LTFAT Y OCTAVE: IMPLEMENTACIÓN DE SISTEMAS DE PROCESAMIENTO DE AUDIO EN TIEMPO REAL

LTFAT Y OCTAVE: IMPLEMENTATION OF REAL TIME AUDIO PROCESSING SYSTEMS

Javier Alducin Castillo
Universidad Autónoma Metropolitana, México
jac@azc.uam.mx

Recepción: 3/noviembre/2021 **Aceptación:** 20/diciembre/2021

Resumen

El procesamiento digital de señales brinda a los ingenieros las herramientas necesarias para acondicionar de forma adecuada las señales de información. Sin embargo, cuando un estudiante de ingeniería se adentra al procesamiento de señales, suele ver los temas de forma intangible y piensa en ellos de manera abstracta. Esto dificulta el entendimiento y comprensión de cómo un sistema digital puede ser utilizado para resolver problemas en ingeniería. Por ello, en este trabajo se presenta una forma de implementar un sistema de procesamiento digital de señales en tiempo real, usando conocimientos básicos de programación y una herramienta de software libre. Así, un alumno de ingeniería podrá desarrollar sistemas básicos de procesamiento con un resultado inmediato, mejorando el entendimiento, y a su vez aplicando los conocimientos adquiridos del procesamiento digital de señales. También podrá modificar de forma intuitiva el funcionamiento del sistema, así como los diversos parámetros, que permitirá adecuar el resultado del proceso.

Palabras Clave: Efectos de sonido, octave, procesamiento de audio, procesamiento digital de señales.

Abstract

Digital signal processing provides engineers with the necessary tools to properly condition information signals. However, when an engineering student gets into signal processing, usually sees the topics in an intangible and abstract way. This makes it

difficult to understand and comprehend how a digital system can be used to solve engineering problems. Therefore, this paper presents a way to implement a real-time digital signal processing system, using basic programming knowledge, and free software. Thus, an engineering student will be able to develop basic processing systems with an immediate result, improving the understanding, and at the same time, applying the acquired knowledge of digital signal processing. The student will be able to intuitively modify the operation of the system, as well as the various parameters, which will allow to modify the result.

Keywords**:** *Audio effects, audio processing, octave, digital signal processing.*

1. Introducción

La introducción al procesamiento digital de señales (PDS), de un alumno de ingeniería, conlleva varias dificultades. Una de ellas es la de converger diversos conocimientos, obtenidos de varios cursos tomados a lo largo de su carrera. Desde matemáticas básicas, teoría de circuitos eléctricos, así como el análisis de señales y sistemas. Es de consenso general que muchos de esos conocimientos adquiridos, son aplicados debido al progreso de la tecnología, y por ello hay una alta demanda de ingenieros, con conocimientos sólidos del PDS, en industrias como la automotriz, comunicaciones digitales, sistemas de control, imagen y video, así como dispositivos médicos. Otra dificultad se presenta en las habilidades y capacidades desarrolladas de programación. Si bien es importante y necesario que los alumnos de ingeniería conozcan diversos lenguajes de programación, así como la implementación de estructuras básicas de programación, para la solución de problemas, es cada vez más frecuente que el alumno tenga dificultades en desarrollar un código para resolver un problema en ingeniería [Fuentes, 2017].

En un curso de PDS, es importante la parte práctica, para que el alumno pueda saber cómo y dónde se puede usar ese conocimiento adquirido, en problemas y situaciones reales. Los estudiantes suelen reforzar su aprendizaje en actividades prácticas [Rickel, 1989]. Por esta razón es importante que el alumno pueda analizar y diseñar sistemas con los que pueda interactuar, modificar, ejecutar y obtener un resultado tangible.

La implementación de sistemas de PDS requiere el dominio no solo de conocimientos teóricos y prácticos, sino también de sistemas de hardware y software, donde se ejecutarán los algoritmos necesarios para la implementación de dichos sistemas. Para que el alumno pueda desarrollar sus primeros sistemas de procesamiento de señales, generalmente se eligen kits de desarrollo o tarjetas de desarrollo que incluye un procesador de señales digitales (DSP) como el usado en [Gómez, 2017]; [Dogan, 2012] o FPGA's [Pfaff, 2007]. Sin embargo, se requiere de una curva de aprendizaje amplia, para poder entender el funcionamiento del DSP; los recursos disponibles, la sintaxis de las instrucciones, así como de la configuración del DSP adecuada para el desarrollo de algún sistema. Además, que el costo de un kit de desarrollo no lo puede costear un alumno por sí solo.

