Universidad Nacional de Rosario
Facultad de Ciencias Exactas, Ingeniería y Agrimensura
Escuela de Ingeniería Electrónica
Proyecto de Ingeniería

Ampliación de un software de edición de audio digital para su uso en investigación en acústica aplicada, música y otras disciplinas.

Autor: Luciano Guillermo Boggino

Director: Ing. Federico Miyara

Noviembre 2014

Universidad Nacional de Rosario
Facultad de Ciencias Exactas, Ingeniería y Agrimensura
Escuela de Ingeniería Electrónica

Resumen

Este documento presenta un proyecto de grado para la obtención del título de Ingeniero Electrónico otorgado por la *Facultad de Ciencias Exactas, Ingeniería y Agrimensura*. El proyecto aborda la adaptación y ampliación de un software libre de edición de audio digital para su uso en la investigación en acústica aplicada, música y otras disciplinas.

Capítulo 1

Agradecimientos

Agradezco al Laboratorio de Acústica y Electroacústica (LAE) dependiente de la Facultad de Ciencias Exactas, Ingeniería y Agrimensura (FCEIA) por prestar sus instalaciones para la realización del proyecto y, en particular, al Ing. Federico Miyara, director del laboratorio, por dirigir el proyecto y al Ing. Ernesto Accolti por su ayuda y orientación; a la Facultad de Ciencias Exactas, Ingenería y Agrimensura (FCEIA) de la Universidad Nacional de Rosario (UNR) por ser el marco académico para el desarrollo del proyecto y a sus docentes; al Ing. Gonzalo Sad y a Simone Campanini por sus aportes en conocimientos y su ayuda; a todas las personas que me apoyaron a lo largo de éste trabajo y la carrera.

Índice general

Capítulo 2

Objetivos

El proyecto tiene por objetivo general la obtención y modificación de las prestaciones de un software de edición, síntesis y análisis de señales de audio digitales para su uso en tareas de investigación, en las que estén involucradas señales acústicas en diversas áreas, entre otras: acústica física, ruido urbano, psicoacústica, audio, música, emisiones acústicas, vibraciones mecánicas.

Los objetivos específicos se detallan a continuación

1. Decodificación del código fuente de Audacity para conocer su composición y posterior documentación.
2. Modificaciónes del código fuente y de herramientas propias del software:
   a) Creación de diversas escalas de colores para la gráfica de espectrogramas.
   b) Integración de la opción para analizar el espectro de la señal en un punto específico de la misma.
   c) Adición de la posibilidad de observar el nivel de presión sonora ponderada en tiempo y frecuencia.
   d) Incorporación de la facultad de importar y exportar los metadatos pertenecientes a un archivo con formato ”.wav”.
3. Desarrollo de herramientas e incorporación al software de las mismas:
   a) Filtrado espectral mediante el dibujo sobre el espectrograma de la señal.
   b) Filtrado utilizando curvas ingresadas por medio de una tabla y posterior interpolación entre ellas
   c) Calculador de niveles estadísticos.
4. Redacción de una guía que ayude a compilar Audacity y crear un Plug-in que pueda ser incorporado al mismo.

Capítulo 2

Marco teórico

En este capítulo se presenta una breve introducción de lo que este proyecto buscaba y los conocimientos generales que son necesarios para la comprensión adecuada del mismo. Esencialmente se explica qué es el audio digital, el funcionamiento de los reproductores digitales, la composición de un archivo de audio digital (en especial el formato *WAVE*), las distintas formas de representación de señales de audio, la descomposición en frecuencias de una señal y el funcionamiento de los filtros digitales, los cuales son la base del proyecto.

2.1 Introducción

La investigación en acústica y otras disciplinas que involucran manejo de señales digitales requiere a menudo la síntesis de señales con características dadas, el análisis y medición de señales adquiridas o generadas, su procesamiento y edición mediante métodos diversos, o su modelado y simulación [17].

La síntesis se utiliza frecuentemente con la finalidad de disponer de señales de prueba con propiedades conocidas para la excitación y ensayo de diversos sistemas, que incluyen materiales y estructuras (por ejemplo determinación de propiedades físicas tales como absorción o aislación sonora, elasticidad, etc.), recintos (como ser, ensayo de reverberación y otras cualidades acústicas), hardware (tales como, transductores, procesadores de señal, dispositivos físicos) ([20]; [18]), software (el caso de algoritmos de procesamiento) o incluso el propio oído humano (entre otros en estudios psicoacústicos o de respuesta subjetiva, efectos del ruido, etc.) ([18]). También es de importancia en la creación de música electroacústica, simulación de paisajes sonoros ([21]) y efectos sonoros en diversas artes audiovisuales.

El análisis de señales permite estudiar el comportamiento de los sistemas que las producen, así como los efectos que dichas señales pueden producir o, a la inversa, predecir el efecto que producirá un sistema previamente conocido sobre la señal (a modo de ejemplo, los efectos de propagación del ruido urbano a través de barreras) ([19]; [22]).

La edición y procesamiento tiene por finalidad la depuración (remoción de ruido, por ejemplo) o adaptación de la señal para una determinada aplicación (diversos tipos de filtrado). Es, incluso, parte de los procesos avanzados de síntesis y de análisis previamente mencionados.

Por último, el modelado de señales tiene aplicaciones en compresión de audio, en codificación y decodificación ([23], [24]), en técnicas de resíntesis, en simulación aural ([25], [26], [27], [28], [29]) y en realidad acústica virtual ([30]), incluyendo efectos sonoros.

Existen numerosas aplicaciones de software que permiten la realización de varios de los procesamientos aquí descriptos. Ellas se pueden dividir en tres grandes grupos: 1) Programas de edición de audio digital (tales como Cool Edit, Cool Edit Pro, Adobe Audition, Sound Forge, GoldWave, Cubase, Nuendo, Spear, Audacity) , 2) Programas matemáticos que permiten implementar toda clase de algoritmos de aplicación al análisis y procesamiento de señales digitales (se puede citar Matlab, Scilab, Octave) y 3) programas de aplicación específica asociados con instrumentos

de medición acústica o adquisidores de señal (tal como LabView), con el estudio de determinados tipos de señales, como la voz (algunos son Anagraf, VoxMetria, Speechstation, CSRE), o con manipulación y síntesis algorítmica o por bloques de sonidos musicales (C-Sound, MAX-SP, Processing), así como instrumentos musicales virtuales (VSTi, Reaktor).

Es necesario entonces, un software que reúna las ventajas de los editores de sonido y de los programas matemáticos. Dicho software no existe al día de hoy.

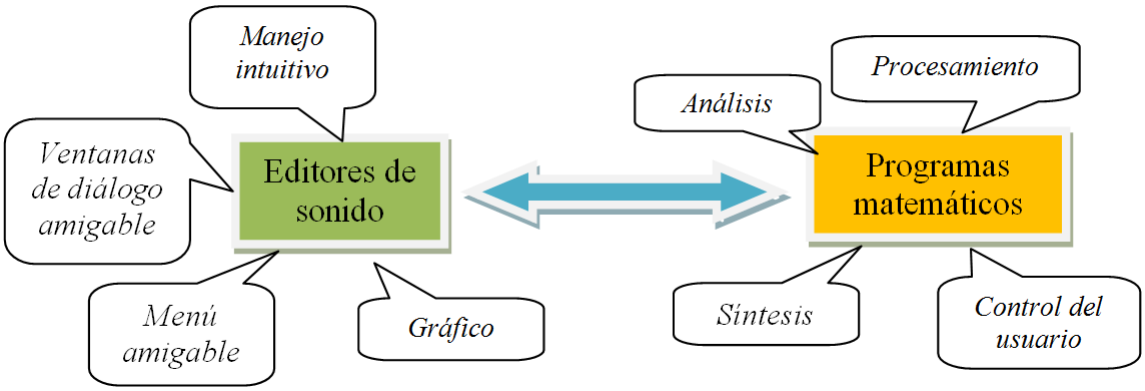


Figura 2.1: Diagrama que sintetiza el objetivo del presente proyecto.

2.2 Audio digital

Una señal es la representación de una magnitud física, como ser la presión sonora o la tensión eléctrica, se puede catalogar en dos grandes grupos, continuas o discretas. Mientras la primera puede adoptar una cantidad infinita de valores en infinitos instantes de tiempo, la segunda sólo puede tomar una cantidad finita de valores en un número finito de instantes. Para obtener una señal digital necesitamos *muestrear* la señal, es decir, obtener el valor de la misma en algunos instantes de tiempo. La señal digital es, entonces, una señal discretizada en tiempo y magnitud.

Las señales digitales presentan, frente a las analógicas, varias ventajas que las hacen atractivas en muchas aplicaciones, especialmente en las metrológicas:

- Almacenada en un medio apropiado son prácticamente inalterables.
- Es posible generar tantas copias como se deseen.
- Pueden ser transmitidas tanto por cable como inalambricamente, sin sufrir alteraciones debido a la existencia de ruido.
- En caso de canales ruidosos o medios delicados es posible implementar algoritmos de corrección de errores basados en redundancia.
- Es posible implementar una gran cantidad de procesos digitales tales como filtros, reducción de ruido, análisis espectral, etc.

Por otro lado presentan como contrapartida que:

- El hardware requerido es generalmente costoso.
- La discretización en tiempo impone asimismo, una severa limitación del espectro en frecuencias elevadas.

Se puede observar a simple vista que las ventajas superan a las desventajas.

2.2.1 Muestreo

El proceso de muestreo consiste en retener el valor de la señal en el instante en que se deséa muestrear hasta que se tome la próxima muestra, como se ve en la Figura 2.2. Esto se puede lograr mediante circutos de *muestreo y retención* o *sample & hold*.

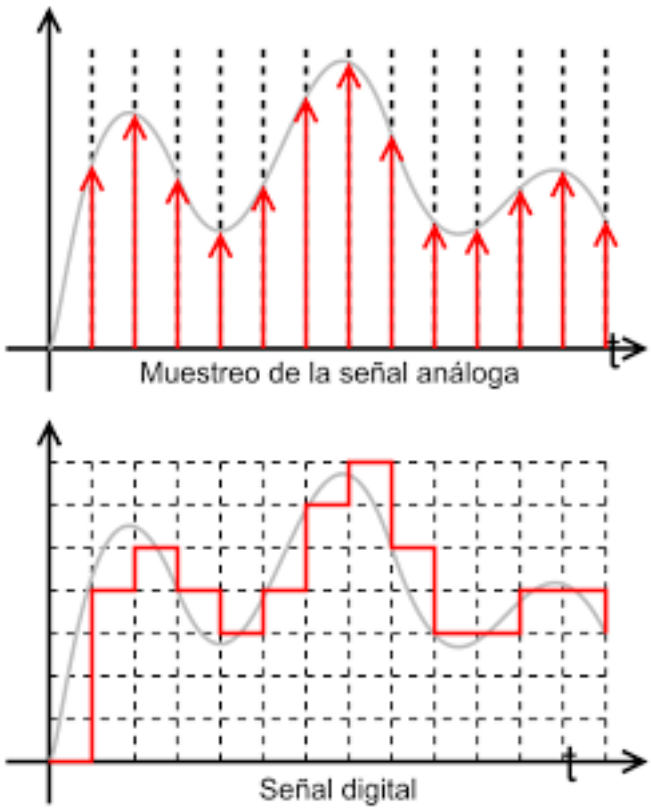


Figura 2.2: Ejemplo de muestreo y renteción. En gris la señal original, en rojo la señal muestreada y en línea de puntos los instantes de muestreo [16].

Es muy importante definir cuántas muestras por unidad de tiempo se tomarán de la señal original, es decir, la *tasa o frecuencia de muestreo* conocida como F_s, que a su vez es la inversa

del *periodo de muestreo* llamado comúnmente T_s siendo éste el valor en segundos del tiempo que transcurre entre dos muestras consecutivas.

$$F_s = \frac{1}{T_s} \tag{2.1}$$

Por un lado es conveniente que la frecuencia de muestreo sea alta para no perder información o detalles de la señal, pero por otro, existen limitaciones desde el punto de vista de la memoria y velocidad de cómputo necesarios para lograrlo.

Es posible demostrar que si la tasa de muestreo es mayor o igual al doble de la máxima frecuencia que contenga la señal, es decir

$$F_s \geq 2f_{max} \tag{2.2}$$

se puede reconstruir a la perfección la señal original a partir de las muestras. Este resultado se conoce como *teorema de muestreo* y la desigualdad como *condición de Nyquist*.

Si no se satisface esta desigualdad aparece un fenómeno denominado *aliasing*, se muestra un ejemplo en al Figura 2.3, que consiste en la aparición de frecuencias inexistentes a la hora de reconstruir la señal original. Estas frecuencias equivalen a reflexiones centradas en $F_s/2$ de todas las frecuencias que excedan este valor.

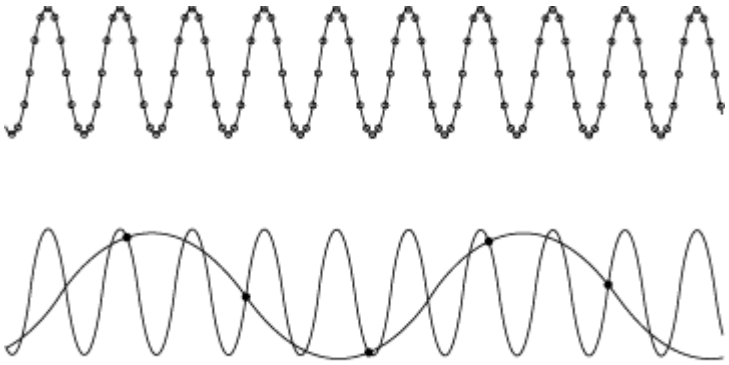


Figura 2.3: Ejemplo de aliasing. Arriba la señal correctamente muestreada y reconstruida, abajo la señal muestreada incorrectamente y su reconstrucción [15].

2.2.2 Digitalización

Una vez muestreada la señal discreta se debe digitalizar. Para ello se utiliza un conversor analógico/digital, el cual asigna a cada valor discreto un número proporcional a si mismo. Si bien se podría utilizar cualquier base de numeración, se utiliza la binaria o base 2, ya que permite usar el concepto de llave abierta o cerrada en circuitos de lógica binaria con dos niveles de tension, 0 y V_{cc}. El uso de esta base permite la mayor inmunidad frente al ruido, ya que sería necesario un ruido superior a $\frac{V_{cc}}{2}$ para producir un error.

Para llevar a cabo la digitalización primero se debe definir un valor referencia V_{ref}, y luego se lo subdivide en 2^n escalones iguales, donde n es la resolución del dato digital. Para señales

positivas se le asigna el valor 0 digital al 0 analógico y el valor máximo $2^n - 1$ a $V_{ref}(1 - \frac{1}{2^n})$ y para señales bidireccionales el 0 digital corresponde a $-\frac{V_{ref}}{2}$ y el valor $2^n - 1$ es asignado a $V_{ref}(\frac{1}{2} - \frac{1}{2^n})$. La falta de simetría se hace irrelevante al aumentar la resolución.

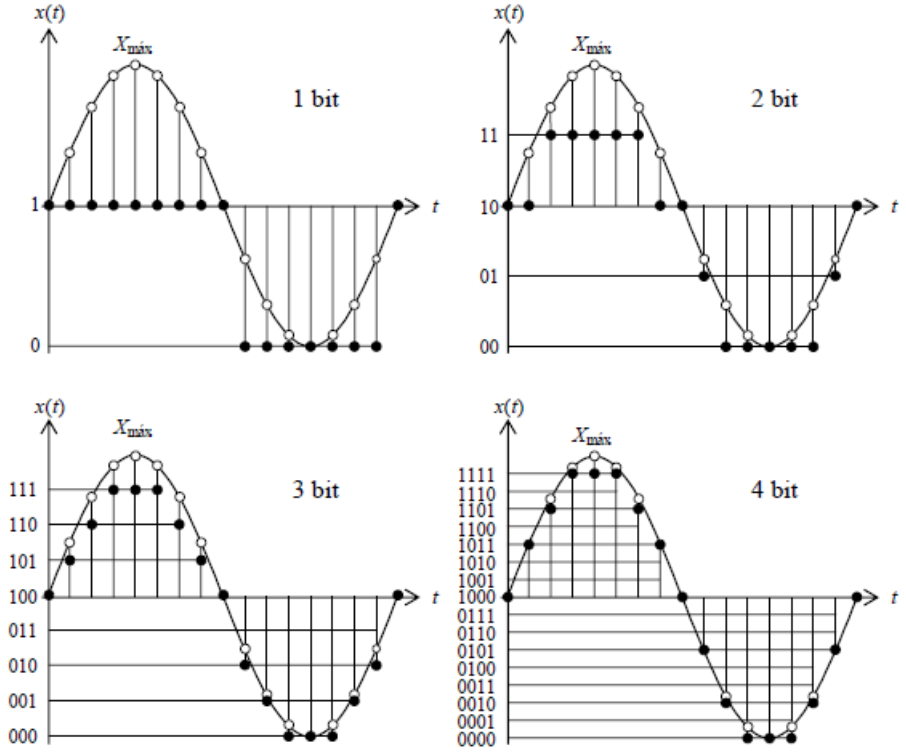


Figura 2.4: Ejemplo de digitalización con 1, 2, 3 y 4 bits [1].

2.3 Formatos de audio digital

Sin importar el medio en que se almacene la información, es necesario que se haga bajo ciertas normas, estas normas compiladas es lo que se llama formato, el cual debe ser comprensible por el usuario y el software que lo utilice. En audio existen diversos formatos, pero se pueden clasificar a grandes rasgos en dos tipos según tengan o no compresión.

Los formatos sin compresión muestran la sucesión de datos PCM[1] tal cual se generaron en la conversión analógica-digital y se suele agregar en algunos casos un encabezamiento que proporciona datos como la resolución, el número de canales, cantidad de datos de audio, etc.

Si bien un archivo comprimido tiene un tamaño menor a uno que no lo es, es necesario un decodificador capaz de interpretar dicha compresión. Todo método de compresión de audio se apoya en la característica de que en cualquier señal de este tipo existe un grado considerable de redundancia.

[1] Modulación de código de pulsos

Es posible a su vez subdividir los formatos comprimidos en *lossy* (con pérdida) y *lossless* (sin pérdida), los primeros utilizan criterios psicoacústicos para eliminar o descartar información que para la mayoría de las personas resulta inaudible y los segundos se basan en consideraciones estadísticas y de teoría de la información.

Para fines metrológicos es necesario tener la mayor cantidad de información sobre los eventos grabados, por eso es recomendable utilizar formatos sin compresión como el WAV o con compresión como el FLAC (Free Losslesss Audio Codec) para el almacenamiento de las mediciones realizadas.

2.3.1 El formato WAVE

El formato WAV está dividido, como se observa en la figura 2.5, en fragmentos (en inglés *chunks*). Cada uno de estos fragmentos contiene cierto tipo de información, como el formato, los datos de audio y las marcas (*cues*). Cada uno está compuesto varias cadenas de caracteres y distintas series de datos numéricos alternados.

El archivo comienza con un encabezamiento que contiene la cadena de caracteres **“RIFF”** (formato de archivo para intercambio de recursos), seguido por 1 byte (equivalente a 8 bits) que define el tamaño del archivo.

Seguido aparece el fragmento correspondiente al formato, que comienza por la cadena de caracteres **“fmt”** y la cantidad de bytes que le siguen dentro del mismo. Luego le siguen los campos que se muestran a continuación:

Información	Bytes
Formato	2
Canales	2
Tasa de muestreo	4
Bytes por segundos	2
Bytes por muestra	2
Bits por muestra	2

Luego aparece el fragmento **“data”** (datos), seguido por la longitud total de bytes que ocupan y los datos de audio ordenados por canal.

Con estos fragmentos ya es posible generar un archivo de audio digital. Sin embargo, en la metrología y otras aplicaciones, resulta útil poder almacenar datos e información adicional que no necesariamente corresponden al audio. Esta información adicional es conocida en el formato WAV cómo *marcas* (cues), *labels* (etiquetas) y *notes* (anotaciones) relativas a eventos acústicos de interés, tales como identificación de fuentes, fecha y hora de la grabación, ubicación, etc. Se suele llamar *metadatos* (metadata), es decir datos que se refieren a datos, al conjunto total.

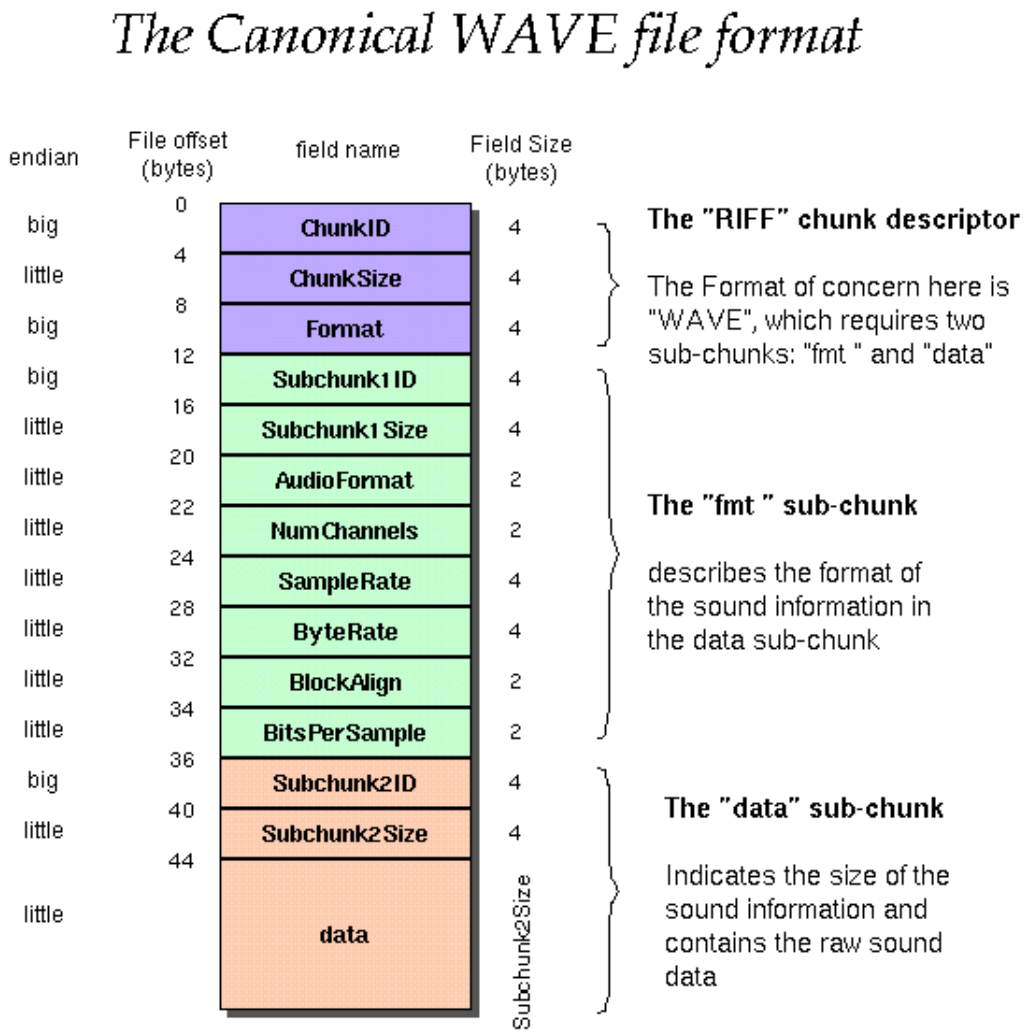


Figura 2.5: Composición de un archivo de formato WAVE [11].

2.4 Descomposición en frecuencias de una señal

Las señales que varían en el tiempo se pueden calificar en periódicas o aperiódicas, figuras 2.6 y 2.7 respectivamente. Una señal es periódica si

$$x(t) = x(t + T) \tag{2.3}$$

donde *t* es el instante de tiempo y *T* es el periodo. Si no se cumple esta igualdad estamos frente a una señal no periodica.

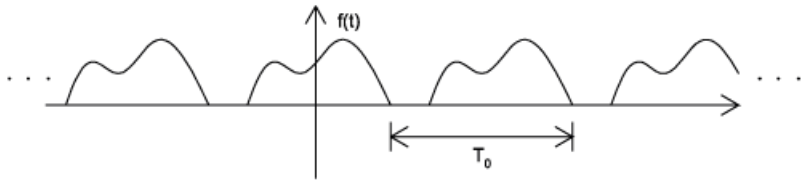


Figura 2.6: Señal periodica [12].

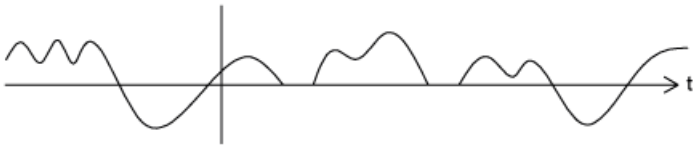


Figura 2.7: Señal no periodica [12].

En el caso de señales discretas se debe cumplir que

$$x(n) = x(n+N) \tag{2.4}$$

donde n es el número de muestra y N es equivale a $\frac{T}{T_s}$ (T período de la señal y T_s período de muestreo).

Las señales periódicas se pueden descomponer en una suma de señales senoidales, ecuación 2.6, como la que se observa en la Figura 2.8, definidas en tiempo como

$$x(t) = Asen(\omega x + \phi) \tag{2.5}$$

donde A es la amplitud, ω es la frecuencia angular equivalente a $2\pi f$ (frecuencia de la señal) y ϕ es el desfasaje.

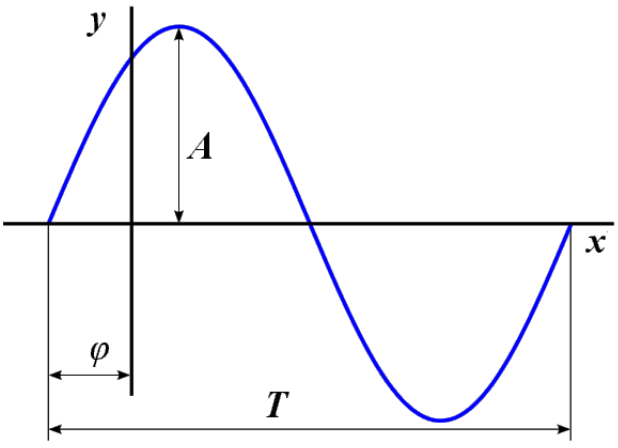


Figura 2.8: Señal senoidal.

Esto es consecuencia del *Teorema de Fourier*:

Cualquier función periódica de frecuencia *f* puede considerarse como la superposición de una serie de ondas senoidales de frecuencias *f*, 2*f*, 3*f*, 4*f*, etc.

$$x(t) = \sum_{n=1}^{\infty} X_n \sin(2\pi n f t + \phi_n) \tag{2.6}$$

de donde se obtienen los X_n que son llamados coeficientes de Fourier y se calculan de la siguiente manera

$$X_n = \sqrt[2]{A_n^2 + B_n^2} \tag{2.7}$$

donde

$$A_n = \frac{2}{T} \int_0^T x(t) \cos(n\omega t) dt \tag{2.8}$$

y

$$B_n = \frac{2}{T} \int_0^T x(t) \sin(n\omega t) dt \tag{2.9}$$

A partir de los X_n se puede dibujar el espectro de la señal, Figura 2.12, que es una gráfica donde el eje X son las frecuencias ($\frac{\omega}{2\pi}$) y el Y las amplitudes de los armónicos (los X_n).

Debido a que la mayoría de las señales no son periódicas, sino que varían de forma aleatoria, es necesario expandir el concepto de serie de Fourier. Para ello seleccionamos, o como se nombra en la mayor parte de la bibliografía correspondiente a este tema, *enventanamos* una porción de duración T de la señal a analizar y la repetimos en forma periódica cada T segundos. Ahora que tenemos una señal periódica podemos analizar su espectro, el problema es que sólo representa el espectro de esa porción, si tomamos un tiempo T mayor el espectro se vuelve más detallado y se reducen las amplitudes. Si quisiéramos hacer tender T a ∞, en el espectro veríamos que tanto la frecuencia fundamental como los coeficientes de Fourier tienden a 0. Se tiende entonces a un espectro continuo.

Cabe aclarar que también se suele utilizar la versión compleja de la seria de Fourier

$$x(t) = \sum_{n=-\infty}^{\infty} C_n e^{jn\omega t} \tag{2.10}$$

donde

$$C_n = \frac{1}{T} \int_{-T/2}^{T/2} x(t) e^{-jn\omega t} dt \tag{2.11}$$

2.4.1 Transformada de Fourier

Se define la transformada de Fourier, utilizable en señales continuas, como la siguiente igualdad

$$F(x(t)) = X(\omega) = \int_{-\infty}^{\infty} x(t)e^{-j\omega t}dt \tag{2.12}$$

a partir de $X(\omega)$ es posible obtener $x(t)$ utilizando la transformada inversa de Fourier

$$F^{-1}(X(\omega)) = x(t) = \frac{1}{2\pi}\int_{-\infty}^{\infty} X(\omega)e^{j\omega t}dw \tag{2.13}$$

2.4.2 Transformada discreta de Fourier (DFT)

Dada la señal discreta $x(k)$ se define la transformada discreta de Fourier cómo

$$F(x(k)) = X(m) = \sum_{k=0}^{N-1} x(k)e^{-j2\pi\frac{mk}{N}} \tag{2.14}$$

a partir de $X(m)$ es posible obtener $x(k)$ utilizando la transformada inversa discreta de Fourier

$$F^{-1}(X(m)) = x(k) = \frac{1}{N}\sum_{m=0}^{N-1} X(m)e^{j2\pi\frac{mk}{N}} \tag{2.15}$$

2.4.3 Transformada rápida de Fourier (FFT)

La transformada discreta de Fourier resulta una herramienta muy útil para el análisis espectral y para muchas otras aplicaciones. El problema es que al agrandar N es necesario realizar una gran cantidad de operaciones, a saber cada componente de la DFT requiere N multiplicaciones y N sumas, llegando por lo tanto a N^2 sumas y multiplicaciones.

El algoritmo denominado *Transformada rápida de Fourier*, desarrollado en 1965 por Cooley y Tuckey, logra reducir la cantidad de operaciones necesarias a $\frac{3}{2}N\log_2 N$ de punto flotante.

El uso de FFT presenta algunas limitaciones, primero suponer que se trata de una señal periódica de período N, o sea, que no se trata de la señal real, y segundo pueden aparecer descontinuidades que produzcan altas frecuencias de alias. El segundo problema se puede solucionar utilizando ventanas que disminuyan la interferencia entre cuadros.

2.5 Filtros digitales basados en FFT

Es un filtrado en el dominio frecuencial, es decir, se multiplica la transformada de Fourier de la señal ($X(k)$) por la ventana filtrante.

$$Y(k) = X(k)W(k) \tag{2.16}$$

donde $Y(k)$ es la transformada de Fourier de la señal filtrada, $X(k)$ de la señal original y $W(k)$ del filtro deseado. Luego se obtiene la señal filtrada a través de la transformada inversa de Fourier. Tal como se plantea, este filtro está limitado a una ventana de análisis de longitud N. Si bien es posible aumentar N, el procedimiento pierde eficiencia y aumenta innecesariamente la resolución del espectro. Para poder aplicar el método a señales largas se subdivide la señal en cuadros o

ventanas de longitud *N* con un solapamiento de *N/2*. Luego se obtiene la FFT de cada cuadro, se multiplica por la ventana filtrante y se le aplica la transformada inversa de Fourier.

En la Figura 2.9 se observa el diagrama en bloques de cómo se aplica un filtro FFT a una señal de audio.

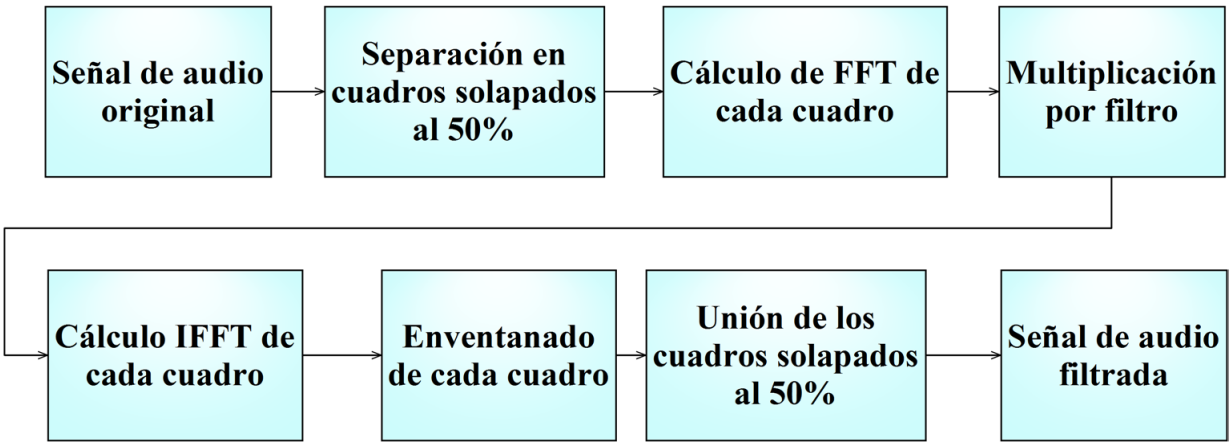


Figura 2.9: Diagrama en bloque de un filtro FFT.

2.6 Representacion de señales de audio digital

Existen diversas formas de representar una señal digital de audio, las más utilizadas y las que nos interesan entre ellas son:

- Forma de onda.

  Muestra la amplitud de la señal en cada instante de tiempo.

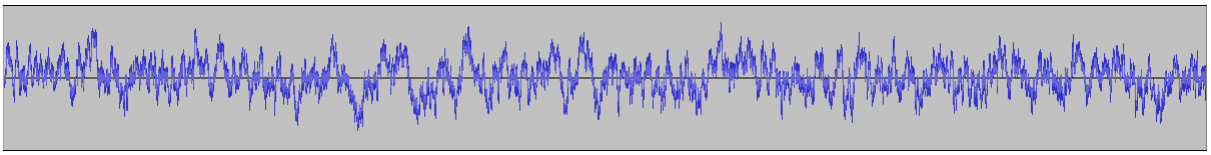


Figura 2.10: Representación de la forma de onda de una señal de audio.

- Espectrograma 2D.

  Se observa la descomposición frecuencial de la señal para fracciones contiguas de la señal, es decir, se calcula la descomposición para una cantidad determinada de muestras y se representa en el eje X el tiempo, en el Y la frecuencia y en una escala de colores la amplitud de cada armónico.

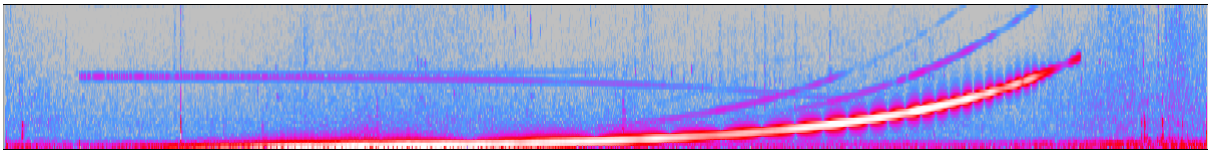


Figura 2.11: Representación del espectrograma de una señal de audio.

- Espectro.

  Permite obtener el promedio del análisis frecuencial a lo largo de toda la señal o de la selección.

