Efectos de audio - Parte 1

27 de marzo de 2015

1. Generalidades

La aprobación de este curso se consigue mediante la correcta implementación de dos pequeños proyectos de programación. Éstos son propuestos aproximadamente un mes antes de cada parcial, y entregados a través de una página web habilitada para tales fines, con fecha límite de entrega fijada poco antes de cada período de parcial. Cada entrega es complementada con una pequeña prueba escrita cuyo objetivo es evaluar aspectos más teóricos relacionados con el propio obligatorio.

Es importante recalcar que **tanto la prueba escrita como el proyecto entregado son individuales.** El sistema de recepción de entregas, además de almacenar los archivos enviados por los estudiantes, realiza un control de copias contra las entregas de otros estudiantes así como de programas similares que se encuentran en la web.

En casos de ser posible, el sistema intentará además compilar y ejecutar la entrega de cada estudiante, de modo de dar un mínimo de información al respecto de qué tan bien funciona la entrega. Dependiendo del proyecto, esta evaluación preliminar estará o no disponible.

En todo caso, la evaluación preliminar mencionada anteriormente **no** determina la nota obtenida en la prueba, siendo ésta definida por una evaluación manual por parte de los docentes.

1.1. Formato del archivo a entregar

El archivo entregado debe ser un archivo comprimido en formato **zip** (NO se pueden subir rar), de nombre **nnnnnnn.zip** donde **nnnnnnn** es su número de cédula, **sin** digito de verificacion, sin espacios ni puntos, ni guión. El contenido del archivo debe incluir los siguientes elementos bajo el subdirectorio **nnnnnnn** (no pueden haber subdirectorios):

- Todos los archivos fuente creados por el estudiante (**.h** y **.c**)
- La biblioteca precompilada entregada por los docentes, si es que existe.
- Un archivo **Makefile** para compilar el o los programas requeridos en el trabajo.

Por ejemplo, supongamos el obligatorio consiste en la generación de un ejecutable de nombre `oblig`, su cédula es 1234567-8, y usted implementó dicho ejecutable en tres módulos **a.c** (y su correspondiente encabezado **a.h**), **b.c** (encabezado **b.h**) y **main.c**. Además, los docentes le entregaron una biblioteca auxiliar con un encabezado **aux.h** y código objeto **aux.o**. Entonces debe subir un archivo de nombre `1234567.zip` con el siguiente contenido:

```
1234567/
  a.c
  a.h
  b.c
  b.h
  aux.h
  aux.o
  main.c
  Makefile
```

El **Makefile** podría ser así:

```
oblig: main.o a.o b.o aux.o
  cc -o oblig main.o a.o b.o aux.o -lm

main.o: main.c
  cc -c main.c

a.o: a.c
  cc -c a.c

b.o: b.c
  cc -c b.c
```

Nota: Pueden crear un zip desde la máquina virtual con el comando `zip`; la sintaxis es

```
$zip -r nombre_archivo.zip carpeta_a_comprimir
```

en el ejemplo anterior, sería `zip -r 1234567.zip 1234567`.

1.2. Metodología de trabajo

Algunas recomendaciones generales sobre cómo trabajar con proyectos como los que se proponen aquí:

- Simplicidad (KISS - Keep It Simple, Stupid). No complicar el código más allá de lo requerido.
- Prolijidad. No importa cuánto aburra, documentar bien lo que se hace es fundamental; es muy fácil olvidarse lo que uno mismo hizo.
- Incrementalidad. Implementar y probar de a pequeños pasos. "No construir un castillo de entrada". Es muy difícl encontrar las causas de un problema si se prueba todo simultáneamente.
- No reinventar la rueda. Antes de implementar una función, buscar a ver si no existe una función en la biblioteca estándar de C que ya lo haga.

2. Introducción al problema

El problema que se plantea en este obligatorio, el procesamiento de audio, es una de las subáreas más populares e importantes del procesamiento de señales. Si bien desde el punto de vista teórico y formal, las herramientas para trabajar con este tipo de problemas se ven recién en los últimos años de la carrera de Ingeniería Eléctrica, es posible trabajar con, y comprender informalmente, muchos algoritmos importantes de procesamiento de audio del estilo de los que se ven en programas de grabación, mezclado y posproducción de audio, así como en muchas de las "pedaleras" comerciales de efectos.

2.1. Representación digital de sonido

Lo primero que tenemos que definir es cómo se representa el sonido digitalmente, de modo de poder almacenarlo en memoria y trabajar con él. Para simplificar el trabajo, nos limitaremos a archivos de un sólo canal, técnicamente denominados "monoaurales", (en contraste, el audio *stereo* tiene dos canales).