Una alternativa es la combinación de ambientes de desarrollo y DSP's [Welch, 2016]. Primero se diseña y analiza el sistema deseado para, posteriormente exportarlo al DSP. Sin embargo, aparecen los problemas planteados previamente. Otra alternativa es el uso de ambientes de desarrollo como Matlab u Octave [Weeks, 2011]; [Stearns, 2002]; [Downey, 2016]. Aunque estos últimos son ampliamente utilizados para ayudar al estudiante a entender los conceptos básicos y realizar ejemplos específicos, pueden no tener una conexión, por sí solos, con sistemas del mundo real o sistemas que les interesen y que puedan mostrarles, a los estudiantes, un vínculo entre los conocimientos teóricos y prácticos.

Así pues, es necesario que un alumno de ingeniería domine no solo conceptos teóricos, sino también prácticos. Además, se requiere de habilidades de programación y análisis de sistemas. Todo lo anterior se puede solventar usando un ambiente de desarrollo, sin embargo, es importante que se puedan diseñar sistemas con los que el alumno pueda interactuar, simularlos y ejecutarlos, de esta forma podrá poner en práctica el procesamiento digital de señales.

LTFAT (The Large Time-Frequency Analysis Toolbox) [Søndergaard, 2012a]; [Průša, 2014] es un *toolbox* para Octave o Matlab, donde es posible realizar un análisis tiempo-frecuencia de una señal. Este toolbox, ha sido utilizado en diversos trabajos para obtener la transformada de Gabor de forma eficiente [Moreno, 2010]; [Søndergaard, 2012b]; [Průša, 2020], para análisis de Fourier usando la

transformada corta de Fourier [Hendry, 2021] así como la mejora de audio a partir de un banco de filtros y redes neuronales [Takeuchi, 2019]. Sin embargo, provee también de herramientas dedicadas para el desarrollo y ejecución de sistemas en tiempo real a través de bloques de procesamiento. Estos bloques se pueden implementar tanto en sistemas operativos Windows como en cualquier distribución Linux, lo que permite, con conocimientos básicos de programación, poder implementar sistemas de procesamiento digital de señales.

En este trabajo, se presenta una forma de implementar sistemas de procesamiento digital de señales en tiempo real, aplicado a señales de audio, usando conocimientos básicos de programación, herramientas de software libre (Octave y LTFAT), en un sistema operativo Linux.

Si bien, es posible usar el *toolbox* en Matlab, el estudiante de ingeniería deberá tener una licencia válida para su uso. Si la institución educativa no cuenta con una licencia institucional, el alumno difícilmente podrá adquirir una licencia de Matlab. Por ello, uno de los objetivos de este trabajo es mostrar cómo se puede realizar la implementación de un sistema de procesamiento de audio, en tiempo real usando software libre.

Es importante aclarar que, la distribución Linux a la que se hará referencia en el presente trabajo es Debian 10 (Buster en su rama estable), sin embargo, cualquier otra distribución Linux podrá usarse, siempre y cuando se tengan los paquetes y dependencias necesarias instaladas.

2. Métodos

El *toolbox* LTFAT hace uso *frames* para permitir y brindar un enfoque flexible, y de esta forma es posible representar una gran variedad de conceptos matemáticos a utilizar. Es ampliamente configurable a partir de diversos parámetros, lo que permite implementar algoritmos de procesamiento de señales de una forma eficiente y en tiempo real [Søndergaard, 2012a]; [Průša, 2014]. Esta última característica, si bien no se ha utilizado ampliamente en la literatura, es muy importante para el propósito de este trabajo.

Instalación y configuración

El *toolbox* está disponible en https://ltfat.github.io y contiene la documentación necesaria para poder introducirse a dicha herramienta; los conceptos, así como la notación básica para poder utilizar LTFAT. Esta notación es de amplio uso en el procesamiento digital de señales. En la documentación de LTFAT, se puede encontrar una descripción amplia y completa al respecto. Sin embargo, es necesario, para poder usar correctamente LTFAT en Octave, seguir los pasos de instalación que se enlistan a continuación:

- Instalar las dependencias *liboctave-dev* y *portaudio19-dev*
  sudo apt install liboctave-dev portaudio19-dev
- Descargar *Portaudio* (http://files.portaudio.com)
- Descargar *Playrec* (http://www.playrec.co.uk)
- Compilar *Playrec* en Octave, habilitando *ALSA* (componente del núcleo de Linux para controlar la tarjeta de sonido)
- Instalar LTFAT en Octave
  pkg install ltfat-forge

Una vez instaladas las dependencias, así como el toolbox, es importante hacer notar que LTFAT hace uso de la programación orientada a objetos. Utiliza clases para poder desarrollo diversos bloques de procesamiento. Una clase es entendida como una colección de métodos y variables para crear objetos. Así una clase, puede derivar en otras clases, que heredan las propiedades de la clase principal. Los objetos que se pueden crear son conocidos como *frames*, y requieren parámetros necesarios para su declaración.