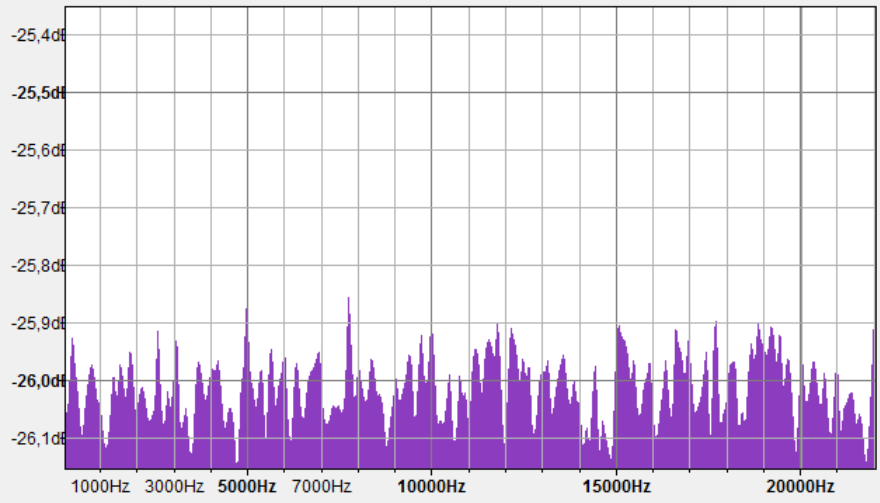


Figura 2.12: Representación del espectro de una señal de audio.

2.7 Mediciones Acústicas

La variable acústica que presenta mayor facilidad de medición es la presión sonora p. Se define cómo $p(t)$ a su valor instantáneo, y cabe aclarar que pocas veces es representativo debido a que dos sonidos similares tanto auditivamente como en sus efectos pueden tener valores instantáneos muy diferentes. Por lo tanto, y debido a que influye en el efecto de los sonidos, es más útil utilizar el promedio energético o valor eficaz calculado en un intervalo temporal T, definido como

$$P_{ef,T} = \sqrt{\frac{1}{T}\int_0^T p^2(t)dt} \tag{2.17}$$

Se suele utilizar también el promedio energético móvil, que varía a lo largo del tiempo

$$P_{ef,T}(t) = \sqrt{\frac{1}{T}\int_t^{t+T} p^2(t)dt} \tag{2.18}$$

Al graficar el promedio energético móvil se obtiene la envolvente energética de la señal. Para lograr la mejor representación, se debe seleccionar un periodo T acorde a la rapidez de variación

de la señal, si la señal varía rápido corresponderá un valor de T chico y viceversa. Es posible obtener la envolvente utilizando un filtro pasa bajos como se muestra en la ecuación siguiente

$$P_\tau(t) = \sqrt{\frac{1}{\tau}\int_0^t p^2(\theta) e^{-\frac{t-\theta}{\tau}} dt} \tag{2.19}$$

Este promedio contiene una ponderación exponencial, de manera que los valores de $\theta \approx t$ son priorizados.

La variable τ recibe el nombre de ponderación temporal y se le suelen asignar uno de los tres valores normalizados τ= 1 s (lenta/slow), τ= 0.125 s (rápida/fast) y τ= 0.035 s (impulsiva/impulsive).

2.7.1 Nivel de presión sonora

Debido a que el rango dinámico de la presión sonora de los sonidos audibles es muy grande, se utiliza el nivel de presión sonora, $L_{p,T}$, definido como:

$$L_{p,T} = 20 \log \frac{P_\tau}{P_{ref}} \tag{2.20}$$

donde P_{ref} = 20 μPa es un valor de referencia que corresponde aproximadamente al umbral auditivo a 1000 Hz. El nivel de presión sonora se expresa en decibeles (dB) y queda definido entonces entre 0 dB y 120 dB.

2.7.2 Nivel de presión sonora equivalente

Se define el nivel de presión equivalente, $L_{eq,T}$, también llamado nivel de presión sonora continuo equivalente, a un nivel de presión sonora que se supone constante durante un periodo *T* que tiene el mismo valor de energía total que la presión sonora variable del sonido a medir durante ese tiempo. Se obtiene utilizando la ecuación:

$$L_{eq,T} = 20 \log \frac{\sqrt{\frac{1}{T}\int_0^T p^2(t) dt}}{P_{ref}} \tag{2.21}$$

2.7.3 Nivel de presión sonora ponderado

Debido a que muchas de las aplicaciones en acústica se vinculan con la percepción y los efectos del ruido sobre el ser humano, se utilizan ponderaciones o compensaciones frecuenciales. El propósito de esto es tener en cuenta que la sensibilidad auditiva varía con la frecuencia. Se utilizan para eso distintas curvas, las primeras de ellas fueron las denominadas A, B y C, que fueron destinadas a niveles cercanos a 40 dB, 70 dB y 100 dB respectivamente, y de las cuales hoy en día sólo se utilizan la A y C para cualquier rango de nivel.

Luego se agregaron las ponderaciones D, para ruido de aviación, y la G, para ruidos de muy baja frecuencia, actualmente se encuentra en reglamentación la ponderación K desarrollada por la Unión Europea de Radiodifusión que busca normalizar, utilizando los descriptores ‘Loudness

Range' y 'Maximun True Peak Level', la sonoridad y el nivel máximo permitido en señales de audio utilizadas en la producción, distribución y transmisión de programas de radiodifusión.

Vemos en las Figuras 2.13, 2.14 y 2.15 las curvas de las distintas ponderaciones frecuenciales.

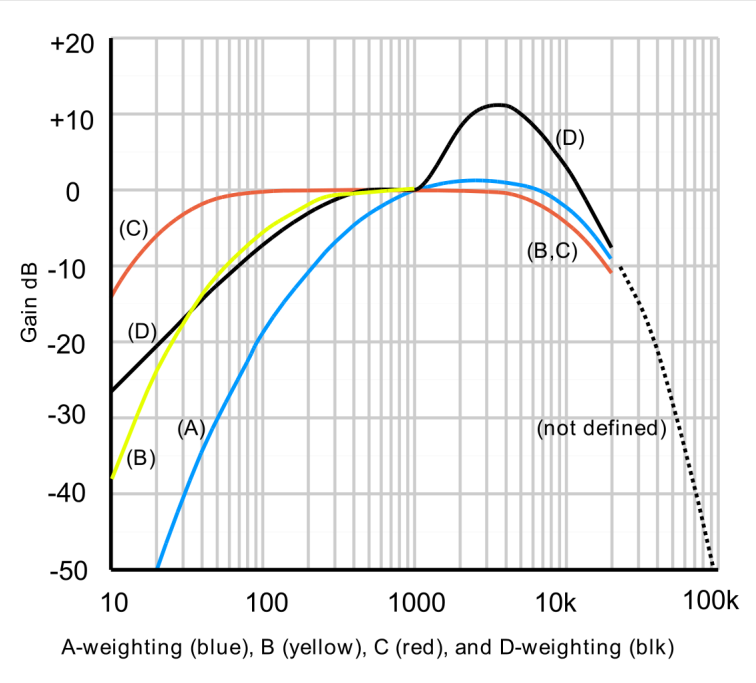


Figura 2.13: Curvas de ponderación frecuencial A, B, C y D [13].

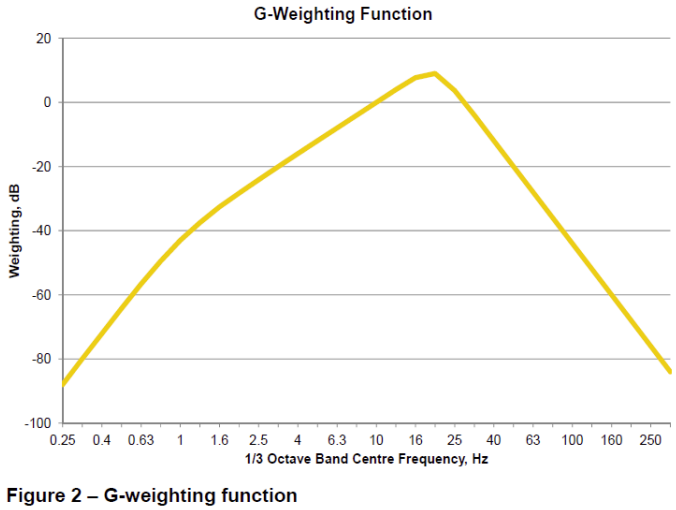


Figure 2 – G-weighting function

Figura 2.14: Curva perteneciente a la ponderación frecuencial G [13].

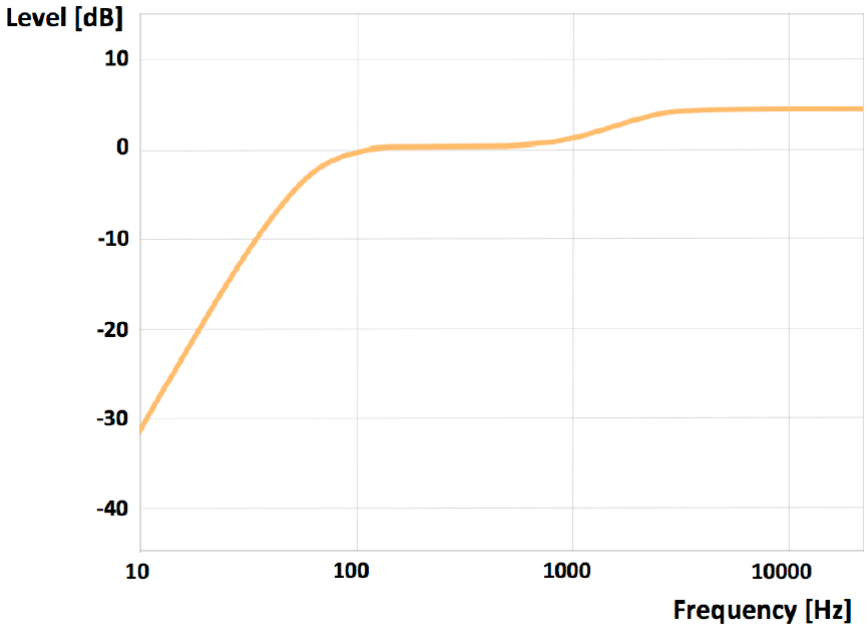


Figura 2.15: Curva de la ponderación K [14].

2.7.4 Niveles estadísticos

Dada una ponderación temporal con un τ definido y una ponderación frecuencial X (donde X puede ser A, B, C, D, Z o cualquier otra) se define al nivel estadístico L_{XN} como aquel valor que superado en un N % del tiempo.

Estos niveles son muy útiles para ser aplicados en ciertos criterios, como la determinación del ruido de fondo y su nivel o clasificación según su procedencia, por ejemplo el nivel estadístico L_{X90} puede asignarse como nivel de ruido y el L_{X95} designarse como nivel proveniente de fuentes lejanas.

2.8 Software libre

Se llama de esta manera a un software que le da al usuario la libertad de copiarlo, distribuirlo, editarlo y mejorarlo. Ofrece la posibilidad de adaptar el software a gusto y poder saber qué es lo que está haciendo en cada instante.

Existen cuatro libertades escenciales que indican si un software es libre o no:

- Libertad 0: la libertad de ejecutar el programa con cualquier propósito.
- Libertad 1: la libertad de estudiar cómo funciona el programa, y cambiarlo para que haga lo que el usuario quiera.
- Libertad 2: la libertad de redistribuir copias para ayudar al prójimo.

- Libertad 3: la libertad de distribuir copias de las versiones modificadas a terceros, de modo que la comunidad se beneficie.

Cabe aclarar que el decir que un software sea libre no siempre quiere decir que sea gratuito. Uno puede no sólo haber pagado o no por él sino también, distribuirlo de manera gratuita o paga según desee. En el último de los casos se suele agregar otros servicios, por ejemplo mantenimiento, actualización, ayuda técnica, manuales impresos, distribución en un medio físico, etc., pero no se prohíbe, persigue ni castiga su distribución gratuita por parte de quien lo reciba.

2.9 Audacity

Audacity es un editor de grabación y edición de audio libre, de código abierto y multiplataforma[2]. En la Figura 2.16 se observa la pantalla principal del programa.

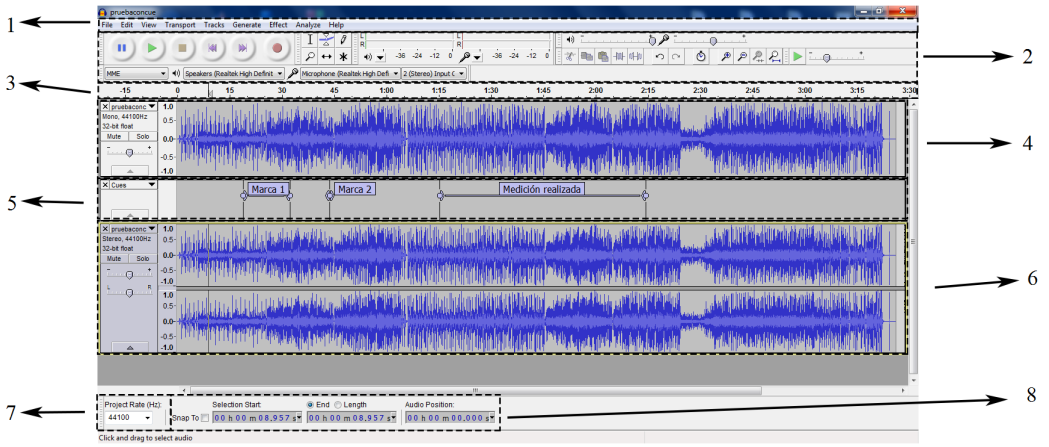


Figura 2.16: Pantalla principal de Audacity.

Dentro de la pantalla podemos encontrar las distintas zonas numeradas, que se conocen de la siguiente forma

1. Barra de menú.
2. Botonera.
3. Regla temporal.
4. Pista de audio mono.
5. Pista de etiquetas.

[2]Significa que se puede utilizar en distintos sistemas operativos como Windows, Linux y MacOS.

6. Pista de audio estéreo.
7. Frecuencia de muestreo del proyecto.
8. Descripción temporal (tiempo de inicio y final de una selección, posición del cursor de reproducción, etc.)

Todos los tipos de pista se ubican en lo que se define como *Track Panel* y se encuentran compuestas por un recuadro con información de la pista, una regla de amplitud y una representación de la señal (permite seleccionar entre oscilograma, espectrograma o nivel de presión sonora, la última de estas fue agregada como parte de este proyecto y no está disponible en la versión original del programa). En el caso de las pistas estéreo existen dos gráficas de la señal (una para cada canal) como se ve en la Figura 2.17.

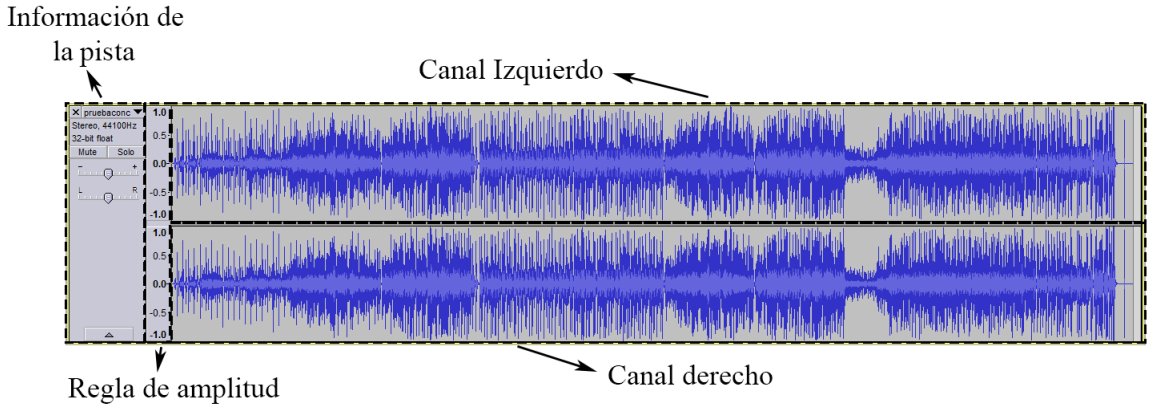


Figura 2.17: Pista estéreo.

Dentro de la barra de herramientas encontramos diferentes menús y cada uno de ellos se encarga de subdividirlas según su funcionalidad

- File: Manejo de archivos.
- Edit: Edición tanto de señales como de preferencias.
- View: Se encarga de la parte visual como ser el zoom de la señal.
- Transport: Herramientas de reproducción tales como reproducir, parar, pausar, etc..
- Tracks: Permite crear, alinear, eliminar, copiar y manipular las pistas.
- Generate: Creación de pistas de ruido[3], tonos[4], chirp[5], etc.

[3] Señal que contiene componentes frecuenciales en todo el espectro audible.
[4] Señal compuesta por una onda senoidal de una determinada frecuencia y amplitud.
[5] Señal senoidal que varía su frecuencia y amplitud de manera constante a lo largo del tiempo.

- Effect: Aplicación de efectos a las señales.
- Analyze: Análisis de las señales.
- Help: Ayuda, manuales, log, etc.

De estos menús hay algunos sobre los cuales vale la pena hacer una explicación más profunda,debido a las modificaciones realizadas y herramientas creadas que se encuentran en ellos, estos son: *File*, *Effect* y *Analyze*. El primero presenta la opción de importar archivos del tipo WAV al proyecto, lo cual nos permite agregar la importación de los *metadatos* y crear nuevas pistas de etiquetas con ellos. Por otra parte los Plug-ins creados se pueden dividir en *efectos* o *herramientas de análisis*, la diferencia entre ellos es que el primero modifica la señal original y el segundo obtiene información sin realizar modificaciones.

Capítulo 3

Desarrollo

Para lograr cada uno de los objetivos fue necesario estudiar los distintos lenguajes de programación utilizados, como C y C++, no sólo para poder entender los archivos pertenecientes al código fuente sino también para poder escribir o desarrollar nuevos programas y ponerlos a prueba a la hora de realizar la depuración de ambos (cabe destacar que se deben seguir pautas definidas por los creadores del software que se encuentran en la página oficial destinada a nuevos desarrolladores de Audacity [6]). Si bien esto se llevó a cabo durante los primeros meses, se puede decir que es una tarea que no termina, ya que uno nunca deja de adquirir conocimientos a la hora de decodificar y redactar códigos funcionales.

El desarrollo de interfaces gráficas se realizó utilizando la librería desarrollada bajo el lenguaje C++ llamada **wxWidgets**, la cual posee su propio código, que también fue estudiado y puesto en práctica a lo largo de todos los trabajos realizados debido a que Audacity lo utiliza para su interfaz gráfica.

Para el desarrollo de códigos se utilizó el programa Microsoft Visual C++ 2008 Express Edition, sugerido por los mismos desarrolladores de Audacity (actualmente se está realizando un transpaso a la versión de Microsoft Visual C++ 2013). Una vez instalado, se prosiguió a agregar y compilar las librerías necesarias, FFTW (permite realizar transformadas de Fourier) y wxWidgets.

La tarea de *decodificación del código fuente y su documentación* es algo que se llevó a cabo a lo largo de todo el proyecto, ya que cualquier avance requería un entendimiento del funcionamiento interno del programa y de la necesidad o no de modificarlo para adaptarlo al trabajo realizado.

A continuación se desarrollan los trabajos realizados a lo largo del proyecto y las aplicaciones de cada uno de ellos.

Capítulo 4
Modificaciones del código fuente

Lo que se buscó dentro de esta tarea fue adaptar herramientas existentes del programa al funcionamiento deseado a la hora de utilizarlas en tareas de investigación.

Para ello se necesitó encontrar dentro del código fuente (ver sección B.1) los archivos que intervienen en las aplicaciónes a modificar a la hora de ser ejecutadas. Dado que el código es extenso se debió recurrir a distintas formas de búsqueda. La principal provino de la creación, mediante el uso del programa Doxygen[8], de un archivo con el formato de una página web que permite encontrar las relaciones entre todos los archivos del código fuente utilizando una red conceptual. En la Figura 4.1 se muestra un ejemplo utilizando el archivo Envelope.h como referencia. Además se utilizó la herramienta de búsqueda perteneciente al programa Visual Studio C++ Express, que permite ubicar una palabra o frases sin importar en que parte del código fuente se encuentre.

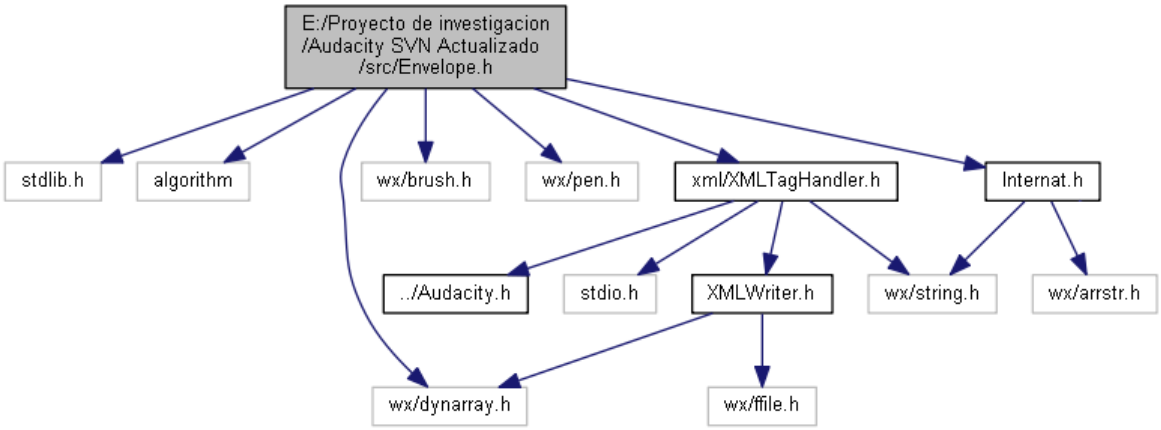


Figura 4.1: Red obtenida para el archivo Envelope.h utilizando Doxygen

4.1 Visor de espectrograma

El visor de pistas (tracks) de Audacity permite elegir entre cinco distintas visualizaciones WaveTrack, WaveTrack (dB), Espectrograma, Espectrograma (dB) y Pitch. En esta parte del trabajo nos centraremos en el espectrograma que, como se explicó anteriormente, muestra de qué forma varía el espectro de la señal a lo largo del tiempo. Se puede ver en las Figuras 4.2, 4.3 y 4.4 el menú de configuración, el espectrograma de una señal obtenido con la escala de colores originales y el que se obtiene utilizando la escala de grises.

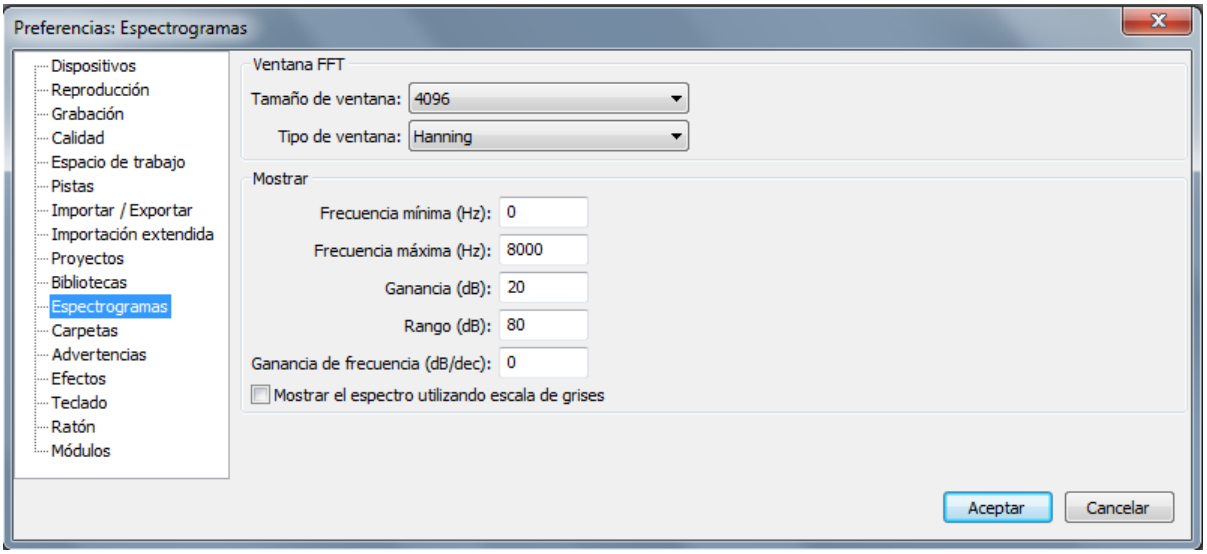


Figura 4.2: Menú original de preferencias de espectrogramas de Audacity.

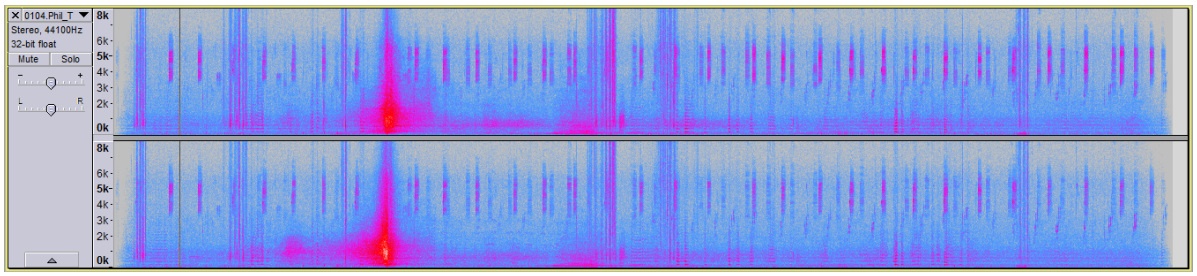


Figura 4.3: Espectrograma de una señal utilizando la escala de colores original de Audacity.

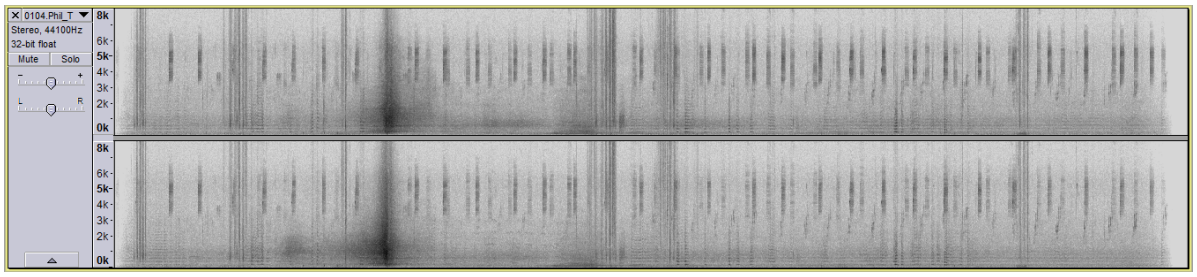


Figura 4.4: Espectrograma de una señal utilizando la escala de grises.

Dentro de las preferencias del espectrograma se puede seleccionar el tamaño y el tipo de ventana. El tamaño define la cantidad de muestras que se toman para calcular el espectro de una banda temporal, además, este valor delimita la resolución temporal y la frecuencial según las siguientes fórmulas

$$\delta T = \frac{W_s}{F_s} \tag{4.1}$$

$$\delta F = \frac{F_s}{W_s} \tag{4.2}$$

donde W_s es el tamaño de la ventana en muestras y F_s es la frecuencia de muestreo (por ejemplo si F_s = 44100 Hz y W_s = 4096 se tendrá una resolución temporal de 0,093 s y una resolución frecuencial de 10,72 Hz). El tipo de ventana, por otro lado, elimina o disminuye el ruido de alta frecuencia que aparece por las discontinuidades entre cuadros consecutivos de la señal.

Lo que se deseaba modificar era la escala de colores con la que se grafica el espectrograma. Para ello se debió analizar el código fuente de Audacity con el fin de encontrar cuáles archivos intervienen en el dibujo y luego obtener mediante ingeniería inversa, las distintas escalas utilizadas por otros softwares de edición de audio a la hora de dibujar espectrogramas. El utilizar escalas de colores similares a las utilizadas en otros programas permite:

1. Interoperabilidad con otros softwares, permitiendo una adaptación e incluso migración al Audacity sin perder lo ya acostumbrado de otros softwares.

2. Compartir espectrogramas con usuarios de otros programas.

3. Adaptar la escala de colores al gusto personal de cada uno.

Luego de analizar el código fuente se encontró que el programa parte de cinco colores definidos a los que les asigna un valor que va de 0 a 1 con saltos de 0,25 y luego genera la escala a través de un gradiente de 256 puntos entre dos colores definidos consecutivos. Una vez encontrado esto se procedió a generar un archivo que contuviera componentes en frecuencia de distintas amplitudes (las 5 necesarias para generar el gradiente), obtener los valores RGB (red, green y blue) que los programas les asignan y modificar el código fuente para que existiera la opción de elegir entre los distintos mapas, obteniendo así el menú de preferencias que se puede observar en la Figura 4.5 y los distintos espectrogramas representados en las Figuras 4.6, 4.7, 4.8 y 4.9.

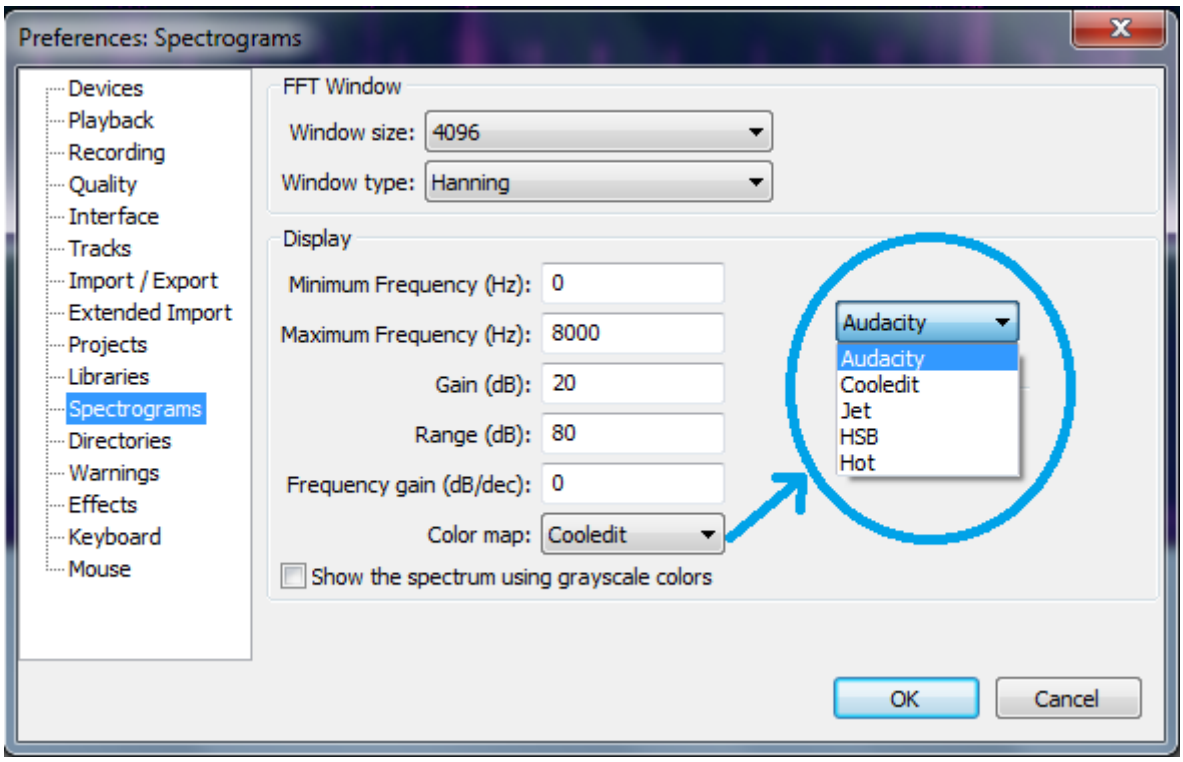


Figura 4.5: Menú de preferencias modificado y vista de la lista desplegable.

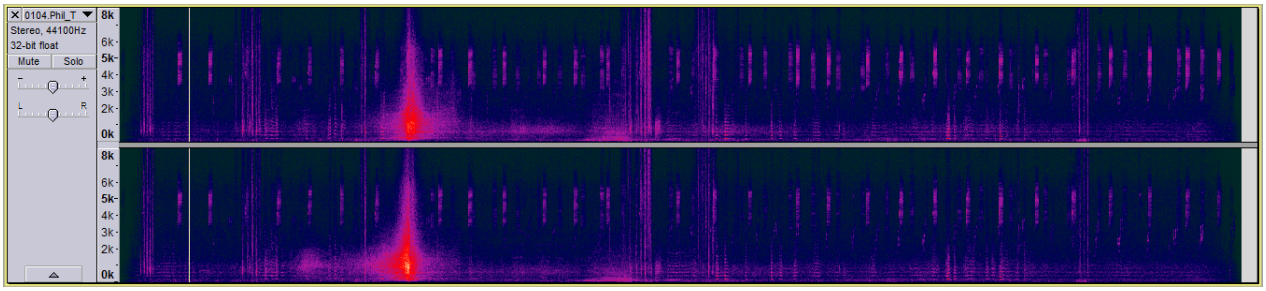


Figura 4.6: Escala de colores 1.

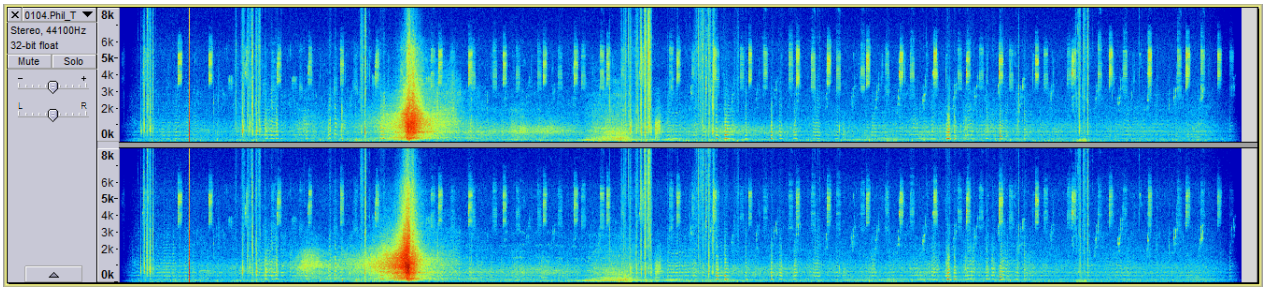


Figura 4.7: Escala de colores 2.

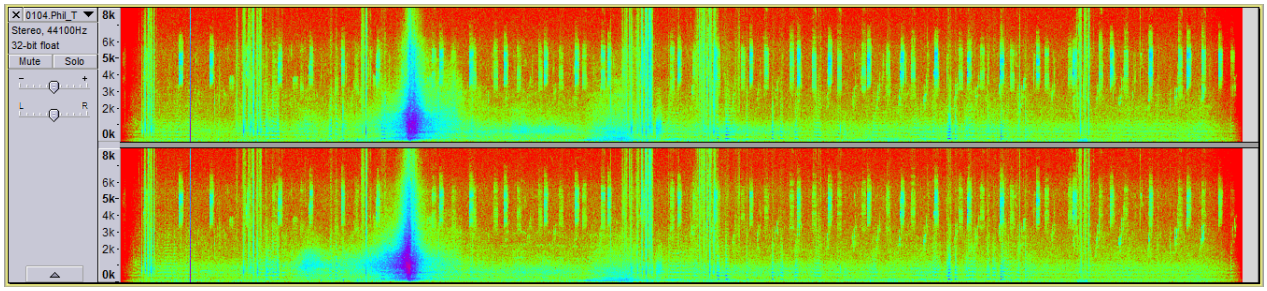


Figura 4.8: Escala de colores 3.

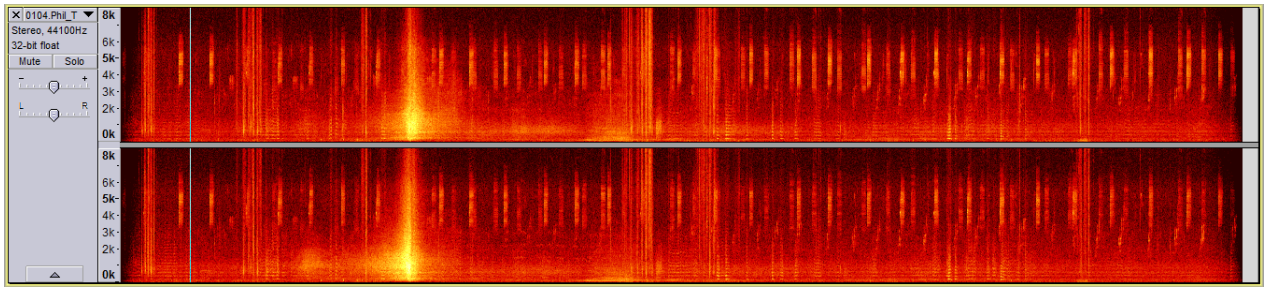


Figura 4.9: Escala de colores 4.

