

Organización del Computador II

Departamento de Computación
Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires

Trabajo Práctico Final: PedalerIA64 *Informe*

Integrante	LU	Correo electrónico
Laporte Matías	686/09	matiaslaporte@gmail.com

1. Introducción

El presente Trabajo Práctico consiste en el desarrollo de diversos efectos de audio que se pueden encontrar en softwares de edición en el contexto musical (Cubase, Reaktor, Audacity, etc.), así como también en pedalerías (de allí el nombre del programa) para el caso específico de una guitarra.

1.1. Proceso de desarrollo del TP

El proceso de desarrollo del **TP** consistió en realizar en primera instancia un rápido prototipado en Matlab/RStudio/Scilab (dependiendo de la disponibilidad de la computadora que se usara en ese momento; de ahora en más, me referiré a esos programas como **MRS**) de algoritmos y ecuaciones que se pudieran encontrar en Internet de diversos efectos. El uso de esos programas como primer acercamiento a un efecto facilitaba enormemente el trabajo, pues al utilizar un lenguaje de muy alto nivel se simplificaba el manejo de la lectura y escritura de los archivos, el espacio en memoria para los mismos, la manipulación de los datos (poder trabajar trabajar sobre un vector entero con una sola operación, por ejemplo), etc.

Una vez certificado que con ese algoritmo se consiguiera el efecto auditivo deseado, se pasó a desarrollarlo en **C**, siempre verificando que el resultado final de la señal coincidiera con el obtenido en **MRS**. Como utilizar el comando *diff* de UNIX directamente sobre los archivos no siempre daba el resultado querido (aventuro a decir que por diferencias de aproximación entre **MRS** y **C**), se optó por otra metodología utilizando el programa **Audacity**, que se explicará en la sección 1.4.4.

Obtenido el resultado deseado utilizando **C**, se pasó finalmente a programar el mismo algoritmo en **Assembler**, haciendo uso de las instrucciones que ofrece el conjunto de instrucciones SSE para manejar múltiples datos con una única instrucción (SIMD). Se utilizaron instrucciones incluidas hasta la extensión SSE4 (principalmente por *PTEST*, y *ROUNDPS/ROUNDSS*). Luego de obtener en **Assembler** un código que parecía aceptable (es decir, que no terminara abruptamente con un segfault), se comenzó con un proceso iterativo de corrección, mediante la comparación del archivo de audio obtenido con el de **C**, utilizando **Audacity** como se mencionó previamente. Las diferencias entre ambos muchas veces provenían de casos bordes, que se los mencionará en la sección ??.

El objeto de la programación en **C** además de Assembler, si bien implica el "doble" de trabajo, se debe a dos puntos en particular. En primer medida, era una buena manera de poder bajar el nivel del código original en **MRS**, despojándolo de las bondades que dichas herramientas ofrecen. Por otro lado, tener el código en **C** sirve también para comparar la mejora de rendimiento con el uso de las instrucciones SIMD.

Para calcular el rendimiento en ambos lenguajes se utilizó la librería *Tiempo.h* utilizada por la cátedra en el **TP N°2** del 1er. Cuatrimestre de 2011. Cuando los resultados con la misma no eran satisfactorios (p.ej., **Assembler** era más lento que **C**), se procedió a utilizar una herramienta de profiling (**callgrind** en conjunto con **KCacheGrind**, se hablará de ellas más adelante) para poder identificar dónde exactamente estaban las secciones lentas del código en **Assembler**, y poder tomar medidas para resolverlo. De los casos puntuales donde se utilizó esto se hablará en las secciones ??, ?? y ??.

Se desarrolló también una rudimentaria interfaz gráfica para tratar de evitar la utilización de la línea de comandos (en particular por la cantidad de argumentos que hay que manipular para los diversos efectos) y que la utilización del programa sea más amigable. Se hablará de ella en la sección 1.4.3.

