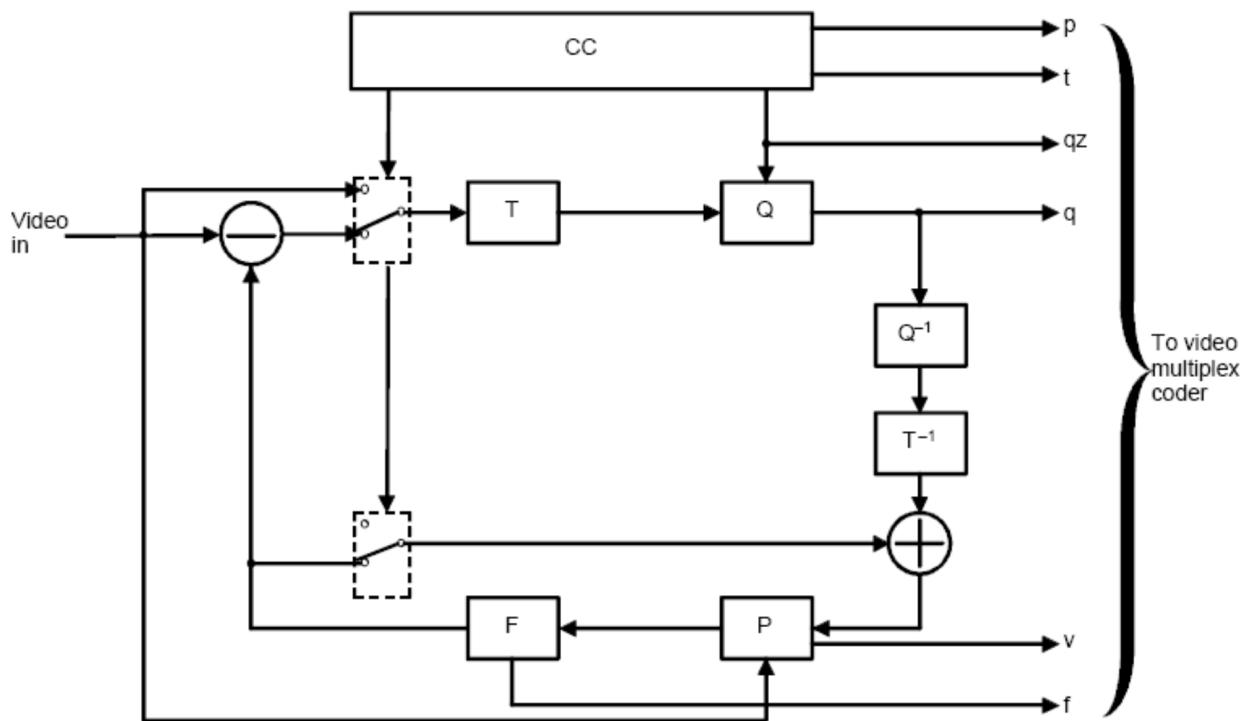


Figura 5.1: H.261 Encoder

Vediamo cosa significano rispettivamente i blocchi:

- **T, T^{-1} :** Sono le operazioni di trasformata e antitrasformata.
- **Q, Q^{-1} :** Rappresentano le fasi di quantizzazione e dequantizzazione.
- **P, F:** Sono i blocchi che permettono di avere l'immagine predetta e di filtrarla, in modo da correggere un minimo gli errori di ricostruzione.
- **Selectors:** Consentono di effettuare selezioni sull'input. (INTER / INTRA)

L'unità fondamentale di questo standard non sono i frame ma bensì i *macroblock*, non esistono in realtà INTRA/INTER Frames ma all'interno dello stesso ci possono essere blocchi trasmessi in un certo modo e altri nella maniera opposta; importante è l'azione del filtro che si avvicina molto a quello gaussiano ma semplificato, infatti ha la seguente forma:

$$\begin{bmatrix} 1/16 & 1/8 & 1/16 \\ 1/8 & 1/4 & 1/8 \\ 1/16 & 1/8 & 1/16 \end{bmatrix}$$

L'applicazione di questo filtro riduce l'entropia di una piccola quantità, nulla di eccezionale ma sempre di aiuto; inoltre per prevenire l'eccessivo calo di qualità, ogni macroblocco deve essere trasmesso **INTRA** almeno una volta ogni 132 volte che viene mandato.

5.1 Bitstream Structure

Esistono 4 livelli all'interno dello stream, ognuno identificato da un diverso *header* in modo da identificare in che sezione della struttura ci troviamo:

1. **Picture**
2. **Group of Blocks**
3. **Macroblock**
4. **Blocks**

Iniziamo a vedere nel dettaglio ogni livello.

5.1.1 Picture Layer

Lo starting code (PSC) è composto da 20 bit:

0000 0000 0000 0001 0000

E come possiamo vedere non è multiplo di un byte siccome non è pensato per essere salvato in memoria rendendo però la ricerca scomoda. (Verrà sistemato in sistemi successivi)

A questo codice segue un altro numero composto da 5 bit, detto *Temporal Reference* (TR), che indica il numero del frame e viene incrementato ogni volta che passa un frame, utile per perdite di frames o se vogliamo skippare intenzionalmente un dato fotogramma; dopo la TR sono presenti 6 bit che indicano *Type Information* (PTYPE), del quale ci interessa solo il bit numero 4 in quanto ci dice se l'immagine è in **QCIF**, value 0, o in **CIF**, value 1.

Infine è presente 1 bit detto *Extra Insertion Information* (PEI), che indica se è presente un byte addizionale nell'header.

5.1.2 Group of Blocks Layer

Ogni *Picture* viene poi diviso in 12, nel caso CIF, o 3, nel caso QCIF, **GOB**, numerati con indici $\in [1, 12]$; ogni GOB è composto da uno starting code (GBSC) da 16 bit:

0000 0000 0000 0001

Seguito da un numero a 4 bit chiamato *Group Number* (GN), che indica l'indice del GOB; la numerazione è utile perché possiamo voler skippare anche dei GOB.

Un'altra informazione importante è rappresentata da 5 bit, nominata *Quantizer Information* (GQUANT), che indica il valore di quantizzazione utilizzato per ogni blocco all'interno dello stesso gruppo; anche qui abbiamo 1 bit addizionale, *GEI* in questo caso, così come per le picture.

5.1.3 Macroblock Layer

Arriviamo alla parte interessante dello stream, ogni GOB è diviso in 11×3 macroblocchi di dimensione 16×16 ed ogni informazione è codificata in modo da risparmiare spazio; quindi se prima avevamo dimensioni predefinite che indicavano le diverse parti di informazione, qua abbiamo molti dati da mandare e quindi fare "economia" di bits diventa molto importante.

La prima cosa da mandare è il numero del Macroblock per la stessa motivazione dei layer precedenti ed questo indirizzo è codificato tramite un codice a lunghezza variabile siccome è facile predire quale sarà il prossimo macroblocco trasmetto; viene dunque inviata una differenza rispetto a quello precedente invece del valore assoluto: di quanti macroblocchi devo spostarmi per avere il prossimo? In questo modo diventa molto probabile avere molti 1.

- Per il primo macroblocco inviamo il suo valore assoluto.
- Per quelli successivi viene mandata la differenza tra quello corrente e quello successivo.
- Se sono alla fine manderò l'inizio del prossimo gruppo o della prossima picture.

MBA	Code	MBA	Code		
1	1	17	0000	0101	10
2	011	18	0000	0101	01
3	010	19	0000	0101	00
4	0011	20	0000	0100	11
5	0010	21	0000	0100	10
6	0001 1	22	0000	0100	011
7	0001 0	23	0000	0100	010
8	0000 111	24	0000	0100	001
9	0000 110	25	0000	0100	000
10	0000 1011	26	0000	0011	111
11	0000 1010	27	0000	0011	110
12	0000 1001	28	0000	0011	101
13	0000 1000	29	0000	0011	100
14	0000 0111	30	0000	0011	011
15	0000 0110	31	0000	0011	010
16	0000 0101 11	32	0000	0011	001
		33	0000	0011	000
		MBA stuffing	0000	0001	111
		Start code	0000	0000	0000 0001

Figura 5.2: MB Standard Table

MBA Stuffing è per non far finire la quantità di dati sotto una determinata soglia, "farcisce" il bit stream in modo da non far perdere priorità agli occhi di alcuni optimizer.

Dopo aver mandato "l'indirizzo", arriva la parte relativa al *Type Information (MTYPE)*, che ci dice se la predizione è INTRA o INTER, se c'è un nuovo valore di quantizzazione, se vogliamo mandare motion vector oppure no, quanti blocchi stiamo inviando e se sono presenti i coefficienti; tutte queste informazioni sono codificate secondo le loro combinazioni tramite la tabella qui sotto:

Prediction	MQUANT	MVD	CBP	TCOEFF	VLC
Intra				x	0001
Intra	x			x	0000 001
Inter			x	x	1
Inter	x		x	x	0000 1
Inter + MC		x			0000 0000 1
Inter + MC		x	x	x	0000 0001
Inter + MC	x	x	x	x	0000 0000 01
Inter + MC + FIL		x			001
Inter + MC + FIL		x	x	x	01
Inter + MC + FIL	x	x	x	x	0000 01

NOTES

1 "x" means that the item is present in the macroblock.

2 It is possible to apply the filter in a non-motion compensated macroblock by declaring it as MC + FIL but with a zero vector.

Figura 5.3: MTYPE Table

Vediamo che uno dei casi più probabili non presenta la differenza ma solo i motion vector, questo vuol dire che il blocco può essere preso ed incollato così com'è dal frame precedente in una nuova posizione.

A seguire possono essere presenti anche queste informazioni come ad esempio 5 bit per indicare il nuovo livello di quantizzazione *MQUANT*; per i motion vector dobbiamo poter essere in grado codificare due numeri ognuno dei quali $\in [-15, +15]$, ci serviranno 5 bit per numero e quindi un totale di 10 bit per i motion vectors. Possiamo fare meglio?

Diverse idee sono nate come ad esempio non utilizzare i valori assoluti dei MVs ma la differenza dal precedente, siccome se qualcosa si sta muovendo a sinistra è molto probabile che il frame dopo si stia ancora muovendo nella stessa direzione, confrontiamo dunque un MV con quello alla sua sinistra; per alcuni non è possibile avere il valore precedente, in questi casi viene usato (0, 0) come predizione.