4.2 Analizador de espectro

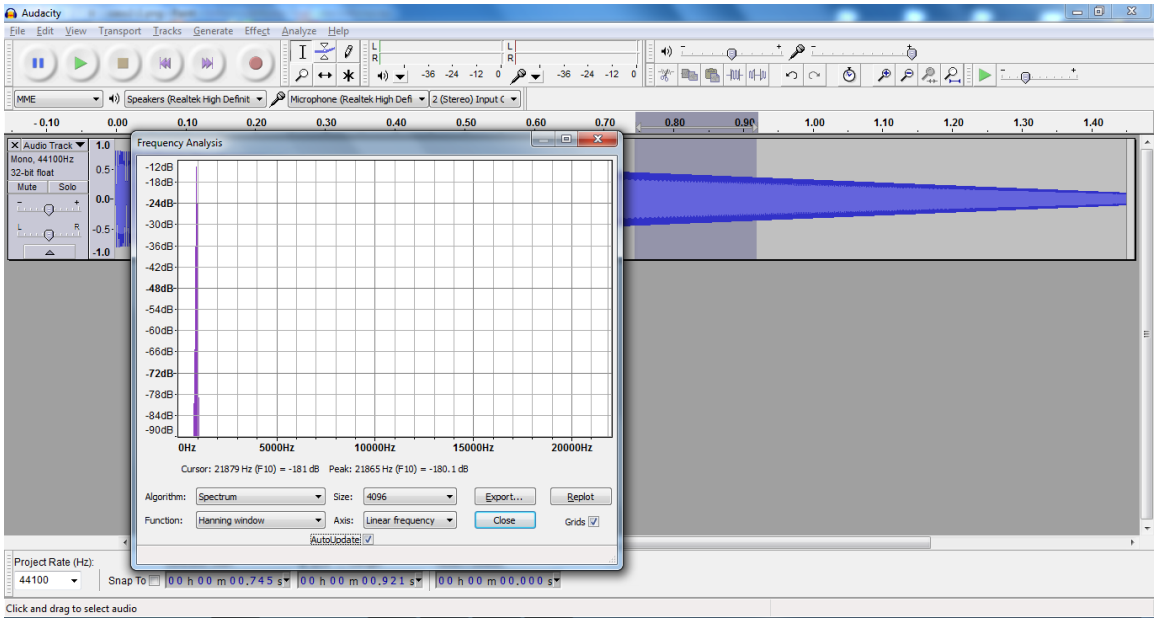
Audacity posee una herramienta llamada *Analizador de Espectro* que calcula y muestra el valor promedio de las componentes frecuenciales a lo largo de toda la pista de audio o de una porción de ella. El problema de esta herramienta es que tiene un límite superior y uno inferior en cuanto a la cantidad de muestras que se pueden seleccionar para realizar el cálculo. El límite superior viene dado por una cantidad de muestras determinada por los desarrolladores del programa para evitar la carga computacional (cosa que hoy en día con las computadoras actuales no es necesario hacer) y el inferior se debe a que se necesita al menos una selección del tamaño de una ventana de análisis para calcular el espectro.

Para solucionar el primer inconveniente se buscó en qué punto el programa tomaba la decisión de acotar la cantidad de muestras y se removió esa limitación. Para quitar el tope inferior se debió estudiar cual sería el comportamiento óptimo y se llegó a la conclusión de que existian distintas situaciones a las cuales el programa debía responder de una forma determinada.

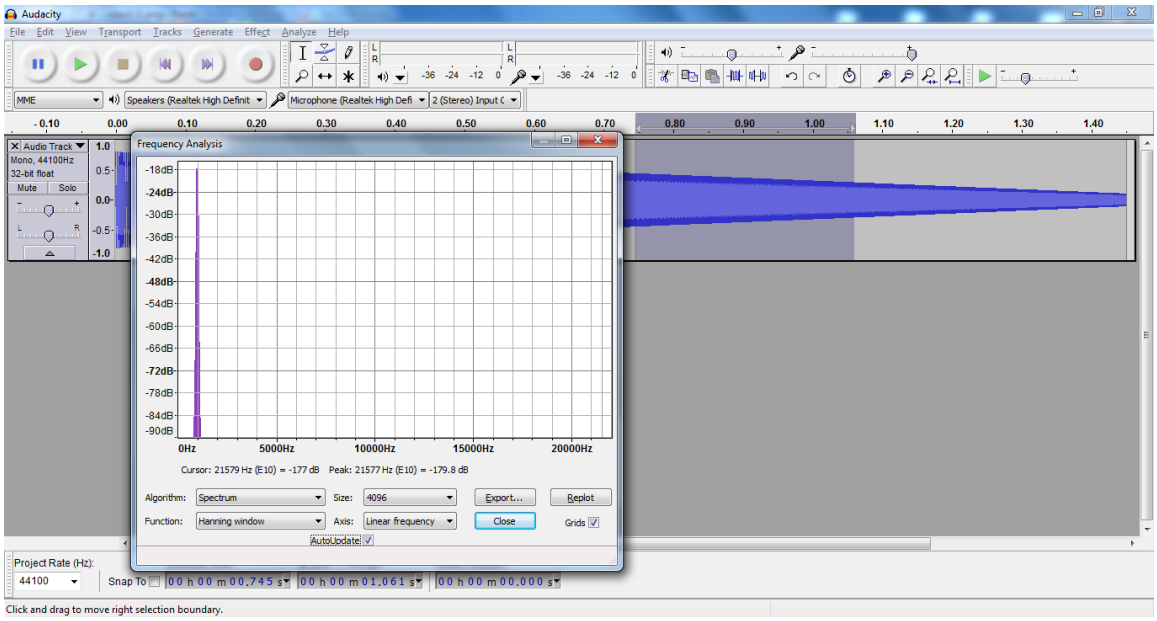
Además se vio como algo útil dar la opción de que la ventana se actualizara automáticamente cada 0,5 segundos, de esta manera el usuario puede ir modificando su selección y ver de qué forma varía el espectro de la señal.

A continuación se muestran las diferentes situaciones que se pueden dar y de qué manera responde el programa:

- Si la cantidad de datos seleccionada es mayor que el tamaño de la ventana (Figura 4.10a), mientras la selección varía se actualiza la gráfica (Figura 4.10b).

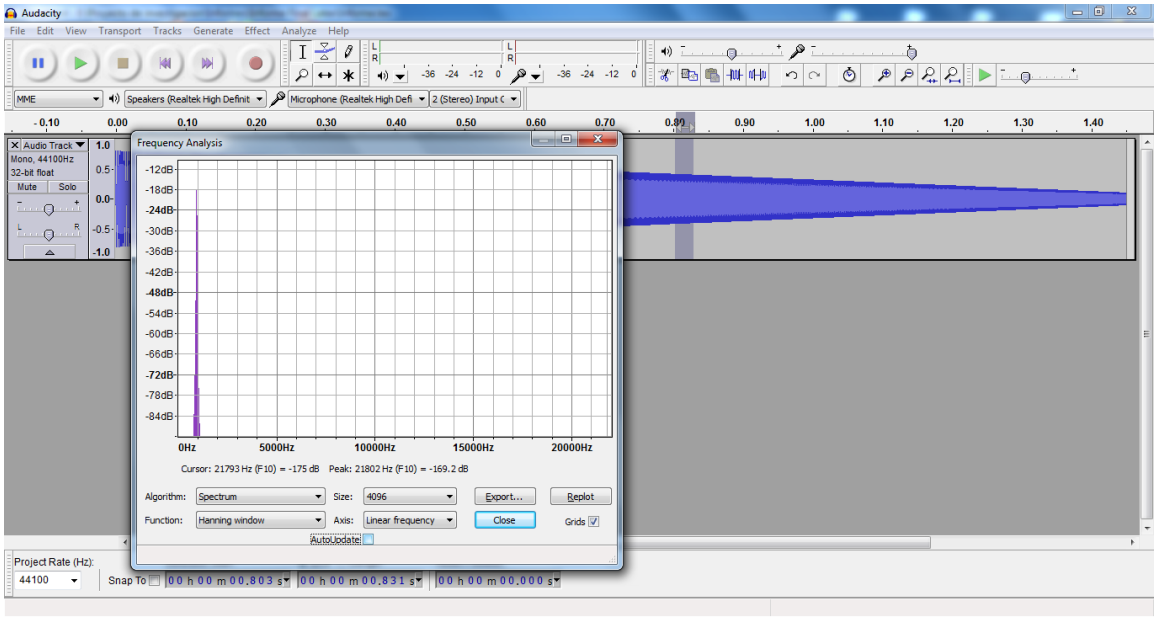


(a)

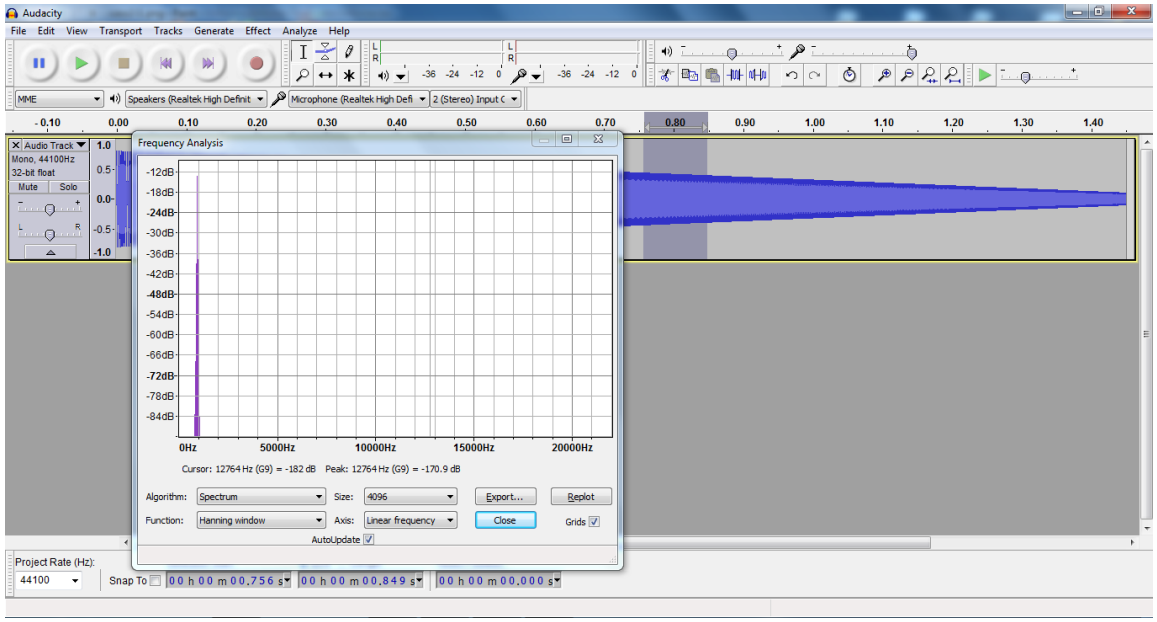


(b)

- Si la cantidad seleccionada es menor que el tamaño de la ventana (Figura 4.11a), se calcula el punto medio y se centra en él, la ventana de selección (Figura 4.11b).

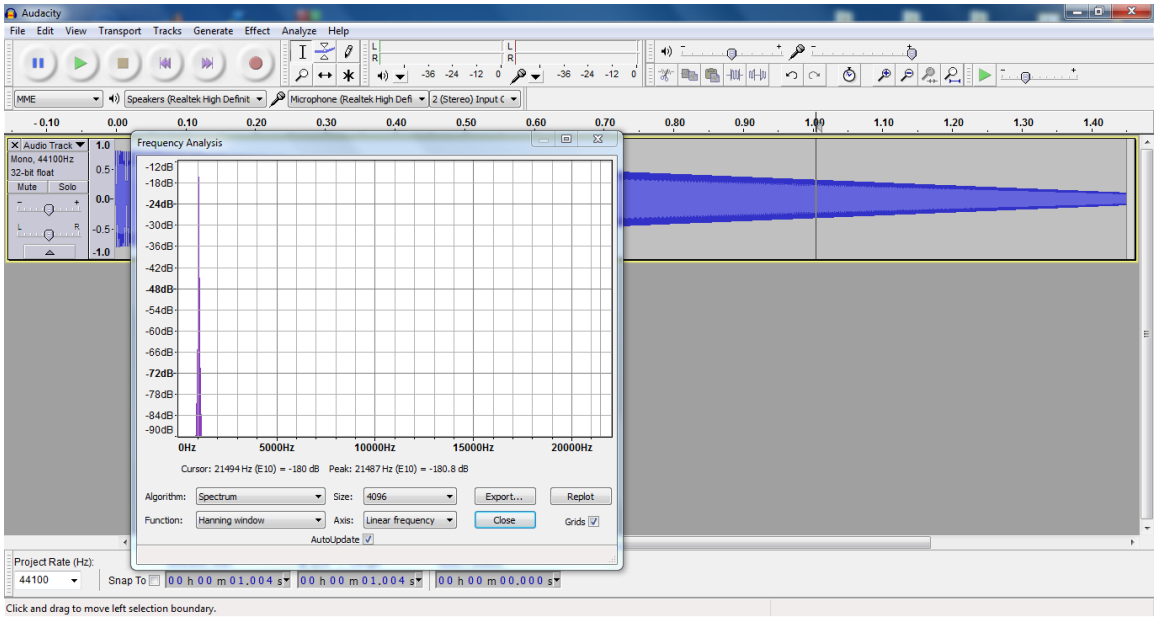


(a)

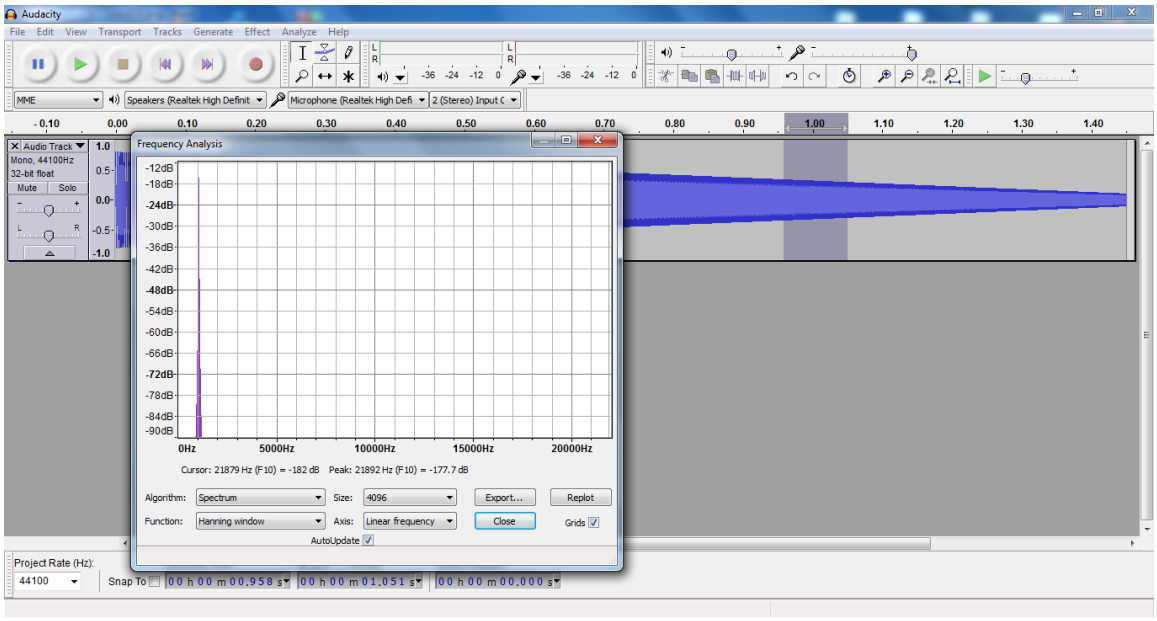


(b)

- Si se selecciona sólo 1 dato dato de la señal (Figura 4.12a), la ventana se centra en ese punto (Figura 4.12b).

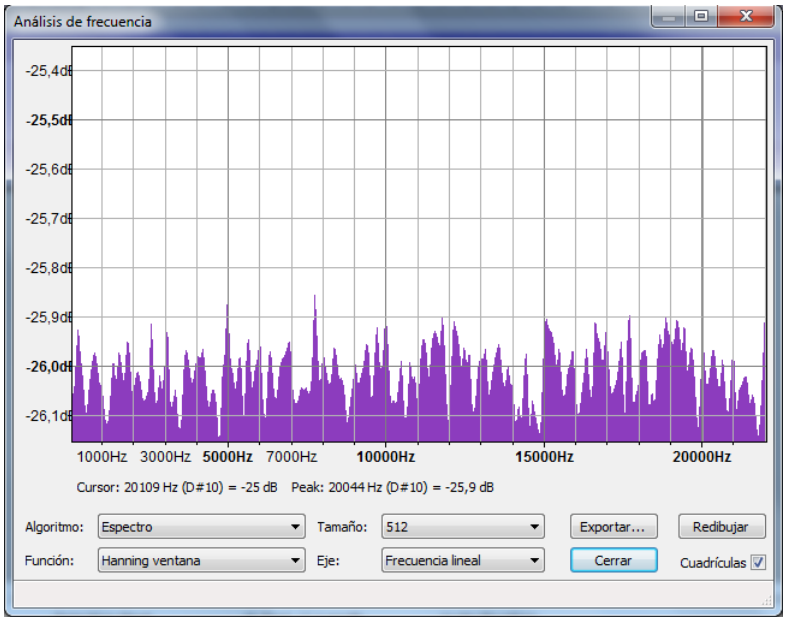


(a)

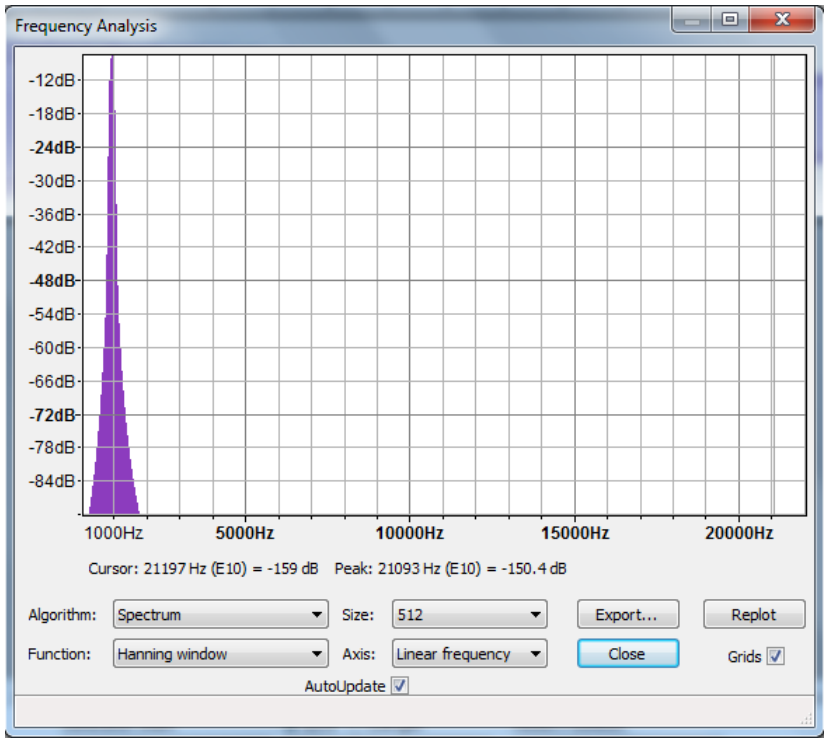


(b)

En las figuras 4.13a y 4.13b, se observan el menú orignal y el modificado, respectivamente. En el segundo se puede ver en la parte media inferior el botón que habilita o inhabilita el *AutoUpdate*.



(a)



(b)

La modificación realizada ayuda a darle al programa

1. Compatibilidad con otros softwares.
2. Mayor precisión en la determinación del lugar donde se desea obtener el espectro.

En la sección A.1 se puede ver la función añadida al código fuente permite actualizar el espectro según la selección y en la Figura 4.14 se observa el diagrama en bloques del funcionamiento del analizador de espectro modificado.

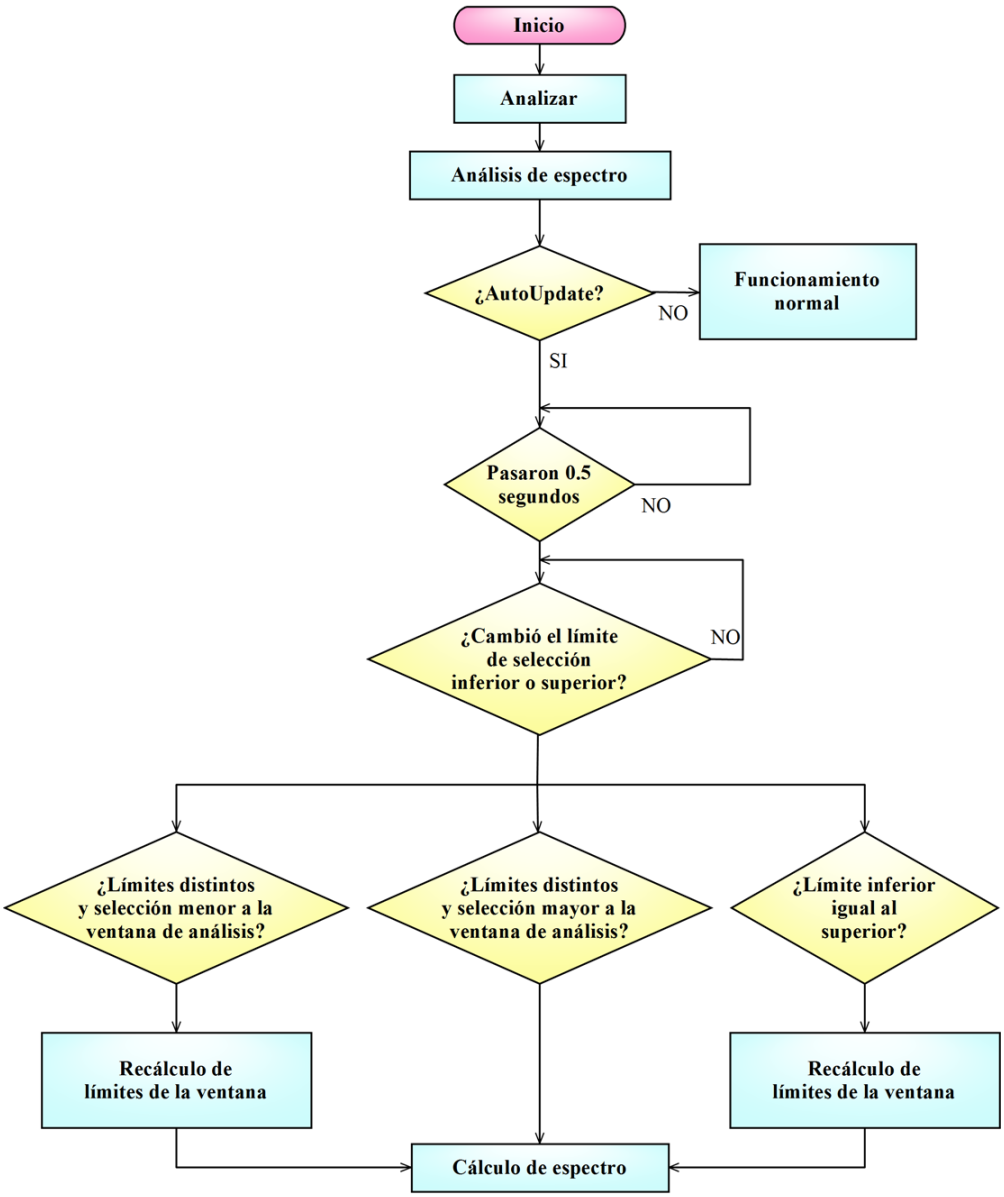


Figura 4.14: Diagrama en bloques del funcionamiento interno del analizador de espectro.

4.3 Visor de nivel

La posibilidad de observar el nivel sonoro de una señal es fundamental a la hora de analizar la misma, ya que permite

1. Mediciones acústicas sin necesidad de hacer cálculos externos.
2. Visualizar la evolución del nivel a lo largo del tiempo, lo que permite detectar eventos específicos.
3. Compararación de diversas ponderaciones frecuenciales y temporales.

Para obtener dicha curva es necesario aplicar diversos algoritmos a la señal, se muestra a continuación un diagrama en bloques donde se ve paso a paso las modificaciones que sufre la onda.

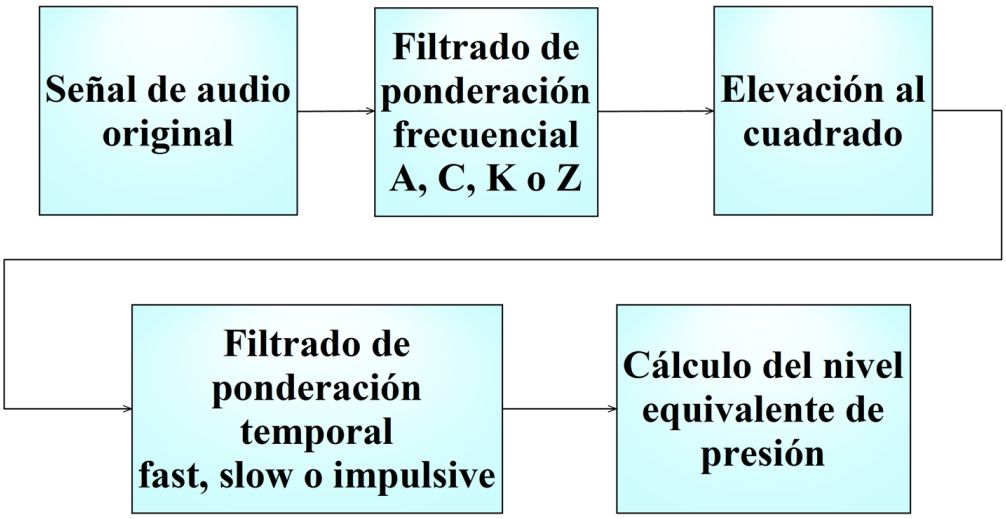


Figura 4.15: Diagrama en bloques del cálculo de nivel de presión sonora equivalente de una señal.

Cómo se explicó anteriormente se aplican dos tipos de ponderaciones a la hora de realizar el cálculo (pasos 2 y 4 del digrama en bloques), una frecuencial y una temporal. La primera busca tener en cuenta que la sensibilidad auditiva varía según la frecuencia y la segunda define el tiempo de integración utilizado para obtener el promedio energético. Para poder definir ambos valores se agregó al menú de preferencias el apartado Level, como se puede observar en la Figura 4.16

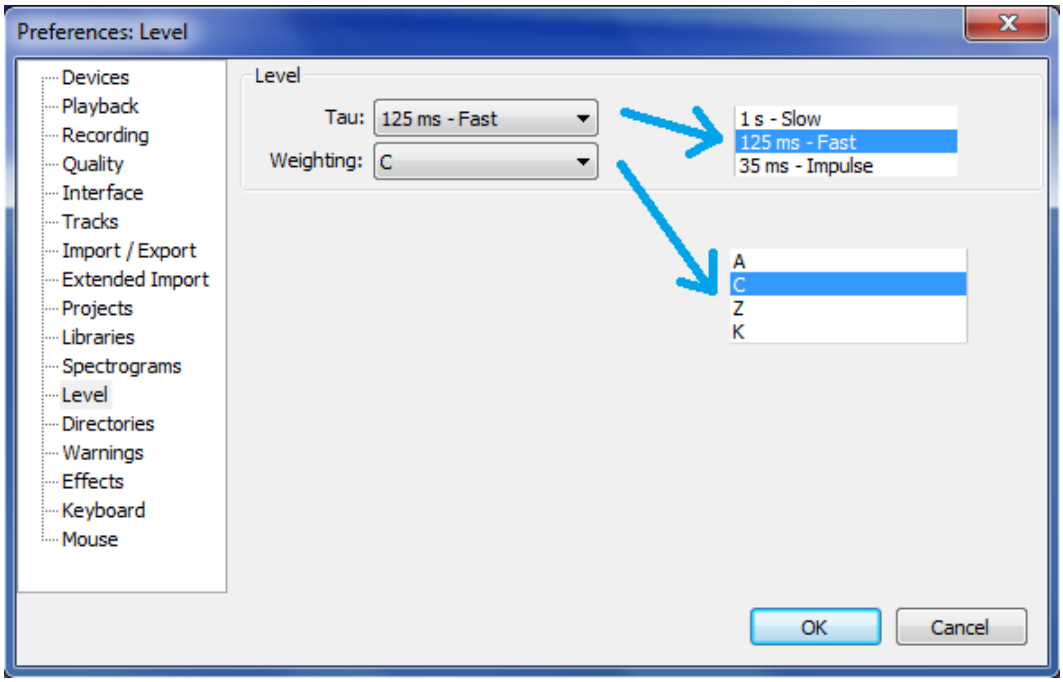


Figura 4.16: Menú de preferencias del visor de nivel.

Además se añadió al TrackMenue (menú de la pista), ver Figura 4.18, la opción para seleccionar la visualización del nivel (de la misma manera que se puede elegir entre ver las distintas representaciones de la señal), y la posibilidad de definir el valor de fondo de escala o asignar un valor conocido de nivel a una porción de la pista (por ejemplo, en el caso de tener tono de calibración) utilizando el menú que se ve en la Figura 4.17 y de esta manera realizar un corrimiento en el eje de amplitud. Las Figuras 4.19, 4.20 y 4.21 muestran cómo se ve la señal orignal, su nivel y de que manera se aplica el corrimiento luego de seleccionar el tono de calibración y asignarle el valor de 94 dB.

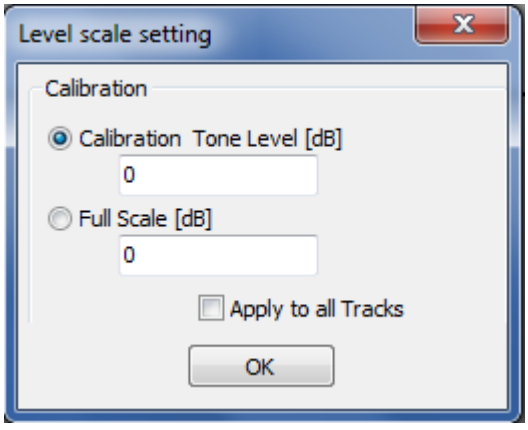


Figura 4.17: Menú de definición del tono de calibración o el fondo de escala.

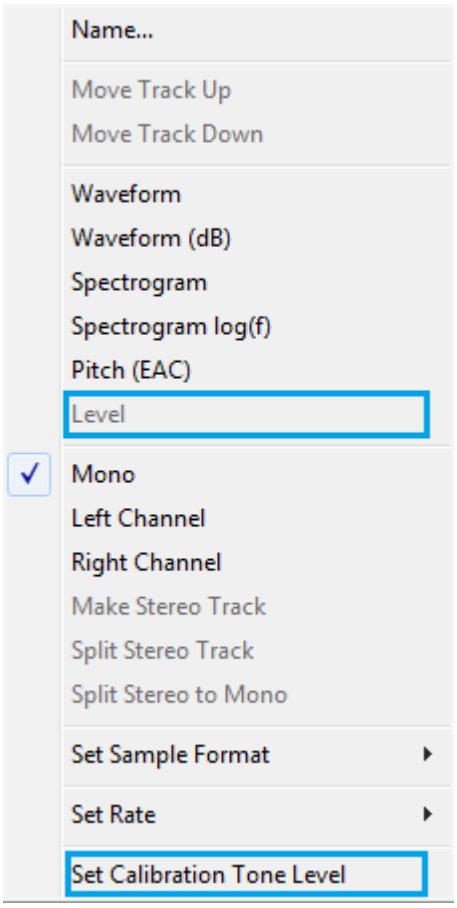


Figura 4.18: Track Menue modificado.

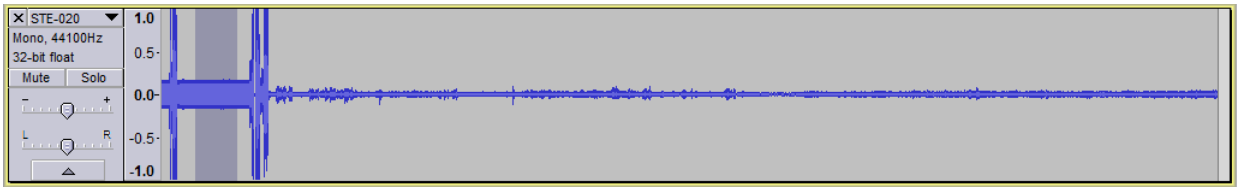


Figura 4.19: Señal original.

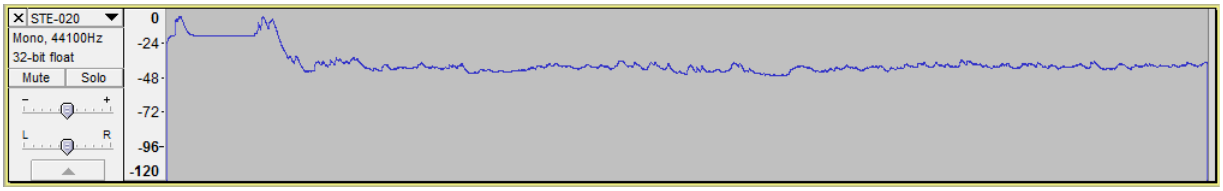


Figura 4.20: Vista del nivel de la señal de la figura 4.19.

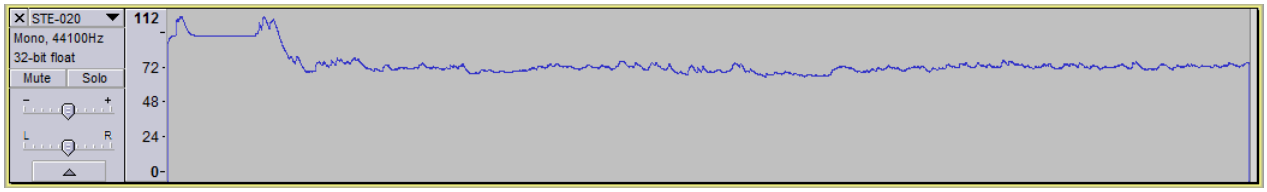


Figura 4.21: Vista del nivel luego de asignarle el valor del tono de calibración seleccionado en la Figura 4.19.

Se puede observa en las Figuras 4.22, 4.23, 4.24 y 4.25 las distintas ponderaciones frecuenciales aplicadas a una señal del tipo chirp entre 20 Hz y 22 KHz con τ = 1 segundo y en las Figuras 4.26 y 4.27 el nivel de una pista con ponderación frecuencial Z con τ = 1 segundo y τ = 0,125 segundos respectivamente.

Figura 4.22: Vista del nivel de una chirp de 20 Hz a 22 KHz con ponderación Z.

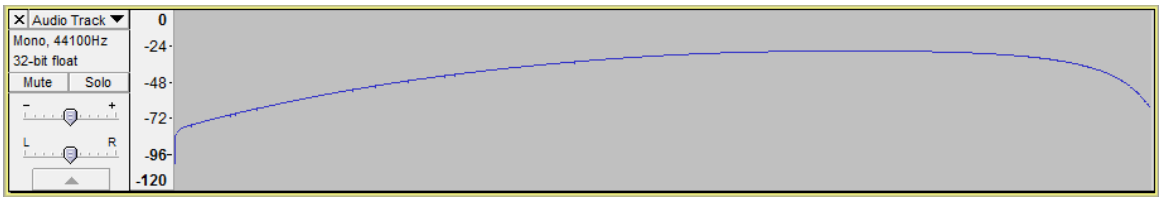


Figura 4.23: Vista del nivel de una chirp de 20 Hz a 22 KHz con ponderación A.

