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 pedaleras (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 Chequeo de diferencias con Audacity (Subsubsección 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 Problemas en el Desarrollo (Subsección 2.10).

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 Desarrollo (Sección 2), Resultados (Sección 3) y Analisis (Sección 4).

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 GUI (Subsubsección 1.4.3).

1.2. Audio

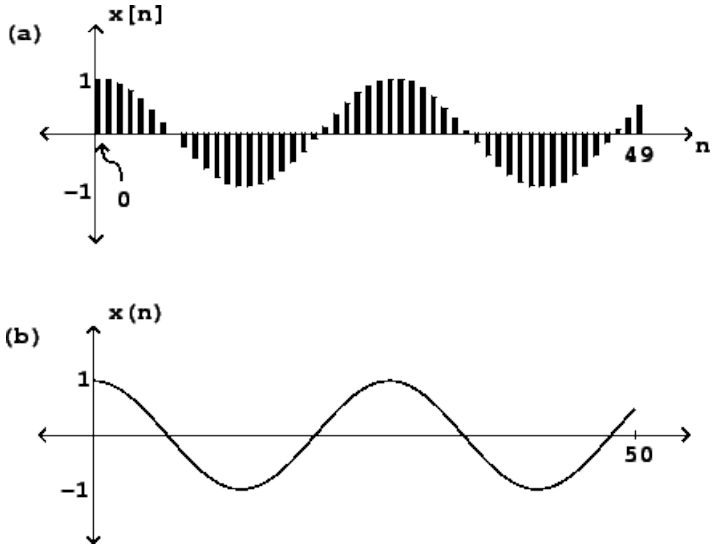

Figura 1: Señales de audio

Simplificándolo extremamente, una señal de audio (**(b)** en la figura 9) 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 9) 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 tipo de archivo se verá en la sección libsndfile (Subsubsección 1.3.1).

[1] Más información aquí:
`http://stackoverflow.com/questions/13039846/what-do-the-bytes-in-a-wav-file-represent`

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.

1.3. Herramientas externas utilizadas

Además de los ya mencionados **Matlab**, **RStudio**, y **Scilab**, se utilizaron las siguientas 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.

API

TODO -¿AGREGAR FUNCIONES USADAS ??

1.3.2. SSE Math

ssemath es una librería hecha en C que provee las funciones trascendentales básicas (seno, coseno, exponenciales) implementadas con instrucciones SIMD. Fue desarrollada para poder suplantar la **Intel Approximate Library**[4], que entre otros detalles era -como su nombre aclara-, aproximada.

Varios efectos necesitan realizar operaciones con seno, por lo que fue necesario buscar y utilizar una librería externa para poder realizar operaciones con seno vectorizables, ya que el set de instrucciones de Intel no ofrece ninguna que cumpla ese cometido. Como se verá en las secciones correspondientes de Resultados (Sección 3) y Analisis (Sección 4), gracias al uso del profiler se descubrió que al utilizar esta librería se generaba una pérdida de performance que ocasionaba que los efectos en **Assembler** sean por lo menos iguales en rendimiento (o en algunos casos considerablemente más lento) que **C**. Finalmente, se recurrió a una adaptación de la solución provista aquí.

1.3.3. 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 en sus dos implementaciones coincide con el efecto a aplicar.

1.3.4. 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

[2] `http://www.mega-nerd.com/libsndfile/api.html`

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

[4] No disponible un link oficial. `http://forum.devmaster.net/t/approximate-math-library/11679/7`

1.3.5. 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.

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.XXXXX (siendo XXXXX 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 3.

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. Linea 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 Desarrollo (Sección 2) de Desarrollo de cada efecto) se pueden consultar en la siguiente tabla:

Nombre	**Caracter**		**Argumentos**			
	ASM	**C**				
Copiar	C	c	Ninguno			
Delay Simple	D	d	Delay: 0.0-5.0 segundos	Decay: 0.00-1.00		
Flanger	F	f	Delay: 0-15 milisegundos	Rate: 0.10-1.00 hertz	Amp: 0.65-0.75	
Vibrato	V	v	Depth: 0-3 milisegundos	Mod: 0.1-5.0 hertz		
Bitcrusher	B	b	Bits: 1-16	Bitrate: 1-44100 hertz		
Wah Wah	W	w	Damp: 0.1-1.0	MinFreq: 400-1000 hertz	MaxFreq: 2500-3500 hertz	WahWah Freq: 1000-3000 hertz

Cuadro 1: Lista de comandos

Además de los efectos definidos en la tabla anterior, se desarrollaron algunas funciones auxiliares que serán oportunamente descriptas en Funciones auxiliares (Subsección 2.2).

