Procesamiento de Señales de Audio

Manual de Prácticas

Cursos relacionados

Procesamiento de Señales de Audio
Procesamiento Digital de Señales
Señales y Sistemas Discretos

Programas educativos

Ingeniería Electrónica
Ingeniería Biomédica
Ingeniería en Telecomunicaciones
Posgrado en Ingeniería Electrónica
Posgrado en Ciencias de la Ingeniería
Maestría en Matemática Aplicada y Física Matemática

Autor

Dr. Alfonso Alba Cadena
Facultad de Ciencias
Universidad Autónoma de San Luis Potosí

Febrero, 2019

Contenido

Lista de Figuras

Lista de Cuadros

Capítulo 1

Introducción

1.1 Presentación

El sonido se define, desde un punto de vista físico, como vibraciones mecánicas que se transmiten por un medio elástico, por ejemplo el aire. Desde un punto de vista psicológico, el sonido consiste en la percepción y la posible interpretación de tales ondas por parte del sistema auditivo y el cerebro de un ser vivo. El sonido, al ser una señal que se propaga por un medio, puede transmitir información; pero la correcta interpretación de esta información dependerá del cerebro que la recibe. Por ejemplo, un médico está entrenado para reconocer anomalías cardiacas o pulmonares mediante la auscultación con un estetoscopio, el cual amplifica los sonidos internos del cuerpo humano; de manera similar, un músico está entrenado para reconocer los distintos intervalos entre diferentes notas musicales.

Al igual que ocurre con otros sentidos del cuerpo humano, principalmente la visión, por siglos se ha buscado modelar matemática y computacionalmente tanto los procesos de audición como de generación o síntesis de ondas de audio que se asemejen a las que se producen por fenómenos naturales. Una de las principales herramientas modernas para estas tareas radica en el procesamiento digital de señales. Existen en la actualidad un gran número de aplicaciones basadas en procesamiento digital de señales de audio, muchas de las cuales son ya ubicuas como la codificación y transmisión de señales de voz en sistemas de telefonía celular, la codificación de archivos de audio como MP3 o FLAC, la grabación, mezcla y masterización digital en la industria del audio y el cine, la síntesis de sonidos con fines tanto musicales como de diseño sonoro, el reconocimiento de voz y las interfaces humano-computadora basadas en voz, y el uso de sonidos para monitoreo y retroalimentación de diversos sistemas, incluido el monitoreo de pacientes en hospitales.

Por estos motivos, el procesamiento digital de señales de audio puede ser tan importante como lo son el procesamiento de imágenes digitales o el procesamiento de señales electrofisiológicas; así mismo, puede ser de interés para

estudiantes de diversas ramas de la ciencia y de la ingeniería.

El presente manual de prácticas está diseñado con la intención de conducir al alumno hacia el desarrollo de aplicaciones basadas en audio, a través de la implementación de bloques de procesamiento fundamentales para la síntesis y el análisis de diversos tipos de señales acústicas.

1.2 Programas de estudio y cursos relacionados

Este manual está orientado a alumnos de cursos relacionados con el Procesamiento Digital de Señales, en cualquier área de la Ingeniería y las Matemáticas Aplicadas, tanto a nivel licenciatura como posgrado. De manera particular, las prácticas que aquí se presentan buscan reforzar la mayor parte de los conceptos estudiados en los siguientes cursos de la Facultad de Ciencias de la UASLP:

- Procesamiento de Señales de Audio (Licenciatura en Ingeniería Electrónica)
- Procesamiento Digital de Señales (Licenciaturas en Ingeniería Biomédica, Ingeniería Electrónica e Ingeniería en Telecomunicaciones)
- Señales y Sistemas Discretos (Posgrado en Ingeniería Electrónica)
- Procesamiento de Señales Digitales (Maestría en Matemática Aplicada y Física Matemática)

Adicionalmente, este manual puede servir como material de apoyo para los siguientes cursos:

- Procesamiento de Señales Aplicado a las Comunicaciones (Ingeniería en Telecomunicaciones)
- Sistemas en Tiempo Real (Licenciatura en Ingeniería Electrónica)
- Programación de Dispositivos Móviles (Licenciatura en Ingeniería Electrónica)
- Filtros Digitales (Licenciatura en Ingeniería Electrónica)
- Procesamiento de Señales en Tiempo Real (Posgrado en Ingeniería Electrónica)

1.3 Objetivos

1.3.1 Objetivo general

El objetivo principal de este manual es reforzar los conocimientos adquiridos en los cursos de procesamiento digital de señales, y a la vez introducir al alumno al procesamiento de señales en tiempo real, mediante el desarrollo de ejemplos y aplicaciones orientadas al procesamiento de audio.

1.3.2 Objetivos particulares

De manera particular se busca que el alumno

- Comprenda, de manera general, el funcionamiento de un sistema de procesamiento de señales en tiempo real.
- Sea capaz de utilizar las entradas y salidas de audio en tiempo real dentro de sus propias aplicaciones.
- Ponga en práctica los conocimientos adquiridos en los cursos de procesamiento digital de señales mediante la implementación de ejemplos de propósito específico.
- Incremente su experiencia en programación estructurada y orientada a objetos.

1.4 Impacto en la formación integral

Para desarrollar las prácticas que aquí se presentan es necesario combinar múltiples conocimientos y habilidades adquiridos durante los primeros semestres de la licenciatura. En particular son indispensables los fundamentos matemáticos y de programación. Idealmente se contará con experiencia en lenguajes de cómputo numérico como Matlab ú Octave y en programación orientada a objetos (C++, Java o alguno de sus derivados).

La totalidad de las prácticas podrán desarrollarse en una computadora común, en un entorno o lenguaje de programación de alto nivel y de libre distribución, de manera que el alumno podrá realizar las prácticas y experimentar incluso fuera del laboratorio.

De esta manera, la implementación de los métodos que aquí se presentan puede contribuir de manera muy significativa a la experiencia del alumno en el ámbito de la programación, introduciéndolo al desarrollo de algoritmos de procesamiento en tiempo real que pueden implementarse en arquitecturas embebidas. En particular, se hace mucho énfasis en la programación orientada a objetos, donde características como la herencia y el polimorfismo permiten desarrollar las prácticas mediante un enfoque modular, en el cual los bloques de procesamiento implementados en las primeras prácticas forman la base para las prácticas intermedias y avanzadas.

1.5 Reglamento y normas de seguridad

Cuando las prácticas se desarrollen en el ámbito de un laboratorio, es importante tener en cuenta el reglamento del mismo. De manera general, en cualquier laboratorio se deben seguir las siguientes normas:

- Mantener orden y silencio. Dada la naturaleza de las prácticas, es importante utilizar audífonos y utilizar un nivel moderado de volumen en las prácticas que generen sonido.

- Cuidar los equipos que se encuentran en los laboratorios.
- No introducir comida ni bebidas al laboratorio.
- Atender las instrucciones de los encargados del laboratorio.

Si bien para estas prácticas no se requiere el manejo de materiales peligrosos, es importante resaltar que la reproducción de audio a alto volumen puede dañar sus oídos (sobre todo al utilizar audífonos) y/o las bocinas del sistema. **Siempre que trabaje en una práctica donde se genere o reproduzca audio, asegúrese de bajar el volumen antes de ejecutar el programa, así como al variar parámetros que podrían ocasionar ruidos abruptos (clicks, pops, altas resonancias, etc).**

1.6 Software a utilizar

Para el desarrollo de la mayoría de las prácticas se requiere, de manera general, un lenguaje de programación de alto nivel y alguna librería para la entrada y salida de audio en tiempo real. Además, será de utilidad contar con algún entorno de cómputo numérico para el desarrollo rápido de algoritmos y el análisis fuera de línea.

Se sugiere al profesor alentar el uso de software multiplataforma de código abierto (open source) y libre distribución, de manera que prácticamente cualquier alumno pueda instalar y utilizar el mismo software que en el laboratorio.

El software recomendado para el desarrollo de las prácticas, y el que se utilizará en los ejemplos, es el siguiente:

- **Processing 3** - `https://processing.org`

  Processing es un lenguaje de programación de alto nivel derivado de Java (y que por lo tanto, cuenta con una sintaxis muy similar a C++) orientado hacia las artes visuales [1, 2]. Si bien no está diseñado para desarrollar aplicaciones grandes para usuarios finales, su facilidad de uso y orientación gráfica lo convierten en un excelente lenguaje para el desarrollo de prototipos. Además, cuenta con un amplio soporte de la comunidad, lo cual incluye librerías de audio desarrolladas tanto por el equipo de Processing como por terceros. En este manual se utiliza la versión 3 de Processing.

- **Beads** - `http://www.beadsproject.net`

  Beads es una librería para Java y Processing para el manejo de audio en tiempo real. A la fecha, es la librería mas completa ya que incluye diversos objetos para la síntesis y análisis de audio. La librería se basa en objetos llamados UGens (Unit Generators), los cuales consisten básicamente en cajas negras con entradas y salidas de audio, que pueden interconectarse entre sí, por lo que es posible desarrollar algoritmos de manera modular [3, 4].

- **Minim** - `http://code.compartmental.net/minim`

  Minim es otra librería de audio para Java y Processing basada en UGens, que incluye diversos objetos pre-implementados para la síntesis y análisis de audio.

- **Code::Blocks + MinGW** - `http://codeblocks.org`

  Code::Blocks es un entorno integrado de desarrollo (IDE) en lenguajes C y C++ para Windows, compatible con múltiples compiladores, incluyendo GCC (GNU C), Microsoft Visual C++, y otros. En particular se recomienda descargar la versión que incluye el compilador MinGW, que es compatible con GNU C.

- **PortAudio** - `http://www.portaudio.com`

  PortAudio es una librería de audio multiplataforma para C y C++, de libre distribución y fácil de utilizar. Es compatible con MinGW, Microsoft Visual C++, y otros.

- **Octave** - `https://www.gnu.org/software/octave`

  Octave es un entorno de cómputo numérico para programación científica, basado en, y compatible hasta cierto punto, con el software comercial Matlab.

- **Puredata** - `http://puredata.info`

  Puredata (Pd) es un entorno de programación visual multiplataforma para procesamiento de audio en tiempo real y multimedia. Desarrollado por Miller Puckette como una alternativa de código abierto al software comercial Max/MSP. Este entorno puede ser de gran utilidad para algunas de las prácticas mas avanzadas. Se recomienda descargar la versión básica (Pd Vanilla) [5].

- **Audacity** - `https://www.audacityteam.org`

  Audacity es un editor de audio multipistas que cuenta con un gran número de funciones de utilería.

En las primeras prácticas se proporcionarán ejemplos tanto en Processing (usando tanto la librería Beads como la librería Minim) y en GNU C/C++ (usando la librería PortAudio para MinGW). Posteriormente se utilizará únicamente la combinación de Processing y Beads para aprovechar las ventajas gráficas y de interfaz de usuario que proporciona este lenguaje. Para las prácticas mas avanzadas y aplicaciones finales podrá utilizarse también un lenguaje de alto nivel orientado al procesamiento de audio, como Puredata o SuperCollider.

Cabe mencionar que la instalación del software, y en particular de las librerías para Java, Processing o C++, queda fuera del alcance de este manual y se deja como tarea al alumno.

1.7 Organización de las prácticas

Las prácticas se dividen en las siguientes cuatro secciones:

1. **Prácticas de nivel básico:** En estas prácticas se pretende que el alumno adquiera una experiencia inicial sobre la programación de aplicaciones para el análisis y procesamiento de audio en tiempo real. Se presentan soluciones completas y en múltiples lenguajes. Estas prácticas son breves, pero fundamentales para adquirir la experiencia necesaria para el desarrollo de los siguientes bloques. Pueden formar parte de cualquier curso inicial de procesamiento digital de señales, así como de cursos mas avanzados o especializados.

2. **Prácticas de nivel intermedio:** Estas prácticas están más enfocadas a la síntesis básica de audio, particularmente la implementación de osciladores con diversas características espectrales, así como de envolventes y líneas de retardo. Estos tres componentes: osciladores, envolventes y líneas de retardo, junto con los filtros (los cuales se estudian en la sección anterior) son los principales bloques a partir de los cuales se construyen muchos de los procesos aplicados a señales de audio. Para estas prácticas se presentan soluciones básicas en el lenguaje Processing, y se proponen diversas mejorías y extensiones que el alumno debe implementar por cuenta propia. Las prácticas de esta sección pueden ser de interés principalmente en cursos orientados al procesamiento de señales en tiempo real y/o sistemas embebidos, pero también pueden servir como casos de estudio en cursos básicos de procesamiento digital de señales.

3. **Prácticas de nivel avanzado:** Las prácticas avanzadas se orientan a la implementación de procesos mas complejos, tomando como base los bloques desarrollados en las secciones anteriores y aprovechando la modularidad de los mismos. Se presentan soluciones parciales, a manera de bosquejos, dejando al alumno la implementación de una aplicación completa con una interfaz de usuario que permita manipular los parámetros del proceso. Estas prácticas están orientadas hacia un curso especializado de procesamiento de señales de audio, pero también pueden considerarse como propuestas para proyectos finales de un curso básico de procesamiento de señales.

4. **Aplicaciones:** Las aplicaciones que aquí se presentan no son prácticas como tales, sino mas bien ideas sobre las cuales un alumno puede desarrollar un proyecto mas elaborado, por ejemplo, como evaluación final del curso. En este sentido, para cada proyecto se presenta una descripción del algoritmo con suficientes detalles técnicos, pero sin demasiados detalles sobre la implementación, la cual queda totalmente a cargo del alumno. También se hacen sugerencias sobre cómo personalizar cada proyecto. Las ideas que aquí se presentan pueden dar lugar a proyectos finales de cursos de procesamiento de señales de audio, procesamiento de señales en tiempo real, y sistemas embebidos.

1.8 Seriación de las prácticas

Este manual consta de 23 prácticas, por lo que es relativamente extenso. Idealmente, en un curso orientado al procesamiento de audio digital (no un curso básico de procesamiento de señales), se puede abarcar el material completo realizando algunas prácticas en sesiones de laboratorio y otras como tareas en casa.

Sin embargo, dependiendo de las necesidades del curso y el tiempo disponible, el profesor puede elegir enfocarse en un subconjunto de prácticas específico. Para esto, se pueden agrupar las prácticas según su enfoque y aplicación de la manera siguiente:

- **Esenciales:** Este conjunto de prácticas forman el pre-requisito necesario para comprender e implementar cualquier práctica subsecuente. En ellas se estudia cómo acceder al sistema de audio e implementar procesos muy básicos en tiempo real. Además, la inclusión de un generador senoidal (Práctica 3.1) proporciona una señal de prueba adicional.
  - 2.1.- Salida de audio
  - 2.2.- Control de amplitud
  - 2.3.- Entrada de audio y retroalimentación
  - 3.1.- Oscilador senoidal
- **Sistemas lineales e invariantes en el tiempo:** En estas prácticas se estudian algunos sistemas LIT comunes, por lo que encajan bien en cualquier curso básico de procesamiento digital de señales o diseño de filtros digitales.
  - 2.4.- Filtros de primer orden
  - 2.5.- Filtros de segundo orden
  - 2.6.- Filtros de ecualización
  - 3.6.- Líneas de retardo
- **Análisis de señales de audio:** Este grupo comprende prácticas orientadas al análisis y la obtención de características básicas de las señales de audio. Debe complementarse con el grupo anterior, ya que los filtros también son de utilidad para el análisis de señales.
  - 2.7.- Estimación de amplitud
  - 3.7.- Análisis de Fourier
  - 3.8.- Estimación de frecuencia
- **Efectos de audio:** En estas prácticas se estudia el funcionamiento de algunos efectos digitales comunes, como distorsión, eco o reverberación. Se requiere haber estudiado las prácticas del bloque de Sistemas LIT.
  - 3.2.- Concepto de modulación y su implementación

  - 3.4.- Waveshaping
  - 4.4.- Retroalimentación Parte II
  - 4.5.- Efectos basados en retardos
  - 4.7.- Filtros pasa-todo y phasers
  - 4.8.- Reverberación artificial

- **Síntesis:** Este grupo abarca las prácticas orientadas a la generación artificial de sonidos utilizando distintas técnicas. Las cuatro técnicas presentadas (aditiva, modulación de frecuencia, sustractiva y modelado físico) son independientes entre sí, por lo que el profesor puede elegir en cuáles enfocarse.
  - 3.3.- Rampas y envolventes
  - 3.5.- Tablas de ondas
  - 4.1.- Síntesis aditiva
  - 4.2.- Síntesis por modulación de frecuencia
  - 4.3.- Síntesis sustractiva
  - 4.6.- Síntesis por modelado físico

Parte I

Prácticas de nivel básico

Contenido

Capítulo 2

Salida de audio

Nombre del estudiante	Calificación

2.1 Relación con los programas de estudio

Materia	Unidad y tema
Procesamiento Digital de Señales (IE / IT / IB)	1.1 - Definición y tipos de señales 4.1 - Muestreo de señales en tiempo continuo
Procesamiento de Señales de Audio (IE)	1.1 - Señales de audio digitales

2.2 Introducción

Una señal es una cantidad que varía, generalmente con respecto al tiempo, pero es posible también encontrar señales que varían con respecto a la posición en el espacio, o con respecto a alguna otra variable [6]. Comunmente, las señales representan cantidades físicas; por ejemplo, un voltaje que se mide entre dos nodos en un circuito eléctrico, la posición de un punto específico sobre una cuerda que vibra, los cambios en la presión del aire producidos por algún sonido, el color de los pixeles en una imagen o en un video digital, etc. Así mismo, se asume que una señal conlleva algún tipo de información, y en muchos casos, una sola señal puede contener simultáneamente múltiples tipos de información. Un ejemplo muy simple y común se presenta precisamente en las señales de audio: a partir de una sola señal, el cerebro humano es capaz de reconocer voces, palabras y frases, sonidos emitidos por animales, ruidos de máquinas y vehículos, notas musicales y los instrumentos que las producen, etc. Mas aún, el contar con dos oídos, cada uno de los cuales recibe una señal ligeramente distinta, nos permite

estimar la orientación y distancia a la que se encuentran las diversas fuentes que emiten los sonidos.