Figura 4.24: Vista del nivel de una chirp de 20 Hz a 22 KHz con ponderación C.

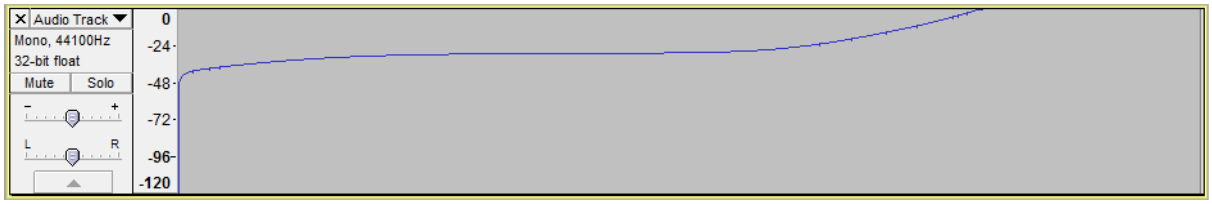


Figura 4.25: Vista del nivel de una chirp de 20 Hz a 22 KHz con ponderación K.

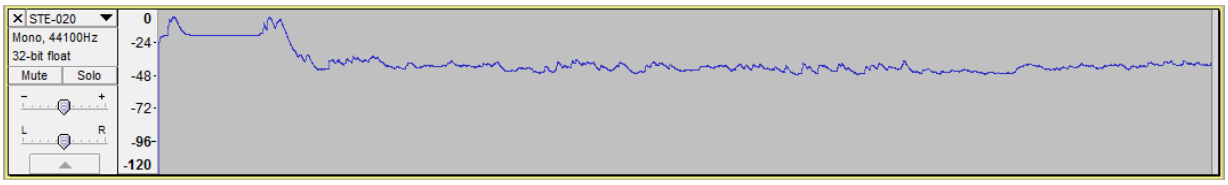


Figura 4.26: Vista del nivel de una señal con ponderación Z y τ = 1 s.

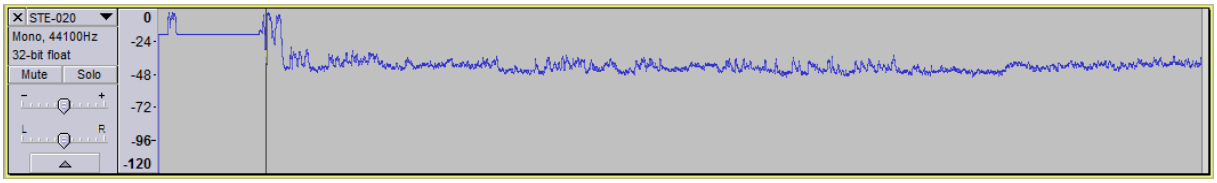


Figura 4.27: Vista del nivel de una señal con ponderación Z y τ = 0,125 s.

En la sección A.2 se observa ver el código que permite calcular el nivel de la señal y en la Figura 4.28 el diagrama de bloques perteneciente al visor de nivel.

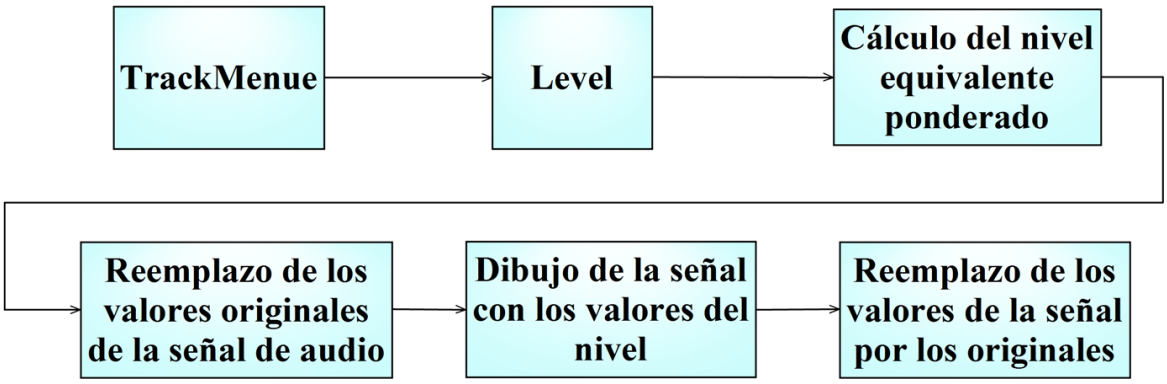


Figura 4.28: Diagrama de bloques del visor de nivel.

4.4 Importador y exportador de metadatos

Esta modificación se encarga de importar o exportar, a partir de un archivo con formato WAVE, las *marcas (cues)*, *etiquetas (labels)* y *anotaciones (notes)* guardadas en el mismo. Las mismas son utilizadas por investigadores, músicos, productores musicales o cualquier persona que lo necesite para señalizar y realizar notas sobre un evento acústico, como por ejemplo identificación de fuentes, datos sobre horario y lugar de medición, tonos de calibración, señales de sincronismo entre varias mediciones, etc. Este tipo información es conocido genéricamente como metadatos (datos que se refieren a otros datos, describiéndolos, clasificándolos o calificándolos).

Los metadatos son agregados de manera interactiva posteriormente a la grabación del archivo original, quedando grabados al final del archivo de audio. Dentro del programa Audacity ésto se puede realizar utilizando pistas especiales denominadas *LabelTracks*.

Audacity permite crear, exportar e importar marcas en formato de texto (.txt) compuestos por tres columnas separadas por tabulaciones que contienen la siguiente información:

Inicio Marca 1	Fin Marca 1	Etiqueta 1
Inicio Marca 2	Fin Marca 2	Etiqueta 2
Inicio Marca 3	Fin Marca 3	Etiqueta 3

Al exportarse e importarse de esta manera no se obtiene el funcionamiento inter-programa deseado, es decir, no se permite que un archivo con marcas creado y grabado en un determinado software se pueda abrir en otro. Por este motivo es que surge la necesidad de agregarle a Audacity la posibilidad de importar y exportar las marcas, etiquetas y notas desde y hacia los archivos.

Luego de analizar y estudiar cómo está compuesto un archivo con formato WAVE (ver Figura 4.29 y el funcionamiento de las LabelTracks (ver Figura 4.30) se llegó a la conclusión de que se debía modificar lo que Audacity define como Label y su composición.

Una Label, según el software, posee tres variables que son el inicio, el final y la etiqueta, se puede ver en la función *Addlabel* perteneciente a las LabelTracks y que para la importación y exportación propia del programa utiliza estos campos.

```
int AddLabel(double t,double t1,const wxString &title=wxT(""));
```

Para lograr el funcionamiento deseado hace falta agregar un campo más a la Label, donde guardar la nota que se puede asociar o no a ella. Por eso se agregó la variable note al objeto Label quedando compuesta entonces por cuatro valores: inicio, fin, etiqueta y nota. De esta manera ahora la función para agregar una etiqueta es la que se ve a continuación.

```
int AddLabel(double t,double t1,const wxString &title=wxT(""),const wxString &note=wxT(""));
```

Una vez logrado esto, y basándose en la composición un archivo WAVE, se desarrollaron los distintos códigos para poder realizar la lectura y escritura de archivos. La primera implementación fue el lector de etiquetas en forma de plug-in dentro del menú de análisis, pero al comentar dentro de la lista de correo electrónico de desarrolladores de Audacity propusieron que no fuera una herramienta externa sino parte del código fuente, por lo cual se realizaron las modificaciones necesarias para añadirlo al mismo.

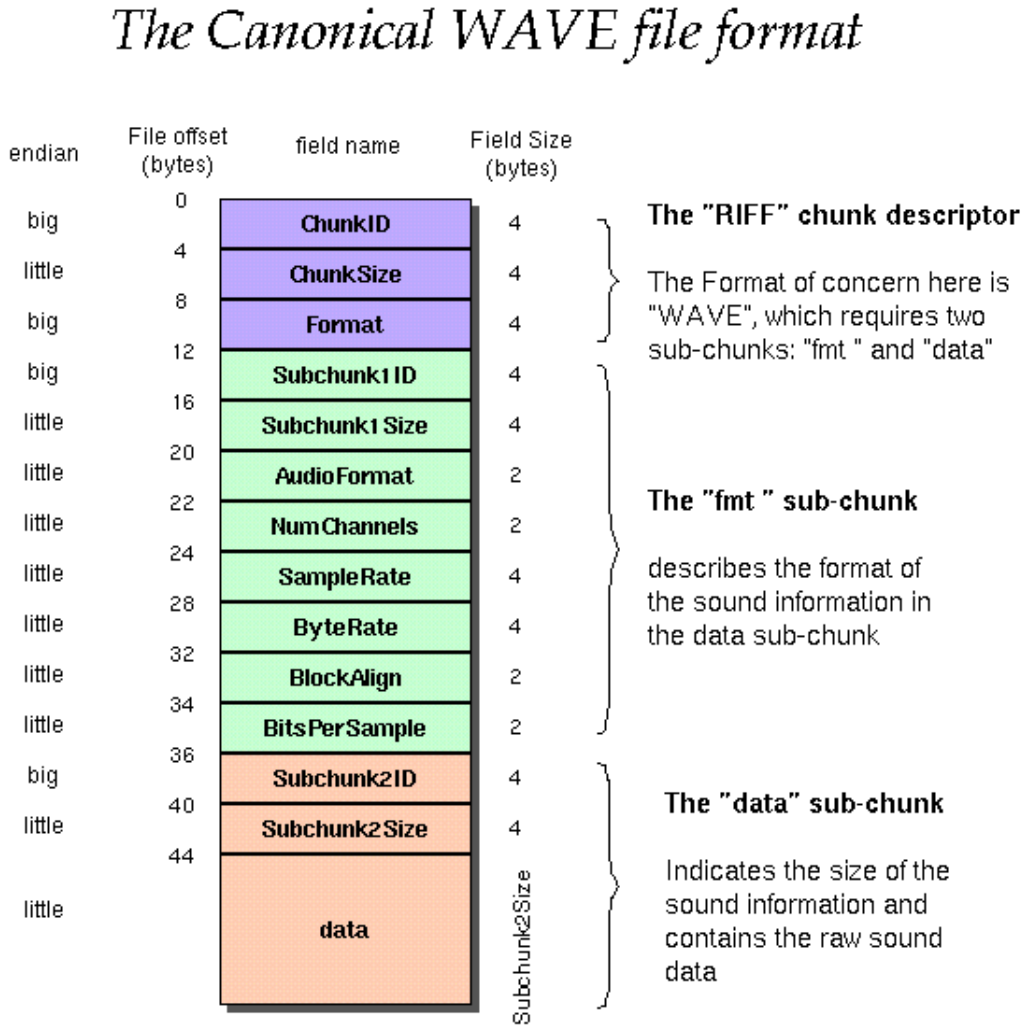


Figura 4.29: Composición de un archivo de formato WAVE [11].

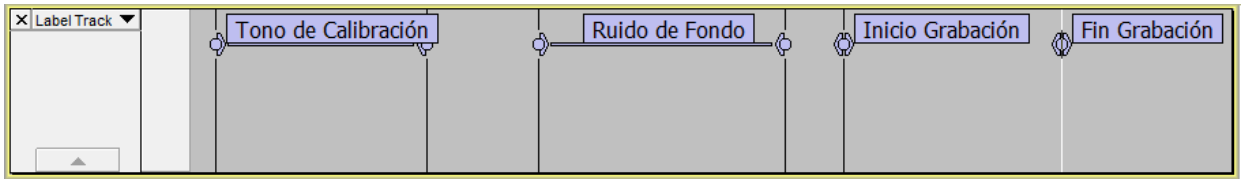


Figura 4.30: LabelTrack con distintos tipos de marcas.

El funcionamiento entonces, pasó a ser el siguiente:

- Si el usuario abre o importa un archivo con extensión .wav (A.3) el programa revisa el encabezado del mismo y determina si posee algún tipo de metadato (A.4).
- En caso de que exista algún metadato, se abre un cuadro de dialogo que pregunta si se desean cargar las etiquetas que el archivo tiene.
- Si el usuario responde de manera afirmativa el programa comienza con la lectura de las etiquetas, marcas y notas, debido al formato se realiza de la siguiente manera:
    1. Lee la muestra de comienzo de cada marca y se obtiene la cantidad de marcas en total que existen.
    2. Guarda la cantidad de muestras de duración y la longitud de la cadena de caracteres de cada marca.
    3. Crea un espacio en memoria del tamaño de la suma de los caracteres de todas las marcas juntas.
    4. Copia en memoria cada etiqueta una seguida de la otra.
    5. Repite el proceso de lectura de etiquetas para obtener las notas y a su correspondiente posición (ya que no todas las etiquetas tiene una nota asociada).
- Crea una LabelTrack y carga las etiquetas asignándole inicio, fin y nota (A.5).

A causa del agregado del campo note se debió modificar también el editor de etiquetas que posee el software, que muestra en una tabla el nombre de la pista, la etiqueta, el inicio y el final de esta última. La modificación consistió en agregar una columna denominada Note y cambiar el código fuente para que cargue y muestre la misma en ella. A continuación, en las Figuras 4.31 y 4.32, se muestra el editor original y el modificado respectivamente.

Por su parte el exportador de etiquetas funciona de manera inversa cuando se desea guardar un archivo con formato .wav o exportar etiquetas y existen *LabelTracks*. A diferencia del importador se necesita no sólo modificar la parte final del archivo para agregar la información sino además cambiar el encabezado para informar que existen otros datos además de los de la señal de audio.

En las Figuras 4.33, 4.34, 4.35 y 4.36 se puede ver el funcionamiento por pasos de cómo importar los metadatos.

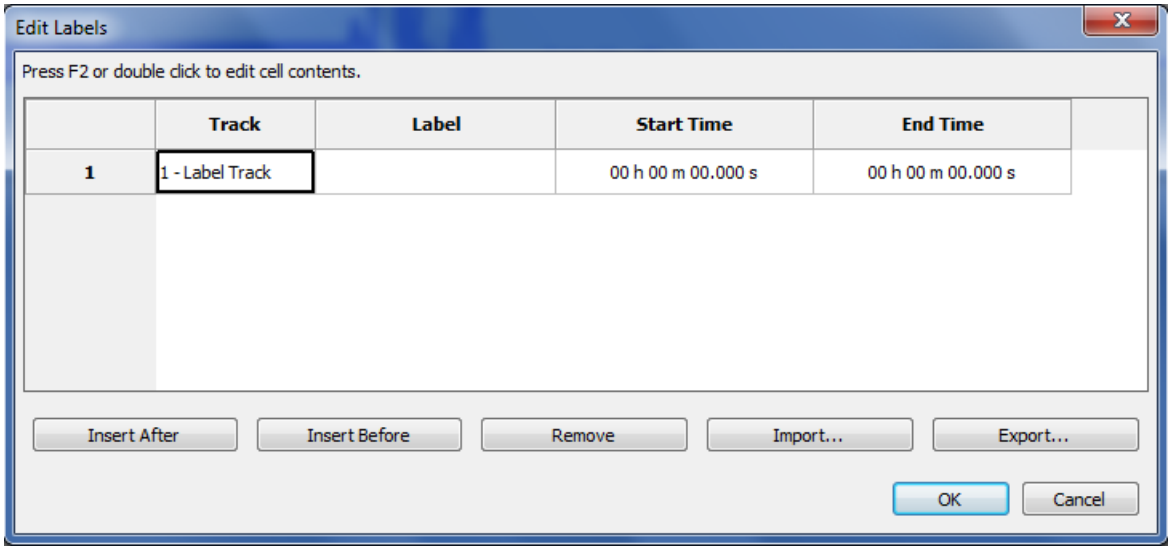


Figura 4.31: Menú de edición de labels original.

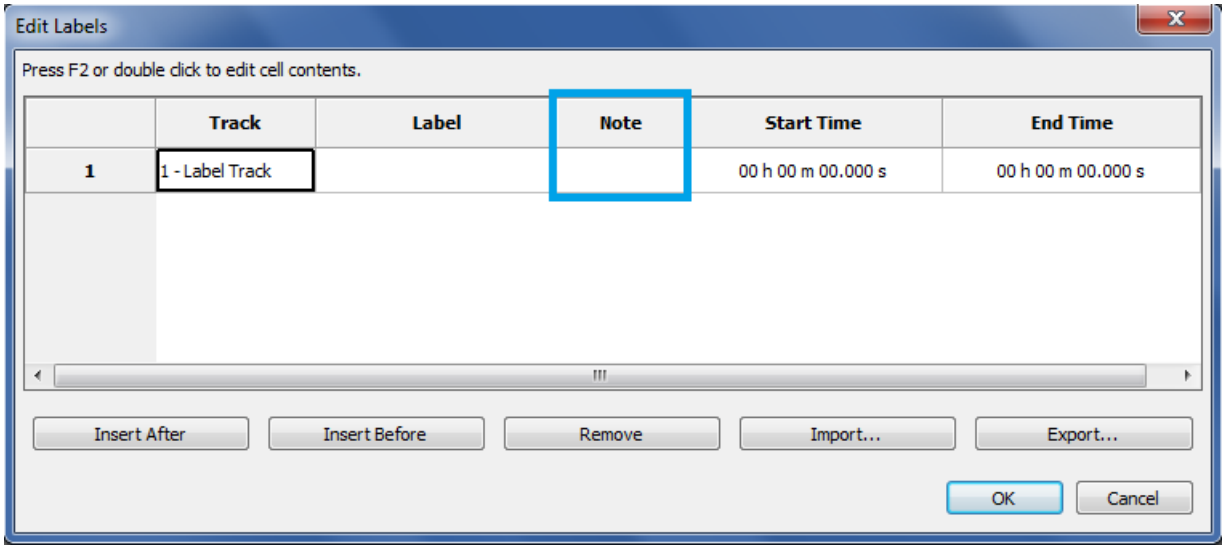


Figura 4.32: Menú de edición de labels modificado.

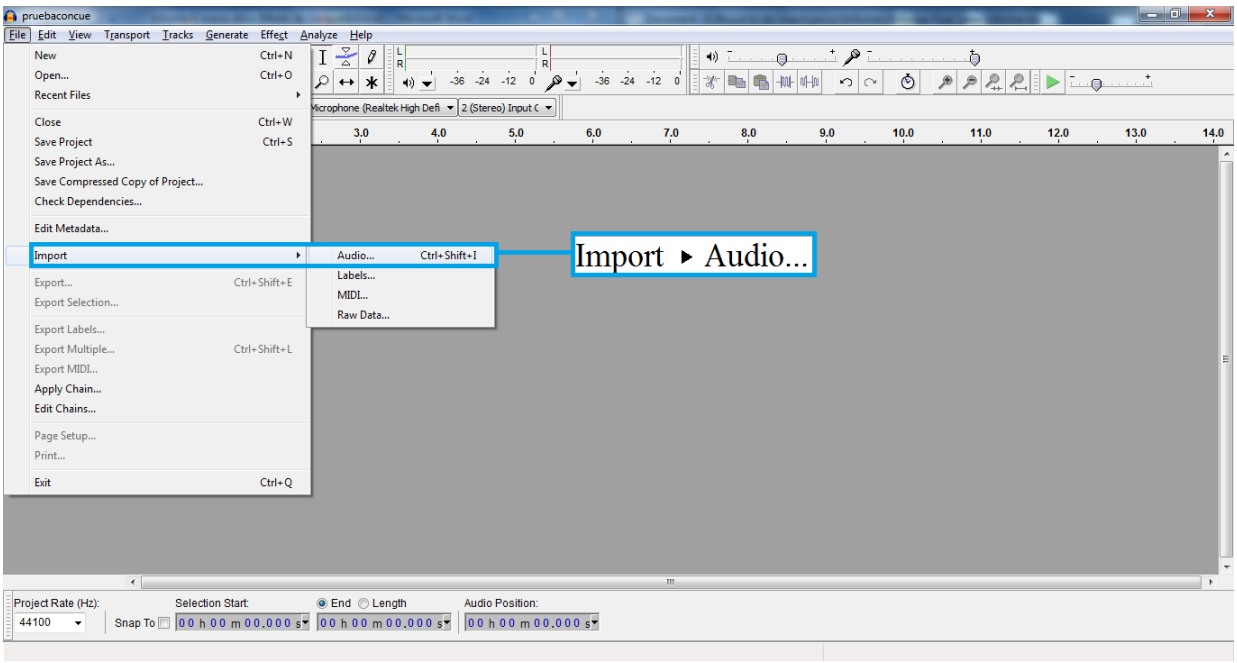


Figura 4.33: Se selecciona Importar->Audio.

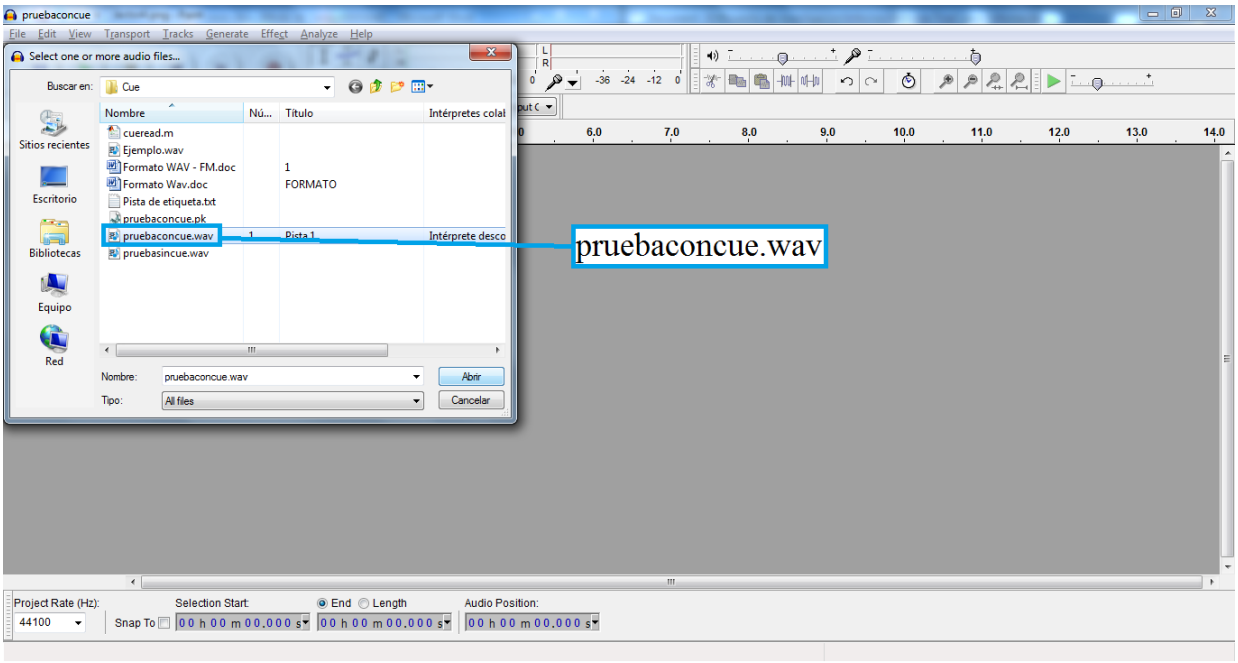


Figura 4.34: Se elige el o los archivos que se desean importar.

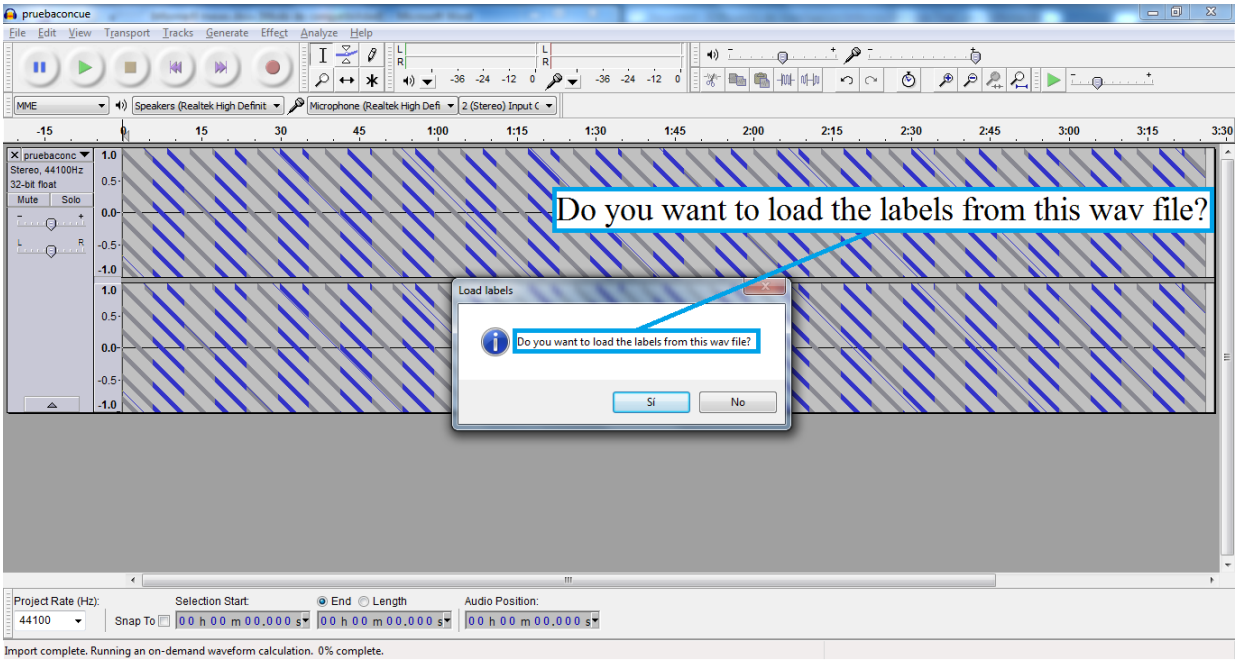


Figura 4.35: Si el archivo tiene marcas, etiquetas o notas aparece un mensaje que nos pregunta si deseamos importarlas.

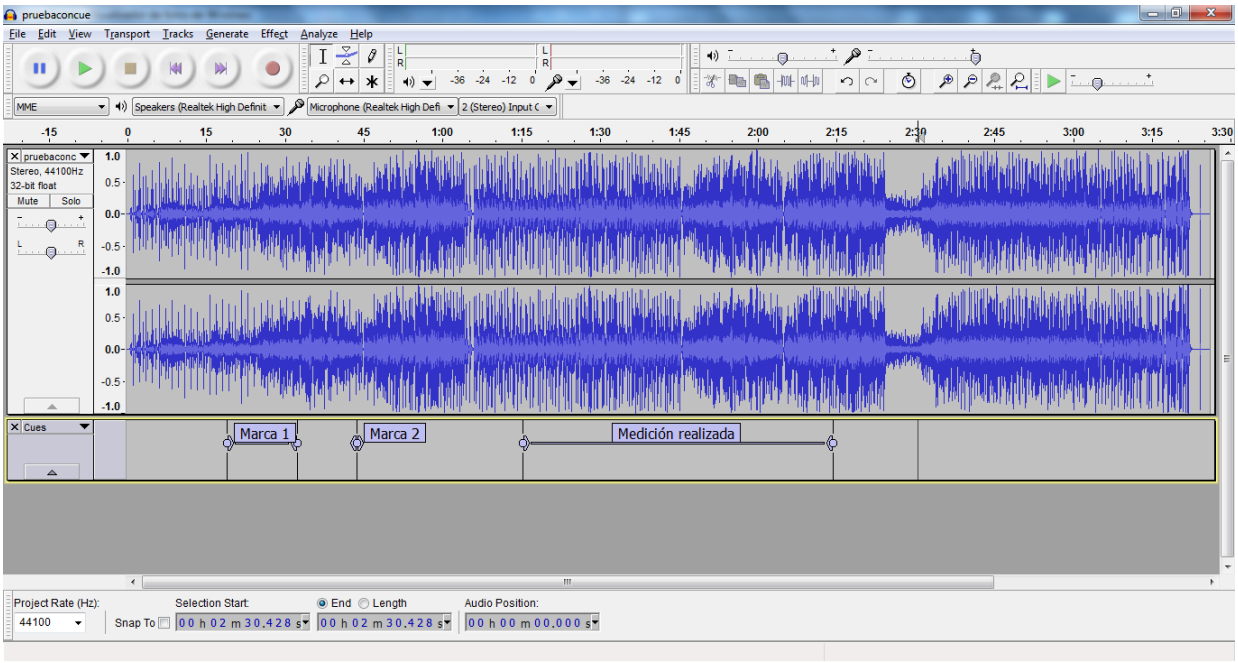


Figura 4.36: Si seleccionamos la opción “Si”se crea la LabelTrack con las etiquetas.

A la hora de exportar las marcas el programa revisa si el formato seleccionado es “.wav”, en ese caso y si existe LabelTracks, aparece el mensaje que se ve en la Figura 4.37

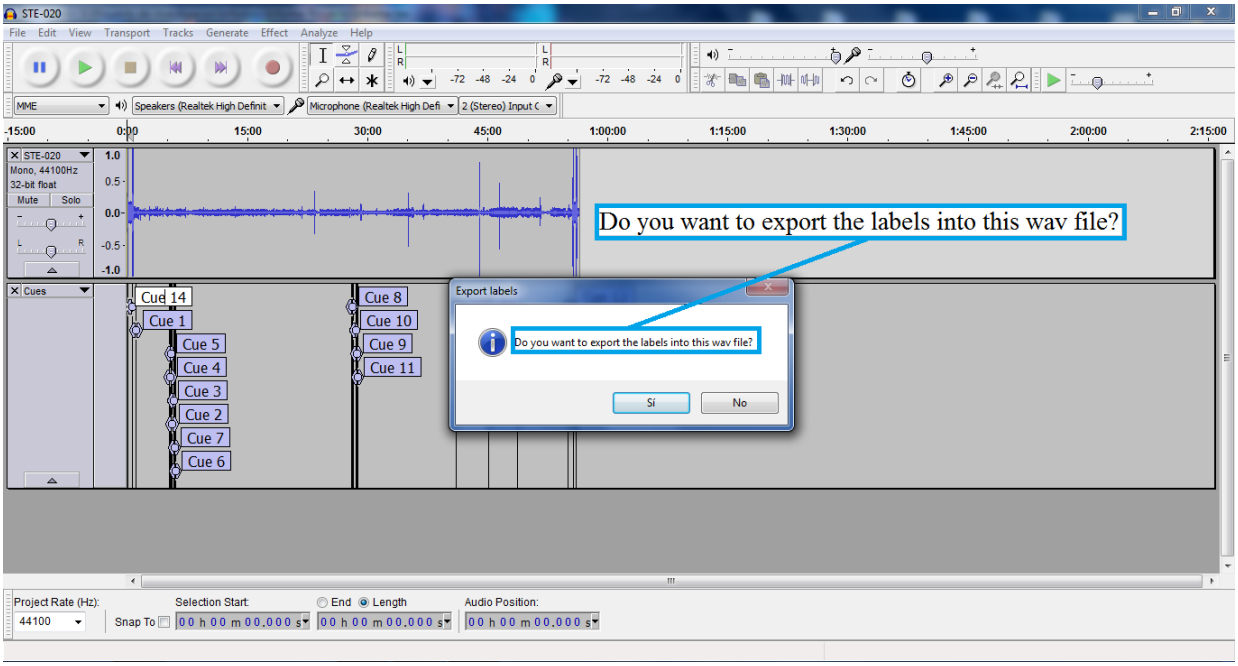


Figura 4.37: Mensaje para decidir si exportar o no las marcas.

Además se implementó, como se ve en la Figura 4.38, la posibilidad de ver si una etiqueta tiene asignada una nota al hacerle click sobre ella en la LabelTrack.

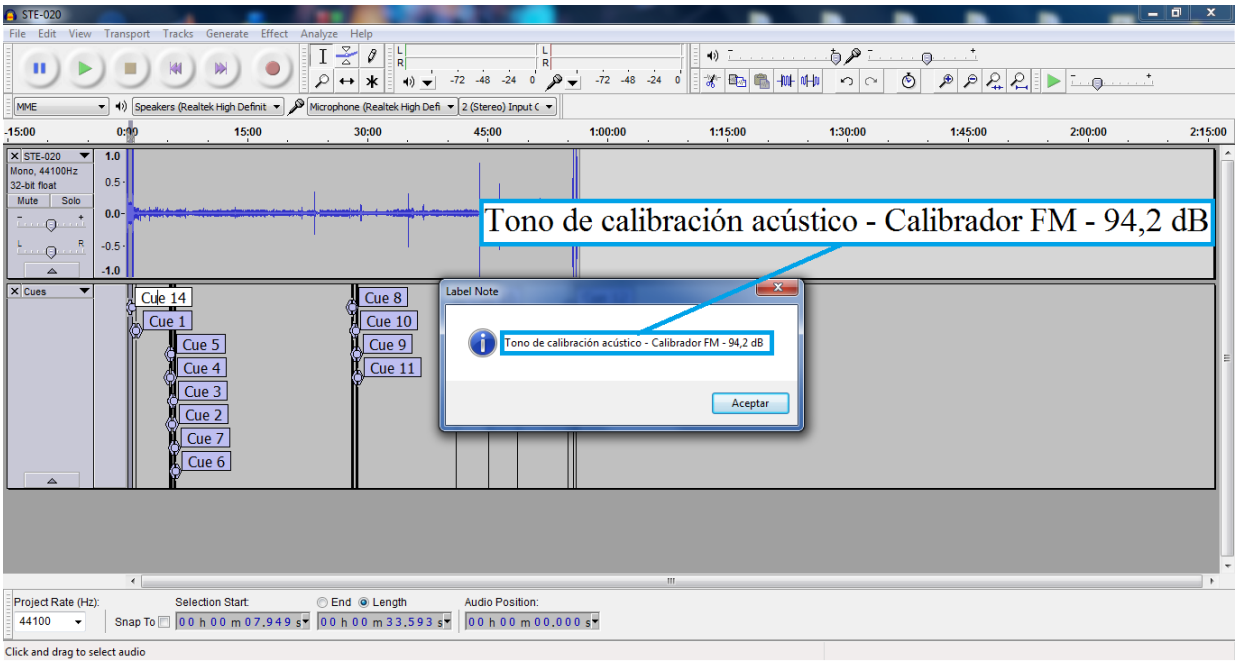


Figura 4.38: Visor de nota al hacer click sobre una marca.

El código de la sección A.6 permite exportar las marcas, etiquetas y notas al archivo .wav

seleccionado.

En las Figuras 4.39 y 4.40 se muestran los diagramas de bloque que describen el funcionamiento del importador y del exportador de marcas, etiquetas y notas.