Como señal física, el sonido es una onda de presión que se propaga desde una fuente vibratoria (por ejemplo una cuerda) a través de un medio (usualmente el aire, pero puede ser otro) y es captada por una membrana sensible a la presión. En el caso del humano, dicha membrana es el tímpano. En

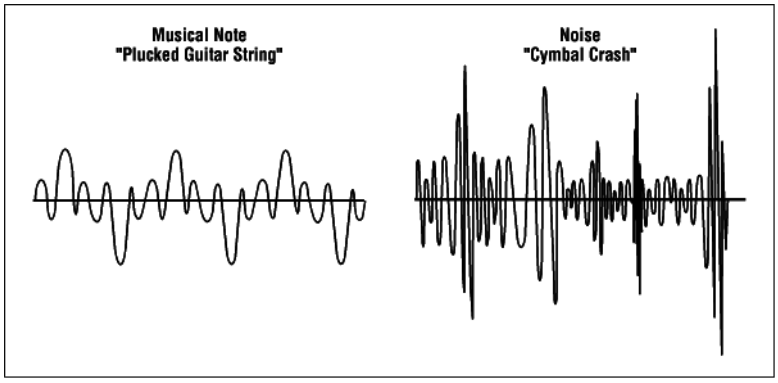

Figura 1: Arriba: ejemplos de señales de audio. guitarra (izq. y cimbal (der).

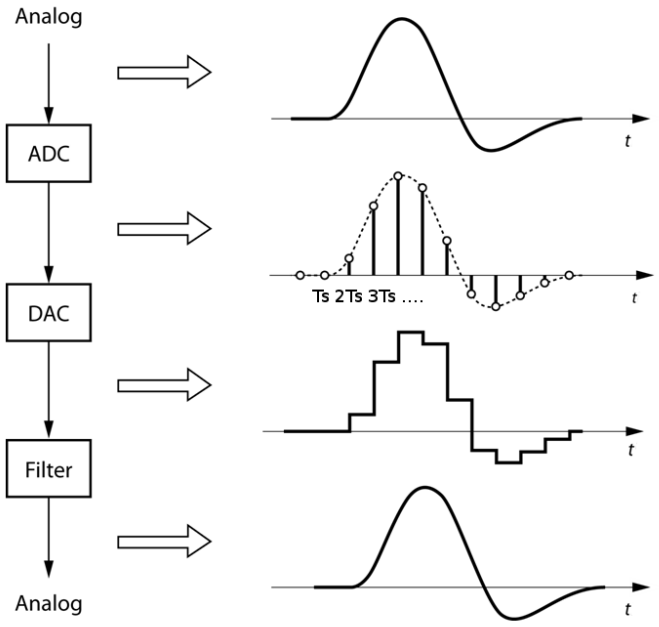

Figura 2: Proceso de conversión analógico-digital y luego digital-analógico.

un equipo de grabación, este rol lo cumple el micrófono, quien convierte las ondas de presión a ondas eléctricas. Estas señales eléctricas generan una tensión que varía con el tiempo t, llamémosle $f(t)$. La figura 1 muestra dos ejemplos de señales de audio continuas representadas como funciones continuas en el tiempo t.

En un sistema de audio digital, la señal $f(t)$ es medida (*muestreada*) periódicamente a intervalos de T_s segundos. Suponiendo que la primera muestra se toma en el tiempo $t = t_0$, la j-ésima muestra tomada corresponderá al valor de $f(t_0 + jT_s)$. Para poder representar las señales audibles con buena calidad, es necesario tomar una gran cantidad de muestras por segundo. La cantidad de muestras que se adquieren por segundo es denominada *frecuencia de muestreo*, y se cumple (trivialmente) que $f_s = 1/T_s$. Por ejemplo, para audio de calidad de CD, $f_s = 44100Hz$.

Para poder almacenar cada uno de estos valores en memoria de un equipo digital (por ejemplo una computadora), es necesario además *cuantificar* el valor de $f(t_0 + jT_s)$ de modo de aproximarlo con un número finito de bits. Los sistemas actuales utilizan entre 16 y 32 bits (por supuesto que los hay de mayor fidelidad) para representar cada muestra, usualmente como enteros con signo. En todo caso, el rango de valores representables por un entero con signo de b bits es $\Omega = [-M - 1, M]$, donde $M = 2^{b-1} - 1$ es el valor máximo (o valor de pico) de la señal. En nuestro caso utilizaremos señales

de $b = 16$ bits por muestra, con lo que $M = 2^{15} - 1 = 32767$.

El proceso anterior se llama *conversión analógica-digital* (también llamado *conversor A/D*, o *ADC* por sus siglas en inglés) y su resultado es lo que se denomina una *señal digital*. Para poder reproducir una señal digital mediante un medio analógico (por ejemplo, un parlante o auricular), es necesario revertir el proceso anterior, algo que se denomina *conversor digital-analógico*, conversor D/A, o DAC por su sigla en inglés. Toda computadora moderna, smartphone, tableta, o dispositivo interactivo moderno posee por lo menos tanto un ADC como un DAC en su hardware. La figura 2 muestra el proceso de conversión analógico-digital y luego digital-analógico típico.