1.2. Audio

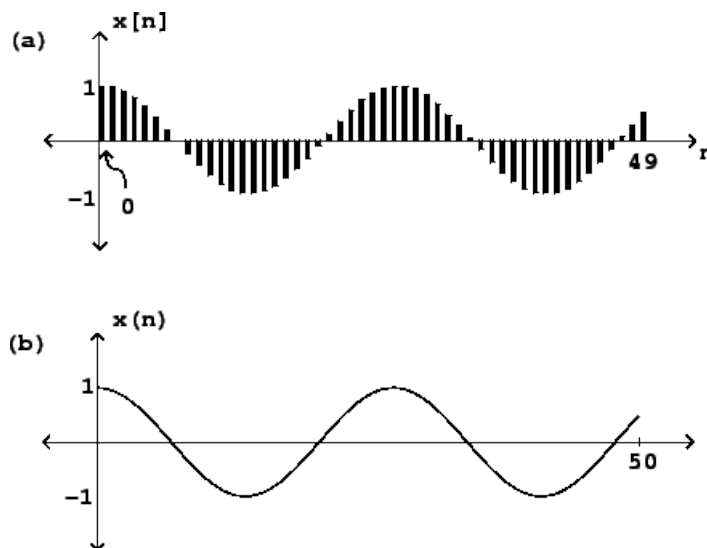


Figura 1: Señales de audio

Simplificándolo extremadamente, una señal de audio ((b) en la figura 1) es una representación del sonido, que puede ser visualizado como una curva continua (en el caso analógico, sus valores representan voltaje eléctrico) en función del tiempo. Al digitalizar una señal, se discretiza la curva ((a) en la figura 1) tomando valores cada cierta cantidad de tiempo, lo que da lugar a la *frecuencia de muestreo* (**sampling rate**, expresada como cantidad de muestras por segundo, unidad **Hz**). A la vez, cada uno de esos valores no puede ser expresado con precisión infinita, sino que al pasar al dominio digital, debe poder ser representado con una cantidad específica de bits, lo que origina la *tasa de bits* (**bit rate**, *resolución* de una señal de audio).

El formato de audio elegido para el **TP** es WAV ¹, por ser uno de los más simples para manipular. Los datos correspondientes al audio no están comprimidos, por lo que es posible realizar directamente sobre ellos las operaciones necesarias para aplicar los diversos efectos. En caso de que un archivo sea **stereo**, la información va intercalada (una muestra del canal izquierdo, otra del derecho, la siguiente del izquierdo, etc.).

La librería utilizada para el manejo de este archivo se verá en la sección 1.3.1.

Nota: por recomendación del profesor durante la presentación del proyecto de **TP**, en el archivo de audio final obtenido luego de la aplicación de alguno de los efectos, la *señal seca* (*dry sound/dry signal*, sin efecto) va por un canal, y la *señal húmeda* (*wet sound/wet signal*) por el otro. De este modo, se puede apreciar con mayor claridad el efecto en cuestión.

Esto implica que todos los archivos de salida tendrán dos canales (**stereo**), a pesar de que el archivo de entrada pudiera haber tenido un único canal. En el caso de archivos de entrada con dos canales, se realiza un promedio de ambos (aunque esto sólo es válido en archivos stereo que mandan el mismo audio por los canales), y sobre ese nuevo “canal” se aplica el efecto correspondiente.

¹Más información: <http://stackoverflow.com/questions/13039846/what-do-the-bytes-in-a-wav-file-represent>

1.3. Herramientas externas utilizadas

Además de los ya mencionados **Matlab**, **RStudio**, y **Scilab**, se utilizaron las siguientes herramientas desarrolladas por terceros.

1.3.1. libsndfile

libsndfile es una librería de código abierto desarrollada en **C** para leer y escribir archivos de audio. Trabaja con el formato WAV, entre otros, por lo que se adaptaba a las necesidades del **TP**. La API puede consultarse aquí ².

Una ventaja de la librería es que hace un pasaje de integer (tipo de datos utilizado en el formato WAV) a float, que es el tipo de datos utilizado para el **TP** ³. Por otro lado, al leer un archivo utiliza una estructura propia llamada SF_INFO, que incluye datos importantes del mismo (cantidad de canales, de muestras, entre otros), y que son necesarios en algunas porciones del **TP** por diversos motivos.

1.3.2. Audacity

Audacity es un editor multiplataforma de audio digital de código abierto y que en el **TP** se utilizó para realizar las comparaciones entre los archivos finales obtenidos de los diferentes efectos para cada lenguaje. De este modo, al no encontrar diferencias entre las señales de dos archivos diferentes, se podía corroborar que el algoritmo coincide con el efecto a aplicar.

1.3.3. PyQt

Para desarrollar la interfaz gráfica del TP, se utilizó PyQt5 (*versión 5.2.1*), que provee bindings de Python (*versión 3.x*) para el framework QT (*versión 5.2.1*). Se verán los paquetes que será necesario instalar para usar la GUI en la sección 1.4.1

1.3.4. Valgrind, KDbg, Callgrind, KCacheGrind

Todas las herramientas mencionadas en el título fueron utilizadas para el debug del **TP**.

Valgrind

Valgrind fue utilizado para saber cómo era el manejo de memoria del programa. En el caso de que el programa terminara repentinamente debido a algún *segmentation fault*, mediante **Valgrind** se podía saber en qué línea del código ocurría, pasando entonces a ver cuál fue el acceso erróneo viendo los valores de los registros con **Kdbg**.

KDbg

Si bien en la materia se recomendaba utilizar DDD, a mi parecer tenía una interfaz bastante anticuada, y *crasheaba* mucho; por esa razón se buscó una alternativa, y **Kdbg** resultó ser una opción más que adecuada para mis requerimientos, más estable y más amigable en cuanto a UI.

