COMPRESIÓN PERCEPTUAL DE AUDIO DIGITAL

TESIS

Que para obtener el grado de

MAESTRO EN CIENCIAS EN INGENIERÍA ELÉCTRICA

presenta

Luis Eduardo Gamboa Guzmán

Félix Calderón Solorio

Director de Tesis

Universidad Michoacana de San Nicolás de Hidalgo

Noviembre 2005

Resumen

En este documento se presenta la implementación de un codificador-decodificador perceptual de sonido que toma como entrada un archivo sin comprimir, muestreado a 44100Hz, cuantizado a 16-bits, estéreo. La señal es procesada en bloques de 512 muestras, y transformada utilizando Transformada Discreta de Coseno Modificada. Estos coeficientes son cuantizados utilizando información de un modelo psicoacústico y posteriormente empaquetados a un archivo utilizando tasa de datos variable. El decodificador toma este archivo y su salida es un archivo con las mismas características que el de origen. Las tasas de datos promedio observadas varían entre 58kbps y 340kbps, estando la más común localizada por arriba de 240kbps. Se muestran resultados de pruebas comparativas de calidad subjetiva entre el formato implementado, MPEG-1 Layer 3 y MPEG-4 Low Complexity Advanced Audio Coding. El codificador-decodificador implementado logra un perfil de alta calidad de acuerdo a los resultados obtenidos.

Abstract

This document presents the implementation of a perceptual CODEC that uses as input an uncompressed file sampled at 44100Hz, quantized at 16-bits, stereo. The waveform is processed in blocks of 512 samples and transformed using Modified Discrete Cosine Transform. These coefficients are quantized based on the information gathered from a psychoacoustic model and packed into a variable-bit-rate file. The decoder uses this file as input and outputs a file with the same format as the original input. The average data rate observed ranged from 58kbps and 340kbps, with data rates around 240kbps being the most common. Results are presented regarding the subjective quality measurement of the implemented format, MPEG-1 Layer 3 and MPEG-4 Low Complexity Advanced Audio Coding. Based on the results of the tests, the CODEC achieves a high quality profile.

Contenido

Resumen . . . iii
Abstract . . . v
Contenido . . . vii
Lista de Figuras . . . xi
Lista de Tablas . . . xiii
Lista de Símbolos . . . xv

1. Introducción 1
1.1. Planteamiento del Problema . . . 1
1.2. Antecedentes . . . 2
1.2.1. Codificación y Compresión de Audio . . . 2
1.3. Objetivos de la Tesis . . . 4
1.4. Descripción de Capítulos . . . 5

2. Audio Digital 7
2.1. Muestreo . . . 7
2.2. Cuantización . . . 9
2.2.1. Cuantizadores Uniformes . . . 9
2.2.2. Cuantizadores No Uniformes . . . 11
2.3. Codificadores de Audio . . . 13
2.4. Calidad objetiva de la representación Digital del Audio . . . 13
2.5. Conclusiones . . . 14

3. Codificación Perceptual 15
3.1. Transformada Discreta de Coseno Modificada . . . 18
3.1.1. Análisis y Síntesis utilizando la MDCT . . . 18
3.1.2. Cálculo de la MDCT e IMDCT . . . 19
3.1.3. Ventanas válidas para la MDCT . . . 22
3.1.4. Implementación rápida de la MDCT e IMDCT . . . 24
3.1.5. La MDCT y la codificación perceptual . . . 27
3.2. Psicoacústica . . . 27
3.2.1. La escala decibel . . . 27
3.2.2. Umbral auditivo . . . 28
3.2.3. Enmascaramiento simultaneo . . . 29

3.2.4. Enmascaramiento temporal 32
3.2.5. Umbral global de enmascaramiento 34
3.3. Nivel de Presión de Sonido y el Procesamiento Digital de Señales 35
3.4. Un ejemplo del proceso de cuantización con un modelo perceptual 36
3.5. Codificación Aritmética 41
3.6. Conclusiones 43

4. Implementación de Un Esquema de Codificación 45
4.1. Características del CODEC Implementado 45
4.2. Proceso de Codificación 46
4.2.1. Cambio de rango dinámico de la señal 46
4.2.2. Cálculo de la Transformada Discreta de Fourier 46
4.2.3. Modelo Psicoacústico 48
4.3. Cálculo de la MDCT 49
4.3.1. Llevar los coeficientes a decibelios 49
4.4. Asignación de bits 49
4.5. Cuantización 50
4.6. Formato de la salida 50
4.6.1. Cabecera de Archivo 51
4.6.2. Paquete 52
4.7. Decodificación 54
4.8. Conclusiones 54

5. Resultados 57
5.1. Rendimiento 57
5.2. Pruebas de calidad perceptual 58
5.2.1. Castañuelas - Castanets 60
5.2.2. Música Clásica 60
5.2.3. Música Disco 61
5.2.4. Chasquidos de Dedos - Finger Snaps 61
5.2.5. Cornos Franceses - French Horns 62
5.2.6. Música New Age 63
5.2.7. Música Pop 64
5.2.8. Música Pop en Español 65
5.2.9. Música Ranchera 65
5.2.10. Música Rock 65
5.2.11. Tímpanos - Timpani 67
5.2.12. Triángulo - Triangle 67
5.2.13. Trompetas - Trumpets 68
5.2.14. Violines - Violins 68
5.2.15. Resultados Finales 69

6. Conclusiones 71
6.1. Conclusiones Generales 71
6.2. Trabajos Futuros 72

A. Código Fuente 75
A.1. main.c 75
A.2. Módulo dsp 83
A.2.1. dsp.h 83
A.2.2. dsp.c 83
A.3. Módulo encode 87
A.3.1. encode.h 87
A.3.2. encode.c 88
A.4. Módulo psychoacoustics 95
A.4.1. psychoacoustics.h 95
A.4.2. psychoacoustics.c 95
A.5. Módulo arithmeticcoder 98
A.5.1. arithmeticcoder.h 98
A.5.2. arithmeticcoder.c 99
A.6. types.h 103
A.7. Módulo wav 103
A.7.1. wav.h 103
A.7.2. wav.c 104

Referencias 107

Glosario 111

Lista de Figuras

2.1. Cuantizadores uniformes 10
2.2. Efecto de la cuantización utilizando Power-Law con $p = 0.5$ 12

3.1. Diagrama del proceso de un codificador perceptual básico 17
3.2. Señal de prueba $x[n] = \sin(2\pi 6n/768)$ 19
3.3. Bloques con $N = 512$ 20
3.4. MDCT de cada bloque 21
3.5. IMDCT de cada bloque 22
3.6. Reconstrucción con cancelación de aliasing en el dominio del tiempo 23
3.7. Traslape de ventanas tipo KBD con $N = 512$ y $\alpha = 4.0$ 25
3.8. Umbral auditivo modificado para un máximo de 75dB 29
3.9. Ejemplo de enmascaramiento en frecuencia 30
3.10. La cóclea y las localidades en que se perciben ciertas frecuencias 31
3.11. Función de propagación del MPEG Psychoacoustic Model 2 32
3.12. Efecto del enmascaramiento en tiempo (versión traducida de la mostrada en [Eberhard Zwicker99]) 33
3.13. Ejemplo del efecto de pre-eco 34
3.14. Transformada Discreta de Fourier y el umbral global de enmascaramiento que produce 37
3.15. MDCT y la máscara 38
3.16. MDCT y MDCT después del proceso de cuantización 39
3.17. Error de la señal y la máscara 40
3.18. Codificación aritmética con un modelo adaptivo de orden 0. 42

4.1. Diagrama de bloques para la codificación 47
4.2. Formato de archivo 51
4.3. Cabecera de archivo 51
4.4. Paquete 53
4.5. Diagrama del proceso de decodificación 55

5.1. Distribución de las edades de los sujetos 60
5.2. Castañuelas, tasa promedio: 244.48kbps. HSD de Tukey=0.347 61
5.3. Música Clásica, tasa promedio: 261.22kbps. HSD de Tukey=0.429 62
5.4. Música Disco, tasa promedio: 288.95kbps. HSD de Tukey=0.352 62

5.5. Chasquidos de Dedos, tasa promedio: 125.92kbps. HSD de Tukey=0.465 . . 63
5.6. Cornos Franceses, tasa promedio: 257.25kbps. HSD de Tukey=0.322 63
5.7. Música New Age, tasa promedio: 251.32kbps. HSD de Tukey=0.189 64
5.8. Música Pop, tasa promedio: 290.61kbps. HSD de Tukey=0.166 64
5.9. Música Pop en Español, tasa promedio: 238.08kbps. HSD de Tukey=0.153 . 65
5.10. Música Ranchera, tasa promedio: 281.91kbps. HSD de Tukey=0.148 66
5.11. Música Rock, tasa promedio: 312.12kbps. HSD de Tukey=0.143 66
5.12. Tímpanos, tasa promedio: 216.82kbps. HSD de Tukey=0.166 67
5.13. Triángulo, tasa promedio: 149.44kbps. HSD de Tukey=0.317 67
5.14. Trompetas. 68
5.15. Violines 69
5.16. Resultados Finales. HSD de Tukey=0.085 69

Lista de Tablas

5.1. Escala de 5 grados de la ITU-R para medir diferencias entre una señal de referencia y una procesada 59

Lista de Símbolos

$\Re(Z)$	Parte real de un número Z.
$\Im(Z)$	Parte imaginaria de un número Z.
N	Número de elementos en un bloque.
F_{max}	Frecuencia máxima representable.
F_S	Frecuencia de muestreo.
$x(t)$	Valor de una señal continua en el tiempo t.
$x[n]$	Valor de una señal discreta en n.
$X[k]$	Valor del k-ésimo coeficiente en el dominio de la frecuencia.
$\tilde{x}[n]$	Valor en n de la señal obtenida con la IMDCT antes de traslapar y sumar.
$< x >$	$\frac{1}{N}\sum_{n=0}^{N-1} x[n]$ para señales discretas.
	$\frac{1}{T}\int_{-T/2}^{T/2} x(t)dt$ para señales continuas.
Δ	Valor en que se disminuye la máscara con respecto a un enmascarador.
$\hat{x}$	Valor de x cuantizado.
q_{step}	Paso de cuantizador.
α	Valor en la ecuación de suma de mascaras.
	Valor que determina la forma de una ventana Kaiser-Bessel.
w	Ventana.
w_a	Ventana de análisis.
w_s	Ventana de síntesis.
$\mathrm{I}_0(x)$	Función Bessel modificada de primer tipo de orden cero.
$z(f)$	Bark correspondiente a la frecuencia f.
dz	Diferencia entre el bark de un componente enmascarado y su enmascarador.

Capítulo 1

Introducción

"La formulación de un problema es más importante que su solución".
Albert Einstein.

1.1. Planteamiento del Problema

Vivimos en un mundo en el que el acceso a la información es cada vez más una necesidad, el envío y recepción de datos se efectúa a lugares y desde lugares en cualquier parte del mundo. Las limitaciones tecnológicas y económicas han impulsado la búsqueda de esquemas para transferir datos de una forma más eficiente y de esquemas para minimizar la cantidad de bytes utilizados para representar la información.

La información multimedia, por su propia naturaleza, demanda grandes cantidades de almacenamiento y de procesamiento. Hasta cierto punto, mantener este contenido de manera local en una estación de trabajo no sería costoso, sin embargo, las transferencias de archivos de sonido o video a través de Internet (o de cualquier red en general) son cada día más comunes. Por lo tanto, es necesario perfeccionar la forma en la que este contenido se almacena para así minimizar el ancho de banda necesario para completar dicha transferencia.

Por ejemplo, un segundo de sonido con calidad Compact Disc (CD) sin comprimir ocuparía:

$$44100\text{muestras/segundo} \times 16\text{bits/muestra} \times 2 = 176,400\text{bytes/segundo}$$

lo que representaría 50.46MB para un archivo de sonido cuya duración sea de 5 minutos.

Como puede observarse, transferir esta cantidad de datos a través de una red es costoso en tiempo y, a menos que se tenga una conexión con una velocidad equivalente a 1.5Mbps, imposible de reproducir en tiempo real mientras es descargado.

Es necesario entonces, encontrar esquemas para reducir la cantidad de información necesaria para generar un contenido con la calidad que demanda la aplicación.

1.2. Antecedentes

1.2.1. Codificación y Compresión de Audio

La codificación es una representación de los datos y existen diferentes objetivos para los que es útil aplicarla. Un objetivo común es crear un flujo de información que sea resistente a errores, en cuyo caso es común que la tasa de datos sea incrementada. En algunos otros casos se requiere que la codificación de la información sea ininteligible para cualquier dispositivo diferente que el receptor, como es el caso de la criptografía.

Otro esquema de codificación, pretende reducir la cantidad de datos necesarios para reconstruir la información, en este caso, la codificación implica compresión. Los códigos Huffman [Huffman52] y codificación LZW [Welch84] son ampliamente usados para este fin.

En particular, cuando se trata de compresión de audio es posible efectuarla de dos maneras:

- Compresión sin pérdida: que busca y elimina redundancias en la información, la señal reconstruida es idéntica bit-a-bit a la original. En general, se utiliza un método de decorrelación inter-canal, un predictor o una aproximación, y finalmente codificación del residuo. Shorten [Robinson94], WavPack [WAV], Monkey's Audio [Ashland] y FLAC [Coalson] son algunos compresores sin pérdida que utilizan estos principios. Es común utilizar LPC (Linear Predictive Coding) como predictor y codificar el residuo utilizando códigos Rice [Robert Rice71],[Pen-Shu Yeh91],[Rice91] útiles cuando los datos tienen una distribución Laplaciana.

- Compresión con pérdida: en la actualidad este esquema se enfoca a explotar las características del oído humano para que las pérdidas de la señal sean imperceptibles.

Esquemas como el VQF de Yamaha, Musepack [Buschmann], Vorbis [VOR], AAC [MPEG01] y el famoso MP3 [MPEG93] utilizan compresión perceptual. ADPCM es un sistema de codificación con pérdida, pero no es perceptual, tan sólo utiliza un predictor y almacena la diferencia entre el valor predicho y el valor real.

Como en todo, cada esquema tiene ventajas y desventajas propias. La compresión sin pérdida logra mejores resultados si lo que se pretende es preservar la fidelidad del origen, a costa de un sacrificio en la relación de compresión. La compresión con pérdida no siempre logra el objetivo de ser perceptualmente transparente, lo cual disminuye drásticamente la calidad (al menos desde un punto de vista subjetivo) de la señal reconstruida, aunque su gran ventaja son las altas relaciones de compresión.

En compresión, la señal puede ser codificada utilizando esquemas de tasa de transferencia constante (*CBR*, del inglés *Constant Bit Rate*) o variable (*VBR*, del inglés *Variable Bit Rate*), siendo VBR una opción más natural, pues la entropía de una señal varía en el tiempo. En el caso de CBR, se necesita la misma cantidad de bytes para almacenar un periodo de tiempo determinado; por otro lado, en VBR, la tasa se ajusta de acuerdo a las demandas de la señal, pues diferentes partes requieren más información que otras, por ejemplo, un silencio prolongado requerirá muy pocos bits.

La decisión de utilizar CBR o VBR depende del objetivo de la codificación. Un esquema CBR es útil cuando se tiene un canal con un ancho de banda limitado y se desea que el proceso de transmitir en tiempo real ocupe exactamente una porción del ancho de banda. Los cambios abruptos en la tasa requerida para transmitir un archivo VBR pueden causar interrupciones en dispositivos que no estén diseñados para manejar esta situación. En CBR se mantiene una tasa de datos constante y la calidad del contenido es variable. La motivación para utilizar VBR radica no sólo en el hecho de que puede ser un codificador más eficiente sino en la posibilidad de mantener un contenido con calidad constante.

1.3. Objetivos de la Tesis

Objetivo general

Diseñar un codificador-decodificador (codec) perceptual y realizar su implementación en software.

Objetivos particulares

- Tomar como entrada un archivo de calidad equivalente a la del formato Compact Disc (CD) y comprimirlo con un perfil de alta calidad, esto es, poniendo más énfasis en la calidad subjetiva de la señal reconstruida a costa de una relación de compresión menor.

- Que el codificador y el decodificador puedan mantener una velocidad de compresión/descompresión suficiente como para efectuar su tarea en tiempo real.

- Que el formato de archivo permita iniciar la reproducción sin necesidad de contar con el archivo entero.

- Que este trabajo sirva como el principio de un proyecto de investigación e innovación en el área. Formatos como MP3 y AAC, a pesar de ser ampliamente utilizados contienen patentes y los documentos de especificación cuestan. En el caso de AAC incluso es necesario contar con una licencia de Dolby Labs. Formatos abiertos como Vorbis han tenido el soporte de la comunidad y continúan creciendo, sin embargo también se persigue desarrollar tecnología propia.

Para poder cumplir con estos objetivos, es necesario hacer un análisis de la señal de audio y eliminar información sin que esta pérdida sea perceptible.

La percepción humana del sonido se efectúa por frecuencia, por lo que es necesario llevar la señal de sonido en el dominio del tiempo a una representación en el dominio de la frecuencia. Una transformada que cumple con este requerimiento es la Transformada Discreta de Coseno Modificada (MDCT) [John P. Princen86], que es ampliamente utilizada.

La psicoacústica estudia los factores psíquicos y físicos de la percepción del sonido. Nos provee de modelos para determinar cuando un sonido es perceptible y cuando deja de serlo.

Una vez que se han determinado los componentes que son imperceptibles, los coeficientes de la MDCT son almacenados con una precisión menor. Esta pérdida de precisión producirá una deformación en la señal de sonido, pero se espera que esta deformación tenga una forma tal que no sea percibida.

En este trabajo, se combinan técnicas y principios de procesamiento digital de señales, psicoacústica y compresión para desarrollar un codec.

1.4. Descripción de Capítulos

El capítulo 2 explica la teoría bajo la que se fundamenta el audio digital, conceptos como cuantización y muestreo son analizados y sus implicaciones. También se introduce el concepto de razón señal a ruido.

El capítulo 3 trata el tema de la codificación perceptual, se introduce la Transformada Discreta de Coseno Modificada y un algoritmo eficiente para su cálculo, la psicoacústica y la codificación aritmética.

El capítulo 4 expone con detalle la implementación del codec que se desarrolló. El capítulo 5 muestra pruebas de sonido que miden la calidad subjetiva de los estándares MP3, AAC y el codec implementado.

Finalmente en el capítulo 6 se exponen las conclusiones y se enumeran los aspectos sobre los que se puede mejorar el codec desarrollado en este trabajo.

El código fuente del codificador-decodificador implementado se lista en el apéndice A.

Capítulo 2

Audio Digital

"La música es para el alma lo que la gimnasia para el cuerpo".
Platón.

El audio es por su propia naturaleza una señal continua, esto representa un problema puesto que la computadora sólo trabaja en intervalos de tiempo determinados por la frecuencia de reloj y con una precisión finita, esto es, en tiempo y valores discretos. En este capítulo se aborda el tema del audio digital y de su representación en un sistema digital sin tomar en cuenta factores más complejos como la percepción humana.

Para resolver el problema del tiempo discreto, se utiliza el muestreo, que obtiene el valor de la amplitud de la onda cada periodo de tiempo determinado. El segundo problema, utilizar valores discretos, se maneja utilizando cuantización, que en términos prácticos es la discretización de un valor de amplitud cuya precisión es infinita. Tanto el muestreo como la cuantización presentan desventajas para representar una señal continua, sin embargo, una combinación adecuada de éstos puede dar resultados satisfactorios.

A continuación abordamos cada uno de estos temas.

2.1. Muestreo

El teorema del muestreo [Shannon48] indica que una función continua en tiempo $f(t)$ que esté limitada en una banda de frecuencia de 0 a F_{max} ciclos por segundo puede ser representada en su totalidad por una señal discreta $x[n]$ cuyos valores estén espaciados

$\frac{1}{2F_{max}}$ en la manera indicada a continuación:

$$f(t) = \sum_{n=-\infty}^{\infty} x[n] \frac{\sin(\pi(2F_{max}t - n))}{\pi(2F_{max}t - n)} \tag{2.1}$$

donde

$$x[n] = f\left(\frac{n}{2F_{max}}\right) \tag{2.2}$$

De aquí, se deduce que la frecuencia de muestreo F_S, cumpla con la siguiente restricción:

$$F_S \geq 2F_{max} \tag{2.3}$$

El mínimo que cumple con esta condición ($2F_{max}$) es llamado Frecuencia de Nyquist de la señal.

Existen casos en los que la señal continua contiene componentes de frecuencia por arriba de $F_S/2$. Para resolver este problema, es necesario aplicar un filtro pasabajas antes de muestrear, de otra manera, las frecuencias superiores a $F_S/2$ serán mezcladas con el contenido por debajo de $F_S/2$. Este fenómeno se conoce como *aliasing* y produce una distorsión en la señal original [Glenn Zelnicker94]. Típicamente, este filtro pasabajas tiene una frecuencia de corte un poco inferior a la frecuencia máxima representable por la señal discreta para evitar un corte abrupto en el espectro y tener una mejor representación de los datos, pues no existen los pulsos de duración cero ni el filtro pasabajas ideal.

Para el muestreo de audio una frecuencia de muestreo cercana al doble de 20kHz es deseable, pues el oído humano difícilmente reconoce frecuencias superiores [Watkinson04].

El CD-Audio, por ejemplo, utiliza una frecuencia de muestreo de 44.1kHz, un número bastante raro, pero con una razón más allá que un simple capricho de diseño [Watkinson04]. En los principios del audio digital, las grabadoras de video fueron adaptadas para almacenar muestras de audio mediante la creación de una onda de video que contenía niveles de blanco y negro. La frecuencia de muestreo entonces tenía que estar ligada al sistema de televisión en uso. Era necesario encontrar una frecuencia única para trabajar con los dos sistemas de televisión predominantes, 525 líneas a 60Hz (NTSC) y 625 líneas a 50Hz (PAL).

En el video a 60Hz solamente hay 490 líneas entrelazadas por fotograma, sólo se utilizan 245 en un instante, ahora si se almacenan 3 muestras por línea, la frecuencia de

muestreo es: $60 \times 245 \times 3 = 44100$Hz. En el video a 50Hz, un fotograma contiene 588 líneas entrelazadas, o 294 líneas en un instante, y la misma frecuencia de muestreo aplica: $50 \times 294 \times 3 = 44100$Hz. Puesto que los masters eran grabados utilizando este sistema, la frecuencia de muestreo del CD-Audio se convirtió en 44.1kHz. [Watkinson04]

El DVD-Audio por ejemplo, maneja frecuencias de muestreo tan altas como 96kHz (para audio 5.1) o 192kHz para audio estéreo.

2.2. Cuantización

El teorema del muestreo asume que la señal muestreada $x[n]$ contiene los valores exactos de amplitud de la señal continua $x(t)$ y provee de un método matemático para la reconstrucción de la señal. Sin embargo, asume que la precisión de las muestras es infinita, algo imposible en una computadora, es entonces necesario tener una representación de estos valores con una precisión finita. La conversión de escalares con precisión infinita a escalares con precisión finita se conoce como cuantización.

Es evidente que cada valor discreto en el rango del cuantizador sólo puede representar un valor. La diferencia entre dos valores consecutivos en este rango se llama paso de cuantizador, y este puede ser constante o variable, en cuyo caso el cuantizador se denominará uniforme o no uniforme, respectivamente.

2.2.1. Cuantizadores Uniformes

Un cuantizador uniforme es aquel que tiene un paso de cuantizador constante. Asumamos que el paso del cuantizador es 0.5, se cuenta con un cuantizador de 3 bits y que el valor $(000)_2$ representa el entero 0. Puesto que el paso del cuantizador es de 0.5, un valor de $(001)_2$ representa el valor 0.5, $(010)_2$ representa 1.0 y así sucesivamente.

Aquí se presenta una interrogante ¿qué sucede con los valores que no sean un múltiplo exacto del paso del cuantizador? La respuesta a esto es sencilla, se redondea al valor más cercano. Por ejemplo, 0.8 se cuantizaría como $(010)_2$ (que representa 1.0), mientras que un valor como 0.6 se cuantizaría como $(001)_2$ (0.5). En la ecuación 2.4 se da una forma

general para este comportamiento.

$$q(t) = \text{nint}\left(\frac{x(t)}{q_{step}}\right) \tag{2.4}$$

donde $\text{nint}(a)$ es una función que regresa el entero más cercano, q es el valor cuantizado y q_{step} es el paso de cuantizador.

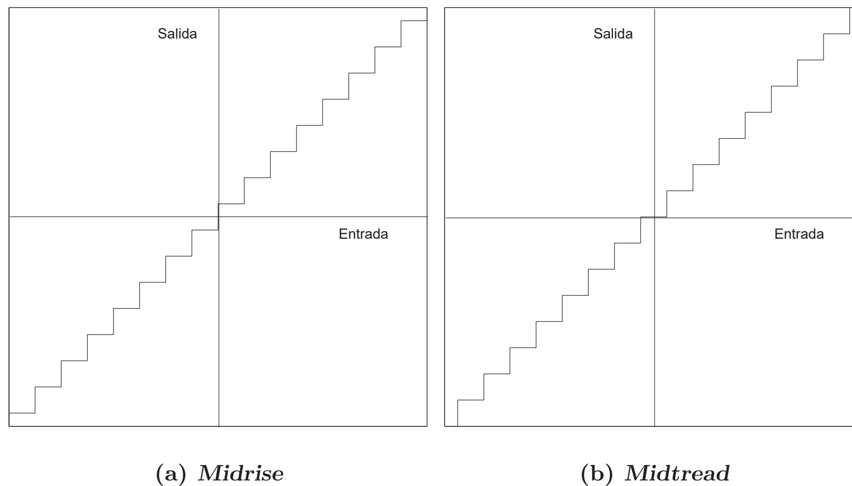
Para decuantizar, es necesario pasar el valor cuantizado $q(t)$ por una función como la ilustrada en la ecuación 2.5.

$$\hat{x}(t) = q(t) \times q_{step} \tag{2.5}$$

donde $\hat{x}(t)$ es meramente una aproximación de $x(t)$, de aquí se puede deducir que un paso de cuantizador más fino generará mejores resultados, pero serán necesarios más bits para representar el valor cuantizado.

Cuantizador Midrise

En el cuantizador midrise no existe una representación del 0.0, la mitad de los valores posibles se asignan para representar números positivos y la otra mitad para representar números negativos. En la figura 2.1(a) se puede apreciar el efecto de un cuantizador midrise. Note como el valor de entrada 0.0 es asignado a un valor positivo o negativo.



(a) ***Midrise*** (b) ***Midtread***

Figura 2.1: Cuantizadores uniformes

Cuantizador Midtread

En un cuantizador midtread, existe una representación exacta del valor 0.0, esto es conveniente cuando se está codificando sonido, pues partes con silencio serán mejor representadas con un valor exacto de 0.0 que con una aproximación. En la figura 2.1(b) note como valores pequeños se asignan a un valor cuantizado equivalente a 0.0.

2.2.2. Cuantizadores No Uniformes

La cuantización no uniforme se basa en la premisa de que no siempre es conveniente que el paso del cuantizador sea constante. Un pequeño error en un valor muy grande puede ser despreciable, no así en el caso de valores muy pequeños. Para obtener este comportamiento, el valor a cuantizar x es primeramente procesado por una función monotónica creciente, y este valor resultante y es cuantizado con un cuantizador uniforme.

$$y = c(x)$$

Para decuantizar, primeramente se aplica el decuantizador uniforme para obtener un valor $\hat{y}$, y el resultado se procesa con la función inversa.

$$\hat{x} = c^{-1}(\hat{y})$$

Cuantizador Power-Law

En este tipo de cuantizador se utiliza una función de la forma:

$$c_{power}(x) = |x|^p \tag{2.6}$$

cuyo parámetro p es un valor entre 0 y 1.0 para asegurar que la pendiente sea alta para valores pequeños de x. En la figura 2.2 se muestra el efecto para $p = 0.5$, note como los valores pequeños obtienen mayor precisión que los valores grandes. MPEG-1 Layer 3 y MPEG-2 AAC utilizan un cuantizador no uniforme del tipo Power-Law que utiliza un valor de $p = 3/4$. Es necesario utilizar valor absoluto y almacenar aparte el signo, pues elevar un número negativo a una potencia entre 0.0 y 1.0 no garantizará que el resultado sea un real.

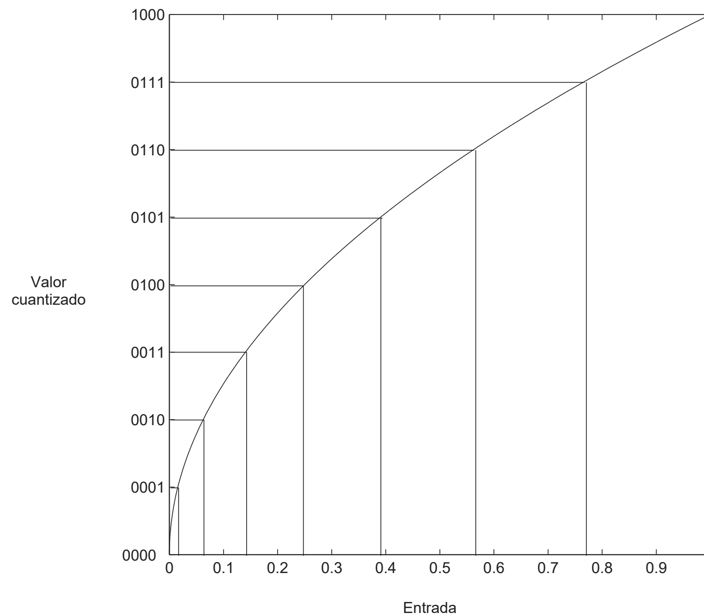


Figura 2.2: Efecto de la cuantización utilizando Power-Law con $p = 0.5$

Cuantizador Logarítmico

Como su nombre lo indica, la función es de tipo logarítimica. Una forma muy popular es la μ law, cuya función es:

$$c_\mu(x) = \frac{\log_b(1 + \mu|x|)}{\log_b(1 + \mu)} \tag{2.7}$$

el parámetro μ se escoge de acuerdo a las necesidades del cuantizador. μ-law es utilizado en los estándares de telefonía de Norte América y Japón [U.S96].