Más adelante en la carrera de Ingeniería Eléctrica adquirirán formalmente todos los conceptos y teoría necesarios para tratar con señales digitales de todo tipo. Por el momento, a nuestros efectos, nos interesará sólamente entender a una señal digital como una secuencia de n números, $\mathbf{x} = (x[0], x[1], \ldots, x[n-1])$, donde $x[0] = f(t_0)$ es la primera muestra adquirida desde que comenzó a grabarse el audio, y $x[j] = f(t_0 + jT_s)$ contiene el valor muestreado de la señal en el instante jT_s. El índice j es denominado *tiempo discreto* de la señal. Algunas notas importanes:

- Asumiremos que $x[j] = 0, j < 0$.
- Cuando la señal en cuestión sea de tiempo discreto, utilizaremos paréntesis rectos [] para denotar su indexado, mientras que los paréntesis curvos () serán usados para indexar señales continuas.

Las señales digitales suelen ser almacenadas en memoria o disco como un arreglo de números enteros o de punto flotante de largo n. Para tener una idea, a 32 bits por muestra, una canción de 4 minutos de un sólo canal ocuparía unos

$$4\,\text{bytes/muestra} \times 44100\,\text{muestras/segundo} \times 240\,\text{segundos} \;=\; 42\text{megabytes}.$$

2.2. Efectos de audio

El objetivo de este obligatorio es el de implementar una serie de *efectos* de sonido a nivel digital, modificando (filtrando) una señal digital original (a la que llamaremos *entrada*) de diversas maneras para así obtener una señal *filtrada* (la *señal de salida*) que luego reproduciremos en nuestro equipo.

Los filtros digitales son una manera barata y sencilla de construir efectos sofisticados de audio del tipo que se utilizan en vivo para modificar el sonido de instrumentos (guitarra, voz, etc.) o en posproducción, para eliminar ruido, amplificar, o cosas más sofisticadas como el famoso *autotune* que permite corregir de manera creíble las melodías producidas por un cantante desafinado. *Nota: En adelante nos referiremos indistintamente a efectos y a filtros.*

Por suerte para nosotros, un gran número de estos filtros pueden construirse como la combinación sencilla de unos pocos *bloques* comunes. En esta primera etapa del obligatorio nos encargaremos de construir dichos bloques y de probar su funcionamiento como parte de unos pocos filtros sencillos. En la segunda etapa se implementarán algunos bloques adicionales y filtros más sofisticados

3. Descripción de la tarea

La tarea consistirá en implementar una serie de filtros descritos a continuación, y luego aplicar los filtros a un conjunto de archivos de audio predefinido. Todos los filtros serán invocados desde un único programa ejecutable de nombre `obligatorio`, el cual deberá generarse como resultado de la compilación del proyecto entregado a través de la utilidad `make` (para lo cual debe disponerse de un archivo `Makefile` apropiado, ver tutorial disponible en el curso). La sintaxis de línea de comandos para la ejecución de cada filtro debe ser la siguiente:

```
./obligatorio filtro parametro entrada.wav salida.wav
```

donde

- `filtro` es el nombre del filtro a aplicar,
- `parametro` es el parámetro numérico del filtro (si lo hubiera, de otro modo puede ser cualquier cosa, **pero debe estar siempre presente**),
- `entrada.wav` es el archivo de entrada al programa y
- `salida.wav` el nombre del archivo de salida del programa.

Biblioteca WAV Tanto para la lectura como para la escritura de archivos de sonido WAV, se contará para esta entrega con una pequeña biblioteca de funciones implementada por los docentes. Esta biblioteca consta de dos archivos:

- **wav.h**: declaración de las funciones de la biblioteca, y de constantes tales como la frecuencia de muestreo f_s (`FREC_MUESTREO`), valor de pico M (`MAX_VAL`), y códigos de error devueltos por la biblioteca.
- **wav.o**: implementación de las funciones anteriores, ya compilada para **Linux 32 bits** (para aquellos que utilicen Linux de 64 bits, podemos proveer una versión bajo solicitud).

El uso de dichas funciones está documentado en el propio archivo `wav.h`, como es usual en la práctica de programación moderna.

3.1. Filtros y bloques

Desde el punto de vista de la programación, haremos una distinción (un poco artificial) entre *bloques* y *filtros* de la siguiente manera: los bloques tomarán y procesarán de a *una* muestra por vez, mientras que los filtros serán aplicados a la señal en su totalidad. De esta manera, podremos combinar los distintos bloques para construir filtros relativamente complejos. En muchos casos, como veremos abajo, los filtros definidos son simplemente la aplicación muestra a muestra de un bloque asociado.