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

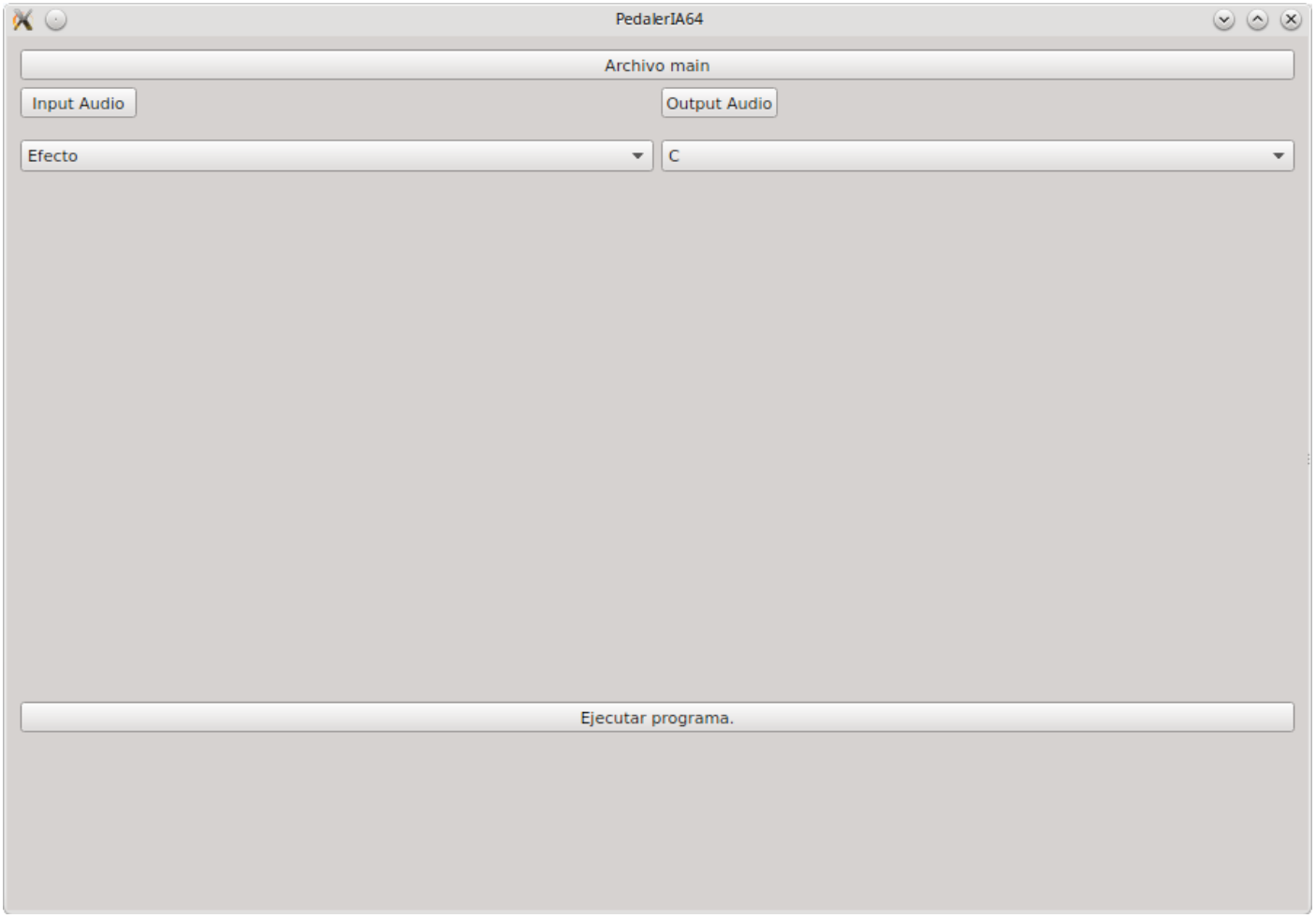

Figura 2: GUI

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.