Cuantizador de Punto Flotante

Es un cuantizador no uniforme, que se divide en dos campos principales: exponente y mantisa. La mantisa indica un valor numérico, mientras que el exponente indica en donde se sitúa este valor. De esta manera los valores pequeños tendrán mayor precisión que los valores grandes, pues los bits destinados a la mantisa siempre son constantes. Para más información, el estándar IEEE-754 [IEEE85] explica en detalle una representación de

números y aritmética con punto flotante.

2.3. Codificadores de Audio

Una combinación entre la forma de muestrear, la forma de cuantizar y la forma de almacenar estos datos se conoce como un codificador de audio. Métodos sofisticados como el introducido en el Super Audio CD (SACD) graban 1 bit de datos a una velocidad de muestreo de cercana a 2822.4kHz obteniendo un rango dinámico de aproximadamente 120dB [Konstantinides03].

Un método muy sencillo de codificación de audio es el de Modulación de Códigos de Pulso (PCM por sus siglas en inglés). En este trabajo se utiliza como entrada un archivo PCM en formato WAV de 2 canales (estéreo), una velocidad de muestreo de 44100Hz y un cuantizador uniforme midtread de 16bits.

2.4. Calidad objetiva de la representación Digital del Audio

Como se expuso con anterioridad, el audio es una señal continua y el audio digital es una representación discreta de esta señal. Esta representación, por buena que sea, jamás será idéntica a la original. Una métrica para medir el grado de similitud entre dos señales es la razón señal a ruido (en lo sucesivo *SNR, Signal to Noise Ratio*, por sus siglas en inglés), y se calcula de la siguiente manera [Marina Bosi02]:

$$\text{SNR} = 10 \log_{10} \frac{< x_{in}^2 >}{< q^2 >} \text{dB} \tag{2.8}$$

donde

$$q(t) = x_{out}(t) - x_{in}(t) \tag{2.9}$$

si el error (ruido) q es pequeño el SNR será un valor grande.

El CD-Audio tiene un SNR de alrededor de 96dB, mientras que el DVD-Audio es capaz de lograr un SNR máximo de 144dB [Konstantinides03]. Para determinar el SNR en audio digital, se utiliza la siguiente ecuación [Marina Bosi02]:

$$\text{SNR} = 10 \log_{10} \left(\frac{< x_{in}^2 >}{x_{max}^2} \right) + 6.021R + 4.771 \ \text{dB} \tag{2.10}$$

donde R es el tamaño del cuantizador en bits y x_{max} indica el valor máximo representable. En la ecuación 2.10 es fácil observar que por cada bit que se agregue al cuantizador, el SNR aumentará alrededor de 6dB.

El SNR no es una medida perceptual de calidad, pues ignora la sensibilidad del oído en diferentes frecuencias y los efectos de enmascaramiento, sin embargo, es una forma sencilla de medir la calidad de una señal y es ampliamente adoptado. En compresión perceptual de audio, el SNR no es una medida útil pues la señal reconstruida, a pesar de ser equivalente para el oído humano no será equivalente en términos numéricos. Entonces ¿cómo medir la calidad de un codec perceptual?, en la actualidad, la mejor herramienta para juzgar la calidad resultante es precisamente el oído humano.

2.5. Conclusiones

El audio digital es una representación de una señal de audio continua, la calidad de ésta no siempre es la deseable pero con una velocidad de muestreo (al menos del doble de lo perceptible por el oído humano) y una cuantización adecuada es posible tener una representación fiel, aunque no exacta, del original. Una medida común para determinar la calidad de sonido es la razón de señal a ruido. La razón señal a ruido es útil para medir la calidad desde el punto de vista de fidelidad pero no es un indicador de la calidad perceptual.

Capítulo 3

Codificación Perceptual

"Un error no se convierte en verdad por el hecho de que todo el mundo crea en él".
Mahatma Gandhi.

Un codificador-decodificador perceptual explota las características del oído humano para comprimir la señal. La finalidad es sencilla: obtener una tasa de transferencia mínima con la mejor calidad posible. Para poder cumplir con ambos requerimientos, es indispensable contar con modelos que asemejen el comportamiento del oído humano y así poder eliminar información sin que esta pérdida sea perceptible.

Es deseable que al reproducir un archivo codificado de audio se tenga un pequeña porción de datos decodificados listos para ser enviados a un dispositivo de reproducción. Pero, ¿cuántos datos en realidad son necesarios?, y aún más ¿qué costo tiene almacenar estos datos?.

La primera pregunta implica conocer el tipo de canal que se va a utilizar, y la velocidad con que los datos van a ser reproducidos. En general, es deseable que en cualquier instante se tenga el número de datos que un dispositivo de reproducción requiere. Por ejemplo, para reproducir un archivo estéreo muestreado a $44,100$Hz, es necesario mantener al menos $88,200$ muestras en una memoria temporal.

Ahora analicemos la segunda interrogante. Asumiendo que siempre se tiene una cantidad de datos suficiente para mantener una reproducción sin cortes, todavía es necesario limitar la cantidad de datos decodificados que se almacenan temporalmente antes de la

reproducción. Es obvio que mantener $5,000,000$ muestras en una memoria temporal cuando en realidad se requieren $88,200$ en un instante es un gasto inútil, al menos bajo condiciones óptimas.

Es necesario entonces, ofrecer flexibilidad para que un programa reproductor tome esta decisión. En codificación perceptual es conveniente mantener los datos en una representación en el dominio de la frecuencia, pues es posible modelar con mejor precisión el oído. El audio digital, como se expuso en el capítulo 2 está representado en tiempo, por lo que se debe utilizar algún tipo de transformación. Los codecs que llevan los datos a una representación diferente y los codifican se conocen como codecs de transformación.

Una opción posible es cuantizar los coeficientes de una transformada Discreta de Fourier (DFT por sus siglas en inglés). Sin embargo, para tener una buena resolución en tiempo y poder ofrecer flexibilidad en el proceso de decodificación, es necesario dividir la señal en bloques. Además, para hacer un análisis más preciso en el dominio de la frecuencia, es necesario utilizar una ventana para que los bordes de cada bloque tengan una transición suave hacia cero. Esta transición previene un corte abrupto que introduce aliasing en el dominio de la frecuencia.

Al momento de calcular la IDFT, la señal obtenida correspondería al bloque multiplicado por la ventana utilizada en la fase de análisis. Para obtener la señal original uno podría pensar en dividir por la ventana, puesto que ésta es conocida en todo momento. Al hacer esto, los errores generados por la cuantización se amplificarían en gran medida cerca de los bordes (donde la ventana tiene un valor cercano a 0), generando artefactos audibles.

Una técnica para resolver este problema es hacer que bloques adyacentes tengan un traslape. Al momento de síntesis se efectúa traslape y suma de éstos. Se pretende que la señal original pueda ser reconstruida exactamente. Esto elimina la necesidad de dividir por valores cercanos a 0, haciendo que el ruido de cuantización no sea amplificado. Sin embargo, esto crea restricciones en el tipo de ventanas que se pueden utilizar (más detalles sobre esto en la próxima sección).

La IDFT requiere N coeficientes para reconstruir N muestras. Asumiendo un traslape de 50 % entre bloques adyacentes, si se pretende reconstruir un bloque con traslape y suma, serán necesarios $2N$ coeficientes de la DFT. Lo que nos dice que estamos doblando

efectivamente la cantidad de datos necesarios para reconstruir nuestra señal. Utilizar un traslape menor es posible a costa de aumentar los artefactos audibles.

Una transformada que resuelve todos estos problemas es la transformada discreta de coseno modificada (en lo sucesivo MDCT por sus siglas en inglés) propuesta en [John P. Princen86] y más desarrollada en [John P. Princen87]. Utilizada en codecs del estado del arte como AAC [MPEG01].

La MDCT ha sido ampliamente adoptada pues matemáticamente ofrece reconstrucción exacta, lleva los datos a una representación en el dominio de la frecuencia y está muestreada críticamente. El muestreo crítico mantiene fija la cantidad de datos para reconstruir una señal, en otras palabras, para reconstruir una señal de tamaño N son necesarios N coeficientes de la MDCT. Además, la MDCT utiliza la técnica de traslape (en un 50 %) y suma.

La figura 3.1 muestra un diagrama básico del funcionamiento de un codificador perceptual de transformación que usa MDCT.

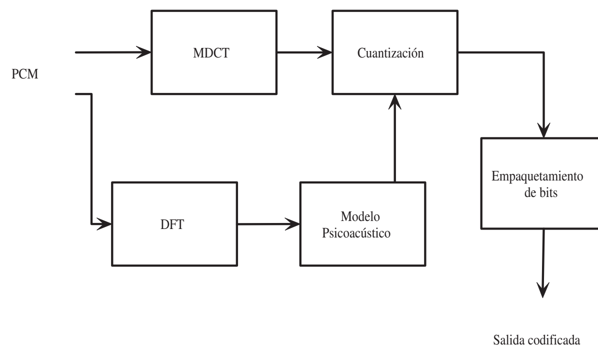


Figura 3.1: Diagrama del proceso de un codificador perceptual básico

Para hacer un análisis psicoacústico de mayor precisión, es común que se utilice una DFT aplicada a un bloque que ha sido multiplicado con una ventana de Hanning. Aunque existen codecs que efectúan todo el proceso exclusivamente con la MDCT, como es el caso de AC-3 [ATS95].

La información obtenida del modelo psicoacústico es utilizada para escoger la precisión con la que serán cuantizados los coeficientes de la MDCT. Una precisión grande ocasionará que la señal reconstruida tenga un error muy pequeño, sin embargo, al utilizar muchos bits la relación de compresión se verá afectada. Utilizar poca precisión hará que la señal reconstruida tenga un error muy grande, pero se tendrá una muy buena compresión.

Al codificar en el dominio de la frecuencia, se sabe que frecuencias tendrán más distorsión. Es aquí donde el modelo psicoacústico juega un papel decisivo, pues éste nos indicará en que porciones del espectro el oído no será capaz de percibir el ruido introducido por la cuantización y, aún más, nos dará información sobre la cantidad máxima de ruido que se puede localizar en dicha frecuencia sin que sea audible.

Una vez que los coeficientes se han cuantizado se procede a empaquetar los bits en algún formato especificado, este formato deberá contener toda la información necesaria para reconstruir cada bloque.

A continuación, analicemos con detalle las fases que se requieren para un codificador de transformación perceptual.

3.1. Transformada Discreta de Coseno Modificada

Esta transformada, es del tipo de Cancelación de Aliasing en el Dominio del Tiempo (TDAC por sus siglas en inglés) que utiliza bloques con traslape para poder reconstruir la señal, ya que su tranformada inversa no es del estilo de una DFT. En el caso de la DFT, su inversa reconstruye exactamente la señal; en cambio la MDCT requiere de una técnica de traslape y suma para tener una reconstrucción exacta.

3.1.1. Análisis y Síntesis utilizando la MDCT

Antes de mostrar ecuaciones, expongamos como es el proceso de análisis y síntesis con la MDCT. Una señal cualquiera, x, podrá ser reconstruida exactamente utilizando MDCT y su inversa si se utiliza una ventana válida y las transformadas inversas se suman en la parte en que se traslapan los bloques.

Tomemos por ejemplo una señal sinusoidal de 768 puntos como la mostrada en la

figura 3.2.

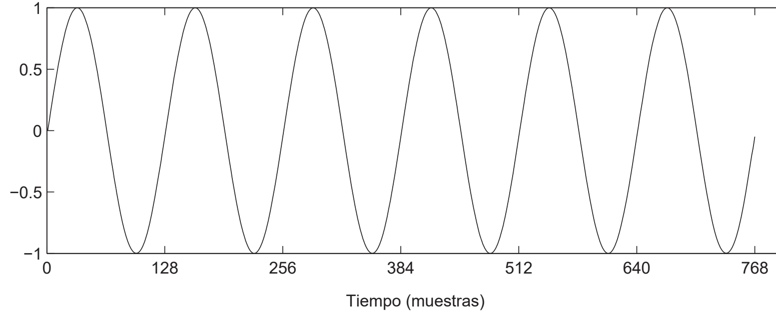


Figura 3.2: Señal de prueba $x[n] = \sin(2\pi 6n/768)$

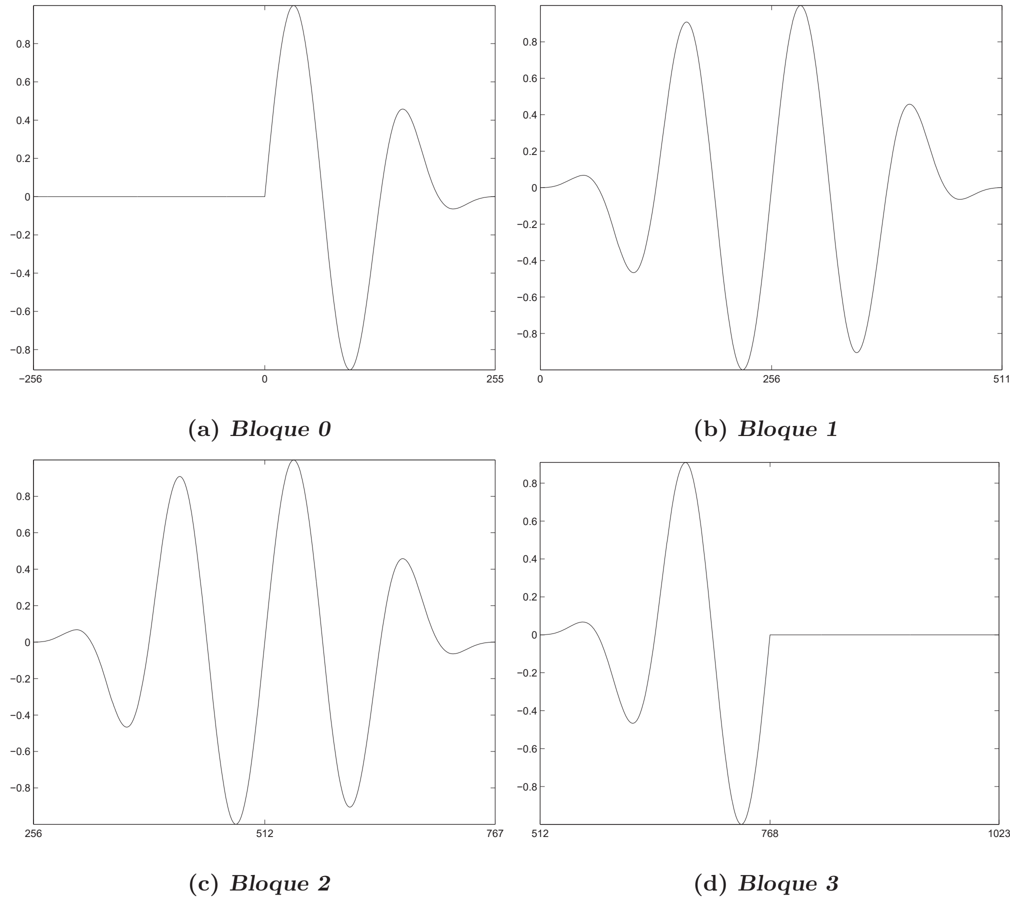
En la fase de análisis se obtienen bloques de $N = 512$ con un traslape del 50 % y se multiplican por una ventana de análisis válida, los 4 bloques obtenidos se muestran en la figura 3.3. Posteriormente se procede a calcular la MDCT, en la figura 3.4 se pueden observar las MDCT de cada bloque.

En la fase de síntesis se aplica la inversa a cada una de estas transformadas (figura 3.5). Note como ningún bloque corresponde a una porción exacta de la señal de prueba, ni al bloque multiplicado por la ventana.

Sin embargo, si sumamos cada uno de los bloques de la figura 3.5, podemos ver como la señal original es progresivamente reconstruida, este comportamiento se ilustra en la figura 3.6. En la figura 3.6(a) sólo se ha sumado la inversa del bloque 0, en la figura 3.6(b) se han sumado con traslape las inversas de los bloques 0 y 1, en la figura 3.6(c) los bloques 0, 1 y 2; finalmente en la figura 3.6(d) los cuatro bloques han sido apropiadamente traslapados y sumados, y la señal obtenida es idéntica a nuestra señal de prueba.

3.1.2. Cálculo de la MDCT e IMDCT

En la ecuación 3.1 se muestra como calcular la MDCT del i-ésimo bloque de datos x de tamaño N (versión mostrada en [Sporer92] y [Chi-Min Liu99], aunque en [Marina Bosi02]



(a) ***Bloque 0*** (b) ***Bloque 1***

(c) ***Bloque 2*** (d) ***Bloque 3***

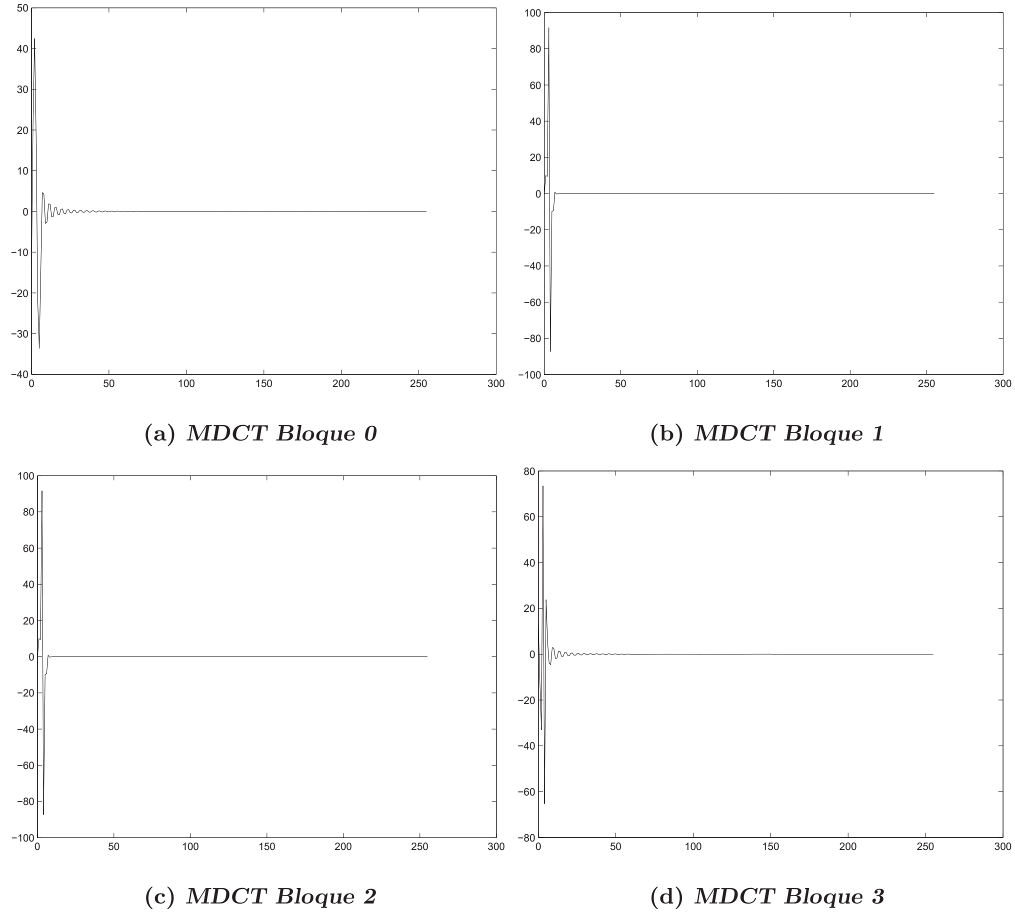
Figura 3.3: Bloques con $N = 512$

y [John P. Princen87] se pueden encontrar formas equivalentes).

$$X_i[k] = \sum_{n=0}^{N-1} w_a^i[n] x_i[n] \cos\left[\frac{\pi}{2N}(2n + 1 + \frac{N}{2})(2k + 1)\right] \qquad (3.1)$$

para $k = 0, \ldots, \frac{N}{2} - 1$. w_a^i es la ventana de análisis para el i-ésimo bloque. Como es de notar, el número de coeficientes de la MDCT es la mitad del tamaño del bloque, a simple vista parecería que se obtiene una reducción efectiva del 50 %, aunque esto sería deseable, no es el caso, pues la señal reconstruida a partir de dichos coeficientes no corresponde a la señal original.

La ecuación para calcular la transformada inversa (IMDCT) (versión modificada



(a) ***MDCT Bloque 0***

(b) ***MDCT Bloque 1***

(c) ***MDCT Bloque 2***

(d) ***MDCT Bloque 3***

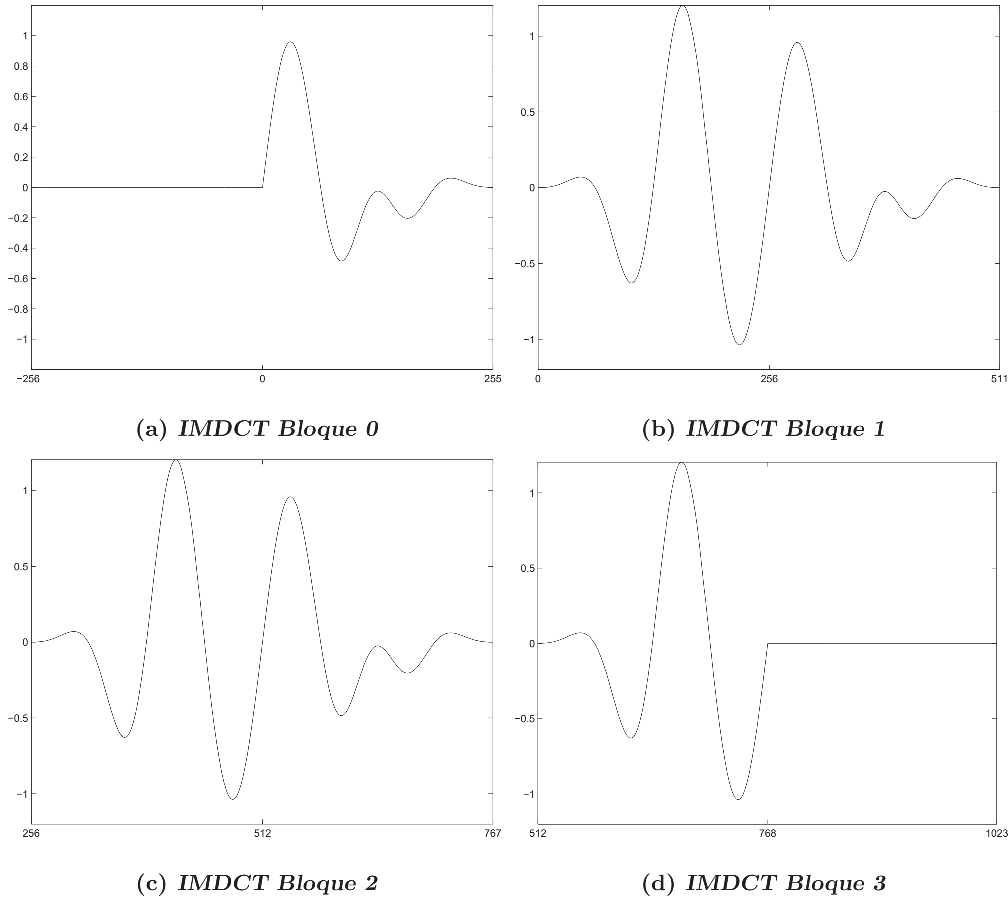
Figura 3.4: MDCT de cada bloque

de la mostrada en [Sporer92]) es:

$$\tilde{x}_i[n] = w_s^i[n]\frac{4}{N}\sum_{k=0}^{\frac{N}{2}-1} X_i[k]\cos\left[\frac{\pi}{2N}(2n+1+\frac{N}{2})(2k+1)\right] \tag{3.2}$$

para $n = 0, \ldots, N-1$. Donde w_s^i es la ventana de síntesis para el i-ésimo bloque.

Para poder reconstruir la señal original es necesario que los bloques a los que se les aplique la MDCT estén traslapados en un 50 %, y posteriormente, sumar las señales generadas por las IMDCT (en las partes que éstas se traslapan). Esta técnica es conocida como cancelación de aliasing en el dominio del tiempo.



(a) ***IMDCT Bloque 0*** (b) ***IMDCT Bloque 1***

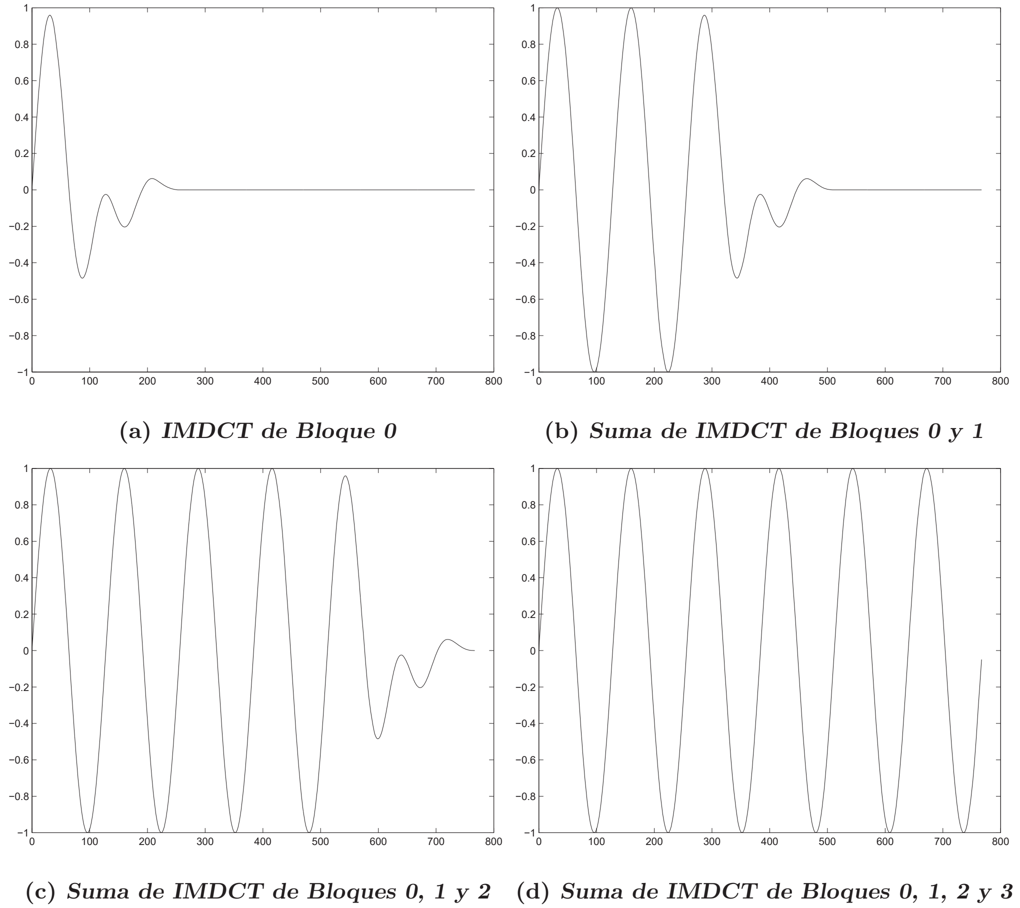
(c) ***IMDCT Bloque 2*** (d) ***IMDCT Bloque 3***

Figura 3.5: IMDCT de cada bloque

3.1.3. Ventanas válidas para la MDCT

En las ecuaciones 3.1 y 3.2 se observan dos términos, w_a^i y w_s^i, que corresponden a la ventana de análisis y de síntesis, respectivamente. Para cumplir con los requerimientos de reconstrucción exacta, es necesario que dichas ventanas cumplan con las siguiente condiciones [Marina Bosi02]:

1. Las ventanas de análisis y de síntesis deben ser inversas en tiempo entre sí en la porción en que se traslapan.



(a) ***IMDCT de Bloque 0*** **(b)** ***Suma de IMDCT de Bloques 0 y 1***

(c) ***Suma de IMDCT de Bloques 0, 1 y 2*** **(d)** ***Suma de IMDCT de Bloques 0, 1, 2 y 3***

Figura 3.6: Reconstrucción con cancelación de aliasing en el dominio del tiempo

2. Para $n = 0, \ldots, \frac{N}{2} - 1$, e i indica el número de bloque:

$$w_a^i[n]w_s^i[n] + w_a^{i-1}[N/2+n]w_s^{i-1}[N/2+n] = 1 \tag{3.3}$$

Por simplicidad, en este trabajo se consideró que la ventana de análisis y de síntesis son del mismo tipo y del mismo tamaño.

Ventana Derivada de Kaiser-Bessel

La ventana Derivada de Kaiser-Bessel (KBD por sus siglas en inglés) se obtiene a partir de una ventana Kaiser-Bessel que se calcula de la siguiente forma:

$$w_{KB}[n] = \frac{\mathrm{I}_0\left(\pi\alpha\sqrt{1.0 - \left(\frac{n-N/2}{N/2}\right)^2}\right)}{\mathrm{I}_0\left(\pi\alpha\right)} \tag{3.4}$$

donde α es un parámetro que define la forma de la ventana, un valor de $\alpha = 0$ corresponde a una ventana rectangular mientras que para valores grandes de α la forma de la ventana Kaiser tiende a una gaussiana. I_0 es la función Bessel modificada de primer tipo de orden cero:

$$\mathrm{I}_0(x) = \sum_{k=0}^{\infty} \frac{(\frac{1}{4}x^2)^k}{(k!)^2} \tag{3.5}$$

La ventana Kaiser-Bessel no satisface las restricciones de traslape y suma de la MDCT, pero cualquier ventana w puede ser normalizada a una ventana w' para que cumpla con estos requerimientos si se procesa de la siguiente manera (versión modificada de [Marina Bosi02]):

$$w'[n] = \begin{cases} \sqrt{\frac{\sum_{p=0}^{n} w[p]}{\sum_{p=0}^{N/2} w[p]}} & 0 \leq n < N/2 \\ \sqrt{\frac{\sum_{p=n-N/2+1}^{N/2} w[p]}{\sum_{p=0}^{N/2} w[p]}} & N/2 \leq n < N \end{cases} \tag{3.6}$$

Si w es una ventana Kaiser-Bessel entonces w' será una ventana que cumple con las restricciones de la MDCT y se conoce como ventana Derivada de Kaiser-Bessel. En la figura 3.7 se puede apreciar la forma de la ventana KBD con $\alpha = 4.0$, y como se traslapan un 50 % en el tiempo.

3.1.4. Implementación rápida de la MDCT e IMDCT

Una implementación directa de las ecuaciones 3.1 y 3.2 tendría un tiempo de corrida $O(N^2)$. En [Gluth91] se presenta una forma para calcular la MDCT y su inversa utilizando una FFT de $N/4$ puntos, $\frac{N}{4}\log_2(\frac{N}{4})$. Los algoritmos son sencillos y se ilustran a continuación.