²<http://www.mega-nerd.com/libsndfile/api.html>

³Originalmente se pensaba utilizar double, pero por recomendación del profesor se decidió pasar a float para poder procesar más datos

Callgrind

Cuando había dudas con respecto a la performance del código en **Assembler** al compararlo con **C**, se buscó información sobre herramientas para profiling. Es posible utilizar **Valgrind** con una serie de argumentos especiales (*-tool=callgrind -dump-instr=yes -collect-jumps=yes*) que devuelven un archivo de nombre `callgrind.out.XXXXXX` (siendo `XXXXXX` el número del proceso corrido) donde se encuentra toda la información sobre los llamados a instrucciones y dónde se pierde más tiempo en la ejecución de un programa.

KCacheGrind

Para visualizar el archivo anterior, se utiliza KCacheGrind, que muestra toda la información de manera completamente intuitiva, y que permite identificar rápidamente dónde están los cuellos de botella del programa. Se verán los resultados obtenidos con el programa en la sección ??.

1.4. Uso TP

1.4.1. Paquetes a instalar

Para esta sección, se instaló la distribución Linux Mint 17.1 (basada en Ubuntu) en una máquina virtual, de modo de poder saber qué es necesario instalar en un sistema desde 0 para poder correr el TP.

Compilar TP

Para poder compilar el **TP** mediante el comando *Make*, se necesitan los paquetes:

- **libsndfile1-dev** (librería libsndfile)
- **build-essential** (librería stdio.h)
- **nasm**

Interfaz gráfica

Para poder ejecutar la interfaz gráfica:

- **python3**
- **python3-pyqt5** (bindings de Qt para python3)
- **python3-pyqt5.multimedia** (para poder reproducir archivos de audio desde la GUI)

Debug

Para las herramientas de debug, es necesario instalar:

- **valgrind**
- **kdbg**
- **kcachegrind**
- **graphviz libgraphviz-perl** (sólo para poder ver el Call Graph en KCacheGrind)

Comparación visual señales de audio

Si se desea realizar la comparación visual de las señales de audio explicada en la sección 1.4.4, será necesario instalar el siguiente paquete:

- **audacity**

1.4.2. Línea comandos

Una vez compilado el **TP** (mediante el comando *make*, pues el archivo Makefile se encuentra incluido), se puede ver la ayuda del programa ejecutando únicamente *./main*. Por razones de completitud, se explica aquí también cómo utilizar el programa.

La estructura para aplicar un efecto a un archivo de audio es la siguiente:

```
./main INFILE OUTFILE EFFECT ARGS
```

- **INFILE** es el archivo de audio de entrada, siempre en formato WAV.
- **OUTFILE** es el nombre deseado del archivo de salida, con extensión .WAV.
- **EFFECT** es un guión, seguido del caracter asociado al efecto a aplicar.
- **ARGS** son los argumentos dependientes del efecto definido en **EFFECT**.

La lista de los efectos y los rangos de los argumentos correspondientes (los mismos se explicarán en la sección ?? de Desarrollo de cada efecto) se pueden consultar en la siguiente tabla:

Nombre	Caracter		Argumentos			
	ASM	C				
Copiar	C	c	Ninguno			
Delay	D	d	<u>Delay</u> : 0.0-5.0 segundos		<u>Decay</u> : 0.00-1.00	
Flanger	F	f	<u>Delay</u> : 0-15 milisegundos	<u>Rate</u> : 0.1-5.0 hertz	<u>Amp</u> : 0.65-0.75	
Vibrato	V	v	<u>Depth</u> : 0-3 milisegundos		<u>Mod</u> : 0.1-5.0 hertz	
Bitcrusher	B	b	<u>Bits</u> : 1-16		<u>Bitrate</u> : 1-44100 hertz	
Wah Wah	W	w	<u>Damp</u> : 0.1-1.0	<u>MinFreq</u> : 400-1000 hertz	<u>MaxFreq</u> : 2500-3500 hertz	<u>WahWah Freq</u> : 1000-3000 hertz

Cuadro 1: Lista de comandos

1.4.3. GUI

La GUI intenta ser una manera más intuitiva de ejecutar el programa, sin necesidad de tener que ingresar todos los argumentos a mano. Se ejecuta (estando en la carpeta **src**) mediante el siguiente comando:

```
python3 gui/main.py
```