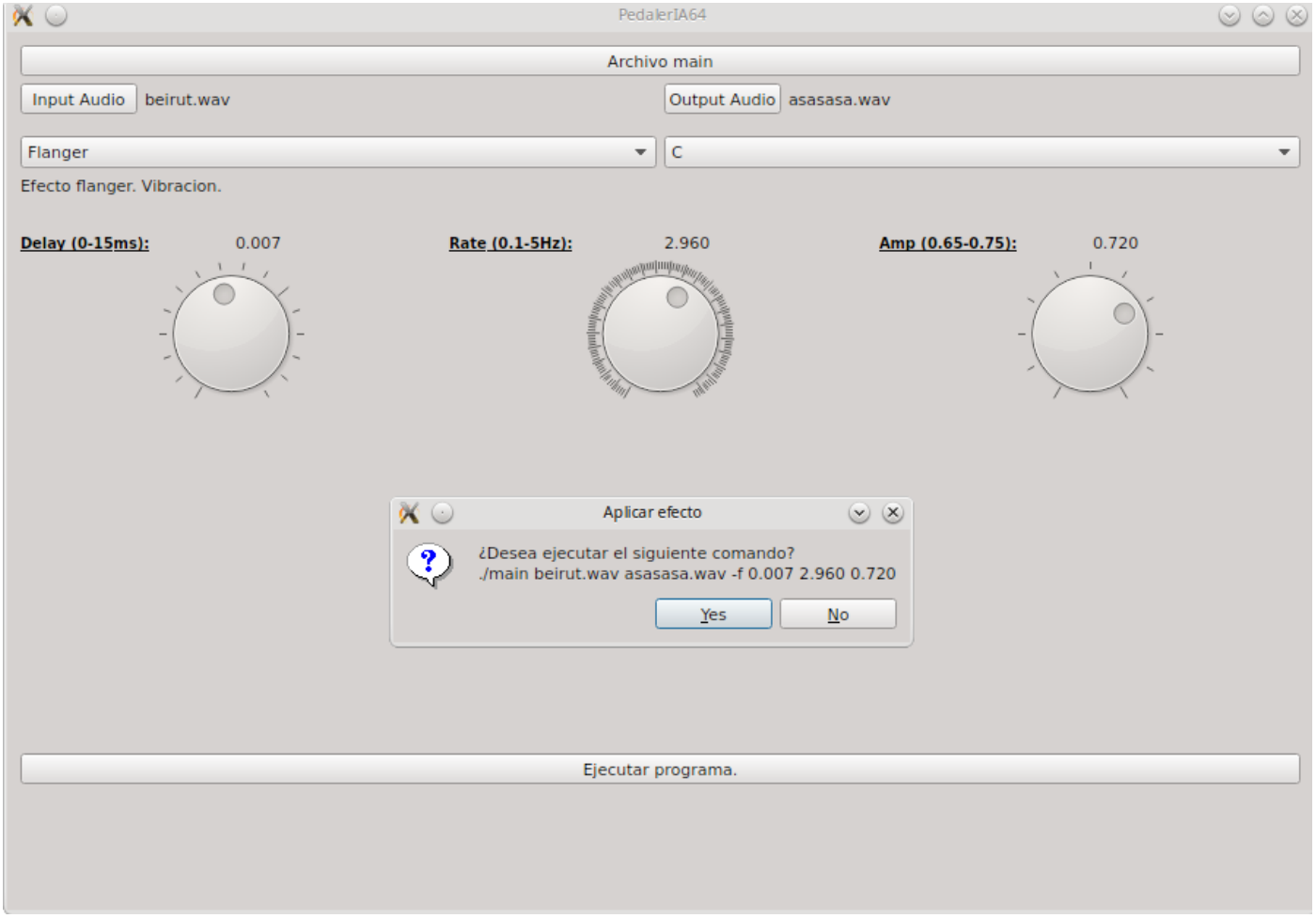

Figura 3: GUI con todas las opciones seleccionadas

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.

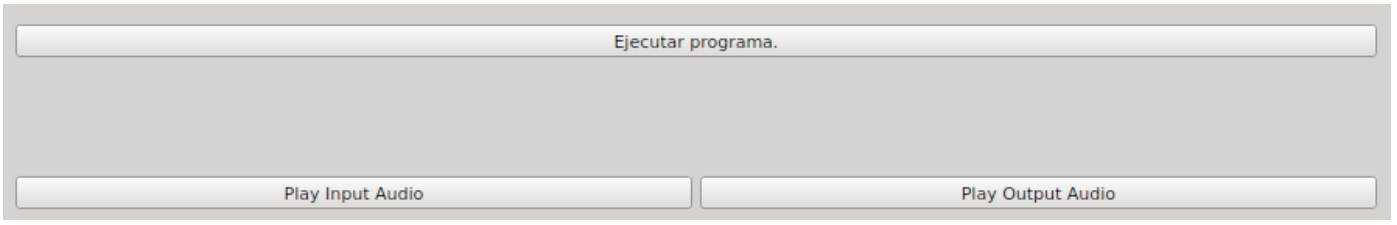

Figura 4: Botones de reproducción

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.