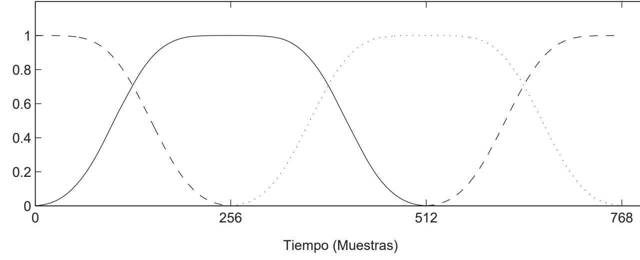


Figura 3.7: Traslape de ventanas tipo KBD con $N = 512$ y $\alpha = 4.0$

Algoritmo para la MDCT

Dado un bloque de datos x de tamaño N, se crea una nueva señal rot de la siguiente forma. Se asume que x contiene un bloque de datos que ya ha sido multiplicado por una ventana válida.

$$rot[r] = -x[r + \frac{3N}{4}], \quad r = 0, \ldots, \frac{N}{4} - 1 \tag{3.7}$$

$$rot[r] = x[r - \frac{N}{4}], \quad r = \frac{N}{4}, \ldots, N - 1 \tag{3.8}$$

Posteriormente se crea una señal compleja y:

$$y[r] = \frac{1}{2}\left[(rot[2r] - rot[N-1-2r]) - j\left(rot[\frac{N}{2} + 2r] - rot[\frac{N}{2} - 1 - 2r]\right)\right] \tag{3.9}$$

para $r = 0, \ldots, \frac{N}{4} - 1$.

Se calcula:

$$Y[k] = 2e^{-j\frac{2\pi}{N}(k+\frac{1}{8})} \sum_{r=0}^{\frac{N}{4}-1} \left(y[r]e^{-j\frac{2\pi}{N}(r+\frac{1}{8})}\right) e^{-j\frac{2\pi}{N/4}rk} \tag{3.10}$$

para $k = 0, \ldots, \frac{N}{4} - 1$. Note como la sumatoria es una transformada de Fourier de una señal compleja de $\frac{N}{4}$ puntos (calculable usando la FFT). Finalmente, se obtiene la MDCT de x haciendo:

$$X[2k] = \Re(Y[k]) \tag{3.11}$$

$$X[\frac{N}{2} - 1 - 2k] = -\Im(Y[k]) \tag{3.12}$$

para $k = 0, \ldots, \frac{N}{4} - 1$. Donde X es la MDCT de x. La única restricción del algoritmo es que $\frac{N}{4}$ sea un número entero.

Algoritmo para la IMDCT

Dados los K coeficientes de X. Se hace:

$$Y[k] = \frac{1}{2} \left(X[2k] + jX[K-1-2k] \right) \tag{3.13}$$

para $k = 0, \ldots, \frac{K}{2} - 1$.

Posteriormente se calcula:

$$y[n] = e^{-j\frac{\pi}{K}(n+\frac{1}{8})} \frac{8}{N} \sum_{k=0}^{\frac{K}{2}-1} \left(Y[k] e^{-j\frac{\pi}{K}(k+\frac{1}{8})} \right) e^{-j\frac{2\pi}{K/2}kn} \tag{3.14}$$

para $n = 0, \ldots, \frac{K}{2}$ y $N = 2K$. La sumatoria es nuevamente una transformada de Fourier de una señal compleja de $K/2$ puntos $(N/4)$, calculable mediante una FFT.

Se crea una nueva señal con los elementos de y de la siguiente forma:

$$rot[2k] = \Re(y[k]) \tag{3.15}$$

$$rot[K+2k] = \Im(y[k]) \tag{3.16}$$

para $k = 0, \ldots, \frac{K}{2} - 1$.

Se reorganizan los elementos de la siguiente forma:

$$rot[k] = -rot[N-1-k] \tag{3.17}$$

para $k = 1, 3, \ldots, N-3, N-1$.

Finalmente para obtener $\tilde{x}$ se hace:

$$\tilde{x}[n] = rot[n + K/2], \qquad n = 0, \ldots, \frac{3K}{2} - 1 \tag{3.18}$$

$$\tilde{x}[n] = -rot[n - 3K/2], \quad n = \frac{3K}{2}, \ldots, N \tag{3.19}$$

Cabe hacer hincapié en que es necesario multiplicar $\tilde{x}$ por una ventana de síntesis válida para completar el proceso apropiadamente.

3.1.5. La MDCT y la codificación perceptual

La MDCT garantiza matemáticamente que la reconstrucción de la señal será exacta en ausencia de cuantización. Para poder comprimir será necesario utilizar el menor número posible de bits para almacenar los coeficientes de la MDCT, con lo que se introducirá un ruido de cuantización que hará que la señal reconstruida no sea igual a la original. Sin embargo, es posible utilizar cuantizadores con pocos bits si el ruido de cuantización producido es imperceptible. Para poder asignar este ruido de cuantización, es necesario conocer la forma en la que el oído humano percibe el sonido, y esta área del conocimiento es tratada en la psicoacústica.

3.2. Psicoacústica

El oído humano tiene muchas características que pueden ser explotadas en la codificación de audio, muchas de éstas son limitantes en el propio oído y otras están localizadas en la forma de procesar el sonido en el cerebro. En particular se tiene:

1. Un rango perceptible de frecuencias de 20Hz a 20000Hz.
2. El umbral auditivo es más bajo en frecuencias que se aproximan a las de la voz y más alto en frecuencias altas y bajas (subsección 3.2.2).
3. Enmascaramiento simultaneo (subsección 3.2.3).
4. Enmascaramiento temporal (subsección 3.2.4).

3.2.1. La escala decibel

Para que un sonido sea perceptible por el oído humano la onda sonora debe tener una intensidad mayor o igual 1×10^{-12}watts/m^2, esta intensidad llamada umbral auditivo es extremadamente pequeña si se compara con el umbral del dolor cuyo valor es de 10watts/m^2.

Para manejar este rango tan amplio, se utiliza una escala logarítmica llamada escala decibel, cuya ecuación se muestra a continuación:

$$L_q = 10 \log_{10} \frac{q}{q_{ref}} \text{dB} \tag{3.20}$$

donde q es la cantidad que se quiere expresar en dB y q_{ref} es una cantidad de referencia.

Cuando la intensidad, potencia y presión de sonido son expresados en decibeles, se les llama nivel de intensidad de sonido, nivel de potencia de sonido y nivel de presión de sonido, respectivamente.

Haciendo uso de la ecuación 3.20, es posible calcular el nivel de intensidad de sonido si se toma como cantidad de referencia $I_{ref} = 1 \times 10^{-12}\text{watts/m}^2$

$$L_I = 10 \log_{10} \frac{I}{I_{ref}} \text{dB} \tag{3.21}$$

De manera similar se calcula el nivel de potencia de sonido, y para el caso particular de nivel de presión de sonido (SPL, por sus siglas en inglés) se utiliza:

$$\text{SPL} = 10 \log_{10} \frac{P^2}{P_{ref}^2} \text{dB} = 20 \log_{10} \frac{P}{P_{ref}} \text{dB} \tag{3.22}$$

donde P es la presión de sonido RMS (del inglés *Root Mean Square*) en Pa y $P_{ref} = 20\mu\text{Pa}$ que representa la presión de sonido del umbral auditivo a 1000Hz.

3.2.2. Umbral auditivo

El umbral auditivo varía según la frecuencia debido a la sensibilidad del oído humano. Este umbral nos indica el nivel de presión de sonido necesario para que un sonido con una frecuencia determinada pueda ser percibido. De acuerdo con [Terhardt79] uno puede obtener una buena aproximación del umbral auditivo utilizando la ecuación 3.23:

$$A(f) = 3.64 \left(\frac{f}{1\text{kHz}}\right)^{-0.8} - 6.5 e^{-0.6\left(\frac{f}{1\text{kHz}} - 3.3\right)^2} + 10^{-3} \left(\frac{f}{1\text{kHz}}\right)^4 \tag{3.23}$$

donde f es la frecuencia expresada en kHz y el resultado A está en dB. En la figura 3.8 se muestra una versión modificada de la gráfica de la ecuación 3.23 cuyo único cambio es el no permitir valores más grandes de 75dB.

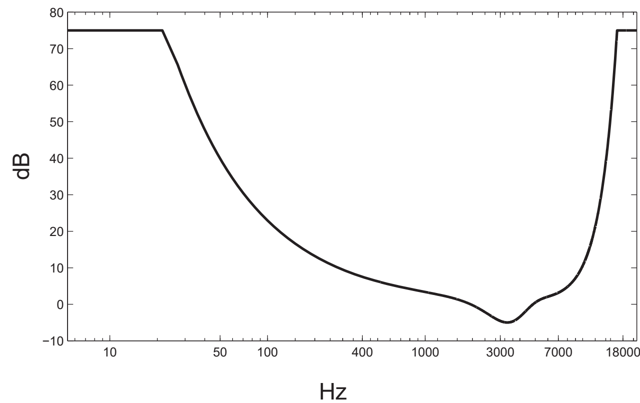


Figura 3.8: Umbral auditivo modificado para un máximo de 75dB

3.2.3. Enmascaramiento simultaneo

El enmascaramiento simultaneo es un fenómeno que se produce cuando un sonido (el enmascarador) hace imperceptible otro que tiene menor presión de sonido (el enmascarado). Como un ejemplo del enmascaramiento, imaginemos que se intenta tener una conversación con otra persona en un concierto de rock, el volumen tan alto de la música hará que la voz de la otra persona sea ininteligible en el mejor de los casos e imperceptible en el peor; la onda de sonido de la voz está allí, pero nuestro oído será incapaz de escucharla con claridad.

A manera ilustrativa, en la figura 3.9 se muestran 3 tonos, 2 de los cuales están bajo un umbral de enmascaramiento producido por un tono de diferente frecuencia pero con mayor nivel de presión de sonido. El oído humano será incapaz de percibir los componentes enmascarados.

El oído humano percibe los sonidos mediante una descomposición en frecuencia realizada en la cóclea: oscilaciones a lo largo de la membrana basilar determinan que frecuencia es audible. La localidad en que esta oscilación tendrá lugar depende de la frecuencia del sonido. En la figura 3.10 se muestra un dibujo de la cóclea y se indican las localidades en que se perciben ciertas frecuencias. Como puede observarse, las frecuencias altas se localizan cerca de la ventana oval y las bajas cerca del helicotrema. Para más detalles sobre la

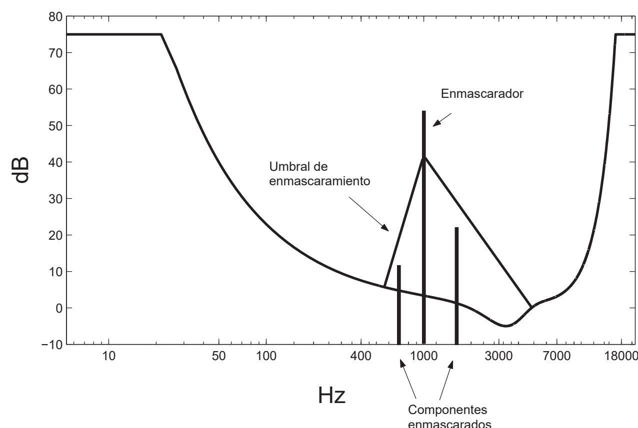


Figura 3.9: Ejemplo de enmascaramiento en frecuencia

anatomía y funcionamiento del oído humano se recomienda consultar [Eberhard Zwicker99].

También se puede observar que las localidades no están dispuestas de manera uniforme respecto a la frecuencia. Para modelar eficientemente el oído será necesario utilizar otra escala, llamada Escala Bark.

Escala Bark

Para asemejar la forma en la que el oído trabaja, se utiliza una escala no lineal que agrupa frecuencias en bandas, esta escala, llamada Escala Bark (en honor a Barkhausen[1]) describe una distancia fija a lo largo de la membrana basilar. Tomando en cuenta que el oído humano difícilmente percibe sonidos con una frecuencia superior a los 20000Hz, esto da como resultado 25 bandas críticas (Bark 0 a Bark 24) y basados en estudios clínicos que demuestran que la membrana basilar tiene una distancia de 32mm, esto indica que cada banda crítica representa aproximadamente 1.3mm de distancia a lo largo de la membrana basilar [Eberhard Zwicker99].

De acuerdo con [Eberhard Zwicker99] el uso de la siguiente ecuación para convertir

[1]Heinrich Georg Barkhausen (1881-1956), físico alemán que descubrió el efecto Barkhausen.

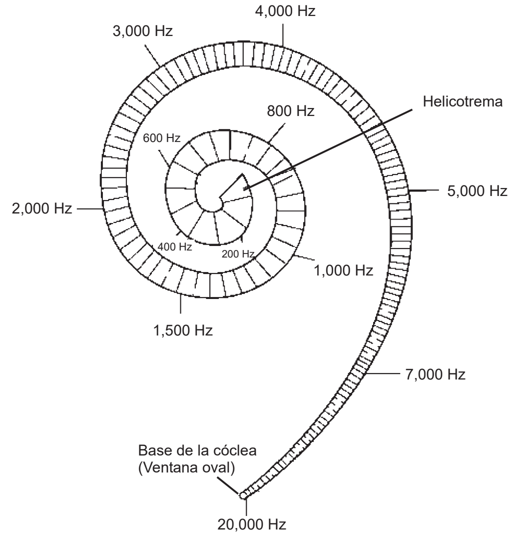


Figura 3.10: La cóclea y las localidades en que se perciben ciertas frecuencias

de kHz a Bark ha demostrado ser útil:

$$z(f)/\text{Bark} = 13 \arctan\left(\frac{0.76f}{1\text{kHz}}\right) + 3.5 \arctan\left(\frac{f}{7.5\text{kHz}}\right)^2 \quad (3.24)$$

donde f es la frecuencia en kHz.

Funciones de propagación de enmascaramiento

Un enmascarador producirá una excitación a lo largo de la membrana basilar que enmascarará un conjunto de frecuencias dependiendo de la frecuencia y el nivel de presión de sonido del enmascarador. Describir máscaras es más sencillo si se utiliza la Escala Bark que utilizando una escala lineal de frecuencia.

La función de propagación utilizada en este trabajo es la propuesta en el ISO/IEC MPEG Psychoacoustic Model 2 [MPEG93], su función está dada por:

$$F(dz) = 15.8111389 + 7.5(1.05dz + 0.474) - 17.5\sqrt{1.0 + (1.05dz + 0.474)^2} + 8\,\text{mín}(0, (1.05dz - 0.5)^2 - 2(1.05dz - 0.5)) + SPL_e - \Delta \quad (3.25)$$

donde $dz = z(f_{enmascarado}) - z(f_{enmascarador})$ y SPL_e es el nivel de presión de sonido del enmascarador. La máscara resultante debe ser disminuida por un Δ que se escoge de acuerdo

a la naturaleza del enmascarador, si el enmascarador es un tono puro un valor apropiado sería $\Delta = 16$dB [Eberhard Zwicker99], mientras que si el enmascarador es ruido entonces el valor adecuado será $\Delta = 6$dB [Moore96].

En la figura 3.11 se muestran 3 diferentes máscaras producidas por la ecuación 3.25 en el Bark 8.0 con niveles de 30db, 60db y 80db (usando $\Delta = 0$dB), note como mientras mayor sea el nivel de presión de sonido la máscara se propaga más.

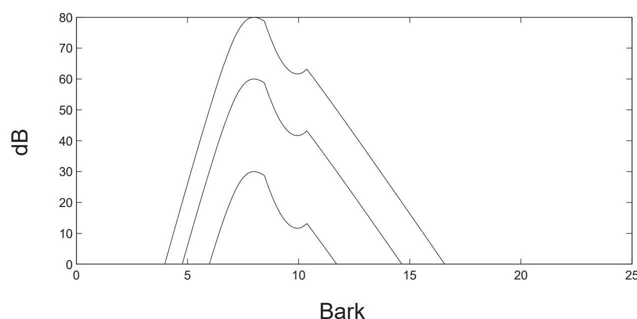


Figura 3.11: Función de propagación del MPEG Psychoacoustic Model 2

Existen muchos tipos de funciones de propagación, como la de ISO/IEC MPEG Psychoacoustic Model 1 [MPEG93], Schroeder [M. R. Schroeder79] y Terhardt [Terhardt79].

3.2.4. Enmascaramiento temporal

Además del enmascaramiento simultaneo, el enmascaramiento puede extenderse en el tiempo y puede ocurrir antes y después de la presencia de un enmascarador. Si el enmascaramiento sucede antes de la presencia de un enmascarador se le denominará pre-enmascaramiento o enmascaramiento pre-stimulus; al caso en que el enmascaramiento suceda cuando el enmascarador ya no esté presente se le conoce como post-enmascaramiento o enmascaramiento post-stimulus [Eberhard Zwicker99].

El pre-enmascaramiento es un fenómeno impredecible, pues sucede antes de la presencia de un enmascarador. El origen de este comportamiento radica en el tiempo que ocupan los sentidos para ajustarse y percibir las sensaciones. El tiempo de duración del pre-enmascaramiento es de alrededor de 20ms mientras que el post-enmascaramiento puede

durar 200ms. En la figura 3.12 se ilustra el efecto del enmascaramiento temporal.

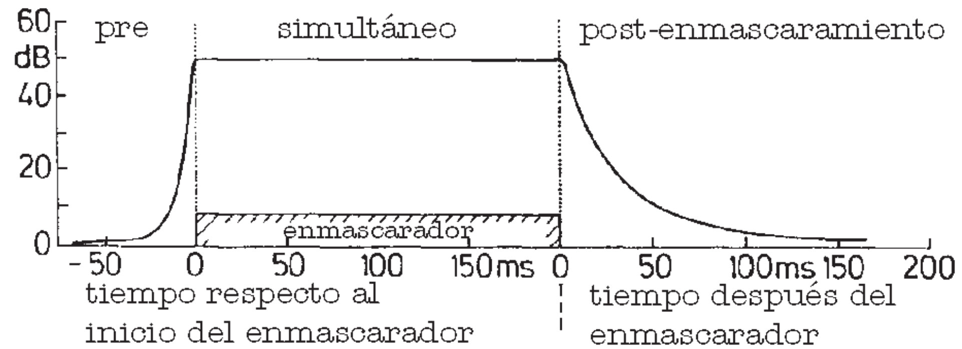


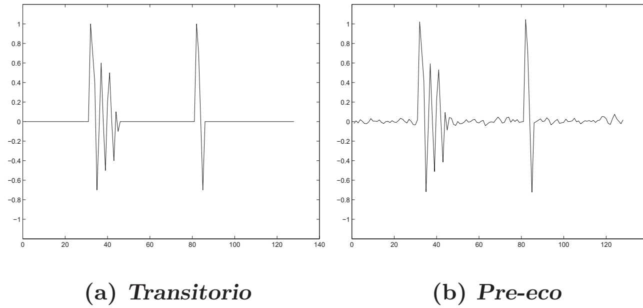
Figura 3.12: Efecto del enmascaramiento en tiempo (versión traducida de la mostrada en [Eberhard Zwicker99])

Pre-eco

El pre-eco es un artefacto que se hace perceptible cuando la energía de un transitorio se extiende en el tiempo debido a cuantización en el dominio de la frecuencia. Cuando existe pre-eco se percibe un pequeño eco antes de la presencia del transitorio.

En la figura 3.13 se muestra la señal original de un transitorio y la señal resultante después de aplicar cuantización a los coeficientes de la DFT. Note como el ruido de cuantización que se agregó al espectro se extiende a lo largo de toda la señal. En la figura 3.13(a) se pueden apreciar 2 transitorios y 3 porciones de silencio, note como en la figura 3.13(b) el silencio no es puro. Durante lapsos de silencio no hay componentes que enmascaren el ruido de la cuantización, es entonces cuando es útil aprovechar el pre-enmascaramiento para hacer este ruido imperceptible.

Para que el pre-enmascaramiento enmascare efectivamente el pre-eco es necesario aumentar la resolución en tiempo cuando se detecte la presencia de un transitorio, esto equivale a reducir el tamaño del bloque de tal manera que el ruido de cuantización sólo se propague en un lapso de tiempo más pequeño, inferior a 20ms.



(a) ***Transitorio*** (b) ***Pre-eco***

Figura 3.13: Ejemplo del efecto de pre-eco

3.2.5. Umbral global de enmascaramiento

El umbral global de enmascaramiento es una curva que indica el nivel de presión de sonido necesario para que un componente espectral determinado sea percibido por el oído humano. Se forma a partir de las curvas de enmascaramiento de los componentes espectrales, el enmascaramiento temporal y el umbral auditivo. El problema radica en la forma de combinarlos. Es posible que se desee sumar, sub-sumar, sobre-sumar o simplemente elegir el valor mayor de las máscaras en una frecuencia determinada. Cualquiera de estos casos puede ser descrito de acuerdo con la siguiente fórmula (versión modificada de la mostrada en [Marina Bosi02]):

$$I_N[f] = \left(\sum_{n=0}^{N-1} I_n^{\alpha}[f] \right)^{\frac{1}{\alpha}} \tag{3.26}$$

donde $I_N[f]$ representa la intensidad de la curva de enmascaramiento resultante de combinar N curvas de enmascaramiento con intensidades $I_n[f]$ en una frecuencia f, note que I_n es calculado utilizando alguna función de propagación de enmascaramiento, como la de la ecuación 3.25. El parámetro α define la forma en que las curvas se combinan. Si $\alpha = 1$ la ecuación equivale a sumar las intensidades, mientras que $\alpha = +\infty$ equivale a utilizar el máximo, un valor $\alpha < 1$ sobre-suma, esto es, arroja una intensidad mayor que la suma de los componentes. De acuerdo con [Lufti83] un valor de $\alpha = 0.33$ es apropiado para enmascaradores de intensidad comparable, implicando que al combinar dos curvas de igual intensidad resulta una sola curva con una intensidad 8 veces mayor.

Una vez que se tiene la curva de enmascaramiento formada por los enmascaradores individuales ésta se combina con el umbral auditivo, la práctica común es escoger el valor máximo en cada frecuencia. La curva resultante es el umbral global de enmascaramiento.

La utilidad del umbral global de enmascaramiento es la de poder asignar ruido sin que éste sea perceptible, en el caso de codificación perceptual, ruido de cuantización. Mientras mayor sea el umbral global de enmascaramiento con relación al componente espectral menor será el número de bits necesarios para cuantizarlo. En otras palabras, es deseable tener una razón señal a ruido superior a la razón señal a máscara. Donde la razón señal a máscara (SMR, por sus siglas en inglés) es la diferencia (en dB) entre un componente con frecuencia f, $X(f)$, y el valor del umbral global de enmascaramiento para esta misma frecuencia, $mask(f)$:

$$SMR(f) = X(f) - mask(f) \tag{3.27}$$

La diferencia entre la razón señal a mascara y la razón señal a ruido es conocida como razón ruido a mascara:

$$NMR(f) = SMR(f) - SNR(f) \tag{3.28}$$

si el NMR es negativo entonces el ruido de cuantización será imperceptible por lo que siempre será deseable mantenerlo negativo o en el caso que el número de bits disponibles sea pequeño [2], mantenerlo en el valor mínimo posible.

Si recordamos en la ecuación 2.10 se apreciaba que por cada bit que se agregara al cuantizador el SNR se incrementaría en 6dB. Es entonces sencillo calcular el tamaño del cuantizador para que la razón señal a ruido sea superior a la razón señal a máscara.

3.3. Nivel de Presión de Sonido y el Procesamiento Digital de Señales

Obtener el nivel de presión de sonido de un origen sonoro es un proceso sencillo cuando la presión de sonido puede ser medida directamente. Pero, ¿qué sucede en el caso de una grabación?, la ganancia y el volumen de la señal no son conocidos a priori, y son

[2]Aplica en el caso de codificación CBR o VBR con límites

determinados por el usuario al momento de reproducción, por ejemplo, éste puede escoger escuchar una pieza de música clásica a un volumen considerable mientras que puede preferir un volumen módico para sonidos estridentes.

Esta situación por trivial que parezca, representa un problema para la codificación perceptual, pues en el proceso de análisis en el dominio de la frecuencia es necesario llevar el espectro a una escala en decibeles.

Para datos PCM con un cuantizador de 16-bits, se asume que una sinusoide con una amplitud igual al nivel de sobrecarga del cuantizador tendrá un nivel de presión de sonido de 96dB ($\frac{6\text{dB}}{\text{bit}} \times 16\text{bits}$), si se define el nivel de sobrecarga del cuantizador como 1.0, el nivel de presión de sonido de una sinusoide de amplitud A estará dado por [Marina Bosi02]:

$$\text{SPL} = 96\text{dB} + 10\log_{10}(A^2) \tag{3.29}$$

Para una transformada discreta de Fourier, podemos calcular el nivel de presión de sonido de la siguiente forma:

$$\text{SPL}_{\text{DFT}} = 96\text{dB} + 10\log_{10}\left(\frac{4}{N^2 < w^2 >}|X[k]|^2\right) \tag{3.30}$$

De manera similar, para una MDCT:

$$\text{SPL}_{\text{MDCT}} = 96\text{dB} + 10\log_{10}\left(\frac{8}{N^2 < w^2 >}|X[k]|^2\right) \tag{3.31}$$

en donde $< w^2 >$ representa la ganancia de la ventana, y para cualquier ventana válida para la MDCT su valor es 0.5. Para la ventana de Hanning este valor es 3/8 [Marina Bosi02].

3.4. Un ejemplo del proceso de cuantización con un modelo perceptual

Ahora ilustremos gráficamente el proceso de cuantización basándonos en información de un modelo perceptual.

Primero calculemos el umbral global de enmascaramiento. Como se mencionó anteriormente es común utilizar una DFT para esta tarea. Esta representación en el dominio

de la frecuencia se lleva a una escala decibel y después se calculan las máscaras individuales. Una vez que se tienen las mascaras, se procede a combinarlas con el umbral auditivo. En la figura 3.14 se puede apreciar la DFT de un bloque obtenido de un archivo de sonido y el umbral global de enmascaramiento que produce.

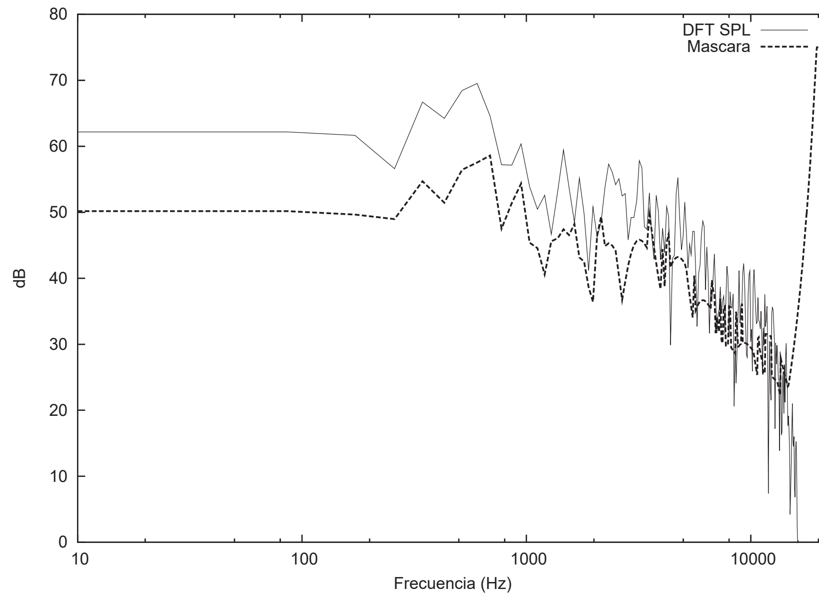


Figura 3.14: Transformada Discreta de Fourier y el umbral global de enmascaramiento que produce

Este umbral, que para efectos de este ejemplo será llamado máscara, nos será de utilidad para determinar con que precisión cuantizar los coeficientes de la MDCT.

En la figura 3.15 se muestra la MDCT (en dB) del mismo bloque y la máscara calculada con la DFT. Note las diferencias entre la MDCT y la DFT. Puesto que la información a codificar son los coeficientes de la MDCT se optó por utilizar ésta para determinar el tamaño del cuantizador. Además, en la fase de pruebas se obtenían mejores relaciones de compresión y se eliminaban algunos artefactos.