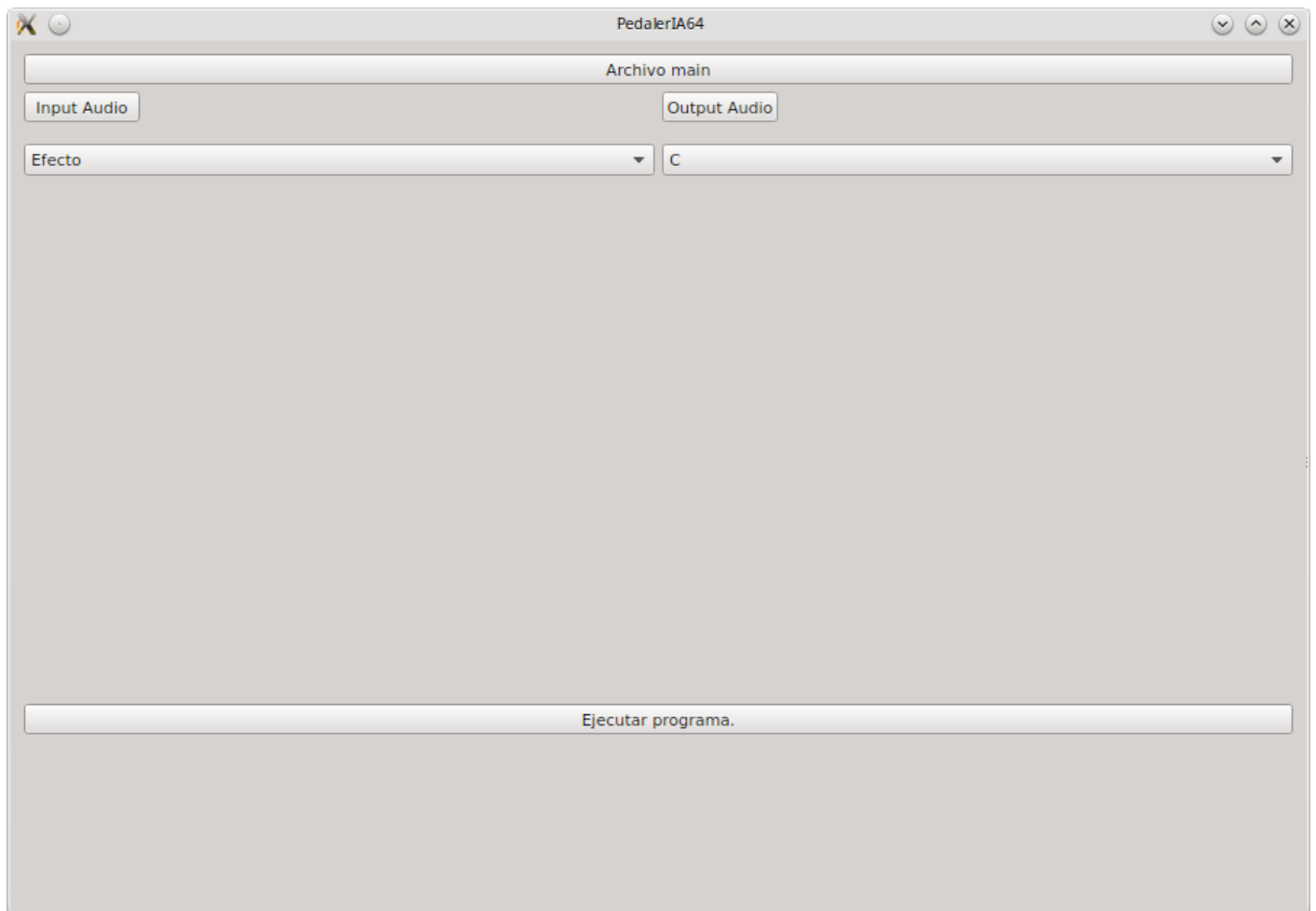


Figura 2: GUI

La GUI limpia se ve en la figura 2. Es necesario seleccionar dónde se encuentra el archivo **main**, el archivo de entrada sobre el cual se quiere aplicar el efecto, cuál es el nombre deseado del archivo de salida (se lo colocará en la misma carpeta donde se encuentra **main**) y, finalmente, el efecto a aplicar junto con sus argumentos.

La GUI con todos los argumentos completados, y luego de seleccionar el botón “ejecutar programa” se ve como en la figura 3.