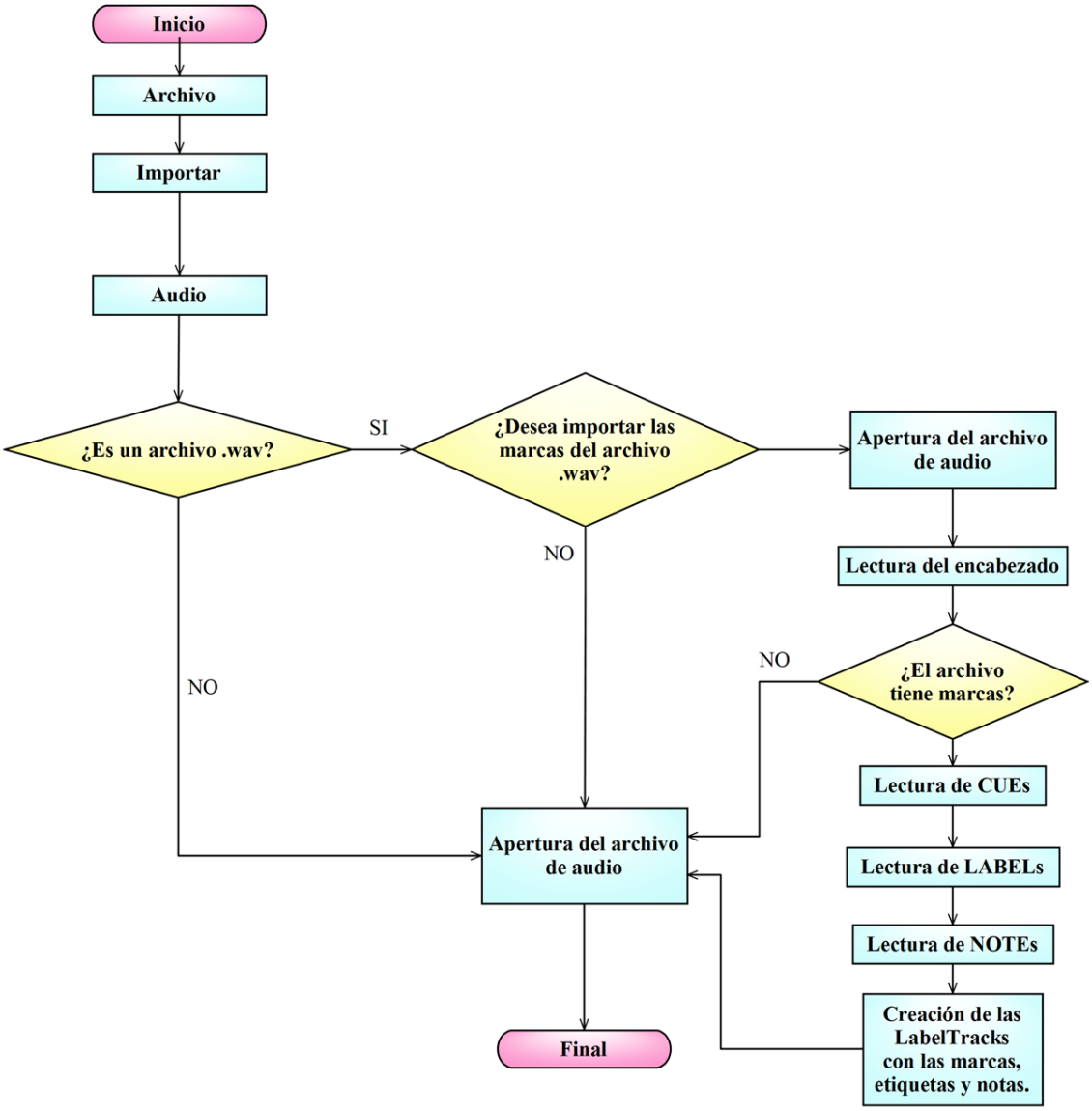


Figura 4.39: Diagrama en bloques del importador de CUEs.

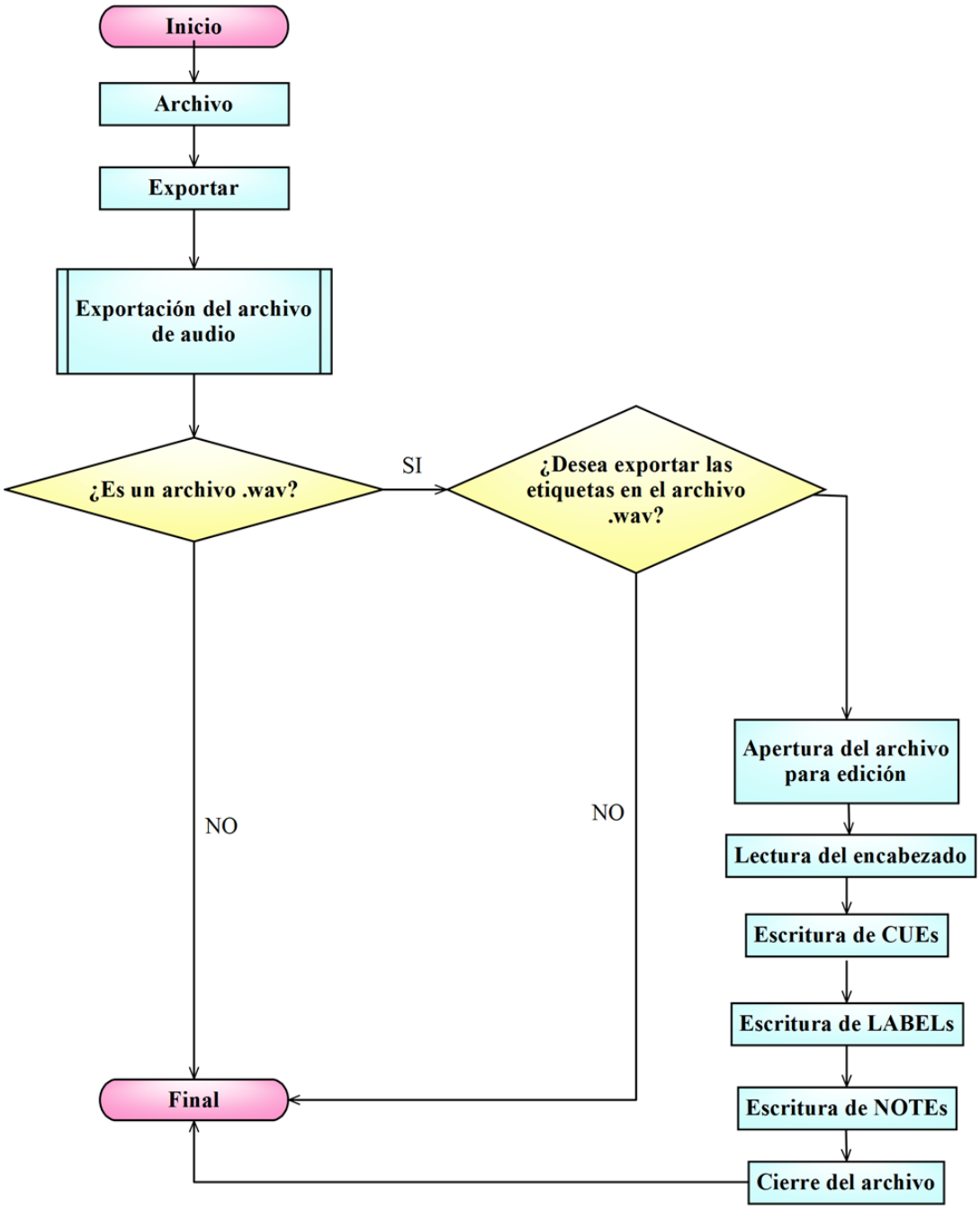


Figura 4.40: Diagrama en bloques del exportador de CUEs.

Capítulo 5
Desarrollo de Plug-ins

5.1 Creación de archivos base

Para llevar a cabo este apartado me puse en contacto con programadores de otras partes del mundo, los cuales ya habían desarrollado herramientas para Audacity anteriormente. A partir de la información recopilada y algunos archivos que intercambié con ellos, creé una guía de cómo obtener e insertar un *Plug-in* de Audacity, partiendo desde cero (ver la sección B.2).

Los archivos fundamentales para el desarrollo de herramientas nuevas son **module.h** y **module.cpp**, creados por el Ingeniero Italiano Simone Campanini de la Università degli Studi di Parma - Facoltà di Ingegneria. Ambos archivos poseen sólo algunas líneas de código capaces de integrar a Audacity el Plug-in desarrollado, luego es trabajo del programador el modificarlos y darle la funcionalidad deseada.

5.2 Spectral Edit Filter

Este Plug-in es una herramienta que permite al usuario dibujar sobre el *espectrograma* de la señal un filtro que varíe a lo largo del tiempo. Posee diversas utilidades entre ellas:

1. Provee una manera muy intuitiva e interactiva de filtrar en el dominio espectral.
2. Aisla o elimina eventos caracterizados por algún rasgo espectral característico.
3. Posibilita determinar la audibilidad o no de determinada porción del espectro.
4. Es muy útil para composición de música electroacústica.
5. Depura de manera interactiva el espectro, útil para la eliminación de ruidos no estacionarios y que por lo tanto no se prestan al uso de la herramienta Noise Removal.
6. Útil como herramienta educativa, ya que permite un rápido entrenamiento y autoentrenamiento en la detección de patrones espectrales característicos (ejemplo, pájaros, perros, voces humanas, vehículos, etc.).

La interfaz gráfica es una ventana semitransparente que se posiciona sobre el espectrograma abierto y permite definir a través de la selección entre distintos tipos de figuras (cuadriláteros, polígonos y curvas arbitratias cerradas) el filtro deseado. Como se explicó anteriormente, debido a que las interfaces gráficas se encuentran escritas bajo la librería wxWidgets se lo utilizó para el desarrollo de ésta GUI (Graphical User Interface o Interfaz Gráfica de Usuario).

El desarrollo se realizó en varias etapas:

1. Diseño de la ventana semitransparente.

2. Desarrollo de la posibilidad de dibujar sobre la ventana.
3. Búsqueda de la forma más conveniente de mostrar el menú e identificación de funciones a mostrarse.
4. Integración de la ventana como Plug-In a Audacity.
5. Adquisición, edición y muestra de datos en pantalla.

El funcionamiento interno, una vez diseñado el filtro, consiste en:

1. Transformar mediante un cálculo las abscisas y ordenadas de los puntos dibujados en valores de tiempo y frecuencia respectivamente.
2. Calcular el comienzo y el final del filtro y reemplazar los valores de la señal anteriores y posteriores a estos valores por ceros (equivalente a aplicar un filtro de ganancia nula constante)
3. Crear una matriz donde cada columna contenga los valores correspondientes a la respuesta del filtro a aplicar en cada ventana. Este filtro está compuesto por unos y ceros que se asignan siguiendo un conjunto de reglas:
   - Ganancia nula hasta encontrar el primer valor en orden creciente de frecuencia correspondiente a ese tiempo (discretizado cada medio cuadro).
   - Ganancia uno hasta encontrar el segundo valor de frecuencia correspondiente a ese tiempo, siempre y cuando exista una diferencia de al menos un $\delta F = F_s/W_s$ con el valor anterior (ver Figura 5.1). Esto evita que una línea vertical se transforme en una tira de ceros y unos alternados.

Figura 5.1: Ejemplo de obtención de filtro a partir del dibujo (última columa vs filtro).

- El filtro generado por múltiples dibujos es la unión menos la intersección de ellos.

La matriz se utiliza para multiplicar el espectro de cada ventana de audio y así obtener la señal filtrada. Cada cuadro de audio tiene un ancho de 4096 muestras (lo que resulta en una resolución frecuencial de 10,72 Hz, próxima a la resolución del oído humano común) y es multiplicado por una ventana de Hanning[1] antes de la edición, luego se obtiene el espectro del mismo y se lo multiplica por el filtro guardado en la matriz. Para evitar añadir ruido al reconstruir la señal luego de ser modificada se toman cuadros con un solapamiento del 50 %.

En resumen el funcionamiento paso a paso es el siguiente:

1. Se genera, a partir del dibujo, la matriz que contiene el filtro para cada cuadro.
2. Se reemplazan las muestras fuera del filtro por ceros.
3. Se separa la señal en cuadros de 4096 muestras solapados al 50 %.
4. Se le calcula el espectro utilizando el método FFT.
5. Se multiplica la FFT por el vector de la matriz de filtro correspondiente.
6. Se vuelve el cuadro al dominio temporal mediante la IFFT.
7. Se multiplica el cuadro por la ventana de hanning.
8. Se repite desde el paso 4 hasta el 7 hasta recorrer todos los cuadros.
9. Se reconstruye la señal sumando los cuadros, nuevamente con el solapamiento del 50 %.

Los puntos 3 a 7 los realiza la función *fftfilter*, desarrollada por el Ing. Gonzalo Sad (basada en un script de MATLAB creado por Federico Miyara originado en una idea de David Giardini), modificada por mí para darle la funcionalidad deseada y cuyo código se muestra en la sección A.4.

A continuación se muestra cómo se utiliza la herramienta, lo primero es seleccionarla (Figura 5.2, luego se aparecen dos cuadros de dialogo, como se ven en las Figuras 5.3 y 5.4 que permiten decidir si se desea por un lado extraer o substraer parte del audio y por otro si se quiere hacerlo en ambos canales por igual, de distinta manera o sólo hacerlo en uno de ellos. Una vez elegida las opciones sólo resta dibujar el filtro sobre la señal, elegir la opción *"Aplicar Filtro"* (se muestra en la Figura 5.5 el menú contextual que aparece al hacer click derecho sobre la ventana) e insertar la frecuencia máxima que aparece en el espectrograma (valor necesario para poder realizar la transformación abscisas/ordenadas en tiempo/frecuencia).

[1] Permite reducir el ruido de alta frecuencia que aparece por saltos bruscos entre cuadro y cuadro

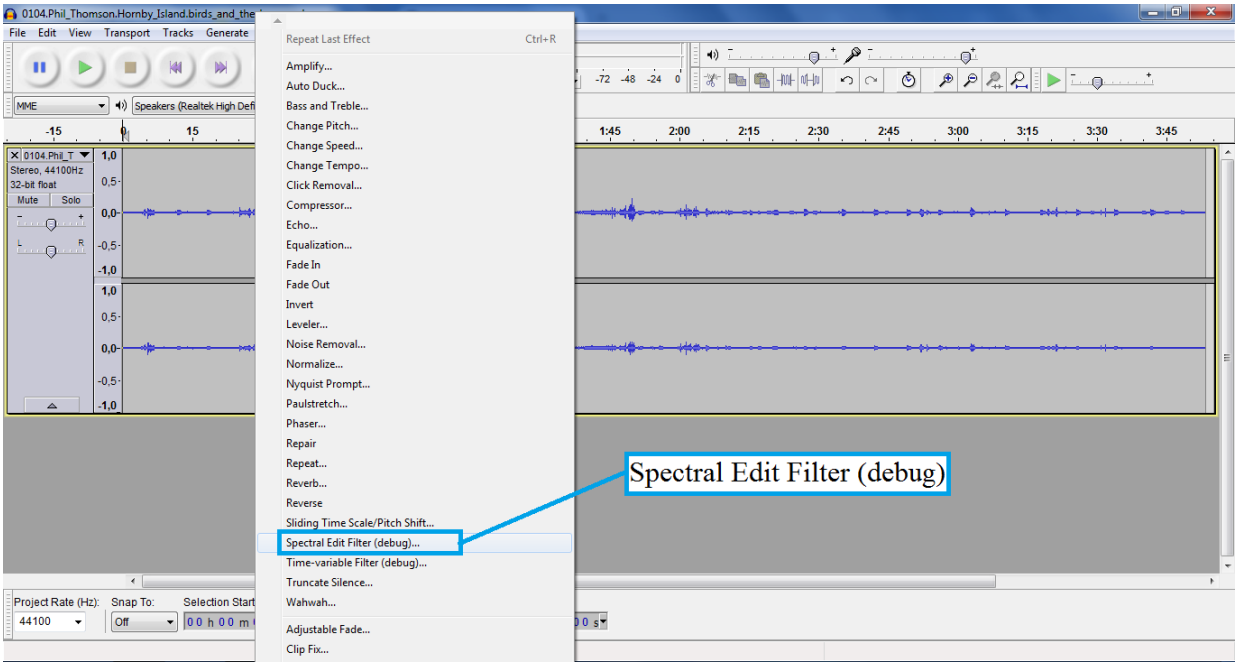


Figura 5.2: Selección del Plug-in.

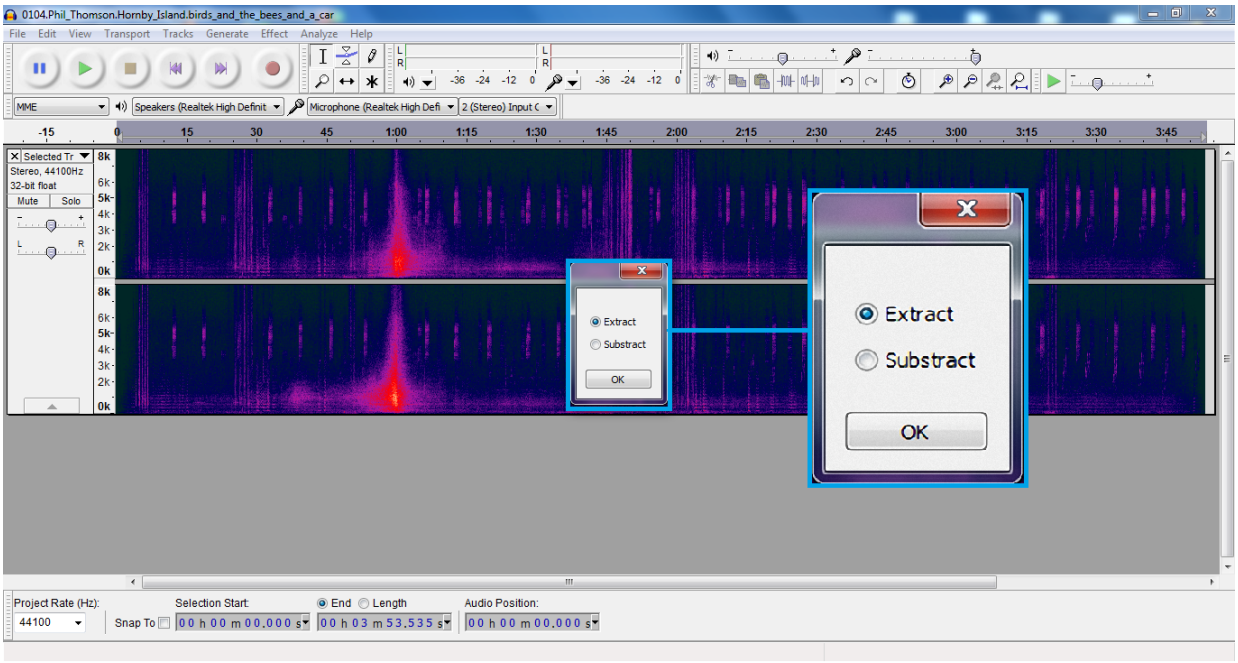


Figura 5.3: Ventana de selección de extracción o substracción.

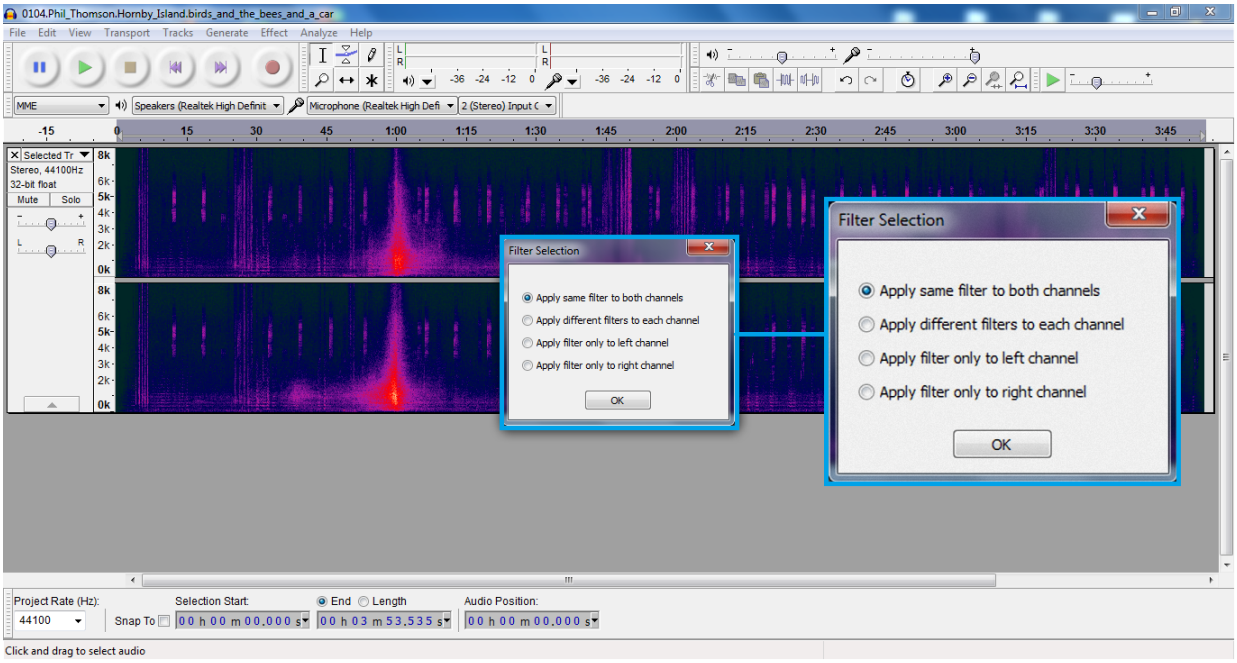


Figura 5.4: Ventana de selección de tipo de filtrado.

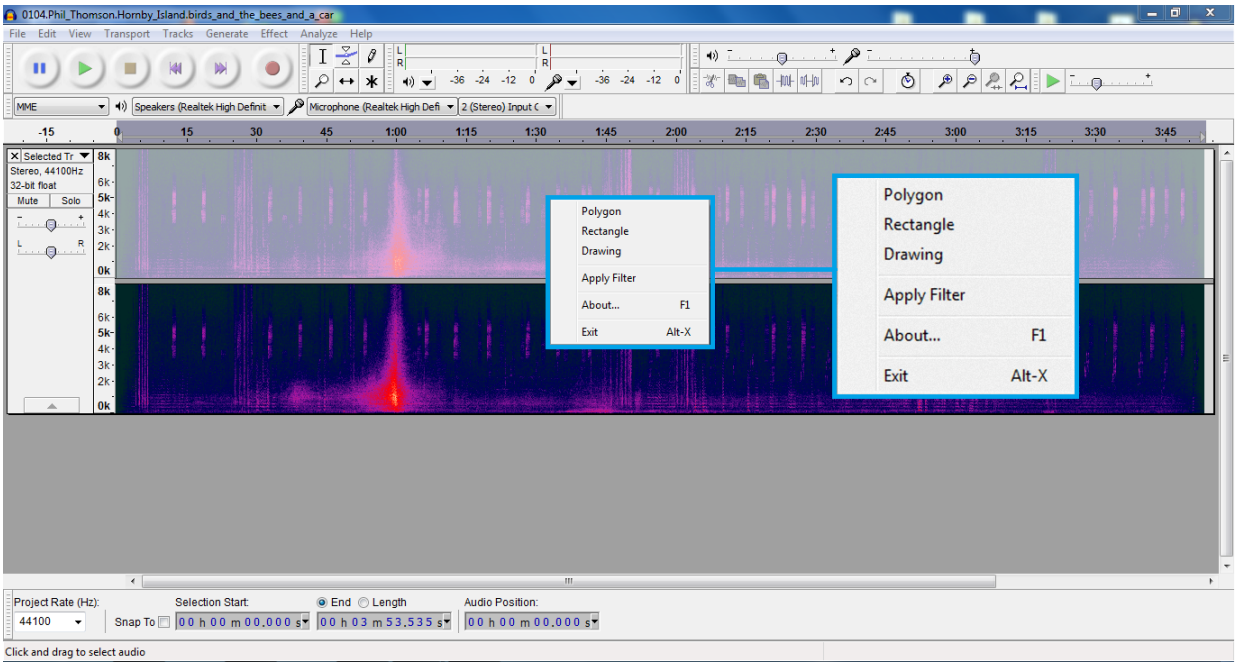


Figura 5.5: Menú contextual que permite seleccionar las distintas herramientas de dibujo.

En las Figuras 5.6, 5.7, 5.8 y 5.9 se muestran los casos de extracción y de substracción.

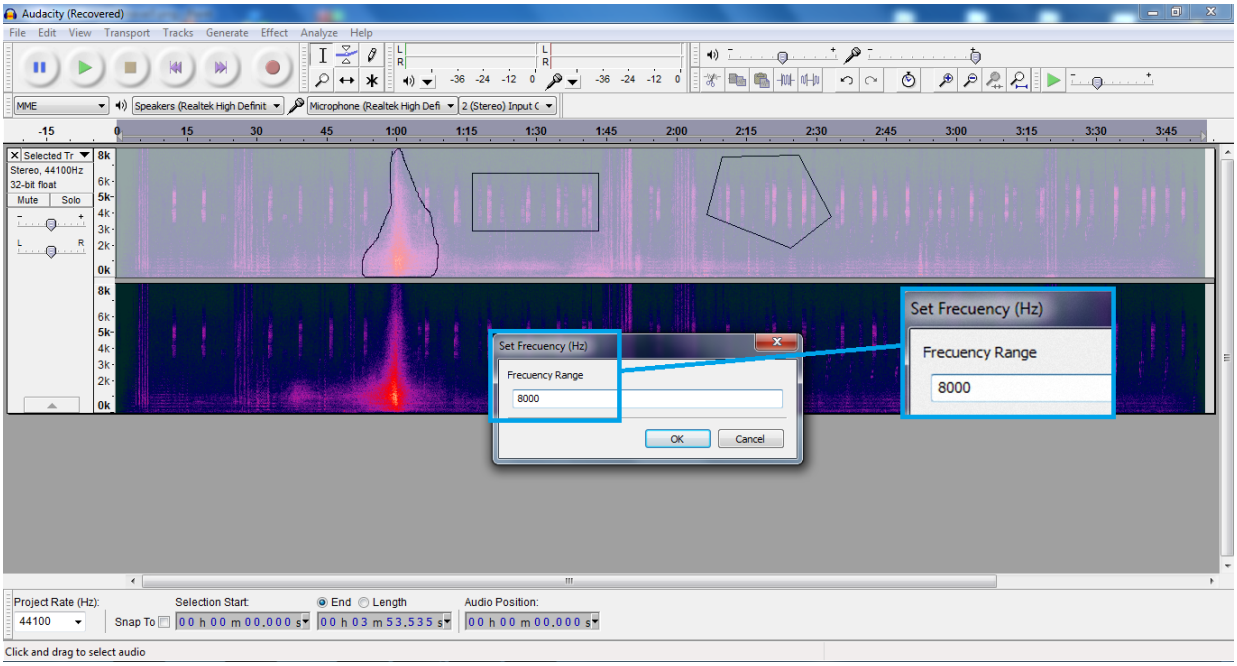


Figura 5.6: Filtro dibujado sobre el espectrograma original, se puede ver el menú de ingreso de la máxima frecuencia visible en el espectrograma.

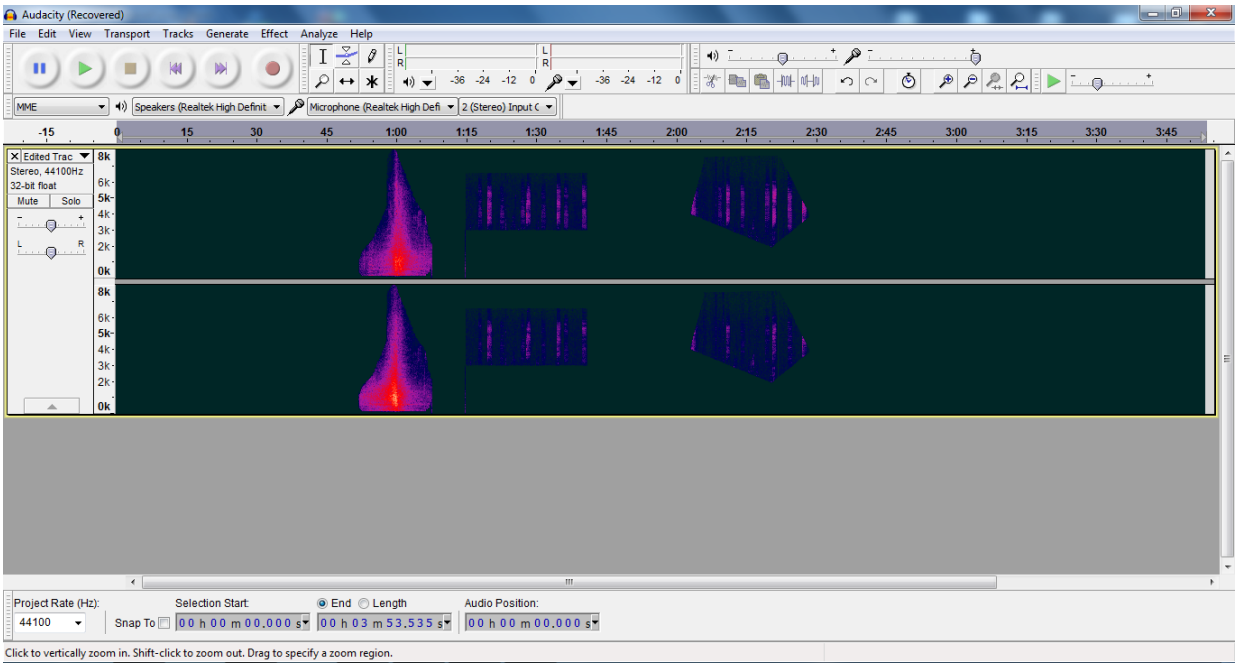


Figura 5.7: Espectrograma luego de extraer de la señal orignal las frecuencias deseadas.

Vemos que, comparando la señal original y la filtrada, sólo se ven en la pista inferior las porciones de espectrograma seleccionados en la Figura 5.6.

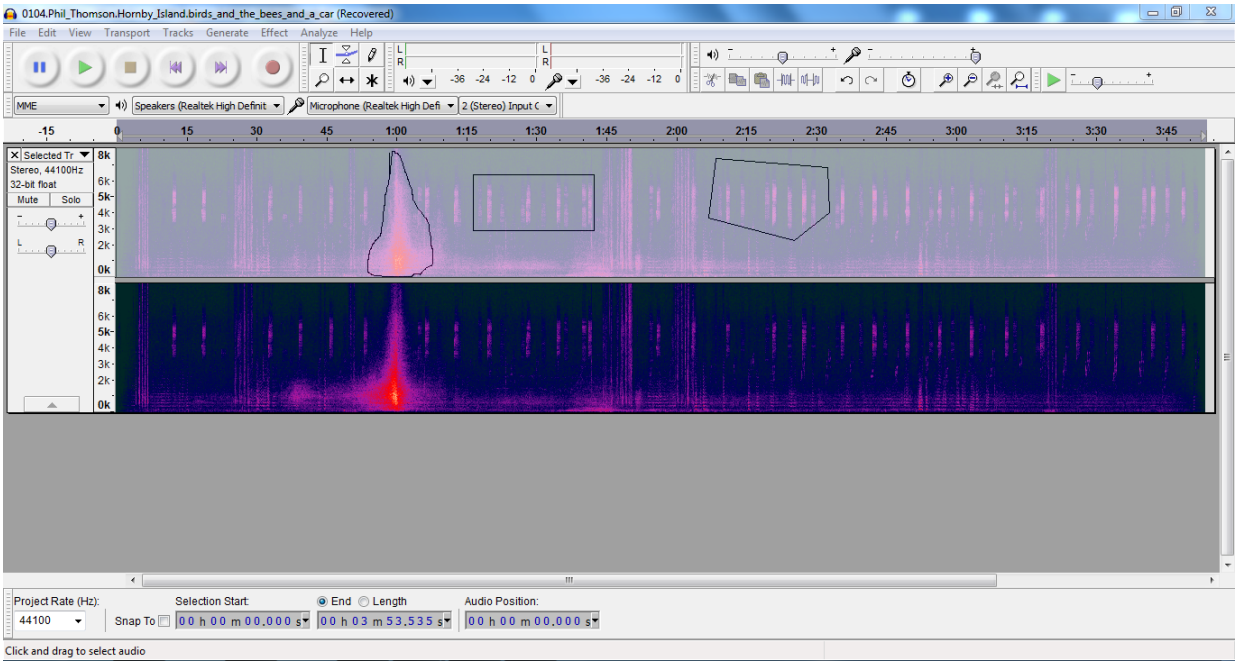


Figura 5.8: Dibujo sobre el espectrograma de las frecuencias a substraer.

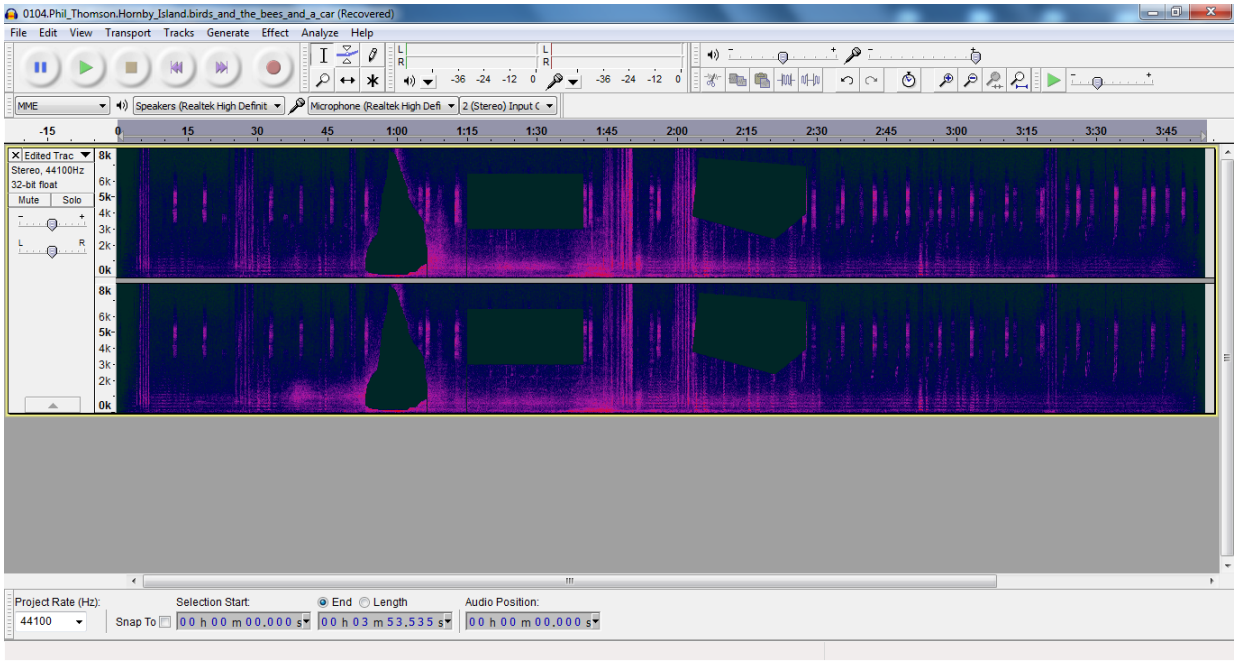


Figura 5.9: Espectrograma luego de aplicar el filtro dibujado.

En este caso, a diferencia del primero, se observó que se obtuvo el espectrograma de la señal original menos la selección que se ve en la Figura 5.8.

