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

Índice

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 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 (en particular 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; los problemas encontrados a lo largo del desarrollo del TP se tratarán 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 Análisis y Conclusiones (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

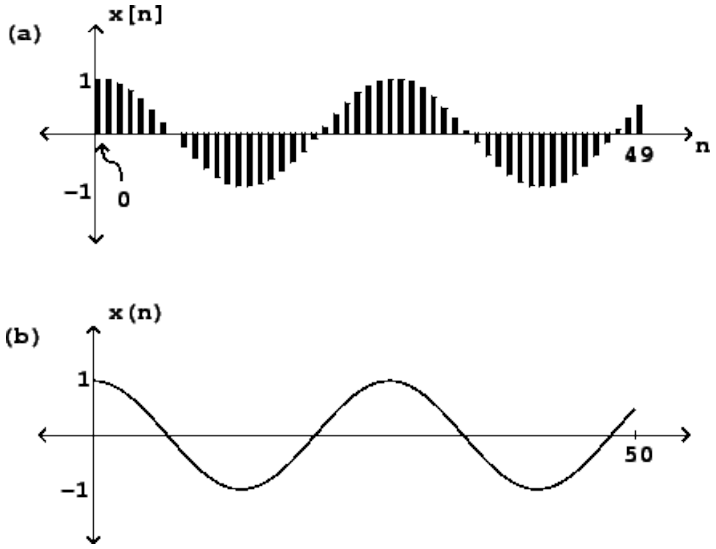


Figura 1: Señales de audio

Simplificándolo extremamente, una señal de audio (**(b)** en la figura 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 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

Las funciones de la librería utilizada son las siguientes:

- **sf_open**: para abrir un archivo de audio.
- **sf_strerror**: para descubrir el error en caso de una apertura fallida de archivo.
- **sf_seek**: para reapuntar los punteros de lectura/escritura del archivo. Útil para volver a leer un archivo abierto (por ej., cuando se corren varias iteraciones del mismo efecto, o en el caso de normalización post-aplicación del efecto (2.13).
- **sf_readf_float**: para leer una cantidad determinada de muestras (o menos, cuando es el final del archivo) de un archivo ya abierto. La función las convierte a tipo float, independientemente del tipo que tengan las muestras en el archivo original.
- **sf_write_float**: para escribir una cantidad determinada de muestras con valores de tipo float
- **sf_write_sync**: fuerza la escritura de los datos en el buffer al archivo de salida (*flush*).

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 Análisis y Conclusiones (Sección 4), gracias al uso del profiler se descubrió que

[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`

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

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.qtmultimedia** (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 CANT_ITER 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.
- **CANT_ITER** cantidad de iteraciones de los efectos.
- **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

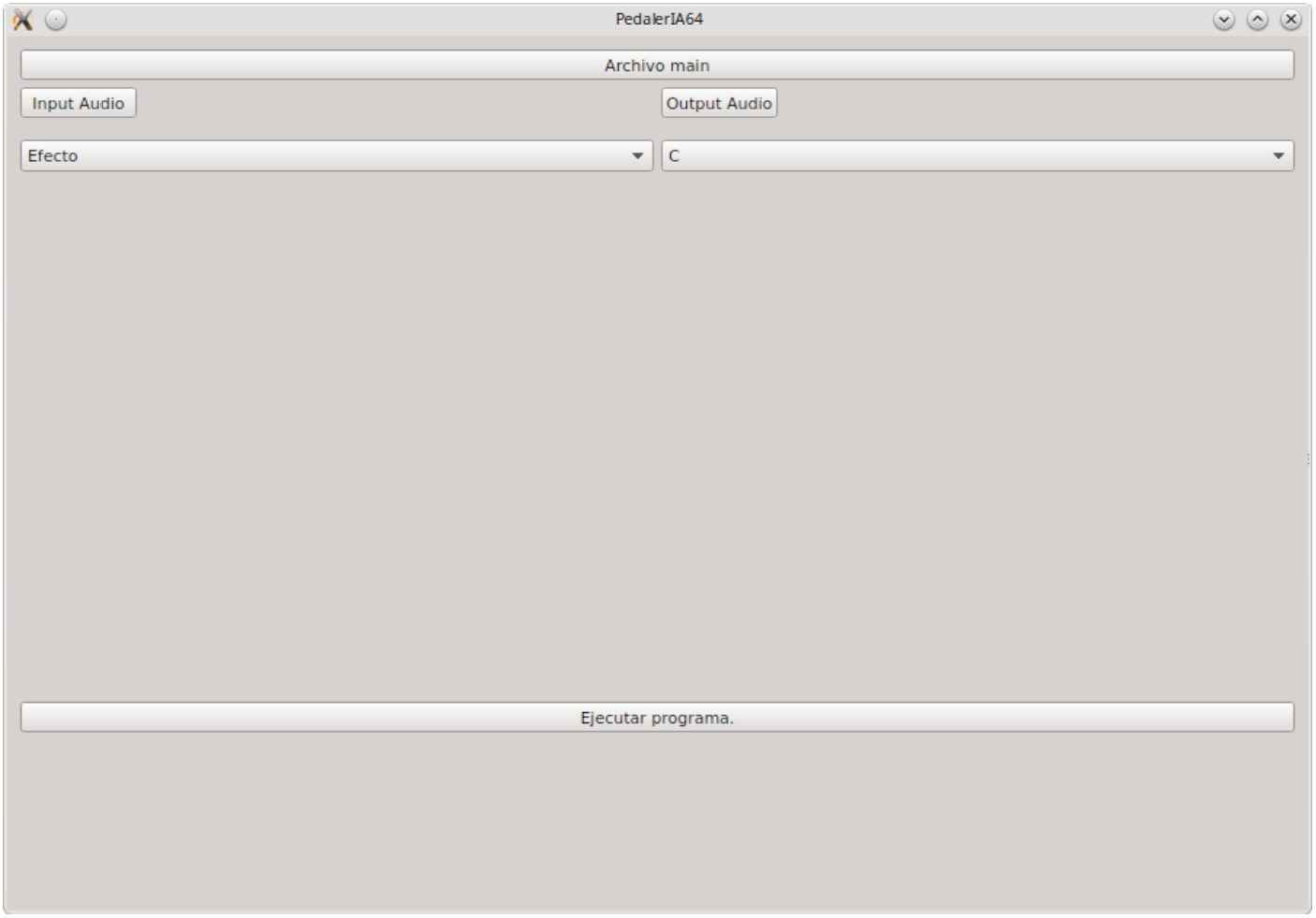


Figura 2: GUI

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.

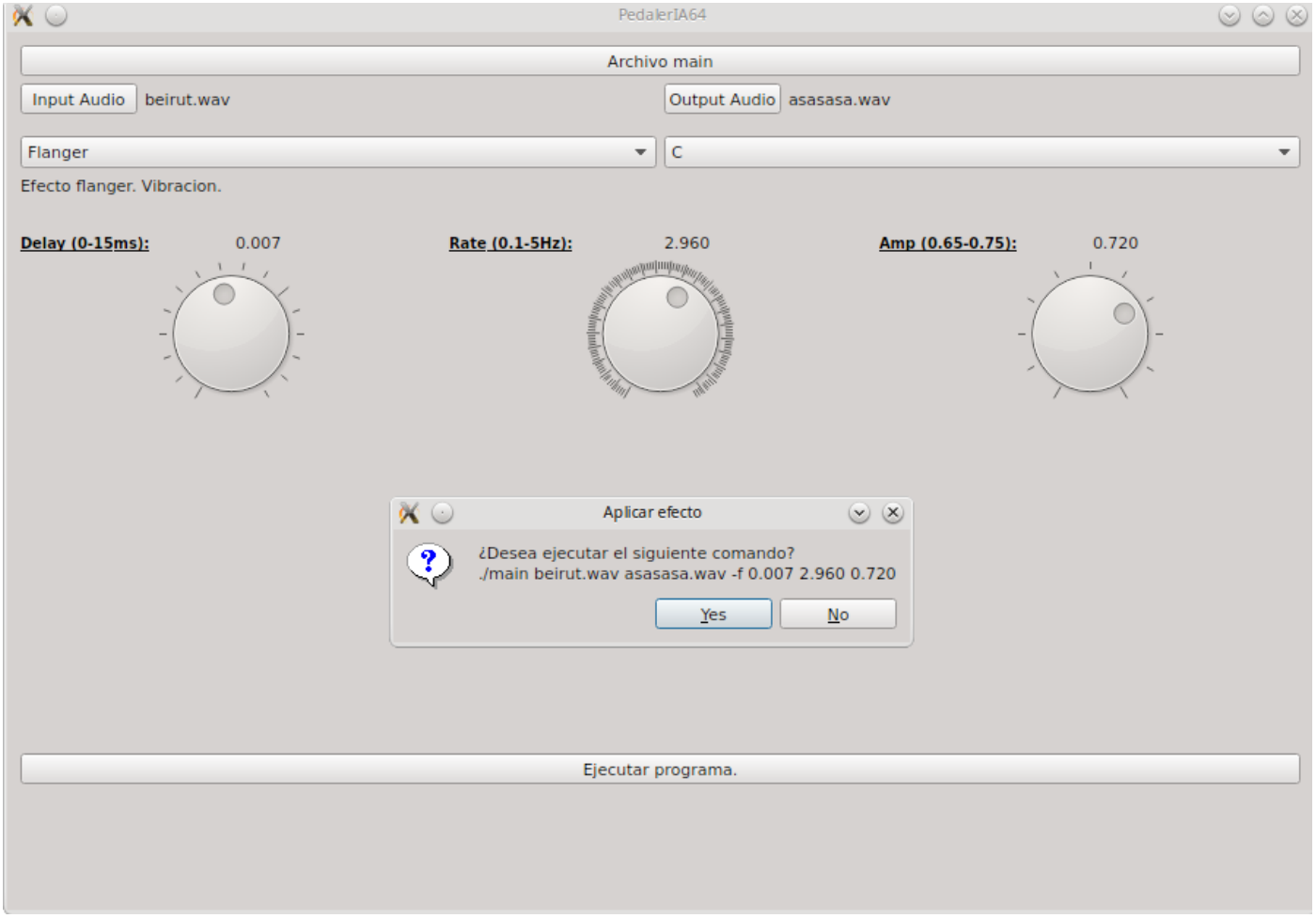


Figura 3: GUI con todas las opciones seleccionadas

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.