Utilizzando la differenza, il massimo range che possiamo ottenere è $[-30, +30]$, diventando perciò il doppio di prima ma, siccome sappiamo il valore precedente, riusciamo a dimezzare il range facendo corrispondere due valori di differenza allo stesso codice:

MVD	Code
-16 & 16	0000 0011 001
-15 & 17	0000 0011 011
-14 & 18	0000 0011 101
-13 & 19	0000 0011 111
-12 & 20	0000 0100 001
-11 & 21	0000 0100 011
-10 & 22	0000 0100 11
-9 & 23	0000 0101 01
-8 & 24	0000 0101 11
-7 & 25	0000 0111
-6 & 26	0000 1001
-5 & 27	0000 1011
-4 & 28	0000 111
-3 & 29	0001 1
-2 & 30	0011
-1	011
0	1
1	010
2 & -30	0010
3 & -29	0001 0
4 & -28	0000 110
5 & -27	0000 1010
6 & -26	0000 1000
7 & -25	0000 0110
8 & -24	0000 0101 10
9 & -23	0000 0101 00
10 & -22	0000 0100 10
11 & -21	0000 0100 010
12 & -20	0000 0100 000
13 & -19	0000 0011 110
14 & -18	0000 0011 100
15 & -17	0000 0011 010

Figura 5.4: Motion Vector Difference Codes

Esempio 5.1.1. Supponiamo che la componente x del MV precedente sia -5 e riceviamo una differenza codificata con:

$$0000 \quad 0111 \longrightarrow \text{può essere } -7 \vee 25$$

Ma siccome $-5 + 25 = 20$ che è un valore fuori dal range $[-15, +15]$, ne deduciamo che si tratta per forza di $-7 \rightarrow -5 + (-7) = -12$ che sarà il valore attuale della x .

L'ultimo campo di questo layer è detto *Coded Block Pattern (CBP)*, una bitmap da 6 bit che ci indica quali blocchi sono presenti all'interno del macroblock secondo la seguente numerazione:

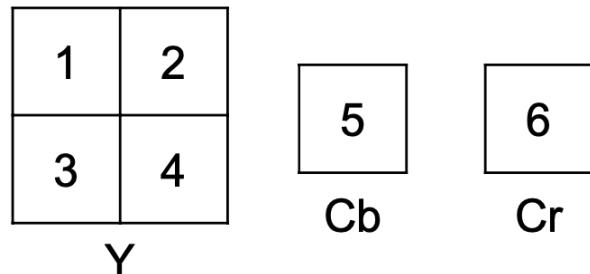


Figura 5.5: CBP Bitmap

Dobbiamo per forza utilizzare 6 bit o possiamo risparmiare? Vediamo sempre le combinazioni più probabili ed otteniamo la seguente tabella:

CBP	Code	CBP	Code
60	111	35	0001 1100
4	1101	13	0001 1011
8	1100	49	0001 1010
16	1011	21	0001 1001
32	1010	41	0001 1000
12	1001 1	14	0001 0111
48	1001 0	50	0001 0110
20	1000 1	22	0001 0101
40	1000 0	42	0001 0100
28	0111 1	15	0001 0011
44	0111 0	51	0001 0010
52	0110 1	23	0001 0001
56	0110 0	43	0001 0000
1	0101 1	25	0000 1111
61	0101 0	37	0000 1110
2	0100 1	26	0000 1101
62	0100 0	38	0000 1100
24	0011 11	29	0000 1011
36	0011 10	45	0000 1010
3	0011 01	53	0000 1001
63	0011 00	57	0000 1000
5	0010 111	30	0000 0111
9	0010 110	46	0000 0110
17	0010 101	54	0000 0101
33	0010 100	58	0000 0100
6	0010 011	31	0000 0011 1
10	0010 010	47	0000 0011 0
18	0010 001	55	0000 0010 1
34	0010 000	59	0000 0010 0
7	0001 1111	27	0000 0001 1
11	0001 1110	39	0000 0001 0
19	0001 1101		

Figura 5.6: CBP Codes Table

5.1.4 Block Layer

Siamo arrivati all'ultimo livello in cui abbiamo gli effettivi coefficienti della **DCT** dei blocchi indicati nel **CBP**, i quali sono codificati secondo l'ordinamento a zig-zag e se il macroblocco è di tipo INTRA allora tutti i singoli blocchi saranno presenti.

Ogni coefficiente AC, sia per INTER che INTRA, sono rappresentati con una coppia (*run, level*) che indicano quanti zeri sono presenti prima del coefficiente non nullo ed il livello, che rappresenta la quantizzazione effettuata. Qua sotto sono rappresentate le coppie più probabili:

Run	Level	Code			
EOB		10			
0	1	1sa) If first coefficient in block	4	1	0011 0s
0	1	11s Not first coefficient in block	4	2	0000 0011 11s
0	2	0100 s	4	3	0000 0001 0010 s
0	3	0010 1s	5	1	0001 11s
0	4	0000 110s	5	2	0000 0010 01s
0	5	0010 0110 s	5	3	0000 0000 1001 0s
0	6	0010 0001 s	6	1	0001 01s
0	7	0000 0010 10s	6	2	0000 0001 1110 s
0	8	0000 0001 1101 s	7	1	0001 00s
0	9	0000 0001 1000 s	7	2	0000 0001 0101 s
0	10	0000 0001 0011 s	8	1	0000 111s
0	11	0000 0001 0000 s	8	2	0000 0001 0001 s
0	12	0000 0000 1101 0s	9	1	0000 101s
0	13	0000 0000 1100 1s	9	2	0000 0000 1000 1s
0	14	0000 0000 1100 0s	10	1	0010 0111 s
0	15	0000 0000 1011 1s	10	2	0000 0000 1000 0s
1	1	011s	11	1	0010 0011 s
1	2	0001 10s	12	1	0010 0010 s
1	3	0010 0101 s	13	1	0010 0000 s
1	4	0000 0011 00s	14	1	0000 0011 10s
1	5	0000 0001 1011 s	15	1	0000 0011 01s
1	6	0000 0000 1011 0s	16	1	0000 0010 00s
1	7	0000 0000 1010 1s	17	1	0000 0001 1111 s
2	1	0101 s	18	1	0000 0001 1010 s
2	2	0000 100s	19	1	0000 0001 1001 s
2	3	0000 0010 11s	20	1	0000 0001 0111 s
2	4	0000 0001 0100 s	21	1	0000 0001 0110 s
2	5	0000 0000 1010 0s	22	1	0000 0000 1111 1s
3	1	0011 1s	23	1	0000 0000 1111 0s
3	2	0010 0100 s	24	1	0000 0000 1110 1s
3	3	0000 0001 1100 s	25	1	0000 0000 1110 0s
3	4	0000 0000 1001 1s	26	1	0000 0000 1101 1s
			Escape		0000 01

a) Never used in INTRA macroblocks.

Figura 5.7: TCOEFF Pair Codes

In caso la coppia non sia presente all'interno della tabella verrà codificata con 20 bit:

escape: 0000 01 "6-bit run" "8-bit level"

Alla fine del blocco abbiamo un codice 10 → *EOB* il quale indica che tutti gli altri coefficienti sono 0; il primo coefficiente per blocchi INTRA è codificato con 8 bit mentre per quelli INTER viene utilizzata la stessa tabella come per gli altri con una piccola ottimizzazione siccome ovviamente non può essere **EOB**.

5.1.5 Quantization

L'ultima cosa che manca ora è come ricostruire, dato il livello, il numero originale utilizzato per calcolare la trasformata. Esistono due diverse formule utilizzate in base al livello di quantizzazione in cui siamo:

$$\text{QUANT Dispari} \begin{cases} REC = QUANT \cdot (2 \cdot level + 1); & level > 0 \\ REC = QUANT \cdot (2 \cdot level - 1); & level < 0 \end{cases}$$

$$\text{QUANT Pari} \begin{cases} REC = QUANT \cdot (2 \cdot level + 1) - 1; & level > 0 \\ REC = QUANT \cdot (2 \cdot level - 1) + 1; & level < 0 \end{cases}$$

In ogni caso: $REC = 0$; $level = 0$

Come mai abbiamo questa separazione? In modo da bilanciare l'errore che commettiamo verso valori sia negativi che positivi.

Capitolo 6

Other Video Standards

Dopo aver visto nel dettaglio come funziona lo standard **H.261**, facciamo una panoramica su altri algoritmi di encoding video che vengono utilizzati.

6.1 MPEG Standard

La prima versione di questo standard, **MPEG-1**, fu ideata per video CD e quindi per simulare le **VHS**; era progettato per frame progressivi ma non aveva alcun supporto per *interlaced videos*, come ad esempio le applicazioni televisive.

Per questo motivo poco dopo è stato introdotto **MPEG-2**, molto efficace e con un bit rate che è passato da **1.5 MB/s** ad un range compreso tra **4 e 9 MB/s**.

6.1.1 Differences with H.261

Vediamo dunque qualche differenza che questo nuovo standard presenta con il vecchio **H.261**.

Il primo concetto fondamentale è quello di *key frame*, il quale permette di ripartire con la trasmissione come se fossimo all'inizio del video permettendo alle persone di connettersi in qualsiasi momento(broadcasting); questo tipo di frame sono definiti *i-frames*, dove la "i" sta per INTRA, in modo da non avere riferimenti a quelli precedenti permettendo appunto di iniziare a ricevere in qualsiasi momento.

L'altro tipo di frame sono detti *p-frames*, in cui la "p" significa predicted, e stavolta abbiamo dei dati collegati ai frame già trasmessi.

Grazie ai progressi fatti con il JPEG, la quantizzazione viene effettuata mediante queste due matrici di valori:

8	16	19	22	26	27	29	34
16	16	22	24	27	29	34	37
19	22	26	27	29	34	34	38
22	22	26	27	29	34	37	40
22	26	27	29	32	35	40	48
26	27	29	32	35	40	48	58
26	27	29	34	38	46	56	69
27	29	35	38	46	56	69	83

Quantization matrix
for *intra* blocks

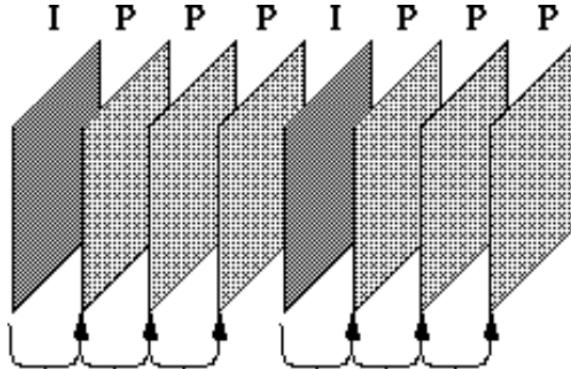
16	16	16	16	16	16	16	16
16	16	16	16	16	16	16	16
16	16	16	16	16	16	16	16
16	16	16	16	16	16	16	16
16	16	16	16	16	16	16	16
16	16	16	16	16	16	16	16
16	16	16	16	16	16	16	16
16	16	16	16	16	16	16	16

Quantization matrix
for *inter* blocks

Come possiamo vedere per i blocchi in INTRA mode, non è una buona idea mantere lo stesso livello di quantizzazione lungo tutta l'immagine mentre per i blocchi INTER non sono satate trovate ragioni per cui si dovrebbero utilizzare valori diversi e dunque si è mantenuto il tutto molto semplice. Un'altra cosa è che qui possiamo regolare la "quantità" di quantizzazione mediante un fattore di scala $\in (0, 1]$.