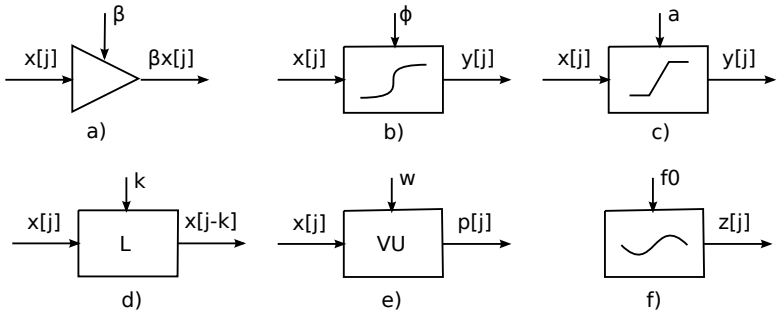

Figura 3: Bloques básicos. a) amplificador, b) transformación, c) clipping, d) retardo, e) vúmetro, f) oscilador de baja frecuencia.

3.2. Filtro amplificador

La salida de este filtro es simplemente $\mathbf{y} = \beta\mathbf{x}$ donde M es el valor de pico de la señal. Cuando $\beta = 1$, la salida será simplemente una copia de la entrada.

3.3. Filtro de normalización

El propósito de este efecto de posprocesamiento es el de amplificar la señal de audio lo máximo posible sin que ocurra distorsión. Es muy útil para mejorar grabaciones donde la señal de interés es muy baja. Esto se logra amplificando la señal, $\mathbf{y} = a\beta_N\mathbf{x}$, con β_N tal que el máximo de $\mathbf{y}$ sea el valor de pico M, $\beta_N = M/\,\text{máx}\{|\,x[j]\,|,\, j = 0, \ldots, n-1\}$ donde $|x|$ es el valor absoluto de x, y $a > 0$ es un parámetro de amplificación opcional.

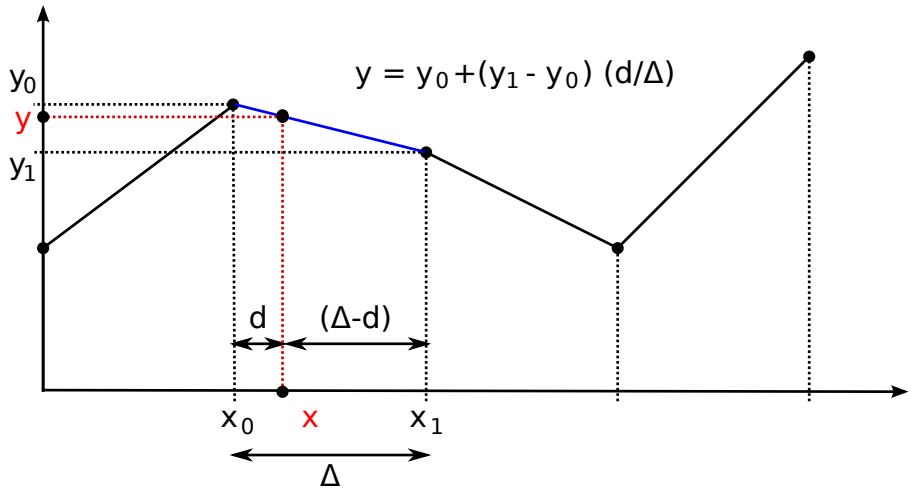

Figura 4: Interpolación lineal de una función $y = f(x)$. Se buscan x_0, x_1 tales que $x_0 \leq x < x_1$ e $y_0 = f(x_0)$ e $y_1 = f(x_1)$ son conocidos, y luego se obtiene el valor interpolado como y tal que (x, y) pasa por la recta definida por el par de puntos (x_0, y_0) y (x_1, y_1).

3.4. Transformación – LUT

La salida de este bloque es $y[j] = \phi(x[j])$, donde $\phi(x) : \Omega \to \Omega$ es una función definida en el rango de la señal de entrada Ω. Cuando evaluar tal función es costoso, se recurre a una *tabla de búsqueda* (o Look Up Table, de ahí la sigla LUT). Esta tabla almacena el valor de $\phi(x)$ para $x \in \mathcal{X} \subset \Omega$. Los valores de $\phi(x)$ para $x \notin \mathcal{X}$, son obtenidos por *interpolación*. En esta tarea utilizaremos el método de interpolación lineal, ilustrado en la figura 4.

Filtro Dada $\phi(x)$, el filtro correspondiente al bloque anterior es aquel que aplica $\phi(x)$ a cáda muestra de la entrada: $y[j] = \phi(x[j]) : j = 1, \ldots, n$.