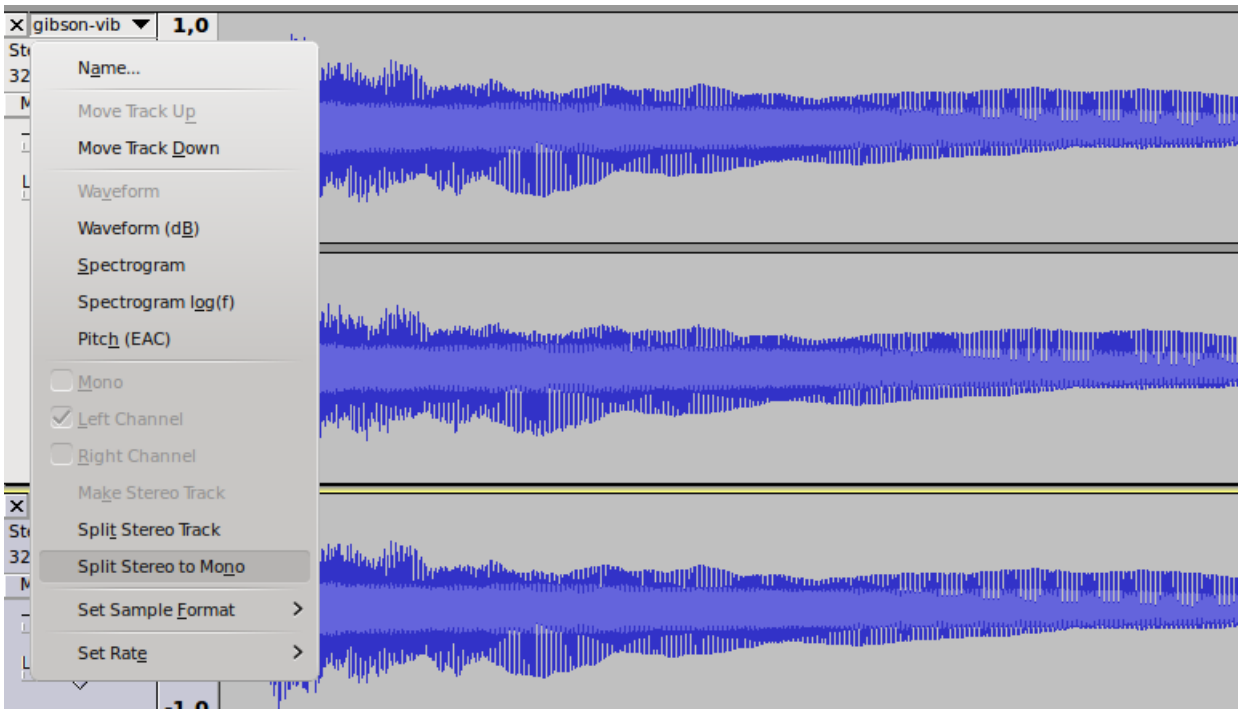

Figura 5: Convertir Stereo a canales Mono

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.

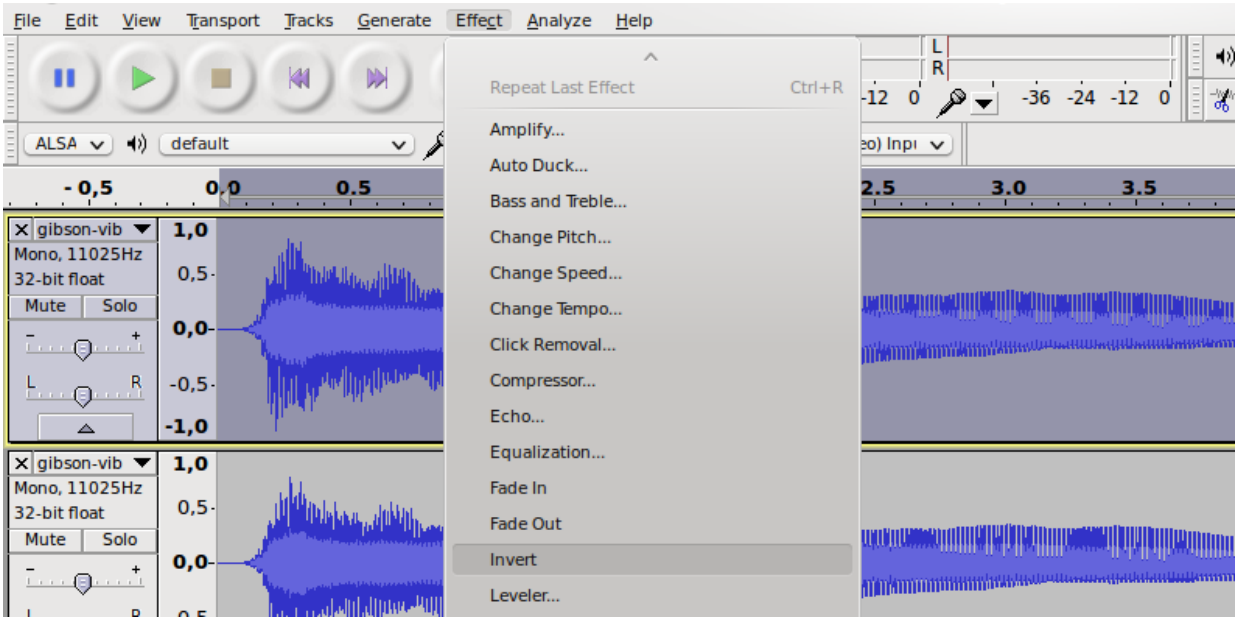

Figura 6: Invertir canal

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.