Una vez que se ha determinado la forma de cuantizar, se procede a efectuar este proceso. En la figura 3.16(a) se ilustran la MDCT y su versión cuantizada. Al parecer, la diferencia entre ambas es pequeña. Sin embargo, en la figura 3.16(b) se muestran ambas en

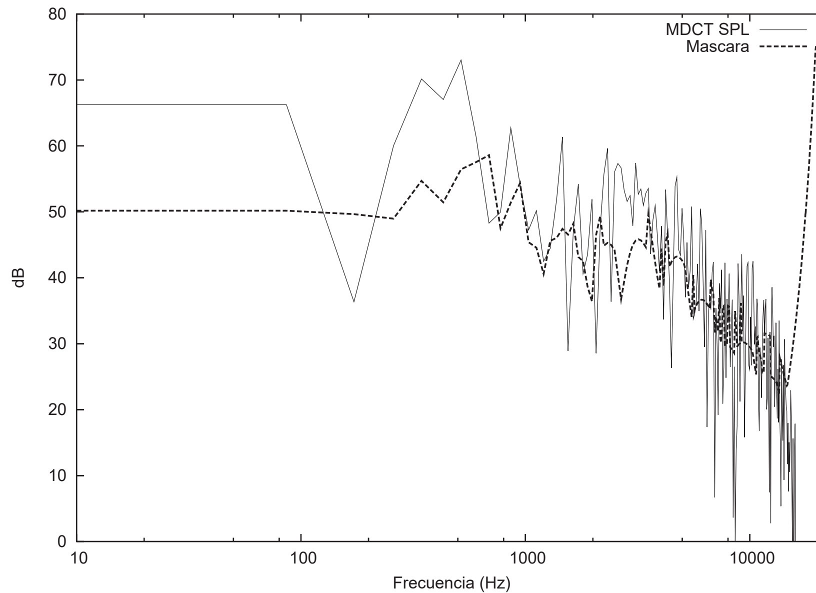
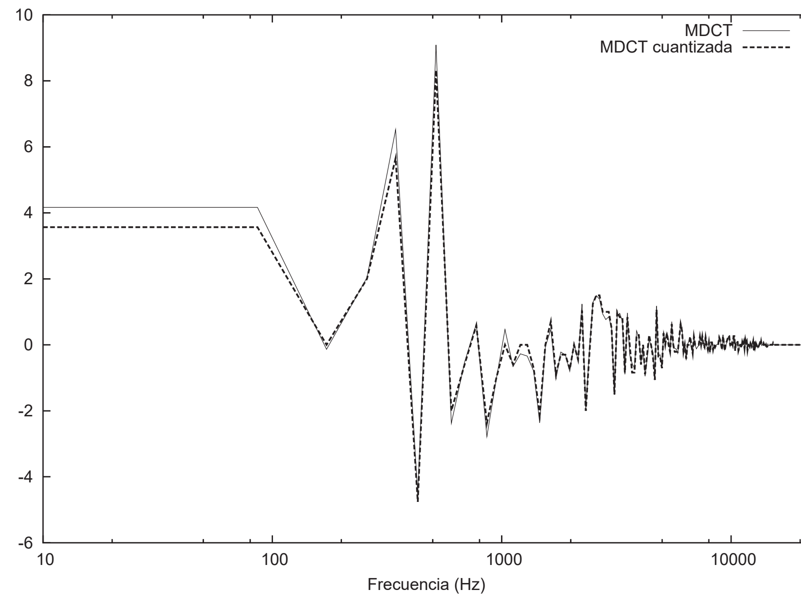


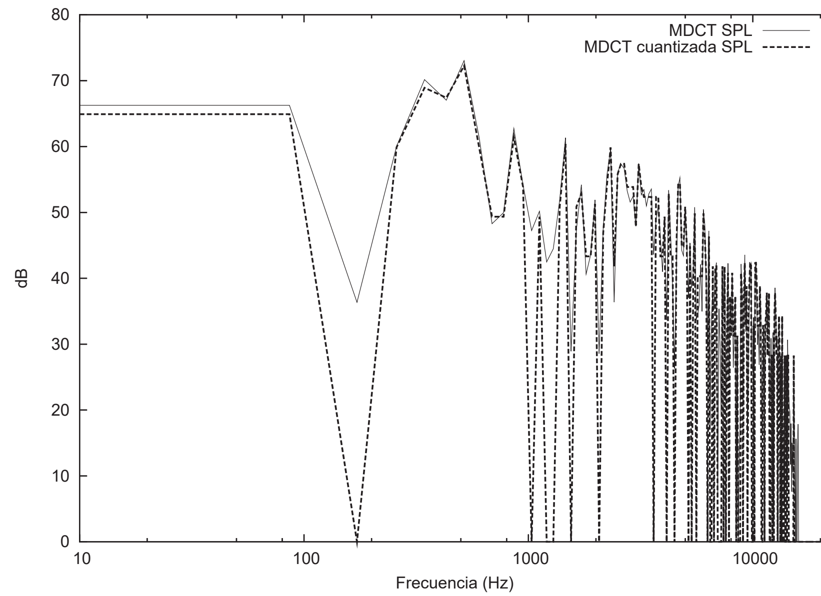
Figura 3.15: MDCT y la máscara

decibeles, en esta representación se puede apreciar la gran diferencia entre ellas.

Esta diferencia se pretende que sea inaudible. Esto es, el error entre la señal original y la señal que produzca nuestro codec debe estar por debajo de la máscara. En la figura 3.17 se muestra la MDCT en decibeles del error, note como el ruido de cuantización tiene una forma que cae por debajo del umbral global de enmascaramiento. Si nuestro modelo psicoacústico y el proceso de cuantización se han hecho de manera adecuada, el oído humano no será capaz de distinguir entre la señal original y la comprimida. Este aspecto es fundamental en todo codec perceptual y es un área en la que se investiga constantemente.



(a) *MDCT y MDCT cuantizada*



(b) *MDCT y MDCT cuantizada (dB)*

Figura 3.16: MDCT y MDCT después del proceso de cuantización

Figura 3.17: Error de la señal y la máscara

3.5. Codificación Aritmética

Una vez que se ha completado el proceso de cuantización con un modelo psicoacústico, la información se empaqueta en algún flujo de bits especificado. A pesar de que se ha efectuado con éxito compresión con pérdida, todavía es posible eliminar redundancia de los datos. Para esto se utiliza algún método de compresión sin pérdida, con el fin de no alterar la calidad subjetiva de la señal obtenida.

Es típico que esta segunda "pasada" comprima entre un 5 % y un 15 % nuestros datos. Como se puede apreciar, es poco lo que aporta si se compara con la compresión con pérdida que varía entre 70 % y 95 %. Sin embargo, si se desea comprimir al máximo, es necesario efectuar este paso.

La codificación aritmética [Eric Bodden02] [William H. Press92] [Ian H. Witten99] codifica una cadena de símbolos en una sola palabra o número. En cambio, en la codificación Huffman [Huffman52] se asigna un código a cada símbolo, la principal desventaja de los códigos Huffman es que por cada símbolo se utiliza un número entero de bits. Puesto que en codificación aritmética se codifica toda la cadena, es posible que para representar un símbolo se utilicen cantidades no enteras de bits o incluso utilizar menos de un bit.

Ambos esquemas están sujetos a una distribución o modelo. Sin embargo, si tanto en codificación Huffman como en codificación aritmética se cuenta con el modelo adecuado, la codificación aritmética comprimirá mejor y se aproximará a la entropía. Incluso, ningún otro algoritmo podrá mejorar el resultado utilizando el mismo modelo [Eric Bodden02].

Como notación, usemos s para denotar una cadena $s \in \Sigma^+$ de símbolos obtenidos de un alfabeto Σ, P el modelo que se utiliza siendo $P(\alpha)$ la probabilidad de ocurrencia de un símbolo $\alpha \in \Sigma$.

Como ejemplo, sea $\Sigma = \{a, b, c\}$ y $s = bccb$, y asumamos un modelo con distribución de probabilidad uniforme (esto es $P(a) = P(b) = P(c) = \frac{1}{3}$), el proceso de codificación se muestra en la figura 3.18.

Note como en un inicio la distribución es uniforme y se tiene un intervalo $[0, 1]$. Cuando b es seleccionado el intervalo es actualizado a los valores que comprende b, esto es $[0.3333, 0.6667]$. Además, la distribución de probabilidad es actualizada para reflejar

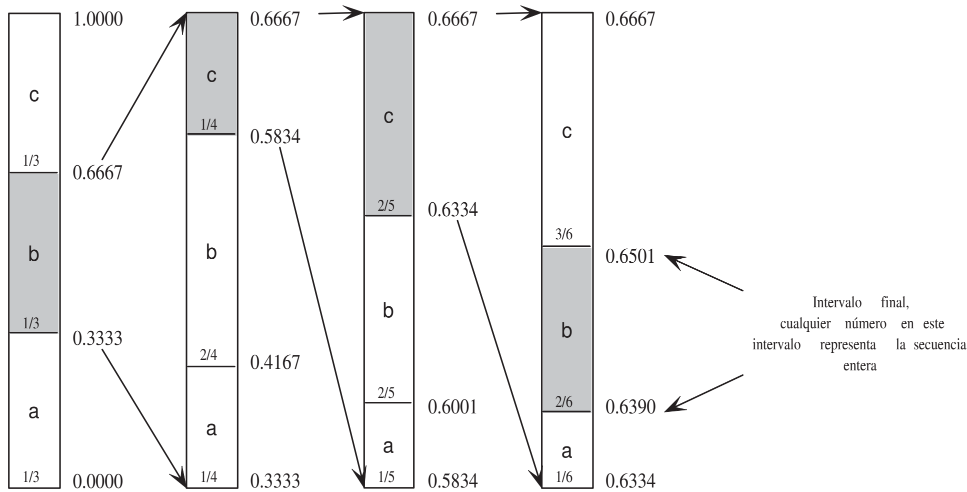


Figura 3.18: Codificación aritmética con un modelo adaptivo de orden 0.

la ocurrencia del símbolo, en este caso, $P(a) = \frac{1}{4}, P(b) = \frac{2}{4}, P(c) = \frac{1}{4}$. Note como esta actualización en el modelo hace que el intervalo que comprende b sea más grande que el de a y c, pues éste es proporcional a la distribución.

De manera similar sucede cuando c y c son seleccionados posteriormente. Para representar la cadena entera, será necesario almacenar cualquier número en el intervalo final $[0.6390, 0.6501]$, digamos 0.64.

Para decodificar, la distribución se inicializa del mismo modo que en el proceso de codificación (en este ejemplo, de manera uniforme), el valor 0.64 indica que el primer símbolo es b, pues 0.64 cae en el intervalo de b. Como el símbolo b ha sido seleccionado, entonces se actualiza el modelo tal y como se hubiera hecho durante el proceso de codificación. Posteriormente se nota que 0.64 cae en el intervalo de c, entonces se selecciona el símbolo y se actualiza el modelo, siguiendo este proceso se reconstruye $bccb$ enteramente.

Aquí se presenta una situación interesante, el número 0.64 representa una cadena de la forma $bccb \cdot \beta_i$ ($\cdot$ denotando concatenación), donde $\beta = \{\beta_i \mid \beta_i \in \Sigma^*$, β_i satisface el modelo para 0.64 después de $bccb\}$. Todo β_i tiene un tamaño $n \geq 0$ diferente. Como puede verse, el conjunto β tiene una cardinalidad infinita, y por ende, existe una cadena

de longitud infinita que satisface el modelo para 0.64. Es entonces necesario indicarle al decodificador cuando detenerse, pues éste podría continuar de manera indefinida generando cadenas cada vez más grandes. Para decodificar con éxito será necesario que $\beta_i = \varepsilon$, esto es, la cadena de longitud 0.

Para solucionar este problema es común utilizar un símbolo terminal cuya frecuencia es 1, pues sólo aparece una vez: al final de la cadena. La ocurrencia de este símbolo indica que el proceso de decodificación debe terminar. La penalización introducida por incluir este símbolo es poco significativa y es una mejor opción que almacenar el número de símbolos en la cadena.

El modelo que se ilustra en la figura 3.18 se dice adaptivo, pues con la ocurrencia de cada símbolo ajusta la probabilidad de ocurrencia. Un modelo que toma en cuenta el contexto es llamado de orden N, siendo N el tamaño del contexto. El contexto es de particular importancia, pues un símbolo puede tener una probabilidad baja de ocurrencia, sin embargo el mismo símbolo puede tener una posibilidad de ocurrencia muy grande si es precedido por otro.

En este trabajo, se utilizó un modelo adaptivo de orden 0 y símbolos de 8-bits para efectuar compresión sin pérdida a los paquetes que contienen los coeficientes cuantizados y factores de escalamiento.

3.6. Conclusiones

Se han revisado los aspectos fundamentales de la codificación perceptual. La MDCT es una transformada que utiliza cancelación de aliasing en el dominio del tiempo y es usada en codecs del estado del arte. Para que la MDCT ofrezca reconstrucción exacta es necesario que las ventanas de análisis y de síntesis cumplan con ciertas restricciones; la ventana derivada de Kaiser-Bessel es una ventana que cumple con estas restricciones y ofrece las ventajas de la ventana Kaiser-Bessel. Además se mostró un método para normalizar cualquier ventana y hacerla válida para su uso con la MDCT. [Marina Bosi02]

El oido humano tiene características que pueden ser explotadas en la codificación de audio, en cuyo caso se denominará codificación perceptual de audio. El enmascaramiento

simultaneo (o en frecuencia) es un fenómeno en el que un sonido (el enmascarador) hace que otros sonidos sean imperceptibles (los enmascarados), para determinar que componentes serán enmascaradas se utiliza un modelo de propagación de la máscara que depende del nivel de presión de sonido y de la frecuencia del enmascarador. Por simpleza es mejor utilizar la escala Bark, que es más apropiada para modelar el oído humano. El enmascaramiento en tiempo tiene la utilidad de poder ocultar el pre-eco, siempre y cuando la resolución en tiempo sea suficiente para que el ruido de cuantización sea enmascarado por un pre-stimulus. Finalmente cada una de las curvas de enmascaramiento de los enmascaradores es combinada con el umbral auditivo para formar el umbral global de enmascaramiento. Una vez que se tiene este umbral se utilizará para cuantizar los componentes espectrales con el menor número posible de bits haciendo el ruido de cuantización imperceptible. Este ruido será imperceptible siempre y cuando la razón señal a ruido sea mayor que la razón señal a máscara, o en otras palabras si la razón ruido a máscara es negativa.

En concreto, en el desarrollo de este trabajo, se utiliza la implementación rápida de la MDCT expuesta (usando una ventana KBD con $\alpha = 0.4$) y la función de propagación de enmascaramiento del MPEG Psychoacoustic Model 2 (ecuación 3.25). Una vez que los coeficientes cuantizados (compresión con pérdida) han sido empaquetados se comprimen sin pérdida utilizando codificación aritmética por byte con un modelo adaptivo de orden 0. En el siguiente capítulo se explica a más detalle el proceso de codificación.

Capítulo 4

Implementación de Un Esquema de Codificación

"La inteligencia consiste no sólo en el conocimiento, sino también en la destreza de aplicar los conocimientos en la práctica".
Aristóteles.

En este capitulo se explican los pasos que el codificador efectúa para lograr una compresión perceptual de un archivo WAV PCM muestreado a 44100Hz, cuantizado con 16-bits, estéreo.

4.1. Características del CODEC Implementado

Se implementó un codec de transformación basado en MDCT, una transformada ampliamente usada en codecs del estado del arte. La MDCT se aplica a bloques de 512 muestras traslapados en un 50 %, dichos coeficientes son cuantizados basándose en datos obtenidos del modelo psicoacústico para dar forma al ruido de cuantización de modo que éste se sitúe por debajo del umbral global de enmascaramiento y por ende, sea inaudible.

Los coeficientes son codificados a un archivo de salida con un formato especificado posteriormente. También se implementó un decodificador cuya salida es nuevamente un archivo WAV con las características del archivo de entrada (44.1kHz, 16-bits, estéreo). El formato WAV PCM se eligió por ser un formato que preserva la calidad del origen, no

está comprimido, es simple y es ampliamente adoptado.

4.2. Proceso de Codificación

Puesto que se manejan dos canales, es necesario primero tomar un bloque de muestras del primer canal, procesarlo y codificarlo, para posteriormente tomar un bloque del segundo canal, esto tiene la finalidad de dejar en el archivo bloques multiplexados para poder efectuar una decodificación más eficiente.

En la figura 4.1 se muestra el funcionamiento del codificador y cada uno de los pasos que sigue se explican a continuación.

4.2.1. Cambio de rango dinámico de la señal

El rango dinámico de un archivo WAV PCM de 16-bits es discreto y está contenido en $[-32768, 32767]$. Para ser consistentes con la teoría expuesta en capítulos anteriores este rango se cambia al rango dinámico $[-1.0, 1.0]$ de los números reales. Por otro lado es común que las ecuaciones utilizadas en compresión de audio perceptual asuman que la señal se encuentra en este rango.

4.2.2. Cálculo de la Transformada Discreta de Fourier

En los experimentos llevados a cabo, se observó que él modelo psicoacústico funcionaba mejor si se tenía como entrada una Transformada Discreta de Fourier en lugar de la MDCT, esta mejoría es reflejada en la calidad percibida de la señal.

Antes de calcular la Transformada Discreta de Fourier, el bloque de datos es multiplicado por una ventana de Hanning para evitar la introducción de frecuencias no deseadas en el espectro. Para calcular la DFT de este bloque se utiliza una FFT, en específico se utiliza la implementación de FFTW.[1]

[1] Fastest Fourier Transform in the West. www.fftw.org

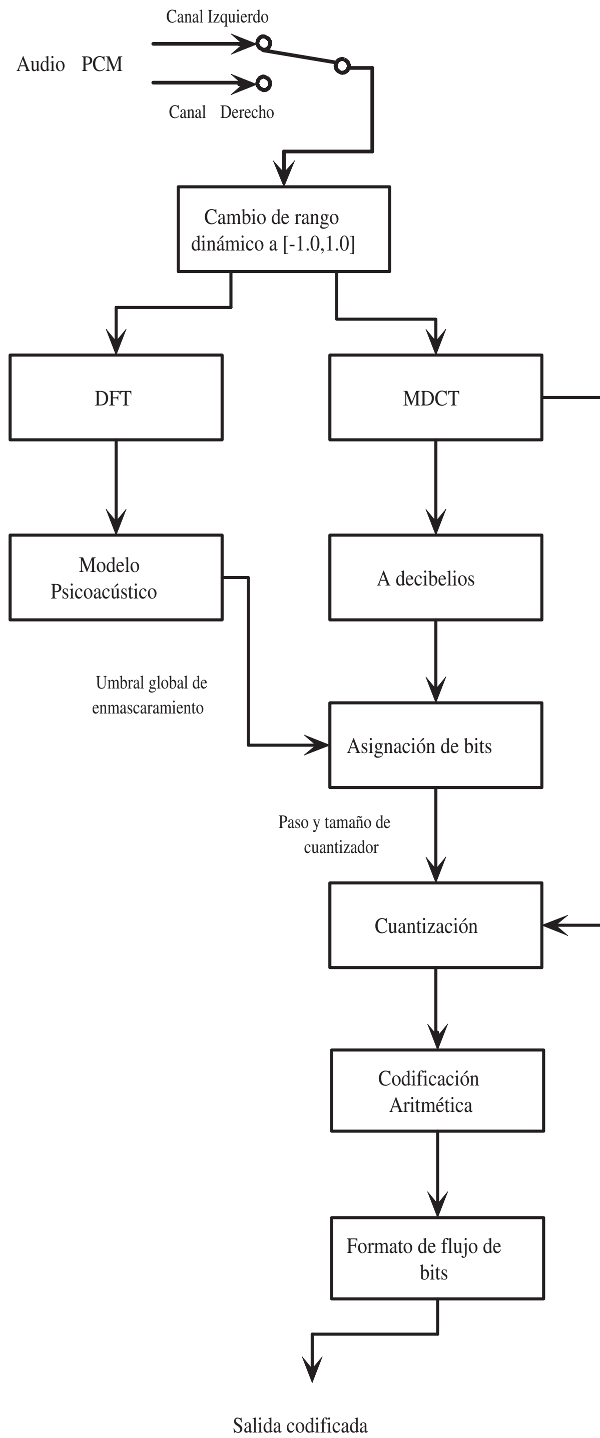


Figura 4.1: Diagrama de bloques para la codificación

4.2.3. Modelo Psicoacústico

La finalidad del modelo psicoacústico es calcular el umbral global de enmascaramiento. Para poder realizar esto, los coeficientes de la DFT son primeramente llevados a una escala decibel y posteriormente el espectro se convierte a la escala Bark.

El espectro es dividido en 25 bandas que corresponden a los 25 Barks (0-24). De cada una de las bandas se escoge el máximo pues este tendrá mayor energía y por ende será el mejor enmascarador, además que este esquema no es computacionalmente costoso. Con cada uno de los máximos se calcula la máscara individual utilizando la ecuación del *MPEG Psychoacoustic Model 2* (ecuación 3.25 [MPEG93]) y la máscara resultante es disminuida en $\Delta = 12$dB, utilizar un valor de $\Delta = 0$dB indicaría que el propio enmascarador está enmascarado por sí mismo, lo cual es ilógico.

El umbral global de enmascaramiento se calcula de manera incremental en base al siguiente criterio:

- Si la máscara a agregar es mayor en 3dB al umbral de enmascaramiento, el umbral de enmascaramiento toma el valor de la máscara a agregar más 6dB, en caso contrario se mantiene el valor.

- Para combinar el umbral auditivo con el umbral de enmascaramiento, se escoge el valor máximo de los dos, con lo que se completa el umbral global de enmascaramiento.

Este criterio fue propuesto pues en el caso de combinar un enmascarador de alta intensidad con un enmascarador de baja intensidad, el enmascarador más pequeño tiene poco que contribuir a la máscara compuesta. No siendo así el caso de enmascaradores con intensidades similares, en cuyo caso la máscara tiene una intensidad mayor que la suma de ambos. En contraste, el esquema propuesto en [Lufti83] de utilizar un valor de $\alpha = 0.33$ (véase la ecuación 3.26) implica hacer suma de potencias (no enteras), lo cual es computacionalmente costoso; nuestro esquema evita el uso de potencias, haciéndolo una opción más rápida.

4.3. Cálculo de la MDCT

Por otro lado, el bloque de datos obtenido de la entrada es multiplicado por una ventana Derivada de Kaiser-Bessel con $\alpha = 4.0$ y se calcula su Transformada Discreta de Coseno Modificada (a través de la implementación rápida expuesta en la sub-sección 3.1.4). Estos coeficientes se cuantizarán, pero también son utilizados en la fase de análisis, en especifico, para el cálculo de la razón señal a máscara.

4.3.1. Llevar los coeficientes a decibelios

Adicionalmente, pero preservando los coeficientes originales, los coeficientes de la MDCT son convertidos a una representación en decibelios y llevados a la escala Bark (ecuaciones 3.31 y 3.24, respectivamente) para así poder obtener la razón señal a máscara. En otros esquemas de codificación de audio se utilizan los coeficientes de la DFT para calcular la razón señal a máscara, en la fase de pruebas se observó que utilizar la MDCT en lugar de la DFT eliminaba algunos artefactos y generaba mejores relaciones de compresión, al menos para los archivos de prueba.

4.4. Asignación de bits

Utilizar el ancho de banda equivalente a un Bark es útil para que el ruido de cuantización esté localizado con un ancho de banda crítica que se aproxime al del oído humano. Cada banda es cuantizada de acuerdo a la razón señal a máscara de la banda, que en este trabajo se calcula de la siguiente manera:

$$\text{SMR}_b = \text{máx}(b) - \text{mín}(mask_b) \tag{4.1}$$

donde b es la banda y $mask_b$ es la porción del umbral global de enmascaramiento que corresponde a la banda b. La asignación de bits se hace para cada banda, esto es, cada banda es cuantizada con un cuantizador de tamaño diferente en bits. Para calcular el número de bits necesarios para que el ruido de cuantización dentro de una banda no sea perceptible se

propone utilizar la siguiente ecuación [Marina Bosi02]:

$$R_b = \frac{P}{K_p} + \frac{\ln 10}{20 \ln 2}\left(\text{SMR}_b - \frac{1}{K_p} N_b \text{SMR}_b\right) \tag{4.2}$$

siendo R_b el tamaño en bits del cuantizador para la banda b, P el número de bits disponibles para las mantisas (esto es, sin factores de escalamiento), K_p el número de coeficientes diferentes de cero, N_b el número de coeficientes en la banda b y SMR_b la razón señal a máscara de la banda.

4.5. Cuantización

Una vez que se ha calculado el tamaño en bits del cuantizador de cada banda, se procede a hacer la cuantización. Puesto que los valores que se van a almacenar son discretos, es necesario contar con factores de escalamiento que ofrezcan una buena granularidad y sea posible codificarlos utilizando pocos bits. Para calcular el paso de cuantizador (o factor de escalamiento) se utiliza la siguiente ecuación:

$$q_{step} = 2^{0.25i} \tag{4.3}$$

donde $i = -32, -31, \ldots, 31$ es un entero de 6 bits. Este factor de escalamiento tiene una granularidad de aproximadamente 1.5dB:

$$1.5\text{dB} \approx 20\frac{\log(2^{0.25})}{\log 10} \tag{4.4}$$

Los valores cuantizados se calculan de la siguiente manera:

$$q(X) = \text{nint}\left(\frac{X}{q_{step}}\right) \tag{4.5}$$

siendo X un coeficiente de la MDCT y nint una función que regresa el entero más cercano.

4.6. Formato de la salida

Finalmente los elementos cuantizados son codificados a un archivo de salida con un formato propuesto de acuerdo a las características del codec. Abarca la información necesaria para poder interpretar los datos que se han codificado, esta información denominada

información de lado (del inglés *side information* o *ancillary data*) comprende básicamente los factores de escalamiento, tamaño de cuantizador, tamaño del bloque, longitud de paquetes, entre otros.

La salida es codificada de acuerdo al formato de la figura 4.2. El archivo consiste de una cabecera de archivo, y un conjunto de paquetes multiplexados que corresponden a cada uno de los canales. A continuación se detalla el formato de la cabecera de archivo y el formato de cada paquete.

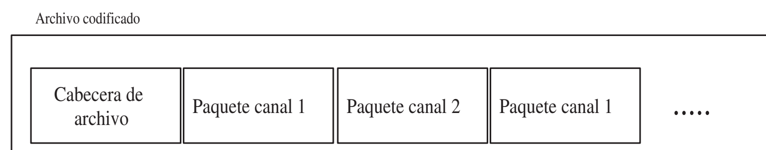


Figura 4.2: Formato de archivo

4.6.1. Cabecera de Archivo

La cabecera del archivo almacena información general necesaria para que sea posible decodificar el archivo. En la figura 4.3 se ilustran el acomodo de los campos de la cabecera.

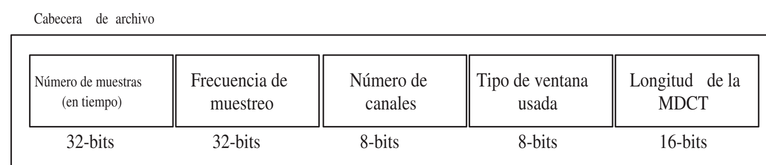


Figura 4.3: Cabecera de archivo

A continuación se detallan cada uno de los campos de la cabecera:

- Número de muestras (en tiempo): Entero sin signo de 32-bits que almacena la longitud total del archivo en muestras de tiempo.

- Frecuencia de muestreo: Entero sin signo de 32-bits que especifica la frecuencia de muestreo de la salida (en Hz), para efectos de este trabajo, su valor siempre es 44100, aunque puede ser cualquier valor entre 1 y $2^{32} - 1$. La duración en segundos del archivo será determinada como el cociente del número de muestras y la frecuencia de muestreo.

- Número de canales: entero sin signo de 8-bits que indica el número de canales de la señal, en este trabajo siempre es 2.

- Tipo de ventana usada: entero sin signo de 8-bits que almacena el tipo de ventana que se usó con la MDCT.

  - 0x0 indica una ventana Seno.
  - 0x1 indica una ventana derivada de Kaiser-Bessel con $\alpha = 4.0$.
  - 0x2 indica la ventana utilizada en Vorbis [2].
  - 0x3 indica una ventana rectangular.

  la ventana escogida en este trabajo es la derivada de Kaiser-Bessel, que es también usada en el estándar MPEG-2 AAC [MPEG97]. En la fase de pruebas se compararon la ventana seno, Vorbis y KBD. Al usar una ventana KBD se obtenían mejores relaciones de compresión.

- Longitud de la MDCT: entero sin signo de 16-bits que indica el número de componentes espectrales que contiene cada MDCT, su valor en este trabajo es 256.

4.6.2. Paquete

Cada paquete contiene información secundaria para decodificar los coeficientes cuantizados. El número de coeficientes cuantizados en cada paquete corresponde al indicado por *Longitud de la MDCT* en la cabecera del archivo. En la figura 4.4 se muestra la estructura de cada paquete.

A continuación se detalla cada campo:

[2]The Ogg Vorbis CODEC project http://www.xiph.org/ogg/vorbis/

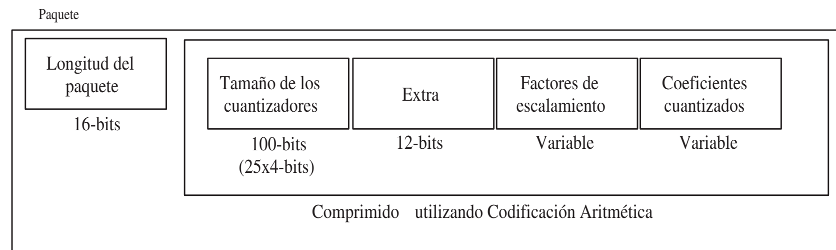


Figura 4.4: Paquete

- Longitud del paquete: entero sin signo de 16-bits que almacena la longitud en bytes de cada paquete (sin incluir los dos bytes ocupados por este campo). Por la naturaleza de este codec, el tamaño del paquete es variable y es determinado de acuerdo a las demandas de los componentes espectrales.
- Tamaño de los cuantizadores: conjunto de 25 enteros sin signo de 4-bits que contienen el tamaño del cuantizador de cada una de las 25 bandas. El cuantizador más grande disponible es de 15-bits ($2^4 - 1$), y un valor de 0 indica que la banda no ha sido codificada.
- Extra: sin uso, pero pensado para futuras ampliaciones, se escogió el valor de 12-bits ya que con esto la porción del paquete que siempre estará presente tendrá un tamaño de 16 bytes.
- Factores de escalamiento: enteros de 6-bits que almacenan el valor de i de la ecuación 4.3. Cabe señalar que si una banda no ha sido codificada (el tamaño de cuantizador es cero), su factor de escalamiento tampoco es almacenado.
- Coeficientes cuantizados: Los componentes espectrales de cada banda cuantizados de acuerdo al tamaño de cuantizador especificado y el factor de escalamiento indicado en los campos anteriores. De nuevo, si el tamaño del cuantizador para una banda determinada es 0, los coeficientes de esta banda no son almacenados.

El paquete (excepto el campo longitud de paquete) es comprimido sin pérdida utilizando codificación aritmética por byte con un modelo adaptivo de orden 0, las rutinas im-

plementadas fueron obtenidas a partir de la modificación de los códigos de [Eric Bodden02][3]. Este esquema logró disminuir la tasa promedio de datos en aproximadamente 7 % sin alterar la calidad del sonido.

4.7. Decodificación

El proceso de decodificación es mucho más simple y rápido que el de codificación. Tan sólo es necesario descomprimir cada paquete, leer los campos de las cabeceras, los coeficientes cuantizados de cada paquete y decuantizarlos de acuerdo con la ecuación 4.6.

$$\hat{X}(q) = q \times q_{step} \tag{4.6}$$

siendo q_{step} calculado con la ecuación 4.3, q la mantisa y $\hat{X}$ el componente decuantizado.

Una vez que todos los coeficientes de un paquete han sido decuantizados se les aplica la transformada discreta de coseno modificada inversa (IMDCT) con la implementación rápida expuesta en la subsección 3.1.4.