3.5. Recorte suave – soft clipping

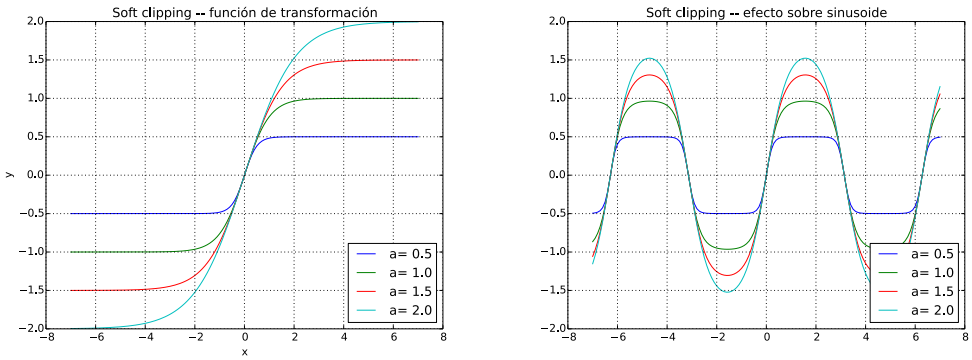

Figura 5: Soft clipping. derecha: función de soft clipping, izquierda: efecto sobre un tono puro (onda sinusoidal).

Caso particular del bloque LUT, esta función cumple un rol muy importante en audio (y electrónica en general), que es el de poder recortar (clip) una señal dentro de un rango predefinido $[-a, a]$. Existen

dos formas de hacerlo. La primera, el recorte "duro", es simplemente:

$$y = h_a(x) = \begin{cases} -a, & x < -a \\ x, & -a \leq x \leq a \\ a, & x \geq a \end{cases}, \tag{1}$$

o dicho de manera más compacta $h_a(x) = \text{máx}\{-a, \text{mín}\{x, a\}\}$. La otra forma, el recorte "blando", $s_a(x)$, busca no cortar bruscamente el valor de x al llegar a a, sino "curvarlo suavemente" hacia ese valor. Una forma posible de hacerlo es mediante la tangente hiperbólica (hay muchas otras),

$$y = s_a(x) = a\frac{e^{x/a} - e^{-x/a}}{e^{x/a} + e^{-x/a}}, \tag{2}$$

donde $a > 0$ es el valor de clipping que define el rango recortado $(-a, a)$. En la figura 5 se puede observar la forma de la curva anterior para varios valores de a, y el efecto correspondiente sobre una onda sinusoidal.

Filtro Como caso particular del filtro de LUT, el filtro asociado a esta función es también la aplicación muestra a muestra del bloque anteriormente descrito.

3.6. Distorsión – overdrive

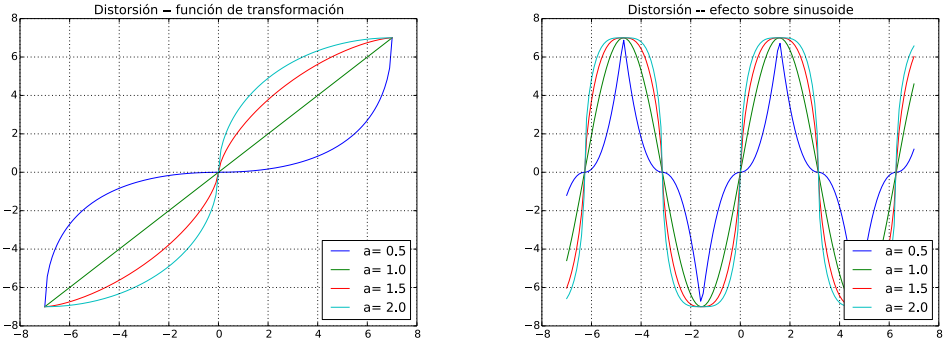

Figura 6: Ovedrive. derecha: función de deformación, izquierda: efecto sobre un tono puro (onda sinusoidal).

Efecto favorito de rockeros, la distorsión *overdrive* es un bloque tipo LUT que modifica la señal de manera que suena más "podrida". Esto se logra deformando la señal de manera que los picos sean más "gordos" (ver figura 6). Hay muchas formas de lograr esto; nosotros usaremos:

$$\phi(x) = \text{sgn}(x)\,[M^a - (|\,|x| - M\,|)^a]^{1/a}\,, \tag{3}$$

En donde M es el valor de pico definido anteriormente en la sección 2.1 y $a > 0$ es un parámetro que controla la cantidad de distorsión a aplicar ($a = 1$ corresponde a no distorsión).

Filtro Siendo también una LUT, el filtro correspondiente surge de aplicar muestra a muestra la función anterior.