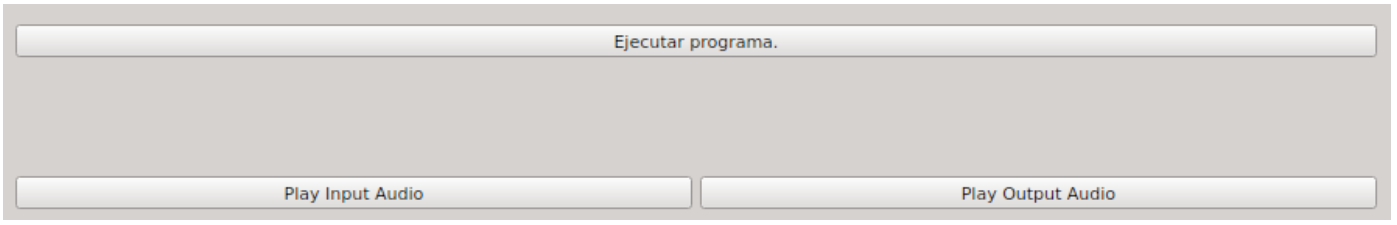


Figura 4: Botones de reproducción

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 cuál es la opción que falta.

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.

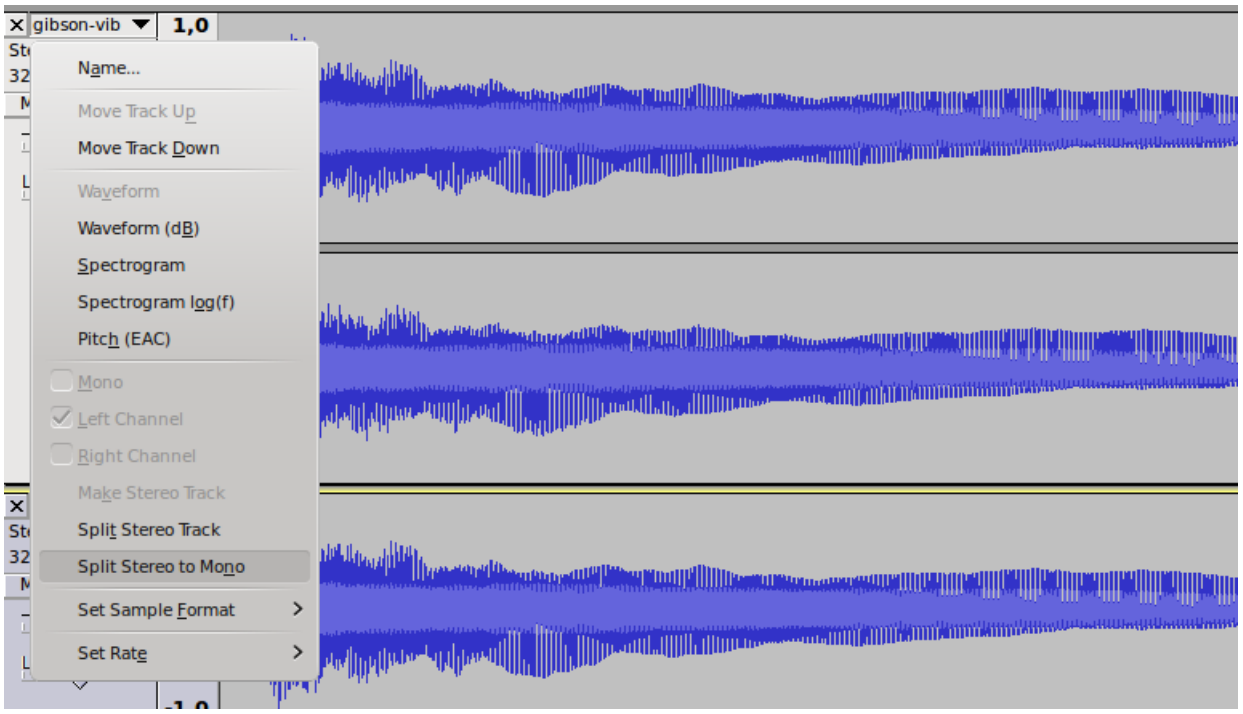


Figura 5: Convertir Stereo a canales Mono

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.

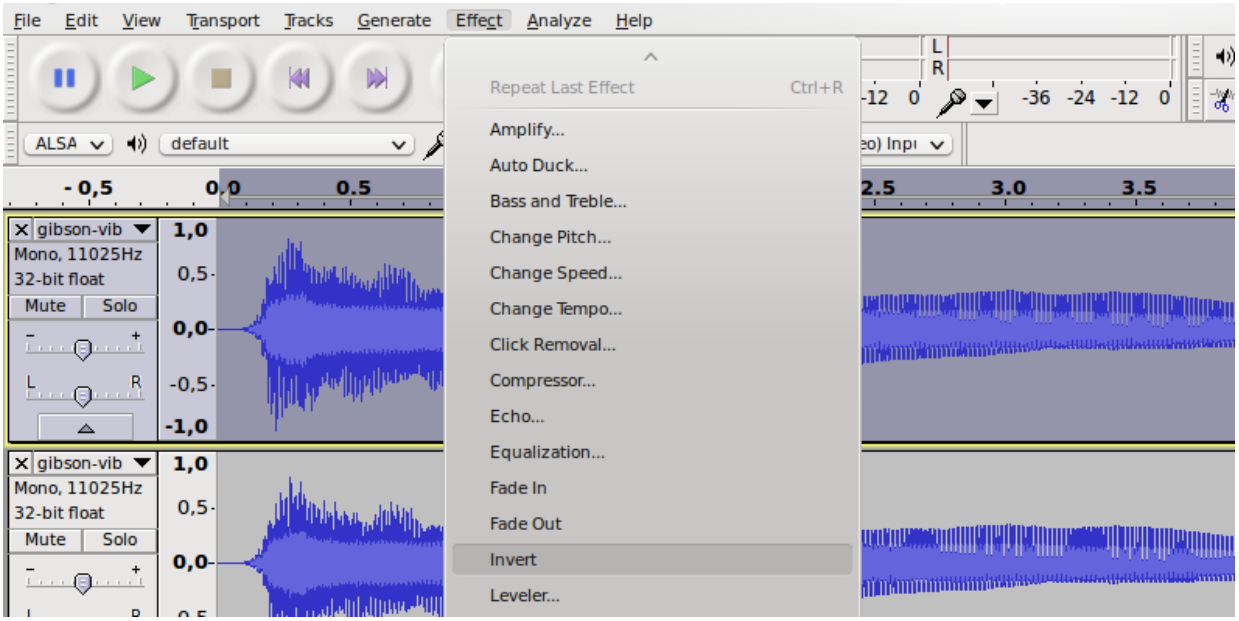


Figura 6: Invertir canal

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.

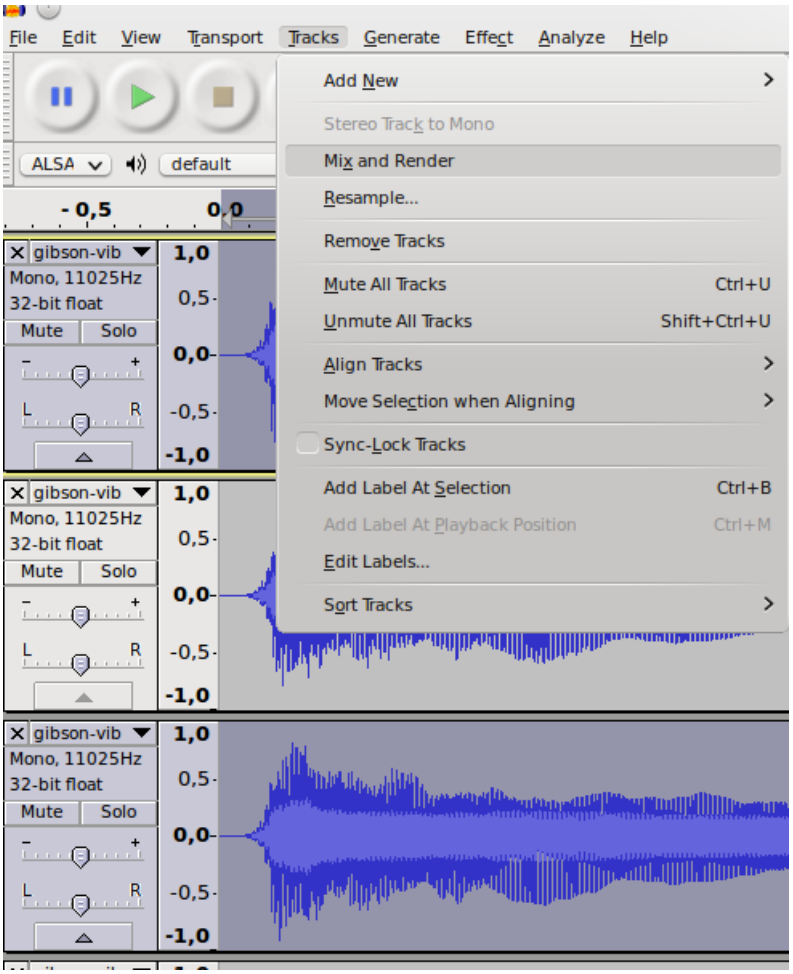


Figura 7: Mix and render

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.