6.1.2 MPEG Novelties

Per quanto detto fin'ora, ci troviamo un una situazione simile a questa:



Sfortunatamente ci sono casi, la maggior parte, in cui questo modo di effettuare predizioni non funziona siccome è molto probabile che non abbiamo alcuna informazione su pezzi del frame precedente ma, se guardiamo il frame successivo questo dato ci torna disponibile.

Vengono dunque introdotti i frame di tipo B, *Bidirectional-Frames*, i quali effettuano una ricerca sia in riferimenti precedenti che in quelli successivi; notiamo che non devono per forza essere il frame esattamente prima e quello subito dopo ma possono essere un generico frame antecedente ed uno qualsiasi successivo:

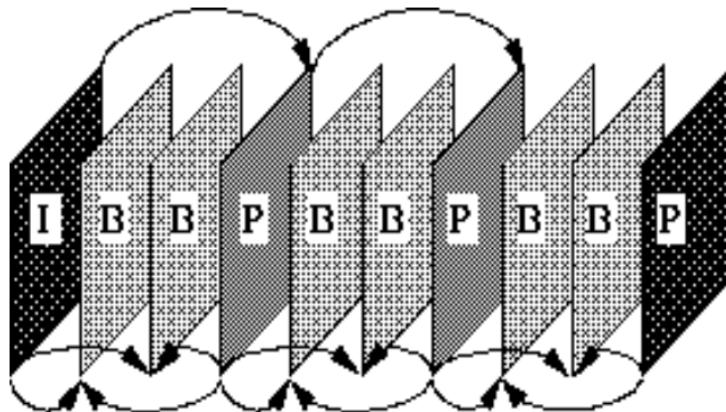


Figura 6.1: Typical GOP

Video Layer and B-Frames

Un video nello standard MPEG è diviso in una serie di livelli atti a compiere e gestire varie cose.

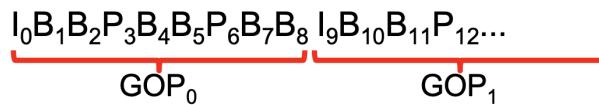
Il primo layer è chiamato *Video Sequence Layer* che rappresenta un video completo, indipendente da ciò che accade prima o dopo; il secondo è il *GOP*, il quale è composto da una combinazione dei vari frames **I**, **B** & **P**.

Dopodiché troviamo il *Pictures Layer*, in cui ogni picture viene divisa in *slices* ed ogni slice è una sequenza di macroblocchi, che a loro volta sono composti da blocchi; molto simile alla struttura dello standard **H.261**.

Ogni livello ha il proprio *byte-aligned start code* da 32 bit.

Introducendo la *forward prediction* le cose si complicano notevolmente, guardiamo questo esempio:

Esempio 6.1.1. Consideriamo la seguente sequenza di frame.



Dato che ogni frame è di tipo diverso, prima di decodificare i frames 1 e 2, dovrò lavorare sul numero 3 siccome quelli di tipo B necessitano informazioni dal successivo P-frame; questo introduce un delay e richiede inoltre della memoria all'interno dei dispositivi ed un tempo non esistevano apparecchi potenti come al giorno d'oggi.

Quello che è stato fatto allora fu cambiare l'ordine in cui i frame venivano inviati:

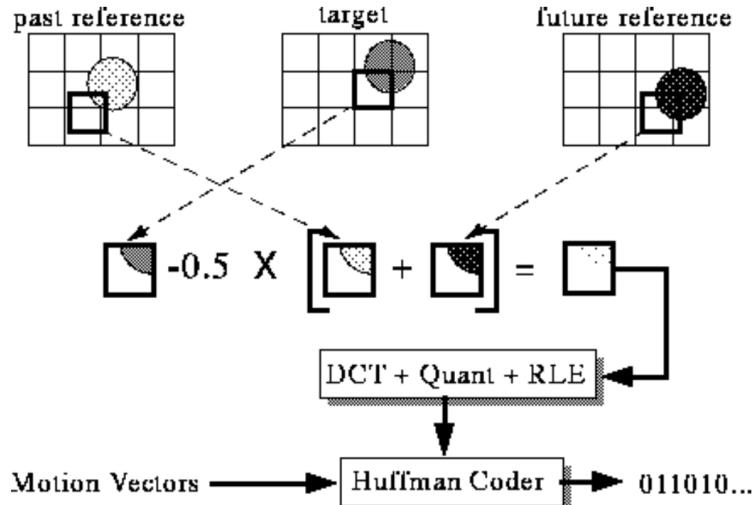
$I_0 P_3 B_1 B_2 P_6 B_4 B_5 I_9 B_7 B_8 P_{12} B_{10} B_{11} \dots$



In questo modo possiamo iniziare il decoding il prima possibile mentre stiamo ancora ricevendo informazioni; idea interessante ma molto complicata.

Il problema di questo approccio è, come possiamo vedere, che alcuni frame appartenenti al GOP_0 entrano a far parte del GOP_1 .

Bidirectional Prediction in B-Frames Per complicare le cose ancora di più, è possibile calcolare la differenza rispetto a due diverse reference, una nel passato e una nel futuro \rightarrow 2 motion vectors:



6.1.3 More Differences With H.261

Siccome ora possiamo considerare frame più "lontani" rispetto al diretto precedente, può essere che le cose siano più in movimento e questo ci porta a dover aumentare l'area di ricerca $\rightarrow \pm 32$. Inoltre i MVs possono utilizzare una half-pixel precision, ciò significa che dobbiamo interpolare il valore dei due vicini in modo da ottenere il mezzo; questo aumenta ancora di più il numero di controlli che dobbiamo eseguire.

Una cosa interessante è però che possiamo fare *Fast-Forward* senza troppi sforzi siccome siamo in grado di saltare da un I-frame al successivo e quindi non abbiamo bisogno di hardware migliore per andare più veloci.

6.2 H.264/AVC

Vediamo ora uno standard ancora più avanzato con l'obiettivo di migliorare le tecnologie già presenti.

Vediamo un elenco di ciò che è stato introdotto in questo nuovo standard:

- Una delle prime cose da notare è l'aggiunta di un supporto per l'encoding di INTRA Pictures.
- Sono stati introdotti blocchi più piccoli rispetto ai macroblocks in modo da non avere più 16×16 fixed-size macroblocchi.

- Se prima avevamo una precisione di mezzo pixel, qui siamo arrivati ad 1/4 mediante un'interpolazione pesata.
- I MVs possono andare oltre i limiti della figura in modo da migliorare la prediction per oggetti che entrano nella scena.
- È stata introdotta una predizione pesata in modo da dare diversa importanza ai vari frame precedenti e successivi.
- La classica 8×8 DCT è stata rimpiazzata da una 4×4 leggermente diversa.
- Utilizza l'**Arithmetic Entropy Coding** invece di Huffman.

6.2.1 Prediction fro INTRA Pictures

Già nel JPEG avevamo una sorta di prediction per questa situazione durante l'encoding dei DCCs, qui cerchiamo di portare questa predizione al livello successivo.

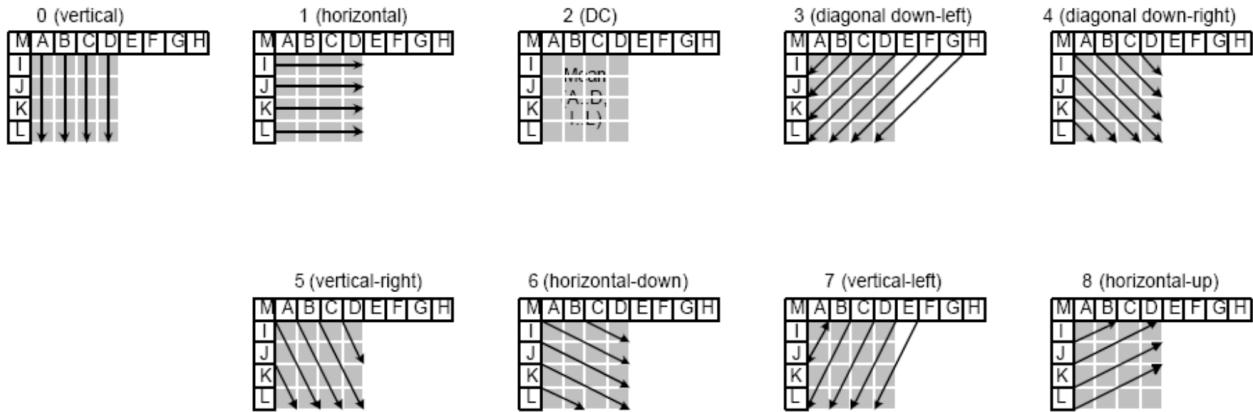


Figura 6.2: Prediction for INTRA Coding

Se invece di blocchi 4×4 vogliamo predirne di dimensione 16×16 , possiamo utilizzare solo la prima riga di metodi.

6.2.2 Variable Block Size

In **H.264** abbiamo 4 diverse possibilità per quanto riguarda la dimensione dei blocchi:

- 16×16
- 16×8
- 8×16
- 8×8

Come facciamo a scegliere quale grandezza utilizzare? L'idea è quella di partire dal più grande per calcolare poi una misura di variazione con il corrispondente al frame precedente; dopodiché proviamo una suddivisione orizzontale e se la varianza in una delle due parti è molto inferiore allora questa separazione ha senso e procediamo così:

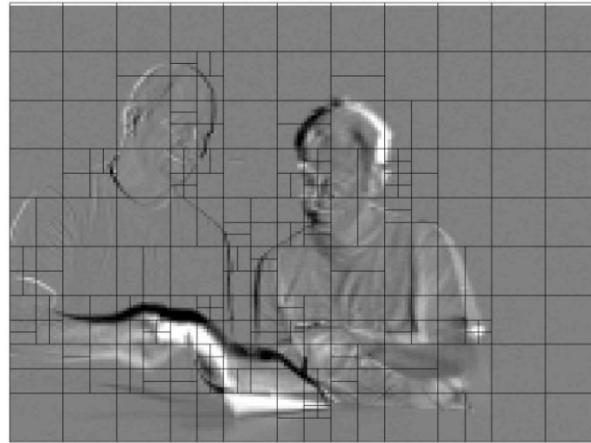


Figura 6.3: Variable Block Sizes

Vediamo che in parti statiche dell'immagine la grandezza dei blocchi rimane la classica 16×16 mentre in altre zone effettuiamo una suddivisione più fine da migliorare la prediction. Dove utilizziamo blocchi 8×8 , possiamo scegliere di effettuare un'ulteriore partizione:

- 8×4
- 4×8
- 4×4

Applicando lo stesso criterio di prima; notiamo che al massimo possiamo avere 16 diversi MVs se arriviamo a selezionare la modalità 4×4 e ciò comporta più vettori da inviare, dobbiamo stare perciò attenti al rapporto grandezza dei blocchi ed effettivo miglioramento.