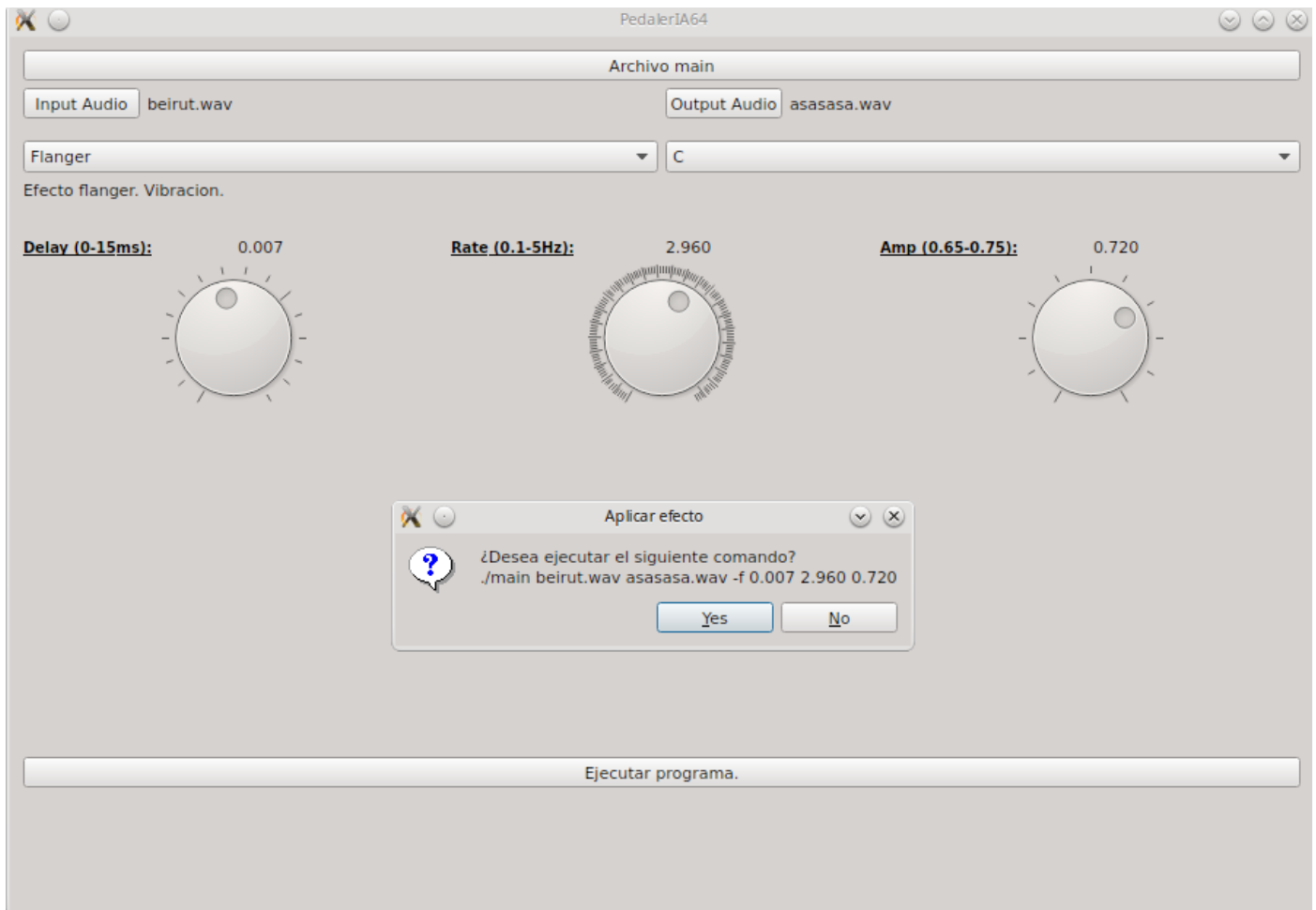


Figura 3: GUI con todas las opciones seleccionadas

Nota: En el popup para confirmar si el comando es el correcto, el mismo no representa exactamente lo que se ejecuta, pues faltan los paths hacia cada archivo. Para que no quede un texto largo e incomprensible en el popup, se decidió poner únicamente los nombres de los MAIN, INFILE y OUTFILE, aún cuando los primeros dos podrían no compartir carpeta (OUTFILE siempre está en la misma carpeta que MAIN).

Al poner “Yes”, la interfaz ejecutará el comando y, en caso de que todo haya salido correctamente, ofrecerá dos botones para poder reproducir el audio de entrada, el de salida, y comparar, como se puede ver en la figura 4.

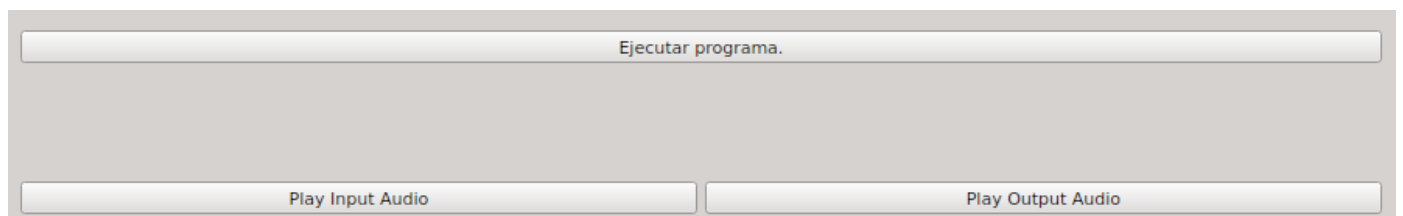


Figura 4: Botones de reproducción

En caso de que falle, se dará noticia de eso, pero no se manejará ni mostrará el error (se recomienda ejecutar el mismo comando que hubiera ejecutado la GUI en la CLI para ver qué pasó; de todos modos, es posible ver el error en la terminal desde donde se haya corrido Python).