Framework

LTFAT ofrece un marco de trabajo unificado para poder trabajar con audio. Dado que el procesamiento de audio requiere diversos pasos para llevarse a cabo: adquirir la señal de información, procesar dicha señal de audio a través de un sistema de procesamiento, cambiando parámetros durante la adquisición, y por último la reproducción de la señal. Para ello, debe elegir un dispositivo o audio de

entrada. Se registran trozos o segmentos de datos de dicha entrada, con el objetivo de mantener un retardo de procesamiento bajo [Průša, 2014]. Estos segmentos de audio se logran capturar en tiempo real, sin la necesidad de utilizar alguna herramienta adicional o algún módulo que implique un retardo en la adquisición de la información. Una vez adquirido cada segmento de señal, se procesará de acuerdo con el sistema diseñado. Los pasos del procedimiento descrito previamente se pueden observar en la figura 1.

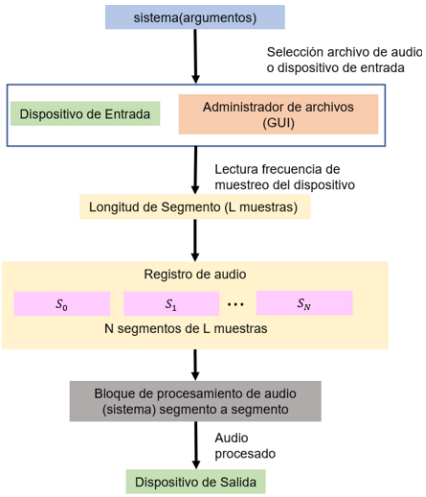

Figura 1 Diagrama a bloques de los pasos necesarios para un proceso en LTFAT.

El procedimiento descrito en la figura 1, se ejecuta a través de una función en Octave, para ello se propone utilizar como base el código 1 mostrado en figura 2.

```
1.  function nombre_funcion(source,varargin)
2.  %bloque interactivo (ganancia)
3.  try
4.      fs = block(source,varargin{:},'loadind',p);
5.  catch
6.      blockdone(p);
7.      err = lasterror;
8.      error (err.message);
9.  end
10. L = floor(time_seg*fs); %longitud de segmento de audio
11. %inicialización de variables (sistema de procesamiento)
12. while flag && p.flag
13.    [f,flag] = blockread(L);
14.    % instrucciones de sistema de procesamiento
15.    blockplay(h);
16. endwhile
17. blockdone(p);
18. endfunction
```

Figura 2 Código base para sistemas en tiempo real usando LTFAT y Octave.

Configuración de dispositivos

Como cualquier función en Octave, el código base es una función que requiere definir argumentos *source* y *varargin*. El argumento *source* indica la fuente de audio que se utilizará. Se puede utilizar cualquiera de estas opciones:

- Archivo de audio en formato .wav: 'audio.wav'. Permite indicar la dirección o ubicación de un archivo de audio .wav, para que sea la entrada del bloque de procesamiento.
- Elegir un archivo de audio con GUI: 'dialog'. Generará una ventana (GUI) para elegir un archivo de audio .wav, usando el administrador de archivos. Este audio será la entrada del bloque de procesamiento, figura 3.
- Elegir la entrada de audio del sistema y procesarlo: 'playrec'. El audio que se registre o reproduzca de una fuente elegida del sistema será la entrada del bloque de procesamiento, figura 4.

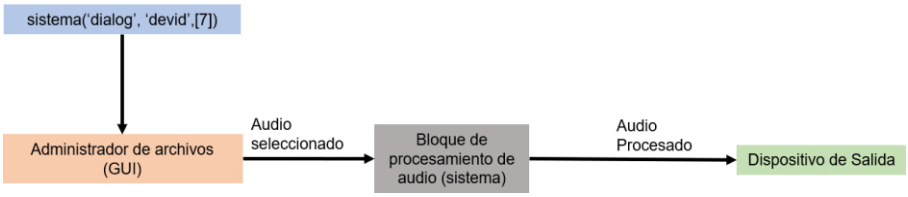

Figura 3 Diagrama a bloques del uso de la opción 'dialog'.

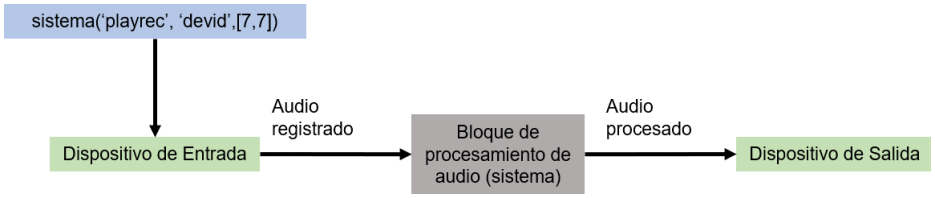

Figura 4 Diagrama a bloques del uso de la opción 'playrec'.

Para las opciones de los puntos 2 y 3 figuras 3 y 4, respectivamente, será necesario conocer cuáles son los dispositivos disponibles del sistema, para que se pueda obtener el audio requerido. Para ello, se puede conocer la lista de dispositivos disponibles del sistema a través de la siguiente instrucción ***blockdevices***.