Posteriormente se multiplican por la ventana apropiada (en este caso KBD $\alpha = 4.0$) y se realiza el traslape y suma. Por cada canal se requiere un buffer pues éstos están multiplexados. La señal reconstruida se encuentra en un rango dinámico de $[-1.0, 1.0]$ por lo que es necesario llevarla al rango discreto de $[-32768, 32767]$ antes de ser escrita como un archivo WAV PCM. En la figura 4.5 se muestra el proceso que se sigue para la decodificación.

4.8. Conclusiones

Se ha expuesto el funcionamiento del codec perceptual propuesto haciendo uso de los principios expuestos en capítulos anteriores. Las tasas de datos, pruebas de calidad perceptual y el rendimiento del codec se exponen en el siguiente capítulo.

[3] www.bodden.de

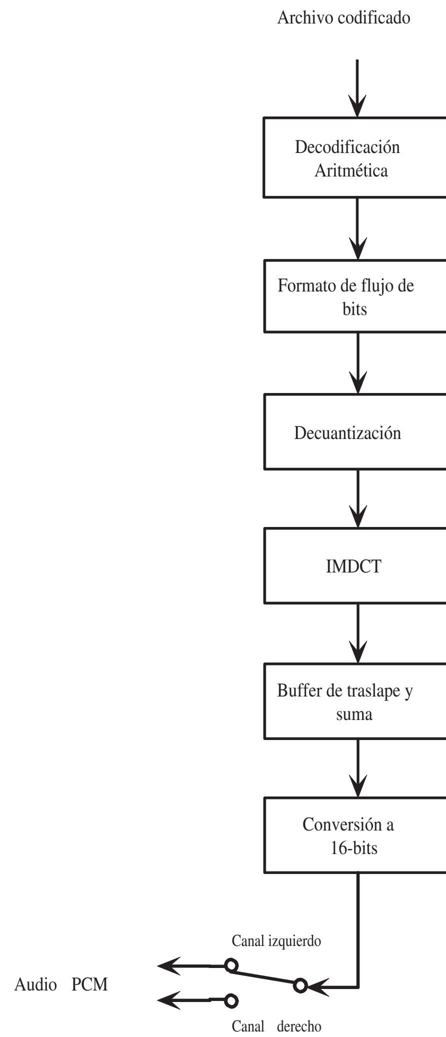


Figura 4.5: Diagrama del proceso de decodificación

Capítulo 5

Resultados

"La teoría es asesinada tarde o temprano por la experiencia".
Albert Einstein.

Experimentalmente se observaron tasas promedio que varían entre 58kbps y 340kbps, que significa una relación de compresión variable entre 24:1 y 4:1. También se observó que la tasa más común se localizaba por arriba de 240kbps (5.8:1), por este motivo se decidió comparar el codec contra los estandares operando a 256kbps.

5.1. Rendimiento

Un aspecto fundamental de todo codec es su rendimiento, es deseable que el tiempo de codificación y de decodificación se mantenga tan pequeño como sea posible. En algunos casos no es fundamental el rendimiento, pues los datos a ser comprimidos o descomprimidos no son requeridos en tiempo real. En audio, este escenario es raro al menos para el caso de la decodificación, pues es típico que el usuario desee oír el archivo de sonido mientras éste es decodificado. Las interrupciones que provoca un proceso de decodificación lento son molestas y pueden ser un factor decisivo para que un usuario opte por cambiar de formato.

Para medir el rendimiento del codec, se utilizó un archivo de 60 segundos en una Pentium 4 a 2.4GHz bajo Windows XP. Los resultados se presentan a continuación:

- Codificación: El codec es capaz de codificar aproximadamente 2.5 segundos de audio (44100Hz, 16 bits, estéreo) por segundo, en otras palabras un archivo de 60 segundos

es codificado en 24 segundos. Por lo que es posible hacer codificación en tiempo real.

- Decodificación: El proceso de decodificación es mucho más sencillo y el codec es capaz de decodificar hasta 12 segundos de audio cada segundo, el mismo archivo de 60 segundos fue decodificado en tan sólo 5 segundos.

Codecs de formatos de gran aceptación cuentan con implementaciones optimizadas que hacen uso de instrucciones específicas de procesador, tales como MMX, SSE o SSE2[1].

El codec implementado no está optimizado de manera específica para utilizar estas instrucciones[2]. Sin embargo, provee lo necesario para efectuar codificación y decodificación a una velocidad mayor que la requerida para procesamiento en tiempo real.

5.2. Pruebas de calidad perceptual

Los 3 codecss a comparar son:

1. legg: El codec desarrollado, la tasa promedio varía según la prueba y es expresada como parte del resultado.
2. MP3: codificado utilizando LAME[3] 3.92 en modo CBR a 256kbps.
3. AAC: codificado utilizando Nero Digital Audio[4] LC-AAC a 256kbps.

La metodología trató de apegarse en la medida que fuera posible a la utilizada en el estándar ITU-R BS.1116 [ITU-R97]. El método de prueba "doble ciega, estímulo triple con referencia escondida" consiste en presentar al sujeto 3 señales (estímulos): la señal de referencia R y las señales de prueba A y B. Una de las señales de prueba es nuevamente la señal de referencia y la otra es la señal codificada. Se denomina "doble ciega" porque el operador y el sujeto de prueba desconocen la correspondencia de las señales de prueba A y B.

[1] Intel Corporation.
[2] Con excepción de FFTW, que sí hace uso de estas instrucciones.
[3] LAME Ain't an MP3 Encoder http://lame.sourceforge.net/
[4] http://www.ahead.de/

Al sujeto se le permite repetir y cambiar entre R, A y B. Posteriormente, se le pide que califique las diferencias entre A y R, y B y R de acuerdo con la escala de calificaciones de la tabla 5.1, puesto que una de las señales de prueba es la misma señal de referencia, una de las dos deberá ser calificada con cinco, mientras que la otra describirá la calidad subjetiva percibida por el sujeto.

5.0	Imperceptible
4.0	Perceptible pero no molesto
3.0	Ligeramente molesto
2.0	Molesto
1.0	Muy molesto

Tabla 5.1: Escala de 5 grados de la ITU-R para medir diferencias entre una señal de referencia y una procesada

Se utilizaron 16 muestras de sonido, 9 de las cuales son ampliamente utilizadas para probar diferencias en los esquemas de compresión (obtenidas de PCABX [5]): castanets, finger snaps, french horns, timpani, triangle, trumpets 1, trumpets 2, violins 1 y violins 2. Las pruebas classic, disco, new age, pop, pop español, ranchera y rock fueron seleccionadas de distintos materiales por comprender diferentes géneros de música.

Las pruebas se efectuaron a 15 personas y los resultados se obtuvieron mediante un análisis de HSD de Tukey (*Tukey's Honestly Significant Difference* [Tukey53]), con intervalo de un 95 % de confianza. En la figura 5.1 se muestra la distribución de las edades de los sujetos de prueba.

A pesar de que a los sujetos a los que se les aplicaron las pruebas se les dio un corto entrenamiento, en su mayoría no son expertos, y por ende les era difícil explicar las diferencias que notaban en las pruebas. Razón por la cual, la mayoría de los artefactos documentados en las pruebas, corresponden a los detectados por el autor y un usuario experto de la comunidad hydrogenaudio. [6]

A continuación se exponen los resultados de las 16 pruebas. El eje Y de las gráficas corresponde a la escala de la tabla 5.1, en el eje X se encuentra cada uno de los codecs a comparar. El intervalo de confianza es ilustrado por una caja con un número a la derecha

[5]PC ABX http://www.pcabx.com/

[6]hydrogenaudio: the audio technology enthusiast's resource http://www.hydrogenaudio.org/

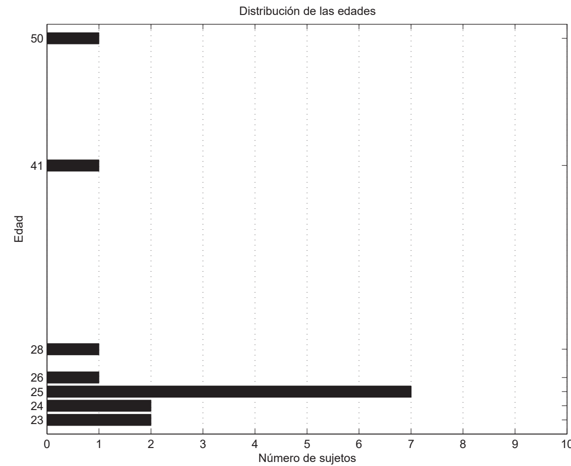


Figura 5.1: Distribución de las edades de los sujetos

que indica la media. Si algún intervalo tiene traslape con otro, indica que ambos codecs están empatados.

5.2.1. Castañuelas - Castanets

El artefacto más predominante que presentó nuestro codec en esta prueba es el de pre-eco. El codec mostró debilidad pues no implementa cambios en el tamaño de bloque para así garantizar que el ruido de cuantización sea enmascarado por el pre-enmascaramiento.

El resultado de esta prueba (figura 5.2) es: los tres codecs empatados.

5.2.2. Música Clásica

- Título: Symphony No. 5 in C minor, op.67 IV Allegro
- Compositor: Ludwig Van Beethoven
- Interprete: Filarmónica de Viena
- Director: Hans Schmidt-Isserstedt
- Año: 1965-1969

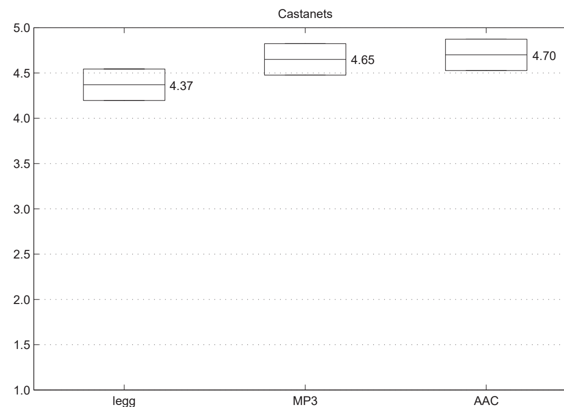


Figura 5.2: Castañuelas, tasa promedio: 244.48kbps. HSD de Tukey=0.347

Las cuerdas están un poco distorsionadas. También se observó un fenómeno que sucede cuando el modelo psicoacústico elimina componentes de alta frecuencia y al poco tiempo los deja presentes, esta intermitencia de altas frecuencias es conocida como *ringing*, las posibles soluciones son aplicar un filtro pasa-bajas a la señal o mejorar el modelo psicoacústico.

El resultado de esta prueba (figura 5.3) es: los tres codecs empatados.

5.2.3. Música Disco

- Título: Uh La La La
- Interprete: Alexia
- Album: Eurodisco '98

El resultado de esta prueba (figura 5.4) es: los tres codecs empatados.

5.2.4. Chasquidos de Dedos - Finger Snaps

El resultado de esta prueba (figura 5.5) es: los tres codecs empatados. Note como el codec logró empatar con los otros aún con una tasa promedio de 125.92kbps.

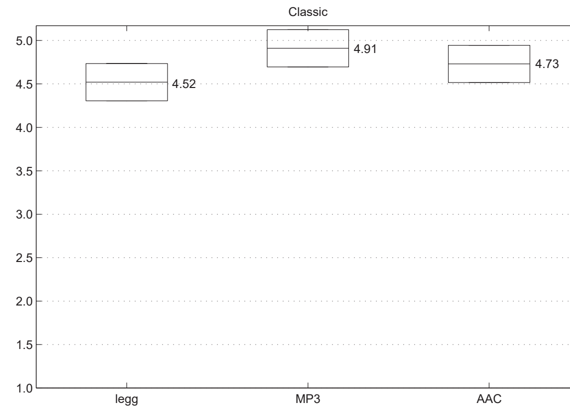


Figura 5.3: Música Clásica, tasa promedio: 261.22kbps. HSD de Tukey=0.429

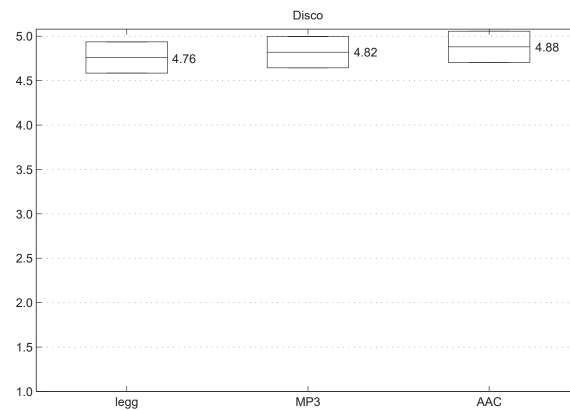


Figura 5.4: Música Disco, tasa promedio: 288.95kbps. HSD de Tukey=0.352

5.2.5. Cornos Franceses - French Horns

Esta prueba (figura 5.6) impactó negativamente al codec con un efecto conocido como gárgaras (*warbling*) que afectó principalmente a las frecuencias bajas, para resolver este problema es necesario mejorar el modelo psicoacústico. El resultado de esta prueba (figura 5.6) es: los tres codecs empatados.

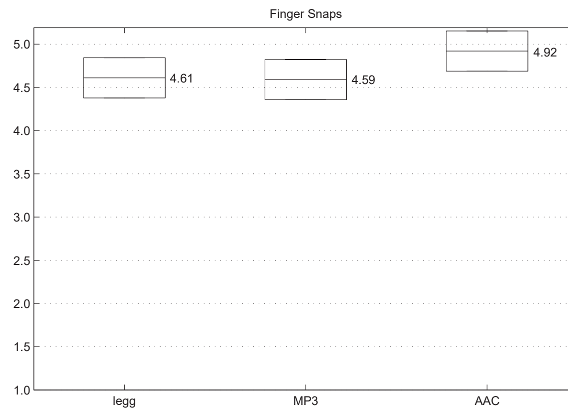


Figura 5.5: Chasquidos de Dedos, tasa promedio: 125.92kbps. HSD de Tukey=0.465

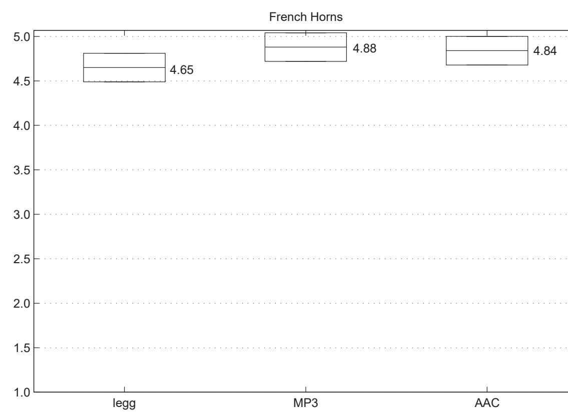


Figura 5.6: Cornos Franceses, tasa promedio: 257.25kbps. HSD de Tukey=0.322

5.2.6. Música New Age

- Título: Rachel's Song
- Interprete: Vangelis
- Album: Reprise 1990-1999

Un poco de distorsión y un problema con componentes altamente tonales. El resultado de esta prueba (figura 5.7) es: los tres codecs empatados.

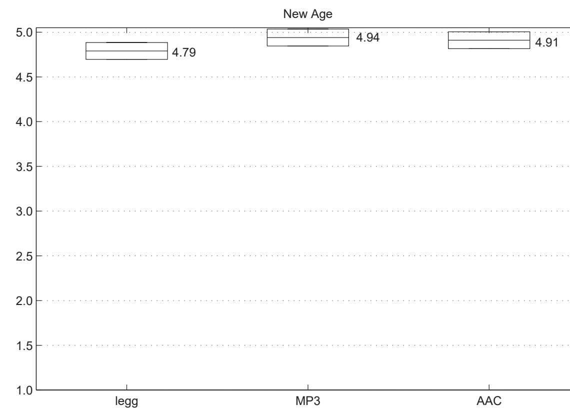


Figura 5.7: Música New Age, tasa promedio: 251.32kbps. HSD de Tukey=0.189

5.2.7. Música Pop

- Título: Jam
- Interprete: Michael Jackson
- Album: Dangerous

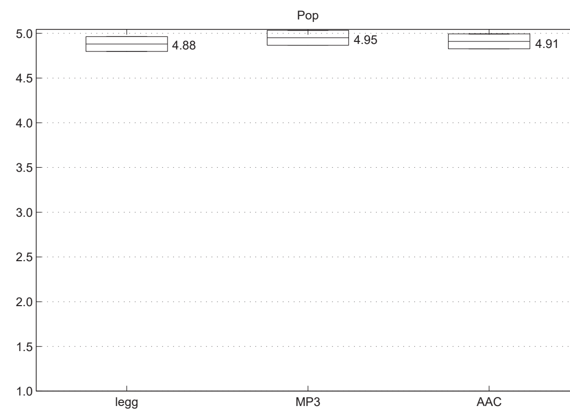


Figura 5.8: Música Pop, tasa promedio: 290.61kbps. HSD de Tukey=0.166

Se introdujo un poco de distorsión no clasificada. El resultado de esta prueba (figura 5.8) es: los tres codecs empatados.

5.2.8. Música Pop en Español

- Título: Hijo de la Luna
- Interprete: Mecano
- Album: Ana, José, Nacho

El problema que se presentó en esta prueba es pre-eco. El resultado de esta prueba (figura 5.9) es: AAC en primer lugar, MP3 y el codec implementado en segundo.

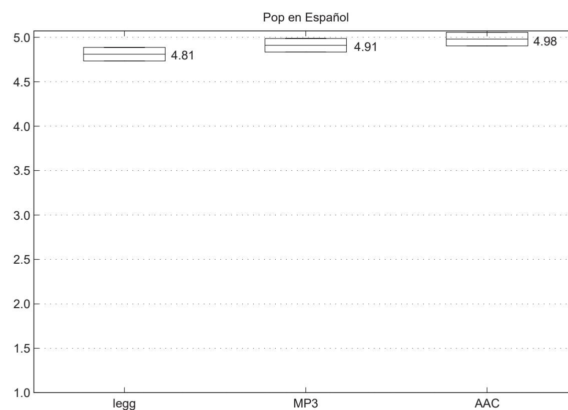


Figura 5.9: Música Pop en Español, tasa promedio: 238.08kbps. HSD de Tukey=0.153

5.2.9. Música Ranchera

- Título: Volver Volver
- Interprete: Vicente Fernández
- Album: Personalidad 20 Éxitos

El resultado de esta prueba (figura 5.10) es: los tres codecs empatados.

5.2.10. Música Rock

- Título: Estranged

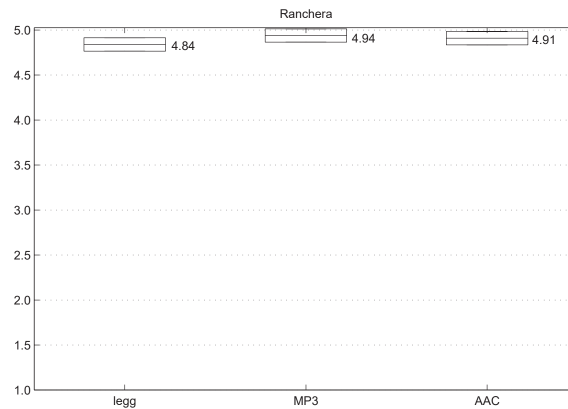


Figura 5.10: Música Ranchera, tasa promedio: 281.91kbps. HSD de Tukey=0.148

- Interprete: Guns N' Roses
- Album: Use Your Illusion II

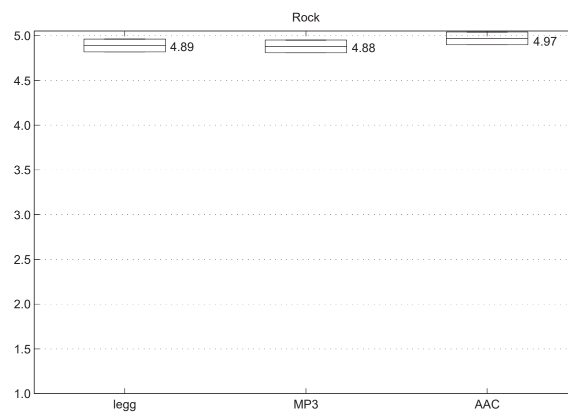


Figura 5.11: Música Rock, tasa promedio: 312.12kbps. HSD de Tukey=0.143

El resultado de esta prueba (figura 5.11) es: los tres codecs empatados.

5.2.11. Tímpanos - Timpani

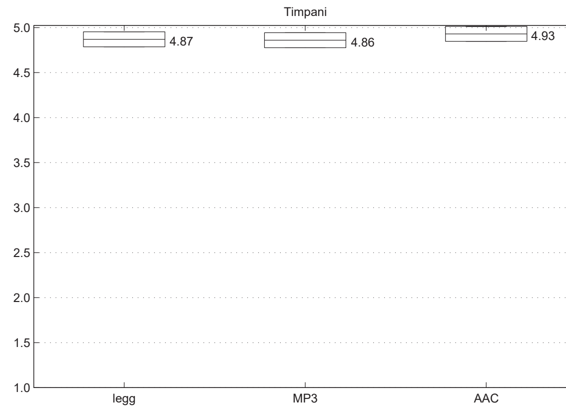


Figura 5.12: Tímpanos, tasa promedio: 216.82kbps. HSD de Tukey=0.166

Cuando el sonido se desvanece, se observa como el silencio es distorsionado, esto se conoce como *noise boost*, en general este fenómeno no es muy molesto y no estará presente a menos que el sonido tenga silencios intermitentes. El resultado de esta prueba (figura 5.12) es: los tres codecs empatados.

5.2.12. Triángulo - Triangle

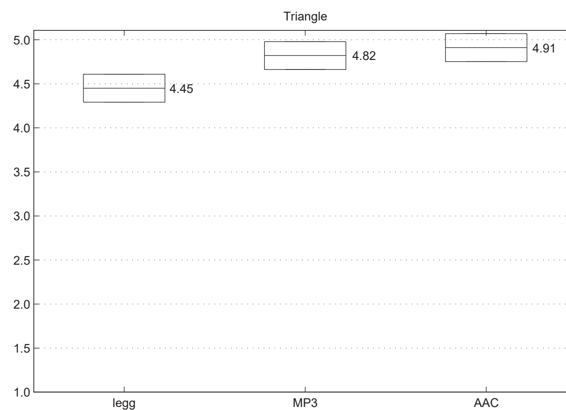
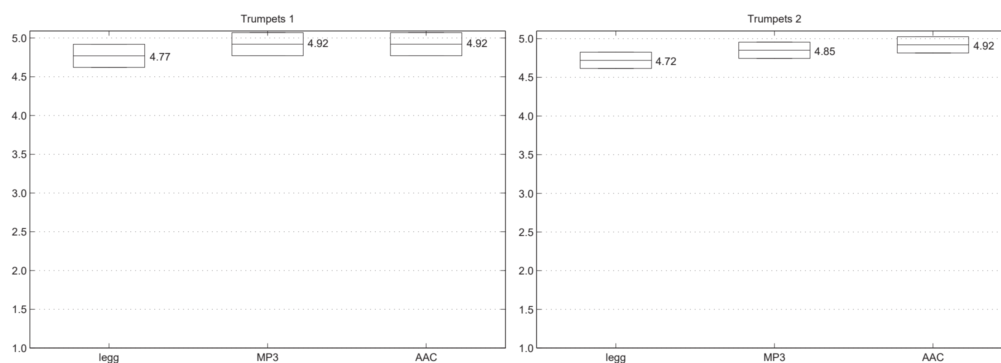


Figura 5.13: Triángulo, tasa promedio: 149.44kbps. HSD de Tukey=0.317

El codec presenta algunos problemas con componentes altamente tonales, en la figura 5.13 se revela esta debilidad. Cuando se toma un bloque de datos y se aplica una ventana, la energía de un tono puro se extiende a coeficientes cercanos, el modelo psicoacústico puede impactar negativamente estos coeficientes o incluso llevarlos a un valor de cero, lo cual deforma la señal drásticamente al momento de síntesis. El resultado de esta prueba (figura 5.13): AAC y MP3 en primer lugar, el codec implementado en segundo. Sin embargo, el codec generó una tasa promedio muy inferior, de 149.44kbps.

5.2.13. Trompetas - Trumpets



(a) ***Trompetas 1, tasa promedio: 259.68kbps. HSD de Tukey=0.298***

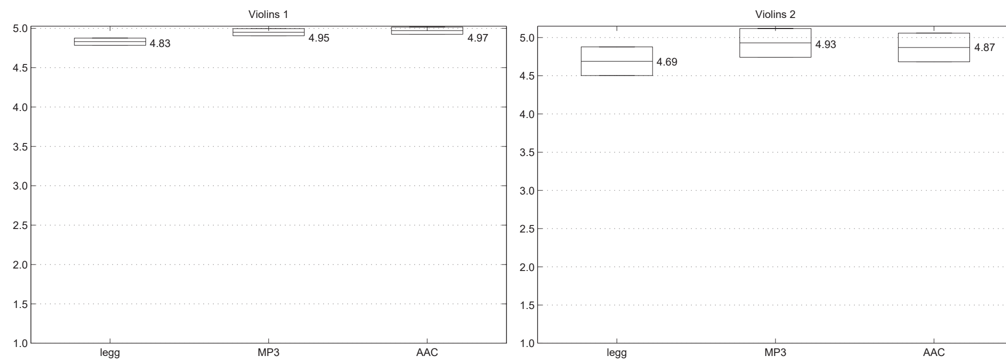
(b) ***Trompetas 2, tasa promedio: 244.81kbps. HSD de Tukey=0.210***

Figura 5.14: Trompetas.

La prueba de la figura 5.14(a) fue afectada por gárgaras. El resultado de ambas pruebas (figuras 5.14(a) y 5.14(b)) es: los tres codecs empatados.

5.2.14. Violines - Violins

En ambas pruebas de violines se observó distorsión en las cuerdas, el violin es un instrumento muy complejo en el dominio de la frecuencia, es necesario mejorar el modelo psicoacústico. El resultado de la prueba "Violines 1"(figura 5.15(a)) es: AAC y MP3 en primer lugar, el codec implementado en segundo. El resultado de "Violines 2"(figura 5.15(b))



(a) ***Violines 1, tasa promedio: 265.22kbps. HSD de Tukey=0.091***

(b) ***Violines 2, tasa promedio: 256.77kbps. HSD de Tukey=0.376***

Figura 5.15: Violines

es: los tres codecs empatados.

5.2.15. Resultados Finales

En la figura 5.16 se muestra el resultado final, calculado utilizando los resultados de todas las pruebas y con un intervalo de confianza de 99 %. Los estándares AAC y MP3 están empatados por el primer lugar y el codec implementado ocupa el segundo lugar.

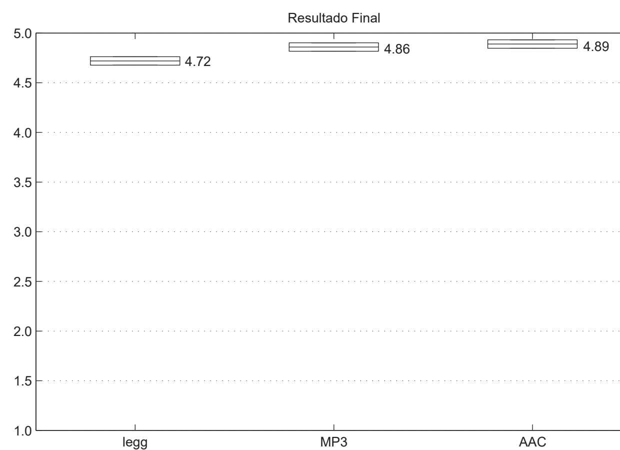


Figura 5.16: Resultados Finales. HSD de Tukey=0.085

Capítulo 6

Conclusiones

A pesar de que el codec no logró empatar con AAC y MP3, su media de 4.72 lo ubica en una muy buena posición que cumple con el objetivo de un perfil de alta calidad. Las pruebas de rendimiento revelan que también es posible realizar compresión y descompresión en tiempo real. Además la estructura de archivo fue diseñada para que una vez que el decodificador tenga la cabecera de archivo, le sea posible decodificar cualquier conjunto de bloques sucesivos sin necesidad de que el total de bloques esté presente.

6.1. Conclusiones Generales

De acuerdo con las pruebas de sonido, los artefactos más comunes que introduce nuestro codec son (ordenados por ocurrencia):

1. Pre-eco: no se cuenta con un mecanismo para reducir este efecto, pues el tamaño de bloque es constante (512 muestras en el dominio del tiempo), se puede apreciar como este fenómeno afectó el resultado de la prueba de castañuelas (figura 5.2).

2. El ruido se hace perceptible (*noise boost*) cuando el volumen disminuye rápidamente hasta el silencio.

3. Gárgaras (*warbling*).

4. Ringing.

El codec propuesto tiene todavía grandes posibilidades de crecimiento, en la siguiente sección se enumeran algunas de las mejoras que se le pueden hacer.

6.2. Trabajos Futuros

1. Implementar el cambio dinámico de tamaño de bloque para lidiar con el pre-eco, se recomienda utilizar un bloque de 2048 muestras para estado estable y 128 muestras para transitorios; este comportamiento es similar al usado en AAC [MPEG97].

2. Explotar la alta correlación que puede haber entre los canales. Por ejemplo, se cuantizan y codifican a y b definidos como $a = L - R$ y $b = L + R$, siendo L y R el canal izquierdo y derecho. Para obtener L y R en el proceso de decodificación se efectúa: $L = (a + b)/2$ y $R = (b - a)/2$. Existen técnicas mucho más avanzadas de agrupación estéreo (*stereo coupling*), como el joint-stereo utilizado en el MP3 [MPEG93] y codificación de intensidad de estéreo usada en AAC [MPEG97].

3. Buscar un modelo de codificación aritmética más apropiado para audio, pues el actual comprime los coeficientes ya cuantizados con un modelo que empieza en una distribución uniforme. Incluso, es posible que un modelo de orden mayor a 0 sea una buena opción.

4. Comparar la ganancia en compresión que ofrece la codificación aritmética contra un conjunto de códigos Huffman especializados por banda. Esto con el fin de mejorar el proceso de codificación sin pérdida y reducir la tasa sin alterar la calidad percibida.

5. Crear perfiles de calidad: en este trabajo se maneja un sólo perfil que pretende la más alta calidad posible bajo tasas razonables (menores que 350kbps). Para ciertas aplicaciones es deseable que la tasa de datos sea más baja aunque la calidad no sea muy alta, por ejemplo radio por Internet que comúnmente maneja tasas de 64kbps y 128kbps.