6.2.3 Different DCT for Smaller Size

Dato che il nostro blocco può arrivare ad avere le dimensioni 4×4 , dobbiamo trovare un modo per applicare la DCT a questa nuova misura; c'è dunque un'approssimazione molto buona che ci consente di utilizzare pesi interi:

$$H = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 2 & 1 & -1 & -2 \\ 1 & -1 & -1 & 1 \\ 1 & -2 & 2 & -1 \end{bmatrix}$$

La parte decimale è stata dunque sposta in una *post-scaling matrix* in modo poter eseguire le operazioni fondamentali solamente con interi così da renderla molto veloce.

Per applicare questa cosa a blocchi più grandi possiamo dividerli prima in sottoblocchi, applicare la trasformata, prendere i DCCs di ogni miniblocco e ricomporre una nuova matrice.

Capitolo 7

The Sound

Per **Suono** consideriamo una variazione ciclica della pressione che arriva alle nostre orecchie, esso richiede un mezzo di propagazione che può essere l'aria, l'acqua e così via.

Come fa però questa variazione a viaggiare attraverso l'aria? Sfrutta le multiple collisioni delle sue particelle. È in questo modo che funzionano gli altoparlanti: la loro membrana si muove avanti e indietro comprimendo e decomprimendo l'aria in fronte a se; ovviamente questo movimento non è istantaneo per tutte le particelle d'aria davanti allo speaker, in base a certe proprietà fisiche queste variazioni saranno percepite ad una certa distanza dopo un tot di tempo → 344 metri in un secondo circa.

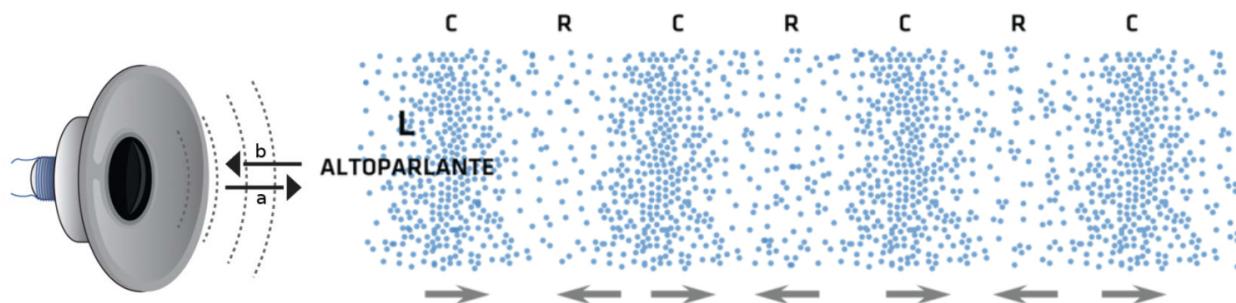


Figura 7.1: Air Particles Via Speaker

Ovviamente non sarà la stessa particella a dover compiere tutta la distanza, essa infatti arriverà fino ad urtare la prossima e così via; per questo motivo dopo un certo spazio la variazione di pressione si assottiglia a tal punto da non essere più percepibile.

Ciò che rende il suono udibile è il fatto che subito dopo aver iniziato a spingere alcune particelle in avanti inizio a richiamarne indietro causando una decompressione; questa struttura può essere visto come un'onda che cambia il suo valore di pressione nel tempo:

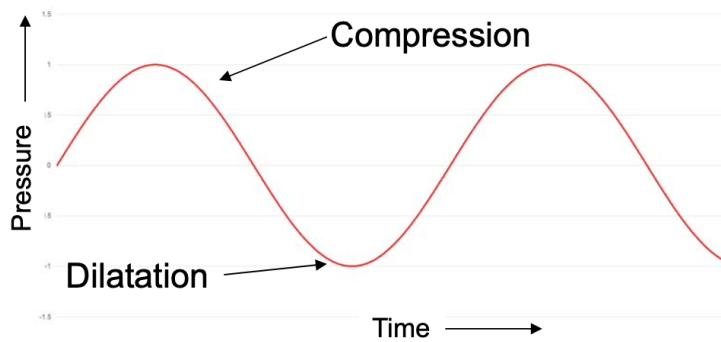


Figura 7.2: Sound Wave

7.1 Sound Waves

Iniziamo a parlare dunque di onde sonore, ossia pressione che viaggia nello spazio sotto forma di sequenza di compressioni e decompressioni; questo movimento produce energia

che si muove dalla sorgente in avanti ma la materia, se misuriamo la sua posizione in media, rimane nello stesso punto siccome viene spinta e tirata indietro in modo ciclico. Le onde sonore sono dunque dette **onde longitudinali** siccome la compressione si muove lungo la stessa direzione verso la quale si propagano.

il suono dunque è un'onda longitudinale generata da successive compressioni e decompressioni del mezzo di trasporto attraverso il quale si propaga.

7.1.1 Sound Properties

Grazie alla Trasformata di Fourier, possiamo descrivere il suono attraverso una combinazione di sinusoidi; una singola sinusoide può essere descritta nel seguente modo:

$$y = A \sin(2\pi ft)$$

In cui:

- **A**: Indica l'ampiezza della cruna, ossia ci dice quanto è grande la quantità di movimento, ci dà l'idea della forza con cui stiamo spingendo e tirando l'aria.
- $\sin(2\pi ft)$: Ogni secondo abbiamo un'onda intera descritta dal fatto che è presente un 2π all'interno dell'angolo; mentre f indica la frequenza ossia quante onde ogni secondo, ossia quanto velocemente stiamo spingendo l'aria; è ciò che governa la nostra percezione di bassi, medi ed alti.

Frequency

La frequenza dunque è il numero di cicli che l'onda compie ogni secondo, è misurata in **Hertz** → **Hz** avente la grandezza fisica di s^{-1} .

Esempio 7.1.1. Vediamo un esempio di onda con frequenza **5Hz**, ossia che compie 5 cicli al secondo.

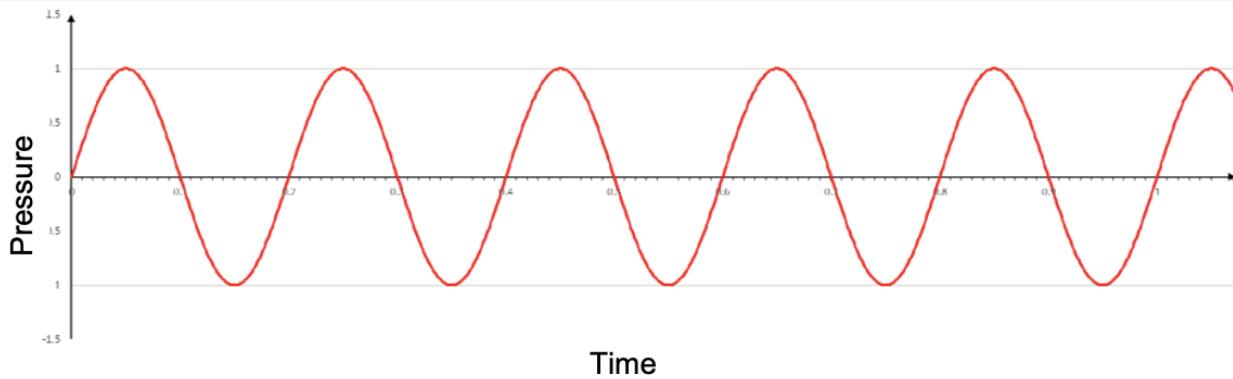


Figura 7.3: 5Hz Wave

Wave Length

Per *wave length* intendiamo la grandezza in metri della lunghezza di un periodo, ossiamo quanto spazio viene fatto tra due punti di massimo dell'onda e si ottiene dalla seguente relazione:

$$\lambda = \frac{c}{f}$$

Come possiamo vedere tutto dipende dalla velocità del suono nel mezzo di propagazione.

Avevamo visto che la velocità del suono è circa **344m/s**, questo significa che un'onda con frequenza di 1Hz, percorrerà 344m in un secondo; l'orecchio umano inizia a percepire qualcosa da frequenze di 20/30Hz in su, che corrispondono ad una lunghezza d'onda di 15/18 metri.

Phase

Per *phase* intendiamo la distanza tra due punti corrispondenti di due diverse forme d'onda nel tempo, in altre parole la quantità di tempo che dobbiamo attendere affinché l'altra wave prenda la stessa forma. Questa proprietà è ciò che permette di amplificare il suono ed è alla base della risonanza degli strumenti musicali.

7.2 Sound Pressure

Ciò che a noi interessa descrivere bene è la pressione siccome come abbiamo detto il suono è un cambiamento nella sound pressure.

Partiamo dal presupposto che siamo sempre in presenza di pressione, c'è l'atmosfera che applica costantemente pressione su di noi ma non ce ne accorgiamo in quanto ci comprime in tutte le direzioni; definiamo il silenzio, lo 0 per intenderci, quando non abbiamo variazione rispetto alla pressione atmosferica $\rightarrow 10^5 \text{ Pa}$.

In momenti in cui udiamo qualcosa, la differenza varia molto velocemente perciò per esprimere la quantità di pressione in un dato intervallo di tempo, utilizziamo l'**RMS-Root Mean Square Error** che ci fornisce ciò che viene denominata *Effective Sound Pressure* - p , più alto questo valore è più "rumoroso" sarà il suono.

La nostra percezione del suono non segue una relazione lineare e dunque di solito non utilizziamo tale misura ma al suo posto consideriamo l'**SPL - Sound Pressure Level** - L_p , la quale è una misura logaritmica calcolata considerando il valore di p :

$$L_p = 20 \log_{10} \frac{p}{p_0} \quad [\text{dB}]$$

In cui p_0 è la *reference sound pressure* $\sim 20 \mu\text{Pa}$.

7.2.1 Microphones

Ma duqne come facciamo noi a misurare questo cambiamento di pressione? Ci serve qualcosa per trasformare l'energia acustica in energia elettrica.

Electrodynamic Microphone

Un primo esempio è l'*Electrodynamic Microphone*, basato sul movimento di una parte elettromagnetica in grado di lavorare insieme ad una membrana oscillante, detta **diaframma**, capace di raccogliere i cambiamenti nell'aria.