En caso de que falte completar alguna opción, al hacer click en “Ejecutar programa” aparecerá un popup informando de cuál es la opción que falta llenar.

1.4.4. Chequeo de diferencias con Audacity

Para poder verificar si hay diferencias entre dos archivos de audio en Audacity, es necesario proceder del siguiente modo. Con el programa abierto, se arrastran los dos archivos hacia la ventana del mismo para que sean importados automáticamente (si es la primera vez, se va a preguntar si se quiere trabajar sobre los mismos archivos, o sobre una copia temporal de los mismos; por seguridad, se recomienda elegir esta última opción, y que el programa la recuerde).

Como ambos archivos son “de salida” (para nuestro programa), serán los dos stereo. Es necesario separar los canales de cada uno de los archivos, para compararlos entre sí. Hacer click sobre la flecha al lado del nombre del archivo, y clickear en “Split Stereo to Mono” (o presionar la tecla **n**); ver figura 5.

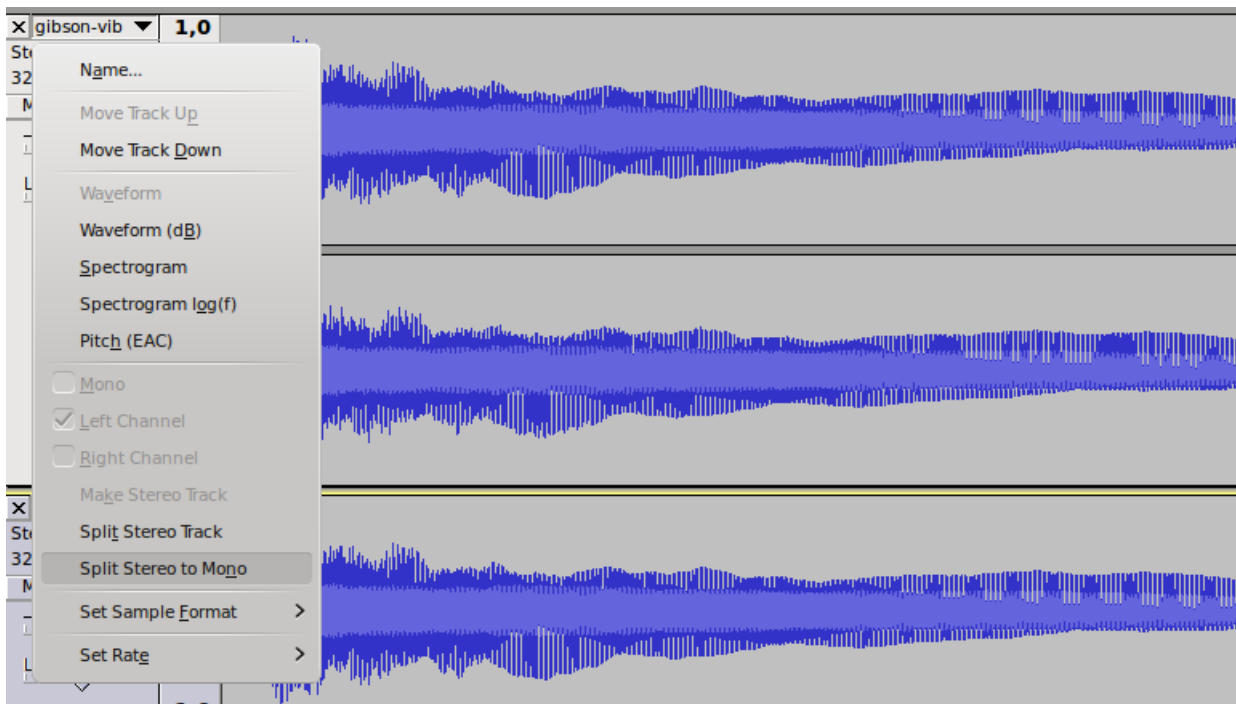


Figura 5: Convertir Stereo a canales Mono

Separados ya los canales de ambos archivos, seleccionar el canal izquierdo del primer archivo (aparecerá con un color diferente al resto), e ir al menú “Effect” (**Alt+c**), y elegir “Invert”; ver figura 6.