3.7. Bloque de retardo, buffer circular y eco

La salida de un bloque de retardo $L_k(\cdot)$ de parámetro k, con $k \in \mathbb{N}$, es el valor que su entrada tomó k muestras antes, es decir,

$$y[j] = \begin{cases} L_k(x[j]) = x[j-k] & , j \geq k \\ 0, & j < k \end{cases} \tag{4}$$

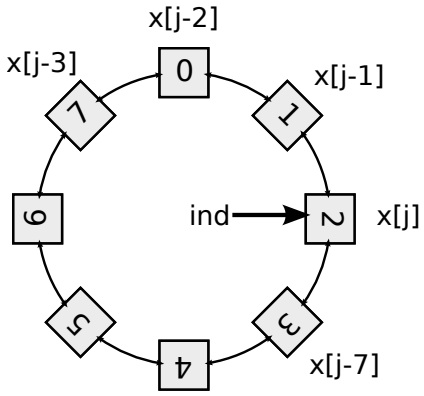

Figura 7: Buffer circular para $K = 8$. Los números dentro de las casillas indican el índice *lineal* dentro del array que se utiliza para implementar el buffer, con las K muestras más recientes correspondientes mostradas al lado de las casillas siendo que el índice a la muestra actual es `ind`=2.

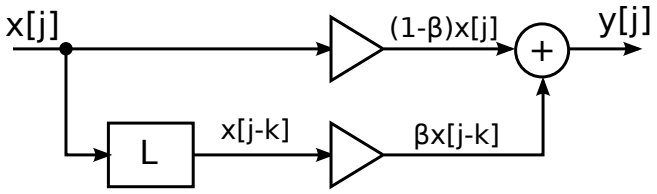

Figura 8: Eco básico (sin realimentación), también llamado "comb" (peine). Como bloque, se utiliza en efectos como el *flanger*.

Este bloque es fundamental para generar efectos del tipo *eco* o *reverberación*, como veremos más adelante. Si bien no implica ninguna operación matemática en sí, implementar este bloque requiere de almacenar rápida y eficientemente las últimas k muestras de la señal, lo cual no es trivial. Además, nos interesará producir retardos para valores de k *no enteros*, para lo cual, nuevamente, deberemos recurrir a técnicas de interpolación.

Buffer circular Para resolver el tema de almacenar las últimas k muestras se utiliza una estructura auxiliar llamada *buffer circular*. Esta estructura consta de un arreglo del tipo de datos que se quiere almacenar (por ejemplo un array de tipo `short` y nombre `buffer`) de largo $K \geq k$ y un índice (llamémosle `ind`) a la *posición actual* del buffer, es decir, a la posición en el buffer de la última muestra ingresada. Cuando se ingresa una nueva muestra, el índice es primero avanzado un lugar hacia adelante, y luego la nueva muestra es almacenada.

Lo que hace *circular* al buffer es que la posición del índice se toma *módulo* K. Por ejemplo, si luego de incrementarse, el índice `ind` llega al final del bufer (K), su valor se reinicia a 0. De la misma manera, siempre y cuando $k \leq K$, si `buffer[ind]` contiene a la muestra actual $x[j]$, la muestra $x[j-k]$ se encontrará en `buffer[(ind-k) % K]`. La figura 7 ilustra estos conceptos.

Filtro básico de eco El bloque de retardo no suele ser utilizado directamente como filtro, sino que se utiliza como parte de un bloque o filtro más complejo. En esta entrega lo utilizaremos para implementar una versión muy básica de *efecto de eco* que, dado un tiempo de retardo $k > 0$ y un parámetro $\beta < 1$, produce como salida

$$y[j] = \beta x[j-k] + (1-\beta)x[j].$$

Lo que se escucha a la salida es entonces la señal original $\mathbf{x}$ sumada a una versión retardada (y atenuada) de sí misma. La figura 8 muestra esto como diagrama de bloques.

3.8. Oscilador de baja frecuencia – LFO

La salida $z[j]$ producida por un LFO no depende de una entrada $x[j]$, sino que se usa para generar una señal periódica $s(t)$ de baja frecuencia (menos de unos pocos Hz) f_0 y amplitud A. Estas señales auxiliares son utilizadas para modificar el comportamiento de otros bloques como el amplificador, o el bloque de retardo, para producir efectos relativamente sofisticados, como veremos más adelante.

Por ejemplo, consideremos que queremos generar una señal sinusoidal de frecuencia f_0,

$$s(t) = A\sin(2\pi f_0 t).$$

Si el período de muestreo del sistema es T_s, y asumimos que $z[0] = s(0)$, entonces

$$z[j] = s(jT_s) = A\sin(2\pi f_0(jT_s)).$$

Otra señal periódica típicamente utilizada en LFOs es la *triangular*, coloquialmente un "sube y baja" entre los niveles $-A$ y A. Si T_0 es el período de la señal, tenemos que

$$s(t) = \begin{cases} -A + \frac{4A}{T_0}\psi & , \quad 0 \leq \psi < T_0/2 \\ A - \frac{4A}{T_0}(\psi - T_0/2) & , \quad T_0/2 \leq \psi < T_0 \end{cases}, \psi = t - \lfloor t/T_0 \rfloor T_0, \tag{5}$$

donde $\lfloor \cdot \rfloor$ es la operación de piso (*floor* en inglés), es decir, el entero no mayor más cercano. A la variable ψ se le denomina *fase* de la onda, y es el tiempo transcurrido desde el comienzo del período en que se encuentra la onda en un cierto instante t. Notar que en el caso de una onda sinusoidal, se tiene automáticamente que $s(t) = s(\psi)$.