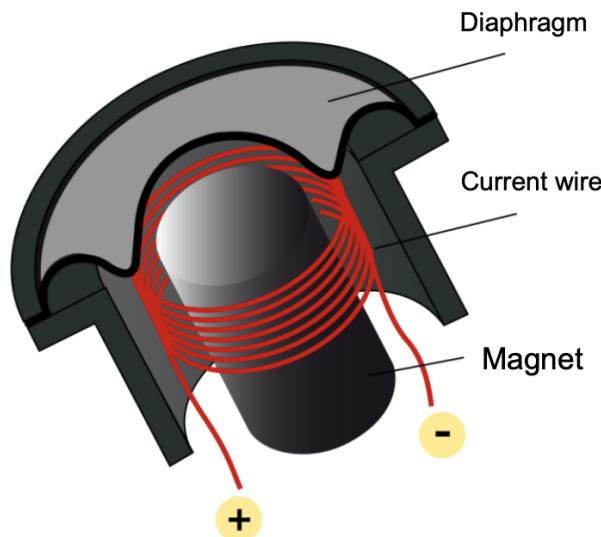


Figura 7.4: Microphone Schema

Il movimento del diaframma è collegato a quello di un elemento con proprietà elettromagnetiche come ad esempio un piccolo pezzo di metallo magnetizzato che, muovendosi

all'interno di una spirale di filo conduttore, genera una differenza di potenziale ai suoi capi in modo da poterla misurare nel tempo.

Questi dispostvi sono molto resistenti, rendendoli perfetti per catturare *low-kick* della batteria oppure suoni molto forti; per costruzione sono molto adatti per catturare vibazioni non troppo veloci, siccome sono composti da pezzi che si devono muovere fisicamente dovendo sottostare alle leggi sulla quantità di moto, limitando la velocità con cui possiamo cambiarne la direzione di movimento → sono dunque adatti a suoni forti ma a frequenze limitate.

Condenser Microphone

Un altro tipo di microfono è il *Condenser Microphone*, conosciuto anche come *Electrostatic Microphone*, il quale al posto di avere una componente elettromagnetica mobile, si basa sulle proprietà elettromagnetiche dei condensatori; una volta costruito il dispositivo iniziamo a caricarlo e sappiamo che la tensione seguirà questa relazione:

$$V = \frac{Q}{C}$$

Se cambiamo la distanza tra le armature stiamo agendo sulla variabile **C** in modo istantaneo producendo cambiamenti istantanei nella **V**.

Abbiamo detto che inizialmente dobbiamo applicare un voltaggio al condensatore, denominato *Phantom Power*; questo tipo di microfoni sono molto sensibili e riproducono il suono in maniera fedele, c'è da stare attenti perché non sono resistenti come quelli precedenti e quidni non sono adatti a catturare variazioni di pressione molto forti.

Polar Patterns

Indipendentemente dal tipo di microfono, ognuno di essi possiede ciò che viene definito *Polar Pattern* che indica le zone in cui il microfono è in grado di percepire il suono:

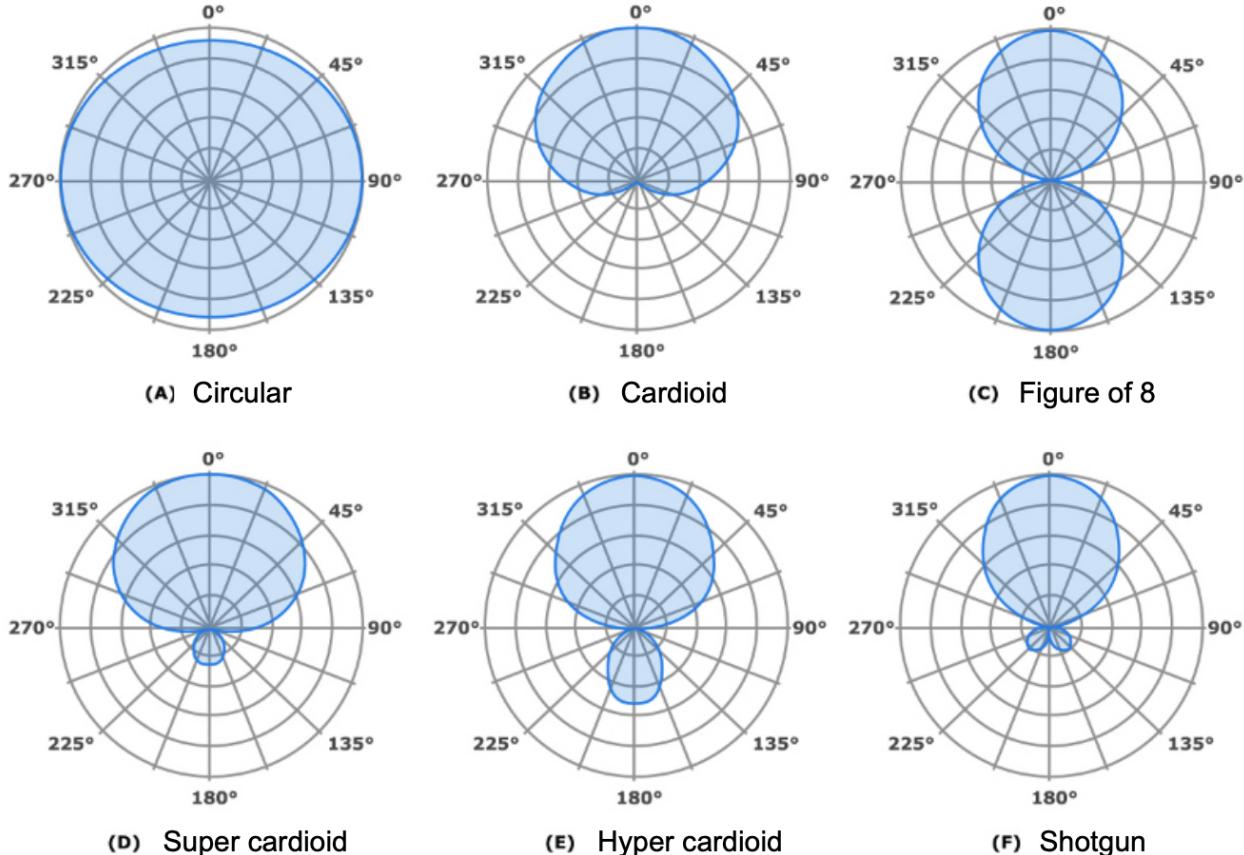


Figura 7.5: Polar Patterns

Microphones Parameters

Normalmente per descrivere la qualità di un microfono viene utilizzata la sua sensibilità, ossia l'abilità di convertire il suono in elettricità, in quanto se la variazione in tensione prodotta è piccola deve essere amplificata, portando però anche un'amplificazione degli errori. Per misurare questo parametro si inizia generando un'onda con frequenza di 1kHz e un SPL di 94dB alla distanza di 1 metro, si mette il microfono davanti alla sorgente e si misura il voltaggio prodotto, in mV, per poi essere convertito in dBV:

$$Sensitivity_{dBV} = 10 \log_{10} \left(\frac{Sensitivity_{mV/Pa}}{1000mV/Pa} \right) [dBV]$$

E siccome è impossibile superare il valore di riferimento al denominatore, avremo sempre un valore negativo.

Un altro parametro interessante è la risposta in frequenza, attraverso un diagramma di Bode:

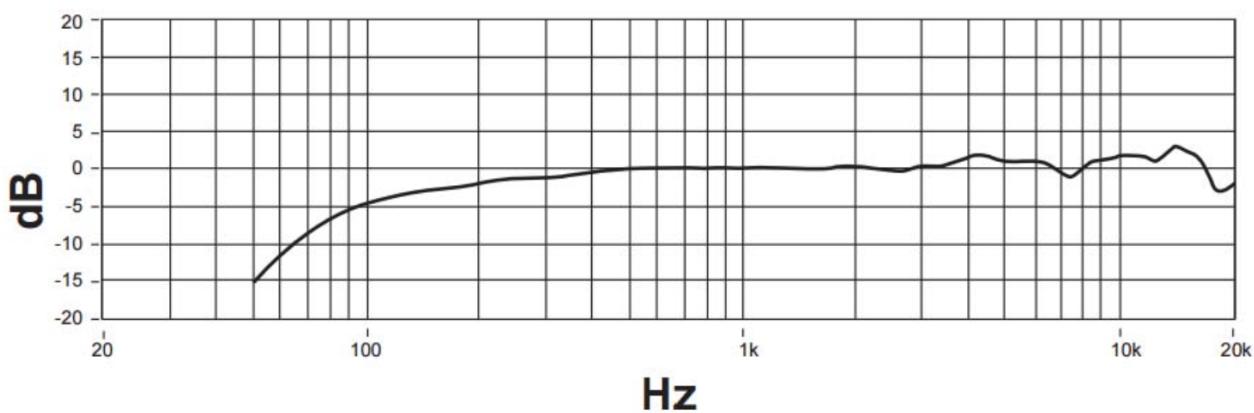


Figura 7.6: Frequency Response

Il grafico avrà sempre lo 0 in corrispondenza di 1kHz, valore di riferimento per la misura.

7.2.2 Sound Reproduction

Abbiamo parlato di come acquisire il suono, ma come facciamo a riprodurlo? In pratica attuiamo il contrario di ciò che abbiamo fatto con il primo tipo di microfono:

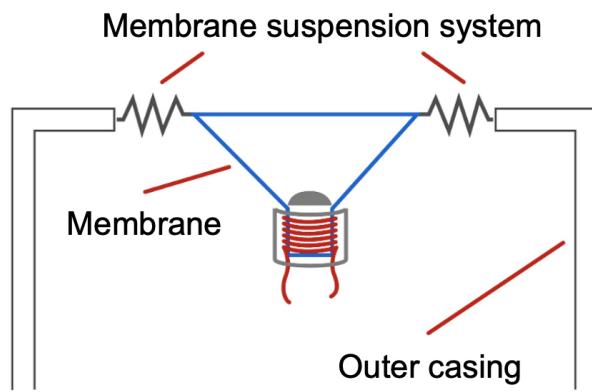


Figura 7.7: Loud Speaker Schema

7.3 Digital Audio

Vediamo la versione digitale del suono ora, come facciamo a tradurre la sequenza di variazioni in qualcosa di capibile da una macchina? Dobbiamo campionare il segnale analogico per convertirlo in un numero.

Ci ricordiamo prima di tutto del teorema di **Nyquist**, il quale afferma che la frequenza di campionamento deve essere almeno il doppio della banda del segnale, di solito 44kHz, ma per avere un po' di libertà in più siamo a 44100Hz; se non rispettiamo questa regola ciò che otteniamo è aliasing, ossia suoni fasulli.

Capitolo 8

Sound Perception

Abbiamo discusso le nozioni principali che ci hanno fornito un'idea di cosa sia il suono, dobbiamo ora cercare di capire come è possibile per un essere umano percepire le variazioni di pressione in modo da poter sentire qualcosa.