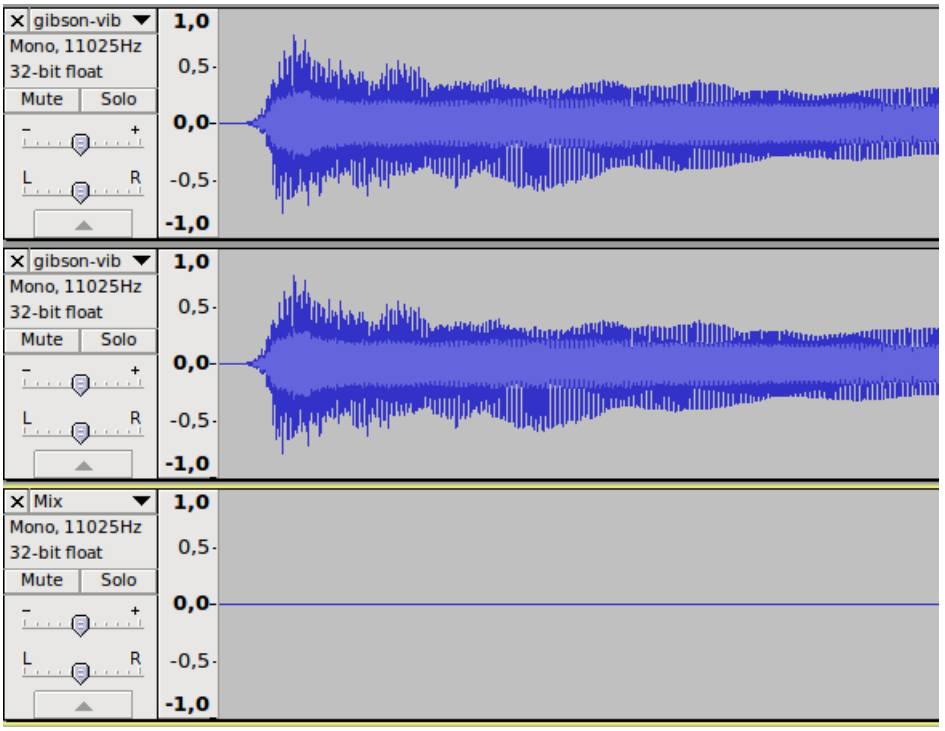


Figura 8: Cancelación de la onda

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

2.2.1. Normalización

Como se verá en la sección Problemas en el Desarrollo (Subsección 2.10), fue necesario desarrollar un algoritmo para normalizar un archivo de audio. Para esto se desarrollaron dos rutinas, una que busca la muestra de mayor valor absoluto en el archivo (**maxsamp_right**), y otra que normaliza el archivo completo en base a dicha muestra (**normalization_right**).

Como la normalización sólo se realiza sobre un archivo al que le fue aplicado el efecto, sabemos que éste siempre va a ser stereo (1.2) y, además, que la operación sólo será necesario hacerla sobre el canal derecho, que es el que tiene la señal húmeda (de allí el nombre **_right**).

2.2.2. Seno

En un principio, se intentó utilizar una librería para calcular el seno con instrucciones SIMD (SSE Math (Subsubsección 1.3.2)) con bastante precisión; sin embargo, la pérdida de rendimiento al usar dicha librería de los algoritmos en ASM frente a los de C era muy notoria. Por esa razón, se recurrió a una solución [5] para calcular el seno que realiza una aproximación mediante parábolas. Esta aproximación, sin embargo, es bastante buena, ya que si se comparan los archivos de salida de **C** (*sinf* de **math.h**, muy buena aproximación) con los de **ASM**, la diferencia entre las señales húmedas no es tan grande.

La rutina se encuentra en el archivo **sine.asm**, aunque el código para calcular el seno se terminó poniendo en cada lugar donde se usara (porque el cálculo de los argumentos era diferente para cada efecto, al igual que las operaciones posteriores a la obtención del seno, y no se consideró que tuviera sentido hacer todo el pasaje de parámetros, llamado a la rutina, etc., por unas pocas líneas de código).

[5] `http://forum.devmaster.net/t/fast-and-accurate-sine-cosine/9648/4`

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.

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 Delay (Subsección 3.1), y el análisis en Análisis y Conclusiones (Sección 4).

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.

La rutina en **ASM** se encuentra dividida en dos archivos, **flanger.asm** y **flanger_index_calc.asm**. Por un lado (en el último archivo mencionado), se calculan cuáles serán los índices de las muestras que se usarán para la señal húmeda (cálculo de los argumentos del seno, aplicación del mismo, y posterior modificación, obteniendo todos los indice_copia), y por el otro se aplica el efecto utilizando tales muestras (primer archivo).

Los resultados de la comparación entre las versiones en **C** y **ASM** de este algoritmo se verán en la sección Flanger (Subsección 3.2), y el análisis en Análisis y Conclusiones (Sección 4).

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 [6], 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.

[6] [4, 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*[7]. 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.

Al igual que para el caso del Flanger (Subsección 2.6), la rutina en **ASM** se encuentra dividida en dos archivos, **vibrato_index_calc.asm** y **vibrato.asm**.

Los resultados de la comparación entre las versiones en **C** y **ASM** de este algoritmo se verán en la sección Vibrato (Subsección 3.3), y el análisis en Análisis y Conclusiones (Sección 4).

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.

[7] 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 (**quantization**).

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).

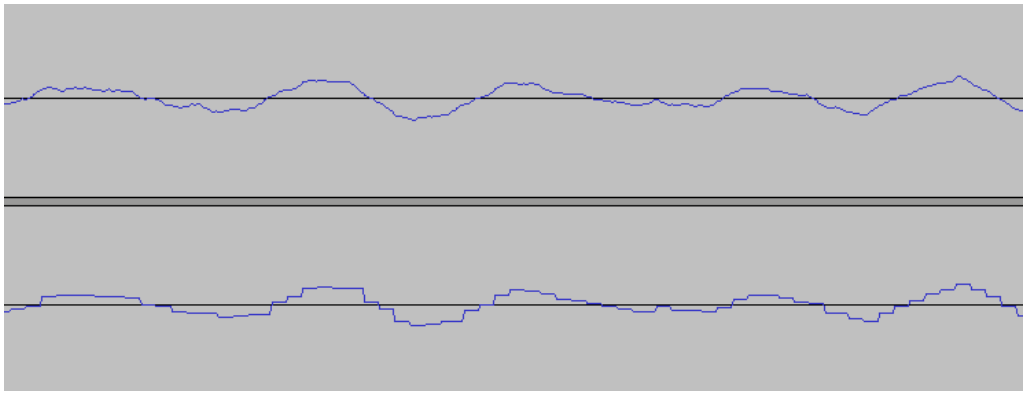


Figura 9: Señal superior: original. Señal inferior: al aplicar el efecto.

Los resultados de la comparación entre las versiones en **C** y **ASM** de este algoritmo se verán en la sección Bitcrusher (Subsección 3.4), y el análisis en Análisis y Conclusiones (Sección 4).

2.8.2. Pseudocódigo

```
Argumentos: bits, freq
step = 1/2^(bits);
phasor = last = 0;
normFreq = freq/archivoEntrada.samplerate

Para cada muestra
  phasor = phasor + normFreq;
  if (phasor >= 1.0) {
  // downsampling, tomo 1 muestra de cada cierta cantidad
    phasor = phasor - 1.0;
    last = step * floor( input(i)/step + 0.5 );
    // quantization, reduzco la calidad
  }

  dataBuffOut.canalDerecho = last
  dataBuffOut.canalIzquierdo = dataBuffIn.muestraCicloActual
```

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 2048 y 11025Hz. Es la frecuencia de sampleo de la señal húmeda.

2.9. WahWah

2.9.1. Descripción

WahWah es uno de los efectos más conocidos y fáciles de identificar, pues su nombre describe el sonido que genera. El efecto se genera con la aplicación de un filtro pasabanda (sólo deja pasar las frecuencias entre dos valores, mínimo y máximo, preestablecidos) que varía con el tiempo. El filtro pasabanda se implementó mediante un filtro de estado variable (que permite separar a la señal original x en tres, debajo del pasa banda y_l, en el pasa banda y_b, y por arriba del pasa banda y_h), que sigue las siguientes ecuaciones:

$$y_l(n) = F_c * y_b(n) + y_l(n-1)$$
$$y_b(n) = F_c * y_h(n) + y_b(n-1)$$
$$y_h(n) = x(n) - y_l(n-1) - Q_1 * y_b(n-1)$$

F_c es un valor que depende de las frecuencias de corte del filtro pasa banda (implementado con una onda triangular con los valores centrales de frecuencia), y Q_1 está relacionado con el argumento *damp*, que especifica el tamaño de las bandas. La onda triangular es cíclica, tiene mínimo **minf** y máximo **maxf**, y en cada punto va creciendo/decreciendo en el valor **delta**. Para no guardar los valores de la onda triangular en un buffer, se generan puntualmente; primero se calcula para cada muestra si correspondería a un punto de un ciclo de crecimiento ("par") o decrecimiento ("impar") de la onda, y luego qué punto de ese ciclo es. Este valor es intermedio al cálculo final del F_c que se ve en la ecuación de arriba.

En este efecto se dio la particularidad de que en algunos casos las operaciones *saturaban* algunas muestras. Por esa razón se debió optar por la Normalización (Subsubsección 2.2.1) del archivo, por razones que se explicarán en Normalización (Subsección 2.14), primero se tuvo que "achicar" el valor de las muestras, por lo que se les aplicó un modificador de 0.1 a la señal húmeda. Si bien esto aumenta el error numérico, fue la solución más convincente.

La rutina en **ASM** se encuentra dividida también en dos archivos, **wah_wah_index_calc.asm** y **wah_wah.asm**.

Los resultados de la comparación entre las versiones en **C** y **ASM** de este algoritmo se verán en la sección WahWah (Subsección 3.5), y el análisis en Análisis y Conclusiones (Sección 4).