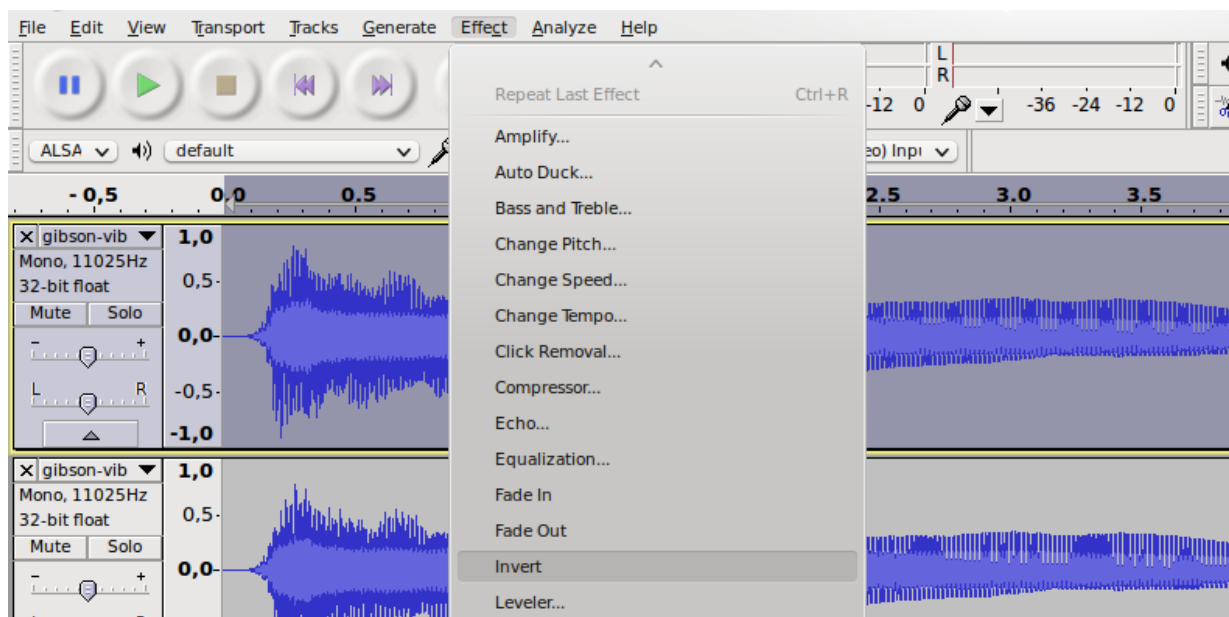


Figura 6: Invertir canal

Invertido, por ejemplo, el canal izquierdo del primer archivo, se selecciona dicho canal, junto con el izquierdo del segundo archivo (**shift+click** en los cuadrantes grises a la izquierda de cada señal). Con los dos canales correspondientes seleccionados, uno de ellos invertido, se va al menú “Tracks” (**Alt+t**), y se selecciona la opción “Mix and Render” (tecla **x**); ver figura 7.