6. Ampliar las frecuencias de muestreo soportadas, actualmente sólo es capaz de procesar archivos WAV muestreados a 44100Hz. En particular sería conveniente agregar soporte

para 32000Hz, 48000Hz, 96000Hz y 192000Hz.

7. Incluir soporte para multicanal (en particular 5.1, 6.1 y 7.1), actualmente sólo se aceptan archivos en estéreo.

8. Agregar algún mecanismo de resistencia y corrección de errores al formato de archivo. El formato actual no provee un sistema para este fin.

9. Desarrollar una extensión del formato de archivo para el manejo de derechos de autor. Esto tiene como aplicación directa evitar copias ilegales del material codificado.

10. Marcas de agua. En particular, se puede aprovechar el umbral global de enmascaramiento para almacenar la marca sin que ésta sea perceptible.

11. Implementar un contenedor para que el flujo de bits pueda ser transmitido de forma segura y así evitar que el contenido sea interceptado y decodificado mientras se transmite.

Apéndice A

Código Fuente

El código fuente en C, fue compilado con Visual C++ 5.0 en un arquitectura x86.

A.1. main.c

Este archivo contiene las funciones principales que llaman a los procesos de codificación y decodificación.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include <time.h>

#include "wav.h"
#include "dsp.h"
#include "psychoacoustics.h"
#include "encode.h"
#include "types.h"
#include "huffman.h"
#include "arithmeticcoder.h"

/*#define COMPUTEGRAPHICS */


#define CLEANARRAY(x,N) {int macrok; for(macrok=0;macrok<N;macrok++) x[macrok]=0x0;}

#define BUFFERSIZE 32768
struct channel
{
    double *buffer;
    int currentpos;
    int extraindex;
};
typedef struct channel CHANNEL;

void encode(WAVFILE *wav,FILE *fout);
void initchannels(WAVFILE *wav,CHANNEL *left,CHANNEL *right);
void getblock(WAVFILE *wav,CHANNEL *left,CHANNEL *right,double *block,int channel,int blocksize,int overlap);
void printinfo(double *fftr,double *ffti,double *X,double *mask,int Nfreq,BYTE *quantizedr,UWORD length);

void decode(const char *strcuant,const char *strsalida);
void addblock(WAVFILE *wav,CHANNEL *left,CHANNEL *right,double *block,int channel,int blocksize,int overlap);
void flushbuffers(WAVFILE *wav,CHANNEL *left,CHANNEL *right);
```

```
int main(int numargs,char *args[])
{
    time_t start,end;
    WAVFILE wav;
    FILE *fout;

    if(numargs!=4)
    {
        printf("Syntax:\n %s input.wav output.legg output.wav\n",args[0]);
        return -1;
    }

    wav.fwav=fopen(args[1],"rb");
    time(&start);

    switch(readinfo(&wav))
    {
        case 0:
            printf("Sampling rate: %d\nChannels: %d\nBits per sample: %d\nSamples: %d, Length %fs\n\n",wav.samplerate,wav.channels,
                wav.bitspersample,wav.totalsamples,(float)(wav.totalsamples)/wav.samplerate);
            break;
        case -1:
            printf("%s is not a WAV file\n",args[1]);
            return -1;
        case -2:
            printf("%s is not a PCM file\n");
            return -1;
        case -3:
            printf("Sampling rate: %d\nChannels: %d\nBits per sample: %d\nSamples: %d\n",wav.samplerate,wav.channels,
                wav.bitspersample,wav.totalsamples);
            printf("Support limited to 44100Hz, 16 bits, Stereo\n");
            return -1;
        default:
            printf("Unknown error\n");
            return -1;
    }
    if(wav.totalsamples<BUFFERSIZE)
    {
        printf("File is too small, at least %d samples are required\n",BUFFERSIZE);
        return -1;
    }

    fout=fopen(args[2],"wb+");
    encode(&wav,fout);
    fclose(wav.fwav);
    fclose(fout);
    decode(args[2],args[3]);

    time(&end);
    printf("\n\nProcessing time: %d seconds\n",end-start);

    return 0;
}

void encode(WAVFILE *wav,FILE *fout)
{
    int i;
    int blocksize=512,Nfreq=blocksize/2;
    int maxfreq=wav->samplerate/2;
    int allocl,allocr;
    int numblocks;
    double *block=(double *)malloc(sizeof(double)*blocksize);
    double *X=(double *)malloc(sizeof(double)*Nfreq);
    double *fftr=(double *)malloc(sizeof(double)*Nfreq);
    double *ffti=(double *)malloc(sizeof(double)*Nfreq);
    double *mask=(double *)malloc(sizeof(double)*Nfreq);
    double *Xred=(double *)malloc(sizeof(double)*Nfreq);
    BYTE *quantizedl,*quantizedr;
    BYTE *codedl,*codedr;
    int lengthql,lengthqr,maxlength;
    double *window=(double *)malloc(sizeof(double)*blocksize);
    double *hanning=(double *)malloc(sizeof(double)*blocksize);
    struct hdrlegg hdr;
    struct hdrpckleggsize pckhdrsize;
    CHANNEL left,right;

    initchannels(wav,&left,&right);


    hdr.N=wav->totalsamples;
    hdr.mdctsize=Nfreq;
    hdr.sr=wav->samplerate;
    hdr.window=KBDW;
    hdr.channels=2;
    fwrite(&hdr,sizeof(struct hdrlegg),1,fout);
```

```
    hanningwindow(hanning,blocksize);
    switch(hdr.window)
    {
        case SINEW:
            sinewindow(window,blocksize);
            break;
        case KBDW:
            kbdwindow(window,blocksize,4.0);
            break;
        case VORBISW:
            vorbiswindow(window,blocksize);
            break;
        default:
            rectangularwindow(window,blocksize);
            break;
    }

    numblocks=(int)ceil(2.0*wav->totalsamples/blocksize)+1;

    /* Worst case scenario: each transform is quantized with a 32-bit quantizer */
    maxlength=sizeof(DWORD*)*Nfreq;
    quantizedl=(BYTE *)malloc(maxlength);
    quantizedr=(BYTE *)malloc(maxlength);
    codedr=(BYTE *)malloc(maxlength);
    codedl=(BYTE *)malloc(maxlength);

    initpsychoacoustic(Nfreq,maxfreq);
    for(i=0,allocl=0,allocr=0;i<numblocks;i++)
    {
        getblock(wav,&left,&right,block,0,blocksize,blocksize/2);
        mdctblock(block,window,X,blocksize);
        fftblock(block,hanning,fftr,ffti,blocksize);

        computemask(fftr,ffti,Nfreq,maxfreq,mask);
        CLEANARRAY(quantizedl,maxlength);
        lengthql=quantizesignal(X,fftr,ffti,mask,Nfreq,quantizedl);

#ifdef COMPUTEGRAPHICS
        if(i==1)
            printinfo(fftr,ffti,X,mask,Nfreq,&quantizedl[sizeof(struct hdrpckleggsize)],(UWORD)lengthql);
#endif
        lengthql=arithmeticencode(&quantizedl[sizeof(struct hdrpckleggsize)],codedl,lengthql);
        allocl+=lengthql;

        getblock(wav,&left,&right,block,1,blocksize,blocksize/2);
        mdctblock(block,window,X,blocksize);
        fftblock(block,hanning,fftr,ffti,blocksize);
        computemask(fftr,ffti,Nfreq,maxfreq,mask);
        CLEANARRAY(quantizedr,maxlength);
        lengthqr=quantizesignal(X,fftr,ffti,mask,Nfreq,quantizedr);
        lengthqr=arithmeticencode(&quantizedr[sizeof(struct hdrpckleggsize)],codedr,lengthqr);
        allocr+=lengthqr;

        /* Writes encoded left channel */
        pckhdrsize.lengthofpacket=lengthql;
        fwrite(&pckhdrsize,sizeof(struct hdrpckleggsize),1,fout);
        fwrite(codedl,sizeof(BYTE),lengthql,fout);

        /* Writes encoded right channel */
        pckhdrsize.lengthofpacket=lengthqr;
        fwrite(&pckhdrsize,sizeof(struct hdrpckleggsize),1,fout);
        fwrite(codedr,sizeof(BYTE),lengthqr,fout);

    }
    freepsychoacoustic();
    free(left.buffer);
    free(right.buffer);


    printf("Left channel, %d coefficients, %d bytes used\n",numblocks*blocksize/2,allocl);
    printf("Right channel, %d coefficients, %d bytes used\n",numblocks*blocksize/2,allocr);
    printf("File size: %d bytes %.4fkbps\n",allocl+allocr+sizeof(struct hdrlegg),(float)(allocl+allocr+sizeof(struct hdrlegg))*8.0/
        (1000.0*(float)wav->totalsamples/wav->samplerate));
    printf("End of Analysis, %d blocks\n\n",numblocks);

    free(quantizedl);
    free(quantizedr);
    free(X);
    free(fftr);
    free(ffti);
    free(mask);
    free(Xred);
    free(window);
    free(hanning);
}
```

```
void initchannels(WAVFILE *wav,CHANNEL *left,CHANNEL *right)
{
    int i;
    struct sample ssample;

    left->currentpos=0;
    right->currentpos=0;
    left->buffer=(double *)malloc(sizeof(double)*BUFFERSIZE);
    right->buffer=(double *)malloc(sizeof(double)*BUFFERSIZE);

    i=0;
    while(i<BUFFERSIZE && !readsample(wav,&ssample))
    {
        left->buffer[i]=ssample.wLeft/32768.0;
        right->buffer[i]=ssample.wRight/32768.0;
        i++;
    }
}

void getblock(WAVFILE *wav,CHANNEL *left,CHANNEL *right,double *block,int channel,int blocksize,int overlap)
{
    CHANNEL *chaux;
    struct sample ssample;
    int pos,i;

    switch(channel)
    {
        case 0:
            chaux=left;
            break;
        case 1:
            chaux=right;
            break;
    }
    /* Fill with 0's to the left */
    if(chaux->currentpos==0)
    {
        for(i=0;i<overlap;i++)
            block[i]=0.0;
        memcpy(&block[i],chaux->buffer,sizeof(double)*(blocksize-overlap));
        chaux->currentpos+=blocksize-overlap;
    }
    else if(chaux->currentpos-overlap+blocksize<BUFFERSIZE)
    { /* Data is in buffer */
        memcpy(block,&(chaux->buffer[chaux->currentpos-overlap]),sizeof(double)*blocksize);
        chaux->currentpos+=blocksize-overlap;
    }
    else
    { /* Data is not in buffer, get from file */
        /* Save in buffer at least 1024 samples before currentpos*/
        /* currentpos of both channels should be the same */
        pos=left->currentpos;
        i=BUFFERSIZE-(pos-1024);
        memmove(left->buffer,&(left->buffer[pos-1024]),sizeof(double)*i);
        memmove(right->buffer,&(right->buffer[pos-1024]),sizeof(double)*i);
        left->currentpos=1024;
        right->currentpos=1024;

        while(i<BUFFERSIZE && !readsample(wav,&ssample))
        {
            left->buffer[i]=ssample.wLeft/32768.0;
            right->buffer[i]=ssample.wRight/32768.0;
            i++;
        }
        /* Fill with 0's to the right */
        if(i<BUFFERSIZE)
        {
            for(;i<BUFFERSIZE;i++)
            {
                left->buffer[i]=0.0;
                right->buffer[i]=0.0;
            }
        }
        memcpy(block,&(chaux->buffer[chaux->currentpos-overlap]),sizeof(double)*blocksize);
        chaux->currentpos+=blocksize-overlap;
    }
}


void printinfo(double *fftr,double *ffti,double *mdct,double *mask,int Nfreq,BYTE *quantized,UWORD length)
{
    struct hdrpckleggsize pckhdrsize;
    struct hdrpcklegg pckhdr;
    BYTE bitallocations[25];
    BYTE *buffer;
    double *daux,*daux1;
```

```
    int i;
    FILE *faux;


    daux=(double *)malloc(sizeof(double)*Nfreq);
    daux1=(double *)malloc(sizeof(double)*Nfreq);

    memcpy(&pckhdr,quantized,sizeof(struct hdrpcklegg));
    buffer=&quantized[sizeof(struct hdrpcklegg)];
    pckhdrsize.lengthofpacket=length;

    /* Reads scale factors and mantisas for left channel */
    bitallocations[0]=pckhdr.alloc0;
    bitallocations[1]=pckhdr.alloc1;
    bitallocations[2]=pckhdr.alloc2;
    bitallocations[3]=pckhdr.alloc3;
    bitallocations[4]=pckhdr.alloc4;
    bitallocations[5]=pckhdr.alloc5;
    bitallocations[6]=pckhdr.alloc6;
    bitallocations[7]=pckhdr.alloc7;
    bitallocations[8]=pckhdr.alloc8;
    bitallocations[9]=pckhdr.alloc9;
    bitallocations[10]=pckhdr.alloc10;
    bitallocations[11]=pckhdr.alloc11;
    bitallocations[12]=pckhdr.alloc12;
    bitallocations[13]=pckhdr.alloc13;
    bitallocations[14]=pckhdr.alloc14;
    bitallocations[15]=pckhdr.alloc15;
    bitallocations[16]=pckhdr.alloc16;
    bitallocations[17]=pckhdr.alloc17;
    bitallocations[18]=pckhdr.alloc18;
    bitallocations[19]=pckhdr.alloc19;
    bitallocations[20]=pckhdr.alloc20;
    bitallocations[21]=pckhdr.alloc21;
    bitallocations[22]=pckhdr.alloc22;
    bitallocations[23]=pckhdr.alloc23;
    bitallocations[24]=pckhdr.alloc24;
    dequantizesignal(bitallocations,buffer,pckhdrsize.lengthofpacket-sizeof(struct hdrpcklegg),daux);

    faux=fopen("graficas\\mdct","wb");
    for(i=0;i<Nfreq;i++)
        fprintf(faux,"%e %e\n",(double)(i*22050)/Nfreq,mdct[i]);
    fclose(faux);
    faux=fopen("graficas\\mdctspl","wb");
    daux1=mdctspl(mdct,Nfreq);
    for(i=0;i<Nfreq;i++)
        fprintf(faux,"%e %e\n",(double)(i*22050)/Nfreq,daux1[i]);
    fclose(faux);
    free(daux1);

    faux=fopen("graficas\\mdctquant","wb");
    for(i=0;i<Nfreq;i++)
        fprintf(faux,"%e %e\n",(double)(i*22050)/Nfreq,daux[i]);
    fclose(faux);
    faux=fopen("graficas\\mdctquantspl","wb");
    daux1=mdctspl(daux,Nfreq);
    for(i=0;i<Nfreq;i++)
        fprintf(faux,"%e %e\n",(double)(i*22050)/Nfreq,daux1[i]);
    fclose(faux);
    free(daux1);

    faux=fopen("graficas\\fftspl","wb");
    daux1=fftspl(fftr,ffti,Nfreq);
    for(i=0;i<Nfreq;i++)
        fprintf(faux,"%e %e\n",(double)(i*22050)/Nfreq,daux1[i]);
    fclose(faux);
    free(daux1);

    faux=fopen("graficas\\mask","wb");
    for(i=0;i<Nfreq;i++)
        fprintf(faux,"%e %e\n",(double)(i*22050)/Nfreq,mask[i]);
    fclose(faux);

    free(daux);
}

void decode(const char *strquant,const char *strout)
{
    FILE *fquant;
    struct hdrlegg hdr;
    struct hdrpcklegg pckhdr;
    struct hdrpckleggsize pckhdrsize;
    double *mdctleft,*mdctright;
    BYTE *buffer;
    BYTE bitallocations[25];
    int maxbufferlength;
```

```
int mdctsize,N;
double *bufferl,*bufferr;
double *window;
int i;
WAVFILE wav;
CHANNEL left,right;
BYTE *codedr,*codedl;

fquant=fopen(strquant,"rb");
wav.fwav=fopen(strout,"wb+");

fread(&hdr,sizeof(struct hdrlegg),1,fquant);

N=hdr.N;
mdctsize=hdr.mdctsize;

printf("File info: %d %d %d %d ",N,mdctsize,hdr.sr,hdr.channels);
wav.samplerate=hdr.sr;
wav.bitspersample=16;
wav.channels=hdr.channels;
wav.totalsamples=hdr.N;
wav.samplesread=0;

left.currentpos=0;
left.extraindex=0;
left.buffer=(double *)malloc(sizeof(double)*BUFFERSIZE);
right.currentpos=0;
right.extraindex=0;
right.buffer=(double *)malloc(sizeof(double)*BUFFERSIZE);

bufferl=(double *)malloc(sizeof(double)*mdctsize*2);
bufferr=(double *)malloc(sizeof(double)*mdctsize*2);
mdctleft=(double *)malloc(sizeof(double)*mdctsize);
mdctright=(double *)malloc(sizeof(double)*mdctsize);
window=(double *)malloc(sizeof(double)*mdctsize*2);

maxbufferlength=sizeof(DWORD)*mdctsize;
buffer=(BYTE *)malloc(maxbufferlength);
codedr=(BYTE *)malloc(maxbufferlength);
codedl=(BYTE *)malloc(maxbufferlength);

if(!bufferl || !bufferr || !mdctleft || !mdctright || !window)
{
    printf("Error, out of memory\n");
    exit(-1);
}

switch(hdr.window)
{
    case SINEW:
        sinewindow(window,mdctsize*2);
        printf("sine ");
        break;
    case KBDW:
        kbdwindow(window,mdctsize*2,4.0);
        printf("kbd ");
        break;
    case VORBISW:
        vorbiswindow(window,mdctsize*2);
        printf("vorbis ");
        break;
    default:
        rectangularwindow(window,mdctsize*2);
        printf("rectangular ");
        break;
}

printf("\n");
writeinfo(&wav);

initpsychoacoustic(mdctsize,hdr.sr/2);
i=0;
while(fread(&pckhdrsize,sizeof(struct hdrpckleggsize),1,fquant))
{
    fread(codedl,sizeof(BYTE),pckhdrsize.lengthofpacket,fquant);
    /* Decodes packet */
    pckhdrsize.lengthofpacket=arithmeticdecode(codedl,buffer,pckhdrsize.lengthofpacket);
    /* Reads scale factors and mantisas for left channel */
    memcpy(&pckhdr,buffer,sizeof(struct hdrpcklegg));
    memmove(buffer,&buffer[sizeof(struct hdrpcklegg)],pckhdrsize.lengthofpacket-sizeof(struct hdrpcklegg));
    bitallocations[0]=pckhdr.alloc0;
    bitallocations[1]=pckhdr.alloc1;
    bitallocations[2]=pckhdr.alloc2;
    bitallocations[3]=pckhdr.alloc3;
    bitallocations[4]=pckhdr.alloc4;
    bitallocations[5]=pckhdr.alloc5;
```

```
        bitallocations[6]=pckhdr.alloc6;
        bitallocations[7]=pckhdr.alloc7;
        bitallocations[8]=pckhdr.alloc8;
        bitallocations[9]=pckhdr.alloc9;
        bitallocations[10]=pckhdr.alloc10;
        bitallocations[11]=pckhdr.alloc11;
        bitallocations[12]=pckhdr.alloc12;
        bitallocations[13]=pckhdr.alloc13;
        bitallocations[14]=pckhdr.alloc14;
        bitallocations[15]=pckhdr.alloc15;
        bitallocations[16]=pckhdr.alloc16;
        bitallocations[17]=pckhdr.alloc17;
        bitallocations[18]=pckhdr.alloc18;
        bitallocations[19]=pckhdr.alloc19;
        bitallocations[20]=pckhdr.alloc20;
        bitallocations[21]=pckhdr.alloc21;
        bitallocations[22]=pckhdr.alloc22;
        bitallocations[23]=pckhdr.alloc23;
        bitallocations[24]=pckhdr.alloc24;
        dequantizesignal(bitallocations,buffer,pckhdrsize.lengthofpacket-sizeof(struct hdrpcklegg),mdctleft);

        /* Reads packet header for right channel */
        fread(&pckhdrsize,sizeof(struct hdrpckleggsize),1,fquant);
        fread(codedr,sizeof(BYTE),pckhdrsize.lengthofpacket,fquant);
        /* Decodes packet */
        pckhdrsize.lengthofpacket=arithmeticdecode(codedr,buffer,pckhdrsize.lengthofpacket);
        /* Reads scale factors and mantisas for left channel */
        memcpy(&pckhdr,buffer,sizeof(struct hdrpcklegg));
        memmove(buffer,&buffer[sizeof(struct hdrpcklegg)],pckhdrsize.lengthofpacket-sizeof(struct hdrpcklegg));
        /* Scale factors and mantisas for right channel */
        bitallocations[0]=pckhdr.alloc0;
        bitallocations[1]=pckhdr.alloc1;
        bitallocations[2]=pckhdr.alloc2;
        bitallocations[3]=pckhdr.alloc3;
        bitallocations[4]=pckhdr.alloc4;
        bitallocations[5]=pckhdr.alloc5;
        bitallocations[6]=pckhdr.alloc6;
        bitallocations[7]=pckhdr.alloc7;
        bitallocations[8]=pckhdr.alloc8;
        bitallocations[9]=pckhdr.alloc9;
        bitallocations[10]=pckhdr.alloc10;
        bitallocations[11]=pckhdr.alloc11;
        bitallocations[12]=pckhdr.alloc12;
        bitallocations[13]=pckhdr.alloc13;
        bitallocations[14]=pckhdr.alloc14;
        bitallocations[15]=pckhdr.alloc15;
        bitallocations[16]=pckhdr.alloc16;
        bitallocations[17]=pckhdr.alloc17;
        bitallocations[18]=pckhdr.alloc18;
        bitallocations[19]=pckhdr.alloc19;
        bitallocations[20]=pckhdr.alloc20;
        bitallocations[21]=pckhdr.alloc21;
        bitallocations[22]=pckhdr.alloc22;
        bitallocations[23]=pckhdr.alloc23;
        bitallocations[24]=pckhdr.alloc24;
        dequantizesignal(bitallocations,buffer,pckhdrsize.lengthofpacket-sizeof(struct hdrpcklegg),mdctright);

        imdctblock(bufferl,window,mdctleft,mdctsize*2);
        imdctblock(bufferr,window,mdctright,mdctsize*2);

        addblock(&wav,&left,&right,bufferl,0,mdctsize*2,mdctsize);
        addblock(&wav,&left,&right,bufferr,1,mdctsize*2,mdctsize);
        i++;
    }
    flushbuffers(&wav,&left,&right);
    freepsychoacoustic();

    printf("End of Synthesis, %d samples\n\n\n",i);
    fclose(wav.fwav);

    free(window);
    free(bufferl);
    free(bufferr);
    free(mdctleft);
    free(mdctright);

    free(left.buffer);
    free(right.buffer);

    fclose(fquant);
}

/*
    addblock: overlaps and adds imdct blocks of the left and right channel,
    when the buffer is full it flushes the buffer contents to wav. Can only
    manage 2 channels.
```

```
  parameters:
    channel: 0 is left, 1 is right
    blocksize: number of samples in the imdct block
    overlap: number of samples that overlap with the previous block
*/

void addblock(WAVFILE *wav,CHANNEL *left,CHANNEL *right,double *block,int channel,int blocksize,int overlap)
{
    CHANNEL *chaux;
    int pos,i,j;

    switch(channel)
    {
        case 0:
            chaux=left;
            break;
        case 1:
            chaux=right;
            break;
    }
    /* Assume 0's to the left */
    if(chaux->currentpos==0)
    {
        for(i=0,j=overlap;j<blocksize;i++,j++)
            chaux->buffer[i]=block[j];
        for(;i<BUFFERSIZE;i++)
            chaux->buffer[i]=0.0;
        chaux->currentpos+=blocksize-overlap;
        chaux->extraindex=0;
    }
    else if(chaux->currentpos-overlap+blocksize<BUFFERSIZE)
    { /* Buffer is not full */
        for(i=chaux->currentpos-overlap,j=0;j<blocksize;i++,j++)
            chaux->buffer[i]+=block[j];
        chaux->extraindex+=overlap;
        chaux->currentpos+=blocksize-overlap;
    }
    else
    { /* Buffer is full, write to file */
        i=0;
        while(i<left->extraindex && i<right->extraindex)
        {
            writesample(wav,left->buffer[i],right->buffer[i]);
            i++;
        }

        pos=i;
        i=left->currentpos-pos;
        left->extraindex-=pos;
        memmove(left->buffer,&(left->buffer[pos]),sizeof(double)*i);
        left->currentpos-=pos;

        i=right->currentpos-pos;
        right->extraindex-=pos;
        memmove(right->buffer,&(right->buffer[pos]),sizeof(double)*i);
        right->currentpos-=pos;

        /* Fill with 0's to the right */
        for(i=left->currentpos;i<BUFFERSIZE;i++)
            left->buffer[i]=0.0;
        for(i=right->currentpos;i<BUFFERSIZE;i++)
            right->buffer[i]=0.0;

        for(i=chaux->currentpos-overlap,j=0;j<blocksize;i++,j++)
            chaux->buffer[i]+=block[j];
        chaux->extraindex+=overlap;
        chaux->currentpos+=blocksize-overlap;
    }
}

void flushbuffers(WAVFILE *wav,CHANNEL *left,CHANNEL *right)
{
    int i;
    if(left->extraindex!=right->extraindex)
        printf("Aguas!!! %d %d \n",left->extraindex,right->extraindex);
    for(i=0;i<left->extraindex;i++)
        writesample(wav,left->buffer[i],right->buffer[i]);
}
```

A.2. Módulo dsp

Los archivos que forman parte del módulo dsp, contienen las funciones de procesamiento digital de señales.

A.2.1. dsp.h

```
#ifndef _DSP_H_
#define _DSP_H_
#include "fftw3.h"

#define PI 3.14159265358979323846264338327950

void fft(double *in,double *outr,double *outi,int N);
void ifft(double *inr,double *ini,double *outr,int N);
void fftblock(double *x,double *window,double *fftr,double *ffti,int blocklength);
double *fftspl(double *xr,double *xi,int Nfreq);

void mdct4(double *in,double *out,int N);
void imdct4(double *in,double *out,int K);

void mdctblock(double *x,double *window,double *result,int blocklength);
void imdctblock(double *buffer,double *window,double *result,int blocklength);
double *mdctspl(double *x,int Nfreq);
static double windowgain;

int verifywindow(double *window,int N);
void sinewindow(double *window,int N);
void kbdwindow(double *window,int N,double alpha);
double BesselI0(double x);
void vorbiswindow(double *window,int N);
void rectangularwindow(double *window,int N);
void hanningwindow(double *window,int N);
double computewindowgain(double *window,int N);

#endif
```

A.2.2. dsp.c

```
#include <memory.h>
#include <stdlib.h>
#include <math.h>
#include "dsp.h"

void fft(double *in,double *outr,double *outi,int N)
{
    int i;
    fftw_complex *out;
    fftw_plan plan;
    double *inp;

    inp=(double *)fftw_malloc(sizeof(double)*N);
    memcpy(inp,in,sizeof(double)*N);
    out=(fftw_complex *)fftw_malloc(sizeof(fftw_complex)*N);
    plan=fftw_plan_dft_r2c_1d(N, inp, out,FFTW_ESTIMATE);
    fftw_execute(plan);
    fftw_destroy_plan(plan);

    for(i=0;i<N/2;i++)
    {
        outr[i]=out[i][0];
        outi[i]=-out[i][1];
    }
    fftw_free(out);
    fftw_free(inp);
}

void ifft(double *inr,double *ini,double *outr,int N)
{
    int i;
    fftw_complex *in;
    fftw_plan plan;

    in=(fftw_complex *)fftw_malloc(sizeof(fftw_complex)*N);
```

```
    for(i=0;i<N;i++)
    {
        in[i][0]=inr[i];
        in[i][1]=ini[i];
    }

    plan=fftw_plan_dft_c2r_1d(N,in,outr,FFTW_ESTIMATE|FFTW_DESTROY_INPUT);
    fftw_execute(plan);
    fftw_destroy_plan(plan);
    fftw_free(in);
}

void fftblock(double *x,double *window,double *fftr,double *ffti,int blocklength)
{
    double *block=(double *)malloc(sizeof(double)*blocklength);
    int i;

    for(i=0;i<blocklength;i++)
            block[i]=x[i]*window[i];

    fft(block,fftr,ffti,blocklength);
    free(block);
}


double *fftspl(double *xr,double *xi,int Nfreq)
{
    double *spl=(double *)malloc(sizeof(double)*Nfreq);
    double fg,mag2;
    int i;
    int N=Nfreq*2;

    fg=4.0/((double)(N*N)*windowgain);
    for(i=0;i<Nfreq;i++)
    {
        mag2=xr[i]*xr[i]+xi[i]*xi[i];
        spl[i]=96.0+10.0*log10(fg*mag2);
        if(spl[i]<0.0 || mag2==0.0)
            spl[i]=0.0;
    }

    return spl;
}

void mdct4(double *in,double *out,int N)
{
    int r,k,r2,k2;
    int N2,N4;
    double a,b,c,d,auxd,theta,*inrot;
    fftw_complex *inc,*outc;
    fftw_plan plan;

    N2=N/2;
    N4=N/4;

    inc=(fftw_complex *)fftw_malloc(sizeof(fftw_complex)*N4);
    outc=(fftw_complex *)fftw_malloc(sizeof(fftw_complex)*N4);
    inrot=(double *)malloc(sizeof(double)*N);
    auxd=PI/N2;

    for(r=0;r<N4;r++)
        inrot[r] = -in[r+3*N4];
    for(;r<N;r++)
        inrot[r] = in[r-N4];

    for(r=0;r<N4;r++)
    {
        theta=auxd*(r+0.125);
        r2=2*r;
        a=0.5*(inrot[r2]-inrot[N-1-r2]);
        b=-0.5*(inrot[N2+r2]-inrot[N2-1-r2]);
        c=cos(theta);
        d=-sin(theta);
        inc[r][0]=a*c-b*d;
        inc[r][1]=a*d+b*c;
    }