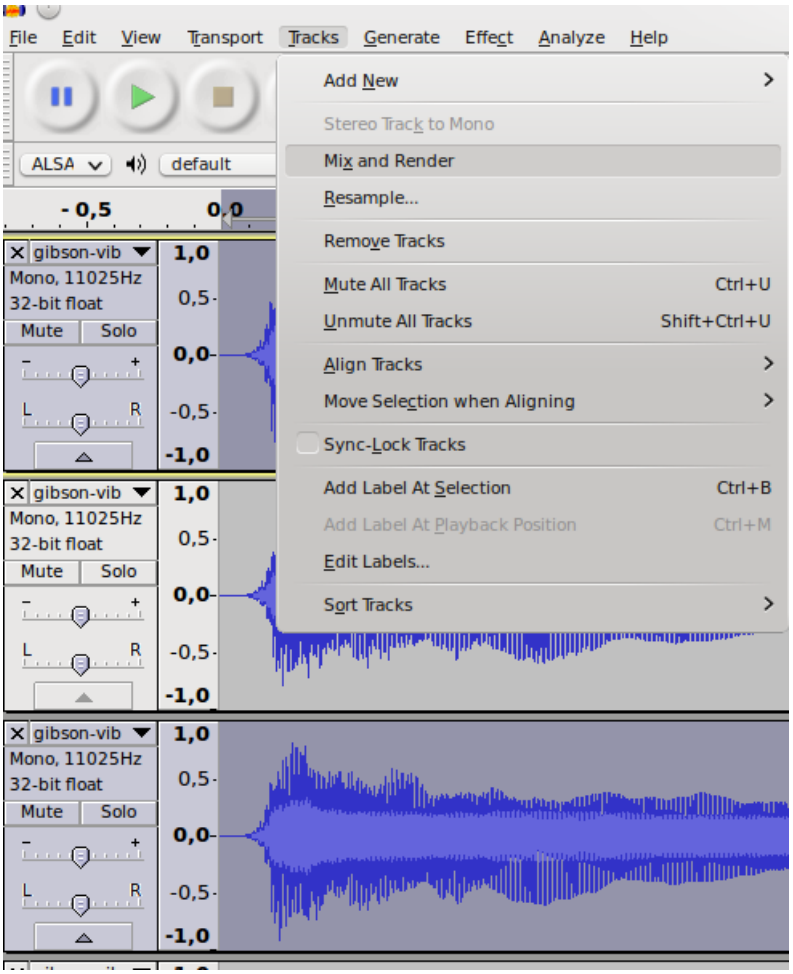

Figura 7: Mix and render

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.

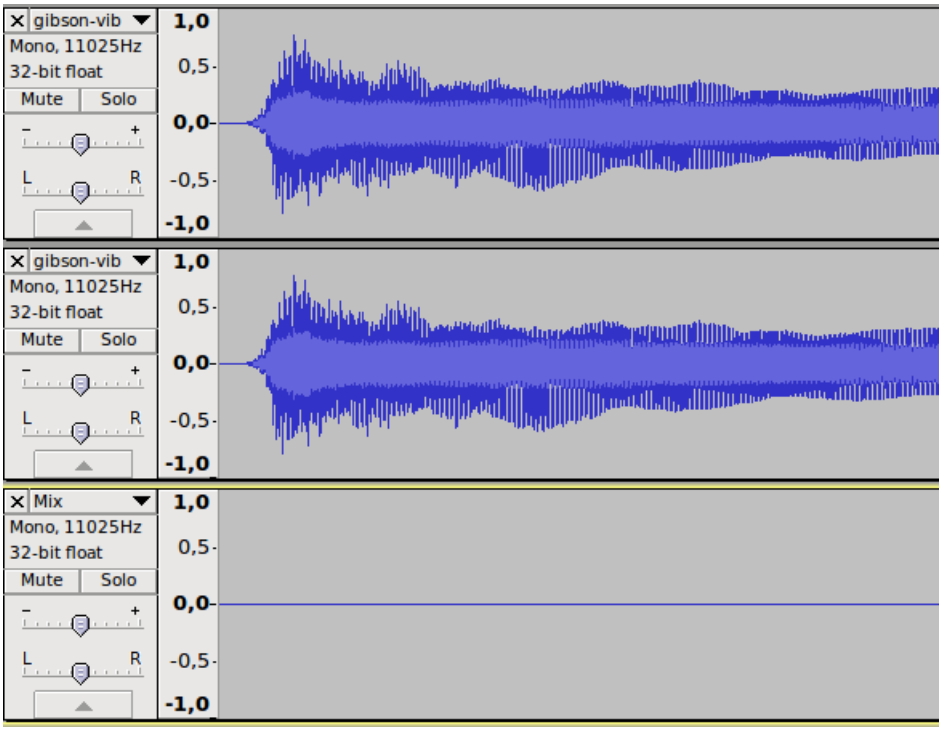

Figura 8: Cancelación de la onda

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

2. Desarrollo

2.1. Estructura del código

La carpeta correspondiente al código del **TP** (**src**) contiene 4 carpetas, y varios archivos. Las carpetas son las siguientes:

- **gui**: contiene únicamente el archivo main.py, que es el que provee la interfaz gráfica para el **TP**.
- **inputExamples**: algunos archivos de audio en formato WAV como ejemplo de entrada.
- **outputExamples**: algunos archivos de audio en formato WAV como ejemplo de salida del programa. En sus nombres se encuentra expresado cuál fue el archivo de audio de entrada utilizado, cuál fue el efecto aplicado (y la versión, **C** o **Assembler**), y los valores de los argumentos de entrada.
- **libs**: incluye los archivos ssemathfun.h (sección SSE Math) y tiempo.h (ver 1.1).