En el siguiente diagrama de bloques se muestra el funcionamiento del Plug-in

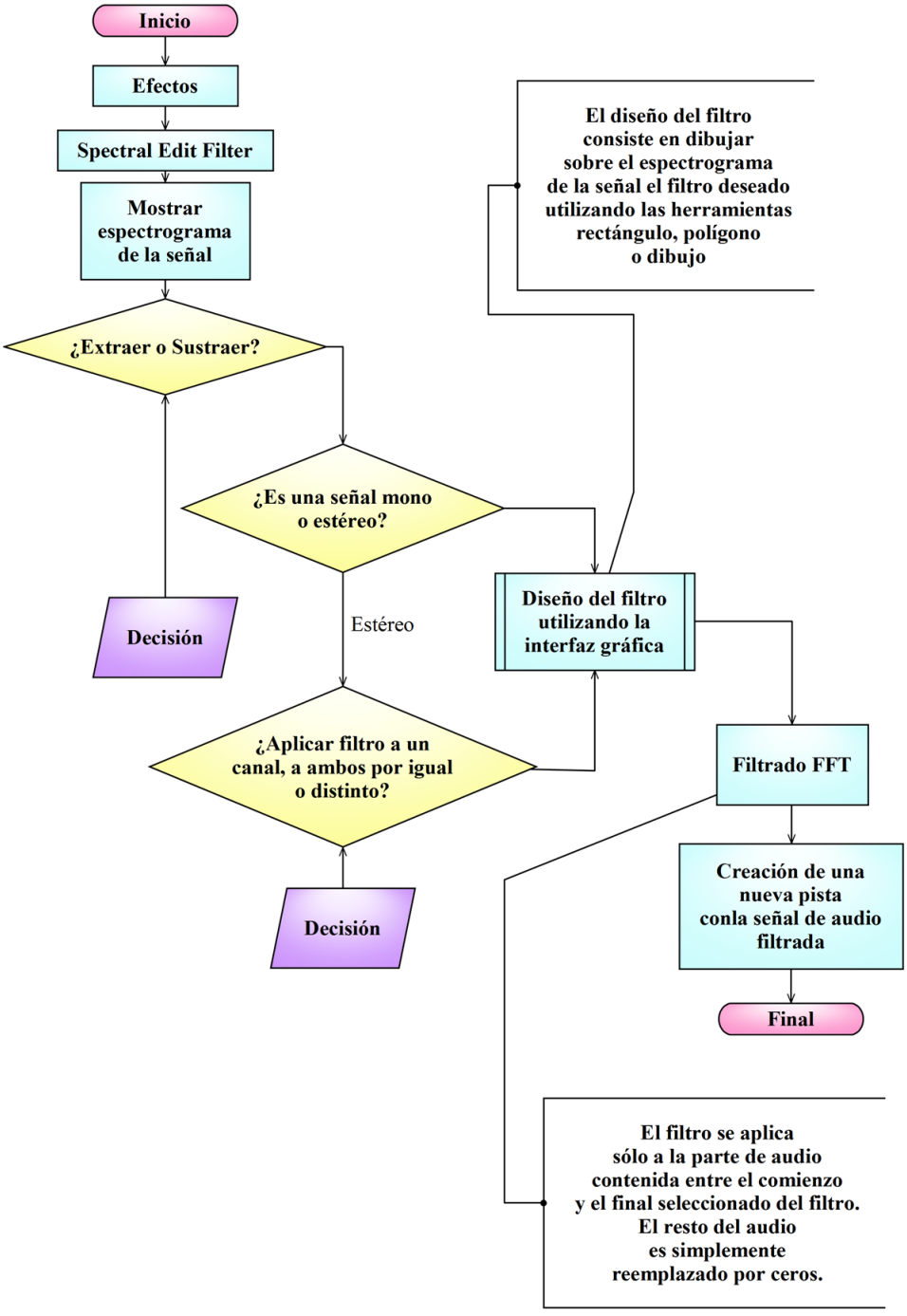


Figura 5.10: Diagrama en bloques del funcionamiento del filtro variable en el tiempo.

5.3 Time Variable Filter

Esta herramienta le permite al usuario definir un filtro que varía en tiempo, frecuencia y amplitud, dando así un mayor control que el Spectral Edit Filter a cambio de ser un poco más complicado de operar. Unas de las principales aplicaciones de este plug-in son los experimentos psicoacústicos, al poder aplicar sucesivos filtros a una misma señal, por ejemplo un ruido blanco, de una sola vez.

A diferencia del Plug-in anterior, que utilizaba un filtro pasa banda (Figura 5.11(a)), el usuario puede definir distintos puntos de paso para el filtro y diseñar el que mejor se adapte al uso que desea darle (Figura 5.11(b)).

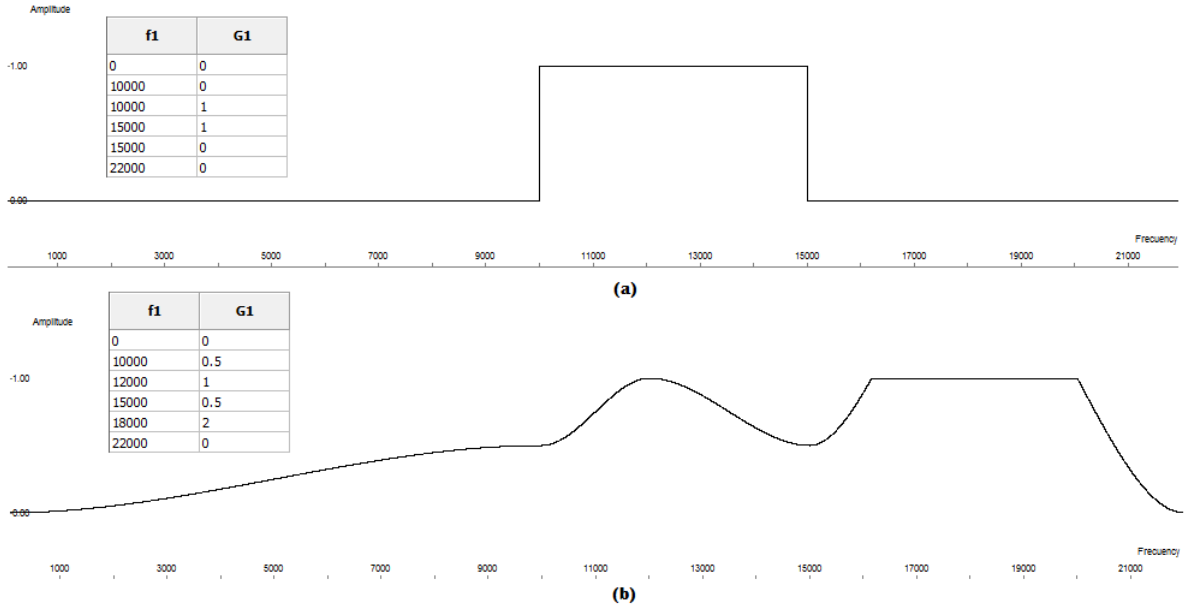


Figura 5.11: Comparación de los tipos de filtros y la tabla que los genera (a) Filtro pasabanda (b) Filtro creado al definir puntos de paso.

El funcionamiento interno del plug-in es similar al Spectral Edit Filter, excepto la forma en la que se genera la matriz filtro. Primero el usuario define puntos de paso del filtro y el tiempo en que desea que se aplique. A partir de los puntos de paso se interpola, con una curva Hermite Spline[2], el filtro buscado y se lo asigna a la columna de la matriz correspondiente. Luego, mediante una interpolación lineal entre sucesivas curvas definidas, se genera el resto de la matriz. Esto implica que se deben definir al menos dos curvas y que a diferencia del Spectral Edit Filter la ganancia de los filtros puede tomar valores entre 0 y 1.

La interfaz visual, como se ve en la Figura 5.12, está compuesta por dos tablas, un conjunto de botones y una gráfica. Las tablas le permiten al usuario definir el tiempo en que comienza cada filtro y los puntos de paso de la curva. Los botones permiten agregar o remover filtros y puntos de paso, graficar cualquiera de los filtros definidos (”Plot Filter”), seleccionar el tipo de interpolación

[2]Curva diferenciable definida en porciones mediante polinomios.

deseado, y aplicar el filtro a la señal. Por último, la gráfica permite al usuario observar si el filtro diseñado es lo que desea o no.

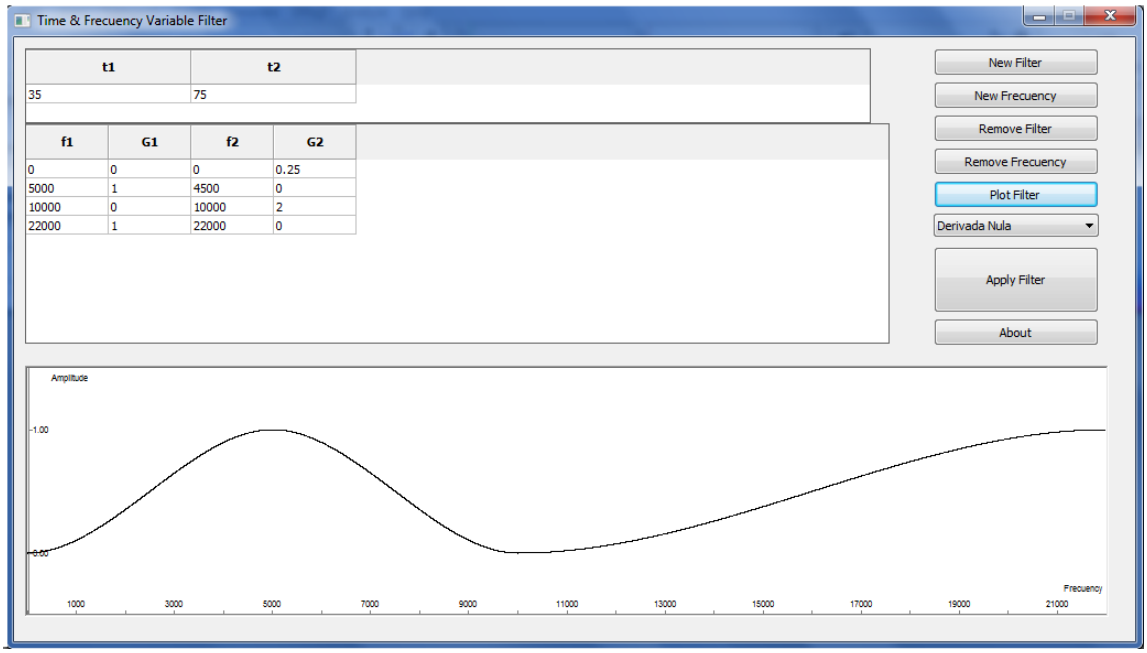


Figura 5.12: Interfaz gráfica.

La gráfica fue diseñada con una librería desarrollada en **wxWidgets** llamada **wxMathPlot** que está compuesta por dos archivos, a saber, mathplot.h y mathplot.cpp, que se pueden descargar desde la página oficial de la librería [10] y posteriormente se deben agregar a cada proyecto que las necesite. Una vez que el usuario define los puntos de paso y selecciona el botón *"Plot Filter"*, se realiza una interpolación del tipo *Spline Cúbica de Hermite* definida según la siguiente fórmula:

$$p(x) = (2t^3-3t^2+1)p_k+(t^3-2t^2+t)(x_{k+1}-x_k)m_k+(-2t^3+3t^2)p_{k+1}+(t^3-t^2)(x_{k+1}-x_k)m_{k+1} \tag{5.1}$$

donde p_k, x_k y m_k definen ganancia, frecuencia y derivada del filtro en un punto *k* y t se define de la siguiente forma

$$t = \frac{(x - x_k)}{(x_{k+q} - x_k)} \tag{5.2}$$

Luego de realizar distintas pruebas se decidió darle al usuario la posibilidad de elegir entre diferentes formas de cálculo de derivadas

- Derivadas nulas (disminuye el cálculo matemático y evita sobrepicos indeseados alrededor de los puntos definidos).
- Cálculo automático del valor de la derivada (a partir de los valores del punto de paso anterior y del posterior).

En la Figura 5.13 se puede observar una comparación entre los distintos criterios de cálculo de la primer derivada.

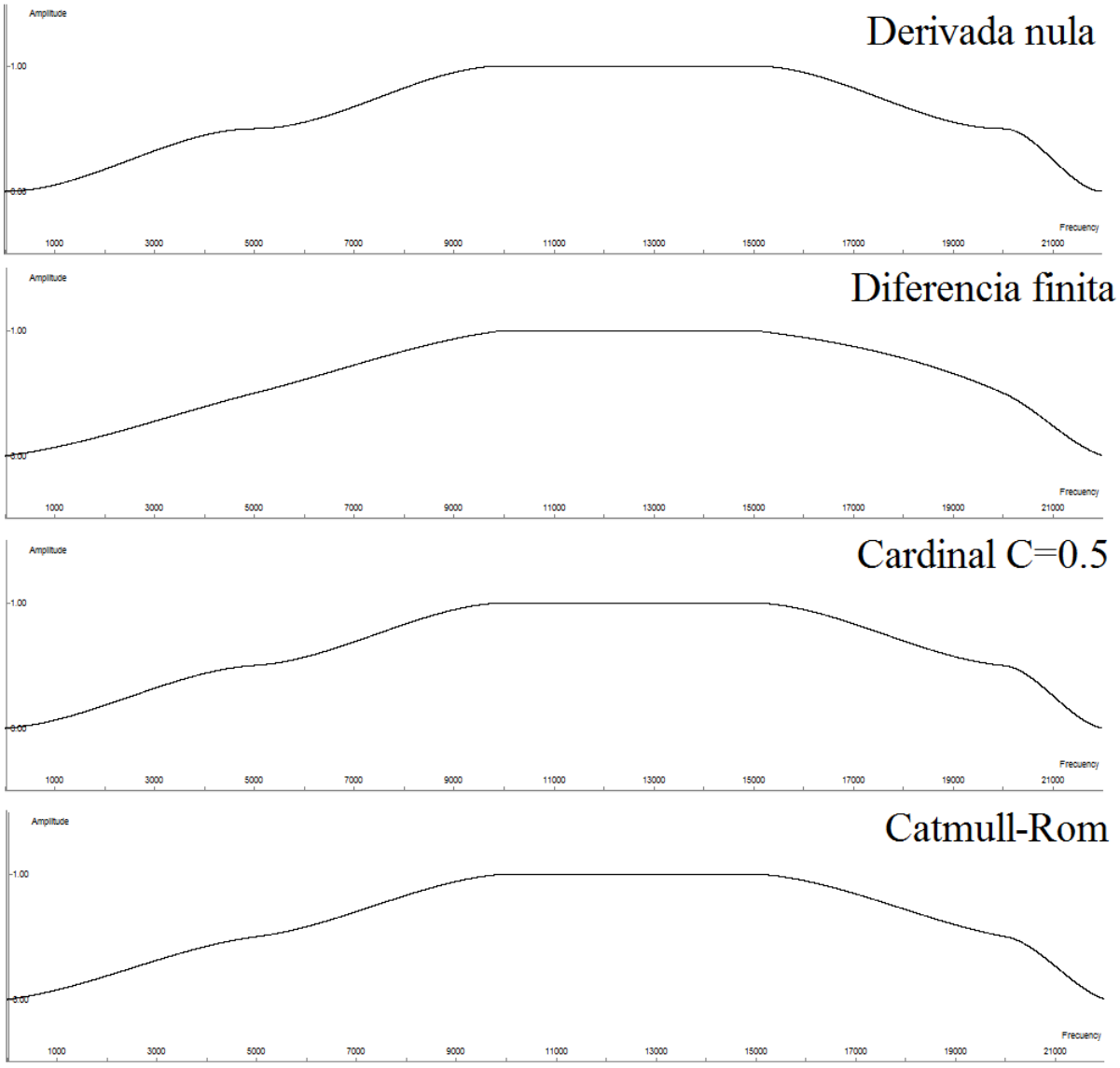


Figura 5.13: Comparación de los criterios de cálculo de derivadas.

A continuación en las Figuras 5.14 a 5.17 vemos el paso a paso como funciona el plug-in aplicándolo a ruido blanco y definiendo 4 filtros y en la sección A.5 se encuentra la porción de código que permite obtener, a partir de los puntos de paso la curva del filtro utilizando la interpolación Hermite Spline.

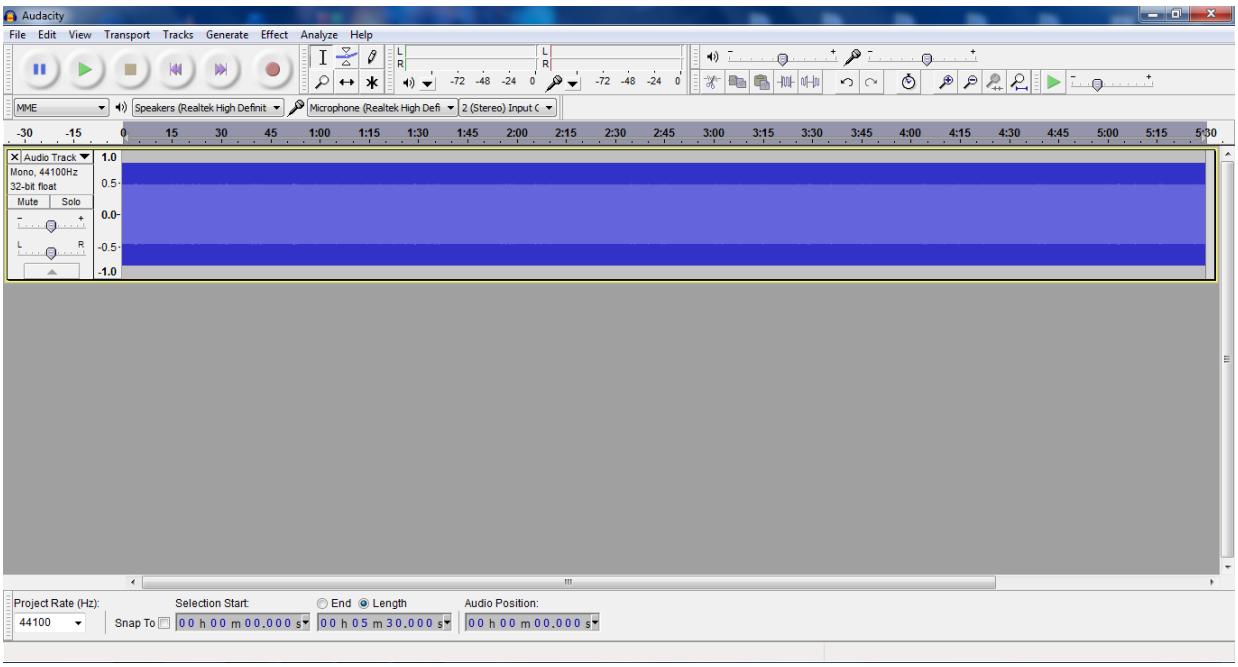


Figura 5.14: Ruido blanco.

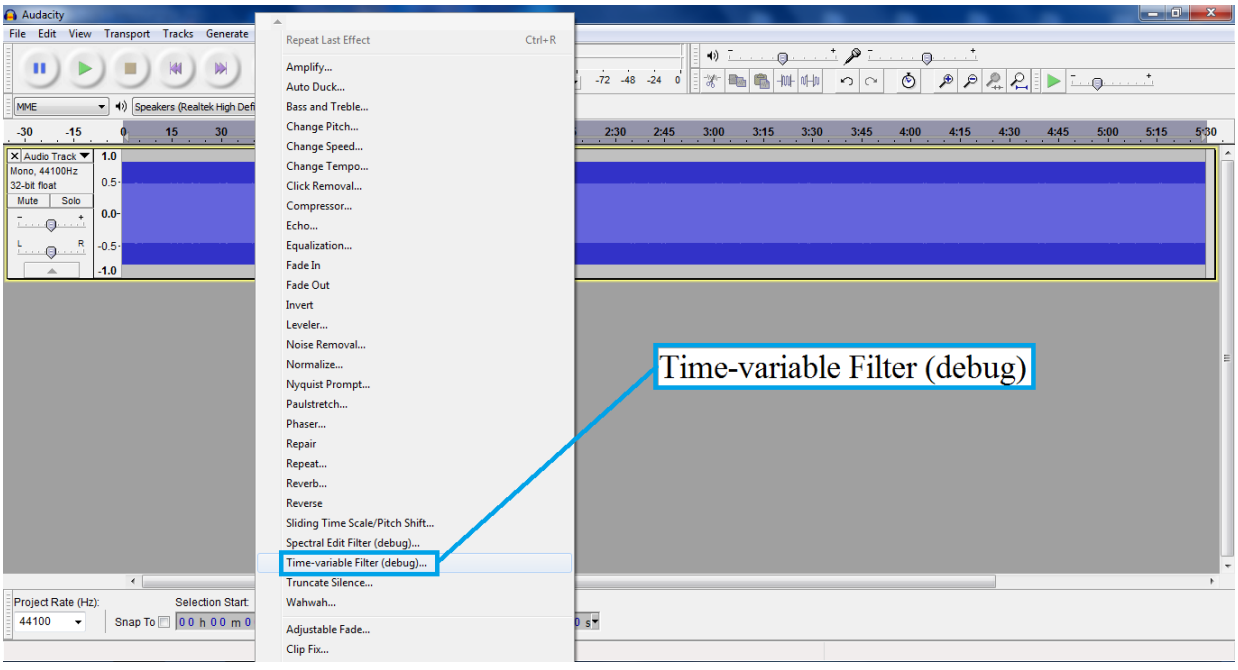


Figura 5.15: Selección del Plug-in.

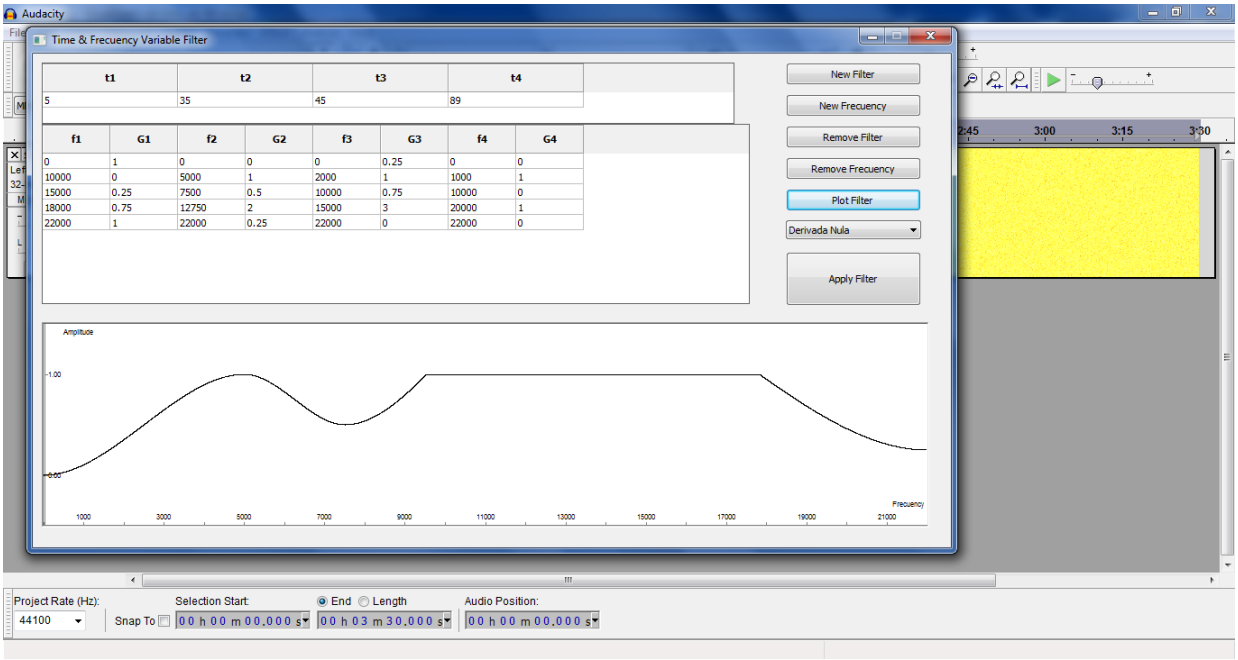


Figura 5.16: Interfaz gráfica del plug-in, definición de los puntos de paso y gráfica del segundo filtro.

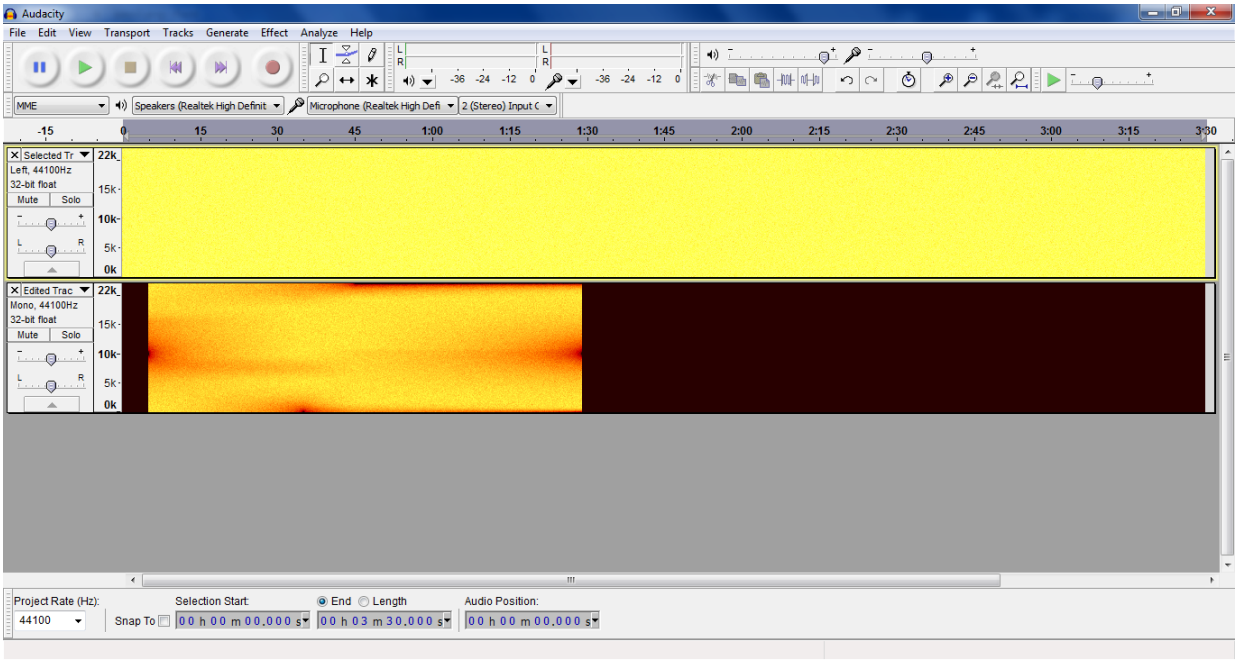


Figura 5.17: Se observa arriba el audio original y debajo el audio filtrado (utilizando un mapa de colores donde 0 es negro y 1 es amarillo).

En la Figura 5.18 se muestra el diagrama en bloques que corresponde al funcionamiento de

la herramienta antes explicada. Finalmente en la sección A.5 vemos una porción de código que permite obtener, partiendo de los puntos ingresados la interpolación Hermite Spline.

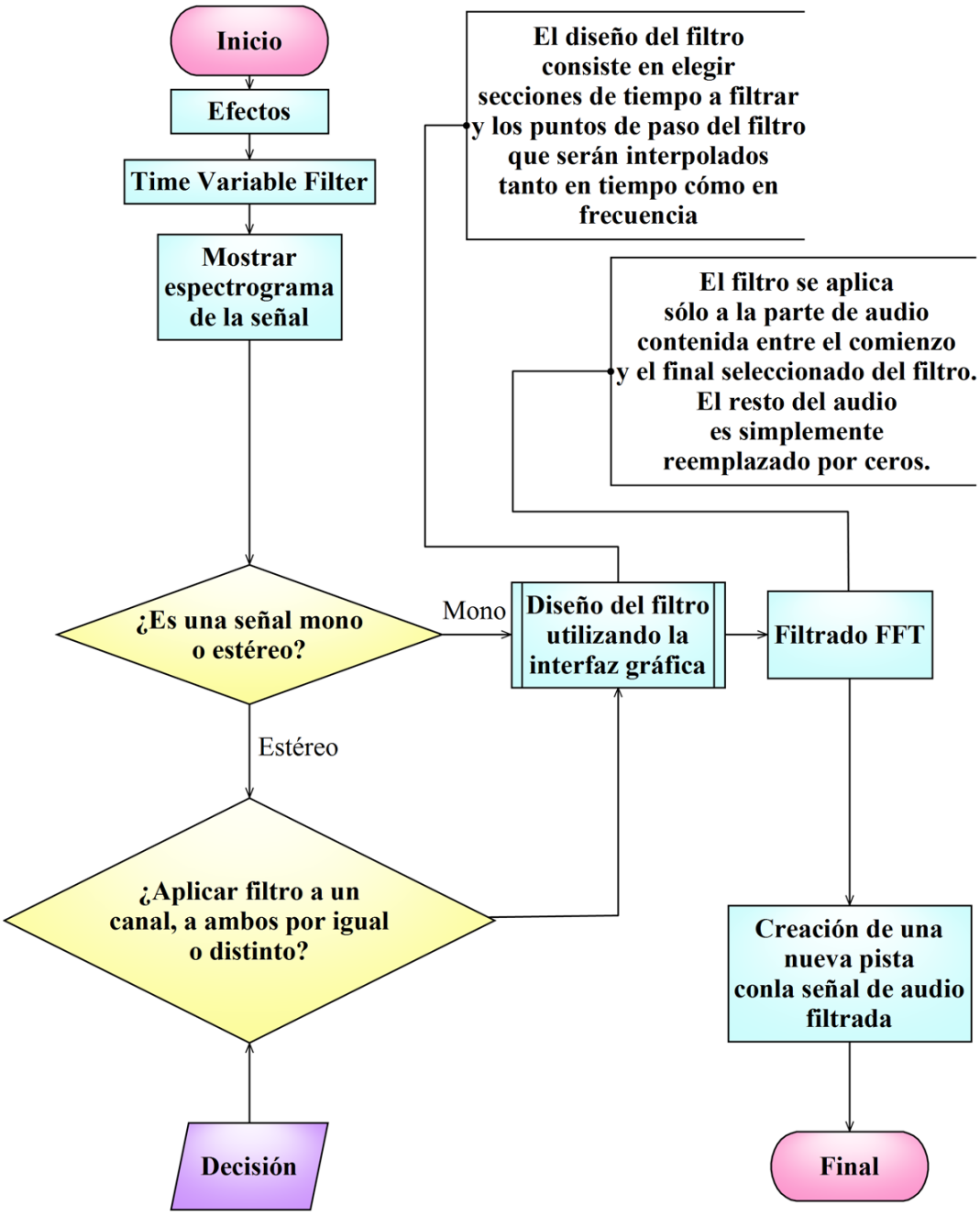


Figura 5.18: Diagrama en bloques del funcionamiento del filtro variable en tiempo y frecuencia.

5.4 Statistical Level Calc

Esta herramienta permite obtener, a partir de la señal o una porción de ella, las gráficas de densidad de probabilidad o de niveles estadísticos calculado además los valores de L_5, L_{10}, L_{50}, L_{90} y L_{95}, permitiendo seleccionar una ponderación temporal, una frecuencial, un rango de tiempo y, en el caso de que se trate de una señal estéreo, el canal a visualizar (izquierdo, derecho o ambos).

Los valores devueltos por el plug-in tienen diversas aplicaciones entre ellas:

1. Muy utilizados en varios criterios psicoacústicos que miden la variabilidad del ruido.
2. El L_{90} se toma generalmente como ruido de fondo debido a las fuentes cercanas y lejanas y el L_{95} como el ruido de fondo debido a fuentes lejanas.
3. El L_{10} o el L_5 se toman en reemplazo del máximo por ser valores mucho más repetibles que el máximo.
4. El L_{10} - L_{90} se denomina çlima del ruidoz es una medida de la variabilidad del ruido
5. Determinados ruidos, por ejemplo el del tránsito, tienen curvas de niveles estadísticos que siguen patrones específicos, podrían usarse para investigar sobre la determinación automática de la presencia de eventos extraños o anómalos .

Internamente el plug-in funciona de la siguiente manera

- Calcula el nivel de presión sonora a lo largo de toda la pista utilizando ambas ponderaciones, la temporal y la frecuencial, elegidas.
- Se ordenan los valores obtenidos de menor a mayor.
- Según el valor se le asigna una posición dentro de 100 rangos posibles de nivel, calculados entre el mayor nivel y el menor.
- Obtiene los puntos de las curvas y los valores L_5, L_{10}, L_{50}, L_{90} y L_{95} (ver código ubicado en la sección A.6).
- Grafica la/s curva/s dependiendo de las configuraciones seleccionadas y muestra los valores obtenidos. Para ello calcula la media aritmética

$$\bar{x} = \frac{1}{N}\sum_{i} = 1^{N} x_i \tag{5.3}$$

y el desvío estadar

$$\sigma = \sqrt{\frac{\sum_i = 1^N (x_i - \bar{x})^2}{N-1}} \tag{5.4}$$

y grafica los valores desde $\bar{x} - 4 \times \sigma$ hasta $\bar{x} + 4 \times \sigma$.

En las Figuras 5.19 y 5.20 se observa cómo varía la interfaz gráfica según se trate de una señal estéreo o mono y en la primera de ellas se muestran las opciones de cada menú, y en la Figura 5.21 se muestra un ejemplo de aplicación seleccionando una porción de audio (se observa que start time y end time del la interfaz de usuario son iguales al tiempo seleccionado en segundos).

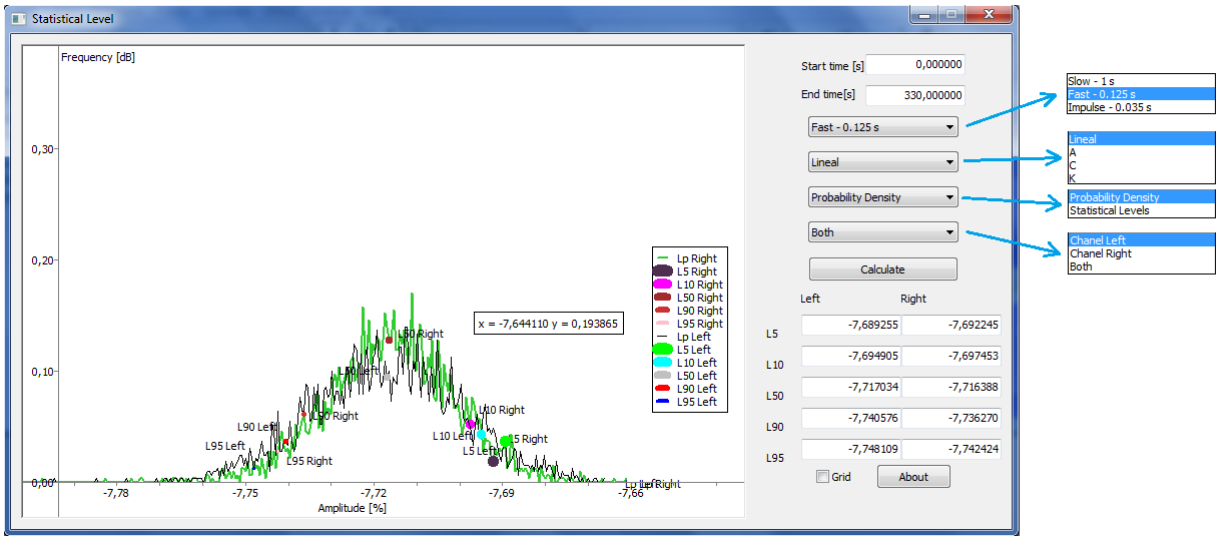


Figura 5.19: Interfaz gráfica correspondiente a una señal del tipo estéreo.