En la figura 5, se puede observar un ejemplo de una lista de dispositivos disponibles del sistema. Es importante elegir aquella opción donde esté ***pulse***, debido a que es

el sistema de audio, que tiene mayor compatibilidad al usar LTFAT y Octave. Además, que este sistema es el utilizado para compilar ***Playrec*** en la instalación del *toolbox*. En la lista de los dispositivos de entrada y salida, se puede observar la información de cada dispositivo del sistema: latencia, número de canales de audio y la frecuencia de muestreo disponible. Para el ejemplo de la figura 5, es posible elegir una frecuencia de muestreo de hasta 192 kHz para el dispositivo de entrada 7 (input) o 48 kHz para el dispositivo de salida 6 (output).

```
Available output devices:
ID = 1: HDA Intel PCH: HDMI 0 (hw:0,3) (ALSA) 2 chan., latency 5--34 ms, fs [32000, 44100, 48000]
ID = 2: HDA Intel PCH: HDMI 1 (hw:0,7) (ALSA) 8 chan., latency 5--34 ms, fs [32000, 44100, 48000, 88200, 96000, 192000]
ID = 3: HDA Intel PCH: HDMI 2 (hw:0,8) (ALSA) 8 chan., latency 5--34 ms, fs [32000, 44100, 48000, 88200, 96000, 192000]
ID = 4: HDA Intel PCH: HDMI 3 (hw:0,9) (ALSA) 8 chan., latency 5--34 ms, fs [32000, 44100, 48000, 88200, 96000, 192000]
ID = 5: HDA Intel PCH: HDMI 4 (hw:0,10) (ALSA) 8 chan., latency 5--34 ms, fs [32000, 44100, 48000, 88200, 96000, 192000]
ID = 6: hdmi (ALSA) 2 chan., latency 5--34 ms, fs [32000, 44100, 48000]
ID = 7: pulse (ALSA) 32 chan., latency 8--34 ms, fs [8000, 11025, 16000, 22050, 32000, 44100, 48000, 88200, 96000, 192000]
ID = 8: default (ALSA) 32 chan., latency 8--34 ms, fs [8000, 11025, 16000, 22050, 32000, 44100, 48000, 88200, 96000, 192000]

Available input devices:
ID = 0: HDA Intel PCH: ALC3234 Alt Analog (hw:0,2) (ALSA) 2 chan., latency 5--34 ms, fs [44100, 48000, 96000, 192000]
ID = 7: pulse (ALSA) 32 chan., latency 8--34 ms, fs [8000, 11025, 16000, 22050, 32000, 44100, 48000, 88200, 96000, 192000]
ID = 8: default (ALSA) 32 chan., latency 8--34 ms, fs [8000, 11025, 16000, 22050, 32000, 44100, 48000, 88200, 96000, 192000]
```

Figura 5 Lista de dispositivos disponibles del sistema.

Para indicar qué dispositivo o dispositivos de entrada y/o salida serán usados, se debe declarar, por ejemplo, *nombre_funcion('dialog', 'devid', [7])*. En este caso, se desea elegir el archivo de audio, usando el administrador de archivos con una GUI. A continuación, se indica que el resultado del bloque de procesamiento se reproduzca en el dispositivo de salida 7 (pulse), que corresponde al audio interno del equipo correspondiente a las bocinas, figura 3.

Otra configuración es *nombre_funcion('playrec, 'devid', [7,7])*. El argumento '*devid'* indica que se le brindará al bloque los identificadores (*'id'*) de los dispositivos a utilizar. Estos identificadores se pueden obtener usando *blockdevices**,*** como en la figura 4. El primer identificador corresponde al dispositivo de salida y el segundo al dispositivo de entrada. En este ejemplo, se indica que el bloque de procesamiento utilizará el audio adquirido del dispositivo de entrada 7 (micrófono) y el resultado, será reproducido por el dispositivo de salida 7, bocinas internas, figura 4.

Una vez elegido el dispositivo de entrada, se pueden colocar bloques interactivos con los que el usuario podrá modificar algunos parámetros. Como siguiente paso, se lee la frecuencia de muestreo disponible del dispositivo, línea 5 de código 1 (Figura 2). Esta frecuencia de muestreo corresponde únicamente al dispositivo

seleccionado. Por ello, es importante elegir un dispositivo con una frecuencia adecuada para el efecto u audio a utilizar. A continuación, se define la longitud en muestras L de los segmentos en los que se leerá el audio. El parámetro *seg_time*, debe ser un número entero dado en segundos (500 ms, por ejemplo). En este punto, se sugiere realizar la inicialización de variables necesarias para el sistema de procesamiento. Más adelante se mostrarán ejemplos al respecto.