2.9.2. Pseudocódigo

```
Argumentos: damp, minFreq, maxFreq, wahWahFreq
q1 = 2*damp
delta = wahWahFreq/archivoEntrada.samplerate
triangleWaveSize = floor((maxFreq-minFreq)/delta)+1

yh = yb = yl = 0

Para cada muestra
  cicloPar = (indice_muestra_actual/triangleWaveSize)%2
  esteCiclo = (indice_muestra_actual) %triangleWaveSize + 1

  fc = (1 - cicloPar) * (minFreq + (esteCiclo-1) *delta) +
          (cicloPar) * (maxFreq - esteCiclo*delta)
```

```
            ; valor del punto de la onda triangular

    fc = 2*seno(PI*fc/archivoEntrada.samplerate)

    yh = archivoEntrada.muestraCicloActual - yl - q1*b     ; aplico filtro
    yb = fc * yh + yb
    yl = fc * yb + yl

    dataBuffOut.canalDerecho = 0.1*yb
    dataBuffOut.canalIzquierdo = dataBuffIn.muestraCicloActual
```

2.9.3. Comando

C:

./main INFILE OUTFILE -w damp minFreq maxFreq wahwahFreq

ASM:

./main INFILE OUTFILE -W damp minFreq maxFreq wahwahFreq

- *damp*: argumento con rango entre 0.01-0.10. Determina el tamaño del filtro (cuánto afecta a la señal).
- *minFreq*: argumento con rango entre 400-1000Hz. Es la frecuencia de corte inferior del filtro pasabandas.
- *maxFreq*: argumento con rango entre 2500-3500Hz. Es la frecuencia de corte superior del filtro pasabandas.
- *wahwahFreq*: argumento con rango entre 1000-3000Hz. Es la frecuencia del filtro.

2.10. Problemas en el Desarrollo

En esta sección se detallarán algunos de los problemas que se fueron teniendo a lo largo del TP y cómo se solucionaron.

2.11. Intercalar sonido original con efecto

Luego de la recomendación de separar el audio original y el efecto en los distintos canales (1.2), se me dificultó saber cómo hacerlo. El problema provenía de no saber cómo hacer para intercalar las muestras correspondientes a los distintos canales, si no era serializando (haciendo sobre cada muestra por separado, poniéndolas en algún registro xmm, y después shuffleando para seguir con la muestra siguiente) las operaciones.

Analizando el set de instrucciones disponibles, se encontraron **punpckchdq** y **punpckcldq** que permiten intercalar los valores "altos" o "bajos" (respectivamente) de dos registros xmm distintos. Teniendo un registro con la señal original y otro con la señal húmeda, se puede lograr justamente lo que se quería.

Nota: si bien las figuras 10 y 11 tienen 64 bits, las instrucciones operan en x64 sobre los registros xmm enteros, de 128 bits.

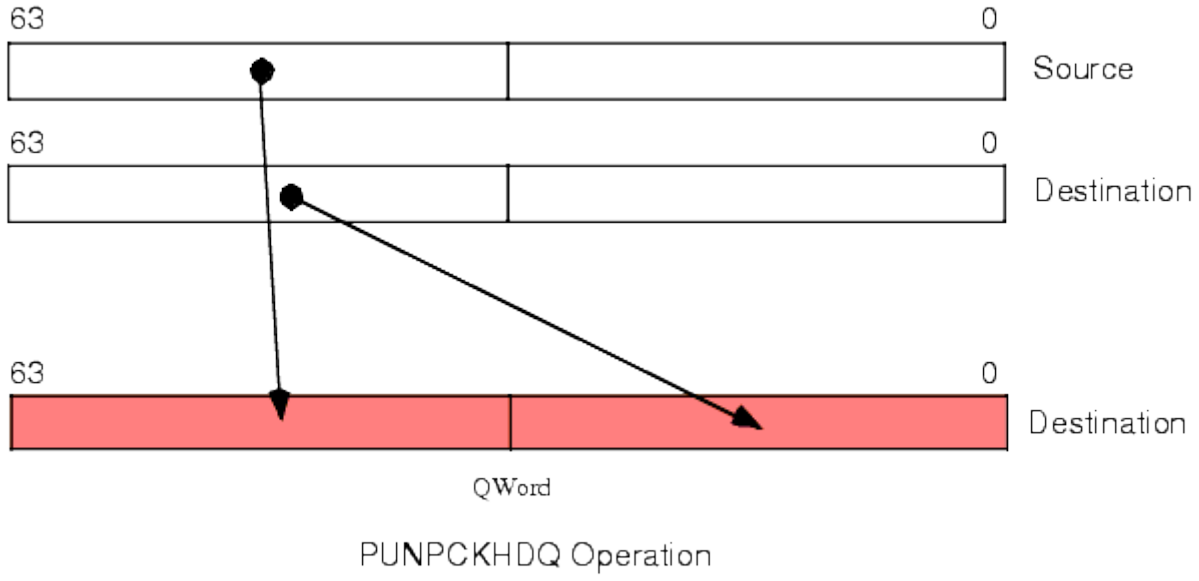


Figura 10:

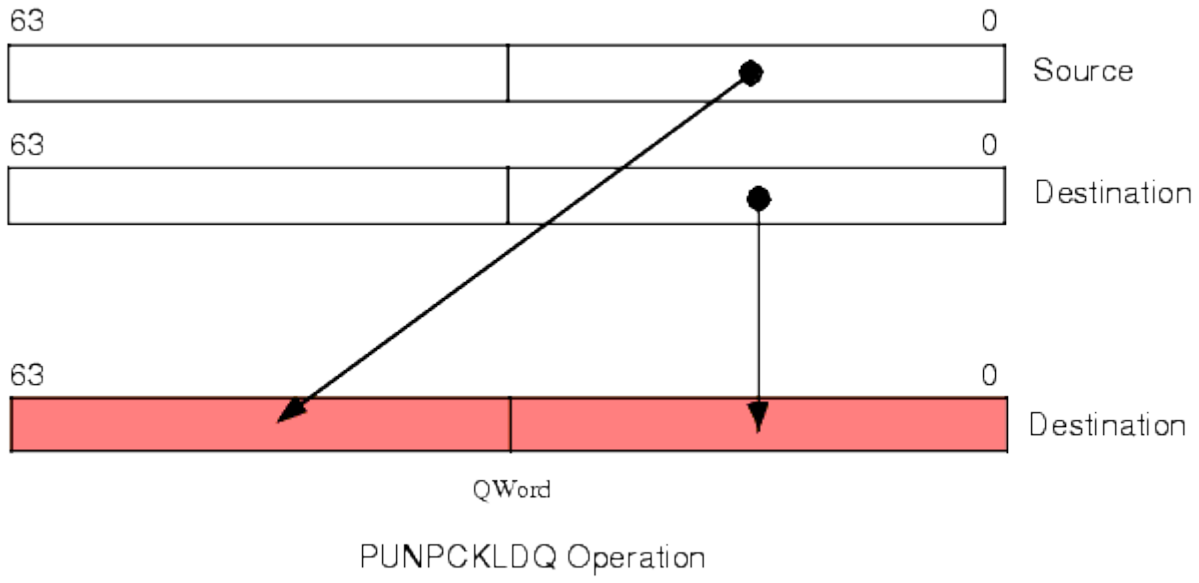
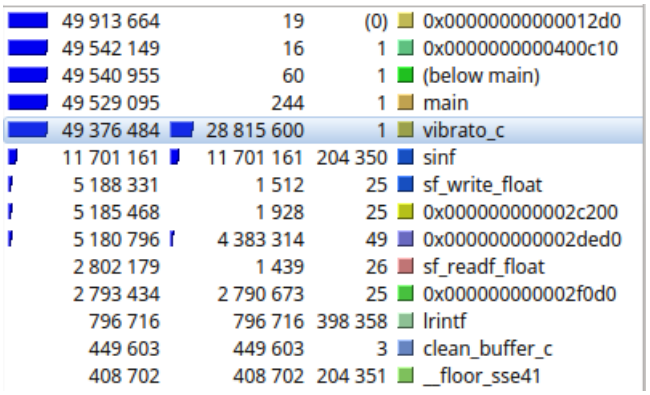


Figura 11:

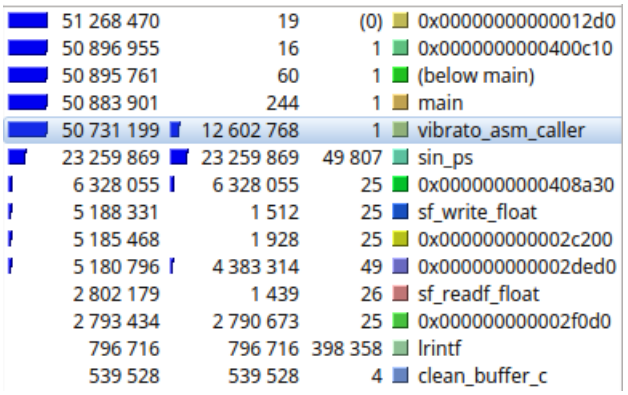
2.12. Seno

En las primeras versiones finales de algunos efectos (flanger, wahwah, vibrato), se utilizó la librería SSE Math (Subsubsección 1.3.2) para poder utilizar el seno sobre un vector de valores. Sin embargo, el análisis de rendimiento mediante la librería **tiempo.h** reveló que los efectos en Assembler eran considerablemente más lentos que los de C, aunque las razones no eran claras. Se realizó entonces un análisis mediante una herramienta de profiling, Callgrind (Sección 1.3.5). Los resultados de este análisis se ven en las siguientes imágenes:



49 913 664	19	(0)	0x00000000000012d0
49 542 149	16	1	0x0000000000400c10
49 540 955	60	1	(below main)
49 529 095	244	1	main
49 376 484	28 815 600	1	vibrato_c
11 701 161	11 701 161	204 350	sinf
5 188 331	1 512	25	sf_write_float
5 185 468	1 928	25	0x000000000002c200
5 180 796	4 383 314	49	0x000000000002ded0
2 802 179	1 439	26	sf_readf_float
2 793 434	2 790 673	25	0x000000000002f0d0
796 716	796 716	398 358	lrintf
449 603	449 603	3	clean_buffer_c
408 702	408 702	204 351	__floor_sse41

Figura 12: Efecto Vibrato en C, argumentos 0.001 y 4.3

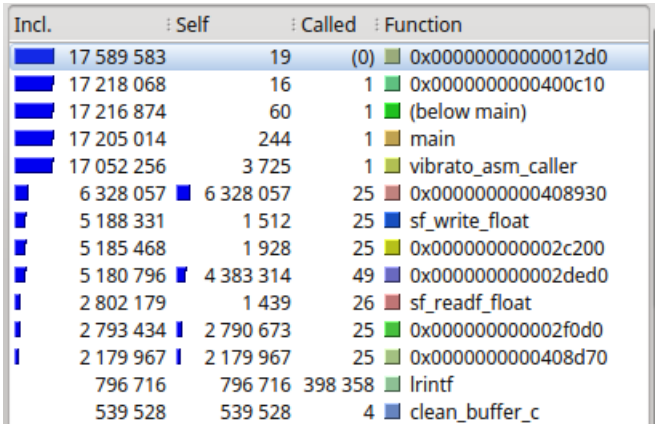


51 268 470	19	(0)	0x00000000000012d0
50 896 955	16	1	0x0000000000400c10
50 895 761	60	1	(below main)
50 883 901	244	1	main
50 731 199	12 602 768	1	vibrato_asm_caller
23 259 869	23 259 869	49 807	sin_ps
6 328 055	6 328 055	25	0x0000000000408a30
5 188 331	1 512	25	sf_write_float
5 185 468	1 928	25	0x000000000002c200
5 180 796	4 383 314	49	0x000000000002ded0
2 802 179	1 439	26	sf_readf_float
2 793 434	2 790 673	25	0x000000000002f0d0
796 716	796 716	398 358	lrintf
539 528	539 528	4	clean_buffer_c

Figura 13: Efecto Vibrato en ASM, argumentos 0.001 y 4.3

En las figuras anteriores, la primera columna indica el costo de la función que fue llamada (tercera columna) sumado al costo de sus hijos, la segunda columna únicamente el costo de esa función (sin el de los hijos), y la tercera columna cuántas veces fue llamada la función. Se puede observar que **C** y **Assembler** poseen un costo similar en total (cerca de los 50.000.000), y que este último tiene un costo bastante alto para la operación correspondiente al seno calculado con la librería **ssemath** (*sin_ps*).

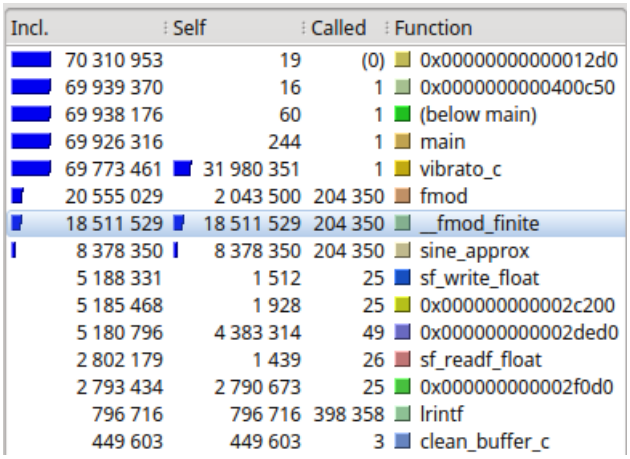
Utilizando la rutina alternativa para el seno (2.2.2) en **ASM**, el resultado es el siguiente:



Incl.	Self	Called	Function
17 589 583	19	(0)	0x00000000000012d0
17 218 068	16	1	0x0000000000400c10
17 216 874	60	1	(below main)
17 205 014	244	1	main
17 052 256	3 725	1	vibrato_asm_caller
6 328 057	6 328 057	25	0x0000000000408930
5 188 331	1 512	25	sf_write_float
5 185 468	1 928	25	0x000000000002c200
5 180 796	4 383 314	49	0x000000000002ded0
2 802 179	1 439	26	sf_readf_float
2 793 434	2 790 673	25	0x000000000002f0d0
2 179 967	2 179 967	25	0x0000000000408d70
796 716	796 716	398 358	lrintf
539 528	539 528	4	clean_buffer_c

Figura 14: Efecto Vibrato (seno aproximado) en ASM, argumentos 0.001 y 4.3

Se puede ver que se logró el efecto buscado: se redujo el costo de aplicar el efecto en **ASM** muy por debajo de su contraparte en **C**. Si bien resulta injusto comparar un seno aproximado con el provisto por la librería **math.h**, utilizar la rutina aproximada en **C** no da buenos resultados, siendo conveniente en este lenguaje utilizar *sinf* (de **math.h**):



Incl.	Self	Called	Function
70 310 953	19	(0)	0x00000000000012d0
69 939 370	16	1	0x0000000000400c50
69 938 176	60	1	(below main)
69 926 316	244	1	main
69 773 461	31 980 351	1	vibrato_c
20 555 029	2 043 500	204 350	fmod
18 511 529	18 511 529	204 350	__fmod_finite
8 378 350	8 378 350	204 350	sine_approx
5 188 331	1 512	25	sf_write_float
5 185 468	1 928	25	0x000000000002c200
5 180 796	4 383 314	49	0x000000000002ded0
2 802 179	1 439	26	sf_readf_float
2 793 434	2 790 673	25	0x000000000002f0d0
796 716	796 716	398 358	lrintf
449 603	449 603	3	clean_buffer_c

Figura 15: Efecto Vibrato (seno aproximado) en C, argumentos 0.001 y 4.3

El código para alternar en **Assembler** entre la librería (**ssemath**) o el seno aproximada, sigue disponible en el archivo **effects_asm.c**, donde hay que comentar las lineas de la que se esté usando actualmente (p.ej., *flanger_index_calc*, y descomentar el ciclo que se encuentra justo en las líneas anteriores.

2.13. Restas versus división

Uno de los requerimientos de la rutina del seno aproximado es que el argumento de la función se debe encontrar entre $(-\pi, \pi)$. Como se puede ver, eso lleva a que en la figura 15 tenga un alto costo lo concerniente a la operación módulo.

En un principio, para el código en **ASM**, se llevaban los argumentos del seno al rango mencionado mediante restas. Sin embargo, como los argumentos podían estar bastante lejos del intervalo $(-\pi, \pi)$ (pues dependen del índice de la muestra, que puede ser un número muy grande aún en archivos de algunos pocos segundos), mediante el uso del profiler se vio que el número de saltos que se hacían en el siguiente pedazo de código perjudicaba bastante el rendimiento:

```
arg_to_interval:
        movaps cmpflag, pi
          ; cmpflag = |pi|pi|pi|pi|
        cmpps cmpflag, sine_args, 0x01
          ; cmpflag = pi < sine_args
        ptest cmpflag, cmpflag
        jz calc_sine      ; 0 => no hay ningún argumento mayor que pi

        movaps tmp, two_pi
        andps tmp, cmpflag
          ; me quedo con 2*pi en los lugares donde pi < sine_args
        subps sine_args, tmp
          ; resto -2*pi en los lugares que corresponden

        jmp arg_to_interval
```

El cálculo del módulo mediante restas se debía a la aversión a utilizar la división, por ser muy costosa generalmente. Sin embargo, en este caso se terminó optando por dividir, quedarse con la parte entera de la división, multiplicar por el divisor, y restar este resultado al valor original, para realizar la operación de módulo. Esta solución probó ser mucho más rápida que la que utilizaba restas, al no tener tantos saltos como el código anteriormente citado.

2.14. Normalización

En la aplicación del efecto WahWah (Subsección 2.9), era posible que la señal húmeda final se encontrara saturada por las operaciones realizadas. Fue necesario entonces realizar la normalización del archivo, comprometiendo un poco la *fidelidad* del efecto, pero entre varias alternativas consideradas fue la más apropiada.

Una opción era realizar una normalización por completo del canal *húmedo* del archivo, luego de la aplicación del efecto. Esta solución no era viable, puesto que el canal ya se encontraba saturado en algunas secciones, y el ruido generado por ese defecto se seguía sintiendo aún después de la normalización.

Otra opción considerada fue realizar normalizaciones "parciales" por cada ciclo de lectura del archivo original. Es decir, leer una sección del archivo de audio original, aplicar el efecto, y normalizar únicamente esta porción de la señal. Debido a la gran diferencia entre una sección de la señal y otras, esto generaba una gran deformación de la señal húmeda, ya que existía un máximo absoluto en cada ciclo, algo que por las características en general de una señal de audio no suele suceder.

Finalmente, se terminó multiplicando la señal húmeda final por un modificador arbitrario, *0.1*, para disminuir la amplitud de la señal, y luego normalizar el archivo entero. Si bien no es lo más óptimo en cuanto a rendimiento (se recorre el archivo entero tres veces: una para aplicar el fecto, otra para encontrar el máximo valor de la señal húmeda, y una tercera para normalizar), es la solución que terminó otorgando la mejor calidad de sonido posible, sin distorsiones auditivas.

3. Resultados

Para generar un conjunto de archivos de salida que pudieran ser usados para la presente sección, se utilizó el archivo **generatorOutputExamples.py**, que para 4 de los archivos de audio de entrada en *inputExamples/* (*guitar.wav, gibson.wav, beirut.wav, DiMarzio.wav*) aplica todos los efectos (generando un archivo de salida por cada uno) con valores al azar en los argumentos (dentro de los rangos determinados). Este archivo genera la lista de comandos (que se halla en **generatorOutputExamples.sh**), y que es el que finalmente se usó para correr el programa y generar los siguientes resultados.