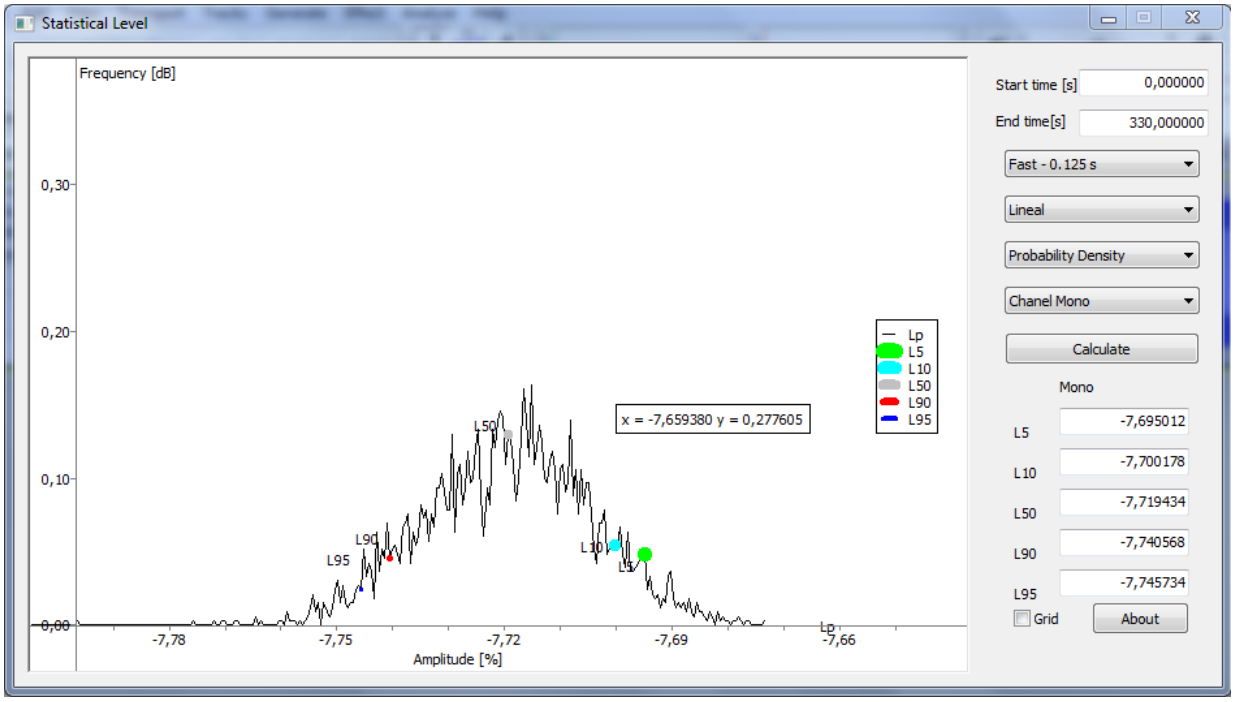


Figura 5.20: Interfaz gráfica para una señal del tipo mono.

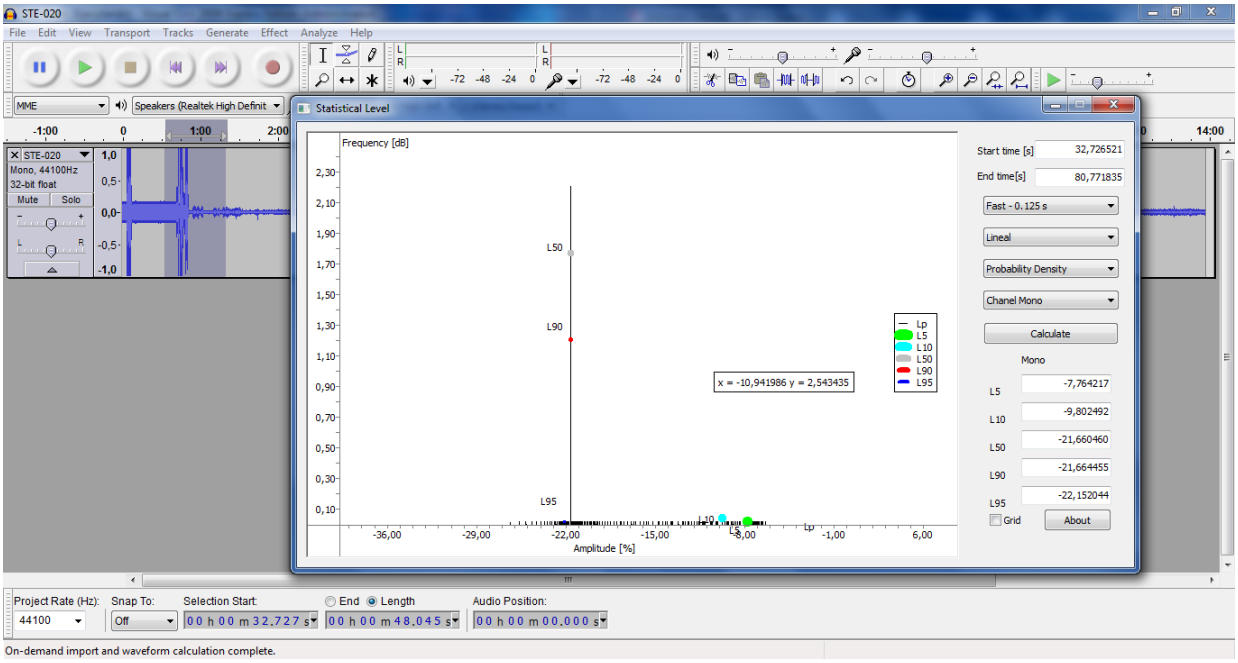


Figura 5.21: Ejemplo de funcionamiento utilizando una porción de audio.

En la Figura 5.22 vemos el diagrama en bloques del funcionamiento de la herramienta.

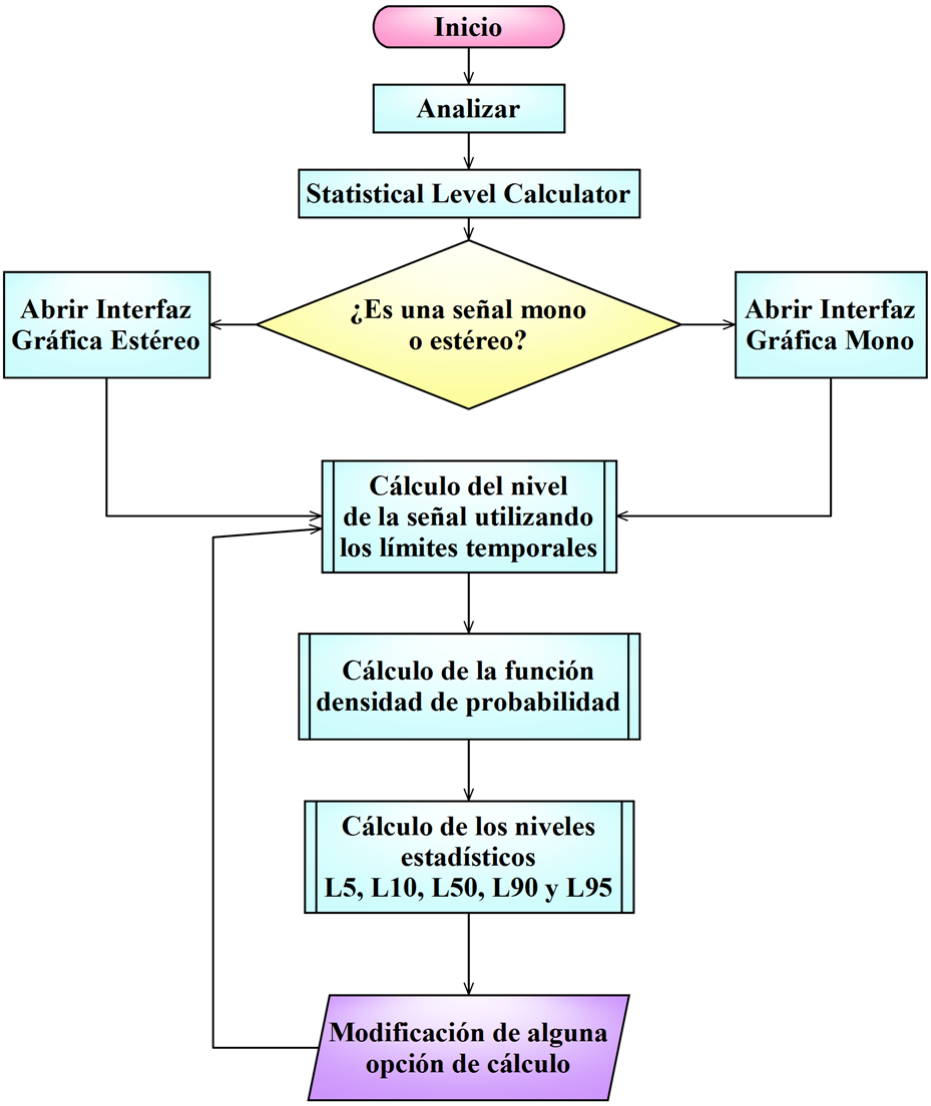


Figura 5.22: Diagrama en bloques de la herramienta Statistical Level Calculator.

Capítulo 6

Conclusiones

A lo largo del proyecto se logró obtener un conjunto de herramientas que no sólo acercan al software Audacity a su uso en investigación, sino que sirven como base sólida para el desarrollo de más aplicaciones útiles para ello. Si bien surgieron dificultades a medida que se avanzaba, se pudieron resolver para así lograr los objetivos propuestos.

El uso de software libre permite acceder a herramientas o programas que ya han sido desarrollados, como Audacity, wxWidgets, wxMathplot, FFTW, y modificarlos o expandirlos para darle la funcionalidad que uno desee. Esto le abre las puertas a toda persona que se encuentre interesada en ofrecer su trabajo y/o ideas para el beneficio propio y ajeno. Sobre el caso puntual de Audacity, cabe aclarar que existe un grupo amplio de gente que desea colaborar para poder lograr que el software mejore día a día.

Es necesario destacar que si bien se trata de un proyecto de ingeniería electrónica, lo alcanzado a lo largo del trabajo realizado es importante y deja antecedentes por ser una tarea interdisciplinaria, aplicándose a diversas ramas de estudio como la acústica, la psicoacústica, la fonoaudiología, entre otras.

Las posibles mejoras sobre lo desarrollado serían aumentar la resolución en frecuencia de los filtros, agregar distintos tipos de interpolación, lograr exportar los filtros diseñados dentro de las herramientas Spectral Edit Filter y Time Variable Filter y los valores de niveles estadísticos obtenidos para las diferentes ponderaciones temporales y frecuenciales. También es posible, en un futuro, desarrollar más herramientas de efectos y análisis de archivos de audio.

Bibliografía

[1] Federico Miyara: Mediciones acústicas basadas en software. Asociación de Acústicos Argentinos. Argentina, 2013 .

[2] Paolo Prandoni y Martin Vetterli: Signal processing for communications. EFPL Press. Italia, 2008.

[3] Federico Miyara: Procesamiento digital de señales. Transparencia.

[4] Juan Carlos Gomez: Análisis Fecuencial de Señales usando la Transformada Discreta de Fourier (DFT). Transparencia.

[5] Página Oficial de Audacity audacity.sourceforge.net/?lang=es

[6] Página Oficial de Audacity destinada a desarrolladores http://wiki.audacityteam.org/wiki/Developer_Guide

[7] Documentación wxWidgets 2.8 docs.wxwidgets.org/2.8/

[8] Página oficial del software Doxygen http://www.stack.nl/~dimitri/doxygen/.

[9] Manual Doxygen www.stack.nl/~dimitri/doxygen/manual/index.html

[10] Documentación librería wxMathPlot wxmathplot.sourceforge.net/docs/

[11] WAVE PCM soundfile format https://ccrma.stanford.edu/courses/422/projects/WaveFormat/

[12] Clasificación y Propiedades de las Señales http://cnx.org/contents/1d4b29f4-410f-4daa-ae2a-cad517cd8252@8/Clasificaci%C3%B3n_y_Propiedades_de.

[13] Filtros de ponderación http://en.wikipedia.org/wiki/Weighting_filter.

[14] Practical guidelines for Production and Implementation in accordance with EBU R 128 https://tech.ebu.ch/docs/tech/tech3343.pdf.

[15] Aliasing http://zone.ni.com/reference/en-XX/help/370051M-01/cvi/libref/analysisconcepts/aliasing/

[16] Señales y sistemas: Muestreo http://anyelosistemas.blogspot.com.ar/2011/08/senales-y-sistemas.html

[17] Miyara, Federico (2001) "¿Ruido o señal? La otra información. En defensa del registro digital del ruido urbano". 4ta Jornada Regional sobre Ruido Urbano, realizada en Montevideo, Uruguay, 14/07/01.

[18] Miyara, F.; Pasch, V.; Yanitelli, M.; Accolti, E.; Cabanellas, S., Miechi, P. (2009) "Contrastación de algoritmos de análisis de espectro con un instrumento normalizado". Actas de las Primeras Jornadas Regionales de Acústica AdAA 2009, Rosario, Argentina, Nov 19-20, 2009. http://www.fceia.unr.edu.ar/acustica/biblio/A032 (Miyara) Contraste algoritmos analizador normalizado.pdf

[19] Miyara, Federico (2000) "Control de Ruido" En "Jornadas Internacionales Multidisciplinarias sobre Violencia Acústica", edición en CD. ASOLOFAL. Rosario, Argentina, 2000.

[20] McAulay, Robert J.; Quatieri, Thomas F. (1986) "Speech Analysis/Synthesis Based on a Sinusoidal Representation". IEEE Transactions on Acoustics, Speech, and Signal Processing, Vol. ASSP-34, NO. 4, August 1986

[21] Kogan, P. (2012) "Paisaje sonoro: ¿Una metamorfosis de la acústica ambiental?. SONAC, Nro 3, Mayo 2012.

[22] Arenas, J. P. (2009): "Noise control by means of sound barriers", Primeras Jornadas Regionales de Acústica - AdAA 2009, Rosario, Argentina (2009). Conferencia en Actas. http://www.fceia.unr.edu.ar/acustica/biblio/AdAA_2009_Libro %20de %20actas.pdf

[23] Marengo Rodriguez, F. A.; Miyara, F. (2009). "Representación de Señales de Audio con Descomposición Empírica de Modos y Submuestreo Adaptativo," Primeras Jornadas Regionales en Acústica AdAA 2009, Noviembre 2009, Rosario.

[24] Marengo Rodriguez, F. A.; Miyara, F. (2012): "Nuevo codificador digital de audio", Concepto Innovador presentado al concurso INNOVAR 2012, ID 13620 (2012), Argentina.

[25] Marengo Rodriguez F.A., Miyara F. (2008b) "Caracterización de ruido vehicular por supresión del efecto Doppler," VI Congreso Iberoamericano de Acústica (FIA), A034. Buenos Aires, Argentina. 2008.

[26] Marengo Rodriguez F.A., Miyara F. (2009) "Parameters extraction and aural simulation of traffic noise by means of a novel Doppler removal method ". XIII Reunión de trabajo en procesamiento de la información y control (RPIC), 182, Pp. 88-92. Rosario, Argentina. 2009.

[27] Marengo Rodriguez, F.A.; Miyara, F. (2009): "Design and implementation of a Doppler removal method for parameters extraction and aural simulation of traffic noise". Mecánica Computacional, No 28, pp. 89-100.

[28] Marengo Rodriguez, F. A.; Miyara, F.; Mignini, E. (2010): "Compensación del efecto Doppler y caracterización del ruido vehicular para su simulación aural", Anales del 2do Congreso Internacional de Acústica UNTreF, Bs. As (2010).

[29] Marengo Rodriguez, F. A., Accolti, E.; Miyara, F. (2011): "Blind Doppler shift compensation and its characterization for traffic noise simulation". Mecánica Computacional, No 30 (2011), pp. 3187-3199.

[30] Tommasini, F. C. (2012): "Sistema de simulación acústica virtual en tiempo real", Tesis Doctoral, Universidad Nacional de Córdoba, Facultad de Ciencias Exactas, Físicas y Naturales, Córdoba, Argentina (2012). http://www.investigacion.frc.utn.edu.ar/cintra/pub/file/tesis_doctoral_tommasini_fabian_2012.pdf

[31] Página oficial Inno Setup http://www.jrsoftware.org/isinfo.php

Apéndice A
Extractos de códigos

A.1 Analizador de espectro

```
void FreqWindow::OnTimer(wxTimerEvent& event)
{
   if(mAutoUpdateON && mTimer->IsRunning()){
      if((p->mViewInfo.sel2 != Lastsel2 || mWindowSize != LastWinSize) && p->mViewInfo.sel1 == p
          ->mViewInfo.sel2){
         LastWinSize = mWindowSize;
         Lastsel2 = p->mViewInfo.sel2;
         Lastsel1 = p->mViewInfo.sel1;
         TrackListIterator iter(p->GetTracks());
         Track *t = iter.First();
         WaveTrack *track = (WaveTrack *)t;
         mDataLen = mWindowSize;
         long center = track->TimeToLongSamples(p->mViewInfo.sel2);
         p->mViewInfo.sel0 = track->LongSamplesToTime(center-mDataLen/2);
         p->mViewInfo.sel1 = track->LongSamplesToTime(center+mDataLen/2);
         GetAudio();
         Plot();
         SetFocus();
      }if(p->mViewInfo.sel2 != Lastsel2 || p->mViewInfo.sel1 != Lastsel1 || p->mViewInfo.sel0 !=
           Lastsel0){
         LastWinSize = mWindowSize;
         bool change = p->mViewInfo.sel1 != Lastsel1 || p->mViewInfo.sel0 != Lastsel0;
         Lastsel2 = p->mViewInfo.sel2;
         Lastsel1 = p->mViewInfo.sel1;
         Lastsel0 = p->mViewInfo.sel0;
         TrackListIterator iter(p->GetTracks());
         Track *t = iter.First();
         WaveTrack *track = (WaveTrack *)t;
         mDataLen = mWindowSize;
         long start = track->TimeToLongSamples(p->mViewInfo.sel0);
         long end = track->TimeToLongSamples(p->mViewInfo.sel1);
         if(start > end){
            long aux = start;
            start = end;
            end = aux;
         }
         if(((end-start) <mWindowSize) ){
            p->mViewInfo.sel0 = track->LongSamplesToTime((end-start-mDataLen)/2+start);
            p->mViewInfo.sel1 = track->LongSamplesToTime((end-start+mDataLen)/2+start);
         }
         GetAudio();
         Plot();
         SetFocus();
      }
   }
}
```

Listing A.1: Código que permite actualizar automáticamente la ventana del analizador de espectro.

A.2 Visor de nivel

```
bool WaveClip::SetLevel(float *buffer, sampleFormat format,
                  sampleCount start, sampleCount len, float tau, int pond)
{
   //Se le pasa a la función el valor del tau para realizar la ponderación temporal
   //y el tipo de ponderación frecuencial
   float b0 = 1/(1+mRate*tau);
   float a1 = -(mRate*tau)/(mRate*tau+1);
   int cant = len/mRate*10;
   sampleCount llen = len*10/mRate;
   float *filtered = new float[mRate];
   float *subfilt = new float[llen];
   float *filter = new float[len];
   int sec = len/mRate;
   float x=0,y=0,y1=0;
   //A-weighting 44100 Hz
   if(pond == 1){
      double reg0[3], reg1[3], reg2[3];
      double a2[3], a1[3], a0[3], b2[3], b1[3], b0[3],g;
      a0[0] = 1;
      a0[1] = 1;
      a0[2] = 1;
      a1[0] = -0.14053608;
      a1[1] = -1.88490121;
      a1[2] = -1.99413888;
      a2[0] = 0.00493759;
      a2[1] = 0.88642147;
      a2[2] = 0.99414746;
      b0[0] = 1;
      b0[1] = 1;
      b0[2] = 1;
      b1[0] = 2;
      b1[1] = -2;
      b1[2] = -2;
      b2[0] = 1;
      b2[1] = 1;
      b2[2] = 1;
      g = 0.2557411;
      for(int j=0; j<3; j++){
         reg0[j] = 0.0;
         reg1[j] = 0.0;
         reg2[j] = 0.0;
      }
      //Separación en cuadros
      for(int j=0;j<sec;j++){
         for(int i=0;i<mRate;i++){
            int delta = j*mRate;
            *(filtered+i) = *(buffer+i+delta);
         }
         //Filtrado de ponderación frecuencial
         for(int i=0;i<mRate;i++){
            for(int k=0;k<3;k++){
               x= *(filtered+i);
               reg0[k] = x - a1[k] * reg1[k] - a2[k] * reg2[k];
               y = g*(b0[k] * reg0[k] + b1[k] * reg1[k] + b2[k] * reg2[k]);
               reg2[k] = reg1[k];
               reg1[k] = reg0[k];
               *(filtered+i)=y;
            }
         }
         //Filtrado de ponderación temporal
         for(int i=0;i<mRate;i++){
            int delta = j*mRate;
```

```
            float b0 = 1/(1+mRate*tau);
            float a1 = -(mRate*tau)/(mRate*tau+1);
            x=*(filtered+i)**(filtered+i);
            y= b0*x-a1*y1;
            y1 = y;
            *(filtered+i)=y;
            x=0;
            if(i==0 || i%(mRate/10)==0 || i==mRate){
               int pos = i*10/mRate;
               *(subfilt+pos+10*j) = *(filtered+i);
            }
            *(filter+i+j*mRate) = *(filtered+i);
         }
      }
   }
   //C-weighting 44100 Hz
   else if(pond == 2){
      double reg0[2], reg1[2], reg2[2];
      double a2[2], a1[2], a0[2], b2[2], b1[2], b0[2],g;
      a0[0] = 1;
      a0[1] = 1;
      a1[0] = -0.14053608;
      a1[1] = -1.99413888;
      a2[0] = 0.00493759;
      a2[1] = 0.99414746;
      b0[0] = 1;
      b0[1] = 1;
      b1[0] = 2;
      b1[1] = -2;
      b2[0] = 1;
      b2[1] = 1;
      g = 0.2170085;
      for(int j=0; j<2; j++){
         reg0[j] = 0.0;
         reg1[j] = 0.0;
         reg2[j] = 0.0;
      }
      //Separación en cuadros
      for(int j=0;j<sec;j++){
         for(int i=0;i<mRate;i++){
            int delta = j*mRate;
            *(filtered+i) = *(buffer+i+delta);
         }
         //Filtrado de ponderación frecuencial
         for(int i=0;i<mRate;i++){
            for(int k=0;k<2;k++){
               x= *(filtered+i);
               reg0[k] = x - a1[k] * reg1[k] - a2[k] * reg2[k];
               y = g*(b0[k] * reg0[k] + b1[k] * reg1[k] + b2[k] * reg2[k]);
               reg2[k] = reg1[k];
               reg1[k] = reg0[k];
               *(filtered+i)=y;
            }
         }
         //Filtrado de ponderación temporal
         for(int i=0;i<mRate;i++){
            int delta = j*mRate;
            float b0 = 1/(1+mRate*tau);
            float a1 = -(mRate*tau)/(mRate*tau+1);
            x=*(filtered+i)**(filtered+i);
            y= b0*x-a1*y1;
            y1 = y;
            *(filtered+i)=y;
            x=0;
            if(i==0 || i%(mRate/10)==0 || i==mRate){
```

```
            int pos = i*10/mRate;
            *(subfilt+pos+10*j) = *(filtered+i);
         }
         *(filter+i+j*mRate) = *(filtered+i);
      }
   }
}
//Z-weighting 44100 Hz
else if(pond == 3){
   //Separación en cuadros
   for(int j=0;j<sec;j++){
   //Filtrado de ponderación temporal
      for(int i=0;i<mRate;i++){
         int delta = j*mRate;
         x=*(buffer+i+delta)**(buffer+i+delta);
         y= b0*x-a1*y1;
         y1 = y;
         *(filtered+i)=y;
         x=0;

         if(i==0 || i%(mRate/10)==0 || i==mRate){
            int pos = i*10/mRate;
            *(subfilt+pos+10*j) = *(filtered+i);
         }
         *(filter+i+j*mRate) = *(filtered+i);
      }
   }
}
//K-weighting 48000 Hz
else if(pond == 4){
   double reg0[2], reg1[2], reg2[2];
   double a2[2], a1[2], a0[2], b2[2], b1[2], b0[2],g;
   a0[0] = 1;
   a0[1] = 1;
   a1[0] = -1.06965929;
   a1[1] = -1.99004745;
   a2[0] = 0.732480774;
   a2[1] = 0.99007225;
   b0[0] = 1.53512485;
   b0[1] = 1;
   b1[0] = -2.6916961;
   b1[1] = -2;
   b2[0] = 1.1983928;
   b2[1] = 1;
   g = 1;
   for(int j=0; j<2; j++){
      reg0[j] = 0.0;
      reg1[j] = 0.0;
      reg2[j] = 0.0;
   }
   //Separación en cuadros
   for(int j=0;j<sec;j++){
      for(int i=0;i<mRate;i++){
         int delta = j*mRate;
         *(filtered+i) = *(buffer+i+delta);
      }
      //Filtrado de ponderación frecuencial
      for(int i=0;i<mRate;i++){
         for(int k=0;k<2;k++){
            x= *(filtered+i);
            reg0[k] = x - a1[k] * reg1[k] - a2[k] * reg2[k];
            y = g*(b0[k] * reg0[k] + b1[k] * reg1[k] + b2[k] * reg2[k]);
            reg2[k] = reg1[k];
            reg1[k] = reg0[k];
            *(filtered+i)=y;
```

```
            }
         }
         //Filtrado de ponderación temporal
         for(int i=0;i<mRate;i++){
            int delta = j*mRate;
            float b0 = 1/(1+mRate*tau);
            float a1 = -(mRate*tau)/(mRate*tau+1);
            x=*(filtered+i)**(filtered+i);
            y= b0*x-a1*y1;
            y1 = y;
            *(filtered+i)=y;
            x=0;
            if(i==0 || i%(mRate/10)==0 || i==mRate){
               int pos = i*10/mRate;
               *(subfilt+pos+10*j) = *(filtered+i);
            }
            *(filter+i+j*mRate) = *(filtered+i);
         }
      }
   }
   realRate = mRate;
   haslevel=true;
   lSequence->Delete(0,llen);
   bool asd = lSequence->Append((samplePtr)subfilt, floatSample, llen);
   delete subfilt;
   delete filtered;
   MarkChanged();
   return true;
}
```

Listing A.2: Código que permite calcular el nivel de la señal.

A.3 Importador y exportador de metadatos

```
wxString type = fileName.Right(3);
wxString wav = _("wav");
int iswav = type.Cmp(wav);
if(!iswav){
   LabelReader *lReader = new LabelReader(fileName);
   if(lReader->hasLabels(fileName)){
      wxMessageDialog *myDialog = new wxMessageDialog(NULL,_T("Do␣you␣want␣to␣load␣the␣labels␣
          from␣this␣wav␣file?"),_T("Load␣labels"),wxYES_NO,wxDefaultPosition);
      if(myDialog->ShowModal() == wxID_YES){
         lReader->ReadLabel();
      }
   }
}
```

Listing A.3: Porción de código que indica si se seleccionó un archivo del formato .wav.

```
bool LabelReader::hasLabels(wxString &fileName){
   //Abre el archivo y obtiene el tamaño
   pWav = fopen(fileName.mb_str(),"rb");
   if (pWav==NULL) {return false;}

   fseek (pWav , 0 , SEEK_END);
   pTamaño = ftell (pWav);
   rewind (pWav);
   buffer = (char*) malloc (sizeof(char)*pTamaño);

   fread (buffer,1,pTamaño,pWav);
```

```
   rewind (pWav);
   //Lee y compara los primeros cuatro caracteres con "RIFF"
   fread (id1,sizeof(char), 4,pWav);
   if(memcmp(id1,"RIFF",4)==0){
   //Lee el tamaño de archivo que indica el encabezado y lo guarda en memoria
   fread(sizefile, sizeof(int), 1, pWav);
   *(sizefile) +=8;
   memcpy(fileSize,sizefile,4);
   //Lee y compara los siguientes cuatro caracteres con "WAVE" si son iguales
   //guarda los valores del resto del encabezado
   fread(id2, sizeof(char), 4, pWav);
      if(memcmp(id2,"WAVE",4)==0){
         fread(id3, sizeof(char), 4, pWav);
         fread(format_length, sizeof(int),1,pWav);
         fread(format_tag, sizeof(short), 1, pWav);
         fread(channels, sizeof(short),1,pWav);
         fread(sample_rate, sizeof(int), 1, pWav);
         fread(avg_bytes_sec, sizeof(int), 1, pWav);
         fread(block_align, sizeof(short), 1, pWav);
         fread(bits_per_sample, sizeof(short), 1, pWav);
      }
   }
   //Lee y compara los caracteres que inician el bloque de datos con "data" y
   //guarda el tamaño de los datos de audio que indica el archivo
   fread (id4,sizeof(char),4,pWav);
   if(memcmp(id4,"data",4)==0){
      fread (sizedata,sizeof(int),1,pWav);
      memcpy(dataSize,sizedata,4);
   }

   fclose (pWav);
   //Decide si el archivo tiene o no metadatos comparando el tamaño del mismo
   //con el la cantidad de datos de audio más el tamaño del archivo y devuelve
   //una respuesta
   if(dataSize[0]+44 == fileSize[0]){
      return false;
   }else{
      return true;
   }
}
```

Listing A.4: Función que devuelve verdadero si el archivo tiene metadatos y falso en caso contrario.

```
//creo una LabelTrack llamada "Cues"
AudacityProject *p = GetActiveProject();
TrackFactory *mFactory = p->GetTrackFactory();
TrackList *mTracks = p->GetTracks();

LabelTrack *tr1 = mFactory->NewLabelTrack();
tr1->SetName(_T("Cues"));

for (int i=0;i<cant_cue[0];i++){
   //obtengo el la etiqueta
   char * label = (char*) malloc(sizeof(char)*(*(labl_length+i)));;
   memcpy(label,labl,(*(labl_length+i)));
   labl+=(*(labl_length+i));
   wxString mystring = wxString::FromAscii(label);
   mystring = mystring.Left(*(labl_length+i));
   //obtengo las marcas de comienzo y de final
   int nmi = *(num_muest+i);
   int sp = sample_rate[0];
   double t1 = (double)nmi/(double)sp;
   int nmf = *(cant_ltxt+i);
```

```
   double t2 = (double)nmf/(double)sp+t1;
   bool tn = false;
   //verifico si la marca tiene nota y en caso afirmativo la obtengo
   //y creo una marca con la etiqueta y la nota
   for(int j=0; j<cant_cue[0];j++){
      if(*(num_note+j)==*(num_cue+i)){
         char * notex = (char*) malloc(sizeof(char)*(*(note_length+j)));;
         memcpy(notex,note,(*(note_length+j)));
         note+=(*(note_length+j));
         wxString mystring2 = wxString::FromAscii(notex);
         mystring2 = mystring2.Left(*(note_length+j));
         int m = tr1->AddLabel(t1,t2,mystring,mystring2);
         tn = true;
      }
   }
   //en caso de no tener creo una marca con etiqueta y sin nota
   if(!tn)
   int m = tr1->AddLabel(t1,t2,mystring,wxT(""));
}
```

Listing A.5: Código que permite crear una LabelTrack y añadirle las marcas importadas.

```
bool LabelWriter::WriteLabel(TrackListIterator *pIter)
{
   AudacityProject *p = GetActiveProject();

   double mRate = p->GetRate();
   pWav = fopen(file.mb_str(),"r+b");
   if (pWav==NULL) {return false;} //si no se abrio salgo
   fseek (pWav , 0 , SEEK_END);// el archivo se abre al final donde se guardaran los metadatos

   int cantlabtotal = 0;

   //Se obtiene el número de cues a agregar
   Track *t = pIter->First();
   while (t) {
      if (t->GetKind() == 3){
         cantlabtotal += (int)((LabelTrack *) t)->GetNumLabels();
      }
      t = pIter->Next();
   }
   t = pIter->First();

   double *start,*end;
   int *longlabi,*delta,*longnotei;
   start = (double *) malloc(sizeof(double)*cantlabtotal);
   end = (double *) malloc(sizeof(double)*cantlabtotal);
   delta = (int *) malloc(sizeof(int)*cantlabtotal);
   longlabi = (int *) malloc(sizeof(int)*cantlabtotal);
   longnotei = (int *) malloc(sizeof(int)*cantlabtotal);

   int labnum,o=0,longlab =0,longnote =0;
   wxString labels,notes;

   //Recorro pista por pista para guardar las marcas de inicio, de fin (en realidad se
   //necesita la cantidad de muestras entre inicio y fin), la etiqueta y la nota

   while (t) {
      if (t->GetKind() == 3){  //Si es del tipo LabelTrack
         int cantlab = (int)((LabelTrack *) t)->GetNumLabels();  //cantidad de etiquetas
         for(int i=0;i<cantlab;i++){
            const LabelStruct *lab= ((LabelTrack *) t)->GetLabel(i);
            *(start+i+o) = lab->t; //inicio
            *(end+i+o) = lab->t1;  //fin
```

```
            *(delta+i+o) = (int) floor((*(end+i+o)-*(start+i+o))*mRate+0.5); //delta
            longlab += lab->title.Len(); //etiqueta
            *(longlabi+i+o)= lab->title.Len();
            labels.Append(lab->title);
            longnote += lab->note.Len(); //nota
            *(longnotei+i+o)= lab->note.Len();
            notes.Append(lab->note);
         }
         o+=cantlab;
      }
      t = pIter->Next(); //Salta a la siguiente Track
   }

   //Grabo el bloque de metadatos "cue " (contiene el inicio)
   fwrite("cue␣",sizeof(char),4,pWav);
   int *longitud,*cantlabtot;
   longitud = (int *) malloc(sizeof(int));
   cantlabtot = (int *) malloc(sizeof(int));
   *(longitud)= 24*cantlabtotal+4;
   *(cantlabtot) = cantlabtotal;
   fwrite(longitud,sizeof(int),1,pWav);
   fwrite(cantlabtot,sizeof(int),1,pWav);
   int *ncue,*muesasoc,*inifrag,*iniblo,*offs;
   ncue = (int *) malloc(sizeof(int));
   muesasoc = (int *) malloc(sizeof(int));
   inifrag = (int *) malloc(sizeof(int));
   iniblo = (int *) malloc(sizeof(int));
   offs = (int *) malloc(sizeof(int));
   for(int i=0;i<cantlabtotal;i++){
      *(ncue)=i+1;
      *(muesasoc)= *(start+i)*mRate;
      *(inifrag)=0;
      *(iniblo)=0;
      *(offs)=*(muesasoc);
      fwrite(ncue,sizeof(int),1,pWav);
      fwrite(muesasoc,sizeof(int),1,pWav);
      fwrite("data",sizeof(char),4,pWav);
      fwrite(inifrag,sizeof(int),1,pWav);
      fwrite(iniblo,sizeof(int),1,pWav);
      fwrite(offs,sizeof(int),1,pWav);
   }

   //Grabo el bloque de metadatos "LIST" (contiene la duración de la marca,
   //es decir, el delta)
   int cuenta=0;
   for(int i=0;i<cantlabtotal;i++){
      cuenta += *(longlabi+i);
      if(*(longlabi+i)%2 == 0)
         cuenta++;
   }
   int *longadtl = (int *) malloc(sizeof(int));
   *(longadtl) = 28*cantlabtotal+13*cantlabtotal+cuenta;

   fwrite("LIST",sizeof(char),4,pWav);
   fwrite(longadtl,sizeof(int),1,pWav);
   fwrite("adtl",sizeof(char),4,pWav);

   int *leng,*nmarc,*cantmues;
   short *pais,*idioma,*dialecto,*codpag;
   leng = (int *) malloc(sizeof(int));
   nmarc = (int *) malloc(sizeof(int));
   cantmues = (int *) malloc(sizeof(int));
   idioma = (short *) malloc(sizeof(short));
   pais = (short *) malloc(sizeof(short));
   dialecto = (short *) malloc(sizeof(short));
```

```
codpag = (short *) malloc(sizeof(short));

for(int i=0;i<cantlabtotal;i++){
   *(leng) = 20;
   *(nmarc) = i;
   *(cantmues) = *(delta+i);
   *(pais) = 0;
   *(dialecto) = 0;
   *(idioma) = 0;
   *(codpag) = 0;
   fwrite("ltxt",sizeof(char),4,pWav);
   fwrite(leng,sizeof(int),1,pWav);
   fwrite(nmarc,sizeof(int),1,pWav);
   fwrite(cantmues,sizeof(int),1,pWav);
   fwrite("rgn␣",sizeof(char),4,pWav);
   fwrite(pais,sizeof(short),1,pWav);
   fwrite(idioma,sizeof(short),1,pWav);
   fwrite(dialecto,sizeof(short),1,pWav);
   fwrite(codpag,sizeof(short),1,pWav);
}

//Grabo el bloque de metadatos "labl" (contiene la etiqueta)
int *nummar,*length;
char *etiq,*n;

etiq = (char *) malloc(sizeof(char)*longlab);
n = (char *) malloc(sizeof(char));
*(n) = 0;
strncpy(etiq, (const char*)labels.utf8_str(), longlab);
length = (int *) malloc(sizeof(int));
nummar = (int *) malloc(sizeof(int));
for(int i=0;i<cantlabtotal;i++){
   fwrite("labl",sizeof(char),4,pWav);
   *(nummar) =i+1;
   *(length) = *(longlabi+i)+4+1;
   fwrite(length,sizeof(int),1,pWav);
   fwrite(nummar,sizeof(int),1,pWav);
   fwrite(etiq,sizeof(char),*(longlabi+i),pWav);
   if(*(longlabi+i)%2 ==0)
      fwrite(n,sizeof(char),1,pWav);
   fwrite(n,sizeof(char),1,pWav);
   etiq +=*(longlabi+i);
}

//Grabo el bloque de metadatos "note" (contiene las notas)
char *nota;

nota = (char *) malloc(sizeof(char)*longlab);
n = (char *) malloc(sizeof(char));
*(n) = 0;
strncpy(nota, (const char*)notes.utf8_str(), longnote);
length = (int *) malloc(sizeof(int));
nummar = (int *) malloc(sizeof(int));
for(int i=0;i<cantlabtotal;i++){
   if(*(longnotei+i) != 0){
      fwrite("note",sizeof(char),4,pWav);
      *(nummar) =i+1;
      *(length) = *(longnotei+i)+4+1;
      fwrite(length,sizeof(int),1,pWav);
      fwrite(nummar,sizeof(int),1,pWav);
      fwrite(nota,sizeof(char),*(longnotei+i),pWav);
      if(*(longnotei+i)%2 ==0)
         fwrite(n,sizeof(char),1,pWav);
      fwrite(n,sizeof(char),1,pWav);
      nota +=*(longnotei+i);
```

```
      }
   }

   //Modifico el encabezado original para que se sepa que existen metadatos
   rewind (pWav);

   fseek (pWav , 0 , SEEK_END);
   pTamaño = ftell (pWav);
   rewind (pWav);

   int *tam = (int *) malloc(sizeof(int));
   *(tam) = 0;
   fseek(pWav,4,SEEK_CUR);
   fread(tam,sizeof(int),1,pWav);
   *(tam) = pTamaño-8;
   rewind (pWav);
   fseek(pWav,4,SEEK_CUR);
   fwrite(tam,sizeof(int),1,pWav);

   free(longlabi);
   free(start);
   free(end);
   free(delta);
   fclose (pWav);
}
```

Listing A.6: Código para grabar los metadatos en un archivo de formato WAVE.