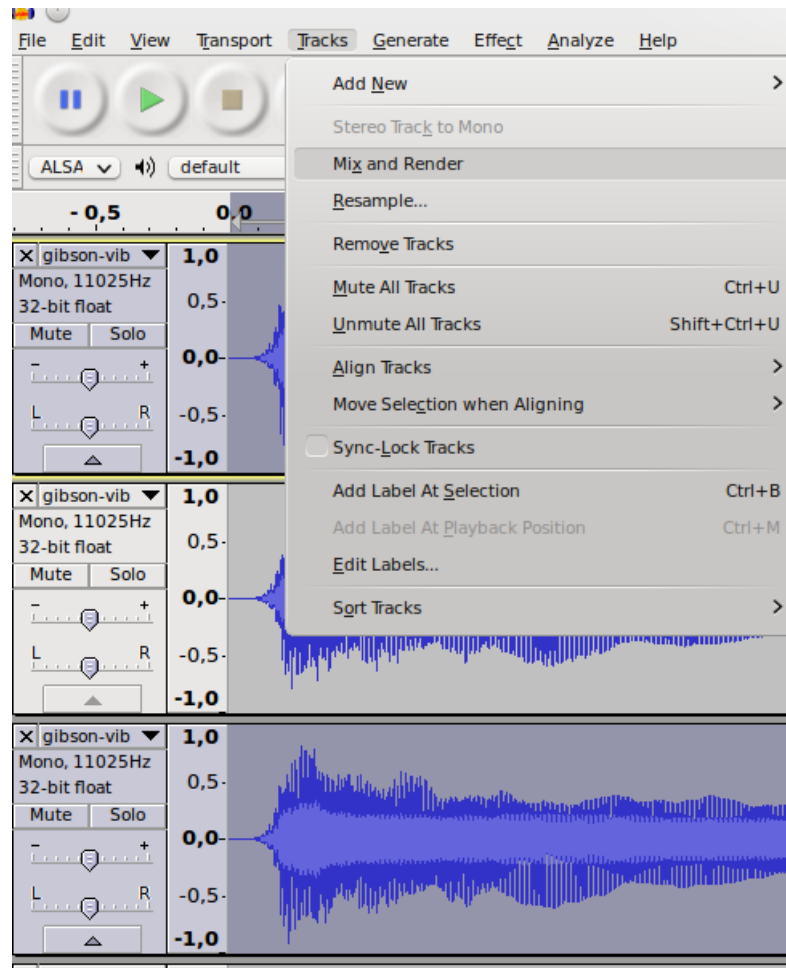


Figura 7: Mix and render

Los dos canales seleccionados que fueron mezclados, originalmente tenían la misma información. Al invertir uno de ellos, y mezclarlos entre sí, se produce una cancelación de la onda. Por lo tanto, si efectivamente contenían la misma información, debería verse el nuevo canal generado completamente vacío; ver figura 8. Durante el desarrollo del **TP**, esto a veces se consiguió y otras no, y las razones de ello se verán en las secciones correspondientes.

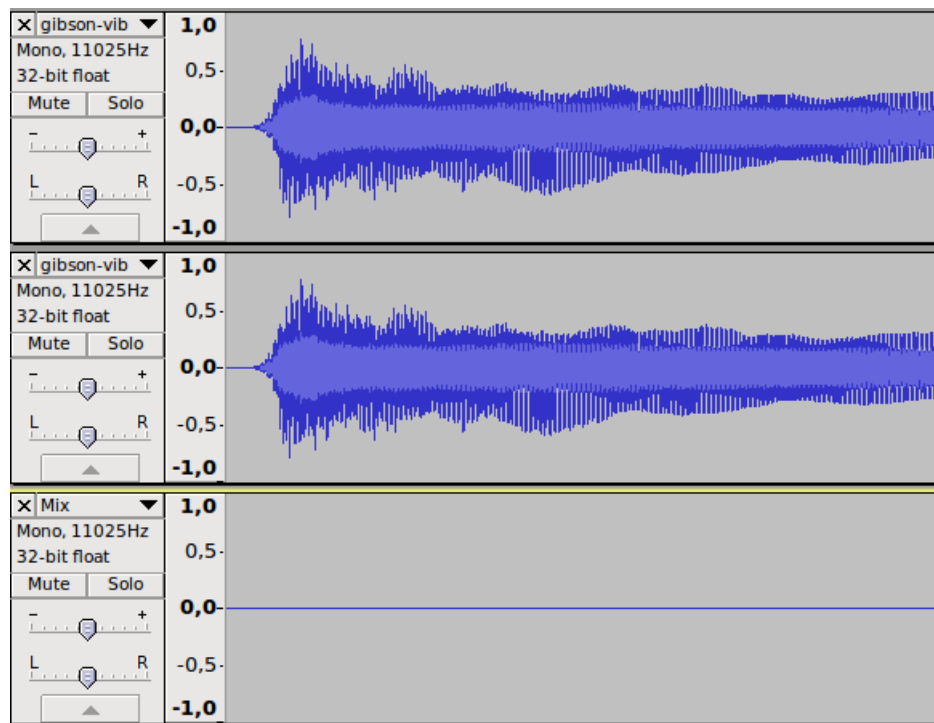


Figura 8: Cancelación de la onda

El canal izquierdo es la señal seca; repetir luego el mismo procedimiento con los dos canales derechos restantes.

2. Desarrollo

3. Fuentes

- Figura 1: <http://msp.ucsd.edu/techniques/v0.11/book-html/node7.html>

4. Bibliografía tentativa

4.1. Libros

Referencias

- [1] Richard Boulanger, Victor Lazzarini *The Audio Programming Book*, 2011, The MIT Press, Massachusetts (USA)
- [2] F. Richard Moore, *Elements of Computer Music*, 1990, Prentice Hall, New Jersey (USA)
- [3] Sophocles J. Orfanidis, *Introduction to Signal Processing*,
<http://www.ece.rutgers.edu/~orfanidi/intro2sp/>
- [4] Curtis Roads, *The Computer Music Tutorial*, 1996, The MIT Press, Massachusetts (USA)
- [5] Davide Rocchesso, *Introduction to Sound Processing*, profs.sci.univr.it/~rocchess/SP/sp.pdf
- [6] Steven W. Smith, *The Scientist and Engineer's Guide to Digital Signal Processing*, Second Edition, 1999, California Technical Publishing, California (USA)
- [7] Udo Zölzer, *DAFX: Digital Audio Effects* Second Edition, 2011, Wiley and Sons, Hamburg (Germany)

4.2. Internet

- <http://www.mega-nerd.com/libsndfile/api.html>
- <http://stackoverflow.com>
- <https://ccrma.stanford.edu/~jos/>
- <http://www.musicdsp.org/>
- <http://www.kvraudio.org/>
- http://en.wikipedia.org/wiki/Sampling_rate
- <http://stackoverflow.com/questions/13039846/what-do-the-bytes-in-a-wav-file-represent>