    /* FFT N/4 */
    plan=fftw_plan_dft_1d(N4, inc, outc,FFTW_FORWARD,FFTW_ESTIMATE);
    fftw_execute(plan);
    fftw_destroy_plan(plan);

    for(k=0;k<N4;k++)
    {
        theta=auxd*(k+0.125);
        a=outc[k][0];
```

```
        b=outc[k][1];
        c=cos(theta);
        d=-sin(theta);

        k2=2*k;
        out[k2]=2.0*(a*c-b*d);
        out[N2-1-k2]=-2.0*(a*d+b*c);
    }
    fftw_free(inc);
    fftw_free(outc);
    free(inrot);
}

void imdct4(double *in,double *out,int K)
{
    fftw_plan plan;
    int k;
    int N=2*K;
    int K2=K/2,k2;
    double auxd,theta,a,b,c,d,*outrot;

    fftw_complex *inc,*outc;
    inc=(fftw_complex *)fftw_malloc(sizeof(fftw_complex)*K2);
    outc=(fftw_complex *)fftw_malloc(sizeof(fftw_complex)*K2);
    outrot=(double *)malloc(sizeof(double)*N);

    auxd=PI/K;
    for(k=0;k<K2;k++)
    {
        theta=auxd*(k+0.125);
        k2=k*2;
        a=0.5*in[k2];
        b=0.5*in[K-1-k2];
        c=cos(theta);
        d=-sin(theta);
        inc[k][0]=a*c-b*d;
        inc[k][1]=a*d+b*c;
    }

    plan=fftw_plan_dft_1d(K2, inc, outc,FFTW_FORWARD,FFTW_ESTIMATE);
    fftw_execute(plan);
    fftw_destroy_plan(plan);

    for(k=0;k<K2;k++)
    {
        theta=auxd*(k+0.125);
        a=outc[k][0];
        b=outc[k][1];
        c=cos(theta);
        d=-sin(theta);
        outc[k][0]=8.0*(a*c-b*d)/(double)N;
        outc[k][1]=8.0*(a*d+b*c)/(double)N;
    }
    for(k=0;k<K2;k++)
    {
        k2=2*k;
        outrot[k2]=outc[k][0];
        outrot[K+k2]=outc[k][1];
    }
    for(k=1;k<N;k=k+2)
        outrot[k]=-outrot[N-1-k];

    for(k=0;k<3*K2;k++)
        out[k]=outrot[k+K2];
    for(;k<N;k++)
        out[k]=-outrot[k-3*K2];
    fftw_free(inc);
    fftw_free(outc);
    free(outrot);
}

void mdctblock(double *x,double *window,double *result,int blocklength)
{
    double *block=(double *)malloc(sizeof(double)*blocklength);
    int i;

    for(i=0;i<blocklength;i++)
            block[i]=x[i]*window[i];

    mdct4(block,result,blocklength);
    free(block);
}

void imdctblock(double *buffer,double *window,double *block,int blocklength)
{
    double *blocki=(double *)malloc(sizeof(double)*blocklength);
```

```
    int i;

    imdct4(block,blocki,blocklength/2);

    for(i=0;i<blocklength;i++)
        buffer[i]=blocki[i]*window[i];
    free(blocki);
}

double *mdctspl(double *x,int Nfreq)
{
    double *spl=(double *)malloc(sizeof(double)*Nfreq);
    double fg;
    int i;
    int N=Nfreq*2;

    /* window gain is always 0.5 for valid MDCT windows */
    fg=8.0/((double)(N*N)*0.5);
    for(i=0;i<Nfreq;i++)
    {
        spl[i]=96.0+10.0*log10(fg*x[i]*x[i]);
        if(spl[i]<0.0 || x[i]==0.0)
            spl[i]=0.0;
    }

    return spl;
}

void sinewindow(double *window,int N)
{
    int i;
    for(i=0;i<N;i++)
        window[i]=sin(PI*((double)(i)+0.5)/N);
}

/* kbdwindow && BesselI0 are modified functions based on code by
Craig Stuart Sapp <craig@ccrma.stanford.edu>
http://ccrma.stanford.edu/courses/422/projects/kbd/kbdwindow.cpp
*/
void kbdwindow(double *window,int N,double alpha)
{
    double sumvalue = 0.0,aux;
    int i;

    for (i=0; i<N/2; i++)
    {
        aux = 4.0*i/N - 1.0;
        aux*=aux;
        sumvalue += BesselI0(PI * alpha * sqrt(1.0 - aux));
        window[i] = sumvalue;
    }

    /* need to add one more value to the nomalization factor at size/2: */
    aux = 4.0*(N/2)/N - 1.0;
    aux*=aux;
    sumvalue += BesselI0(PI * alpha * sqrt(1.0 - aux));

    /* normalize the window and fill in the righthand side of the window: */
    for (i=0; i<N/2; i++)
    {
        window[i] = sqrt(window[i]/sumvalue);
        window[N-1-i] = window[i];
    }
}
double BesselI0(double x)
{
    double denominator;
    double numerator;
    double z;

    if (x == 0.0)
    {
        return 1.0;
    }
    else
    {
        z = x * x;
        numerator = (z* (z* (z* (z* (z* (z* (z* (z* (z* (z* (z* (z* (z*
                (z* 0.210580722890567e-22  + 0.380715242345326e-19 ) +
                0.479440257548300e-16) + 0.435125971262668e-13 ) +
                0.300931127112960e-10) + 0.160224679395361e-7  ) +
                0.654858370096785e-5)  + 0.202591084143397e-2  ) +
                0.463076284721000e0)   + 0.754337328948189e2   ) +
                0.830792541809429e4)   + 0.571661130563785e6   ) +
                0.216415572361227e8)   + 0.356644482244025e9   ) +
                0.144048298227235e10);
```

```
        denominator = (z*(z*(z-0.307646912682801e4)+
                0.347626332405882e7)-0.144048298227235e10);
    }
    return -numerator/denominator;
}

void vorbiswindow(double *window,int N)
{
    int i;
    double aux;
    for(i=0;i<N;i++)
    {
        aux=sin(PI*((double)(i)+0.5)/N);
        aux*=aux;
        window[i]=sin(PI/2 * aux);
    }
}
void rectangularwindow(double *window,int N)
{
    int i;
    double aux=1.0/sqrt(2.0);
    for(i=0;i<N;i++)
        window[i]=aux;
}

void hanningwindow(double *window,int N)
{
    int i;
    double aux=2.0*PI/N;
    for(i=0;i<N;i++)
        window[i]=0.5*(1.0-cos(aux*(i+0.5)));
    windowgain=3.0/8.0;
}

int verifywindow(double *window,int N)
{
    int i,Ns2=N/2;
    double sum;
    for(i=0;i<Ns2;i++)
    {
        sum=window[i]*window[i] + window[Ns2+i]*window[Ns2+i];
        printf("%d %e\n",i,sum);
    }
    return 1;
}

double computewindowgain(double *window,int N)
{
    int i;
    double windowgain=0.0;
    for(i=0;i<N;i++)
        windowgain+=window[i]*window[i];
    windowgain/=(double)N;
    printf("%e\n",windowgain);
    return windowgain;
}
```

A.3. Módulo encode

El módulo encode contiene funciones necesarias para codificar y decodificar los archivos del formato propuesto.

A.3.1. encode.h

```
#ifndef _ENCODE_H_
#define _ENCODE_H_

#include <stdio.h>
#include "types.h"

#define SINEW 0x0
#define KBDW 0x1
#define VORBISW 0x2
```

```
#define RECTANGULARW 0xf

#define FDOUBLE 0x0
#define F16BITS 0x1
#define F8BITS 0x2
#define PSYCHOACOUSTIC 0x3

#define SCALEFACTORBITS 6
#define SFLOWER -32
#define SFUPPER 31
#define COMPUTESF(scalefactor) pow(2.0,(double)(scalefactor)/4.0)
#define QUANTIZERRANGE(Nbits) (rangeofquant[Nbits])


struct hdrlegg
{
    unsigned int N; /* 32-bit unsigned int */
    unsigned int sr; /* 32-bit unsigned int */
    BYTE channels;
    BYTE window;
    UWORD mdctsize;
};

struct hdrpckleggsize
{
    UWORD lengthofpacket; /* Length of packet, not including this header */
};
struct hdrpcklegg
{
    struct bitalloc
    {
        BYTE alloc0:4; /* Quantizer size for each sub-band */
        BYTE alloc1:4;
        BYTE alloc2:4;
        BYTE alloc3:4;
        BYTE alloc4:4;
        BYTE alloc5:4;
        BYTE alloc6:4;
        BYTE alloc7:4;
        BYTE alloc8:4;
        BYTE alloc9:4;
        BYTE alloc10:4;
        BYTE alloc11:4;
        BYTE alloc12:4;
        BYTE alloc13:4;
        BYTE alloc14:4;
        BYTE alloc15:4;
        BYTE alloc16:4;
        BYTE alloc17:4;
        BYTE alloc18:4;
        BYTE alloc19:4;
        BYTE alloc20:4;
        BYTE alloc21:4;
        BYTE alloc22:4;
        BYTE alloc23:4;
        BYTE alloc24:4;
        BYTE extra1:4;
    };
        BYTE extra2;            /* Not used, yet */
};


UWORD quantizesignal(double *mdct,double *fftr,double *ffti,double *mask,int Nfreq,BYTE *result);
int quantizeband(BYTE *result,double *mdct,int lower,int upper,int bits,WORD scalefactor,int currentbit);
DWORD quantize(double number,WORD scalefactor,int Nbits);
double dequantize(DWORD quantized,WORD scalefactor,int Nbits);
double midrisevalue(DWORD data,int Nbits);
DWORD midrisequantize(double number,int Nbits);
double midtreadvalue(DWORD data,int Nbits);
DWORD midtreadquantize(double number,int Nbits);
int writebits(BYTE *buffer,DWORD data,int numberofbits,int startbit);
int readbits(BYTE *buffer,DWORD *data,int numberofbits,int startbit);

void dequantizesignal(BYTE *bitallocations,BYTE *buffer,int lengthofpacket,double *mdctleft);
int dequantizeband(BYTE *buffer,double *normalizedbands,int lower,int upper,int bits,WORD scalefactor,int startbit);

#endif
```

A.3.2. encode.c

```
#include <stdlib.h>
#include <math.h>
#include <memory.h>
```

```
#include "encode.h"
#include "dsp.h"
#include "wav.h"
#include "psychoacoustics.h"

int rangeofquant[17]={0,1,1,3,7,15,31,63,127,255,511,
                        1023,2047,4095,8191,16383,32767};

/*
    K: Number of spectral samples
    Kp: Number of non-zero spectral samples being passed.
    P: Bit pool available for mantissas (i.e. without scale factors)
    b: band

    return:
        number of bits to allocate to band b

*/
int allocatebitstoband(double *spl,double *mask,int Kp,int P,int b)
{
    int Nb; /* Number of samples in sub-band b */
    double SMRb; /* Signal to Mask Ratio for sub-band b */
    double aux1,aux2;
    double f;

    Nb=getbandsize(b);

    SMRb=signaltomaskratio(spl,mask,b);


    aux1=(double)P/(double)Kp;
    aux2=SMRb-(double)(Nb)*SMRb/(double)Kp;

    aux2=log(10.0)*aux2/(20.0*log(2.0));

    if(aux1+aux2<0.0)
        return 0;

    f=floor(aux1+aux2);
    if(Nb==1)
    {
        if(f<1.0)
            return 0;
        else
            return 1;
    }
    if(aux1+aux2-f>0.7)
        return (int)ceil(aux1+aux2);
    return (int)f;
}

/* Return:
    Length (in bytes) of result */
UWORD quantizesignal(double *mdct,double *fftr,double *ffti,double *mask,int Nfreq,BYTE *result)
{
    int currentbit,lower,upper;
    int i,j;
    double *spl;
    double *mdctscaled;
    double PE=0.0,ln2=log(2.0);
    double max,aux;
    struct hdrpcklegg pckhdr;
    struct hdrpckleggsize pckhdrsize;
    BYTE bitallocations[25];
    WORD scalefactorsind[25];
    int bitsperblock;

    /* Nfreq is the length of the mdct */
    switch(Nfreq)
    {
        case 128:
            bitsperblock=100;
            break;
        case 256:
            bitsperblock=200;
            break;
        case 512:
            bitsperblock=400;
            break;
        case 1024:
            bitsperblock=900;
            break;
        default:
            bitsperblock=150;
            break;
    }
```

```
/* Sounds better to compute bit allocations by using the SPL
of the mdct against the mask :S */
spl=mdctspl(mdct,Nfreq);
mdctscaled=(double *)malloc(sizeof(double)*Nfreq);

/* Copy the coefficients */
for(i=0;i<Nfreq;i++)
    mdctscaled[i]=mdct[i];

/* Compute bit allocation and scale factors for each subband */
for(i=0;i<25;i++)
{
    bitallocations[i]=allocatebitstoband(spl,mask,Nfreq,bitsperblock,i);
    /* 15-bit (including sign) is the maximum allowed quantizer */
    if(bitallocations[i]>15)
        bitallocations[i]=15;
    /* Forbit 1 bit quantizer */
    if(bitallocations[i]==1)
        bitallocations[i]++;

    getbandlimits(i,&lower,&upper);
    for(max=0.0,j=lower;j<=upper;j++)
    {
        aux=fabs(mdctscaled[j]);
        if(aux>max)
            max=aux;
    }
    /* Select appropiate scalefactor according to the bits allocated */
    aux=max/(double)QUANTIZERRANGE(bitallocations[i]);
    for(j=SFUPPER;j>=SFLOWER;j--)
    {
        if(aux>=COMPUTESF(j))
            break;
    }
    if(j<SFLOWER)
        j=SFLOWER;
    if(j>SFUPPER)
        j=SFUPPER;

    scalefactorsind[i]=j;
}

/* currentbit is set so that header can go first, however the header is not fully
computed yet */
currentbit=(sizeof(struct hdrpckleggsize)+sizeof(struct hdrpcklegg))*8;
/* Store scale factors, if there are no bits allocated to a given subband the scale
factor is not stored */
for(j=0;j<25;j++)
{
    if(bitallocations[j]!=0)
    {
        currentbit=writebits(result,(DWORD)(scalefactorsind[j])&((0x1<<SCALEFACTORBITS)-1),SCALEFACTORBITS,currentbit);
    }
}
/* Quantize each sub-band and store in result */
for(i=0;i<25;i++)
{
    getbandlimits(i,&lower,&upper);
    currentbit=quantizeband(result,mdctscaled,lower,upper,bitallocations[i],scalefactorsind[i],currentbit);
}

free(spl);
free(mdctscaled);

if(currentbit%8) /* Wastes a few bits */
    pckhdrsize.lengthofpacket=currentbit/8+1-sizeof(struct hdrpckleggsize);
else
    pckhdrsize.lengthofpacket=currentbit/8-sizeof(struct hdrpckleggsize);

/* Store length of packet and bit allocations at the beginning of result */
pckhdr.alloc0=bitallocations[0];
pckhdr.alloc1=bitallocations[1];
pckhdr.alloc2=bitallocations[2];
pckhdr.alloc3=bitallocations[3];
pckhdr.alloc4=bitallocations[4];
pckhdr.alloc5=bitallocations[5];
pckhdr.alloc6=bitallocations[6];
pckhdr.alloc7=bitallocations[7];
pckhdr.alloc8=bitallocations[8];
pckhdr.alloc9=bitallocations[9];
pckhdr.alloc10=bitallocations[10];
pckhdr.alloc11=bitallocations[11];
pckhdr.alloc12=bitallocations[12];
pckhdr.alloc13=bitallocations[13];
pckhdr.alloc14=bitallocations[14];
```

```
    pckhdr.alloc15=bitallocations[15];
    pckhdr.alloc16=bitallocations[16];
    pckhdr.alloc17=bitallocations[17];
    pckhdr.alloc18=bitallocations[18];
    pckhdr.alloc19=bitallocations[19];
    pckhdr.alloc20=bitallocations[20];
    pckhdr.alloc21=bitallocations[21];
    pckhdr.alloc22=bitallocations[22];
    pckhdr.alloc23=bitallocations[23];
    pckhdr.alloc24=bitallocations[24];
    pckhdr.extra1=0;
    pckhdr.extra2=0;
    memcpy(result,&pckhdrsize,sizeof(struct hdrpckleggsize));
    memcpy(&result[sizeof(struct hdrpckleggsize)],&pckhdr,sizeof(struct hdrpcklegg));

    return pckhdrsize.lengthofpacket;
}

int quantizeband(BYTE *result,double *mdct,int lower,int upper,int bits,WORD scalefactor,int startbit)
{
    int currentbit=startbit;
    int i;
    int bitstogo;
    DWORD data;


    for(i=lower;i<=upper;i++)
    {
        /* Store all coefficients */
        data=quantize(mdct[i],scalefactor,bits);
        bitstogo=bits;
        currentbit=writebits(result,data,bitstogo,currentbit);
    }
    return currentbit;
}

int writebits(BYTE *buffer,DWORD data,int numberofbits,int startbit)
{
    int currentbit=startbit;
    int currentbyte=currentbit/8;
    int currentbitinbyte=currentbit%8;
    int aux;
    int bitstogo=numberofbits;
    int bitindata=0;
    DWORD mask;
    /* Writes the bits to the stream */
    while(bitstogo>0)
    {
        /* Bits left in current byte */
        aux=8-currentbitinbyte;
        /* Mask with size of bits left */
        mask=(0x1<<aux)-1;
        buffer[currentbyte]=buffer[currentbyte] |
            (BYTE)(((data>>bitindata)&mask )<<currentbitinbyte);

        /* Data did not fit in the remaining of current byte */
        if(bitstogo-aux>=0)
        {
            currentbit+=aux;
            currentbyte++;
            currentbitinbyte=0;
            bitstogo-=aux;
            bitindata+=aux;
        }
        else /* Remaining data fit in byte */
        {
            currentbitinbyte+=bitstogo;
            currentbit+=bitstogo;
            bitstogo=0;
        }
    }
    return currentbit;
}

DWORD quantize(double number,WORD scalefactor,int Nbits)
{
    double sf,scaled;
    DWORD quantized,mask;
    double range=QUANTIZERRANGE(Nbits);
    double c,f;

    mask = (0x1<<Nbits)-1;
    sf = COMPUTESF(scalefactor);

    /* scaled is a real within range to be quantized by an Nbit quantizer */
    scaled = number / sf;
```

```
    c = ceil(scaled);
    f = floor(scaled);
    if(scaled>0.5)
    {
        if(scaled>range)
            scaled=range;
        /* Proper round */
        if(scaled-f > 0.5)
        {
            if(c<=range)
                scaled=c;
        }
    }
    else if(scaled<-0.5)
    {
        if(scaled<-range-1)
            scaled=-range-1;
        if(scaled-c<-0.5)
        {
            if(-f<=range+1)
                scaled=f;
        }
    }
    quantized=(DWORD)scaled;
    quantized=quantized & mask;

    return quantized;
}

double dequantize(DWORD quantized,WORD scalefactor,int Nbits)
{
    DWORD mask;
    double number;
    double sf;

    mask = 0x1<<(Nbits-1);
    if(mask & quantized)
    {
        /* Negative number, fill with ones to the left */
        mask = (0x1<<(sizeof(DWORD)*8))-1;
        mask = mask << Nbits;
        quantized = quantized | mask;
    }
    sf = COMPUTESF(scalefactor);

    number = (double)(int)(quantized) * sf;

    return number;
}

double midrisevalue(DWORD data,int Nbits)
{
    double step;
    DWORD sign,Nbits1;
    DWORD bmask;
    double number;

    Nbits1=Nbits-1;
    sign=(0x1<<Nbits1)&data;
    step=2.0/(double)((0x1<<Nbits)-1);
    bmask=(0x1<<Nbits1)-1;
    number=1.0-step*(bmask-(data&bmask));

    if(sign)
        return -number;

    return number;
}

DWORD midrisequantize(double number,int Nbits)
{
    double step;
    DWORD sign,Nbits1;
    DWORD quantized=0x0;
    DWORD aux;

    Nbits1=Nbits-1;
    if(number<0.0)
    {
        sign=(0x1<<Nbits1);
        number=-1.0*number;
    }
    else
        sign=0x0;

    step=1.0/(double)((0x1<<Nbits)-1);
```

```
    aux=(DWORD)(number/step);
    if(aux%2==0)
        aux++;

    aux=aux/2;
    quantized=aux+sign;

    return quantized;
}

double midtreadvalue(DWORD data,int Nbits)
{
    double step;
    DWORD sign,Nbits1;
    DWORD bmask;
    double number;

    Nbits1=Nbits-1;
    sign=(0x1<<Nbits1)&data;
    step=1.0/(double)((0x1<<Nbits1)-1);
    bmask=(0x1<<Nbits1)-1;
    number=step*(data&bmask);

    if(sign)
        return -number;

    return number;
}

/* Will not work with 1 bit quantizer */
DWORD midtreadquantize(double number,int Nbits)
{
    double step;
    DWORD sign,Nbits1;
    DWORD quantized=0x0;
    DWORD aux;

    Nbits1=Nbits-1;
    if(number<0.0)
    {
        sign=(0x1<<Nbits1);
        number=-1.0*number;
    }
    else
        sign=0x0;

    step=0.5/(double)((0x1<<Nbits1)-1);

    aux=(DWORD)(number/step);
    if(aux%2==1)
        aux++;
    aux=aux/2;
    quantized=aux+sign;

    return quantized;
}


void dequantizesignal(BYTE *bitallocations,BYTE *buffer,int lengthofpacket,double *mdct)
{
    WORD scalefactorind[25];
    DWORD dataaux;
    int i;
    int lower,upper;
    int currentbit;

    /* Extracts scale factors from buffer */
    currentbit=0;
    for(i=0;i<25;i++)
    {
        /* Is band i encoded? */
        if(bitallocations[i]!=0)
        {
            /* Copies the scale factor */
            currentbit=readbits(buffer,&dataaux,SCALEFACTORBITS,currentbit);
            //printf("%x ",dataaux);
            if(dataaux & 0x1<<(SCALEFACTORBITS-1))
                dataaux=dataaux | (((0x1<<28)-1)<<SCALEFACTORBITS);
            scalefactorind[i]=dataaux;
        }
        else
        {
            /* Band not encoded, set scale factor to 0.0 */
            scalefactorind[i]=0x0;
        }
```

```
    }

    /* Dequantize each sub-band and store in mdct */
    for(i=0;i<25;i++)
    {
        getbandlimits(i,&lower,&upper);
        currentbit=dequantizeband(buffer,mdct,lower,upper,bitallocations[i],scalefactorind[i],currentbit);
    }

}

int dequantizeband(BYTE *buffer,double *normalizedbands,int lower,int upper,int bits,WORD scalefactor,int startbit)
{
    int currentbit=startbit;
    int currentbyte=currentbit/8;
    int currentbitinbyte=currentbit%8;
    int i;
    DWORD data;


    if(bits==0)
    {
        for(i=lower;i<=upper;i++)
        {
            normalizedbands[i]=0.0;
        }
        return currentbit;
    }
    else
    {   /* Decode all coefficients */
        for(i=lower;i<=upper;i++)
        {
            currentbit=readbits(buffer,&data,bits,currentbit);
            normalizedbands[i]=dequantize(data,scalefactor,bits);
        }
    }
    return currentbit;
}

int readbits(BYTE *buffer,DWORD *data,int numberofbits,int startbit)
{
    int currentbit=startbit;
    int currentbyte=currentbit/8;
    int currentbitinbyte=currentbit%8;
    int aux;
    int bitstogo=numberofbits;
    int bitindata;
    DWORD mask;

    bitindata=0;
    *data=0x0;
    while(bitstogo>0)
    {
        /* Bits left in current byte */
        aux=8-currentbitinbyte;
        /* Mask with size of bits left in byte or bits left in data */
        if(bitstogo<aux)
            aux=bitstogo;
        mask=(0x1<<aux)-1;

        *data = *data | (((buffer[currentbyte]>>currentbitinbyte) & mask)<<bitindata);

        /* Remaining data outside of current byte */
        if(currentbitinbyte+aux>=8)
        {
            currentbit+=aux;
            currentbyte++;
            currentbitinbyte=0;
            bitstogo-=aux;
            bitindata+=aux;
        }
        else /* No remaining data */
        {
            currentbitinbyte+=bitstogo;
            currentbit+=bitstogo;
            bitstogo=0;
        }
    }
    return currentbit;
}
```

A.4. Módulo psychoacoustics

El módulo psychoacoustics contiene funciones del modelo psicoacústico.

A.4.1. psychoacoustics.h

```
#ifndef _PSYCHOACOUSTICS_H_
#define _PSYCHOACOUSTICS_H_

#define UNMASKED 0
#define COMPUTED 1
#define MASKED 70


void initpsychoacoustic(int Nfreq,int samplingrate);
void freepsychoacoustic(void);

void mpegpm2(double spl,int todo,int Nfreq,int start,int end,double delta,double alpha);
void computemask(double *fftr,double *ffti,int Nfreq,int maxfreq,double *result);
int nexttocomputemask(double *spl,short int *barkflagged,int Nbark);
double signaltomaskratio(double *spl,double *mask,int b);
int getbandsize(int b);
void getbandlimits(int b,int *lower,int *upper);
#endif
```

A.4.2. psychoacoustics.c

```
#include "psychoacoustics.h"
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include "dsp.h"

#define ALPHA 1

short int *splflags=NULL;
double *bark=NULL;
double *thresholdofhearing=NULL;
double *globalmask=NULL;
int barklimits[26][2];
short int barkflags[25];


void mpegpm2(double spl,int todo,int Nfreq,int start,int end,double delta,double alpha)
{
    double dz,aux1,aux2,aux3,aux4;
    double adding;
    int i;

    for(i=start;i<end;i++)
    {
        dz=bark[i]-bark[todo];

        aux1=1.05*dz+0.474;
        aux1*=aux1;
        aux2=1.05*dz-0.5;
        aux2*=aux2;
        aux3=aux2-2.0*(1.05*dz-0.5);
        if(aux3>0.0)
            aux3=0.0;
        aux4=15.8111389+7.5*(1.05*dz+0.474)-17.5*sqrt(1.0+aux1)+8.0*aux3+spl-delta;

        if(aux4>globalmask[i])
        {
            /* Mask addition */
            if(aux4-globalmask[i]<3.0)
                adding=6.0;
            else
                adding=0.0;
            globalmask[i]=aux4+adding;
        }
    }
}

void initpsychoacoustic(int Nfreq,int maxfreq)
{
    int i,j;
```

```
    double f;
    if(bark!=NULL)
        free(bark);
    if(splflags!=NULL)
        free(splflags);
    if(thresholdofhearing!=NULL)
        free(thresholdofhearing);
    if(globalmask!=NULL)
        free(globalmask);

    bark=(double *)malloc(sizeof(double)*Nfreq);
    splflags=(short int*)malloc(sizeof(short int)*Nfreq);
    thresholdofhearing=(double *)malloc(sizeof(double)*Nfreq);
    globalmask=(double *)malloc(sizeof(double)*Nfreq);

    /* Frequency to bark mapping and threshold of hearing */
    for(i=0;i<Nfreq;i++)
    {
        f = (double)(i*maxfreq)/Nfreq;
        bark[i] = 13.0*atan(0.76*f/1000.0)+3.5*atan((f/7500.0)*(f/7500.0));

        f/=1000.0;
        thresholdofhearing[i]=3.64*pow(f,-0.8)-6.5*exp(-0.6*(f-3.3)*(f-3.3))+0.001*f*f*f*f;
        thresholdofhearing[i]/=2.0;
        /* Max TOH at 75dB */
        if(thresholdofhearing[i]>75.0)
            thresholdofhearing[i]=75.0;
    }

    /* Calculates bark limits */
    for(i=1,j=0;i<=25 && j<Nfreq;i++)
    {
        barklimits[i-1][0]=j;
        for(;bark[j]<i && j<Nfreq;j++);
        barklimits[i-1][1]=j-1;
    }
    barklimits[25][0]=Nfreq;
    barklimits[25][1]=Nfreq;
}

void freepsychoacoustic(void)
{
    if(bark!=NULL)
        free(bark);
    if(splflags!=NULL)
        free(splflags);
    if(thresholdofhearing!=NULL)
        free(thresholdofhearing);
    if(globalmask!=NULL)
        free(globalmask);

    bark=NULL;
    splflags=NULL;
    thresholdofhearing=NULL;
    globalmask=NULL;
}

void computemask(double *fftr,double *ffti,int Nfreq,int maxfreq,double *mask)
{
    int i,todo;
    double *spl;
    double delta=12.0; /* Small masking */
    int sub=4;
    int add=11;
    int start,end;


    spl=fftspl(fftr,ffti,Nfreq);

    for(i=0;i<Nfreq;i++)
    {
        splflags[i]=UNMASKED;
        globalmask[i]=0.0;
    }
    for(i=0;i<25;i++)
        barkflags[i]=UNMASKED;

    todo=nexttocomputemask(spl,splflags,Nfreq);
    while(todo!=-1)
    {
        start=(int)(bark[todo])-sub;
        if(start<0)
            start=0;
        else
            start=barklimits[start][0];
```

```
        end=(int)(bark[todo])+add;
        if(end>24)
            end=barklimits[24][1];
        else
            end=barklimits[end][1];

        mpegpm2(spl[todo],todo,Nfreq,start,end,delta,ALPHA);
        for(i=start;i<end;i++)
        {
            if(globalmask[i]>spl[i])
            //if((10.0*log10(globalmask[i])/ALPHA)>spl[i])
                splflags[i]=MASKED;
        }
        todo=nexttocomputemask(spl,splflags,Nfreq);
    }

    /* Post process global mask */
    for(i=0;i<Nfreq;i++)
    {
        if(thresholdofhearing[i]>globalmask[i])
            globalmask[i]=thresholdofhearing[i];
    }

    memcpy(mask,globalmask,sizeof(double)*Nfreq);

    free(spl);
}

int nexttocomputemask(double *spl,short int *splflags,int Nbark)
{
    int i,j=-1,k;
    double max;
    for(i=0;i<25;i++)
    {
        if(barkflags[i]<COMPUTED)
        {
            max=0.0;
            for(k=barklimits[i][0];k<=barklimits[i][1];k++)
            {
                if(splflags[k]==UNMASKED && spl[k]>max)
                {
                    j=k;
                    max=spl[k];
                }
            }
            barkflags[i]++;
            if(j!=-1)
            {
                splflags[j]=COMPUTED;
                return j;
            }
        }
    }
    return j;
}

double signaltomaskratio(double *spl,double *mask,int b)
{
    int i;
    double maxsignal,minmask;

    maxsignal=spl[barklimits[b][0]];
    minmask=mask[barklimits[b][0]];
    for(i=barklimits[b][0];i<=barklimits[b][1];i++)
    {
        if(mask[i]<minmask)
            minmask=mask[i];
        if(maxsignal<spl[i])
            maxsignal=spl[i];
    }

    return maxsignal-minmask;
}

int getbandsize(int b)
{
    return 1+barklimits[b][1]-barklimits[b][0];
}

void getbandlimits(int b,int *lower,int *upper)
{
    *lower=barklimits[b][0];
    *upper=barklimits[b][1];
}
```

A.5. Módulo arithmeticcoder

El módulo arithmeticcoder es un conjunto de funciones para efectuar codificación aritmética.

A.5.1. arithmeticcoder.h