Matemáticamente, una señal simple (real y univariada) se representa como una función del tiempo $x(t)$, donde tanto t como $x(t)$ son números reales. Para poder almacenar y/o procesar una señal en una computadora, es necesario considerar una versión discretizada $x[n]$ de la señal $x(t)$, donde n es un número entero. La señal en tiempo discreto se obtiene mediante un proceso de muestreo donde se "capturan" los valores de la señal continua a intervalos de tiempo regulares. Esto se expresa matematicamente como sigue:

$$x[n] = x(nT),$$

donde T es el periodo de muestreo, y $f_s = 1/T$ es la frecuencia de muestreo (el número de capturas o muestras que se realizan por unidad de tiempo). En el mundo del audio digital es común encontrar frecuencias de muestreo de 44100 Hz o 44.1 kHz (el estandar usado en CDs), 48 kHz, 88.1 kHz, 96 kHz y 192 kHz.

También es posible encontrar señales multivariadas; es decir, donde existen dos o mas cantidades que varían con el tiempo. Un ejemplo común en el ámbito del audio son las señales estéreo, las cuales consisten en dos señales que corresponden a los canales izquierdo y derecho de un sistema de audio (por ejemplo, nuestros oidos). Para denotar este tipo de señales podemos utilizar dos funciones, por ejemplo $x_L(t)$ y $x_R(t)$ respectivamente para los canales izquierdo y derecho; o bien, una sola función vectorial $x(t) \in \mathcal{R}^2$, donde la primer componente corresponde al canal izquierdo y la segunda al derecho. En el caso de las señales en tiempo discreto, se asume que todas las componentes de una señal multivariada o multicanal son muestreadas con la misma frecuencia de muestreo; es decir $x_L[n] = x_L(nT)$ y $x_R[n] = x_R(nT)$. Existen también sistemas de audio con un mayor número de canales, como los sistemas cuadrafónicos o los sistemas surround que contienen desde 3 hasta 24 canales.

Intuitivamente, una señal discreta $x[n]$ no contiene toda la información que conlleva la señal continua subyacente $x(t)$. El teorema de muestreo de Nyquist establece las condiciones que deben cumplirse para poder recuperar $x(t)$ a partir de $x[n]$, así como lo que ocurre cuando estas condiciones no se cumplen. De manera resumida, para poder recuperar la señal continua $x(t)$ a partir de la señal discreta $x[n]$, se requiere que la frecuencia de muestreo sea por lo menos el doble del ancho de banda de la señal continua.

Por otra parte, aunque formalmente $x[n]$ representa a un número real (para un valor de n determinado), este número tendrá que convertirse a uno de los tipos de datos numéricos con los que podemos trabajar en un lenguaje de programación; ya sea entero (como `int` o `short`) o de punto flotante (como `float` o `double`). Esto limitará tanto el rango como la resolución de los valores con los que se pueda representar cada muestra de la señal en la computadora, y podrá ocasionar distorsiones en la señal adquirida, las cuales también se estudiarán mas adelante.

Para poder estudiar los fenómenos mencionados anteriormente, así como muchos otros, será útil desarrollar un marco de trabajo dentro del cual podamos

experimentar con diversos procesos de una manera relativamente independiente a la plataforma. El primer paso consistirá en elegir una plataforma de trabajo e implementar las bases para la generación y reproducción de señales de audio en tiempo real.

La mayoría de los sistemas de procesamiento de audio digital involucran la transferencia y manipulación de cientos de miles de datos por segundo. Para realizar esto de manera mas eficiente, las señales de audio se adquieren, procesan y transmiten por bloques (también llamados *buffers*), donde cada bloque abarca un cierto número B de muestras (usualmente B es una potencia de 2, como 128 o 1024). Por ejemplo, cuando reproducimos un archivo de audio, el sistema llena primero un buffer con las primeras B muestras. Una vez que el buffer se llena, se envía todo el bloque al DAC para reproducir el segmento de audio correspondiente, a la vez que se va llenando nuevamente el buffer con las siguientes B muestras. Esto tiene tres consecuencias importantes:

1. **Latencia.-** Una vez que se inicia la reproducción del audio, el usuario no escuchará nada sino hasta que se haya llenado el buffer por primera vez. De manera mas general, el usuario estará en todo momento escuchando la señal que corresponde al buffer *anterior* al que está siendo procesado. Este retardo entre el tiempo en que se procesa una señal y el tiempo en que se escucha la señal procesada se conoce como *latencia* y es aproximadamente igual a el tamaño de bloque B dividido entre la frecuencia de muestreo f_s. Por ejemplo, un sistema que tiene una frecuencia de muestreo de 44100 Hz y un tamaño de bloque de 512 muestras tendrá una latencia aproximada de 11.6 ms. Dependiendo de la aplicación, podría ser deseable reducir la latencia (ya sea reduciendo el tamaño de bloque o incrementando la frecuencia de muestreo) a costa de un incremento en el costo computacional.

2. **Responsividad.-** El llenado del buffer debe realizarse de manera sincronizada para evitar interrupciones en la reproducción del audio. Por este motivo, este proceso se realiza típicamente durante una interrupción del CPU durante la cual no se procesan otros eventos. Esto significa que en una aplicación interactiva, las acciones del usuario no tendrán efecto en el buffer que se está procesando al momento de la acción, sino hasta el siguiente buffer. En muchos casos, este retardo es imperceptible para el usuario (por ejemplo, en dispositivos comandados por voz), pero en algunas aplicaciones puede representar un serio problema (como en el caso de un procesador de efectos para guitarra o de una batería electrónica).

3. **Costo computacional.-** Digamos que en un determinado momento el buffer A se está enviando al DAC para su reproducción. Simultáneamente, otro buffer B debe estarse procesando y llenando, para que quede listo para enviarse al DAC tan pronto concluya la reproducción del buffer A. Claramente, el procesado y llenado de un buffer debe realizarse antes de que concluya la reproducción del buffer anterior, por lo que el tiempo

disponible para procesar o generar la señal de audio es relativamente limitado (y acotado por la latencia). Además, este proceso conlleva también un *overhead* ya que es necesario guardar el estado de la máquina antes de la interrupción, preparar los buffers, etc. Si el llenado del buffer no se concluye a tiempo, podría haber interrupciones en el audio que suelen apreciarse como ruidos abruptos (*clicks* y *pops*) o *stutter* (cuando el buffer anterior se reproduce una y otra vez mientras se alista el siguiente buffer).

2.3 Objetivos didácticos

- Ayudar en la comprensión de los siguientes conceptos básicos: señal en tiempo discreto, frecuencia de muestreo
- Comprender el funcionamiento básico de un sistema de reproducción de audio en tiempo real y el concepto de latencia
- Implementar un marco de trabajo para la reproducción de audio en tiempo real en un lenguaje de alto nivel

2.4 Material

- Una computadora con alguna de las siguientes combinaciones de software instalado:
  - Processing y Beads
  - Processing y Minim
  - GNU C++ (e.g., MinGW) y PortAudio
- Bocinas o audífonos estéreo

2.5 Procedimiento

Elija una de las tres plataformas mencionadas en la sección de Material, o bien, la que el profesor le indique, y escriba el programa correspondiente para generar ruido blanco llenando el buffer de salida con números aleatorios. También es recomendable implementar el programa en las tres plataformas, o por lo menos leer los detalles de cada una de ellas, para posteriormente decidir aquella que el alumno considere la mas adecuada para futuras práticas.

Precaución: Antes de ejecutar el programa, asegúrese de tener el volumen a un nivel bajo para evitar ruidos abruptos.

A muy grandes rasgos, toda aplicación que procese o genere audio requiere de las siguientes etapas:

1. **Inicializar el sistema de audio.-** Esto se realiza por lo general al inicio del programa, y la manera de hacerlo depende completamente de

la librería de audio elegida; sin embargo, el procedimiento será siempre el mismo para todas las prácticas, por lo que conviene englobarlo en una función que pueda reutilizarse fácilmente.

2. **Procesado de los bloques.-** Esto se realiza mediante una función *callback*, a la que el sistema de audio llama automáticamente cada vez que se requiera procesar un bloque o buffer de audio. La implementación de la función callback cambiará dependiendo del proceso que se desea implementar, por lo que será diferente en cada práctica, y en ningún caso se llamará de manera explícita a la función callback desde el programa del alumno.

3. **Cerrar el sistema de audio.-** Justo antes de que termine el programa puede ser necesario realizar tareas adicionales para detener la entrada y salida de audio y liberar los recursos asociados al sistema de audio. En el caso del lenguaje Processing, estas tareas se realizan de manera automática.

Como consideración adicional, utilizaremos en todos los casos números de punto flotante de 32 bits (i.e., variables de tipo `float`) para representar las muestras de la señal de audio. Algunas librerías (como PortAudio) permiten usar distintos formatos, incluyendo enteros de 8, 16, 24 y 32 bits, pero el uso de números de punto flotante nos permitirá realizar operaciones con mayor precisión y reducirá las diferencias de implementación entre una y otra plataforma. El rango dinámico para las señales de audio representadas en formato de punto flotante va de -1.0 a 1.0. Valores fuera de ese rango pueden causar distorsión en las señales. En esta práctica el ruido se generará mediante números aleatorios de -1 a 1.

2.5.1 Processing y Beads

Una vez instalado Processing 3 es posible instalar la librería Beads a través de la opción `Añadir herramienta...` del menú `Herramientas`. Se recomienda cerrar y reiniciar Processing después de instalar cualquier librería.

Todo el procesamiento de audio en Beads se realiza a través de clases llamadas *Unidades Generadoras* o simplemente *UGens*. Una UGen es una caja negra con entradas de audio y salidas de audio. La UGen toma un bloque de las señales de entrada, realiza algún proceso, y calcula un bloque para las señales de salida. Dentro de Beads se incluye ya un gran número de UGens para realizar diversos tipos de procesos, que van desde generación básica de audio (osciladores, ruido, reproducción de archivos de audio), procesado (filtros, retardos, efectos), y hasta análisis (transformada de fourier y espectrogramas). Algo interesante es que las entradas de audio de una UGen pueden ser, a su vez, otras UGens, lo que permite construir procesos complejos de manera modular, a partir de procesos mas simples.

Para ser justos y consistentes con aquellos que usarán lenguaje C y PortAudio, no utilizaremos en este manual ninguna de las UGens que vienen ya inclu-

idas con Beads, sino que implementaremos nuestras propias UGens, heredadas de la clase base `UGen`, que sirva para nuestro propósito. En esta ocasión, solamente es necesario implementar dos funciones: el constructor, en el cual se define el número de entradas y salidas que tiene la UGen, y el método `calculateBuffer()`, donde se procesa un bloque de audio. Ciertamente, `calculateBuffer()` es una función callback que será llamada por el sistema de audio en el momento que se requiera procesar un bloque, mas no será llamada explícitamente dentro de nuestro programa.

Toda UGen cuenta además con dos arreglos bidimensionales de tipo `float[][]`, llamados `bufIn` y `bufOut`, que contienen, respectivamente, los buffers de entrada y salida del bloque de audio que se procesa al momento de llamar a `calculateBuffer()`. El primer subíndice de estos arreglos representa el número de canal, mientras que el segundo subíndice representa el número de muestra dentro del bloque. El tamaño del bloque está dado por el miembro `bufferSize` de la clase`UGen`.

Con esta información, es posible implementar una UGen para generar ruido blanco como sigue:

```
// Para usar las clases de la librería Beads es necesario importarla
import beads.*;

// Implementamos una Unidad Generadora (UGen) para reproducir ruido blanco
public class RuidoBlanco extends UGen {

  // En este caso, el constructor simplemente llama al constructor de UGen
  public RuidoBlanco(AudioContext context, int canales) {
    super(context, canales);
  }

  // calculateBuffer() se llama automáticamente cuando se requiere llenar un buffer
  public void calculateBuffer() {
    for (int c = 0; c < getOuts(); c++) {
      float[] out = bufOut[c];
      for (int i = 0; i < bufferSize; i++) {
        out[i] = random(-1, 1);
      }
    }
  }
}
```

Para inicializar el audio es necesario crear un objeto de clase `AudioContext`, a través del cual se tendrá acceso a las entradas y salidas físicas de audio. Además, es necesario pasar el objeto `AudioContext` a los constructores de todas las UGen. Finalmente, se inicia el procesado de audio mediante el método `start()` de la clase `AudioContext`.

```
// La clase AudioContext proporciona el acceso a las entradas y salidas de audio
```

```
AudioContext ac;

void setup() {
  RuidoBlanco ug;            // Declara una UGen para generar ruido blanco
  size(400, 300);            // Define el tamaño de la ventana gráfica
  ac = new AudioContext();   // Crear AudioContext asociado a las I/O por default
  ug = new RuidoBlanco(ac, 2);  // Crear la UGen para generar ruido con 2 canales
  ac.out.addInput(ug);       // Conectar nuestro UGen a la salida de audio
  ac.start();                // Iniciar el procesado en tiempo real
}
```

2.5.2 Processing y Minim

De manera similar a la librería Beads, la instalación de la librería Minim se realiza después de haber instalado Processing 3, a través de la opción `Agregar herramienta...` del menú `Herramientas`, y reiniciando Processing después de haber instalado la librería. Así mismo, Minim también está basado en Unidades Generadoras (UGens) y contiene un gran número de UGens para síntesis, procesado y análisis de audio.

Existen, sin embargo, dos diferencias importantes en la manera en que Minim procesa el audio con respecto a Beads. La primera es que, en su forma mas básica, las UGens de Minim procesan solamente una muestra a la vez (en lugar de un bloque, como lo hace Beads). La segunda es que el procesamiento se realiza en el ciclo de la función `draw()` de un sketch de Processing, por lo que es necesario implementar la función `draw()` aunque esta no haga nada.

Para esta práctica, el único método de la clase `UGen` que nos interesa implementar en nuestra UGen es el método callback `uGenerate()`, que recibe como único argumento un arreglo de floats con una muestra por canal (el tamaño del arreglo es igual al número de canales). El mecanismo es muy simple: los datos del arreglo deben ser reemplazados con las muestras de salida. El número de canales de un UGen en Minim es dinámico; es decir, que se adapta de acuerdo a los objetos a los cuales el UGen esté conectado. Lo mejor es obtener siempre el número de canales en el momento mediante el método `channelCount()`.

Con base en lo anterior, es fácil implementar una UGen para generar ruido blanco en Minim como sigue:

```
public class RuidoBlanco extends UGen {
  // El método uGenerate debe calcular una muestra por cada canal
  protected void uGenerate(float[] channels) {
    int cc = channelCount();
    for (int c = 0; c < cc; c++) {
      channels[c] = random(-1, 1);
    }
  }
}
```

Además, la clase `UGen` contiene diversos métodos (los cuales serán heredados por nuestra UGen) de gran utilidad. Por ejemplo, es posible conocer el número de canales y la frecuencia de muestreo para cualquier objeto UGen mediante las funciones `channelCount()` y `sampleRate()`, respectivamente. También es posible conectar las salidas un UGen con otro UGen mediante los métodos `patch()` o `addInput()`.

Finalmente, la inicialización del sistema de audio requiere, en este caso, dos objetos. Uno de clase `Minim`, que será el que proporciona acceso a las entradas y salidas físicas de audio, y otro de clase `AudioOutput`, que será el que se asocie con la salida de audio, y que se conectará con nuestra UGen. El resto del código queda como sigue:

```
// Para usar las clases de la librería Minim es necesario importarla
import ddf.minim.*;

// La clase Minim proporciona el acceso a las entradas y salidas de audio
Minim minim;

// La clase AudioOutput se asocia a una salida de audio física
AudioOutput out;

void setup() {
  RuidoBlanco ug;           // Declaramos una UGen para generar ruido
  size(400, 300);           // Se define el tamaño de la ventana gráfica
  minim = new Minim(this);  // Crea el objeto a partir para acceder a la I/O de audio
  out = minim.getLineOut(); // Obtiene la salida de audio por default (estereo)
  ug = new RuidoBlanco();   // Crear un UGen para generar o procesar audio
  ug.patch(out);            // Conectar nuestro UGen a la salida de audio
}

// Es necesario implementar la función draw() para realizar el procesado de audio
void draw() {
}
```

Como último comentario, el método `getLineOut()` de la clase Minim puede recibir parámetros adicionales para establecer el número de canales (`Minim.MONO` o `Minim.STEREO`), el tamaño del buffer (en muestras por canal), la frecuencia de muestreo, y el formato de las muestras (número de bits por muestra). Consulte la documentación de Minim para mayor información.

2.5.3 GNU C++ y PortAudio

La librería PortAudio es una librería escrita en lenguaje C cuyos principales objetivos son la simplicidad y la portabilidad. PortAudio se enfoca únicamente en proporcionar un enlace con las entradas y salidas físicas de audio a través de funciones *callback*, que el programador se encarga de escribir. Los pasos necesarios para instalar la librería dependen de la plataforma (Windows, Mac OSX,

Linux) y el compilador, por lo que no es posible abarcar todas las posibilidades en este documento. Se recomienda al alumno leer la guía de instalación en el sitio web de PortAudio.

Las prácticas de este manual se han desarrollado utilizando el entorno de programación Code::Blocks con el compilador MinGW integrado, bajo el sistema operativo Windows. Los ejemplos se compilan y ejecutan en modo de consola, a diferencia de las implementaciones en Processing que se ejecutan en una ventana gráfica. Para poder disponer de funciones básicas para la interfaz con el usuario, los ejemplos en C++ utilizan funciones de entrada y salida a consola (Console I/O) que se encuentran en el archivo de encabezado `conio.h`, las cuales solamente están disponibles bajo Windows. Usuarios de C++ en Linux o Mac OSX tendrán que buscar alguna alternativa (por ejemplo, las librerías Termios, Curses o NCurses).

PortAudio carece de una arquitectura similar a las UGen (unidades generadoras) de las librerías Beads y Minim. Sin embargo, no es muy difícil implementar una clase en C++ que tenga funcionalidad similar a las UGen de Beads, de manera que podamos utilizar un enfoque similar en todas las prácticas, independientemente del lenguaje con el que se implementen. En el Apéndice B se presenta el listado de la clase `UGen` para C++ que utilizaremos en los ejemplos, así como una descripción general del funcionamiento de la misma. Por ahora, consideraremos simplemente que uno puede crear nuevas UGen heredando esta clase y reescribiendo el constructor y el método `processBuffer()`. El constructor toma uno o dos argumentos especificando el número de canales de salida y (opcionalmente) el número de entradas del UGen. Por su parte, el método `processBuffer()` no toma argumentos y se llama de manera automática cuando el buffer de salida requiere ser calculado; para hacer esto simplemente se obtiene un apuntador al arreglo que corresponde al buffer mediante el método `getBuffer()` y se llena el buffer intercalando las muestras de los distintos canales; es decir que un las señales de salida para todos los canales se guardan en un solo arreglo unidimensional. El número de canales de salida es el que se especifica como primer argumento en el constructor y puede obtenerse en cualquier momento a través del método `getNumOutputs`. El resto de los métodos de la clase `UGen` se estudiarán en las prácticas siguientes.