8.1 The Human Ear

L'orecchio umano fa da **trasduttore**, dato che trasforma l'energia acustica prima in energia meccanica e successivamente in energia elettrica, in modo da poter trasmettere impulsi al nostro cervello; il sistema uditivo è composto da 3 parti principali:

1. Outer Ear

2. Middle Ear

3. Inner Ear

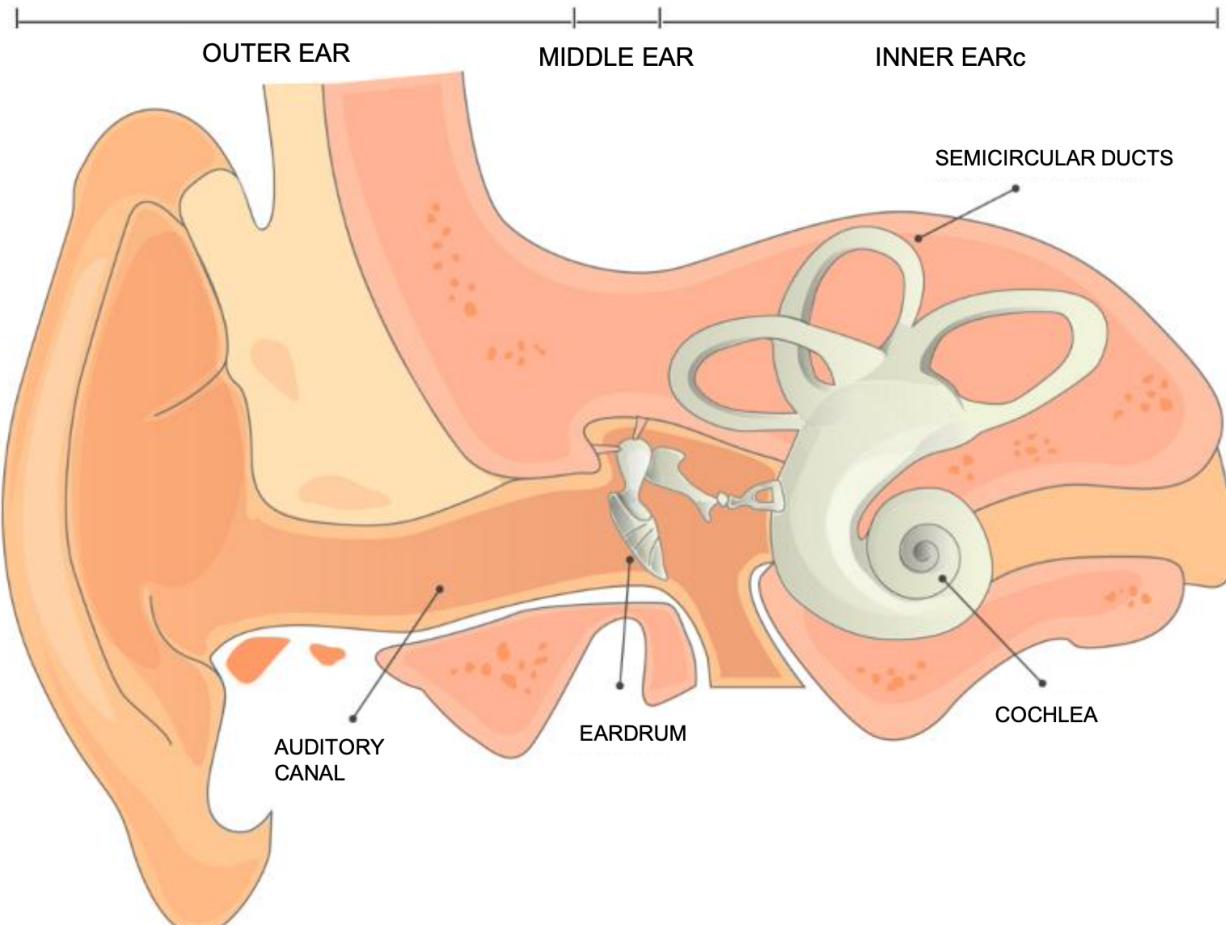


Figura 8.1: Human Ear

8.1.1 Outer Ear

La prima parte che compone questo pezzo di orecchio è detta **padiglione auricolare**, la quale si occupa di riflettere le onde sonore captate per convogliarle al centro, verso il **canale uditivo**.

8.1.2 Middle Ear

Il canale uditivo termina a contatto con una membrana chiamata **timpano**, la quale vibra a seconda delle vibrazioni che ha ricevuto dall'esterno; dalla parte opposta sono presenti 3 piccole ossa: **martello, incudine e staffa**, che hanno la funzione di amplificare le vibrazioni del timpano per poi trasmetterle alla **coclea**. (Questa amplificazione è necessaria in quanto il timpano è una membrana sottile a contatto con aria mentre la coclea è piena di un fluido denso).

8.1.3 Inner Ear

Ogni parte della coclea fa risuonare maggiormente una certa frequenza che poi sarà convertita in energia elettrica grazie all'**Organo del Corti**, il quale al suo interno presenta circa 4000 "ciglia" che vibrano in base alla vibrazione del fluido; queste ciglia sono connesse in gruppi (ogni gruppo non è uniforme) ed ognuno dei quali è collegato alle relative terminazioni nervose. Questa divisione è ciò che porta al concetto di **critical band**.

Critical Band

Questo fenomeno è detto **masking phenomenon** ed è alla base di moltissimi algoritmi di compressione audio: se due suoni cadono in bande critiche differenti allora l'uno non interferirà con l'altro cosa che invece, nel caso contrario, non succede ed otterremo una singola frequenza che rappresenterà entrambe.

8.2 Anechoic Chamber

Abbiamo detto che non siamo in grado di percepire tutti i suoni allo stesso modo e per questo motivo sono state ideate delle stanze che permettono di studiare il suono senza alcuna interferenza in quanto le pareti di questa stanza sono in grado di assorbire tutte le vibrazioni senza mandarne indietro alcuna.

8.2.1 Isophonic Curves

Per prima cosa abbiamo una sorgente sonora capace di emettere onde sinusoidale a frequenza variabile ma ampiezza costante: impostiamo l'ampiezza ad un certo **SPL**, ad esempio 80 dB; cambiando la frequenza udiremo suoni più forti o più deboli anche se la "potenza" è la stessa, appunto perché non percepiamo le diverse frequenze nella stessa maniera.

Una **Isophonic Curve** ci indica quanti dB necessitiamo per percepire lo stesso suono con la stessa intensità per frequenze diverse.

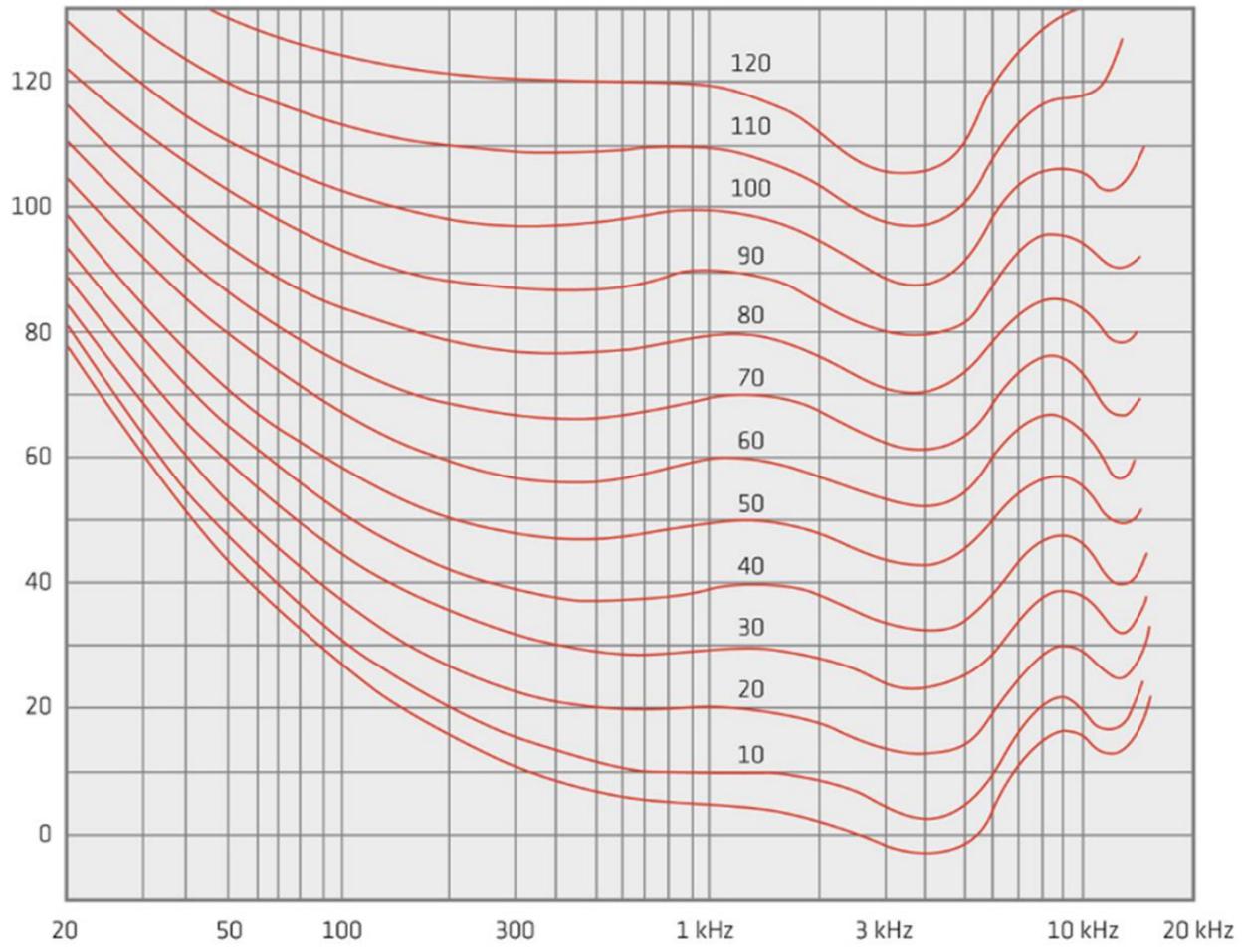


Figura 8.2: Isophonic Curves

L'ultima di queste curve è detta **audibility threshold**.