```
#ifndef __ARITHMETICCODER_H__
#define __ARITHMETICCODER_H__


#include "types.h"

struct arcoder
{
    // bit operations
    unsigned char mBitBuffer;
    unsigned char mBitCount;

    // in-/output stream
    BYTE *mByteBuffer;
    int *ByteBufferLength;
    int ByteBufferPointer;

    // encoder & decoder
    unsigned int mLow;
    unsigned int mHigh;
    unsigned int mStep;
    unsigned int mScale;

    // decoder
    unsigned int mBuffer;
};
typedef struct arcoder ARCODER;


enum ModeE
{
    MODE_ENCODE = 0,
    MODE_DECODE
};

struct model
{
    ARCODER mAC;
    BYTE *mSource;
    int mSourcePointer;
    int nSource;
    BYTE *mTarget;
    int mTargetPointer;
    int nTarget;
    unsigned int mCumCount[ 257 ];
    unsigned int mTotal;
};
typedef struct model MODEL;
/* AC */
void initarithmeticcoder(ARCODER *ac);
void ACSetBuffer(BYTE *buffer,int *length,ARCODER *ac);
void ACEncode(const unsigned int low_count, const unsigned int high_count,const unsigned int total,ARCODER *ac);
void ACEncodeFinish(ARCODER *ac);
void ACDecodeStart(ARCODER *ac);
unsigned int ACDecodeTarget(const unsigned int total,ARCODER *ac);
void ACDecode(const unsigned int low_count,const unsigned int high_count,ARCODER *ac);
void ACSetBit(const unsigned char bit,ARCODER *ac);
void ACSetBitFlush(ARCODER *ac);
unsigned char ACGetBit(ARCODER *ac);

/* Model */
void ModelProcess(BYTE *source,BYTE *target,int Nsource,int *Ntarget,enum ModeE mode,MODEL *mmodel);
void initmodel(MODEL *mmodel);
void ModelEncode(MODEL *mmodel);
void ModelDecode(MODEL *mmodel);

int arithmeticencode(BYTE *input,BYTE *output,int Ninput);
int arithmeticdecode(BYTE *input,BYTE *output,int Ninput);

#endif
```

A.5.2. arithmeticcoder.c

```
#include "arithmeticcoder.h"

const unsigned int g_FirstQuarter = 0x20000000;
const unsigned int g_ThirdQuarter = 0x60000000;
const unsigned int g_Half         = 0x40000000;

void initarithmeticcoder(ARCODER *ac)
{
    ac->mBitCount = 0;
    ac->mBitBuffer = 0;

    ac->mLow = 0;
    ac->mHigh = 0x7FFFFFFF; // just work with least significant 31 bits
    ac->mScale = 0;

    ac->mBuffer = 0;
    ac->mStep = 0;
}

void ACSetBuffer(BYTE *buffer, int *length,ARCODER *ac)
{
    ac->mByteBuffer = buffer;
    ac->ByteBufferPointer=0;
    ac->ByteBufferLength = length;
}

void ACSetBit( const unsigned char bit,ARCODER *ac )
{
    // add bit to the buffer
    ac->mBitBuffer = (ac->mBitBuffer << 1) | bit;
    ac->mBitCount++;

    if(ac->mBitCount == 8) // buffer full
    {
        // write
        ac->mByteBuffer[ac->ByteBufferPointer]=ac->mBitBuffer;
        ac->ByteBufferPointer++;
        ac->mBitCount = 0;
    }
}

void ACSetBitFlush(ARCODER *ac)
{
    // fill buffer with 0 up to the next byte
    while( ac->mBitCount != 0 )
        ACSetBit( 0,ac );
}

unsigned char ACGetBit(ARCODER *ac)
{
    unsigned char bit;
    if(ac->mBitCount == 0) // buffer empty
    {
        if( ac->ByteBufferPointer < *(ac->ByteBufferLength) )
        { // file read completely?
            ac->mBitBuffer=ac->mByteBuffer[ac->ByteBufferPointer];
            ac->ByteBufferPointer++;
        }
        else
            ac->mBitBuffer = 0; // append zeros

        ac->mBitCount = 8;
    }

    // extract bit from buffer
    bit = ac->mBitBuffer >> 7;
    ac->mBitBuffer <<= 1;
    ac->mBitCount--;

    return bit;
}

void ACEncode( const unsigned int low_count,const unsigned int high_count,const unsigned int total,ARCODER *ac )
// total < 2^29
{
    // partition number space into single steps
    ac->mStep = ( ac->mHigh - ac->mLow + 1 ) / total; // interval open at the top => +1

    // update upper bound
    ac->mHigh = ac->mLow + ac->mStep * high_count - 1; // interval open at the top => -1

    // update lower bound
    ac->mLow = ac->mLow + ac->mStep * low_count;
```

```
    // apply e1/e2 mapping
    while( ( ac->mHigh < g_Half ) || ( ac->mLow >= g_Half ) )
        {
            if( ac->mHigh < g_Half )
            {
                ACSetBit( 0,ac );
                ac->mLow = ac->mLow * 2;
                ac->mHigh = ac->mHigh * 2 + 1;

                // perform e3 mappings
                for(; ac->mScale > 0; ac->mScale-- )
                    ACSetBit( 1,ac );
            }
            else if( ac->mLow >= g_Half )
            {
                ACSetBit( 1,ac );
                ac->mLow = 2 * ( ac->mLow - g_Half );
                ac->mHigh = 2 * ( ac->mHigh - g_Half ) + 1;

                // perform e3 mappings
                for(; ac->mScale > 0; ac->mScale-- )
                    ACSetBit( 0,ac );
            }
        }

    // e3
    while( ( g_FirstQuarter <= ac->mLow ) && ( ac->mHigh < g_ThirdQuarter ) )
    {
        // keep necessary e3 mappings in mind
        ac->mScale++;
        ac->mLow = 2 * ( ac->mLow - g_FirstQuarter );
        ac->mHigh = 2 * ( ac->mHigh - g_FirstQuarter ) + 1;
    }
}

void ACEncodeFinish(ARCODER *ac)
{
    // There are two possibilities of how mLow and mHigh can be distributed,
    // which means that two bits are enough to distinguish them.
    unsigned int i;
    if( ac->mLow < g_FirstQuarter ) // mLow < FirstQuarter < Half <= mHigh
    {
        ACSetBit( 0,ac );

        for( i=0; i<ac->mScale+1; i++ ) // perform e3-skaling
            ACSetBit(1,ac);
    }
    else // mLow < Half < ThirdQuarter <= mHigh
    {
        ACSetBit( 1,ac ); // zeros added automatically by the decoder; no need to send them
    }

    // empty the output buffer
    ACSetBitFlush(ac);
    *(ac->ByteBufferLength)=ac->ByteBufferPointer;
}

void ACDecodeStart(ARCODER *ac)
{
    // Fill buffer with bits from the input stream
    int i;
    for( i=0; i<31; i++ ) // just use the 31 least significant bits
        ac->mBuffer = ( ac->mBuffer << 1 ) | ACGetBit(ac);
}

unsigned int ACDecodeTarget(const unsigned int total,ARCODER *ac)
// total < 2^29
{
    // split number space into single steps
    ac->mStep = ( ac->mHigh - ac->mLow + 1 ) / total; // interval open at the top => +1

    // return current value
    return ( ac->mBuffer - ac->mLow ) / ac->mStep;
}

void ACDecode( const unsigned int low_count,const unsigned int high_count,ARCODER *ac )
{
    // update upper bound
    ac->mHigh = ac->mLow + ac->mStep * high_count - 1; // interval open at the top => -1

    // update lower bound
    ac->mLow = ac->mLow + ac->mStep * low_count;

    // e1/e2 mapping
    while( ( ac->mHigh < g_Half ) || ( ac->mLow >= g_Half ) )
        {
```

```
            if( ac->mHigh < g_Half )
            {
                ac->mLow = ac->mLow * 2;
                ac->mHigh = ac->mHigh * 2 + 1;
                ac->mBuffer = 2 * ac->mBuffer + ACGetBit(ac);
            }
            else if( ac->mLow >= g_Half )
            {
                ac->mLow = 2 * ( ac->mLow - g_Half );
                ac->mHigh = 2 * ( ac->mHigh - g_Half ) + 1;
                ac->mBuffer = 2 * ( ac->mBuffer - g_Half ) + ACGetBit(ac);
            }
            ac->mScale = 0;
        }

    // e3 mapping
    while( ( g_FirstQuarter <= ac->mLow ) && ( ac->mHigh < g_ThirdQuarter ) )
    {
        ac->mScale++;
        ac->mLow = 2 * ( ac->mLow - g_FirstQuarter );
        ac->mHigh = 2 * ( ac->mHigh - g_FirstQuarter ) + 1;
        ac->mBuffer = 2 * ( ac->mBuffer - g_FirstQuarter ) + ACGetBit(ac);
    }
}

/* Model*/
void ModelProcess(BYTE *source,BYTE *target,int Nsource,int *Ntarget,enum ModeE mode,MODEL *mmodel)
{
    mmodel->mSource = source;
    mmodel->mTarget = target;
    mmodel->nSource = Nsource;
    mmodel->mSourcePointer=0;
    mmodel->mTargetPointer=0;

    if( mode == MODE_ENCODE )
    {
        ACSetBuffer( mmodel->mTarget,Ntarget,&(mmodel->mAC) );

        // encode
        ModelEncode(mmodel);

        ACEncodeFinish(&(mmodel->mAC));
    }
    else // MODE_DECODE
    {
        ACSetBuffer( mmodel->mSource,&Nsource,&(mmodel->mAC));

        ACDecodeStart(&(mmodel->mAC));

        // decode
        ModelDecode(mmodel);
        *Ntarget=mmodel->mTargetPointer;
    }
};

void initmodel(MODEL *mmodel)
{
    // initialize probabilities with 1
    unsigned int i;
    initarithmeticcoder(&(mmodel->mAC));

    mmodel->mTotal = 257; // 256 + escape symbol for termination
    for( i=0; i<257; i++ )
        mmodel->mCumCount[i] = 1;
}

void ModelEncode(MODEL *mmodel)
{

    while( mmodel->mSourcePointer < mmodel->nSource )
    {
        unsigned char symbol;

        // read symbol
        symbol = mmodel->mSource[mmodel->mSourcePointer];
        mmodel->mSourcePointer++;

        if( mmodel->mSourcePointer <= mmodel->nSource )
        {
            // cumulate frequencies
            unsigned int low_count = 0;
            unsigned char j;
            for(  j=0; j<symbol; j++ )
                low_count += mmodel->mCumCount[j];

            // encode symbol
```

```
            ACEncode( low_count, low_count + mmodel->mCumCount[j], mmodel->mTotal,&(mmodel->mAC));

            // update model
            mmodel->mCumCount[ symbol ]++;
            mmodel->mTotal++;
        }
    }

    // write escape symbol for termination
    ACEncode( mmodel->mTotal-1, mmodel->mTotal, mmodel->mTotal,&(mmodel->mAC));
}

void ModelDecode(MODEL *mmodel)
{
    unsigned int symbol;

    do
    {
        unsigned int value;
        unsigned int low_count = 0;

        // read value
        value = ACDecodeTarget( mmodel->mTotal,&(mmodel->mAC) );


        // determine symbol
        for( symbol=0; low_count + mmodel->mCumCount[symbol] <= value; symbol++ )
            low_count += mmodel->mCumCount[symbol];

        // write symbol
        if( symbol < 256 )
        {
            mmodel->mTarget[mmodel->mTargetPointer]=symbol;
            mmodel->mTargetPointer++;
        }

        // adapt decoder
        ACDecode( low_count, low_count + mmodel->mCumCount[ symbol ],&(mmodel->mAC) );

        // update model
        mmodel->mCumCount[ symbol ]++;
        mmodel->mTotal++;
    }
    while( symbol != 256 ); // until termination symbol read
}

int arithmeticencode(BYTE *input,BYTE *output,int Ninput)
{
    MODEL mmodel;
    int Noutput;

    // choose model, here just order-0
    initmodel(&mmodel);

    /* Encode */
    //cout << "Encoding " << endl;
    ModelProcess( input, output, Ninput, &Noutput,MODE_ENCODE,&mmodel);

    return Noutput;
}

int arithmeticdecode(BYTE *input,BYTE *output,int Ninput)
{
    MODEL mmodel;
    int Noutput;

    // choose model, here just order-0
    initmodel(&mmodel);

    //cout << "Decoding " << endl;
    ModelProcess( input, output, Ninput, &Noutput,MODE_DECODE,&mmodel);


    return Noutput;
}
```

A.6. types.h

En este archivo se definen los tipos necesarios para manipular adecuadamente los archivos.

```
#ifndef _TYPES_H_
#define _TYPES_H_

/* Types should be adjusted depending upon the computer architecture.
Data in a WAV file is stored in little endian format.
These types work for Win32 and Linux x86 IA32:
    typedef unsigned char BYTE;
    typedef short int WORD;
    typedef unsigned int DWORD;
 */
typedef unsigned char BYTE;
typedef short int WORD;
typedef unsigned int DWORD;
typedef unsigned short int UWORD;
typedef unsigned long int QWORD;
#endif
```

A.7. Módulo wav

El módulo wav contiene funciones para la lectura y escritura de archivos PCM WAV.

A.7.1. wav.h

```
#ifndef _WAV_H_
#define _WAV_H_

#include "types.h"


#define WAVE_FORMAT_PCM     0x0001
#define FORMAT_MULAW        0x0101
#define IBM_FORMAT_ALAW     0x0102
#define IBM_FORMAT_ADPCM    0x0103

struct wavfile
{
    FILE *fwav;
    int samplerate;
    int bitspersample;
    int channels;
    unsigned int totalsamples;
    unsigned int samplesread;
};
typedef struct wavfile WAVFILE;


struct riff_hdr
{
    BYTE id[4];         /* identifier string = "RIFF" */
    DWORD len;          /* remaining length after this header */
};

struct wav_hdr
{
    BYTE wave_id[4];    /* File identifier = "WAVE" */
    BYTE id[4];         /* identifier, "fmt " or "data" */
    DWORD len;          /* Size of data after this header */
};


struct wavfmtck
{
    WORD wFormatTag;            /* Format */
```

```
    WORD wChannels;             /* Number of channels */
    DWORD dwSamplesPerSec;      /* Sampling frequency */
    DWORD dwAvgBytesPerSec;     /* Used for buffer estimation */
    WORD wBlockAlign;           /* Size of data block */
    WORD wBitsPerSample;        /* Bits per sample */
};

struct datachunk_hdr
{
    BYTE chunkID[4];        /* Identifier, "data" for sound chunk */
    DWORD chunkSize;        /* Size of chunk */
};

struct sample
{
    WORD wLeft;
    WORD wRight;
}; /* Assumes 16 bits, stereo */

int readinfo(WAVFILE *wav);
int readsample(WAVFILE *wav,struct sample *ssample);
void writeinfo(WAVFILE *wav);
int writesample(WAVFILE *wav,double left,double right);
int writewav(WAVFILE *wav,double *left,double *right,double maxl,double maxr);
int readwav(WAVFILE *wav,double **left,double **right);

#endif
```

A.7.2. wav.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include "wav.h"


/* Reads the information of a WAV file pointed by wav.
    Returns:
         0: On success.
        -1: If wav is not a WAV file.
        -2: If the WAV is not in PCM format.
        -3: If the file is not PCM 44100Hz, 16bits, stereo.
    For 0 and -3, the arguments samplerate, bitspersample, channels and nsamples contain
    the sampling frequency, bits per sample, number of channels and the number of samples
    in the file. */
int readinfo(WAVFILE *wav)
{
    struct riff_hdr riffheader;
    struct wav_hdr wavheader;
    struct wavfmtck format;
    struct datachunk_hdr dthdr;
    fseek(wav->fwav,0,SEEK_SET);
    fread(&riffheader,sizeof(struct riff_hdr),1,wav->fwav);
    fread(&wavheader,sizeof(struct wav_hdr),1,wav->fwav);
    fread(&format,sizeof(struct wavfmtck),1,wav->fwav);

    if(strncmp(wavheader.wave_id,"WAVE",4))
        return -1;
    if(format.wFormatTag!=WAVE_FORMAT_PCM)
        return -2;

    wav->samplerate=format.dwSamplesPerSec;
    wav->bitspersample=format.wBitsPerSample;
    wav->channels=format.wChannels;

    if(wav->channels!=2 || wav->samplerate!=44100 || wav->bitspersample!=16)
        return -3;

    /* Moves until it reaches the "data" chunk which contains the samples */
    dthdr.chunkSize=wavheader.len-sizeof(struct wavfmtck);
    do
    {
        fseek(wav->fwav,dthdr.chunkSize,SEEK_CUR);
        fread(&dthdr,sizeof(struct datachunk_hdr),1,wav->fwav);
    }while(strncmp(dthdr.chunkID,"data",4));

    /* Computes number of samples */
    wav->totalsamples = dthdr.chunkSize / wav->channels / (wav->bitspersample/8);
    wav->samplesread=0;

    return 0;
}
```

```
/* Reads the next sample of the wav file and stores it in sample.
    Returns:
         0: On success.
        -1: On failure (most likely end of file).
    WARNING: This function only reads the data located at the current file position.
    You should use readinfo before using this function for the first time. */
int readsample(WAVFILE *wav,struct sample *ssample)
{
    int i;
    if(wav->samplesread < wav->totalsamples)
        i=fread(ssample,sizeof(struct sample),1,wav->fwav);
    if(i!=1)
        return -1;
    wav->samplesread++;
    return 0;
}

void writeinfo(WAVFILE *wav)
{
    struct riff_hdr riffheader;
    struct wav_hdr wavheader;
    struct wavfmtck format;
    struct datachunk_hdr dthdr;
    fseek(wav->fwav,0,SEEK_SET);

    wav->samplesread=0;
    riffheader.id[0]='R';
    riffheader.id[1]='I';
    riffheader.id[2]='F';
    riffheader.id[3]='F';
    riffheader.len=wav->totalsamples*wav->bitspersample/8*wav->channels + sizeof(struct wav_hdr) + sizeof(struct wavfmtck) +
        sizeof(struct datachunk_hdr);

    wavheader.wave_id[0]='W';
    wavheader.wave_id[1]='A';
    wavheader.wave_id[2]='V';
    wavheader.wave_id[3]='E';
    wavheader.id[0]='f';
    wavheader.id[1]='m';
    wavheader.id[2]='t';
    wavheader.id[3]=' ';
    wavheader.len=sizeof(struct wavfmtck);

    format.wFormatTag=WAVE_FORMAT_PCM;
    format.dwSamplesPerSec=wav->samplerate;
    format.wBitsPerSample=wav->bitspersample;
    format.wChannels=wav->channels;
    format.wBlockAlign=4;
    format.dwAvgBytesPerSec=wav->bitspersample/8 * wav->samplerate*wav->channels;

    dthdr.chunkID[0]='d';
    dthdr.chunkID[1]='a';
    dthdr.chunkID[2]='t';
    dthdr.chunkID[3]='a';
    dthdr.chunkSize = wav->totalsamples * wav->bitspersample/8 * wav->channels;

    fwrite(&riffheader,sizeof(struct riff_hdr),1,wav->fwav);
    fwrite(&wavheader,sizeof(struct wav_hdr),1,wav->fwav);
    fwrite(&format,sizeof(struct wavfmtck),1,wav->fwav);
    fwrite(&dthdr,sizeof(struct datachunk_hdr),1,wav->fwav);
}

int writesample(WAVFILE *wav,double left,double right)
{
    int i;
    struct sample samplewav;
    double norm;
    if(wav->samplesread < wav->totalsamples)
    {
        norm=fabs(left);
        if(norm<1.0)
            norm=1.0;
        samplewav.wLeft=(WORD)(left*32767.0/norm);
        norm=fabs(right);
        if(norm<1.0)
            norm=1.0;
        samplewav.wRight=(WORD)(right*32767.0/norm);
        i=fwrite(&samplewav,sizeof(struct sample),1,wav->fwav);
    }
    if(i!=1)
        return -1;
    wav->samplesread++;
    return 0;
}
```

```
int writewav(WAVFILE *wav,double *left,double *right,double maxl,double maxr)
{
    unsigned int i;
    struct sample samplewav;

    /* Signal is within range, no need to adjust the dynamic range */
    if(maxl<1.0)
        maxl=1.0;
    if(maxr<1.0)
        maxr=1.0;

    writeinfo(wav);
    for(i=0;i<wav->totalsamples;i++)
    {
        samplewav.wLeft=(WORD)(left[i]*32767.0/maxl);
        samplewav.wRight=(WORD)(right[i]*32767.0/maxr);
        fwrite(&samplewav,sizeof(struct sample),1,wav->fwav);
    }

    return 0;
}

/* Returns:
        0 on success
        same code as readinfo otherwise.
    Allocates the needed memory to *left and *right */
int readwav(WAVFILE *wav,double **left,double **right)
{
    struct sample ssample;
    unsigned int count;
    int ret;

    ret=readinfo(wav);
    if(ret)
        return ret;

    count=0;
    *left=(double *)malloc(sizeof(double)*(wav->totalsamples));
    *right=(double *)malloc(sizeof(double)*(wav->totalsamples));
    while(!readsample(wav,&ssample))
    {
        //printf("%d || %1.8f %1.8f\n",cuenta,muestra.wLeft/32768.0,muestra.wRight/32768.0);
        (*left)[count]=ssample.wLeft/32768.0;
        (*right)[count]=ssample.wRight/32768.0;
        count++;
    }
    return 0;
}
```

Referencias

[Ashland] Ashland, M. T. Monkey's audio. http://www.monkeysaudio.com/. Consulta: Agosto 2005.

[ATS95] *United States Advanced Television Systems Committee Digital Audio Compresion (AC-3) Standard, Doc A/52/10*, December 1995.

[Buschmann] Buschmann, A. Musepack: Living audio compression. http://musepack.net/. Consulta: Agosto 2005.

[Chi-Min Liu99] Chi-Min Liu, W.-C. L. A unified fast algorithm for cosine modulated filter banks in current audio coding standards. *Journal Audio Engineering Society*, 47(12):1061–1075, 1999.

[Coalson] Coalson, J. Flac: Free lossless audio codec. http://flac.sourceforge.net/. Consulta: Agosto 2005.

[Eberhard Zwicker99] Eberhard Zwicker, H. F. *Psychoacoustics: Facts and Models*. Information Sciences. Springer, 2ª edón., 1999. ISBN 3-540-65063-6.

[Eric Bodden02] Eric Bodden, J. K., Malte Clasen. *Arithmetic Coding Revealed: A guide from theory to praxis*. RWTH Aachen University, 2002.

[Glenn Zelnicker94] Glenn Zelnicker, F. J. T. *Advanced Digital Signal Processing: Theory and Applications*. Marcel Dekker, 1ª edón., 1994. ISBN 0-8247-9145-2.

[Gluth91] Gluth, R. Regular fft-related transform kernels for dct/dst-based polyphase filter banks. *Proc. of the ICASSP*, 3:2205–2208, 1991.

[Huffman52] Huffman, D. A. A method for the construction of minimum-redundancy codes. *Proceedings of the I.R.E.*, 40(9):1098–1101, September 1952.

[Ian H. Witten99] Ian H. Witten, T. C. B., Alistair Moffat. *Managing Gigabytes: Compressing and Indexing Documents and Images*. Morgan Kaufmann Publishers, 2ª ed^ón^., 1999. ISBN 1-55860-570-3.

[IEEE85] IEEE. *IEEE 754-1985 IEEE Standard for Binary Floating-Point Arithmetic*. IEEE, 1985.

[ITU-R97] ITU-R. *Methods for the Subjective Assesment of Small Impairments in Audio Systems Including Multichannel Sound Systems*. International Telecommunications Union, Radiocomunnication Sector BS.1116 (rev. 1), 1997.

[John P. Princen86] John P. Princen, A. B. B. Analysis/synthesis filter bank design based on time domain aliasing cancellation. *IEEE TRANSACTIONS ON ACOUSTICS, SPEECH, AND SIGNAL PROCESSING*, ASSP-34(5):1153–1161, 1986.

[John P. Princen87] John P. Princen, A. B. B., A. W. Johnson. Subband/transform coding using filter bank designs based on time domain aliasing cancellation. *Proc. of the ICASSP*, págs. 2161–2164, 1987.

[Konstantinides03] Konstantinides, K. An introduction to super audio cd and dvd-audio. *IEEE SIGNAL PROCESSING MAGAZINE*, 20(4):71–82, July 2003.

[Lufti83] Lufti, R. A. Additivity of simultaneous masking. *Journal of the Acoustical Society of America*, 73(1):162–267, January 1983.

[M. R. Schroeder79] M. R. Schroeder, J. L. H., B. S. Atal. Optimizing digital speech coders by exploiting masking properties of the human ear. *Journal of the Acoustical Society of America*, 66(6):1647–1652, December 1979.

[Marina Bosi02] Marina Bosi, R. E. G. *Introduction to Digital Audio Coding and Standards*. Engineering and Computer Science. Kluwer Academic Publishers, 1ª edón., 2002. ISBN 1-4020-7357-7.

[Moore96] Moore, B. C. J. Masking in the human auditory system. *Collected Papers on Digital Audio Bit-Rate Reduction*, págs. 9–19, May 1996.

[MPEG93] MPEG. *ISO/IEC 11172-3, Coding of Moving Pictures And Associated Audio For Digital Storage Media at up to About 1.5 Mbit/s Part 3 Audio*. ISO/IEC, 1993.

[MPEG97] MPEG. *ISO/IEC 13818-7, Generic coding of moving pictures and associated audio information – Part 7: Advanced Audio Coding (AAC)*. ISO/IEC, 1997.

[MPEG01] MPEG. *ISO/IEC 14496-3, Coding of audio-visual objects – Part 3: Audio*. ISO/IEC, 1999-2001.

[Pen-Shu Yeh91] Pen-Shu Yeh, W. M., Robert Rice. On the optimality of code options for a universal noiseless coder. *Jet Propulsion Laboratories Publication*, 91(2), February 1991.

[Rice91] Rice, R. F. Some practical noiseless coding techniques, part ii. *Jet Propulsion Laboratories Publication*, 91(3), November 1991.

[Robert Rice71] Robert Rice, J. P. Adaptive variable-length coding for efficient compression of spacecraft television data. *IEEE Transactions on Communications*, 19(6):889–897, December 1971.

[Robinson94] Robinson, T. Shorten: Simple lossless and near-lossless waveform compression. *Technical Report CUED/F-INFENG/TR.156*, December 1994.

[Shannon48] Shannon, C. E. A mathematical theory of communications. *Bell System Technical Journal*, 27:379–423, July 1948.

[Sporer92] Sporer, E., Brandenburg. The use of multirate filter banks for coding of high quality digital audio. *6th European Signal Processing Conference (EUSIPCO)*, 1:211–214, June 1992.

[Terhardt79] Terhardt, E. Calculating virtual pitch. *Hear. Res.*, 1:155–182, 1979.

[Tukey53] Tukey, J. W. The problem of multiple comparisons. *The collected works of John W. Tukey*, VIII, 1953.

[U.S96] U.S. National Communications System Technology & Standards Division. *Federal Standard 1037C*, August 1996.

[VOR] The ogg vorbis codec project. http://www.vorbis.com/. Consulta: Agosto 2005.

[Watkinson04] Watkinson, J. *The MPEG Handbook*. Focal Press, 2ª edón., 2004. ISBN 0-240-80578-X.

[WAV] Wavpack hybrid lossless audio compression. http://www.wavpack.com/. Consulta: Agosto 2005.

[Welch84] Welch, T. A. A technique for high-performance data compression. *IEEE Computer*, 17(6):8–19, June 1984.

[William H. Press92] William H. Press, W. T. V. B. P. F., Saul A. Teukolsky. *Numerical Recipes in C: The Art of Scientific Computing*. Cambridge University Press, 2ª edón., 1992. ISBN 0-521-43108-5.

Glosario

artefacto Ruido perceptible introducido por un proceso de compresión.

CD *Compact Disc*-Disco Compacto

codec *coder-decoder*-Codificador decodificador

DFT *Discrete Fourier Transform*-Transformada Discreta de Fourier

FFT *Fast Fourier Transform*-Transformada Rápida de Fourier

IMDCT *Inverse Modified Discrete Cosine Transform*-Transformada Discreta de Coseno Modificada Inversa

KBD *Kaiser-Bessel Derived*-Ventana del tipo Derivada de Kaiser-Bessel

kbps *kilobits per second*-kilobits por segundo

MDCT *Modified Discrete Cosine Transform*-Transformada Discreta de Coseno Modificada

NMR *Noise to Mask Ratio*-Razón ruido a máscara

SMR *Signal to Mask Ratio*-Razón señal a máscara

SNR *Signal to Noise Ratio*-Razón señal a ruido

SPL *Sound Pressure Level*-Nivel de Presión de Sonido

TOH *Threshold of Hearing*-Umbral auditivo

Índice

alias, 6
asignación de bits, 37

Bark
  escala, 24

codec, 13
  de transformación, 13
codificación, 2
  aritmética, 30
compresión
  con pérdida, 2
  sin pérdida, 2
cuantización, 7
  logarítmica, 10
  midrise, 8
  midtread, 8
  paso de, 7, 38
  Power-Law, 9
  punto flotante, 10
  ruido de, 28

decibel
  escala, 22

enmascaramiento
  simultaneo, 23, 36
  temporal, 26

IMDCT, 15
  implementación rápida de, 20

MDCT, 13, 14
  implementación rápida de, 19
muestreo
  frecuencia de, 6
  teorema del, 5

nivel de intensidad de sonido, 22
nivel de presión de sonido, 22
  de una MDCT, 29
  de una transformada de Fourier, 29
Nyquist, frecuencia de, 6

pre-eco, 26

signal to noise ratio, 11
  CD-Audio, 11
  DVD-Audio, 11

umbral
  auditivo, 21, 22, 36
  auditivo, ecuación de, 22
  del dolor, 22
  global de enmascaramiento, 21, 27

ventana

Derivada de Kaiser-Bessel, 15

válida para la MDCT, 15