Ambas señales, la sinusoidal y la triangular son muy sencillas de implementar en software. En todo caso, las LFO se parametrizan mediante dos parámetros: su amplitud A y su frecuencia f_0 (o, equivalentemente, su período T_0).

Una forma de generar señales como la anterior es simplemente evaluar la función $s(t)$ en cada tiempo de muestreo jT_s. Cuando esto es muy costoso desde el punto de vista de cálculo, se suele almacenar un período de la señal en una LUT y luego tomar los valores de allí.

Filtro Si bien este bloque no tiene un filtro asociado, nosotros lo utilizaremos también como *generador de señales*. Dados f_s, la amplitud A y el largo de la señal n, la salida de dicho filtro será una secuencia $\mathbf{z}$ de largo n.

4. Especificación de requerimientos

4.1. Generalidades

- El tipo de las muestras es `short`.
- Los bloques devuelven su salida como valor de retorno
- Los filtros *no* devuelven un valor, por lo que deben declararse con valor de retorno `void`.
- Aquellos bloques que utilizan un buffer deben actualizar el buffer, escribiendo la muestra recibida en la posición referida por el índice pasado por referencia, y finalmente incrementar el valor de dicho índice al terminar la función. Se recomienda, (pero no es obligatorio), definir un par de funciones para encapsular las operaciones de lectura y escritura sobre buffers circulares.
- Todos los buffers utilizados se asumen de tamaño $K = 2^{13} = 8192$, definido en **wav.h** como la constante `TAM_BUFFER`
- Puede utilizar la función `strcmp` de **string.h** para comparar cadenas de texto.
- Para utilizar funciones matemáticas de **math.h** deben luego linkear su ejecutable con la biblioteca de matemática con la opción `-lm` *al final de la linea* que genera el ejecutable, para que dichas funciones estén definidas.

4.2. Bloques

1. Implementar el bloque LUT descrito en la sección 3.4 como una función de nombre `bloque_lut` que tome como entrada una muestra de entrada x_0 y un arreglo T precargada con los valores de una función $\phi(x)$ evaluada en $x \in \mathcal{X}$, de modo que $T[0] = \phi(-M-1)$, $T[1] = \phi(-M-1+\Delta)$, etc., y produzca como salida una aproximación por interpolación lineal de $\phi(x)$. La tabla será almacenada como valores en punto flotante de precisión simple, para facilitar los cálculos de interpolación.

   Asumiremos que $\mathcal{X} = \{-M-1, -M-1+\Delta, \ldots, M+1-2\Delta, M+1-\Delta\}$ con $\Delta = 2^6$, lo que da una tabla de tamaño $|\mathcal{X}| = 1024$ elementos. Los parámetros en cuestión son:

   - muestra de entrada
   - tabla de lookup (arreglo de punto flotante de prec. simple)

2. Implementar el bloque de eco de la figure 8 como una función de nombre `bloque_eco` con los siguientes parámetros. La muestra de entrada debe ser almacenada en la posición actual del buffer, y dicha posición debe ser actualizada al finalizar la función:

   - muestra de entrada
   - retardo medido en cantidad de muestras $k \in \mathbb{N}^+$ (flotante de prec. simple)
   - parámetro de amplificación β (flotante de prec. simple)
   - buffer de retardo (arreglo de muestras)
   - puntero posición actual del buffer (puntero a entero)

3. Implementar el bloque LFO descrito en la sección 3.8 en una función de nombre `bloque_lfo` que: a) evalúe la función correspondiente en la fase ψ pasada como parámetro, y b) actualice la fase ψ según el período de muestreo y la frecuencia de oscilación según $\psi_{\text{nuevo}} = (\psi_{\text{viejo}} + 1/f_s) \bmod T_0$ donde $T_0 = 1/f_0$ es el período de la onda. Los parámetros son los siguientes:

   - tipo de LFO: 0 indica sinusoidal, 1 indica triangular (entero)
   - frecuencia de la onda: f_0 (flotante de precisión simple)
   - amplitud de la oscilación: a (flotante de precisión simple)
   - puntero a fase de la onda $0 \leq \psi < T_0$ (puntero a flotante de precisión simple)

Filtros Como se mencionara anteriormente, la idea es que los siguientes efectos sean implementados utilizando los bloques correspondientes definidos en el apartado anterior. En esta parte, el usuario podrá modificar *un sólo* parámetro de los filtros, de modo de hacer la interfaz más sencilla. El valor de este parámetro será tomado del argumento `parametro` definido en la linea de comandos tal como se describió en la sección 3.