8.3 Masking

Il fenomeno del masking avviene quando due suoni che ricadono nella stessa banda critica vengono riprodotti simultaneamente: due frequenze sono percepite nello stesso momento e nella stessa zona dell'orecchio; questo fa sì che il suono più alto vada a mascherare la percezione del più basso aumentando l'**audibility threshold** spingendola verso il suo centro:

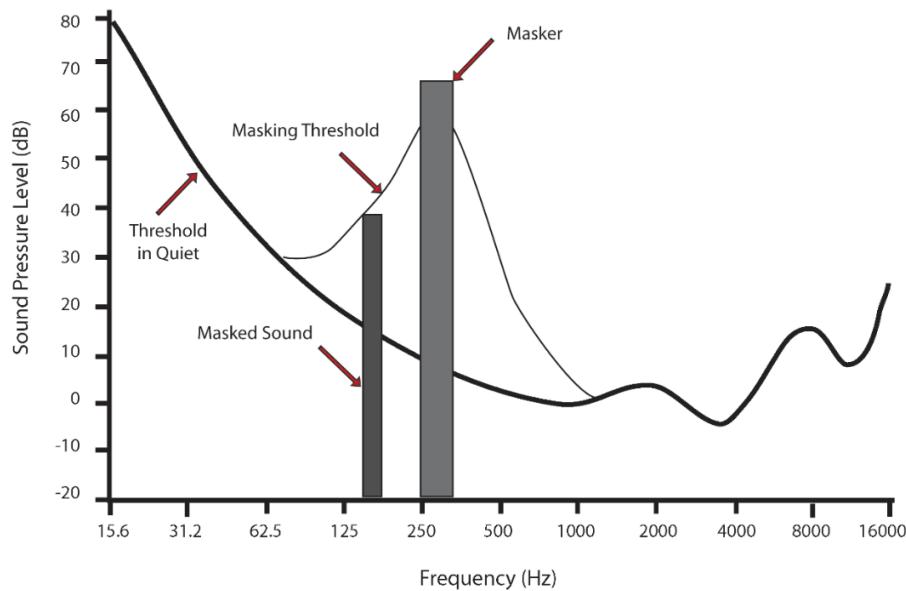


Figura 8.3: Human Ear

Questo fenomeno è detto **Simultaneous Masking**.

8.3.1 Temporal Masking

Esiste un altro tipo di masking detto **Temporal Masking** che avviene quando un suono forte ci rende "sordi" alle medesime frequenze per un periodo di tempo dopo averlo udito: questo ci dice che non comprimiamo i dati istante per istante ma dobbiamo tenere conto di ciò che accade nella linea temporale.

Capitolo 9

AAC

AAC sta per **Advanced Audio Coding**, il quale rappresenta uno standard lossy per la compressione di file audio.

Questo standard può lavorare con segnali campionati con un rate $8 \div 96$ kHz e supporta fino 48 canali normali ed altri 16 addizionali per le basse frequenze; inoltre può lavorare sia con bitrate costanti che variabili. AAC sfrutta tecniche psicoacustiche in modo da riuscire ad ottenere risultati come ad esempio *block switching, prediction, temporal noise shaping, masking effects, non-uniform quantization...*

9.1 Perceptual Audio Coding

Un sistema di encoding/decoding di tipo *perceptual* è tipicamente composto dai seguenti blocchi:

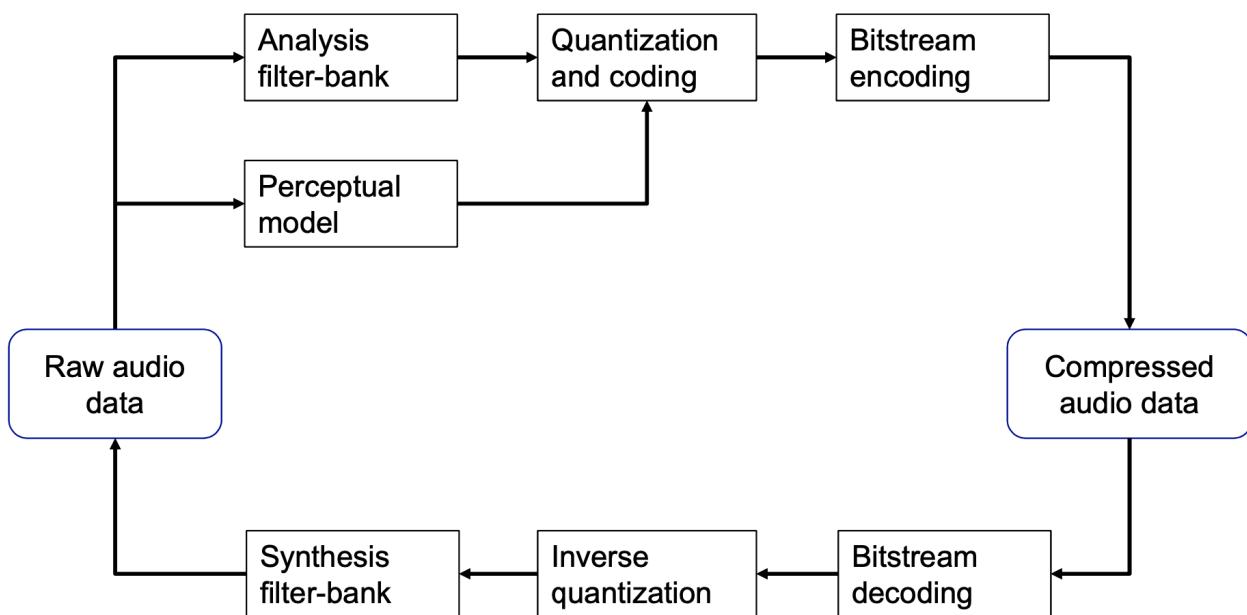


Figura 9.1: Isophonic Curves

Un fenomeno che avviene è denominato **quantization noise**, una nuova onda creata dalla differenza del segnale originale e quello quantizzato; questa cosa non è accettabile siccome è troppo percepibile e dunque vogliamo dare una forma al rumore in modo da poterlo "nascondere". Ciò che vogliamo fare è ottenere un segnale quantizzato a partire da un segnale "raw":

$$x_q(n) = x(n) + e(n)$$

In modo tale che

$$\arg \min_{e(n)} PE(x(n), e(n))$$

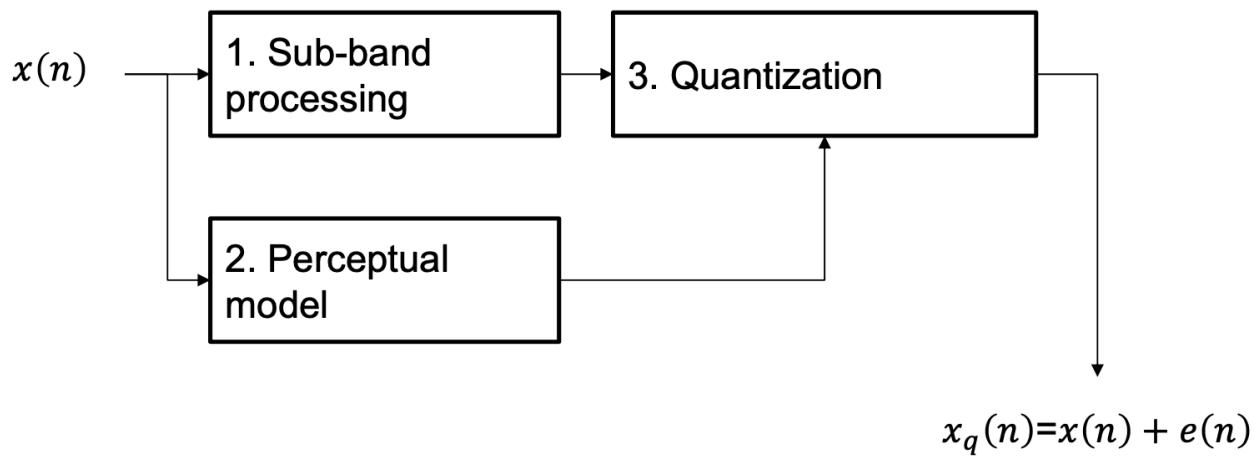


Figura 9.2: Quantization Noise Problem

Come possiamo fare ciò?

9.1.1 Sub-Band Processing

La prima cosa che dobbiamo fare è spostarci nel dominio della frequenza attraverso una versione modificata della DCT chiamata *MDCT: Modified Discrete Cosine Transform*:

$$X_k = \sum_{n=0}^{2N-1} x_n w_n \cos \left[\frac{\pi}{N} \left(n + \frac{1}{2} + \frac{N}{2} \right) \left(k + \frac{1}{2} \right) \right] \quad k \in [0, N-1]$$

Riceviamo un segnale x e ne prendiamo una finestra da 0 a $2N - 1$ (N siccome possiamo avere finestre variabili) in modo da avere $2N$ campioni a cui applicare la trasformata con le frequenze $\in [0, N - 1]$ così da ottenere la metà degli elementi.

La sua inversa, *IMDCT*:

$$y_n = \frac{2}{N} w_n \sum_{k=0}^{N-1} X_k \cos \left[\frac{\pi}{N} \left(n + \frac{1}{2} + \frac{N}{2} \right) \left(k + \frac{1}{2} \right) \right] \quad k \in [0, 2N-1]$$

In modo da tornare ad avere il doppio dei valori, ossia quelli originali.

Si tratta di una *lapped transform*, ossia è calcolata con blocchi soveapposti del segnale di ingresso.

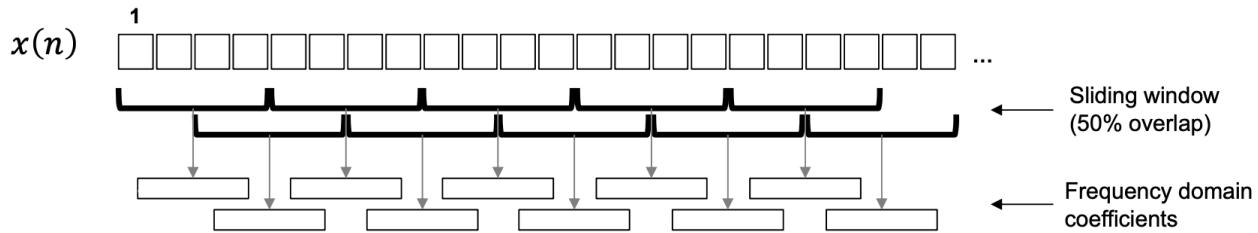


Figura 9.3: Quantization Noise Problem

Come comprimiamo dunque?