El siguiente paso, consiste en leer los segmentos audio uno a uno, dentro de un bucle, a partir de la línea 12, código 1. Procesar dicho audio, según el sistema diseñado (insertar líneas de sistema entre las líneas 13 y 15), y entregar el resultado para su reproducción (línea 15). Por último, se cierra el bloque de procesamiento, con lo que ha concluido la ejecución del sistema. Como se ha mencionado, el código 1 es la base para poder realizar un sistema de procesamiento de señales de audio. Para ejemplificar el uso del código base, se mostrará la implementación del efecto de un solo eco FIR en LTFAT en tiempo real.

Sistema efecto eco FIR

Se sabe que un filtro de un solo eco FIR (filtro peine FIR) está descrito por la ecuación 1.

$$y[n] = x[n] + \alpha x[n - D] \tag{1}$$

Donde la salida del sistema es el resultado de la suma de la señal de entrada más una versión retrasada de la entrada en D muestras, ponderada por un factor α. El tiempo de retardo τ se obtiene al multiplicar $D * fs$. El diagrama a bloques del filtro de un solo eco o filtro peine FIR puede observarse en la figura 6.

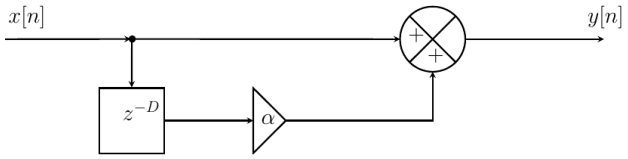

Figura 6 Diagrama a bloques del filtro de un solo eco FIR.

Este sistema se puede ejecutar, aplicado a cualquier señal, usando las funciones que Octave incluye, como lo es *filter*. Sin embargo, esta función requiere tener

almacenada la señal de entrada para poder transfórmala. Dado que el objetivo del trabajo es mostrar el procesamiento en tiempo real, es necesario encontrar la manera de ejecutar el mismo sistema, pero, usando operaciones entre vectores (o matrices) tal como van obteniéndose los datos de la señal de entrada. Supongamos que tenemos en un vector columna los datos que representan a $x[n]$, tabla 1, columna 1. Se quiere obtener una versión retrasada en 3 muestras y ponderada por el factor de 0.4. Basta con recorrer a $x[n]$ en tres filas agregando 0's iniciales (columna 2) y por último obtener $y[n]$ (columna 4). Esta forma básica de obtener un sistema peine FIR es necesaria para poder implementarlo en tiempo real.

Tabla 1 Ejemplo cómo pueden usarse de operaciones de vectores.

x[n]	x[n-3]	0.4*x[n-3]	y =x[n]+ 0.4*x[n-3]
1.1	0	0	1.1
1.2	0	0	1.2
1.3	0	0	1.3
1.4	1.1	0.44	1.84
1.5	1.2	0.48	1.98
1.6	1.3	0.52	2.12
1.7	1.4	0.56	2.26
1.8	1.5	0.6	2.4
1.9	1.6	0.64	2.54
2	1.7	0.68	2.68
2.1	1.8	0.72	2.82
2.2	1.9	0.76	2.96
2.3	2	0.8	3.1
2.4	2.1	0.84	3.24
2.5	2.2	0.88	3.38
2.6	2.3	0.92	3.52
2.7	2.4	0.96	3.66
2.8	2.5	1	3.8
2.9	2.6	1.04	3.94
3	2.7	1.08	4.08
3.1	2.8	1.12	4.22
3.2	2.9	1.16	4.36
3.3	3	1.2	4.5
3.4	3.1	1.24	4.64

La implementación de un filtro de un solo eco FIR se muestra en el código 2, figura 7. Se toma como base el código 1, agregando algunas líneas de configuración y de bloques interactivos. Se ejecutó usando *'dialog'* y el dispositivo de entrada 7, como se muestra en la instrucción *eco_real_time('dialog', 'devid'[7])*.

El bloque definido, en la línea 2, crea una ventana de ganancia (volumen en dB) con el que el usuario podrá interactuar para incrementar el volumen, figura 8.

```
1.  function eco_real_time(source,varargin)
2.  p = blockpanel({{'GdB','Gain',-20,20,0,21}});
3.  try
4.      fs = block(source,varargin{:},'loadind',p);
5.  catch
6.      blockdone(p);
7.      err = lasterror;
8.      error(err.message);
9.  end
10. L = floor(500e-3*fs);
11. flag = 1;
12. g = zeros(L,1);
13. h = [];
14. alpha = 0.6;
15. while flag && p.flag
16.    gain = blockpanelget(p,'GdB');
17.    gain = 10^(gain/20);
18.    [f,flag] = blockread(L);
19.        h =  [alpha*g(:,1)+f(:,1)];
20.        g = f;
21.    blockplay(h*gain);
22. endwhile
23. blockdone(p);
24. endfunction
```

Figura 7 Código implementado para el filtro de un solo eco FIR en Octave y LTFAT.