Cada efecto se ejecutó 100 veces, para que la librería *tiempo.h* pudiera tener una cantidad aceptable de iteraciones sobre la cual evaluar la cantidad de ciclos de procesador utilizados en cada algoritmo. Se incluyen en cada sección los comandos utilizados para generar esos resultados (aunque se omitieron los directorios que formaban parte del path absoluto de cada uno).

En las mediciones con Callgrind, una única iteración del algoritmo era suficiente (pues la cantidad de instrucciones llamadas en cada iteración es siempre la misma). Los archivos de salida de **callgrind** (para abrirlos con **KCacheGrind**) se encuentran en el directorio *callgrind/*, y fueron generados por el mismo script **generatorOutputExamples.py**, con el código que se encuentra comentado donde corresponde. Los valores fueron sacados del costo inclusivo en el llamado al efecto (por ejemplo, *delay_simple_c* o *delay_simple_asm_caller*), para obviar todo el overhead del archivo **main.c**.

3.1. Delay

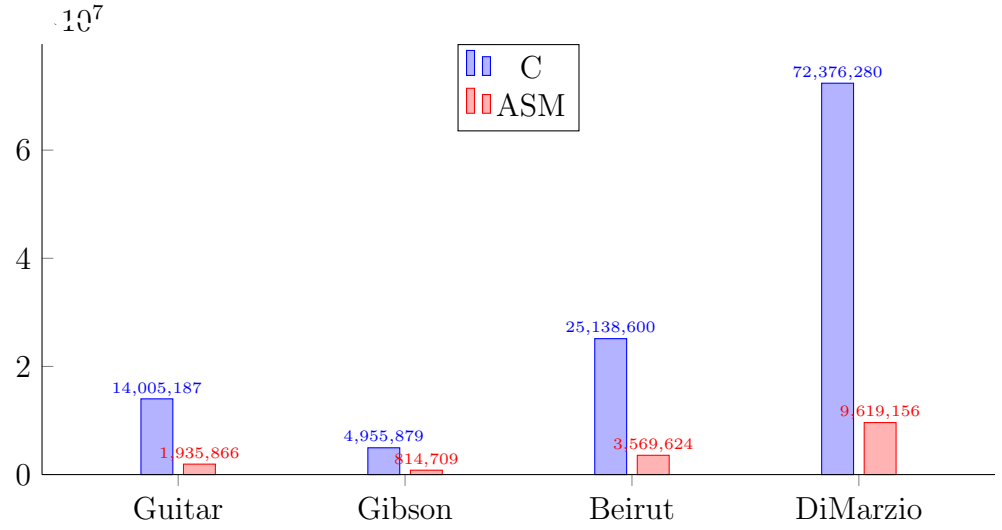


Figura 16: Delay Libreria Tiempo.h

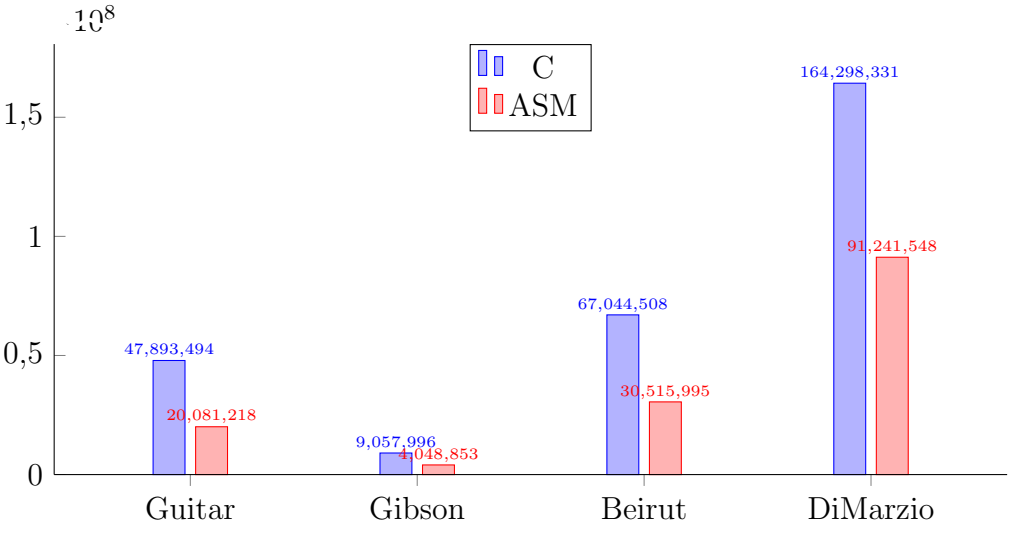


Figura 17: Delay Callgrind

3.2. Flanger

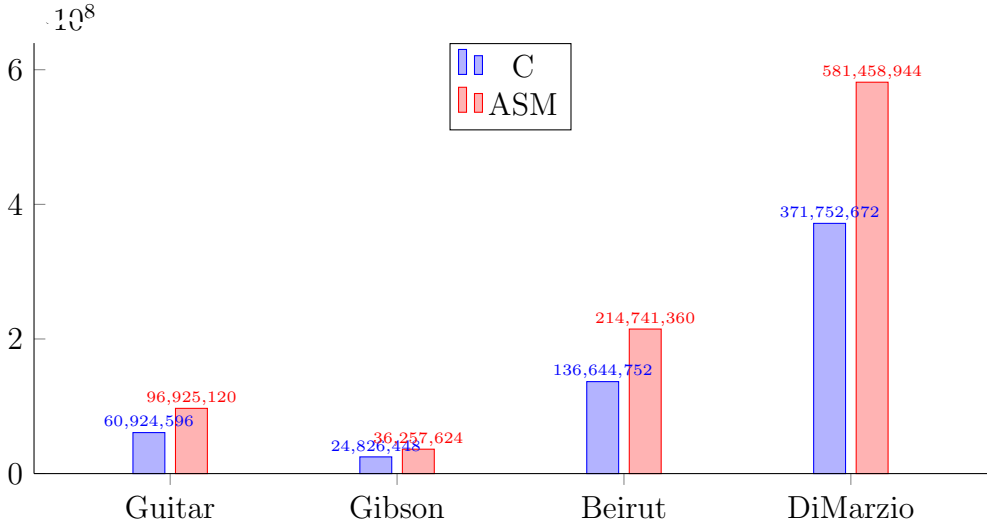


Figura 18: Flanger Libreria Tiempo.h

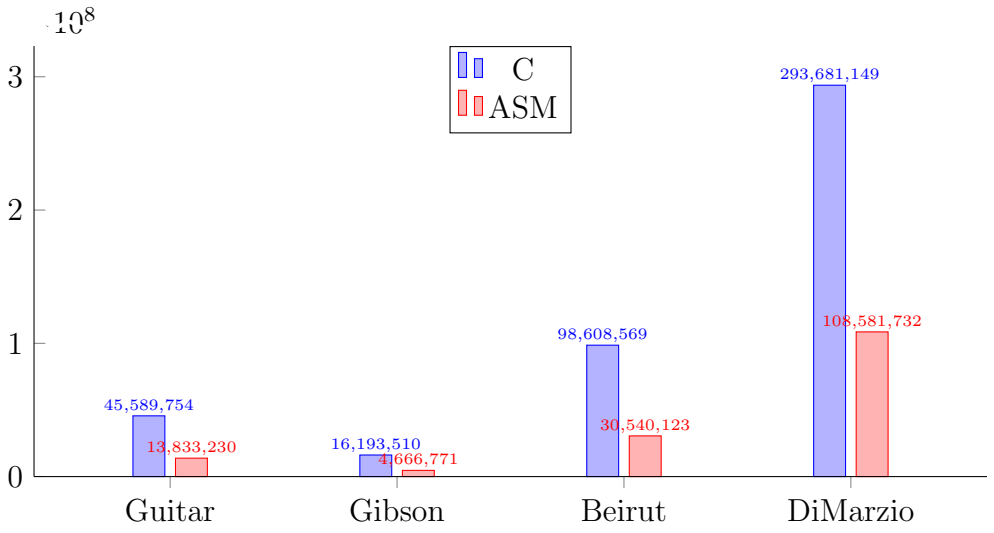


Figura 19: Flanger Callgrind

3.3. Vibrato

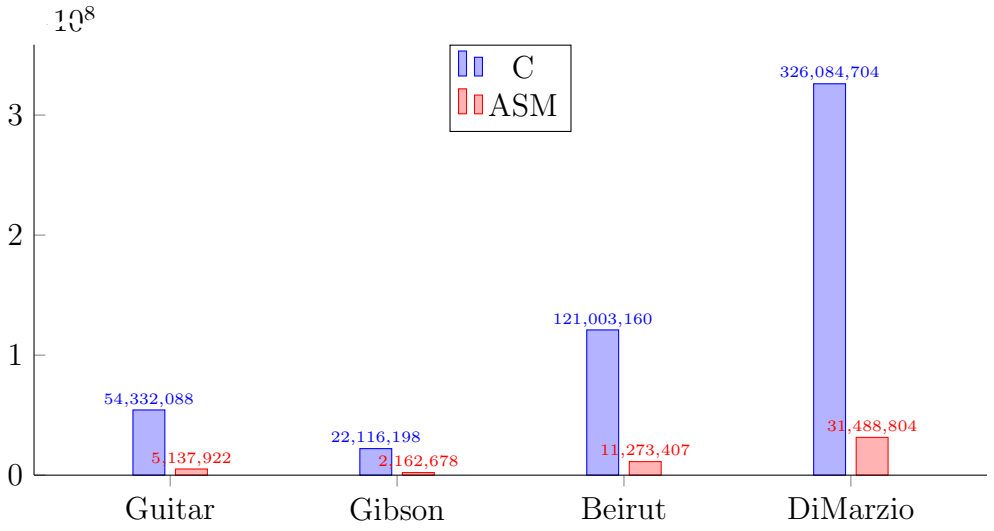