Para utilizar esta clase en sus proyectos es necesario incluir el archivo de encabezado `ugen.h` y agregar el archivo fuente `ugen.cpp` al proyecto. O bien, si el proyecto consta de un solo archivo fuente, se puede incluir directamente la clase mediante la directiva `#include "ugen.cpp"`

Por ejemplo, la implementación de una UGen que genere ruido blanco podría ser como sigue:

```
class RuidoBlanco : public UGen {
public:
    RuidoBlanco(int outs) : UGen(outs) { }

    void processBuffer() {
        float *buffer = getBuffer();
```

```
        int n = getBufferSize() * getNumOutputs();
        for (int i = 0; i < n; i++) {
            buffer[i] = 2.0 * rand() / RAND_MAX - 1;
        }
    }
};
```

El constructor de la clase `RuidoBlanco` simplemente llama al constructor de la clase padre (`UGen`) especificando el número de canales de salida (este módulo carece de entradas), mientras que el método `processBuffer()` obtiene un apuntador al arreglo que representa el buffer de salida y lo llena con números aleatorios entre -1 y 1. El número de muestras a generar es igual al tamaño del buffer multiplicado por el número de canales de salida. Recuerde que las muestras están intercaladas por canales; es decir, si el UGen tiene N canales de salida, entonces los primeros N datos del buffer corresponden a la primer muestra para cada uno de los N canales, los siguientes N datos corresponden a la segunda muestra, etc.

Para poder escuchar la salida de un UGen en tiempo real es necesario implementar la función callback de PortAudio, la cual debe tener el siguiente encabezado:

```
static int paCallback(
    const void *inputBuffer,
    void *outputBuffer,
    unsigned long framesPerBuffer,
    const PaStreamCallbackTimeInfo *timeInfo,
    PaStreamCallbackFlags statusFlags,
    void *userData
);
```

donde los argumentos de la función son los siguientes:

- `const void *inputBuffer` - Un arreglo genérico que contiene la señal de entrada. Es necesario hacer un typecast al tipo de datos que corresponda con los parámetros definidos a la hora de inicializar el audio. Otra consideración que se debe tomar en cuenta es que en el caso de señales multi-canal (e.g., señales estéreo), las muestras van intercaladas por canales.

- `void *outputBuffer` - Un arreglo genérico donde se almacenarán las muestras para la señal de salida. Al igual que `inputBuffer`, las muestras para distintos canales están intercaladas, y es necesario hacer un typecast al tipo correcto de datos.

- `unsigned long framesPerBuffer` - El número de muestras que contienen los buffers de entrada y salida, por cada canal.

- `const PaStreamCallbackTimeInfo *timeInfo` - Esta es una estructura de datos que contiene información sobre el tiempo (en segundos) en el que

se adquiere, procesa y reproduce el bloque actual. No utilizaremos esta información en las prácticas de nivel básico.

- `PaStreamCallbackFlags statusFlags` - Contiene información sobre el estado del sistema de audio. No se utilizará en ninguna de las prácticas.
- `void *userData` - Un apuntador genérico que el programador puede utilizar para pasar información adicional a la función callback. Será de gran utilidad para proporcionar a la función callback un apuntador a la UGen que generará la señal de salida.

No se preocupe si no tiene claro el propósito de todos los argumentos de la función callback. Recuerde que usted nunca llamará explícitamente a esta función, sino que la función será llamada automáticamente por el sistema de audio, y recibirá todos los argumentos necesarios para que usted utilice solo aquellos que le convengan.

Para nuestros propósitos, la función callback podría quedar así:

```
static int paCallback(const void *inputBuffer, void *outputBuffer,
        unsigned long framesPerBuffer,
        const PaStreamCallbackTimeInfo *timeInfo,
        PaStreamCallbackFlags statusFlags,
        void *userData) {
    if (!userData) return 0;
    UGen *ugen = (UGen *)userData;
    if (ugen->getNumOutputs() != 2) return 0;
    ugen->update();
    memcpy(outputBuffer, ugen->getBuffer(), framesPerBuffer * 2 * sizeof(float));
    UGen::tick();
    return 0;
}
```

Esta función verifica en primer lugar que el apuntador `userData` sea válido, y posteriormente lo reinterpreta como un apuntador a una UGen (la cual especificaremos mas adelante), y se verifica que la UGen tenga exactamente dos canales de salida, ya que se conectará con la salida estéreo física. Posteriormente se llama al método `update()` de la UGen para actualizar su buffer de salida; el método `update()` llamará, en caso de ser necesario, al método `processBuffer()` que nosotros hayamos implementado. Una vez calculado el buffer de salida de la UGen, lo copiamos al buffer de salida de PortAudio, el cual también debe contener las muestras intercaladas por canales; por lo tanto, podemos simplemente usar la función `memcpy()` para copiar un arreglo en otro. Finalmente, es necesario llamar al método `tick()` de la clase `UGen` para indicar que el buffer ha sido procesado; este es un método estático por lo que no se llama desde un objeto de clase `UGen`, sino a través de su nombre completo: `UGen::tick()`.

La inicialización del audio se realiza en varias etapas: primero se llama a la función `Pa_Initialize()` para iniciar el sistema de audio; posteriormente se llama a la función `Pa_OpenDefaultStream()` para abrir los canales de entrada

y/o salida de audio, así como para fijar ciertos parámetos como la frecuencia de muestreo, el tamaño de bloque, y el formato de las muestras. Finalmente, se llama a la función `Pa_StartStream()` para iniciar el procesado de los bloques. Cada una de estas funciones devuelve un resultado de tipo `PaError` que indica si la función tuvo éxito o si ocurrió algún error. Podemos englobar todo este proceso en una sola función (que podrá ser reutilizada en futuras prácticas):

```
void inicializaAudio(PaStream **stream, UGen *ugen) {
    PaError err;

    // Inicializar PortAudio
    err = Pa_Initialize();
    if (err != paNoError) {
        cerr << "No se pudo inicializar PortAudio" << endl;
        exit(-1);
    }

    // Inicializar el flujo de audio con entrada y salida estereo
    err = Pa_OpenDefaultStream(stream, 2, 2, paFloat32,
     UGen::getSampleRate(), UGen::getBufferSize(), paCallback, ugen);
    if (err != paNoError) {
        cerr << "No se pudo inicializar el dispositivo de audio" << endl;
        exit(-1);
    }

    // Iniciar el proceso en tiempo real
    err = Pa_StartStream(*stream);
    if (err != paNoError) {
        cerr << "No se pudo iniciar la reproduccion" << endl;
        exit(-1);
    }
}
```

La función `inicializaAudio` recibe como argumentos un doble apuntador a una estructura `PaStream` a través de la cual podremos manipular el sistema de audio y un apuntador a la UGen a partir de la cual se generará el audio de salida. Cabe resaltar que esta UGen debe tener exactamente dos canales de salida.

De particular interés es la llamada a la función `Pa_OpenDefaultStream`, la cual abre los canales de entrada y salida de audio definidos por defecto en el sistema operativo. A esta función se le pasan los siguientes argumentos (en orden): un doble apuntador a una estructura `Pa_Stream` que se asociará con la entrada y salida física de audio, el número de canales de entrada y de salida (en este caso dos y dos), el formato de las muestras de la señal (`paFloat32`), la frecuencia de muestreo, el tamaño de bloque, el nombre de la función callback, y un apuntador genérico que se pasará a la función callback como información adicional (en este caso, la UGen asociada con la salida de audio). Es importante que la frecuencia de muestreo y el tamaño de bloque coincidan con los establecidos para la clase UGen, por lo que es necesario inicializar primero la clase UGen antes de inicializar PortAudio.

Una vez que se tienen la función de inicialización y la función callback, no queda mas que implementar la función `main()`. Cabe mencionar que es necesario incluir el archivo de encabezado `portaudio.h` y enlazar el programa

con la librería `libportaudio` que corresponda al sistema operativo en uso, así como el archivo de encabezado (o de implementación) de la clase `UGen` que se muestra en el Apéndice B.

```
#include <portaudio.h>
#include <cstdlib>
#include <cstring>
#include <iostream>
#include <conio.h>
#include "ugen.cpp"

using namespace std;

int main() {
   // Inicializa parámetros de las UGen (frecuencia de muestreo y tamaño de bloque)
   // Es importante hacer esto antes de crear objetos UGen o llamar a inicializaAudio
    UGen::setup(44100, 256);

    // Crea una instancia del UGen que genera ruido con dos canales de salida
    RuidoBlanco *ruido = new RuidoBlanco(2);

    PaStream *stream; // PaStream asociado a las entradas y salidas de audio

    // Inicializa el sistema de audio y obtiene el PaStream; asocia un UGen a la salida
    inicializaAudio(&stream, ruido);
    _getch();               // Espera a que se pulse una tecla
    Pa_CloseStream(stream); // Concluye el procesado del audio
    Pa_Terminate();         // Cierra el sistema de audio
    delete ruido;
    return 0;
}
```

2.6 Evaluación y reporte de resultados

1.- Llene la siguiente tabla con los valores solicitados:

Lenguaje utilizado	
Librería utilizada	
Función que llena el buffer	
Tamaño del buffer	
Frecuencia de muestreo	
Latencia (en ms)	

2.- Implemente una UGen que genere ruido blanco únicamente en el canal derecho, y silencio en el canal izquierdo.

3.- Describa, brevemente, una ventaja y una desventaja de cada una de las plataformas utilizadas.

C + PortAudio Ventaja	
C + PortAudio Desventaja	
Processing + Beads Ventaja	
Processing + Beads Desventaja	
Processing + Minim Ventaja	
Processing + Minim Desventaja	

2.7 Retos

Implemente una UGen para generar ruido blanco de manera que el ruido se reproduzca solamente por el canal izquierdo o el canal derecho, alternando el canal en cada segundo (i.e., un segundo por el canal izquierdo y el siguiente segundo por el derecho, etc.).

2.8 Conclusiones

Redacte en el recuadro sus conclusiones acerca de los conceptos aprendidos, las actividades realizadas y los objetivos planteados en esta práctica. Si lo desea, puede considerar las preguntas de apoyo que se muestran abajo.

Preguntas de apoyo

- Cómo funciona, a grandes rasgos, un sistema de procesado de audio en tiempo real?
- Cuales son las frecuencias de muestreo típicas en un sistema de audio digital?
- Qué es la latencia y qué la causa?
- Qué relación existe entre el tamaño del buffer de audio y la responsividad del sistema?
- Qué es una función callback y cuál es su función en el desarrollo de aplicaciones de audio?

Capítulo 3

Control de amplitud

Nombre del estudiante	Calificación

3.1 Relación con los programas de estudio

Materia	Unidad y tema
Procesamiento Digital de Señales (IE / IT / IB)	1.4 - Sistemas lineales e invariantes en el tiempo
Procesamiento de Señales de Audio (IE)	1.1 - Medidas y unidades de amplitud y frecuencia 1.2 - Superposición de señales 1.3 - Sistemas lineales e invariantes en el tiempo

3.2 Introducción

De acuerdo con diversos autores, las propiedades fundamentales de un sonido, desde un punto de vista perceptual, son las siguientes [7]:

- **Tono.-** El tono es la propiedad que permite a nuestro sistema auditivo distinguir los sonidos en términos de qué tan graves o agudos son. Desde un punto de vista físico y matemático, el tono está relacionado con la *frecuencia fundamental* de una onda periódica.

- **Intensidad.-** La intensidad es la propiedad que describe la fuerza o volumen de un sonido. Está relacionada con la amplitud de la onda que representa a ese sonido, o desde un punto de vista estrictamente físico, con la magnitud de los cambios en la presión aérea causados por una onda acústica.

- **Timbre.-** El timbre es la propiedad que nos permite distinguir la fuente del sonido, aún cuando el tono y la intensidad sean similares entre distintas fuentes. Por ejemplo, es posible distinguir el sonido de un violín, de una flauta y de un piano, aunque toquen la misma nota al mismo volumen, debido a sus respectivos timbres. También es posible distinguir las voces de distintas personas en una conversación grabada, donde no se cuenta con la información visual sobre quién está hablando en qué momento. Matemáticamente, el timbre está relacionado con el *espectro* y con la *forma de onda* de un sonido.

- **Duración.-** La duración de un sonido es el tiempo transcurrido desde que el sonido se percibe por primera vez hasta que éste cesa o cambia lo suficiente como para ser considerado un sonido distinto. Una correcta detección de la duración de un sonido es de suma importancia para diversas aplicaciones, como el reconocimiento del habla y la decodificación en ciertos sistemas de transmisión (e.g., clave Morse).

Existen otras propiedades no fundamentales del sonido como la textura, la cual está relacionada con la forma en que interactúan múltiples fuentes de sonido [8] (considere, por ejemplo, la diferencia entre el sonido de un mercado y el de una fiesta infantil), y como la localización o dirección de donde proviene el sonido [9], sin embargo, el análisis y/o modelado de estas propiedades va mucho mas allá de los objetivos de este manual. Otro aspecto a considerar es que algunas de las propiedades de un sonido pueden cambiar a lo largo de la existencia del mismo. Por ejemplo, si uno pulsa la cuerda de una guitarra o la tecla de un piano, el sonido iniciará con su mayor intensidad (una fase conocida como *ataque*), la cual disminuirá rápidamente con el tiempo.

El proceso mas simple y común que podemos realizar a una señal de audio es controlar su intensidad, que como ya mencionamos, está directamente relacionada con la amplitud de la señal. Esto puede modelarse fácilmente como el siguiente sistema lineal e invariante en el tiempo (LIT) [10]:

$$y[n] = g \cdot x[n],$$

donde $x[n]$ y $y[n]$ son, respectivamente, las señales de entrada y salida del sistema, y g es el factor de ganancia. A este sistema se le conoce simplemente como *amplificador*, aún cuando se tenga $|g| < 1$ y el sistema en realidad atenúe la señal de entrada.

En la ecuación anterior, el factor de ganancia g es adimensional, pero es común también expresarlo en decibeles (db). La conversión a decibeles está dada por [10]:

$$g_{db} = 20 \log_{10} g.$$

Note que 0 db corresponde a una ganancia unitaria (la señal de salida tendrá la misma amplitud que la de entrada), mientras que una ganancia negativa en decibeles corresponde a una atenuación.

3.3 Objetivos didácticos

- Ayudar en la comprensión de los siguientes conceptos básicos: amplitud, factor de ganancia, sistema lineal e invariante en el tiempo.
- Implementar un sistema básico para procesar señales de audio implementando módulos que incorporen canales de entrada tanto como de salida.
- Comprender las ventajas y el uso de un enfoque modular que permita conectar diversos bloques de procesamiento entre sí.

3.4 Material

- Una computadora con alguna de las siguientes combinaciones de software instalado:
  - Processing y Beads
  - Processing y Minim
  - GNU C++ (e.g., MinGW) y PortAudio
- Bocinas o audífonos estéreo

3.5 Procedimiento

Para esta práctica se sugiere partir de la implementación realizada en la práctica anterior. A grandes rasgos, los pasos a seguir son los siguientes:

1. Implementar una nueva UGen (llamada *Amplificador*) que tome una señal de entrada y calcule la señal de salida multiplicando la entrada por un factor de ganancia. Además, la clase debe incluir métodos *set* y *get* para manipular el factor de ganancia.
2. Conectar una UGen de clase `RuidoBlanco` a la entrada de una UGen `Amplificador`, y la salida del amplificador a la salida física de audio. De esta manera podremos controlar desde nuestro programa el nivel o intensidad del ruido que se produce.
3. Implementar una interfaz de usuario sencilla para controlar el volumen del ruido y mostrar el factor de ganancia en decibeles.

La implementación de cada paso dependerá nuevamente de la arquitectura elegida, aunque dado que ya se cuenta con el marco de desarrollo establecido en la práctica anterior, se apreciará que las tres implementaciones siguen la misma lógica. En todos los casos, crearemos una nueva clase llamada `Amplificador`, heredada de `UGen`, la cual tendrá el mismo número de canales de entrada y de salida. Incluiremos en la clase una variable de punto flotante llamada `ganancia`

con sus respectivos métodos *set* y *get* (prácticamente idénticos en las tres implementaciones), e implementaremos la función callback que calcula la señal de salida. Posteriormente, en la función principal del programa, crearemos un objeto `RuidoBlanco` y un objeto `Amplificador` y los conectaremos entre sí para obtener ruido amplificado. Finalmente agregaremos la interfaz de usuario para controlar el volumen del ruido.

3.5.1 Processing y Beads

Recordemos que en Beads, las UGen deben implementar el método callback `calculateBuffer()` dentro del cual se tienen disponibles los arreglos bidimensionales `bufIn` y `bufOut` para las señales de entrada y salida, respectivamente. Además, utilizaremos la forma del constructor que permite indicar el número de canales de salida tanto como de entrada. Teniendo esto en cuenta, la implementación de la clase `Amplificador` queda como sigue:

```
// UGen para control básico de amplitud
public class Amplificador extends UGen {
    float ganancia;

    // constructor: define el número de canales de entrada y salida
    public Amplificador(AudioContext context, int canales) {
      super(context, canales, canales);
      ganancia = 0;
    }

    // métodos get y set para manipular la ganancia
    float ganancia() { return ganancia; }
    void setGanancia(float g) { ganancia = g; }

    // método callback para calcular la salida
    public void calculateBuffer() {
      for (int c = 0; c < getOuts(); c++) {
        float[] in = bufIn[c];
        float[] out = bufOut[c];
        for (int i = 0; i < bufferSize; i++) {
          out[i] = ganancia * in[i];
        }
      }
    }
}
```

La conexión de una UGen a la entrada de otra UGen se realiza llamando al método `addInput` de la UGen receptora. Por ejemplo, para conectar la UGen `ruido` a la entrada de la UGen `amp`, utilizamos la instrucción `amp.addInput(ruido);`. Para controlar la ganacia, aprovecharemos las funciones de Processing para control a través del mouse, de manera que podamos controlar la ganancia moviendo

el mouse de izquierda a derecha sobre la ventana gráfica de nuestro programa. Para poder utilizar las funciones de interfaz de usuario será necesario implementar también la función `draw()`, aún cuando esta no haga nada.