Los archivos son los siguientes:

- **main.c**: el archivo principal, que da nombre al ejecutable. Muestra la ayuda del programa, hace chequeo básico de errores en cuanto a los parámetros de entrada, crea punteros a los archivos de entrada y de salida, y llama al efecto correspondiente.
- **effects.h**: archivo donde se incluyen las librerías utilizadas, se declaran variables y constantes globales, y los encabezados de las funciones tanto en **C** como en **Assembler** (que serán de tipo *extern*). Al final de este archivo se encuentra comentado el template básico (Código común para los efectos (Subsección 2.3)) con el código común que utilizan todos los efectos.
- **effects.c**: aquí se encuentran definidas todas las funciones auxiliares, y los efectos hechos en **C**.
- **effects_asm.c**: el mismo contenido el anterior, pero donde se aplica un efecto o se realiza una operación en una función auxiliar, se llama a la función correspondiente en **Assembler**.
- **ARCHIVO.asm**: cada archivo con extensión .ASM corresponde al efecto o función auxiliar en cuestión.
- **Makefile**: archivo que permite compilar todo el **TP** mediante el comando *make*.

2.2. Funciones auxiliares

2.3. Código común para los efectos

Todos los efectos tienen una parte común en su código, en lo que respecta a *setteo* de variables, la creación de los buffers que serán utilizados, la lectura del archivo de entrada y escritura del archivo de salida. Esa parte común sigue más o menos la siguiente estructura. El tamaño de los buffers depende de

cada efecto en particular, pues no todos necesitan acceder en un ciclo a la misma cantidad de datos. Si bien se definió un tamaño "común" (BUFFERSIZE, definido en **effects.h**, de 8192), en algunos casos un efecto puede necesitar acceder a una cantidad de elementos que es función de alguno de los argumentos (en el caso de los efectos con delay, por ejemplo, donde se necesita que el tamaño del buffer sea múltiplo del argumento en cuestión).

El análisis del rendimiento se hace específicamente sobre las partes del código que involucran la aplicación del efecto, y no en secciones colaterales como lectura y escritura del archivo, creación e inicialización de los buffers, etc.

2.3.1. Pseudocódigo

```
Definición de variables para la creación de los buffers
        (de entrada, de salida, y los necesarios para los efectos)

Crear los buffers necesarios con el tamaño adecuado

Definición de variables utilizadas para los efectos (no ocurre en todos)

Limpieza de los buffers

Mientras haya datos por leer
    Leer archivo de entrada y guardar los datos en el buffer de entrada
    Recorrer el buffer de entrada
      Si el archivo es stereo, calcular promedio de los dos canales

      Contar cantidad de ciclos de reloj
        Aplicar operación sobre los datos de entrada
      Dejar de contar cantidad de ciclos de reloj

      Guardar el resultado en el buffer de salida
    Dejar de recorrer el buffer de entrada

    Guardar el buffer de salida en el archivo de salida
Dejar de leer datos

Liberar memoria utilizada por los buffers
```

2.4. Copy

2.4.1. Descripción

No es un efecto en sí, pero fue desarrollado como prueba de concepto para el preinforme, como método para verificar que se estuviera usando bien la API de libsndfile (1.3.1), la convención de llamado de funciones de ASM desde C, entre otras cosas.

Nota: en el preinforme, este algoritmo utilizaba doubles en vez de floats.

Los resultados de la comparación entre las versiones en **C** y **ASM** de este algoritmo se verán en la sección **??** (**??**), y el análisis en **??** (**??**).

2.4.2. Pseudocódigo

No se adjunta el pseudocódigo para este algoritmo por no aportar nada, pues es literalmente grabar en el buffer de salida lo que contiene el buffer de entrada.

2.4.3. Comando

C:

./main INFILE OUTFILE -c

ASM:

./main INFILE OUTFILE -C

2.5. Delay simple

2.5.1. Descripción

El delay simple es uno de los efectos más básicos en Audio DSP. Consiste simplemente en retrasar la entrada una cantidad arbitraria de segundos; se puede, también, aplicar un modificador para que la señal húmeda sea un porcentaje de la señal original (para que no suenen ambas con la misma intensidad).

A diferencia de otros efectos en los que se hace uso de una cantidad mínima (medida en milisegundos) para delay, aquí tiene una magnitud mayor, por lo que el archivo de salida tendrá una duración mayor que el de entrada. Esto ocasiona que cuando ya no quedan más datos para leer, se realice un ciclo más de escritura, donde se vierte la entrada obtenida en el último ciclo de lectura.

En este efecto, se calcula a cuántas muestras (**frames**) equivale el argumento de delay (que está en segundos), mediante el cálculo

$$delayInFrames = ceil(delayInSec * inFileStr.samplerate).$$

El tamaño de los buffers a usar (*dataBuffIn*, *dataBuffOut* y *dataBuffEffect*) será el máximo entre *delayInFrames* y el mayor múltiplo de dicho valor que sea menor que BUFFERSIZE (8192). *dataBuffEffect* contiene siempre la entrada del ciclo anterior; de este modo, nos aseguramos que en cada ciclo de lecto/escritura se pueda acceder mediante *dataBuffEffect* a lo que se leyó en el ciclo anterior, que pasó hace una cantidad *delay* de segundos, que es lo que necesita el efecto.

Los resultados de la comparación entre las versiones en **C** y **ASM** de este algoritmo se verán en la sección **??** (**??**), y el análisis en **??** (**??**).