Figura 8 Ventana interactiva para manipular el volumen del audio ejemplo 1.

Se obtiene la frecuencia de muestreo del dispositivo elegido, salida de audio interna, línea 4. Se elige leer el audio en segmentos de 500 ms, ya que es el tiempo de retraso seleccionado para implementar el eco, es decir, se tendrá $500\,ms * fs$, donde $fs = 44{,}100\,Hz$. Por lo tanto, se tiene una longitud de 22,050 muestras por segmento. Este parámetro se puede modificar según el efecto deseado. En la ecuación 2, se modela el sistema implementado.

$$y[n] = x[n] + 0.6x[n - 22050] \tag{2}$$

Se inicializan dos variables g y h (líneas 12 y 13, código 2). La primera variable almacenará la versión recorrida de la entrada $(x[n - 22050])$. La variable h, almacenará la versión actual de la entrada $x[n]$. Se declara el factor de atenuación del eco en 0.6 (línea 14, código 2). Si el usuario interactúa con la barra de ganancia, se lee el valor actualizado en dB, lo que aumentará el volumen de la reproducción. En la línea 18, se lee el audio en segmentos de L muestras y se guarda en la variable f. Una vez leído el segmento inicial, se suma con una versión anterior (g). Sin embargo, al estar inicializado el sistema, g contiene muestras en ceros. El valor

actual de salida h, se asigna a un bloque de reproducción ponderada por la ganancia, así se podrá escuchar el audio procesado con el filtro eco FIR en tiempo real. El valor del segmento inicial de audio f, se guarda en g. El valor actual f, se ve afectado por g ponderado. De esta forma, se implementa el efecto de sonido usando operaciones entre vectores en tiempo real, sin algún retardo de tiempo significativo. Como se puede observar, la implementación de un filtro de un solo eco FIR es relativamente sencilla. El alumno debe conocer la ecuación de diferencias del sistema. Y, analizar cómo puede realizar dicho sistema a través de operaciones entre vectores. Así, podrá realizar el sistema en tiempo real, haciendo uso de los bloques de procesamiento de LTFAT.

Se realizó un segundo efecto de sonido, para ejemplificar, cómo se puede usar el código base para realizar un sistema con un resultado en tiempo real.

Sistema efecto flanger

El efecto flanger superpone dos audios: un audio original y otro retrasado, ecuación 3. Este último, no tiene un retraso constante. El retraso es variable en el tiempo ($d[n]$), obteniendo una señal variable y periódica. Para dicho retraso puede usarse una señal sinusoidal para representar a ($d[n]$) (ecuación 4), que dependerá de un valor d_0. Este valor, es el tiempo de retraso máximo deseado y f_f será la frecuencia de oscilación. Para este efecto, se sugiere que sea un valor de baja frecuencia, y así obtener un oscilador de baja frecuencia (LFO, por sus siglas en inglés).

$$y[n] = \alpha x[n] + \beta x\big[n - d[n]\big] \tag{3}$$

$$d[n] = d_0 \sin(2\pi f_f nT) \tag{4}$$

En la figura 9, se muestra el diagrama a bloques del sistema discreto que genera el efecto flanger, donde, se puede observar que el efecto de un solo eco FIR, figura 5 y flanger son sistemas parecidos. La diferencia entre ambos sistemas es que, el retraso en muestras es variable para el segundo efecto y para el primer es estático. En la figura 10, se puede observar la forma sinusoidal del retraso variable $d[n]$, donde los retrasos deben tomar muestras enteras.

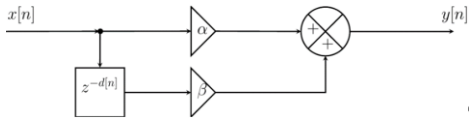

Figura 9 Diagrama a bloques del sistema discreto para producir el efecto *flanger.*

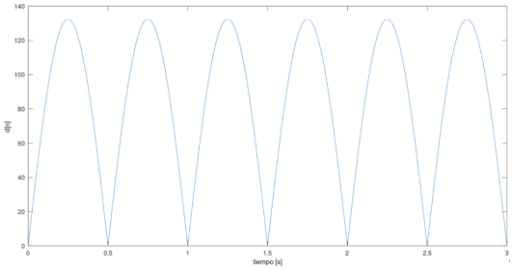

Figura 10 Señal variable y periódica que representa a $d[n]$ para el efecto flanger.