Con base en lo anterior, el resto del programa puede quedar como sigue:

```
// Declaración de objetos de acceso global
AudioContext ac;
RuidoBlanco ruido;
Amplificador amp;

void setup() {
  size(400, 300);                  // tamaño de la ventana gráfica
  ac = new AudioContext();         // Crear AudioContext asociado a las I/O por default
  ruido = new RuidoBlanco(ac, 2);  // Crear un UGen para generar ruido
  amp = new Amplificador(ac, 2);   // Crear un UGen para controlar la amplitud
  amp.addInput(ruido);             // Conectar la fuente de ruido al amplificador
  ac.out.addInput(amp);            // Conectar el amplificador a la salida de audio
  amp.setGanancia(0);              // Establecer la ganancia inicial
  ac.start();                      // Iniciar el procesado en tiempo real
}

// función callback que se llama automáticamente cuando se mueve el mouse
void mouseMoved() {
  amp.setGanancia((float)mouseX / width);   // ganancia entre 0 y 1
}

void draw() { }
```

3.5.2 Processing y Minim

En Minim, las señales de entrada de una UGen deben declararse como miembros de la clase de tipo `UGenInput`, la cual es también una clase heredada de `UGen`. Recordemos que el método callback `uGenerate(float[] channels)` de las UGen de Minim solamente procesa una muestra a la vez (no un bloque). Dentro de este método, uno puede obtener las muestras de las señales de entrada llamando al método `getLastValues()` de los objetos `UGenInput`, para luego realizar los cálculos correspondientes y escribir las muestras de la señal de salida en el arreglo `channels`.

Existen dos tipos de objetos de clase `UGenInput`: aquellos que representan señales de audio, los cuales siempre tienen el mismo número de canales que el UGen anfitrión, y aquellos que representan señales de control, los cuales constan de un solo canal. El tipo de entrada se define en el constructor de `UGenInput`, el cual recibe como argumento una de dos posibles constantes: `UGen.InputType.AUDIO` o `UGen.InputType.CONTROL`. En este caso, el amplificador estará diseñado para procesar señales de audio con uno o mas canales. En prácticas posteriores se estudiará el uso de señales de control.

De esta manera, la clase `Amplificador` para Minim puede quedar así:

```
// UGen para control básico de amplitud
public class Amplificador extends UGen {
  float ganancia;
  UGenInput audio_in;

  Amplificador() {
    super();
    ganancia = 0;
    audio_in = new UGenInput(UGen.InputType.AUDIO);
  }

  float ganancia() { return ganancia; }
  void setGanancia(float g) { ganancia = g; }

  protected void uGenerate(float[] channels) {
    int cc = channelCount();
    float[] in = audio_in.getLastValues();
    for (int c = 0; c < cc; c++) {
      channels[c] = ganancia * in[c];
    }
  }
}
```

El constructor de `Amplificador` simplemente llama al constructor de `UGen` y posteriormente inicializa la ganancia y la entrada de audio. El método `uGenerate()` obtiene primero el número de canales y el arreglo de muestras de la entrada de audio, para luego calcular las muestras de salida.

Únicamente resta por implementar la inicialización de las UGen y la conexión entre ellas. A diferencia de Beads, las UGen en Minim se conectan llamando al método `patch()` desde el objeto emisor, pasando como argumento el objeto receptor; de manera que si deseamos conectar la salida de la UGen `ruido` a la entrada de la UGen `amp`, la instrucción que debe escribirse es `ruido.patch(amp);` Por lo tanto, el resto del programa puede quedar así:

```
// La clase Minim proporciona el acceso a las entradas y salidas de audio
Minim minim;

// La clase AudioOutput se asocia a una salida de audio física
AudioOutput out;

// Declaración de UGens de acceso global
RuidoBlanco ruido;
Amplificador amp;

void setup() {
```

```
  size(400, 300);
  minim = new Minim(this);   // Crea el objeto Minim a partir del cual se obtienen entradas
  out = minim.getLineOut();  // Obtiene la salida de audio por default (estereo)
  ruido = new RuidoBlanco(); // Crea una UGen para generar ruido
  amp = new Amplificador();  // Crea una UGen para controlar la amplitud
  ruido.patch(amp);          // Conecta la fuente de ruido al amplificador
  amp.patch(out);            // Conecta el amplificador a la salida de audio
  amp.setGanancia(0);        // Fija la ganancia inicial
}


// función callback que se llama automáticamente cuando se mueve el mouse
void mouseMoved() {
  amp.setGanancia((float)mouseX / width);   // ganancia entre 0 y 1
}

void draw() {
}
```

3.5.3 C++ y PortAudio

Para la implementación en C++ tomaremos ventaja de la clase `UGen` que se muestra en el Apéndice B, en la cual cada UGen puede contar con cero o mas entradas cuyas señales provienen de otros UGen. El mecanismo es muy simple: se define el número de entradas como segundo argumento en el constructor de `UGen` (recuerde que el primer argumento del constructor es el número de canales de salida), y posteriormente se asigna un UGen a cada entrada mediante el método `setInput()`. Dentro del método callback `processBuffer()`, se puede acceder a cada entrada a través del método `getInput()`, el cual devuelve un apuntador al UGen asociado a una cierta entrada, o bien, devuelve `NULL` si la entrada no ha sido asignada (es responsabilidad del programador verificar que el valor devuelto por `getInput()` sea distinto de NULL). Una vez que se tiene un apuntador al UGen asociado a la entrada, se puede acceder a la señal de ese UGen mediante el método `getBuffer()`.

Un aspecto a tomar en cuenta es que todo UGen puede tener mas de un canal de salida, por lo que si se desea implementar un módulo para procesar, por ejemplo, una señal estéreo, el módulo tendrá dos canales de salida, pero solo requiere una entrada, a la cual se asignará una UGen que también tenga salida estéreo. Es responsabilidad del programador verificar siempre que el número de canales de salida de las UGen asociadas a las entradas de otra UGen cumpla con las condiciones que requiera la UGen anfitrión.

Con estas consideraciones, procedemos a implementar la clase `Amplificador`:

```
class Amplificador : public UGen {
public:
    float ganancia;
```

```
    Amplificador(int canales) : UGen(canales, 1) {
        ganancia = 0;
    }

    float getGanancia() {
        return ganancia;
    }

    void setGanancia(float g) {
        ganancia = g;
    }

    void processBuffer() {
        UGen *ugen = getInput(0);
        if (ugen == NULL) return;
        if (ugen->getNumOutputs() != getNumOutputs()) return;
        float *in = ugen->getBuffer();
        float *out = getBuffer();
        int n = getBufferSize() * getNumOutputs();
        for (int i = 0; i < n; i++) {
            out[i] = in[i] * ganancia;
        }
    }
};
```

Al igual que en las implementaciones para Beads y Minim, la clase `Amplificador` en C++ incluye una variable para la ganancia, así como funciones *set* y *get* para manipularla. Además, se implementa un constructor que llama al constructor de `UGen` para definir el número de canales de salida y una entrada. También se agrega un método `setInput()` para asignar una UGen a la entrada. Finalmente, el método callback `processBuffer()` obtiene la UGen asignada a la entrada (asegurándose de que exista y que tenga el mismo número de canales que el amplificador) y calcula el buffer de salida del amplificador multiplicando cada muestra del buffer de entrada por la ganancia.

Dado que las implementaciones en C++ se realizarán en modo consola, utilizaremos una interfaz mas rudimentaria basada en el teclado, utilizando las teclas del '0' al '9' para controlar la ganancia del amplificador en pasos discretos y la tecla ESC para terminar el programa. De esta manera, la función `main()` puede quedar como sigue:

```
int main() {
    UGen::setup(44100, 256);

    RuidoBlanco *ruido = new RuidoBlanco(2);
    Amplificador *amp = new Amplificador(2);
```

```
    amp->setInput(ruido);
    amp->setGanancia(0);

    PaStream *stream;
    initializeAudio(&stream, amp);

    char c = 0;
    do {
        if (_kbhit()) {
            c = _getch();
            if (c >= '0' && c <= '9') {
                float g = (float)(c - '0') / 9;
                amp->setGanancia(g * g);
            }
        }
    } while (c != 27);

    Pa_CloseStream(stream);
    Pa_Terminate();
    delete ruido;
    delete amp;
    return 0;
}
```

En la función `main()` se inicializan primero los parámetros generales de las UGen (frecuencia de muestreo y tamaño de bloque). Es importante hacer esto antes de crear objetos UGen, ya que el constructor de UGen reserva la memoria necesaria para el buffer de salida (cuyo tamaño depende del tamaño de bloque). También es importante inicializar la clase `UGen` antes de llamar a `inicializaAudio()`, ya que esta función toma los parámetros de `UGen` para inicializar PortAudio. Posteriormente se crean los objetos `ruido` y `amp`, ambos con dos canales de salida, y se conecta la fuente de ruido a la entrada del amplificador mediante la instrucción `amp->setInput(ruido);`.

Luego de inicializar el sistema de audio, se entra en un ciclo infinito dentro del cual se detecta si se ha presionado alguna tecla mediante `_kbhit()`, en cuyo caso se lee el código correspondiente a la tecla usando la función `_getch()`. Si el código corresponde a un dígito numérico entonces se ajusta la ganancia siguiendo una curva cuadrática. Si el código corresponde a la tecla ESC, el ciclo se rompe y el programa concluye.

3.6 Evaluación y reporte de resultados

1.- Describa cómo diseñar una UGen que tenga una entrada de audio (además de una salida de audio) en la plataforma de su elección.

2.- Describa como interconectar la salida de una UGen hacia la entrada de otra UGen en la plataforma de su elección.

3.- Modifique el programa realizado en esta práctica para que muestre en todo momento el factor de ganancia expresado en decibeles (puede usar la función `text` de Processing o el objeto `cout` de C++ para imprimir texto). Considere que si el factor de ganancia es cero, la pantalla debe mostrar $-\infty$ db.

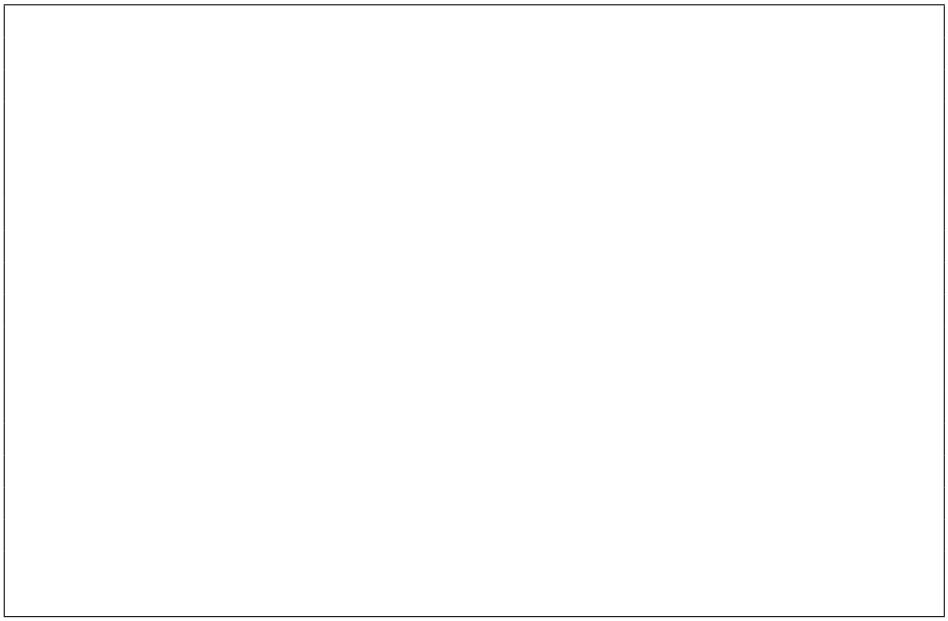
4.- Diseñe una nueva UGen llamada `Suma2` que tome dos señales de entrada (cada una de ellas con n canales) y calcule como salida la suma de ambas, canal a canal.

3.7 Retos

Otro de los bloques de gran utilidad en el procesamiento de audio es un *mezclador* de N canales, el cual toma N entradas x_k, $k = 1, \ldots, N$ (posiblemente multicanal) y un factor de ganancia g_k para cada una de las entradas, y calcula como salida la suma pesada de las entradas $y = g_1 \cdot x_1 + g_2 \cdot x_2 + \ldots + g_N \cdot x_n$. Implemente una mezcladora como una nueva clase heredada de UGen que reciba el número de señales de entrada como argumento del constructor, y que proporcione funciones `set` y `get` para manipular la ganancia de cada entrada.

3.8 Conclusiones

Redacte en el recuadro sus conclusiones acerca de los conceptos aprendidos, las actividades realizadas y los objetivos planteados en esta práctica. Si lo desea, puede considerar las preguntas de apoyo que se muestran abajo.



Preguntas de apoyo

- Cuáles son las propiedades fundamentales de un sonido y cuál de ellas se relaciona con la amplitud de la onda?
- Cómo se describe matemáticamente el sistema amplificador? Qué es la ganancia y cómo se representa esta en decibeles?
- Qué consideraciones hay que tener en cuenta para implementar una UGen para procesar audio? Es decir, una UGen que toma una señal de entrada y genera una señal de salida.

Capítulo 4

Entrada de audio y retroalimentación

Nombre del estudiante	Calificación

4.1 Relación con los programas de estudio

Materia	Unidad y tema
Procesamiento Digital de Señales (IE / IT / IB)	1.4 - Sistemas lineales e invariantes en el tiempo 1.5 - Convolución y sus propiedades 2.1 - Respuesta de un sistema LIT a una exponencial compleja 2.2 - Transformada de Fourier 2.3 - Propiedades de la Transformada de Fourier
Procesamiento de Señales de Audio (IE)	1.5 - Sistemas lineales e invariantes en el tiempo 1.6 - Convolución y respuesta al impulso 1.9 - Respuesta en frecuencia

4.2 Introducción

De acuerdo con McLoughlin [11], podemos clasificar los sistemas de audio digital en tres grandes clases :

- **Sistemas de síntesis y/o generación de audio.-** Estos sistemas generan una o mas señales de audio partiendo de información que no es de

origen acústico, sino de otra índole. Por ejemplo: síntesis de voz a partir de texto escrito, síntesis de sonidos musicales a partir de controladores físicos (teclados, perillas, pads, etc.), sistemas de monitoreo basados en audio, y representación auditiva de datos (sonificación).

- **Sistemas de análisis y/o reconocimiento de audio.-** Estos sistemas toman como entrada información acústica (es decir, señales de audio digitalizadas) para analizarla, caracterizarla, y devolver como salida información descriptiva, pero de índole distinto al acústico, acerca del sonido de entrada. Ejemplos: sistemas de reconocimiento de voz y dictado automático, interfaces humano-computadora controladas por voz o gestos mecánicos, sistemas de localización basados en ondas acústicas (sonar, localización binaural), análisis y clasificación de señales bioacústicas (fonocardiograma, fonomiograma, ronquidos, sonidos de animales, etc.).

- **Sistemas de procesamiento, almacenamiento y/o transmisión de audio.-** Estos sistemas toman como entrada una o más señales de audio y generan como salida otra señal de audio dependiente de la señal de entrada. La señal de salida puede obtenerse simplemente aplicando algún proceso a la señal de entrada, o también puede sintetizarse a partir de información obtenida a partir del análisis de la señal de entrada (lo cual se conoce como una *representación intermedia* o *codificación*). Ejemplos de estos sistemas son: ecualizadores, procesadores de efectos artificiales (reverberación, eco, cambio de tono, distorsión, etc.), procesadores de dinámica (compresores, compuertas de ruido, etc.), codecs (codificadores / decodificadores) de audio (MP3, FLAC, etc.).

Existe además una cuarta clase de sistemas, comúnmente llamados *sistemas de medición* o *analizadores*, los cuales permiten caracterizar a otros sistemas de procesamiento. Un analizador primero genera señales de audio que serán enviadas al sistema que se desea caracterizar para excitarlo, y posteriormente captura la salida del sistema para analizarla. En este caso, la salida del analizador es completamente sintética y no depende de la entrada, sino que la entrada del analizador será una versión procesada (por el sistema a caracterizar) de la salida. Por lo general, las señales emitidas por el analizador suelen ser simples (ruido blanco u ondas senoidales con frecuencia constante o con un barrido de frecuencia), lo cual simplifica el análisis.

Tres de estas cuatro clases requieren que el sistema cuente con una entrada física de audio. Por este motivo, es importante conocer de qué manera podemos adquirir audio en tiempo real bajo la plataforma elegida. Las tres plataformas contempladas en este manual (Beads, Minim y PortAudio) cuentan con mecanismos muy simples para la adquisición de audio, pero completamente distintos entre sí, por lo que vale la pena estudiar cada uno a fondo.

La mayoría de las computadoras modernas cuenta con un micrófono integrado (particularmente las computadoras portátiles y algunos modelos de escritorio), y posiblemente una entrada de audio auxiliar, la cual es estéreo. En

el caso de la entrada auxiliar, es posible conectar un smartphone o un reproductor de MP3 portátil mediante un cable de audio estéreo de 3.5 mm común y corriente.

Existen también en el mercado diversas interfaces de audio que permiten conectar micrófonos, guitarras eléctricas y cualquier otro tipo de instrumentos. Estas pueden utilizarse también para las prácticas, aunque es posible que sea necesario configurar el sistema operativo para que las utilice como entrada y salida de audio por defecto.

4.3 Objetivos didácticos

- Complementar el marco de trabajo que se ha desarrollado hasta el momento agregando la captura de audio en tiempo real.
- Profundizar en la comprensión de la arquitectura y los mecanismos relacionados con las distintas librerías para manejo de audio en tiempo real que se utilizan en este manual.
- Experimentar con la latencia y la retroalimentación controlada, y comprender la relación entre ambas.
- Implementar una herramienta para visualizar señales simulando un osciloscopio rudimentario en ventana gráfica (Processing) o de consola (C++).

4.4 Material

- Una computadora con entrada de audio (auxiliar o micrófono) alguna de las siguientes combinaciones de software instalado:
  - Processing y Beads
  - Processing y Minim
  - GNU C++ (e.g., MinGW) y PortAudio
- Bocinas y audífonos estéreo (se sugiere contar con ambos).
- Micrófono o dispositivo para reproducir audio (smartphone, reproductor de MP3, etc)
- Cable de audio para conectar dispositivo a la entrada auxiliar de la computadora (usualmente un cable estereo con plugs de 1/8").

4.5 Procedimiento

En cualquiera de las plataformas, será necesario obtener un UGen que genere como salida la señal de audio que se adquiere en la entrada física de audio

(micrófono o auxiliar). Esta parte es sencilla, ya que ambas librerías para Processing (Beads y Minim) proporcionan las funciones necesarias para crear una UGen asociado a la entrada física de audio. En el caso de PortAudio, la señal de entrada se recibe directamente en la función callback, por lo que tendremos que copiar esta señal en el buffer de salida de una UGen genérica, para poder luego conectar esa UGen a otras.