2.5.2. Pseudocódigo

```
Argumentos: delay, decay
dataBuffOut.canalDerecho = dataBuffIn.muestraCicloAnterior * decay
dataBuffOut.canalIzquierdo = dataBuffIn.muestracicloActual
```

2.5.3. Comando

C:

./main INFILE OUTFILE -d delay decay

ASM:

./main INFILE OUTFILE -D delay decay

- *delay*: argumento sin rango específico, pero por conveniencia se lo limitó en la GUI al rango [0.0, 5.0] segundos. Es la cantidad de segundos de retraso que se quiere tener en la señal húmeda.
- *decay*: argumento con rango entre 0.00 y 1.00. Es el porcentaje de la amplitud de la señal seca que se quiere en la señal húmeda.

2.6. Flanger

2.6.1. Descripción

Flanger es un efecto que en sus orígenes se conseguía del siguiente modo. Se tenían dos cintas con el mismo material de audio, el original y una copia, y se mezclaban en un tercer canal. Este hecho ya generaba una pequeña diferencia de fase; pero además, durante la reproducción de la cinta duplicada, se presionaba sutilmente con el dedo el borde (*flange*) de la bobina de la cinta, lo que afectaba la velocidad de reproducción y agregaba a la diferencia de fase una leve diferencia temporal, prounounciado el efecto.

En el caso digital, se utiliza un **LFO** (*low frequency oscillator*) para variar la velocidad de reproducción de la copia. El LFO se genera calculando el valor absoluto del seno de un valor que es función del índice de la muestra actual y del parámetro **rate** del efecto; este resultado parcial (que está entre 0 y 1, y es una onda que varía periódicamente según el índice de la muestra) se multiplica por el parámetro **delay**; de este modo, se obtiene para el índice actual (señal original), cuál es la muestra anterior (de la señal duplicada) que se le debe sumar. Esta señal original no se añade en su totalidad, sino que se multiplica por el parámetro **amp** para atenuarla levemente.

Los resultados de la comparación entre las versiones en **C** y **ASM** de este algoritmo se verán en la sección **??** (**??**), y el análisis en **??** (**??**).

2.6.2. Pseudocódigo

```
Argumentos: delay, rate, amp
Para cada muestra
  arg_seno = 2*PI*indice_muestra * rate/archivoEntrada.samplerate
  seno_actual = | seno(arg_seno) |
  delay_actual = ceiling(seno_actual*delay)
  indice_copia = indice_muestra-delay_actual

  dataBuffOut.canalDerecho =  dataBuffIn.muestraCicloActual*amp +
                              dataBuffIn.muestra_indice_copia*amp
  dataBuffOut.canalIzquierdo = dataBuffIn.muestraCicloActual
```

2.6.3. Comando

C:

./main INFILE OUTFILE -f delay rate amp

ASM:

./main INFILE OUTFILE -F delay rate amp

- *delay*: argumento con rango entre 0.000-0.015s. Es el delay máximo que puede tener la muestra duplicada.
- *rate*: argumento con rango entre 0.1-5Hz. Es la frecuencia del LFO.

- *amp*: argumento con rango entre 0.65 y 0.75. Según bibliografía consultada [5], el valor es 0.7, pero para hacerlo variable se eligió el rango 0.65-0.75. Es el porcentaje de la amplitud de la señal duplicada.

[5] [7, p. 77]

2.7. Vibrato

2.7.1. Descripción

Vibrato es un efecto que consiste en *la variación periódica de la frecuencia de un sonido*[6]. Esta variación puede conseguirse utilizando un **LFO**, como en el efecto anterior, pero en este caso lo que oscilará es la frecuencia (**pitch**) del sonido, y no el delay de la señal (que además, en el vibrato, el delay toma un valor mucho menor, 0-3ms).

Para este efecto, se utilizó un buffer circular (*dataBuffEffect*). El **LFO** toma los mismos parámetros que en Flanger (Subsección 2.6), el argumento **mod** del efecto (que simboliza la frecuencia de modulación) y el índice de la muestra actual. La parte entera de este resultado es el índice de la muestra que se utiliza en la señal húmeda, mientras que la parte fraccionaria se usa para realizar una interpolación entre la mencionada muestra y la anterior.

Los resultados de la comparación entre las versiones en **C** y **ASM** de este algoritmo se verán en la sección **??** (**??**), y el análisis en **??** (**??**).

2.7.2. Pseudocódigo

```
Argumentos = mod, depth
depth = redondear(depth*archivoEntrada.samplerate)
delay = depth
mod = mod/archivoEntrada.samplerate
Para cada muestra
  mod_actual = sen(mod*2*PI*indice_actual)
  tap = 1+delay+depth*mod_actual
  indice_muestra_humeda = floor(tap)
  frac = tap - indice_muestra_humeda

  dataBuffOut.canalDerecho =  dataBuffIn.muestra_humeda*frac +
                              dataBuffIn.(muestra_humeda-1)*(1-frac)
  dataBuffOut.canalIzquierdo = dataBuffIn.muestraCicloActual
```

2.7.3. Comando

C:

./main INFILE OUTFILE -v depth mod

ASM:

./main INFILE OUTFILE -V depth mod

- *depth*: argumento con rango entre 0.000 y 0.003s. Es el delay de la señal de entrada.
- *mod*: argumento con rango entre 0.10 y 5.00Hz. Es la frecuencia de modulación del efecto.