A.4 Spectral Edit Filter

```
void fftfilter(float *x, int Nx, float *w, int Nw, float *y){
   int i,j,Q,N,L;

   N = Nw;
   Q = Nx/((float)N/2); // Es similar a  floor(Nx/(N/2))
   L = (Q+1)*(N/2);

   // --- Asigno espacio ------------------------------------------------
   double **xx_re;
   double **xx_im;
   xx_re = (double **)malloc(sizeof(double *)*Q);
   xx_im = (double **)malloc(sizeof(double *)*Q);
   for(i=0;i<Q;i++){
      xx_re[i] = (double *)malloc(sizeof(double)*N);
      xx_im[i] = (double *)malloc(sizeof(double)*N);
   }

   // --- Separo en frames ----------------------------------------------
   for(i=0;i<Q-1;i++){
      for(j=0;j<N;j++){
         xx_re[i][j] = x[i*(N/2) + j];
         xx_im[i][j] = 0.0;
      }
   }
   // Ultimo frame, parte señal y relleno de ceros...
   int aux1,aux2;
   aux1 = (Q-1)*(N/2);
   aux2 = Nx - aux1;
   for(i=0;i<aux2;i++){
      xx_re[Q-1][i] = x[aux1 + i];
      xx_im[Q-1][i] = 0.0;
   }
```

```
  for(i=aux2;i<N;i++){
     xx_re[Q-1][i] = 0.0;
     xx_im[Q-1][i] = 0.0;
  }

  // --- Calculo FFT ----------------------------------------------------
  for(i=0;i<Q;i++){
     fft(xx_re[i],xx_im[i],N);
  }

  // --- Multiplico por filtro ------------------------------------------
  for(i=0;i<Q;i++){
     for(j=0;j<N;j++){
        xx_re[i][j] = xx_re[i][j]*w[j+i*4096];
        xx_im[i][j] = xx_im[i][j]*w[j+i*4096];
     }
  }

  // --- Calculo IFFT ---------------------------------------------------
 for(i=0;i<Q;i++){
     ifft(xx_re[i],xx_im[i],N);
  }

  // --- Preparo ventanas INICIAL y FINAL -------------------------------
  float * hann;
  hann = hanning(N,1);
  double *hann1, *hann2;
  hann1 = (double *)malloc(sizeof(double)*N);
  hann2 = (double *)malloc(sizeof(double)*N);
  for(i=0;i<N/2;i++){
     hann1[i] = 1.0;
     hann2[i] = hann[i];
  }
  for(i=N/2;i<N;i++){
     hann1[i] = hann[i];
     hann2[i] = 1.0;
  }

  // --- Genero matrices de frames pares e impares ----------------------
  double *z1, *z2;
  z1 = (double *)malloc(sizeof(double)*L);
  z2 = (double *)malloc(sizeof(double)*L);
  for(i=0;i<L;i++){    // Inicializo a 0...
     z1[i] = 0.0;
     z2[i] = 0.0;
  }

  // --- Asigno impares -------------------------------------------------
  long k_impar;
  for(i=0;i<N;i++)
  z1[i] = xx_re[0][i]*hann1[i];

  k_impar = N-1;
  for(i=2;i<Q-1;i=i+2){
     for(j=0;j<N;j++){
        k_impar++;
        z1[k_impar] = xx_re[i][j]*hann[j];
     }
  }

  // --- Asigno pares ---------------------------------------------------
  long k_par;
  k_par = (N/2)-1;
  for(i=1;i<Q-1;i=i+2){
     for(j=0;j<N;j++){
```

```
            k_par++;
            z2[k_par] = xx_re[i][j]*hann[j];
        }
    }

    // --- Asigno ultimo frame ------------------------------------------------
    if (Q & 1){
        //printf("%i is odd\n", n);
        for(j=0;j<N;j++){
            k_impar++;
            z1[k_impar] = xx_re[Q-1][j]*hann2[j];
        }
    }
    else{
        //printf("%i is even\n", n);
        for(j=0;j<N;j++){
            k_par++;
            z2[k_par] = xx_re[Q-1][j]*hann2[j];
        }
    }

    // --- Genero vector final ------------------------------
    for(i=0;i<Nx;i++)
        y[i] = z1[i] + z2[i];

    // --- Libero Memoria -----------------------------------
    for(i=0;i<Q;i++){
        free(xx_re[i]);
        free(xx_im[i]);
    }
    free(xx_re);
    free(xx_im);
    free(hann);
    free(hann1);
    free(hann2);
    free(z1);
    free(z2);
}
```

Listing A.7: Código para realizar el filtrado FFT.

A.5 Time Variable Filter

```
//Recibe como argumentos los puntos de paso en f (frecuencia) y g (ganancia),
//los vectores de salida x e y, la cantidad de columnas de la tabla y el tipo de derivada
void Hermite(int *f,float *g, int *x, float *y,int cantrow,int tipder){
    float *frecs = (float *) malloc(sizeof(float)*22000);
    for (int j=0;j<cantrow-1;j++){
        for (int i=*(f+j);i<*(f+j+1);i++){
            *(frecs+i) = (i-(float)*(f+j))/((float)*(f+j+1)-(float)*(f+j));
        }
    }
    float *deriv = (float *) malloc(sizeof(float)*cantrow);
    for(int i=0;i<22000;i++){
        *(x+i) = i;
    }
    switch(tipder){
        case 0: //derivada nula
            for(int i=0;i<cantrow;i++)
                *(deriv+i)=0;
            break;
        case 1: //diferencia finita
```

```
         *(deriv) = (*(g+1)-*(g))/(2*(*(f+1)-*(f)));
         for(int i=1;i<cantrow-1;i++){
            *(deriv+i) = (*(g+i+1)-*(g+i))/(2*(*(f+i+1)-*(f+i))) + (*(g+i)-*(g+i-1))/(2*(*(f+i)
                -*(f+i-1)));
         }
         *(deriv+cantrow-1) = (*(g+cantrow-1)-*(g+cantrow-2))/(2*(*(f+cantrow-1)-*(f+cantrow-2))
             );
         break;
      case 2: // cardinal c=0.5
         *(deriv) = 0.5*(*(g+1)-*(g))/(2*(*(f+1)-*(f)));
         for(int i=1;i<cantrow-1;i++){
            *(deriv+i) = 0.5*(*(g+i+1)-*(g+i-1))/(2*(*(f+i+1)-*(f+i-1)));
         }
         *(deriv+cantrow-1) = 0.5*(*(g+cantrow-1)-*(g+cantrow-2))/(2*(*(f+cantrow-1)-*(f+cantrow
             -2)));
         break;
      case 3: // Catmull-Rom
         *(deriv) = ((*(g+1)-*(g))/(2*(*(f+1)-*(f))));
         for(int i=1;i<cantrow-1;i++){
            (*(deriv+i) = (*(g+i+1)-*(g+i-1))/(2*(*(f+i+1)-*(f+i-1))));
         }
         *(deriv+cantrow-1) =((*(g+cantrow-1)-*(g+cantrow-2))/(2*(*(f+cantrow-1)-*(f+cantrow-2))
             ));
         break;
   }
   for(int j=0;j<cantrow-1;j++){
      for(int k=*(f+j);k<*(f+j+1);k++){
         float cub = (*(frecs+k))*(*(frecs+k))*(*(frecs+k));
         float cua = (*(frecs+k))*(*(frecs+k));
         float uno = (*(frecs+k));
         float cer = 1.0;
         *(y+k)= (2*cub-3*cua+cer)*(*(g+j))+(cub-2*cua+uno)*(*(f+j+1)-*(f+j))*(*(deriv+j))+ (-2*
             cub+3*cua)*(*(g+j+1))+(cub-cua)*(*(f+j+1)-*(f+j))*(*(deriv+j+1));
      }
   }
   for(int j=0;j<22000;j++){
      if(*(y+j)<0)
         *(y+j)=0;
      else if(*(y+j)>1)
         *(y+j)=1;
   }

}
```

Listing A.8: Función para calcular la curva Spline a partir de los puntos de paso.

A.6 Statistical Level Calculator

```
void MyFrame::Calculate(float *signal, float *perc, float *lev, float *L, float *iL, int chanel)
    {

//Se ingresa la señal de audio, cuatro punteros donde guardar los valores obtenidos y
//el canal al que pertenece el audio (izquierdo, derecho o mono)

   double Tau;
   switch(tau){
      case 0: Tau = 1.0;
         break;
      case 1: Tau = 0.125;
         break;
      case 2: Tau = 0.032;
         break;
```

```
}
float b0 = 1/(1+rate*Tau);
float a1 = -(rate*Tau)/(rate*Tau+1);
slen = (timeend-timestart)*rate;
llen = slen*10/rate;
levelbuffer = new float[llen];
float *filtered = new float[slen];
int sec = slen/rate;
float x=0,y=0,y1=0;
if(pond == 1){
   double reg0[3], reg1[3], reg2[3];
   double a2[3], a1[3], a0[3], b2[3], b1[3], b0[3],g;
   a0[0] = 1;
   a0[1] = 1;
   a0[2] = 1;
   a1[0] = -0.14053608;
   a1[1] = -1.88490121;
   a1[2] = -1.99413888;
   a2[0] = 0.00493759;
   a2[1] = 0.88642147;
   a2[2] = 0.99414746;
   b0[0] = 1;
   b0[1] = 1;
   b0[2] = 1;
   b1[0] = 2;
   b1[1] = -2;
   b1[2] = -2;
   b2[0] = 1;
   b2[1] = 1;
   b2[2] = 1;
   g = 0.2557411;
   for(int j=0; j<3; j++){
      reg0[j] = 0.0;
      reg1[j] = 0.0;
      reg2[j] = 0.0;
   }

   for(int j=0;j<sec;j++){
      for(int i=0;i<rate;i++){
         int delta = j*rate;
         *(filtered+i) = *(signal+i+delta);
      }
      for(int i=0;i<rate;i++){
         for(int k=0;k<3;k++){
            x= *(filtered+i);
            reg0[k] = x - a1[k] * reg1[k] - a2[k] * reg2[k];
            y = g*(b0[k] * reg0[k] + b1[k] * reg1[k] + b2[k] * reg2[k]);
            reg2[k] = reg1[k];
            reg1[k] = reg0[k];
            *(filtered+i)=y;
         }
      }
      for(int i=0;i<rate;i++){
         int delta = j*rate;
         float b0 = 1/(1+rate*Tau);
         float a1 = -(rate*Tau)/(rate*Tau+1);
         x=*(filtered+i)**(filtered+i);
         y= b0*x-a1*y1;
         y1 = y;
         *(filtered+i)=y;
         x=0;
         if(i==0 || i%(rate/10)==0 || i==rate){
            int pos = i*10/rate;
            *(levelbuffer+pos+10*j) = 10*log(*(filtered+i));
         }
```

```
        }
    }
}
else if(pond == 2){
    double reg0[2], reg1[2], reg2[2];
    double a2[2], a1[2], a0[2], b2[2], b1[2], b0[2],g;
    a0[0] = 1;
    a0[1] = 1;
    a1[0] = -0.14053608;
    a1[1] = -1.99413888;
    a2[0] = 0.00493759;
    a2[1] = 0.99414746;
    b0[0] = 1;
    b0[1] = 1;
    b1[0] = 2;
    b1[1] = -2;
    b2[0] = 1;
    b2[1] = 1;
    g = 0.2170085;
    for(int j=0; j<2; j++){
        reg0[j] = 0.0;
        reg1[j] = 0.0;
        reg2[j] = 0.0;
    }
    for(int j=0;j<sec;j++){
        for(int i=0;i<rate;i++){
            int delta = j*rate;
            *(filtered+i) = *(signal+i+delta);
        }
        for(int i=0;i<rate;i++){
            for(int k=0;k<2;k++){
                x= *(filtered+i);
                reg0[k] = x - a1[k] * reg1[k] - a2[k] * reg2[k];
                y = g*(b0[k] * reg0[k] + b1[k] * reg1[k] + b2[k] * reg2[k]);
                reg2[k] = reg1[k];
                reg1[k] = reg0[k];
                *(filtered+i)=y;
            }
        }
        for(int i=0;i<rate;i++){
            int delta = j*rate;
            float b0 = 1/(1+rate*Tau);
            float a1 = -(rate*Tau)/(rate*Tau+1);
            x=*(filtered+i)**(filtered+i);
            y= b0*x-a1*y1;
            y1 = y;
            *(filtered+i)=y;
            x=0;
            if(i==0 || i%(rate/10)==0 || i==rate){
                int pos = i*10/rate;
                *(levelbuffer+pos+10*j) = 10*log(*(filtered+i));
            }
        }
    }
}
else if(pond == 3){
    double reg0[2], reg1[2], reg2[2];
    double a2[2], a1[2], a0[2], b2[2], b1[2], b0[2],g;
    a0[0] = 1;
    a0[1] = 1;
    a1[0] = -1.06965929;
    a1[1] = -1.99004745;
    a2[0] = 0.732480774;
    a2[1] = 0.99007225;
    b0[0] = 1.53512485;
```

```
    b0[1] = 1;
    b1[0] = -2.6916961;
    b1[1] = -2;
    b2[0] = 1.1983928;
    b2[1] = 1;
    g = 1;
    for(int j=0; j<2; j++){
       reg0[j] = 0.0;
       reg1[j] = 0.0;
       reg2[j] = 0.0;
    }
    for(int j=0;j<sec;j++){
       for(int i=0;i<rate;i++){
          int delta = j*rate;
          *(filtered+i) = *(signal+i+delta);
       }
       for(int i=0;i<rate;i++){
          for(int k=0;k<2;k++){
             x= *(filtered+i);
             reg0[k] = x - a1[k] * reg1[k] - a2[k] * reg2[k];
             y = g*(b0[k] * reg0[k] + b1[k] * reg1[k] + b2[k] * reg2[k]);
             reg2[k] = reg1[k];
             reg1[k] = reg0[k];
             *(filtered+i)=y;
          }
       }
       for(int i=0;i<rate;i++){
          int delta = j*rate;
          float b0 = 1/(1+rate*Tau);
          float a1 = -(rate*Tau)/(rate*Tau+1);
          x=*(filtered+i)**(filtered+i);
          y= b0*x-a1*y1;
          y1 = y;
          *(filtered+i)=y;
          x=0;
          if(i==0 || i%(rate/10)==0 || i==rate){
             int pos = i*10/rate;
             *(levelbuffer+pos+10*j) = 10*log(*(filtered+i));
          }
       }
    }
 }
 else if(pond == 0){
    for(int j=0;j<sec;j++){
       for(int i=0;i<rate;i++){
          int delta = j*rate;
          x=*(signal+i+delta)**(signal+i+delta);
          y= b0*x-a1*y1;
          y1 = y;
          *(filtered+i)=y;
          x=0;

          if(i==0 || i%(rate/10)==0 || i==rate){
             int pos = i*10/rate;
             *(levelbuffer+pos+10*j) = 10*log(*(filtered+i));
          }
       }
    }
 }
 levelbuffer++;
 float aux;
 for(int i=0;i<llen-2;i++){
    for(int j=0;j<llen-2;j++){
       if(*(levelbuffer+j)>*(levelbuffer+j+1)){
          aux = *(levelbuffer+j);
```

```
            *(levelbuffer+j) = *(levelbuffer+j+1);
            *(levelbuffer+j+1) = aux;
         }
      }
   }
   levelmin = *(levelbuffer+llen-2);
   levelmax = *(levelbuffer+4);

   cantpuntos = 100;
   double delta = (levelmax-levelmin)/cantpuntos;
   perc = new float[cantpuntos];
   lev = new float[cantpuntos];
   float total=0;
   for(int j=0;j<cantpuntos;j++){
      *(perc+j)=0;
      *(lev+j) = 0;
      *(lev+j) = (levelmin+delta*j)/2;
      for(int i=0;i<llen-1;i++){
         float min=(levelmin+delta*j), max=(levelmin+delta*(j+1));
         if(*(levelbuffer+i)<=min && *(levelbuffer+i)>=max){
            *(perc+j) = *(perc+j)+1;
         }
      }
      *(perc+j) = *(perc+j)/llen*100;
      total += *(perc+j);
   }
   L = new float[5];
   iL = new float[5];
   float porc=0;

   for(int i=0;i<cantpuntos;i++){
      porc +=*(perc+i);
      if(porc > 5.0){
         *(L) = *(lev+i);
         *(iL) = *(perc+i);
         break;
      }
   }
   porc=0;
   for(int i=0;i<cantpuntos;i++){
      porc +=*(perc+i);
      if(porc > 10.0){
         *(L+1) = *(lev+i);
         *(iL+1) = *(perc+i);
         break;
      }
   }
   porc=0;
   for(int i=0;i<cantpuntos;i++){
      porc +=*(perc+i);
      if(porc > 50.0){
         *(L+2) = *(lev+i);
         *(iL+2) = *(perc+i);
         break;
      }
   }
   porc=0;
   for(int i=0;i<cantpuntos;i++){
      porc +=*(perc+i);
      if(porc > 90.0){
         *(L+3) = *(lev+i);
         *(iL+3) = *(perc+i);
         break;
      }
   }
```

```
   porc=0;
   for(int i=0;i<cantpuntos;i++){
      porc +=*(perc+i);
      if(porc > 95.0){
         *(L+4) = *(lev+i);
         *(iL+4) = *(perc+i);
         break;
      }
   }
   if(chanel == 0){
      L5C->SetValue(wxString::Format(wxT("%f"), *(L)));
      L10C->SetValue(wxString::Format(wxT("%f"), *(L+1)));
      L50C->SetValue(wxString::Format(wxT("%f"), *(L+2)));
      L90C->SetValue(wxString::Format(wxT("%f"), *(L+3)));
      L95C->SetValue(wxString::Format(wxT("%f"), *(L+4)));
   }else if (chanel == 1){
      L5C2->SetValue(wxString::Format(wxT("%f"), *(L)));
      L10C2->SetValue(wxString::Format(wxT("%f"), *(L+1)));
      L50C2->SetValue(wxString::Format(wxT("%f"), *(L+2)));
      L90C2->SetValue(wxString::Format(wxT("%f"), *(L+3)));
      L95C2->SetValue(wxString::Format(wxT("%f"), *(L+4)));
   }

   Plot(perc,lev,L,iL,chanel);
   delete filtered;
}
```

Listing A.9: Código para calcular los valores de las curvas y los niveles estadísticos.

Apéndice B
Guía para crear un Plug-in

Si uno desea desarrollar un plug-in para incorporarlo a Audacity lo primero que necesita es compilar el código fuente del programa y luego recién puede comenzar a trabajar en la herramienta deseada, partiendo los dos archivos base antes comentados (a saber *module.cpp* y *module.h*).

B.1 Compilación del código fuente de Audacity

A continuación se datalla paso a paso como lograr compilar el código fuente.

1. Instalar Microsoft Visual C++ Express Edition 2008 (actualmente se está llevando a cabo una migración a la versión 2013 de este programa)[1].
2. Descargar la librería wxWidgets[2] 2.8.12 (en estos momentos se está realizando una migración a la versión 3.0.1) desde la página http://sourceforge.net/projects/wxwindows/files/2.8.12/wxMSW-2.8.12.zip/download?use_mirror=ufpr.
3. Obtener el código fuente de Audacity, utilizando la herramienta TortoiseSVN[3], desde http://audacity.googlecode.com/svn/audacity-src/trunk/
4. Compilar wxWidgets y Audacity siguiendo las instrucciones que se encuentran en el archivo http://audacity.googlecode.com/svn/audacity-src/trunk/win/compile.txt (es recomendable realizar los cuatro tipos de compilaciones de wxWidgets (Debug (d), Unicode Debug (ud) , Release () y Unicode Release (u)) para ya tener todos los juegos de archivos .dll necesarios)

Una vez realizado ésto ya podemos crear un proyecto que nos permita incorporar nuestro Plug-in a Audacity.

[1] Es un entorno de desarrollo integrado (IDE) para lenguajes de programación C, C++ y C++/CLI, gratuito que se puede descargar desde la página de Microsoft.

[2] Es una librería multiplataforma y libre, para el desarrollo de interfaces gráficas (GUI, del inglés *graphical user interface*)programadas en lenguaje C++

[3] Es un programa que permite a los desarrolladores obtener diferentes versiones de códigos fuentes de un determinado software

B.2 Creación de un proyecto para incorporar un Plug-in

Ahora debemos crear un proyecto en **Microsoft Visual C++** que contenga los dos archivos base, para ello debemos seguir los siguientes pasos.

1. Crear un proyecto del tipo dll[4].
   - $Archivo \rightarrow Nuevo \rightarrow Proyecto \rightarrow Aplicación\,de\,consola\,Win32 \rightarrow Aceptar$
2. En el asistente para aplicaciones Win32
   - $Siguiente \rightarrow Tipo\,de\,aplicación: Biblioteca\,de\,vínculos\,dinámicos\,(dll)$
     $Opciones\,adicionales: Proyecto\,vacío$
3. Haciendo click derecho sobre el proyecto seleccionamos la opción *Propiedades* y aparecerá una ventana donde debemos completar la configuración del proyecto.
   - $C/C++ \rightarrow General \rightarrow Directorios\,de\,inclusión\,adicionales$
     Aquí se debe añadir los directorios donde buscar los archivos del tipo .h y .cpp que se incluyan, especialmente la dirección de la carpeta include de wxWidgets y la carpeta del código fuente de Audacity.
   - $C/C++ \rightarrow Preprocesador \rightarrow Definiciones\,de\,proprocesador$
     Acá se deben agregar las definiciones de wxWidgets que varían según el proyecto (en nuestro caso son WIN32;__WXMSW__;_WINDOWS;_DEBUG;__WXDEBUG__)
   - $Vinculador \rightarrow General \rightarrow Directorios\,de\,bibliotecas\,adicionales$
     En esta opción se deben insertar las direcciones de las librerías (.lib) que se usarán en el proyecto, no se debe olvidar incluir la dirección de las librerías de Audacity y wxWidget:
     <pathto>[5]/audacity-src-2.x.y/win/Unicode Debug
     <pathto>/wxWidgets-2.8.12/lib/vc_dll respectivamente
   - $Vinculador \rightarrow Entrada \rightarrow Dependencias\,adicionales$
     Debemos ingresar: Audacity.lib wxbase28ud.lib wxbase28ud_net.lib wxmsw28ud_adv.lib wxmsw28ud_core.lib wxmsw28ud_html.lib (es necesario recordar que la ud viene de Unicode Debug, es decir, esta configuración corresponde a la compilación en modo DEBUG, sino deberá utilizarse solo la u)

Para poder ir probando nuestro Plug-in a medida que avanzamos o, como se le dice, *Depurar (del inglés Debug)*[6] nuestro programa, debemos completar los siguientes campos de configuración.

[4] Dynamic Link Library, o en castellano biblioteca de enlaces dinámicos

[5] Se denomina de esa manera al directorio donde se encuentra la carpeta que sigue a <pathto>/, ya que depende de donde cada persona la instale (en nuestro caso es la carpeta donde descargamos el código fuente de Audacity)

[6] Es la acción probar un programa y eliminar errores que contenga.

- $Depuración \rightarrow Comando$

  Agregar $< pathto > \backslash Audacity \backslash win \backslash UnicodeDebug \backslash Audacity.exe$

- $Eventos\ de\ compilación \rightarrow Evento\ posterior\ a\ la\ compilación \rightarrow Línea\ de\ comandos$

  Añadir *copy*"$(TargetPath)"" < pathto > \audacity - src - 2.x.y \win \ UnicodeDebug \ Modules$"

Una vez configurado esto cada vez que utilicemos la opción *Debug* del programa Microsoft Visual Studio C++ se iniciará el programa Audacity, el cual cargará los Plug-ins agregados.

Finalmente una vez que hayamos desarrollado las herramientas que deseamos es posible crear un instalador que los contenga, para ello debemos utilizar el programa *Inno Setup*[31] (utilizado por los desarrolladores de Audacity para crear sus propios instaladores) y el archivo *audacity.iss* que se encuentra en la carpeta win del código fuente de Audacity. Luego de abrir el archivo sólo resta compilarlo y nos generará el instalador deseado.

Apéndice C

Listado de archivos desarrollados y modificados

Se detallará en esta sección cuáles archivos del código fuente fueron modificados y cuáles se crearon para obtener lo logrado a lo largo de todo el proyecto. Para ello en la Figura C.1 se muestran los proyectos correspondientes a los *plug-ins* creados.

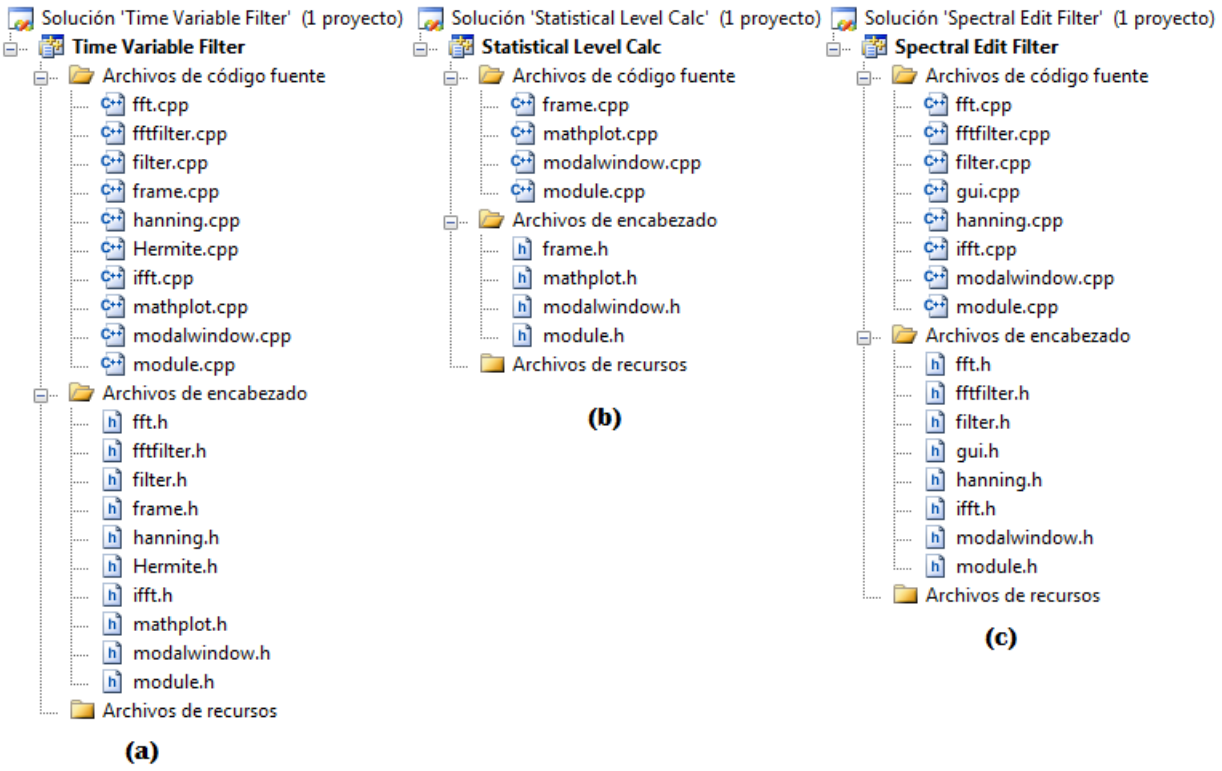


Figura C.1: Proyectos de Visual C++ de los plug-ins creados. (a) Time Variable Filter (b) Statistical Level Calc (c) Spectral Edit Filter

C.1 Visor de espectrograma

Los archivos pertenecientes al código fuente de Audacity que se modificaron para lograr los distintos mapas de colores se detallan a continuación

Acolor	En él se definen los mapas de colores utilizados en el espectrograma y los colores que se utilicen en todo el software.
SpectrumPrefs	Describe y crea el menú de preferencias de los espectrogramas.
TrackArtist	Maneja los ploteos o dibujos que se realizan en el panel de pistas o Track Panel.

C.2 Analizador de espectro

Para lograr las mejoras que se realizaron en el analizador de espectro se debieron modificar los siguientes archivos

FreqWindow	Maneja el funcionamiento y la interfaz gráfica del analizador de espectro original del software.
TrackPanel	Define los puntos de inicio y fin de la selección que se utiliza para calcular el espectro.

C.3 Visor de nivel

Los archivos modificados para poder implementar el visor de nivel y añadir la posibilidad de ingresar un tono de calibración o un valor de fondo de escala fueron

TrackMenue	Define el menú de pista donde se puede seleccionar el tipo de representación de la señal que se desea ver.
TrackArtist	Se encarga de dibujar el TrackPanel y todo lo que interviene en la gráfica de la señal.
WaveClip	Se encarga de guardar los valores de la señal y en este caso el valor del nivel de ella.

C.4 Importador y exportador de metadatos

Los archivos agregados a la carpeta src correspondiente al código fuente de Audacity son los siguientes

LabelReader	Utilizado para definir las variables necesarias a la hora de leer los metadatos donde se encuentra el código que permite leer los metadatos.
LabelWriter	Utilizado para definir las variables necesarias a la hora de grabar los metadatos y el código que permite llevar a cabo la escritura de los metadatos.

Además se modificaron los archivos

Menus	Se encarga de definir todos los menús del software y dentro de ellos se encuentra la rutina de importación y exportación de los archivos de audio.
ExportPCM	Maneja la exportación de los archivos .wav.

C.5 Spectral Edit Filter

La composición del proyecto de la herramienta de filtrado de edición espectral se ve en la Figura C.1 (c) y la siguiente lista define la utilidad de cada archivo (se explica el funcionamiento en conjunto de cada par *.h* y *.cpp*)

fft	Permite calcular la transformada rápida de Fourier de una cantidad determinada de datos.
ifft	Se usa para obtener la transformada rápida inversa de Fourier de la señal
fftfilter	A partir de los datos de la señal y un filtro, obtiene la señal filtrada. Desarrollado por Gonzalo Sad (basado en un script de MATLAB creado por Federico Miyara, originado en una idea de David Giardini) y modificado por mi para que el filtro evolucionara en el tiempo.
filter	Utiliza el archivo fftfilter y los datos ingresados por el usuario para crear los filtros y aplicarlos.
gui	Se encarga del manejo de la interfaz gráfica.
hanning	Obtiene la versión multiplicada por una ventana de Hanning de los datos originales.
modalwindow	Transforma la interfaz gráfica en una ventana que acapara los eventos del mouse (clicks, desplazamientos, etc.) para poder llevar a cabo el diseño del filtro sin perder información.
module	Integra el Plug-in a Audacity y se encarga del manejo de datos entre las funciones y archivos antes detallados, ya que los incluye a todos y puede utilizar las funciones que necesite de cada uno de ellos (por ejemplo llamar a la interfaz gráfica).

C.6 Time Variable Filter

El proyecto y sus archivos se pueden ver en la Figura C.1 (a), siendo éstos ultimos los detallados a continuación

fft	Permite calcular la transformada rápida de Fourier de una cantidad determinada de datos.
ifft	Se usa para obtener la transformada rápida inversa de Fourier de la señal
fftfilter	A partir de los datos de la señal y un filtro, obtiene la señal filtrada. Desarrollado por Gonzalo Sad (basado en un script de MATLAB creado por Federico Miyara originado en una idea de David Giardini) y modificado por mi para que el filtro evolucionara en el tiempo.
filter	Utiliza el archivo fftfilter y los datos ingresados por el usuario para crear los filtros y aplicarlos.
frame	Se encarga del manejo de la interfaz gráfica.
hanning	Obtiene la versión multiplicada por una ventana de Hanning de los datos originales.
modalwindow	Transforma la interfaz gráfica en una ventana que acapara los eventos del mouse (clicks, desplazamientos, etc.) para poder llevar a cabo el diseño del filtro sin perder información.
mathplot	Es una librería que sirve para poder dibujar gráficas, se utilizó para poder dibujar la interpolación obtenida con ayuda del archivo Hermite, descargada desde su página oficial [10].
Hermite	Regresa, a partir de la tabla completada por el usuario, el valor de cada punto de la interpolación del tipo Hermite.
module	Integra el Plug-in a Audacity y se encarga del manejo de datos entre las funciones y archivos antes detallados, ya que los incluye a todos y puede utilizar las funciones que necesite de cada uno de ellos (por ejemplo llamar a la interfaz gráfica).

C.7 Statistical Level Calc

Los archivos que componen el proyecto de la Figura C.1 (b) son

frame	Se encarga del manejo de la interfaz gráfica.
module	Integra el Plug-in a Audacity y se encarga del manejo de datos entre las funciones y archivos, ya que los incluye a todos y puede utilizar las funciones que necesite de cada uno de ellos (por ejemplo llamar a la interfaz gráfica).
modalwindow	Transforma la interfaz gráfica en una ventana que acapara los eventos del mouse (clicks, desplazamientos, etc.).
mathplot	Es una librería que sirve para poder dibujar gráficas, descargada desde su página oficial [10], que se utilizó para poder dibujar las curvas de niveles estadísticos y de densidad de probabilidad.

Apéndice D

Detalle del contenido del CDs adjuntados

Junto a la versión impresa del informe se adjunta dos discos cuyos contenidos son los siguiente:

CD 1:

- Copia digital del informe.

CD 2:

- Instalador de la última versión de Audacity compilada con todas las modificaciones realizadas y los Plug-ins desarrollados funcionando para Windows 7.
- Copia del código fuente de audacity con las modificaciones realizadas.
- Copia del proyecto (LAE Tools) de Microsoft Visual C++ 2008 que permite obtener los Plug-ins creados luego de compilarlo.