El interés de contar con una UGen asociada a la entrada de audio radica precisamente en la posibilidad de conectar esa UGen a otras UGens que procesen la señal de entrada, para posteriormente analizarla o enviarla a la salida física (audífonos o bocinas).

En esta práctica, dado que hasta ahora solo contamos con un módulo de procesamiento (el amplificador), solamente buscaremos conectar la entrada de audio a un amplificador para controlar su volumen, y posteriormente enviar la señal resultante a la salida de audio. En caso de que la entrada de audio sea el micrófono, se recomienda inicialmente utilizar audífonos para la salida de audio, ya que el usar las bocinas de la computadora resultará en un ciclo de retroalimentación (feedback) cuando el audio de las bocinas ingrese nuevamente por el micrófono. Si bien la retroalimentación controlada puede producir resultados interesantes (ver sección de evaluación), también puede producir un ruido excesivamente molesto o incluso peligroso, que puede dañar nuestros oídos y/o el equipo.

Una vez depurado el programa, se pedirá al estudiante utilizar micrófono y bocinas para experimentar con retroalimentación controlada, ya que ésta es de gran utilidad en el desarrollo de diversos tipos de filtros, efectos y métodos de síntesis. Matemáticamente, se puede modelar la retroalimentación de la siguiente manera: la salida $y(t)$ del sistema, emitida por las bocinas, es una versión amplificada y retardada de la señal $x(t)$ capturada por el micrófono; es decir, $y(t) = g \cdot x(t-d)$, donde el retardo d depende en gran medida de la latencia del sistema. Pero el micrófono capta también el sonido emitido por las bocinas, por lo que la señal $x(t)$ es igual a la suma del audio emitido por las bocinas mas una componente adicional que corresponde a cualquier sonido $w(t)$ que no es emitido por las bocinas (e.g., la voz del usuario y/o el ruido ambiental). En otras palabras: $x(t) = y(t) + w(t)$. Combinando los dos modelos anteriores se llega a

$$\begin{aligned}
y(t) &= gx(t-d) \\
&= gw(t-d) + gy(t-d) \\
&= gw(t-d) + g\left(gw(t-2d) + gy(t-2d)\right) \\
&= gw(t-d) + g^2 w(t-2d) + g^2\left(gw(t-3d) + gy(t-3d)\right) \\
&\vdots \\
&= \sum_{k=1}^{\infty} g^k w(t-kd).
\end{aligned}$$

Lo anterior corresponde a un sistema LIT, ya que $y(t)$ puede obtenerse al

convolucionar la señal de entrada $w(t)$ con el kernel $h(t)$ dado por

$$h(t) = \sum_{k=1}^{\infty} g^k \delta(t - kd),$$

donde $\delta(t)$ es la señal impulso. Por lo tanto, la respuesta al impulso de este sistema está dada por un tren de impulsos que se van atenuando exponencialmente (cuando $|g| < 1$). Es fácil ver que la respuesta en frecuencia de este sistema es

$$H(\omega) = \sum_{k=1}^{\infty} g^k e^{-jkd\omega},$$

la cual es una función tipo peine que muestra picos pronunciados con un espaciamiento en frecuencia de $2\pi/d$ y donde g controla qué tan altos y angostos son los picos (para $g = 1$ los picos se convierten en impulsos).

Por lo tanto, cuando hablamos de *retroalimentación controlada*, nos referimos a controlar tanto el valor de g como el de d para obtener una respuesta al impulso o respuesta en frecuencia específicas para una cierta aplicación.

Finalmente, agregaremos como elemento adicional la visualización (relativamente burda) de la señal de salida simulando un osciloscopio, para lo cual se requerirá acceder directamente al buffer de la última UGen de la cadena (en este caso, el amplificador) fuera de la función callback.

De esta manera, los pasos a seguir en esta práctica son:

1. Crear una UGen asociada a la entrada física de audio.
2. Conectar esa UGen a un amplificador, y la salida del amplificador a la salida física de audio.
3. Implementar un visualizador para mostrar en la pantalla la señal de salida.
4. Apreciar de manera audible y medir la latencia del sistema.
5. Apreciar de manera audible el fenómeno de retroalimentación.

En todos los casos partiremos de los programas elaborados en la práctica anterior, modificando las funciones principales (`setup()`, `draw()` y `main()`) y agregando la funcionalidad que haga falta.

4.5.1 Processing y Beads - Entrada de audio

Para acceder a la entrada de audio en Beads simplemente hay que llamar al método `getAudioInput()` del objeto `AudioContext`, el cual devuelve un objeto `UGen` asociado a la entrada de audio. Luego podemos asignar esta UGen como entrada de otra UGen a través del método `addInput()`, como se ha hecho anteriormente.

De esta manera, es sencillo crear una cadena de audio donde ruteando la entrada de audio a través de un amplificador y luego hacia la salida de audio. Además, usaremos el mouse para controlar la ganancia del amplificador como en la práctica anterior:

```
AudioContext ac;
Amplificador amp;
UGen input;

void setup() {
  size(400, 300);

  ac = new AudioContext();        // Crear AudioContext asociado a las I/O por default
  amp = new Amplificador(ac, 2);  // Crear amplificador estereo
  input = ac.getAudioInput();     // Obtener entrada de audio
  amp.addInput(input);            // Conectar la entrada de audio al amplificador
  ac.out.addInput(amp);           // Conectar el amplificador a la salida de audio
  amp.setGanancia(0.0);           //  Iniciar en silencio (ganancia cero)
  ac.start();                     // Iniciar el procesado en tiempo real
}

void mouseMoved() {
  amp.setGanancia((float)mouseX / width);
}

void draw() {
}
```

4.5.2 Processing y Minim - Entrada de audio

Minim cuenta con una clase específicamente diseñada para la entrada de audio, llamada `LiveInput`, la cual es heredada de `UGen`. Sin embargo, la creación de un objeto de clase `LiveInput` es ligeramente complicada, ya que el constructor requiere como argumento un objeto de clase `AudioStream` que se obtiene a partir del método `getInputStream()` del objeto de clase `Minim` creado en el programa. A su vez, el método `getInputStream()` requiere como argumentos el número de canales del flujo de entrada, el tamaño de buffer, la frecuencia de muestreo y el número de bits por muestra. Todos estos parámetros pueden obtenerse directamente del objeto `AudioOutput` asociado con la salida de audio, de manera que los parámetros coincidan para los flujos de entrada y salida.

Una vez construido el objeto `LiveInput`, uno puede llamar a su método `patch()` para conectarlo hacia otro UGen. El siguiente programa ejemplifica lo anterior:

```
// Objetos de acceso global
Minim minim;
AudioOutput out;
LiveInput  input;
Amplificador amp;

void setup() {
```

```
  size(400, 300);

  minim = new Minim(this);  // Crea el objeto Minim
  out = minim.getLineOut(); // Obtiene la salida de audio por default (estereo)

  // Crea objeto AudioStream asociado a la entrada de audio
  // con mismos parámetros que la salida de audio
  AudioStream stream = minim.getInputStream(
    out.getFormat().getChannels(), out.bufferSize(),
    out.sampleRate(), out.getFormat().getSampleSizeInBits());

  input = new LiveInput(stream);  // Crea la UGen asociada a la entrada de audio
  amp = new Amplificador();       // Crea un amplificador
  input.patch(amp);               // Conecta la entrada al amplificador
  amp.patch(out);                 // Conecta el amplificador a la salida
  amp.setGanancia(0);             // Fija la ganancia inicial
}
```

4.5.3 Processing (con Beads o Minim) - Osciloscopio

Para visualizar la señal de salida, implementaremos un osciloscopio muy básico mediante una función llamada `simplescopio()`, la cual tomará como entrada un arreglo unidimensional que contendrá el buffer correspondiente a la señal que se desea visualizar, y graficará los datos como una función continua a lo largo de toda la ventana gráfica. Por ejemplo, la función puede quedar como sigue:

```
void simplescopio(float[] buffer) {
  int x, t;
  float[] y = new float[width];
  float dt = (float)buffer.length / width;
  for (x = 0; x < width; x++) {
    t = (int)(x * dt);
    y[x] = height / 2 - buffer[t] * height / 4;
  }

  for (x = 0; x < width - 1; x++) {
    line(x, y[x], x + 1, y[x+1]);
  }
}
```

El procedimiento es muy simple. Se crea un arreglo `y[]` que contendrá la coordenada y correspondiente a cada coordenada x de la gráfica (donde x va de cero a `width-1`). Luego se calcula el factor de proporción entre las unidades de tiempo del buffer (dadas en muestras) con respecto al ancho de la ventana gráfica. En el primer ciclo, se calcula la coordenada y de la gráfica para cada valor de x, de manera que la gráfica quede centrada (alrededor de `height/2`) y

con un rango igual a `height / 2`, cubriendo así la mitad central de la ventana. Finalmente, el segundo ciclo traza la gráfica dibujando líneas entre los puntos consecutivos.

Para visualizar la salida de alguna UGen de Beads mediante la función `simplescopio()`, simplemente hay que pasarle como argumento el buffer correspondiente al canal que se desea visualizar (la función solo puede graficar un canal a la vez). En Beads, podemos obtener el buffer de cualquier canal de una UGen mediante el método `getOutBuffer()` al cual se le pasa como argumento el número de canal. Por ejemplo, para visualizar el canal izquierdo de la salida del amplificador, podemos implementar la siguiente función `draw()`:

```
void draw() {
  background(0);                      // pinta el fondo negro
  stroke(255);                        // elige el color blanco para el trazo
  simplescopio(amp.getOutBuffer(0));  // grafica el canal izquierdo
}
```

También es sencillo visualizar ambos canales, por ejemplo, utilizando distintos colores y colocando cada gráfica a una altura diferente:

```
void draw() {
  background(0);                      // pinta el fondo negro
  translate(0, -height / 4);          // traslada la gráfica hacia arriba
  stroke(0, 255, 0);                  // elige el color verde para el trazo
  simplescopio(amp.getOutBuffer(0));  // grafica el canal izquierdo
  translate(0, height / 2);           // traslada la gráfica hacia abajo
  stroke(255, 0, 255);                // elige el color magenta para el trazo
  simplescopio(amp.getOutBuffer(0));  // grafica el canal derecho
}
```

En el caso de Minim, utilizar la función `simplescopio()` es un poco más complicado debido a que las UGens de Minim procesan una sola muestra a la vez, por lo que solamente podemos acceder a la última muestra de una UGen (mediante el método `getLastValues()`). Tampoco es posible graficar una muestra a la vez, ya que la frecuencia de actualización de la ventana (60 Hz) es mucho menor que la frecuencia de muestreo del audio (44100 Hz). Por este motivo, no existe una manera directa de obtener el buffer de un UGen. Sin embargo, es posible obtener el buffer asociado a la salida de audio del objeto `AudioOutput` que se obtiene durante la función `setup()`. Este objeto cuenta con dos miembros llamados `left` y `right`, los cuales son de clase `AudioBuffer`, y cuentan a la vez con el método `toArray()` que permite obtener una copia del buffer.

De esta manera, la función `draw()` para visualizar el canal izquierdo usando Minim es la siguiente:

```
void draw() {
  background(0);                      // pinta el fondo negro
  stroke(255);                        // elige el color blanco para el trazo
```

```
  simplescopio(out.left.toArray());   // grafica el canal izquierdo
}
```

Similarmente, podemos visualizar ambos canales, en posiciones distintas y con colores distintos:

```
void draw() {
  background(0);                      // pinta el fondo negro
  translate(0, -height / 4);          // traslada la gráfica hacia arriba
  stroke(0, 255, 0);                  // elige el color verde para el trazo
  simplescopio(out.left.toArray());   // grafica el canal izquierdo
  translate(0, height / 2);           // traslada la gráfica hacia abajo
  stroke(255, 0, 255);                // elige el color magenta para el trazo
  simplescopio(out.right.toArray());  // grafica el canal derecho
}
```

4.5.4 C++ y PortAudio

En el caso de PortAudio, la señal de entrada se almacena en un buffer que se pasa como primer argumento a la función callback (refiérase a la sección 2.5.3 para ver la descripción detallada de esta función). Sin embargo, sería deseable tener a esta señal como salida de una UGen que pueda conectarse con otras UGen. Para esto, simplemente crearemos una UGen genérica (es decir, de clase `UGen`) y dentro de la función callback copiaremos el buffer de entrada en el buffer de la UGen, antes de llamar al método `update()` de la UGen que se asociará a la salida de audio.

En otras palabras, será necesario modificar la función callback para que ahora podamos pasarle los apuntadores a dos UGens: una asociada a la entrada de audio, y otra asociada a la salida. Dado que solamente podemos pasar un argumento de usuario a la función callback (el apuntador `void *userData`), este argumento tendrá que ser un apuntador a un arreglo que contenga los apuntadores de las dos UGen. Adoptaremos la convención de que el primer elemento de este arreglo corresponda a la UGen de entrada, mientras que el segundo elemento apuntará a la UGen de salida.

Con base en lo anterior, la nueva función callback queda como sigue:

```
static int paCallback(const void *inputBuffer, void *outputBuffer,
        unsigned long framesPerBuffer,
        const PaStreamCallbackTimeInfo *timeInfo,
        PaStreamCallbackFlags statusFlags,
        void *userData) {
    if (!userData) return 0;

    //  ahora userData se interpreta como un arreglo de apuntadores a UGens estereo
    //  asociados a la entrada y salida de audio, respectivamente
    UGen **ugen = (UGen **)userData;
```

```
    //  copia el buffer de entrada de PortAudio al buffer del UGen de entrada
    if (ugen[0] != NULL && ugen[0]->getNumOutputs() == 2) {
       memcpy(ugen[0]->getBuffer(), inputBuffer, framesPerBuffer * 2 * sizeof(float));
    }

    //  calcula la señal de salida y la copia en el buffer de salida de PortAudio
    if (ugen[1] != NULL && ugen[1]->getNumOutputs() == 2) {
        ugen[1]->update();
        memcpy(outputBuffer, ugen[1]->getBuffer(), framesPerBuffer * 2 * sizeof(float));
    }

    //  actualiza el tiempo global de las UGen
    UGen::tick();
    return 0;
}
```

Por otra parte, la función `inicializaAudio()` que implementamos en la primer práctica debe recibir como segundo argumento un arreglo de apuntadores a UGens, en lugar de un solo apuntador a UGen. Esto se logra simplemente modificando el encabezado de la función:

```
void initializeAudio(PaStream **stream, UGen **ugen) {
...
}
```

Finalmente, hay que crear una UGen genérica asociada con la entrada de audio, conectarla a un amplificador, colocar los apuntadores a ambas UGen en un arreglo que se pasará como argumento a la función `inicializaAudio()`. Esto puede hacerse en la función `main()`, tomando como base el programa elaborado en la práctica anterior:

```
int main() {
// Inicializa los parámetros generales de las UGen
    UGen::setup(44100, 512);

// Crea UGens estereo para la entrada de audio y el amplificador
    UGen *input = new UGen(2);
    Amplificador *amp = new Amplificador(2);
    amp->setInput(input);
    amp->setGanancia(0.0);

// Inicializa PortAudio
    PaStream *stream;
    UGen *io[2];
    io[0] = input; io[1] = amp;
    initializeAudio(&stream, io);
```

```
// Ciclo de interacción con el usuario
    char c = 0;
    do {
        // revisa si se ha presionado una tecla
        if (_kbhit()) {
            c = _getch();
            if (c >= '0' && c <= '9') {
                float g = (float)(c - '0') / 9;
                amp->setGanancia(g * g);
            }
        }

        // hacer otras cosas

    } while (c != 27);

// Cierra el sistema de audio y libera la memoria reservada
    Pa_CloseStream(stream);
    Pa_Terminate();
    delete amp;
    delete input;
    return 0;
}
```

Para concluir, se implementará la función `simplescopio()` para visualizar la señal correspondiente a un canal del buffer de salida de alguna UGen. Dado que el programa se ejecuta en modo consola, la visualización será mucho mas rudimentaria que en el caso de Processing. Por supuesto, es posible visualizar la señal en una ventana gráfica usando C++, pero esto requiere el uso de funciones propias del sistema operativo, o bien, del uso de alguna librería adicional, lo cual queda fuera del alcance de los objetivos de este manual.

La visualización en C++ se realizará de manera similar, calculando primero las coordenadas y correspondientes a cada coordenada x de la gráfica que se desea trazar, salvo que ahora el rango de x será mucho menor (de 0 a 63) ya que la resolución está en caracteres, no en pixeles. La otra diferencia es que en lugar de dibujar lineas uniendo los puntos de la gráfica, simplemente se dibujará una línea vertical en cada posición x cuya altura será $y(x)$ (algo similar a la función `stem()` de Matlab y Octave). La función `simplescopio()` tomará como argumentos un apuntador al UGen cuya señal se desea visualizar y el número de canal deseado:

```
void simplescopio(UGen *ugen, int canal, int zoom = 10) {
    const int ancho = 64;
    int i, x, y[ancho];

// verifica validez de los argumentos
```

```
    if (ugen == NULL) return;
    if (canal < 0 || canal >= ugen->getNumOutputs()) canal = 0;

    // Obtiene el buffer para el canal deseado y calcula desplazamiento
    float *buffer = ugen->getBuffer() + canal;
    int salto = (UGen::getBufferSize() / ancho) * ugen->getNumOutputs();

    // calcula coordenadas y[x]
    i = 0;
    for (x = 0; x < ancho; x++) {
        y[x] = (int)(buffer[i] * zoom);
        if (y[x] < -zoom) y[x] = -zoom;
        if (y[x] > zoom) y[x] = zoom;
        i += salto;
    }

// borra pantalla llamando a un comando del sistema operativo
    system("cls");  // borrar pantalla en Windows
    // system("clear"); // borrar pantalla en Linux o MacOSX

    // dibuja grafica, renglón por renglón, caracter por caracter
    cout << endl;
    for (i = zoom; i >= -zoom; i--) {
        for (x = 0; x < ancho; x++) {
            if (i == 0) cout << "-"; // pinta el eje X
            else if ((y[x] * i > 0) && abs(y[x]) >= abs(i)) cout << \*";
            else cout << " ";
        }
        cout << endl;
    }
}
```

Para utilizar la función anterior, simplemente hay que llamarla dentro del ciclo `do ... while` que se encuentra en la función `main()`. Por ejemplo,

```
    do {
        // revisa si se ha presionado una tecla
        if (_kbhit()) {
            c = _getch();
            if (c >= '0' && c <= '9') {
                float g = (float)(c - '0') / 9;
                amp->setGanancia(g * g);
            }
        }

        // hacer otras cosas
        simplescopio(amp, 0);
```

```
} while (c != 27);
```

4.6 Evaluación y reporte de resultados

1. Pruebe el sistema implementado con un micrófono como dispositivo de entrada y audífonos como dispositivo de salida. Haga algún sonido percusivo como aplaudir o chasquear los dedos. Note que existe un retardo entre la ejecución del aplauso y el sonido emitido por los audífonos. Explique a qué se debe este retardo y calcule el tiempo de retardo con la mayor precisión posible.