[6] `https://es.wikipedia.org/wiki/Vibrato`

2.8. Bitcrusher

2.8.1. Descripción

En la sección Audio (Subsección 1.2) se habló sobre dos medidas de audio digital que son quienes determinan la *calidad* del sonido: la frecuencia de muestreo (*sampling rate*) y la resolución de la muestra (*bit rate*). El efecto **bitcrusher** provoca una distorsión de la señal original, reduciendo tanto el muestreo (se toma en la señal húmeda una de cada cierta cantidad de muestras de la señal original, **downsampling**) como la cantidad de bits con la que se puede expresar cada muestra.

Los resultados de este efecto muchas veces hacen recordar a la música de los primeros juegos de consola (también conocidas como **chiptunes**), pues eran generadas con chips de 8 bits.

En la siguiente imagen, puede apreciarse cómo el efecto distorsiona la señal original (onda superior).

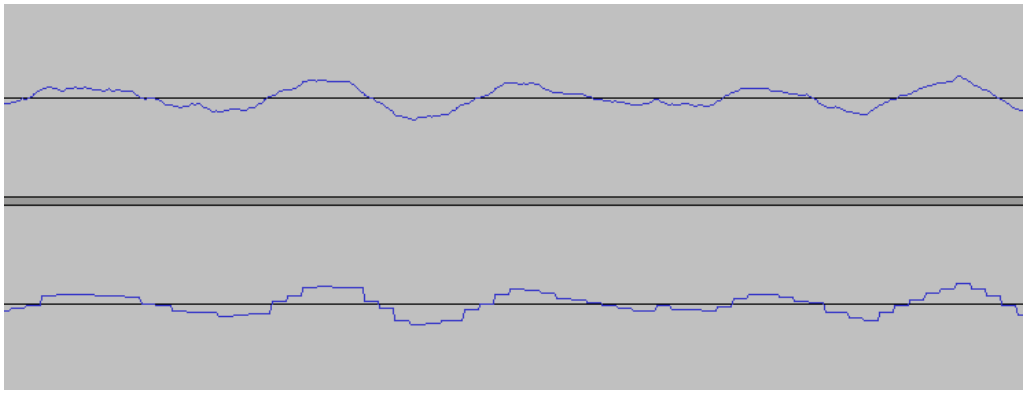

Figura 9: Señal superior: original. Señal inferior: al aplicar el efecto.

Los resultados de la comparación entre las versiones en **C** y **ASM** de este algoritmo se verán en la sección **??** (**??**), y el análisis en **??** (**??**).

2.8.2. Pseudocódigo

2.8.3. Comando

C:

./main INFILE OUTFILE -

ASM:

./main INFILE OUTFILE -

- *bits*: argumento con rango entre 1 y 16. Es la cantidad de bits que se pueden utilizar para el valor de cada muestra.
- *freq*: argumento con rango entre 1 y 11025Hz. Es la frecuencia de sampleo de la señal húmeda.

2.9. WahWah

2.9.1. Descripción

Los resultados de la comparación entre las versiones en **C** y **ASM** de este algoritmo se verán en la sección **??** (**??**), y el análisis en **??** (**??**).

2.9.2. Pseudocódigo

2.9.3. Comando

C:

./main INFILE OUTFILE -

ASM:

./main INFILE OUTFILE -

- : argumento con rango entre .
- : argumento con rango entre .

2.10. Problemas en el Desarrollo

3. Resultados

4. Analisis

5. Bibliografía tentativa

5.1. Libros

Referencias

[1] Richard Boulanger, Victor Lazzarini *The Audio Programming Book*, 2011, The MIT Press, Massachussets (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, Massachussets (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)

5.2. Internet - links generales

- API de libsndfile
- Ejemplos de uso de pyqt5
- Más ejemplos de uso de pyqt5
- Curso de Stanford: Introduction to Digital Filters
- Curso de Stanford: Physical Audio Signal Processing
- Wikipedia: WAV file
- Wikipedia: Audio signal
- Wikipedia: Sampling rate
- Wikipedia: Bit rate
- Valgrind: opciones para callgrind
- KCacheGrind: profiling tips
- Definición flanging
- Conversor de números al formato IEEE754
- Calculadora para cambio de base

- http://stackoverflow.com
- http://www.musicdsp.org/
- http://www.kvraudio.org/
- https://en.wikibooks.org/wiki/X86_Assembly/SSE

5.3. Fuentes de cosas específicas del TP

- Figura 9: http://msp.ucsd.edu/techniques/v0.11/book-html/node7.html
- Ejemplo de uso de libsndfile (Subsubsección 1.3.1): http://www.labbookpages.co.uk/audio/wavFiles.html#c
- Algoritmo Delay: variación de Stack Overflow, MusicDSP
- Algoritmo Flanger: adaptación de Cardiff School of Computer Science & Informatics: MATLAB, DSP, Graphics CM0268
- Algoritmo Bitcrusher: adaptación de MusicDSP