En el código 3 (Figura 11), se muestra cómo procesar un audio con el efecto flanger. Se toma como base el código 1, agregando líneas de código necesarias, para poder implementar el efecto deseado.

```
1.  function flanging_real_time(source,varargin)
2.  p = blockpanel({{'GdB','Gain',-20,20,0,21}});
3.  try
4.      fs = block(source,varargin{:},'loadind',p);
5.  catch
6.      blockdone(p);
7.      err = lasterror;
8.      error(err.message);
9.  end
10. L = floor(25e-3*fs);
11. t = [0:L-1]/fs;
12. inx = [1:L]';
13. flag = 1;
14. Beta = alpha = 0.7;
15. timeOfDelay = 3e-3;
16. dn = abs(round((sin(2*pi*1.*t))*(timeOfDelay *fs)))';
17. x_delay=[];
18. while flag && p.flag
19.    gain = blockpanelget(p,'GdB');
20.    gain = 10^(gain/20);
21.    [f,flag] = blockread(L);
22.        x_delay = f(inx-dn:);
23.        y = (alpha.*f)+(beta.*x_delay);
24.    blockplay(y*gain);
25.
26. end
27. blockdone(p);
```

Figura 11 Código implementado para el efecto flanger usando Octave y LTFAT.

Se crea el bloque de ganancia de audio. Se lee la frecuencia de muestreo del dispositivo fuente. Para este efecto, se elige que el audio se lea en bloques de 25 ms (línea 10). Se crea un vector t (tiempo en segundos) que muestra el tiempo para cada una de las muestras tomadas del audio. Este vector de tiempo será utilizado

para crear el vector de retrasos variable $d[n]$. Se define un valor de atenuación de 0.7 para α y β. Además, se define el tiempo de retardo máximo para el efecto de flanger (d_0) con un valor de 3 ms. Es posible modificar este parámetro en función del audio a procesar. El valor elegido, dio buenos resultados para los diferentes audios utilizados. Usando la ecuación 4, se define $d[n]$. Este vector, contendrá los valores enteros de retraso a utilizar. En la línea 22, se reproduce una variante de la operación entre vectores mostrada en la figura 6. Se crea el vector correspondiente a la operación $x[n - d[n]]$, usando los índices de las filas correspondientes. De esta forma, se obtiene como resultado, una versión retrasada según el comportamiento de $d[n]$, sin la necesidad de algún otro bucle u operaciones adicionales. Se registra la salida $y[n]$ (línea 23), donde, se suma la versión original del audio y la versión retrasada, ambos ponderados por 0.7. Es decir, se tiene la misma atenuación en ambos audios. Esto produce un audio donde la versión retrasada es variable en la frecuencia, creando un efecto oscilatorio.

3. Resultados

El código base mostrado en este trabajo, puede utilizarse para generar diversos efectos de sonido que se puedan ejecutar en tiempo real. Se presentaron dos códigos para ejemplificar el uso del código base propuesto. Ambos efectos mostrados, eco y flanger, se pueden ejecutar en cualquier equipo, aún con bajas características de procesamiento: CPU, RAM, etc.

Es necesario tener instalado Octave y LTFAT correctamente, para poder acceder a la configuración necesaria del sistema, y se puedan utilizar los módulos de audio apropiadamente. Es posible elegir los dispositivos de entrada y salida, con una configuración sencilla, para que la calidad de sonido sea la indicada y se pueda ejecutar el sistema, de acuerdo a la entrada que se desee: elegir un archivo de audio o elegir una combinación de dispositivos de entrada-salida. Para los ejemplos mostrados, se desarrolló solo un bloque interactivo, con el que se podía modificar el parámetro de volumen en la reproducción del audio obtenido.

Se utilizaron 5 audios diferentes para verificar cada uno de los efectos mostrados en el trabajo. En cada uno de ellos es posible distinguir claramente el efecto

realizado, ya sea de eco o flanger. Sin embargo, se debe considerar que cada uno de estos efectos son subjetivos, es decir, depende del gusto auditivo de cada persona.

4. Discusión

La configuración mostrada en este trabajo fue, para un equipo con sistema operativo Debian (Linux). Es posible utilizar cualquier otra distribución Linux, solo es necesario instalar las dependencias necesarias. Si bien, la documentación indica que se puede usar LTFAT en Windows, usando Octave y/o Matlab, para este trabajo, se no se verificó si los bloques de procesamiento de audio en tiempo real funcionarían en el sistema operativo Windows. Por ello será, importante mostrar en un trabajo futuro si se requiere una configuración adicional en Windows para que el código base pueda ser utilizado en Octave y Windows o Matlab y Windows.

Cabe destacar que la frecuencia de muestreo depende únicamente del dispositivo seleccionado, para obtener la señal de entrada. Si bien, se puede utilizar cualquier tarjeta de audio incluida en el equipo de cómputo, se tendrá una mejor calidad de audio con una frecuencia de muestreo de al menos 44.1 kHz. Si el dispositivo seleccionado tiene una frecuencia de muestreo menor, es preferible cambiar de dispositivo a aquel que permita una frecuencia de muestreo adecuada.

El objetivo del trabajo es mostrar y compartir un código base para la implementación de diversos sistemas de procesamiento de audio, en tiempo real usando LTFAT y Octave. Si bien las implementaciones de los dos efectos, mostrados en el trabajo, son relativamente sencillas, sería interesante mostrar sistemas más complejos, por ejemplo, el efecto *Wah-Wah* usando este enfoque.