2.- Utilice el programa elaborado en esta práctica para experimentar con retroalimentación controlada, usando el micrófono como entrada de audio y las bocinas como salida. Incremente lenta y gradualmente la ganancia del amplificador (y/o el volumen de las bocinas) hasta que comience a apreciar los siguientes fenómenos:

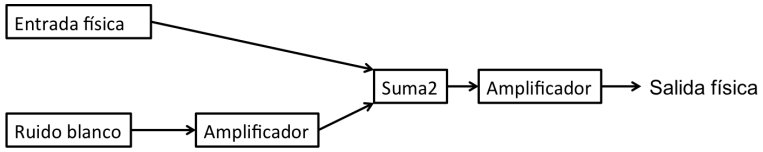
- El sonido que ingresa por el micrófono se reproduce por las bocinas con una especie de eco.
- Conforme aumenta la ganancia a la salida, se escucha un tono agudo. Este fenómeno se conoce como *resonancia*.
- Si la ganancia aumenta demasiado, el tono emitido debido a la resonancia adquiere una forma de onda cuadrada.

Considerando lo anterior, así como el modelo matemático de retroalimentación expuesto en la sección 4.5, responda las siguientes preguntas:

1. Intente explicar cada uno de los fenómenos descritos arriba.
2. Estime, de manera aproximada, el tiempo entre cada uno de los ecos. Considere, además de la latencia del sistema, el tiempo que tarda el sonido en llegar de las bocinas al micrófono dado que la velocidad del sonido es 343 m/s.
3. Cómo estimaría la frecuencia de resonancia del sistema? Qué se puede hacer para modificar esta frecuencia?

4.7 Retos

Utilizando la UGen `Suma2` que implementó en la evaluación de la práctica anterior (Sección 3.6), modifique el programa elaborado en esta práctica para que agregue una cierta cantidad de ruido a la señal proveniente de la entrada física de audio. Considere, por ejemplo, el siguiente diagrama de bloques (cada bloque representa una UGen):



La ganancia del primer amplificador controla el nivel de ruido agregado a la entrada de audio, mientras que el segundo amplificador controla el volumen total de la salida. Agregue elementos de interfaz de usuario para controlar el nivel de ruido.

4.8 Conclusiones

Redacte en el recuadro sus conclusiones acerca de los conceptos aprendidos, las actividades realizadas y los objetivos planteados en esta práctica.

Capítulo 5

Filtros de primer orden

Nombre del estudiante	Calificación

5.1 Relación con los programas de estudio

Materia	Unidad y tema
Procesamiento Digital de Señales (IE / IT / IB)	1.3 - Sistemas discretos y sus características 1.4 - Sistemas lineales e invariantes en el tiempo 1.6 - Representación de sistemas LIT mediante ecuaciones en diferencias 5.1 - Definición de Transformada Z 5.3 - Transformada Z racional 5.4 - Propiedades de la Transformada Z 5.5 - Representación de sistemas LIT en el dominio Z 6.2 - Consideraciones para el diseño de filtros 6.4 - Diseño de filtros IIR
Procesamiento de Señales de Audio (IE)	1.5 - Sistemas lineales e invariantes en el tiempo 1.7 - Sistemas LIT dados como ecuaciones en diferencias 1.8 - Filtros FIR e IIR 1.9 - Respuesta en frecuencia 1.11 - Transformada Z 2.1 - Filtros de primer orden

5.2 Introducción

De una manera muy general, se le llama *filtro* a cualquier medio o dispositivo por el que puede pasar una señal de audio (ya sea acústica, electrónica o digital) para ser alterada de alguna manera [12]. En muchas ocasiones, es conveniente describir la acción del filtro en términos del contenido de espectral de la señal de salida con respecto al de la señal de entrada. Por ejemplo, el tracto vocal de los humanos y otros animales actúa como un filtro, amplificando ciertas frecuencias a las que se conoce como *formantes*, dando lugar al timbre característico de voz de cada persona; además, al cambiar la forma de la cavidad bucal es posible cambiar las características de filtrado del tracto vocal para producir los sonidos de las distintas vocales. De manera similar, la mayoría de los instrumentos musicales acústicos cuentan con una caja resonante que modifica el sonido producido por el generador inicial (usualmente cuerdas o membranas), y es en buena parte responsable del sonido característico o timbre de cada instrumento. Por otra parte, la mayoría de los materiales tienden a absorber las altas frecuencias con mayor facilidad que las bajas frecuencias, por lo que los sonidos cambian sus características espectrales al atravesar o rebotar en estos materiales. Un ejemplo interesante es nuestra propia cabeza, que actúa como un filtro para el sonido que llega al oído que está del lado opuesto a la fuente; la diferencia entre el sonido filtrado que llega a un oído con respecto al sonido sin filtrar que llega al otro oído, es uno de los principales rasgos que permiten a nuestro cerebro estimar la dirección de la fuente.

En el dominio digital, un filtro es cualquier procedimiento que tome como entrada una secuencia de muestras y devuelva una versión modificada de ésta. Esta es una definición extremadamente general, lo que significa que los filtros son bloques fundamentales para el diseño de procesos mas sofisticados. Por este motivo es importante comprender las distintas maneras de caracterizarlos, clasificarlos, diseñarlos e implementarlos.

Matemáticamente, representaremos un filtro digital como una transformación $y = T\{x\}$ que toma como entrada una señal x y devuelve como salida la señal y correspondiente. En lo posible, se buscará mantener esta notación a lo largo del manual.

5.2.1 Propiedades fundamentales de los filtros

Un filtro $T\{\cdot\}$ puede tener cero o más de las siguientes propiedades [13, 6]:

- **Linealidad:** Cuando para cualesquiera dos señales $x[n]$, $y[n]$ y cualesquiera escalares $a, b \neq 0$ se cumple que $T\{ax + by\} = aT\{x\} + bT\{y\}$.
- **Invarianza en el tiempo:** Sea $y = T\{x\}$. Supongamos además que $x_d[n] = x[n-d]$ y $y_d[n] = y[n-d]$; es decir, x_d y y_d son versiones retardadas de x y y, respectivamente, donde el tiempo de retardo es precisamente d. El filtro T es invariante en el tiempo (o invariante al retardo) si se cumple que $y_d = T\{x_d\}$ para cualquier $d \in \mathbb{Z}$.

- **Causalidad:** Un filtro es causal cuando la salida $y[n]$ depende únicamente de las entradas $x[m],\ m \leq n$, pero no depende de valores "futuros" de la señal de entrada.

- **Memoria:** Decimos que un filtro *tiene memoria* cuando la salida $y[n]$ depende de una o mas entradas $x[m]$ con $m \neq n$. Un filtro cuya salida $y[n]$ depende únicamente de la muestra $x[n]$ se conoce como un filtro *sin memoria*, *waveshaper* o una *función de transferencia de tonos* (en el contexto del procesamiento de imágenes).

- **Estabilidad:** Un filtro es estable si una entrada acotada produce una salida acotada. En otras palabras, dado M_x tal que $|x[n]| < M_x$ para todo n, entonces el filtro es estable si existe un real M_y tal que $|y[n]| < M_y$ para todo n.

Para facilitar el análisis y caracterización de los filtros, los textos suelen enfocarse en aquellos que son lineales e invariantes en el tiempo (LIT). Cualquier filtro LIT puede caracterizarse a través de su respuesta al impulso, dada por $h = T\{\delta\}$, donde $\delta[n] = 1$ si $n = 0$ y $\delta[n] = 0$ para $n \neq 0$, por su respuesta en frecuencia $H(\omega)$, la cual cumple que $T\{\exp(j\omega n)\} = H(\omega)\exp(j\omega n)$, o por su función de transferencia $H(z),\ z \in \mathbb{C}$, la cual se obtiene mediante la Transformada Z. Por otra parte, existen diversas maneras de implementar un filtro LIT: mediante convolución, a través de ecuaciones en diferencias, o mediante filtrado en el dominio de la frecuencia. Esto permite elegir la implementación que mas convenga para cada aplicación.

Sin embargo, prácticamente todos los fenómenos de filtrado que ocurren en la naturaleza son de carácter no-lineal y/o variantes en el tiempo. Una manera de modelar algunos de estos fenómenos es introduciendo una función de transferencia no-lineal antes o después de aplicar un filtro LIT. Por otra parte, la varianza en el tiempo se puede modelar a través de un filtro LIT cuya descripción matemática depende de uno o más parámetros, los cuales pueden modularse (cambiarse) mientras la señal está siendo procesada. En otras palabras, es posible simular algunos procesos variantes en el tiempo mediante un filtro LIT multi-canal parametrizado, donde algunos de los canales de entrada corresponden a las señales moduladoras de los parámetros.

Finalmente, para la implementación de procesos en tiempo real, por lo general es deseable el uso de filtros causales, dado que al procesar una muestra o un buffer se desconocen las muestras futuras (mas allá del buffer actual). Es posible el uso de filtros no causales, pero esto requiere esperar a conocer las muestras futuras, introduciendo un retardo adicional en la salida del sistema e incrementando la latencia.

5.2.2 Ultra-breve taxonomía de los filtros LIT

Los filtros LIT pueden clasificarse de diversas maneras. Las siguientes clasificaciones son relevantes para los propósitos de este manual.

Por su respuesta en frecuencia: pueden ser pasa-bajas, pasa-altas, pasa-banda, rechaza-banda (también llamados filtros *notch*), o pasa-todo. Los primeros cuatro suelen utilizarse para resaltar o eliminar bandas de frecuencia específicas, mientras que los filtros pasa-todo son aquellos cuya respuesta en frecuencia tiene una magnitud unitaria para todas las frecuencias. Estos últimos incluyen a los retardos y modificadores de fase. Existen además una gran variedad de filtros que no encajan en ninguna de estas cinco categorías [14].

Por su respuesta al impulso: pueden ser de respuesta finita (FIR) o de respuesta infinita (IIR). Los filtros FIR suelen implementarse mediante convolución o filtrado en frecuencia, mientras que los filtros IIR suelen implementarse a partir de una descripción dada por ecuaciones en diferencias.

5.2.3 Filtro IIR de un polo

En esta práctica exploraremos los filtros LIT de respuesta infinita al impulso (IIR) expresados mediante ecuaciones en diferencias de primer orden. En este caso, el modelo mas sencillo está dado por

$$y[n] = ay[n-1] + bx[n], \tag{5.1}$$

donde a y b son coeficientes reales fijos. La función de transferencia para este filtro está dada por

$$H(z) = \frac{b}{1 - az^{-1}} = \frac{bz}{z - a},$$

con $|a| < 1$, de manera que el filtro tiene un polo en $z = a$ y un cero en $z = 0$.

La respuesta en frecuencia de un filtro LIT puede obtenerse evaluando la función de transferencia en el círculo unitario $z = e^{j\omega}$, donde ω es una frecuencia dada en radianes por muestra. En este caso, la respuesta en frecuencia está dada por

$$H(e^{j\omega}) = b\frac{e^{j\omega}}{e^{j\omega} - a}.$$

Por otra parte, los polos son puntos alrededor de los cuales la respuesta del filtro crece (ya que el denominador de la función de transferencia se acerca a cero). Dado que el valor del polo a es real, pueden darse tres casos:

- Caso $a = 0$. En este caso, la función de transferencia es constante: $H(z) = b$, lo que corresponde a un amplificador con ganancia b.
- Caso $a > 0$. El polo se encuentra del lado derecho del eje real, con un ángulo igual a cero en el plano complejo. Por lo tanto, el filtro tendrá una mayor ganancia para las frecuencias cercanas a cero, dando como resultado un filtro pasa-bajas. El efecto del filtro será mas pronunciado conforme a se acerque a 1 (al círculo unitario).
- Caso $a < 0$. El polo se encuentra del lado izquierdo del eje real, con un ángulo igual π, lo que corresponde a un filtro pasa-altas. El efecto del filtro será mas pronunciado conforme a se aproxime a -1.

En todos los casos, se puede observar que b actúa simplemente como un factor de ganancia. Conforme el polo a se acerca al círculo unitario (ya sea a 1 o -1), el denominador de la respuesta en frecuencia podría tomar valores muy cercanos a cero, con lo cual la respuesta a ciertas frecuencias (bajas o altas, dependiendo del signo del polo) crecería desmesuradamente, ocasionando distorsión a la salida del filtro. Una manera de evitar esto es ajustar el factor de ganancia b para que la ganancia máxima sea unitaria. Por ejemplo, para el caso pasa-bajas ($a > 0$), la ganancia máxima se obtiene en la frecuencia $\omega = 0$, donde la respuesta es

$$H(e^{j0}) = H(1) = b/(1-a),$$

por lo que ajustando $b = 1 - a$ se obtiene una ganancia máxima unitaria. Similarmente, se puede ver que para el caso pasa-altas ($a < 0$), la ganancia máxima unitaria se obtiene con $b = 1 + a$.

Alternativamente, podemos representar el valor del polo como $a = sp$, donde p es la magnitud o valor absoluto del polo (con $0 \leq p < 1$) y $s \in \{+1, -1\}$ es su signo. De esta manera, s indica si el filtro es pasa-bajas ($s = +1$) o pasa-altas ($s = -1$), mientras que p representa la intensidad del efecto de filtrado. También se puede decir que $1 - p$ es la *apertura* del filtro, que es una medida de qué tanto contenido espectral se deja pasar. Con esta representación, el factor de ganancia máxima unitaria es $b = 1 - p$ para ambos casos (pasa-bajas y pasa-altas).

Aunque el filtro de un polo es relativamente limitado, este filtro es de gran utilidad para un control general de tono, para generar distintos tipos o colores de ruido, y para suavizar señales de control. Algunas de estas aplicaciones se estudiarán mas adelante.

Como comentario final, es posible obtener un filtro pasa-banda de un polo si el polo se coloca fuera del eje real, específicamente en $a = p \exp\{j\omega_0\}$, donde ω_0 es la frecuencia central del filtro. Sin embargo, esto requiere cálculos con números complejos, lo cual incrementa la complejidad computacional del filtro. En la medida de lo posible, se buscará que los procesos implementados utilicen únicamente operaciones con números reales.

5.3 Objetivos didácticos

- Comprender la definición y propiedades básicas de los filtros, y cuáles de ellas son de interés para el diseño e implementación de filtros en tiempo real.

- Analizar e implementar el filtro de respuesta infinita con un polo para procesar señales en tiempo real. Explorar el efecto que tienen los parámetros del filtro en la señal procesada.

5.4 Material

- Una computadora con alguna de las siguientes combinaciones de software instalado:
  - Processing y Beads
  - Processing y Minim
  - GNU C++ (e.g., MinGW) y PortAudio
- Bocinas o audífonos estéreo

5.5 Procedimiento

El procedimiento para esta práctica será sencillo. Como primer paso, se implementará el filtro IIR de un polo como una nueva UGen llamada `Filtro1P` (Filtro de un polo). La UGen comprenderá tanto el filtro pasa-bajas como el pasa-altas a través de un parámetro que controlará el signo del polo. Posteriormente, utilizaremos el filtro para procesar una señal de prueba multicanal - en este caso, ruido blanco - y apreciar el efecto que tiene el filtro en la señal procesada.

Un aspecto nuevo e importante es la necesidad de conservar en todo momento el valor de la última muestra de salida de cada canal; es decir, $y[n-1]$, para poder implementar la Ecuación 5.1. Esto lo haremos incluyendo como miembro de datos de la clase `Filtro1P` un arreglo de floats, llamado `anterior[]`, que almacenarán la última muestra procesada de cada canal, y se irá actualizando con el cálculo de cada muestra. Este arreglo debe crearse en el momento en que se define el número de canales de la UGen.

La clase `Filtro1P` incluirá también dos miembros de datos para especificar los parámetros del filtro: una variable booleana `pasaAltas` que determinará el signo del polo, y un flotante `polo` que corresponde al valor absoluto del polo. Además, se implementaran funciones *set* y *get* para manipular estos parámetros, las cuales son idénticas para las tres plataformas. El factor de ganancia del filtro se determinará automáticamente para una ganancia máxima unitaria.

De manera general, el procesado en las tres plataformas se realiza calculando primero los coeficientes a y b del filtro mediante el siguiente código:

```
float a = pasaAltas ? -polo : polo;
float b = 1 - polo;
```

y luego se calcula cada muestra de salida mediante la ecuación del filtro (Ec. 5.1) y se actualiza la muestra anterior. Esto se logra con el siguiente pseudo-código:

```
out[c][i] = a * anterior[c] + b * in[c][i];
anterior[c] = out[c][i];
```

donde `c` e `i` son contadores que indexan los canales y las muestras del buffer, respectivamente.

A continuación se presentan las implementaciones en las tres plataformas.