Figura 20: Vibrato Libreria Tiempo.h

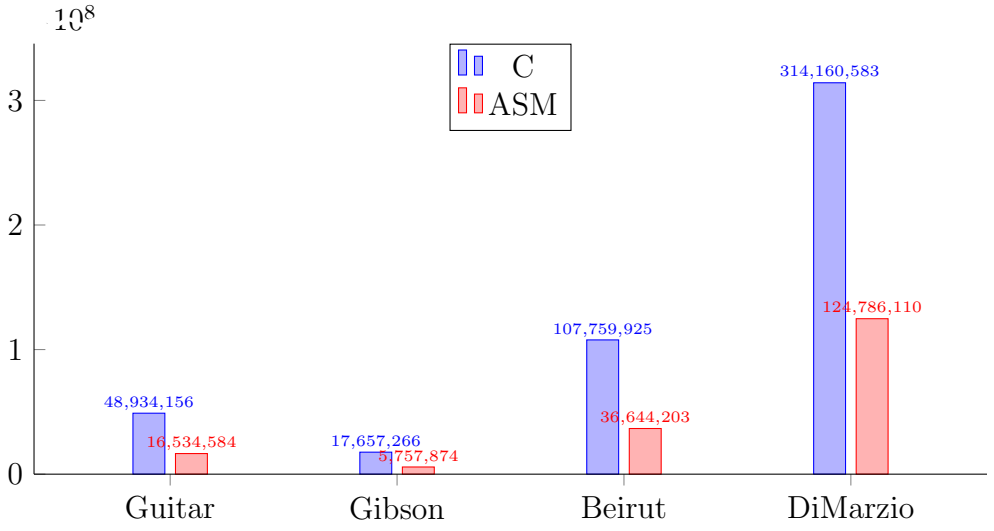


Figura 21: Vibrato Callgrind

3.4. Bitcrusher

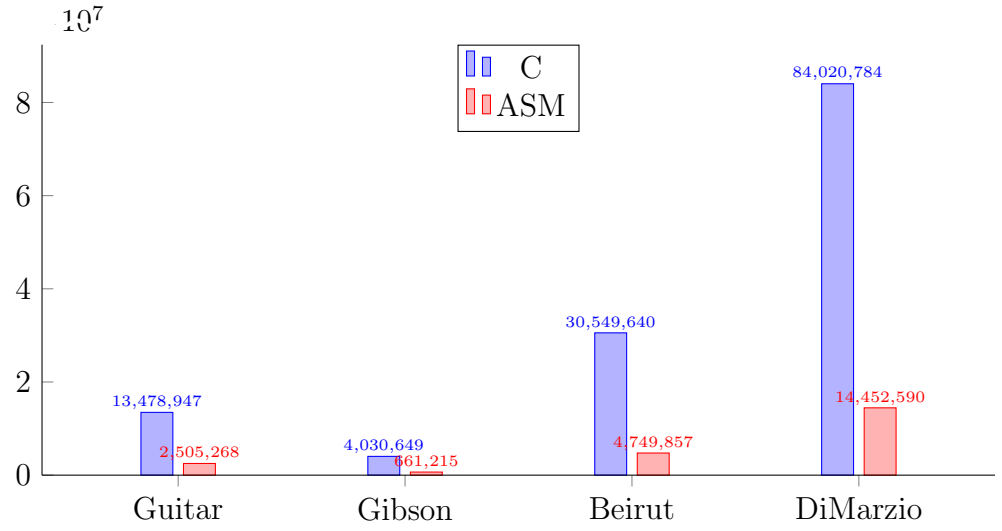


Figura 22: Bitcrusher Libreria Tiempo.h

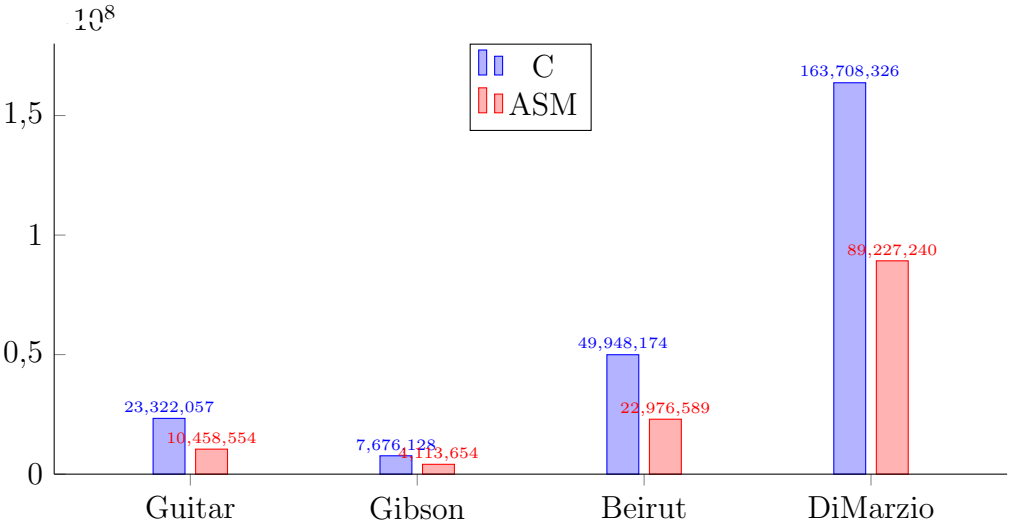


Figura 23: Bitcrusher Callgrind

3.5. WahWah

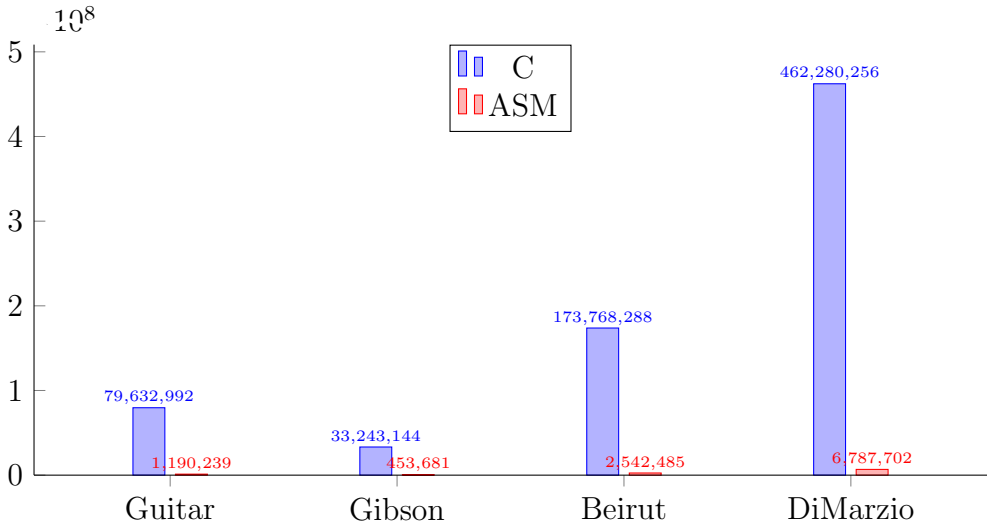


Figura 24: WahWah Libreria Tiempo.h

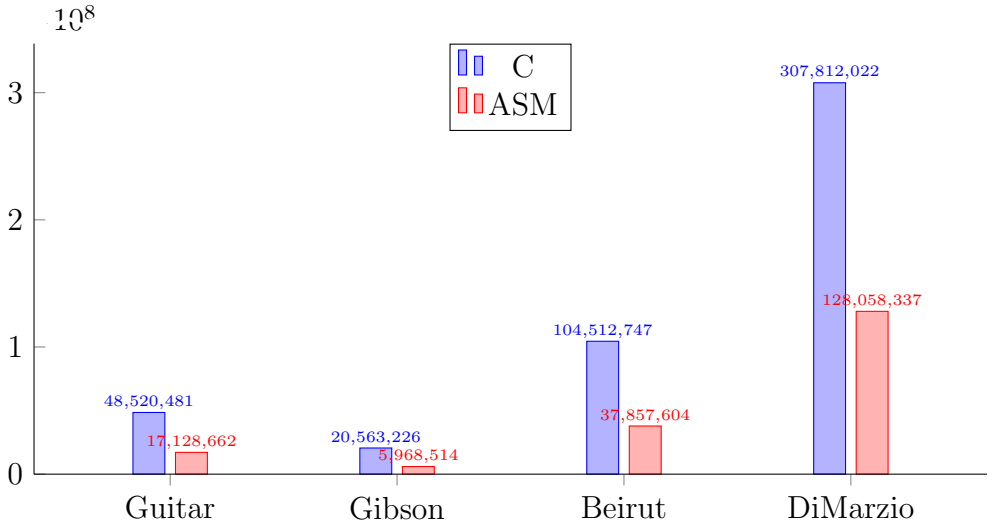


Figura 25: WahWah Callgrind

4. Análisis y Conclusiones

Aclaración sobre las mediciones

Hay que aclarar que ninguna de las dos mediciones, *tiempo.h* ni *callgrind*, son un reflejo al 100 % de la diferencia de rendimiento entre ambas versiones de un mismo efecto. Por un lado, *tiempo.h* está sujeto a las vicisitudes del scheduling del OS en que se corra; si bien se trató de minimizar este efecto cerrando todos los programas en la computadora donde se estaban realizando las mediciones, además de corriendo los algoritmos una gran cantidad de veces (100 iteraciones), sigue sin ser una medición perfecta. Por otro lado, el costo total de instrucciones con *callgrind* incluye también los llamados a la API libsndfile, donde en algunos casos (viendo los archivos en *callgrind/*) se puede ver que son gran parte del aporte total, por lo que el rendimiento no es realmente "4 veces mejor" en **Assembler** que en **C**. Ejemplificaremos esto con las figuras 26 y 27.