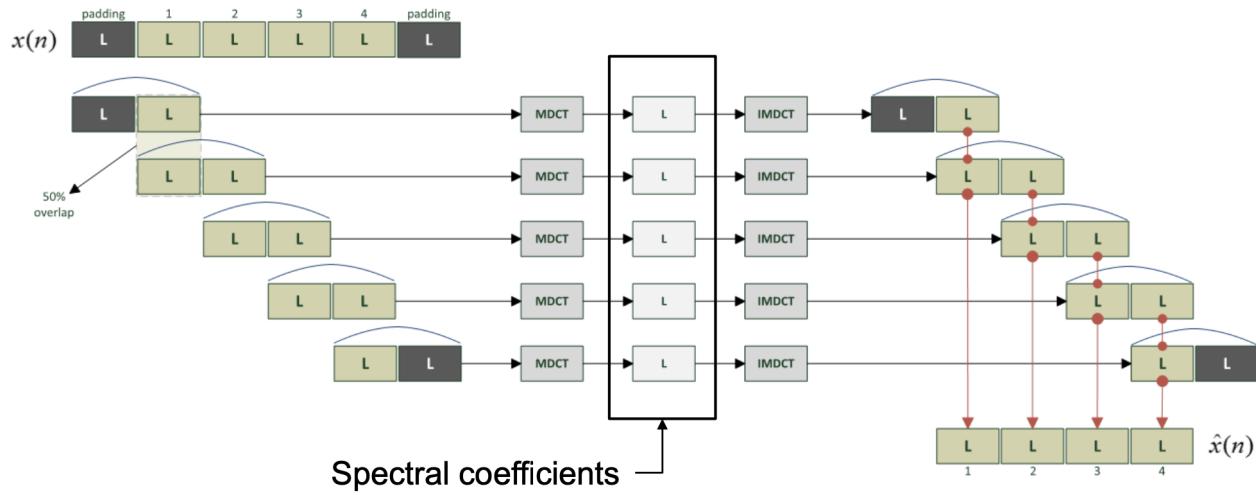


Figura 9.4: Analysis/Synthesis via MDCT

Per prima cosa dobbiamo applicare un padding corrispondente alla metà della finestra sia all'inizio che alla fine del segnale; da qui trasformiamo insieme due L ottenendone uno in uscita e proseguiamo spostando del 50% la finestra in avanti.

Un encoder AAC processa 2048 campioni audio, che costituiscono un frame, alla volta ma in certi casi possiamo utilizzare una finestra di 256 andando a produrre 8×128 coefficienti in modo da evitare il *pre-echo effect*. La finestra utilizzata può essere di due tipi, ci concentriamo principalmente su quella sinusoidale:

$$w_n = \sin \left[\frac{\pi}{2N} \left(n + \frac{1}{2} \right) \right], \quad n \in [0, 2N - 1]$$

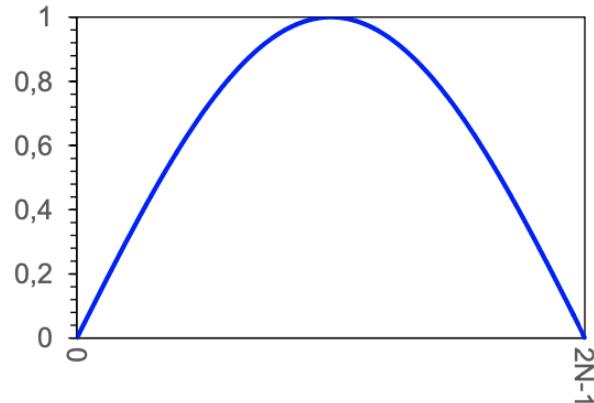


Figura 9.5: Sine Window

9.1.2 Perceptual Model

Abbiamo parlato della parte matematica, ora vediamo come integrare le parti relative alla psicoacustica: integriamo conoscenze a priori sull'acustica umana tenendo conto di vari fattori come ad esempio *Absolute Threshold*, *Critical Bands*, *Tone-Masking-Noise*, *Noise-Masking-Tone*, *Spread of Masking*...

Absolute Threshold of Hearing

Consideriamo il minor livello sono che possiamo udire:

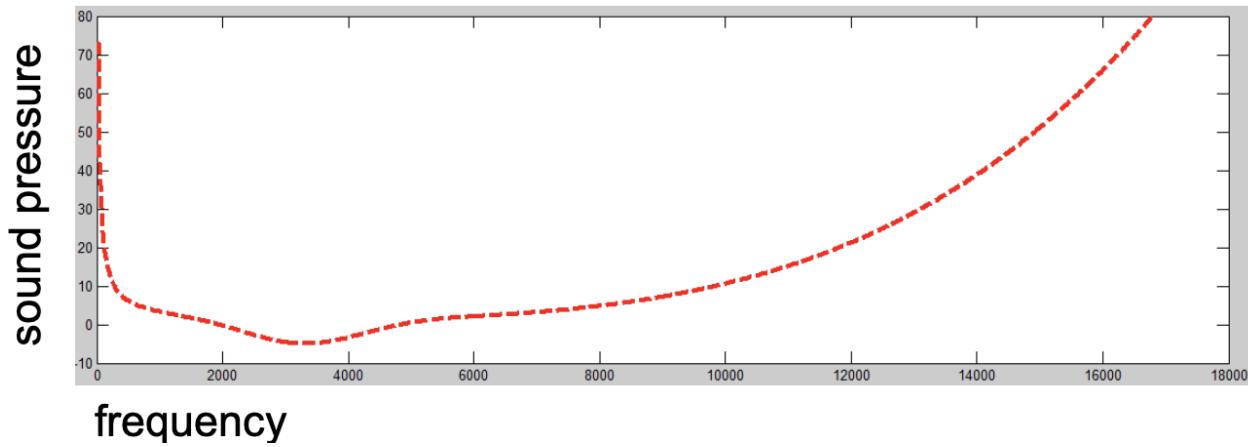


Figura 9.6: Analysis/Synthesis via MDCT

Se qualcosa sta al di sotto non possiamo udirlo e dunque dobbiamo tenere il segnale rumoroso al di sotto di tale soglia.

Tone-Masking-Noise / Noise-Masking-Tone

Abbiamo un gruppo di casi in cui un tono, ossia una nota un piccolo e corto "elemento della frequenza", può mascherare del rumore e viceversa; sappiamo che un tono può mascherare un rumore che è $\sim 24\text{dB}$ più basso, mentre per il caso inverso la soglia è $\sim 4\text{dB}$, il rumore è molto più forte nel masking.

Capitolo 10

Containers

In questo capitolo parleremo di *Container Formats*, nati dalla necessità di combinare tra di loro più tipi di dato come ad esempio audio, video, sottotitoli e così via; un container nel nostro caso è un file che ci permette di memorizzare stream multipli in maniera non continua.

Eistono diversi container come ad esempio *AVI*, *MP4*, *M4A* e, nel nostro caso, ci concentreremo sul *MKV*: *Matroska*.

10.1 Extensible Binary Meta Language

Ogni byte viene chiamato ottetto e così come XML o JSON, la sintassi e la semantica sono separate: ogni file deve avere 1 e 1 solo header, il quale permette di descrivere il resto del file e quindi di scegliere "l'interpretazione" corretta. I tipi di dato supportati da questo meta-linguaggio sono:

- **Signed Integer:** Big-Endian, da 1 a 8 ottetti.
- **Unsigned Integer.**
- **Float:** Big-Endian, 4 o 8 ottetti → 32,64 bit.
- **String:** ASCII.
- **UTF-8.** Unicode.
- **Date.**
- **Master Element.**
- **Binary**

10.1.1 EBML Element

Ogni elemento ha la seguente struttura:

`< ID > < size > < data >`

ID e size sono memorizzati come integers a grandezza variabile: in particolare il campo *ID* può essere rappresentato con un numero di byte compreso tra 1 e 4:

Width	Size	Representation
1	2^7	1xxx xxxx
2	2^{14}	01xx xxxx xxxx xxxx
3	2^{21}	001x xxxx xxxx xxxx xxxx xxxx
4	2^{28}	0001 xxxx xxxx xxxx xxxx xxxx xxxx xxxx

Figura 10.1: EMLB ID

Ed è obbligatorio rappresentare i valori con meno bytes possibili e le altre combinazioni per rappresentare lo stesso valore non sono valide, riducendo le possibilità a:

Class	Base	Width	Number of IDs
A	0x8x	1	$2^7-2 = 126$
B	0x4x	2	$2^{14}-2^7 = 16 \text{ } 256$
C	0x2x	3	$2^{21}-2^{14} = 2 \text{ } 080 \text{ } 768$
D	0x1x	4	$2^{28}-2^{21} = 266 \text{ } 338 \text{ } 304$

Figura 10.2: EBML ID Combinations

Per quanto riguarda il campo "size", può essere rappresentato mediante l'uso di byte in numero da 1 a 8:

bits, big-endian	
1xxx xxxx	- 0 to 2^7-2
01xx xxxx xxxx xxxx	- 0 to $2^{14}-2$
001x xxxx xxxx xxxx xxxx	- 0 to $2^{21}-2$
0001 xxxx xxxx xxxx xxxx xxxx	- 0 to $2^{28}-2$
0000 1xxx xxxx xxxx xxxx xxxx xxxx	- 0 to $2^{35}-2$
0000 01xx xxxx xxxx xxxx xxxx xxxx xxxx	- 0 to $2^{42}-2$
0000 001x xxxx xxxx xxxx xxxx xxxx xxxx xxxx	- 0 to $2^{49}-2$
0000 0001 xxxx xxxx xxxx xxxx xxxx xxxx xxxx xxxx	- 0 to $2^{56}-2$

Figura 10.3: EBML Size

10.2 Matroska

Questo formato, basato su EBML, ci permette di memorizzare un numero arbitrario di *tracks* ed al livello più alto ogni file è composto da due elementi: *header* e *segment*:

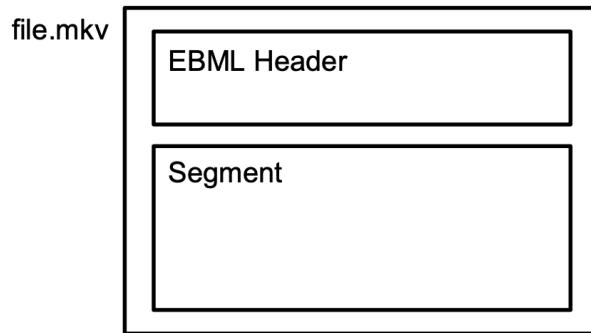


Figura 10.4: MKV level 0

10.2.1 Segment

La parte segment contiene tutti gli elementi di livello 1:

SeekHead

Questo elemento è un indice che ci dice dove sono tutti gli elementi all'interno del file; non è obbligatorio e serve al decoder per muoversi nel file senza la necessità di decodificare tutto ciò che incontra:

< SeekID > < SeekPosition >

Tracks

Questo segmento contiene informazioni su tutte le tracce presenti nel file:

TrackNumber: track number used later in clusters.

TrackUID: a unique identifier for the track.

TrackType: an 8-bit integer for the track type (1: video, 2: audio, 3: complex=audio and video combined, 0x10: logo, 0x11: subtitle, 0x12: buttons, 0x20: control)

Language: the language of the track.

CodecID: the identifier of the codec used to encode the track; the format specifications indicate the IDs available for the various codecs.

CodecPrivate: specific additional information of the codec.

Video: if the track is video, this master element contains all the video settings.

Audio: if the track is audio, this master element contains all the audio settings.

Figura 10.5: Tracks Segment

Block

I blocchi utilizzano un header non EBML e contengono il *Track Number*, il *Timecode* relativo al cluster ed un insieme di *Flags*