5.5.1 Processing y Beads

En el caso de Beads, el número de canales de una UGen se define en el constructor; por lo tanto, el constructor es el lugar adecuado para inicializar el arreglo de muestras `anterior[]`. Por otra parte, el método `calculateBuffer()` utiliza un ciclo para recorrer los canales, y otro para recorrer las muestras para cada canal, dentro de los cuales se calcula cada muestra de salida y se actualiza la muestra anterior para cada canal.

```
public class Filtro1P extends UGen {
  protected float[] anterior;
  float polo;
  boolean pasaAltas;

  public Filtro1P(AudioContext context, int canales) {
    super(context, canales, canales);
    anterior = new float[canales];
    polo = 0;
    pasaAltas = false;
  }

  float polo() {
    return polo;
  }
  void setPolo(float p) {
    if (p < 0) p = 0;
    if (p > 1) p = 1;
    polo = p;
  }

  boolean pasaAltas() { return pasaAltas; }

  void setPasaAltas(boolean pa) { pasaAltas = pa; }

  public void calculateBuffer() {
    float a = pasaAltas ? -polo : polo;
    float b = 1 - polo;
    for (int c = 0; c < getOuts(); c++) {
      float[] in = bufIn[c];
      float[] out = bufOut[c];
      for (int i = 0; i < bufferSize; i++) {
        out[i] = a * anterior[c] + b * in[i];
        anterior[c] = out[i];
      }
    }
  }
}
```

Para el programa de prueba, conectaremos un generador de ruido al filtro, y el filtro a un amplificador, que finalmente se conectará a la salida de audio. Utilizaremos el mouse para controlar la apertura del filtro (eje horizontal), la ganancia del amplificador (eje vertical) y el tipo de filtro (botón del mouse), y graficaremos ambos canales de la señal de salida utilizando la función `simplescopio()` implementada en la práctica anterior. El siguiente código debería ser auto-explicativo.

```
AudioContext ac;
RuidoBlanco ruido;
Amplificador amp;
Filtro1P filtro;

void setup() {
  size(400, 300);

  ac = new AudioContext();
  ruido = new RuidoBlanco(ac, 2);
  filtro = new Filtro1P(ac, 2);
  amp = new Amplificador(ac, 2);

  filtro.addInput(ruido);
  amp.addInput(filtro);
  ac.out.addInput(amp);
  amp.setGanancia(0.0);
  ac.start();
}

void mouseMoved() {
  amp.setGanancia(1 - (float)mouseY / height);
  filtro.setPolo(1 - pow((float)mouseX / width, 2));
}

void mouseClicked() {
  filtro.setPasaAltas(!filtro.pasaAltas());
}

void draw() {
  background(0);
  fill(255);
  text("ganancia = " + amp.ganancia(), 10, 10);
  text("polo = " + filtro.polo(), 10, 20);
  translate(0, -height / 4);
  stroke(0, 255, 0);
  simplescopio(amp.getOutBuffer(0));
  translate(0, height / 2);
```

```
  stroke(255, 0, 255);
  simplescopio(amp.getOutBuffer(1));
}
```

5.5.2 Processing y Minim

En el caso de Minim, el número de canales de una UGen no se determina en el constructor, sino de manera dinámica a la hora de interconectar las UGens entre sí. Cuando el número de canales cambia, se llama automáticamente a un método llamado `channelCountChanged()`, el cual podemos reimplementar en las clases heredadas de `UGen` para agregar la funcionalidad deseada. Es en este método donde crearemos el arreglo que contendrá la muestra anterior para cada canal. Por otra parte, dado que en Minim las UGen procesan solamente una muestra a la vez, la función `uGenerate()` que realiza el procesado de la señal será mas sencilla (aunque menos eficiente) que en las otras implementaciones. De esta manera, la clase `Filtro1P` para Minim puede implementarse como sigue:

```
public class Filtro1P extends UGen {
  protected float[] anterior;
  float polo;
  boolean pasaAltas;
  UGenInput audio_in;

  public Filtro1P() {
    super();
    audio_in = new UGenInput(UGen.InputType.AUDIO);
    polo = 0;
    pasaAltas = false;
  }

  protected void channelCountChanged() {
    super.channelCountChanged();
    anterior = new float[channelCount()];
  }

  float polo() {
    return polo;
  }
  void setPolo(float p) {
    if (p < 0) p = 0;
    if (p > 1) p = 1;
    polo = p;
  }

  boolean pasaAltas() { return pasaAltas; }
```

```
  void setPasaAltas(boolean pa) { pasaAltas = pa; }

  protected void uGenerate(float[] channels) {
    int cc = channelCount();
    float a = pasaAltas ? -polo : polo;
    float b = 1 - polo;
    float[] in = audio_in.getLastValues();
    for (int c = 0; c < cc; c++) {
      channels[c] =  a * anterior[c] + b * in[c];
      anterior[c] = channels[c];
    }
  }
}
```

Para el programa principal, conectaremos un generador de ruido al filtro, y luego a un amplificador. Al igual que en Beads, aprovecharemos la interfaz del mouse para controlar la apertura del filtro (eje horizontal), la ganancia del amplificador (eje vertical) y el tipo de filtro (botón), mostrando los dos canales de la señal de salida.

```
Minim minim;
AudioOutput out;
RuidoBlanco ruido;
Filtro1P filtro;
Amplificador amp;

void setup() {
  size(400, 300);

  minim = new Minim(this);
  out = minim.getLineOut();

  ruido = new RuidoBlanco();
  filtro = new Filtro1P();
  amp = new Amplificador();

  ruido.patch(filtro);
  filtro.patch(amp);
  amp.patch(out);
  amp.setGanancia(0);
}

void mouseMoved() {
  amp.setGanancia(1 - (float)mouseY / height);
  filtro.setPolo(1 - pow((float)mouseX / width, 2));
}
```

```
void mouseClicked() {
  filtro.setPasaAltas(!filtro.pasaAltas());
}

void draw() {
  background(0);
  fill(255);
  text("ganancia = " + amp.ganancia(), 10, 10);
  text("polo = " + filtro.polo(), 10, 20);
  translate(0, -height / 4);
  stroke(0, 255, 0);
  simplescopio(out.left.toArray());
  translate(0, height / 2);
  stroke(255, 0, 255);
  simplescopio(out.right.toArray());
}
```

5.5.3 C++ y PortAudio

La implementación en C++ (basada en la clase `UGen` del Apéndice B) es muy similar a la de Beads, ya que el número de canales se establece en el constructor, por lo que ahí se creará el arreglo para las muestras anteriores de todos los canales y se inicializará con ceros. Ya que se reservará memoria adicional durante el constructor, será necesario implementar un destructor donde se libere la memoria reservada.

Por otra parte, en la función callback `processBuffer()` será necesario realizar comprobaciones adicionales para verificar que el número de canales de la entrada del filtro es igual al número de canales de salida, y considerar que los buffers de salida de las UGen contienen los datos intercalados por canales, lo cual modifica el orden de los ciclos con respecto a la implementación en Beads: es decir, primero se realizará un ciclo para recorrer las muestras, y dentro de éste se tendrá el ciclo que recorre los distintos canales.

```
class Filtro1P : public UGen {
protected:
    float *anterior;
    float polo;
    bool pasaAltas;

public:
    Filtro1P(int canales) : UGen(canales, 1) {
        anterior = new float[canales];
        memset(anterior, 0, canales * sizeof(float));
        polo = 0;
        pasaAltas = false;
```

```
    }

    ~Filtro1P() {
        if (anterior) delete[] anterior;
        anterior = NULL;
    }

    float getPolo() { return polo; }

    void setPolo(float p) {
        if (p < 0) p = 0;
        if (p > 1) p = 1;
        polo = p;
    }

    bool isPasaAltas() { return pasaAltas; }

    void setPasaAltas(bool pa) { pasaAltas = pa; }

    void processBuffer() {
        int numOuts = getNumOutputs();
        UGen *ugen = getInput(0);
        if (ugen == NULL) return;
        if (ugen->getNumOutputs() != numOuts) return;

        float *in = ugen->getBuffer();
        float *out = getBuffer();
        int N = getBufferSize() * numOuts;
        float a = pasaAltas ? -polo : polo;
        float b = 1 - polo;
        for (int i = 0; i < N; ) {
            for (int c = 0; c < numOuts; c++) {
                out[i] = a * anterior[c] + b * in[i];
                anterior[c] = out[i];
                i++;
            }
        }
    }
};
```

Finalmente, la función principal se encargará de crear e interconectar un generador de ruido, un filtro y un amplificador (en ese orden), y realizar la interfaz de usuario para controlar los parámetros del proceso. Dado que el programa se ejecutará en modo de consola, la interfaz de usuario estará limitada a mostrar únicamente un canal de salida (el izquierdo) y controlar solamente la magnitud del polo del filtro (mediante las teclas del '0' al '9') y el tipo de filtro

(mediante la tecla 'S'). La ganancia del amplificador se mantendrá siempre a 1. En este caso, se tendría el mismo resultado si se omite por completo el amplificador; sin embargo, se decidió mantener una arquitectura similar para las tres implementaciones.

```
int main() {
    UGen::setup(44100, 512);

    RuidoBlanco *ruido = new RuidoBlanco(2);
    Filtro1P *filtro = new Filtro1P(2);
    Amplificador *amp = new Amplificador(2);

    filtro->setInput(ruido);
    amp->setInput(filtro);
    amp->setGanancia(1.0);

    PaStream *stream;
    UGen *io[2];
    io[0] = NULL; io[1] = amp;
    initializeAudio(&stream, io);

    char c = 0;
    do {
        if (_kbhit()) {
            c = _getch();
            if (c >= '0' && c <= '9') {
                float g = (float)(c - '0') / 10;
                filtro->setPolo(1 - g * g);
            }
            if (c == 's' || c == 'S') {
                filtro->setPasaAltas(!filtro->isPasaAltas());
            }
        }
        simplescopio(amp, 0);
    } while (c != 27);

    Pa_CloseStream(stream);
    Pa_Terminate();
    delete amp;
    delete input;
    return 0;
}
```

5.6 Evaluación y reporte de resultados

1.- Mencione tres fenómenos que puedan ser considerados como filtros para señales de audio, distintos a los que se contemplan en la sección de Introducción.

1.

2.

3.

2.- Describa las propiedades mas deseables que debe tener un filtro para su implementación en tiempo real.

3.- Considere el filtro implementado en esta práctica con un polo $a = sp$, donde p representa la magnitud del polo. El factor de compensación de ganancia $b = 1 - p$ elegido en nuestra implementación asegura que la ganancia máxima no sobrepasa la unidad. Sin embargo, conforme p se acerca a uno, la banda de paso del filtro se vuelve muy angosta, por lo cual la mayor parte del espectro es atenuada, restando una gran cantidad de energía (volumen) a la señal de salida. Una alternativa es controlar la forma de la curva de compensación de manera que en un extremo se tenga una compensación nula ($b = 1$), lo que puede ocasionar saturación o distorsión, y en el otro extremo se tenga una compensación completa ($b = 1 - p$) con la consecuente pérdida de energía. Una manera de lograr esto es eligiendo b de la siguiente manera:

$$b = (1 - p)^s,$$

donde s es el parámetro de compensación de ganancia, el cual varía desde $s = 0$ (sin compensación) hasta $s = 1$ para una compensación completa.

Modifique la clase `Filtro1P` para agregar el parámetro de compensación de ganancia, con sus respectivas funciones *set* y *get* para manipularlo. Incorpore al programa de prueba un mecanismo de interfaz para variar este parámetro y encuentre un valor que produzca buenos resultados sin llegar a distorsionar seriamente la señal de salida cuando la magnitud p del polo se acerca a la unidad.

5.7 Retos

El efecto de un filtro pasa-bajas o pasa-altas se puede acentuar colocando el cero en el círculo unitario del lado opuesto al polo. La forma general de la función de transferencia de un filtro con un polo y un cero (ambos reales) es

$$H(z) = b\frac{z-c}{z-sp},$$

donde s y p representan, respectivamente, el signo y la magnitud del polo, c es la posición del cero, y b es un factor de compensación de ganancia. La posición del cero será $c = -1$ para un filtro pasa-bajas, y $c = +1$ para un pasa-altas (es decir, $c = -s$).

a) Muestre que el factor de ganancia para una ganancia máxima unitaria es $b = (1-p)/2$, tanto para el caso pasa-bajas como el pasa-altas.

b) Muestre que la ecuación en diferencias mediante la que se puede implementar el filtro es

$$y[n] = py[n-1] + b(x[n] - cx[n-1]).$$

c) Implemente este filtro como una nueva UGen y compare, de manera cualitativa, su desempeño con respecto al filtro `Filtro1P`.

5.8 Conclusiones

Redacte en el recuadro sus conclusiones acerca de los conceptos aprendidos, las actividades realizadas y los objetivos planteados en esta práctica.

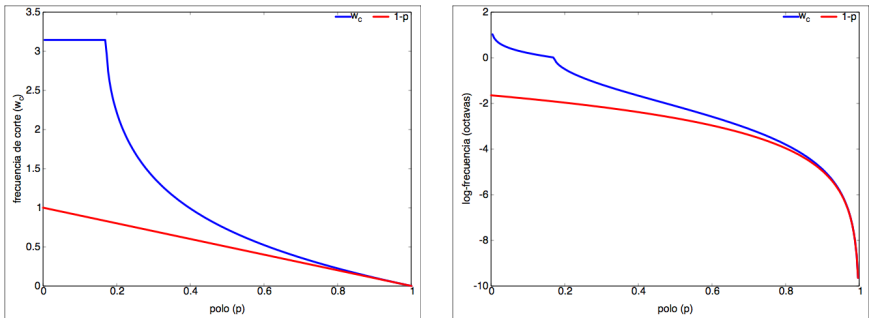
Capítulo 6

Filtros de segundo orden

Nombre del estudiante	Calificación

6.1 Relación con los programas de estudio

Materia	Unidad y tema
Procesamiento Digital de Señales (IE / IT / IB)	1.6 - Representación de sistemas LIT mediante ecuaciones en diferencias 5.3 - Transformada Z racional 5.4 - Propiedades de la Transformada Z 5.5 - Representación de sistemas LIT en el dominio Z 6.2 - Consideraciones para el diseño de filtros 6.4 - Diseño de filtros IIR
Procesamiento de Señales de Audio (IE)	1.5 - Sistemas lineales e invariantes en el tiempo 1.7 - Sistemas LIT dados como ecuaciones en diferencias 1.8 - Filtros FIR e IIR 1.9 - Respuesta en frecuencia 1.11 - Transformada Z 2.2 - Filtros de segundo orden



(a) (b)

Figura 6.1: Frecuencia de corte ω_c de un filtro pasa-bajas de un polo con respecto a la magnitud del polo p. En el panel (a) se muestra la frecuencia en escala lineal en radianes por muestra, mientras que en el panel (b) se muestra la frecuencia en escala logarítmica en octavas, donde la frecuencia de Nyquist corresponde a la octava cero. El trazo azul corresponde a la verdadera frecuencia de corte (aquella para la cual la potencia se reduce a la mitad), mientras que el trazo rojo representa la aproximación $\omega_c = 1 - p$.

6.2 Introducción

6.2.1 Más sobre el filtro pasa-bajas de un polo

En la práctica anterior se estudiaron los filtros de primer orden (un polo) con coeficientes reales. Si bien estos filtros tienen varias aplicaciones y se utilizan frecuentemente como parte de procesos mas elaborados, tienen también un par de limitaciones importantes. La primera es que solo se puede obtener una respuesta pasa-bajas o pasa-altas. Es posible diseñar un filtro pasa-banda o rechaza-banda de primer orden, pero tendría que tener coeficientes complejos, lo cual resulta en una implementación menos eficiente. La segunda desventaja es que la curva de respuesta del filtro tiene una pendiente muy poco pronunciada cuando la frecuencia de corte es relativamente alta. Debido a esto, el filtro de primer orden no es muy útil para separar distintas bandas de frecuencia, o aislar frecuencias específicas. Una buena analogía en términos de carpintería es pensar que el filtro de primer orden es como una lija: es útil para limar asperezas y redondear bordes, pero no para cortar un trozo de madera.

Para ilustrar lo anterior, calculemos la frecuencia de corte de un filtro pasa-bajas de un polo en términos de la magnitud del polo p. En este caso, definimos la frecuencia de corte ω_c como aquella para la cual la potencia de la señal se reduce a la mitad de la potencia máxima (una ganancia de 3 db por debajo de la ganancia máxima). En otras palabras, $|H(e^{j\omega_c})|^2 = 1/2$ para un filtro con

ganancia máxima unitaria. Por lo tanto:

$$\begin{aligned}
\frac{1}{2} &= |H(e^{j\omega_c})|^2 \\
1 &= 2\left|\frac{1-p}{1-pe^{-j\omega_c}}\right|^2 \\
|1-pe^{-j\omega_c}|^2 &= 2(1-p)^2 \\
(1-p\cos\omega)^2 + p^2\,\mathrm{sen}^2\,\omega &= 2(1-2p+p^2) \\
1-2p\cos\omega+p^2 &= 2-4p+2p^2 \\
\omega &= \cos^{-1}\left(-\frac{p^2-4p+1}{2p}\right).
\end{aligned}$$

La Figura 6.1a muestra una gráfica (trazo azul) de ω_c vs p, de acuerdo a la ecuación anterior. Para $p < 0.17$, el efecto del filtro es tan sutil que la frecuencia de corte es precisamente la frecuencia de Nyquist ($\omega_c = \pi$), lo cual corresponde prácticamente a un filtro pasa-todo.

Conforme p se incrementa, pareciera que la frecuencia de corte cae rápidamente; sin embargo, el oído humano analiza las frecuencias de una manera no lineal, sino aproximadamente logarítmica [8]. Para el oído, las frecuencias 100 Hz, 200 Hz, 300, Hz, 400 Hz, etc. no parecen estar equi-espaciadas, pero sí lo parecen las frecuencias 100 Hz, 200 Hz, 400 Hz, 800 Hz, etc. Si ahora graficamos la frecuencia de corte ω_c en una escala logarítmica (Figura 6.1b), se observa que la caída de la frecuencia de corte se vuelve considerablemente pronunciada solo a partir de $p > 0.8$. Mas aún, la relación entre p y ω_c es aproximadamente lineal en este rango ($p > 0.8$) y puede aproximarse bien como $\omega_c \approx 1 - p$, como lo muestran los trazos rojos de la Figura 6.1. Varios autores utilizan esta aproximación considerando que en la práctica se utilizan valores altos de p para este filtro [10, 12, 15].

6.2.2 Filtro pasa-bandas de dos polos

Una manera de construir un filtro pasa-banda consiste en tomar la respuesta en frecuencia de un filtro pasa-bajas y aplicarle un desplazamiento en frecuencia mediante la transformación $\omega \longrightarrow \omega - \omega_0$, donde ω_0 es la frecuencia central esperada del filtro pasa-banda (también llamada *frecuencia de entonamiento*). Equivalentemente, podemos tomar la función de transferencia del filtro pasa-bajas y aplicar el corrimiento en frecuencia mediante la transformación $z \longrightarrow ze^{-j\omega_0}$. Por lo tanto, podemos convertir el filtro pasa-bajas de un polo H_{lp} en un filtro pasa-bandas de un polo H_{bp} como se muestra a continuación:

$$\begin{aligned}
H_{\mathrm{bp}}(z) &= H_{\mathrm{lp}}(ze^{-j\omega_0}) \\
&= \frac{1-p}{1-p(ze^{-j\omega_0})^{-1}} \\
&= \frac{1-p}{1-(pe^{j\omega_0})z^{-1}}.
\end{aligned}$$

Claramente, este filtro tiene un polo en $z = pe^{j\omega_0}$ (y conserva el cero en $z = 0$). Sin embargo, dado que ahora el polo es complejo, la ecuación en diferencias correspondiente tendría coeficientes complejos y daría como salida una señal también compleja, lo cual resulta en una implementación menos eficiente.