5. Conclusiones

El desarrollo de sistemas de procesamiento de señales puede complicarse cuando un alumno de ingeniería se adentra al PDS. Por ello, es necesario acercar a los alumnos de distintas formas, para que puedan dominar no solo los conceptos básicos necesarios, sino también sean capaces de desarrollar distintos tipos de sistemas de procesamiento de señales. Como se mostró en este trabajo, se puede

hacer uso de herramientas de desarrollo como Octave, que en conjunto con LTFAT, se puedan desarrollar sistemas que realicen algún efecto de sonido. Basta que el alumno modifique unas cuantas líneas, y pueda desarrollar sus propios sistemas de procesamiento de audio. Esto, probablemente tenga repercusión en el interés del estudiante en el PDS, al brindar un resultado en tiempo real y pueda ajustar parámetros. O incluso, con un poco de conocimientos, pueda construir una interfaz gráfica, con la que pueda interactuar con su sistema de procesamiento, acondicionando en tiempo real el audio a placer. Lo anterior, no significa que el alumno pueda desarrollar sistemas sin los conceptos del procesamiento digital de señales adecuados, al contrario, conociendo los diversos métodos de análisis y modelado de sistemas discretos, le será útil para desarrollar sus propios sistemas a partir de un código base. Se provee de un código para que se pueda modificar, e incluir las líneas de código necesarias, para desarrollar un efecto de audio deseado o un sistema de procesamiento de audio. Esto permitirá que el alumno pueda, con tan solo modificar unas cuantas líneas, desarrollar sistemas básicos o incluso más complejos, para procesar un audio con la menor latencia posible y sin requerir de hardware especializado, al que muy probablemente no tenga acceso por efecto de la emergencia sanitaria, o que en su institución no cuenten con kits de desarrollo de DSP's.

6. Bibliografía y Referencias

[1] Dogan, Ibrahim, Teaching digital signal processing. Procedia-Social and Behavioral Sciences, No. 46, 4441-4445, 2012.

[2] Downey, A., Think DSP: Digital signal processing in Python. O'Really Media Inc. 2016.

[3] Fuentes, J. I., & Moo M., Dificultades de aprender a programar. Revista Educación en Ingeniería, No. 12(24), 76-82, 2017.

[4] Gómez, C., Alducin, C., Implementación de efectos de sonido para guitarra eléctrica en la tarjeta C6713DSK. Pistas Educativas, No. 128, 539-556, 2018.

[5] Hendry J., Nur-Rifai I., & Mileniandi Y.. Sampled and discretized of short-time Fourier transform and non-negative matrix factorization: the single-channel

source separation case. Jurnal Teknologi Dan Sistem Komputer, No. 9(1), 41-48, 2021.

[6] Moreno, S., Arevalillo, M., & Diaz, W., A linear cost algorithm to compute the discrete Gabor transform. IEEE Transactions on Signal Processing, No. 58(5), 2667-2674, 2010.

[7] Pfaff, M., Malzner, D., Seifert, J., Traxler, J., Weber, H., & Wiendl, G., Implementing digital audio effects using a hardware/software co-design approach. Proc. of the 10th Int. Conference on Digital Audio Effects (DAFx-07), 2007.

[8] Průša Z., Søndergaard P., Holighaus N., Wiesmeyr C., Balazs P., The Large Time-Frequency Analysis Toolbox 2.0. Sound, Music, and Motion, Lecture Notes in Computer Science 2014, pp 419-442, 2014.

[9] Průša Z., & Holighaus N., Fast Matching Pursuit with Multi-Gabor Dictionaries. ACM Transactions on Mathematical Software, 2020.

[10] Rickel, J. W., Intelligent computer-aided instruction: a survey organized around system components. IEEE Trans Systs Man Cybernet, Vol; 19, 40-47, 1989.

[11] Søndergaard P., Torrésani B., Balazs P., The Linear Time-Frequency Analysis Toolbox. International Journal of Wavelets, Multiresolution Analysis and Information Processing, 10(4), 2012a.

[12] Søndergaard P., Efficient algorithms for the discrete Gabor transform with a long FIR window. Journal of Fourier Analysis and Applications, No. 18(3), 456-470, 2012b.

[13] Stearns, S. D., & Hush D. R., Digital Signal processing with examples in Matlab. CRC Press. 2002.

[14] Takeuchi, D., Yatabe, K., Koizumi, Y., Oikawa, Y., & Harada N., Data-driven design of perfect reconstruction filterbank for DNN-based sound source enhancement. IEEE International Conference on Acoustics, Speech and Signal Processing ICASSP 2019, pp. 596-600, 2019.

[15] Weeks, M., Digital signal processing using Matlab & wavelets. Jones & Bartlett Publishers, 2011.