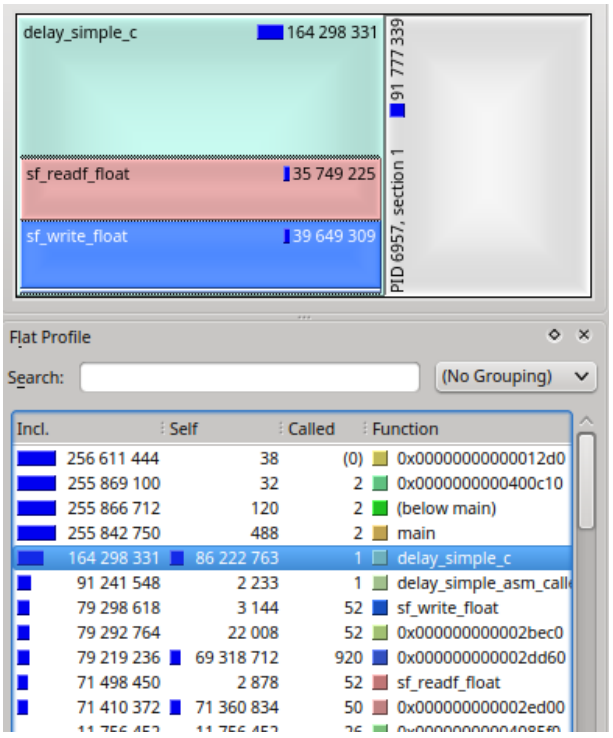


Figura 26: Delay libsndfile Overhead C

En la figura 26, la columna de la izquierda, con 3 filas, describe lo que pasa en el llamado a la función *delay_simple_c*. Su costo total (164.000.000), está compuesto en un poco menos del 50 % (35,000,000 + 40,000,000 = 75,000,000) por los llamados a lectura y escritura de archivos.

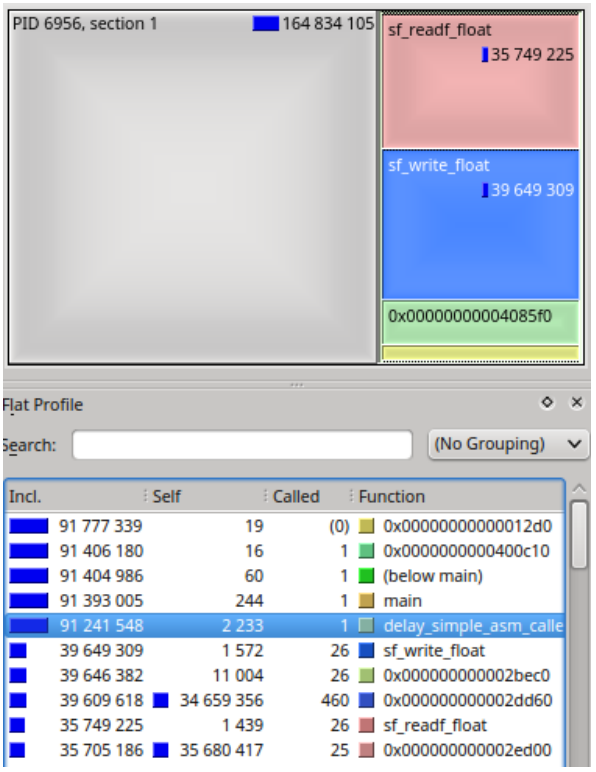


Figura 27: Delay libsndfile Overhead ASM

En la figura 27, la columna de la derecha describe la función *delay_simple_asm_caller*; en este caso, la fila verde representa el código en el archivo **delay.asm**, y es notorio que su costo es muchísimo menor que los llamados a las funciones de la librería **libsndfile**, que son los que engrosan la cantidad total de instrucciones.

4.1. Análisis

En casi todos los gráficos presentados en la sección Resultados 3) se puede observar que la diferencia de rendimiento de haber aplicado los efectos en **ASM** frente a **C** es notoria, siendo más rápido Assembler, obteniéndose entonces el resultado esperado. La proporción de la variación varía según se vean los resultados de la librería **tiempo.h**, o los de **callgrind**. Dado que se está trabajando de a 4 muestras a la vez, se esperaría un rendimiento como mucho hasta 4 veces mejor. La librería **tiempo.h** excede esta mejoría, por motivos que desconozco, mientras que el uso de **callgrind** representa casi siempre una mejoría de entre 2.5 a 3 veces, más acorde al rendimiento esperado; sin embargo, hay que tener en cuenta que hay una cantidad fija de instrucciones que se encuentran en C y ASM, debido a la utilización de libsndfile, además de, potencialmente, otras.

El único resultado (y la razón por la que se puso *casi* al inicio del párrafo anterior) que llama la atención es el de la Flanger Libreria Tiempo.h (Figura 18), donde **ASM** se muestra considerablemente peor que **C**; sin embargo, en la medición Flanger Callgrind (Figura 19), más confiable, se muestra que efectivamente es menos "costosa" la implementación en **ASM**.

Cabe destacar que en varios de los efectos (Vibrato, WahWah por ejemplo) se requiere en un ciclo N acceder a muestras no contiguas del ciclo N-1, lo que imposibilita en gran medida una paralelización completa, ya que el acceso a esos índices específicos se hace por separado, aumentando considerablemente la cantidad de accesos a memoria, lo que ocasiona cierta pérdida de rendimiento (aunque por suerte, no lo suficiente como para haber ocasionado problemas, por lo que se vio en los resultados).

4.2. Conclusiones

El TP probó ser bastante largo, por el proceso de desarrollo en sí, explicado en Proceso de desarrollo del TP (Subsección 1.1). El proceso iterativo de encontrar un algoritmo inteligible que provocara un efecto convincente y adecuado (que no es fácil, dado que muchas soluciones son privadas), aplicar el primer acercamiento en **MRS**, bajarlo de nivel a C y hacer que funcione leyendo un archivo de a partes (lo que llevó a pensar algunas soluciones específicas, como el caso del buffer circular en Vibrato), para finalmente ver cómo paralelizarlo en Assembler, es bastante largo.

A la vez, hubo muchas versiones de cada uno de los efectos, basadas en las mejoras que se iban haciendo en unos y que hacían dar cuenta que era posible mejorar los previamente hechos. Por ejemplo, en un principio en el código en **assembler** había dos ciclos completamente distintos para manejar archivos de entrada **stereo** y **mono** por el otro lado (recordemos que en el primer caso, es necesario "convertirlo a mono" para poder tener un canal libre en la salida y guardar la señal húmeda allí). Durante el desarrollo de un efecto posterior, se pensó en una solución más fácil: sólo separar el manejo de llevar la entrada desde memoria a un registro, y a partir de ahí las operaciones iban a ser como si se estuviera trabajando sobre una señal mono. Esta mejora se hizo retroactiva a los primeros efectos diseñados, lo que agregó bastante tiempo al desarrollo, pero redujo y simplificó el código de manera notables. Otra mejora fue la estandarización de los nombres de los registros en los distintos efectos.

Por otro lado, programar en **assembler** no es para nada fácil. Hubo bastantes problemas con casos bordes (cuando no se podían procesar 4 frames en simultáneo, por ejemplo), cosas que pasaban en archivos **stereo** y no en **mono** (sobre todo cuando los ciclos estaban separados), y el debugging para saber cómo se estaban realizando las operaciones y si coincidían los resultados de cada muestra de la señal entre **ASM** y **C** se hizo bastante engorroso (por lo insatisfactorio que resultaba el debugging con **ddd** (explicado en 1.3.5), y hasta que se encontró **kdbg** pasó un tiempo).

A pesar de estas "contras", que hicieron el trabajo muchísimo más extenso de lo que se pensaba (y que en varios casos hizo replantear si no era conveniente dar el final), uno está satisfecho con los resultados obtenidos. Si bien en algunos casos fue desesperanzante ver que un algoritmo en **ASM** era más lento que en **C**, con la aplicación del profiler (conocimiento adquirido durante el **TP**) se pudo descubrir exactamente dónde estaba el problema, y luego pensar cómo mejorarlo (2.12, 2.13). Los efectos son algunos de los que se pueden encontrar en casi cualquier pedalera o aplicación de audio, y el resultado obtenido con los mismos es fidedigno al uso de otras alternativas, por lo que se logró auditivamente lo que se quería.

Los conocimientos adquiridos (sobre ASM, profiling, debugging, audio y hasta temas de GUI) fueron varios (lo que llevaba a una constante revisión de lo ya hecho), y es un orgullo haber podido juntar dos intereses propios (programación, y música) en un mismo trabajo, lujo que uno no siempre puede darse en la carrera, o en el ámbito laboral.

Como puntos a mejorar, personalmente me hubiera gustado poder hacer algunos más efectos, pero encontrar cómo se consigue ese resultado auditivo no me resultó particularmente fácil (muchas veces se encuentran graficados con circuitos electrónicos, lo cual escapa a mi entendimiento y hubiera sido un camino muy a la deriva para el contexto del **TP**). Por otro lado, me gustaría probar eventualmente (ya que me gustaría seguir con algunas cosas de esto como hobbie, y no apremiado por los tiempos como sucedió en este caso) la inclusión de un ARDUINO (con potenciómetros para el manejo de los argumentos, como muestra la **GUI** desarrollada) para ver cómo se comportan los algoritmos en tiempo real (algo que, nuevamente, el apuro por cerrar el **TP** imposibilitó saciar esa curiosidad).

5. Bibliografía

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] Curtis Roads, *The Computer Music Tutorial*, 1996, The MIT Press, Massachussets (USA)

[4] 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 1: `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 acá.
- Algoritmo Vibrato: adaptación de acá.
- Algoritmo Wahwah: adaptación de acá.
- Algoritmo Bitcrusher: adaptación de MusicDSP