1. Implementar el filtro de amplificador (comando `amp`) en una función de nombre `filtro_amp`. Además de realizar lo descrito en la sección 3.2, este filtro aplicará el *hard clipping* descrito en la sección 3.5 para mantener la señal amplificada dentro del rango representable: $y[j] = h_M(\beta x[j])$, donde $h_M(\cdot)$ es la función de *hard clipping* descrita en la ecuación (1).
   Los parámetros de la función son los siguientes:

   - arreglo muestras de entrada
   - cantidad de muestras (entero sin signo)
   - factor de amplificación $a > 0$ (flotante de precisión simple)
   - arreglo de muestras de salida

2. Implementar el filtro de normalización (comando `norm`) en una función de nombre `filtro_norm`. Además de realizar lo descrito en la sección 3.3, este filtro aplicará luego el *soft clipping* de la ecuación (2) con parámetro M para mantener la señal dentro del rango representable. Los parámetros de la función son los siguientes:

   - arreglo muestras de entrada
   - cantidad de muestras (entero sin signo)
   - factor de amplificación $a > 0$ (flotante de precisión simple)
   - arreglo de muestras de salida

3. Implementar el filtro de eco simple de la figura 8 (comando `eco`) en una función de nombre `filtro_eco`. La función debe tomar los siguientes parámetros, en el siguiente orden:

   - arreglo muestras de entrada
   - cantidad de muestras (entero sin signo)
   - retardo del eco en segundos: $k > 0$ (flotante de precisión simple)
   - factor de mezcla β (flotante de precisión simple)
   - arreglo de muestras de salida

   A los efectos de invocar el comando `eco`, el valor de β se asumirá siempre 0,25.

4. Implementar el filtro de overdrive descrito en la sección 3.6 (comando `over`) en una función de nombre `filtro_over` que tome la siguiente lista de parámetros, en el siguiente orden, y deforme la señal de acuerdo a la función $\phi(x)$ definida en la ecuación (3):

   - arreglo muestras de entrada
   - cantidad de muestras (entero sin signo)
   - factor de distorsión: $a > 0$ (flotante de precisión simple)
   - arreglo de muestras de salida

   Se sugiere inicializar una LUT con los valores de $\phi(x)$ y luego utilizar el bloque LUT definido anteriormente.

5. Implementar el filtro de *soft clipping* (comando `clip`) en una función de nombre `filtro_clip` que tome la siguiente lista de parámetros, en el siguiente orden, y aplique la función $s_a(x)$ definida en la ecuación (2):

   - arreglo muestras de entrada
   - cantidad de muestras (entero sin signo)
   - rango de clipping **dado como fraccion de valor de pico**: $u > 0$ de modo que la señal se recorte entre $(-a, a)$ con $a = uM$ (flotante de precisión simple)
   - arreglo de muestras de salida

   Se sugiere inicializar una LUT con los valores de $s_a(x)$ y luego utilizar el bloque LUT definido anteriormente.

6. Implementar un generador de ondas triangulares (comando `tri`) en una función de nombre `filtro_tri` que tome la siguiente lista de parámetros, en el siguiente orden:

   - cantidad de muestras (entero sin signo)
   - frecuencia en Hz de la señal a generar (flotante de precisión simple)
   - amplitud de la señal a generar (flotante de precisión simple)

- arreglo de muestras de salida

A los efectos de invocar el comando `tri`, la amplitud de la señal a generar será M, y el largo estará dado por el largo del archivo de entrada (que a otros efectos será ignorado).

7. Implementar un generador de ondas sinusoidales (comando `sen`) en una función de nombre `filtro_sen` que tome la siguiente lista de parámetros, en el siguiente orden:

   - cantidad de muestras (entero sin signo)
   - frecuencia en Hz de la señal a generar (flotante de precisión simple)
   - amplitud de la señal a generar (flotante de precisión simple)
   - arreglo de muestras de salida

   A los efectos de invocar el comando `tri`, la amplitud de la señal a generar será M, y el largo estará dado por el largo del archivo de entrada (que a otros efectos será ignorado).

8. Implementar la función `main` que invoque a cada uno de los filtros anteriores, con los parámetros correspondientes, de acuerdo a los argumentos de entrada en la línea de comandos, tal como se explicara en la sección anterior. Más concretamente, la función main debe:

   - Leer los argumentos de línea de comandos y definir de allí los valores del tipo de filtro a aplicar, parámetro, archivo de entrada y archivo de salida.
   - Cargar la señal de entrada utilizando las funciones provistas en **wav.h**
   - Invocar al filtro especificado sobre la señal de entrada, generando una señal de salida
   - Combinar entrada y salida del filtro mediante el parámetro de mezcla β descrito anteriormente, produciendo así la salida final del programa
   - Guardar la señal de salida final a un archivo `WAV` utilizando las funciones provistas en **wav.h**