Para asegurar que los coeficientes de la ecuación en diferencias sean todos reales, es necesario que los polos y los ceros sean reales, o bien que existan pares de polos (o ceros) conjugados. Por lo tanto, una manera de obtener una ecuación completamente real consiste en agregar polos y/o ceros adicionales correspondientes a los conjugados de los polos o ceros ya existentes. Para el caso del filtro pasa-banda anterior, se tiene ya un polo en $z = pe^{j\omega_0}$, por lo que es necesario agregar otro polo en $z = pe^{-j\omega_0}$ para obtener una ecuación con coeficientes reales. Por otra parte, dado que se está diseñando un filtro pasa-banda, puede ser deseable incorporar una atenuación adicional en las frecuencias cero y π, lo cual se logra colocando dos ceros en el eje real: uno cercano a $+1$ y otro cercano a -1. De esta manera, podemos escribir la función de transferencia del filtro pasa-bandas de dos polos como sigue:

$$H(z) = \frac{(z-c)(z+c)}{(z-pe^{j\omega_0})(z-pe^{-j\omega_0})}, \tag{6.1}$$

donde p es la magnitud de los polos y c es la magnitud de los ceros (los cuales están ubicados en $z = c$ y $z = -c$). Tanto p como c se asumen entre 0 y 1.

La ecuación en diferencias que corresponde a este filtro es:

$$y[n] = a_1 y[n-1] + a_2 y[n-2] + x[n] + b_2 x[n-2], \tag{6.2}$$

donde $a_1 = 2p\cos\omega_0$, $a_2 = -p^2$ y $b_2 = -c^2$.

La ganancia para frecuencias alrededor de la frecuencia de entonamiento ω_0 crece desmesuradamente conforme los polos se acercan al círculo unitario; es decir, cuando p se aproxima a uno. Por esta razón, este filtro se conoce como un filtro pasa-banda *resonante*. Una manera de controlar la resonancia consiste en atenuar la señal de entrada por un factor elegido adecuadamente para acotar la ganancia máxima del filtro $|H(e^{j\omega_0})|$, la cual está dada por

$$\begin{aligned} |H(e^{j\omega_0})| &= \left| \frac{(e^{j\omega_0}-c)(e^{j\omega_0}+c)}{(e^{j\omega_0}-pe^{j\omega_0})(e^{j\omega_0}-pe^{-j\omega_0})} \right| \\ &= \frac{|e^{j2\omega_0}-c^2|}{|(1-p)e^{j\omega_0}(e^{j\omega_0}-pe^{-j\omega_0})|} \\ &= \frac{1}{1-p} \cdot \frac{|e^{j2\omega_0}-c^2|}{|e^{j2\omega_0}-p|}. \end{aligned}$$

Si la magnitud de los ceros se elige como $c = \sqrt{p}$, entonces el segundo factor de la ecuación anterior se reduce a uno, y la ganancia máxima del filtro sería $1/(1-p)$. Por lo tanto, se puede compensar esta ganancia multiplicando la señal de entrada por un factor $1-p$; sin embargo, una compensación completa puede resultar en una gran pérdida de energía a la salida del filtro, por lo que suele ser buena idea introducir un factor de compensación variable de la forma

$(1-p)^s$ donde $s \in [0,1]$ controla la forma de la curva de compensación. La compensación puede aplicarse ya sea a la entrada o a la salida del filtro, pero por cuestiones de estabilidad numérica, suele ser conveniente compensar a la entrada; es decir, reemplazando $x[n]$ por $b_0 x[n]$, donde $b_0 = (1-p)^s$.

6.2.3 Calidad (Q) de un filtro pasa-banda resonante

La magnitud de los polos p controla la ganancia a la frecuencia central ω_0, la cual es la frecuencia donde se produce la ganancia máxima, y por lo tanto es la frecuencia de resonancia del filtro. Así mismo, p controla el ancho de banda del filtro: entre mayor sea p, mas angosta será la banda de paso. Para valores altos de p, el ancho de banda es aproximadamente $1-p$ (por qué?).

En realidad, el ancho de banda depende hasta cierto punto de la frecuencia de entonamiento, ya que la curva de respuesta en frecuencia se vuelve asimétrica conforme la frecuencia central se acerca a las frecuencias bajas o altas. Debido a esto, en muchas ocasiones, se especifica el factor de calidad Q de un filtro pasa-banda en lugar de su ancho de banda. El factor de calidad Q es un número estrictamente positivo que se obtiene como la razón entre la frecuencia central y el ancho de banda del filtro. De acuerdo con esta definición, la calidad del filtro definido en la sección anterior es aproximadamente $Q = \omega_0/(1-p)$. Dependiendo de la aplicación, podría ser deseable especificar la Q del filtro (y calcular p de manera acorde) para lograr una mayor consistencia a lo largo de todo el espectro.

6.2.4 Filtros rechaza-banda

Es también posible obtener filtros rechaza-banda de dos polos entonados a una frecuencia específica ω_0. Existen básicamente dos enfoques que se pueden seguir. El primero consiste en tomar la función de transferencia de un filtro pasa-banda con ganancia máxima unitaria $H_{\text{bp}}(z)$ y construir el filtro rechaza-banda como $H_{\text{br}}(z) = 1 - H_{\text{bp}}(z)$. El segundo enfoque consiste en modificar el filtro pasa-banda $H_{\text{bp}}(z)$ colocando los ceros justamente en el círculo unitario, en ángulos ω_0 y $-\omega_0$. Se puede verificar que ambos enfoques llevan esencialmente a la misma solución, y en ambos casos la magnitud p de los polos controla el grado de atenuación y lo angosto de la banda rechazada. La implementación de un filtro rechaza-banda se deja como reto al final de la práctica.

6.2.5 El filtro resonante como generador de tonos

Un filtro resonante con una calidad Q o una magnitud de polos p suficientemente alta, tendrá una banda de paso sumamente angosta, dejando pasar únicamente las componentes de frecuencia muy cercanas a ω_0. Si se alimenta a este filtro con una señal de entrada de banda ancha, como lo es el ruido blanco, la salida del filtro corresponderá, aproximadamente, a un tono aislado; es decir, una señal aproximadamente sinusoidal con frecuencia ω_0. De esta manera es posible emular, de manera rudimentaria, un generador de tonos pasando ruido blanco a través de un filtro pasa-bandas altamente resonante ($Q > 500$) entonado a la

frecuencia $\omega_0 = 2\pi f_0/f_m$, donde f_0 es la frecuencia deseada del oscilador en Hz, y f_m es la frecuencia de muestreo en Hz. Este oscilador no es perfecto ya que el filtro siempre deja pasar una cierta cantidad de ruido.

Por ejemplo, si se desea generar tonos musicales, es útil conocer que la octava se divide en doce semitonos (C, C#, D, D#, E, F, F#, G, G#, A, A#, B). La octava es una unidad de frecuencia relativa en escala logarítmica con base 2, ampliamente utilizada en audio, así como el decibel se utiliza para representar amplitudes relativas. La diferencia, en octavas, entre una frecuencia f y una frecuencia de referencia f_0 se calcula como $\log_2(f/f_0)$. Esta misma diferencia, expresada en semitonos, se obtiene como $n = 12\log_2(f/f_0)$. Si se conoce la frecuencia f_0 de una nota de referencia, entonces la frecuencia de otra nota que se encuentra a n semitonos de la referencia está dada por $f = f_0 2^{n/12}$. El nombre de una nota suele acompañarse del número de octava a la cual pertenece; por ejemplo, la nota A4 corresponde a la nota A (La) en la cuarta octava de un piano. Esta nota suele afinarse a 440 Hz, por lo que puede usarse como frecuencia de referencia: $f_{\mathrm{A4}} = 440$ Hz.

6.3 Objetivos didácticos

- Comprender el diseño e implementación de filtros pasa-banda resonantes de dos polos.
- Entender la forma en que se mide el ancho de banda y/o la calidad Q de un filtro pasa-banda, y su relación con los parámetros de diseño como la frecuencia central y la magnitud de los polos.
- Ser capaz de modificar el filtro pasa-banda para implementar otro tipo de procesos, como filtros rechaza-banda y generadores de tonos.

6.4 Material

- Una computadora con alguna de las siguientes combinaciones de software instalado:
  - Processing y Beads
  - Processing y Minim
  - GNU C++ (e.g., MinGW) y PortAudio
- Bocinas o audífonos estéreo

6.5 Procedimiento

El procedimiento para esta práctica es similar al de la práctica anterior:

1. Implementar el filtro pasa-banda resonante de dos polos como una nueva UGen llamada `FiltroPBR`. Esta UGen incluirá los métodos *set* y *get* necesarios para manipular los parámetros del filtro: frecuencia central, Q y compensación de ganancia, así como el constructor y la función callback para calcular la señal de salida.

2. Probar el filtro con una señal de entrada de prueba (e.g., ruido blanco) e implementar una interfaz que permita variar en tiempo real los principales parámetros del filtro: la frecuencia de entonamiento y la resonancia Q.

Al igual que en el caso del filtro de un polo, será necesario almacenar muestras anteriores de la señal de salida; sin embargo, dado que ahora se trata de un filtro de segundo orden, debemos conservar las últimas dos muestras de la salida, y también las últimas dos muestras de entrada, por cada canal. Estas se almacenarán en arreglos `ynm1[]` y `ynm2[]` para $y[n-1]$ y $y[n-2]$, respectivamente, así como `xnm1[]` y `xnm2[]` para $x[n-1]$ y $x[n-2]$.

La UGen contendrá, además, las variables de punto flotante `frecuencia`, `Q` y `comp`, que representan la frecuencia central f_c en Hz, la calidad Q y la compensación de ganancia s. También contendrá varios parámetros internos que se calcularán automáticamente a partir de los parámetros principales; estos son: `omega` (frecuencia central en radianes por muestra), `polo` (magnitud de los polos), `cero` (magnitud de los ceros), `ganancia` (la ganancia aplicada a la señal de entrada), y los coeficientes de la ecuación en diferencias `a1`, `a2` y `b2`. Todos los parámetros secundarios se calcularán (de acuerdo a la teoría en la Sección 6.2) en un método llamado `calculaCoeficientes()`, el cual es casi idéntico para las tres implementaciones. La única diferencia radica en la manera en que se obtiene la frecuencia de muestreo para cada plataforma. En el caso de Beads, se obtiene mediante el método `context.getSampleRate()` (donde `context` es el objeto de clase `AudioContext` que se pasa a todo UGen en el constuctor). En Minim simplemente se llama al método `sampleRate()` de la misma UGen, y finalmente, en C++ se llama al método estático `getSampleRate()` de la clase base `UGen`.

Dicho lo anterior, la implementación del método `calculaCoeficientes()` es la siguiente:

```
void calculaCoeficientes() {
  // reemplazar sampleRate() por el equivalente necesario
  omega = 2 * PI * frecuencia / sampleRate();
  polo = 1 - omega / Q;
  if (polo < 0) polo = 0;
  cero = sqrt(polo);
  a1 = 2 * polo * cos(omega);
  a2 = -polo * polo;
  b2 =  -cero * cero;
  ganancia = pow(1 - polo, comp);
}
```

Así mismo, las funciones *set* y *get* para manipular los parámetros principales, serán prácticamente idénticas en las tres implementaciones (salvo por la manera de obtener la frecuencia de muestreo). Todas las funciones *set* llaman además a `calculaCoeficientes()` para actualizar los parámetros secundarios cuando varía alguno de los parámetros principales:

```
float frecuencia() { return frecuencia; }

float Q() { return Q; }

float compensacion() { return comp; }

void setFrecuencia(float f) {
  if (f < 0) f = 0;
  // reemplazar sampleRate() por el equivalente necesario
  if (f > sampleRate() / 2) f = sampleRate() / 2;
  frecuencia = f;
  calculaCoeficientes();
}

void setQ(float q) {
  if (q < 1) q = 1;
  Q = q;
  calculaCoeficientes();
}

void setCompensacion(float c) {
  if (c < 0) c = 0;
  if (c > 1) c = 1;
  comp = c;
  calculaCoeficientes();
}
```

En todos los casos, la función callback implementa la ecuación en diferencias para un filtro de dos polos y dos ceros dada por la Ecuación 6.2. El siguiente pseudo-código ejemplifica la implementación para la muestra `i` del canal `c`, considerando el factor de ganancia aplicado a la señal de entrada:

```
in_sample = in[c][i] * ganancia;
out[c][i] = a1 * ynm1[c] + a2 * ynm2[c] + in_sample + b2 * xnm2[c];
ynm2[c] = ynm1[c];
ynm1[c] = out[c][i];
xnm2[c] = xnm1[c];
xnm1[c] = in_sample;
```

El resto de la implementación depende de la plataforma elegida.

6.5.1 Processing y Beads

La implementación en Beads no conlleva ninguna consideración adicional. Dentro de la clase `FiltroPBR` se agregan los miembros de datos que almacenarán los arreglos de muestras anteriores para cada canal, los parámetros principales del filtro, y los parámetros secundarios. Se deben incluyen los métodos comunes (ver sección anterior), así como un constructor para crear los arreglos e inicializar los parámetros, y la función callback que implementa la ecuación en diferencias.

```
public class FiltroPBR extends UGen {
  // arreglos de muestras anteriores para cada canal
  protected float[] ynm1, ynm2, xnm1, xnm2;

  // parametros principales del filtro
  protected float frecuencia, Q, comp;

  // parametros secundarios y coeficientes
  protected float omega, polo, cero, ganancia, a1, a2, b2;

  /*****************************************************************************/
  /*** Incluir metodos comunes aqui
  /*** Usar context.getSampleRate() para obtener la frecuencia de muestreo
  /*****************************************************************************/

  // constructor
  public FiltroPBR(AudioContext context, int canales) {
    super(context, canales, canales);
    ynm1 = new float[canales];
    ynm2 = new float[canales];
    xnm1 = new float[canales];
    xnm2 = new float[canales];
    frecuencia = context.getSampleRate() / 2;
    Q = 1;
    comp = 0.6;
    calculaCoeficientes();
  }

  // funcion callback
  public void calculateBuffer() {
    for (int c = 0; c < getOuts(); c++) {
      float[] in = bufIn[c];
      float[] out = bufOut[c];
      float in_sample;
      for (int i = 0; i < bufferSize; i++) {
        in_sample = in[i] * ganancia;
        out[i] = a1 * ynm1[c] + a2 * ynm2[c] + in_sample + b2 * xnm2[c];
```

```
        ynm2[c] = ynm1[c];
        ynm1[c] = out[i];
        xnm2[c] = xnm1[c];
        xnm1[c] = in_sample;
      }
    }
  }
}
```

Para el programa principal utilizaremos la misma interfaz que en la práctica anterior, pero con la diferencia de que la coordenada Y del mouse controlará la resonancia (Q) del filtro, en un rango de 1 a 50 en lugar de la ganancia del amplificador. Esto significa que el amplificador ya no será necesario, por lo que la arquitectura del sistema consiste simplemente de una fuente de ruido cuya salida se conectará al filtro pasa-banda resonante, y posteriormente a la salida física de audio.

```
AudioContext ac;
RuidoBlanco ruido;
FiltroPBR filtro;

void setup() {
  size(400, 300);

  ac = new AudioContext();
  ruido = new RuidoBlanco(ac, 2);
  filtro = new FiltroPBR(ac, 2);
  filtro.addInput(ruido);
  ac.out.addInput(filtro);
  ac.start();
}

void mouseMoved() {
  filtro.setQ(50 * (1 - (float)mouseY / height));
  filtro.setFrecuencia(pow(2, 14 * (float)mouseX / (width - 1)));
}

void draw() {
  background(0);
  fill(255);
  text("frecuencia = " + filtro.frecuencia(), 10, 10);
  text("Q = " + filtro.Q(), 10, 20);
  translate(0, -height / 4);
  stroke(0, 255, 0);
  simplescopio(filtro.getOutBuffer(0));
  translate(0, height / 2);
```

```
  stroke(255, 0, 255);
  simplescopio(filtro.getOutBuffer(1));
}
```

6.5.2 Processing y Minim

Para el caso de Minim, hay que recordar que el número de canales se desconoce al momento de construir una UGen, por lo que la creación de los arreglos de muestras anteriores debe realizarse en el método `channelCountChanged()`. De la misma manera, la frecuencia de muestreo no se conoce durante la construcción de la UGen, sino hasta que la UGen se conecta a otra, por lo que será necesario recalcular todos los parámetros secundarios cuando esto ocurra; eso podemos realizarlo dentro del método `sampleRateChanged()` de la clase UGen.

```
public class FiltroPBR extends UGen {
  UGenInput audio_in;

  // arreglos de muestras anteriores para cada canal
  protected float[] ynm1, ynm2, xnm1, xnm2;

  // parametros principales del filtro
  protected float frecuencia, Q, comp;

  // parametros secundarios y coeficientes
  protected float omega, polo, cero, ganancia, a1, a2, b2;

  /*************************************************************************/
  /*** Incluir metodos comunes aqui
  /*** Usar sampleRate() para obtener la frecuencia de muestreo
  /*************************************************************************/

  // constructor
  public FiltroPBR() {
    super();
    audio_in = new UGenInput(UGen.InputType.AUDIO);
    frecuencia = 1000;
    Q = 1;
    comp = 0.6;
  }

  // crear arreglos de muestras anteriores
  protected void channelCountChanged() {
    super.channelCountChanged();
    ynm1 = new float[channelCount()];
    ynm2 = new float[channelCount()];
    xnm1 = new float[channelCount()];
```

```
    xnm2 = new float[channelCount()];
  }

  // actualizar parametros secundarios cuando cambia la frecuencia de muestreo
  protected void sampleRateChanged() {
    super.sampleRateChanged();
    calculaCoeficientes();
  }

  // funcion callback
  protected void uGenerate(float[] channels) {
    int cc = channelCount();
    float[] in = audio_in.getLastValues();
    float in_sample;
    //a1 = 0;
    for (int c = 0; c < cc; c++) {
      in_sample = in[c] * ganancia;
      channels[c] = a1 * ynm1[c] + a2 * ynm2[c] + in_sample + b2 * xnm2[c];
      ynm2[c] = ynm1[c];
      ynm1[c] = channels[c];
      xnm2[c] = xnm1[c];
      xnm1[c] = in_sample;
    }
  }
}
```

La interfaz para el programa principal es similar a la versión de Beads.

```
Minim minim;
AudioOutput out;
RuidoBlanco ruido;
FiltroPBR filtro;

void setup() {
  size(400, 300);

  minim = new Minim(this);
  out = minim.getLineOut();
  ruido = new RuidoBlanco();
  filtro = new FiltroPBR();

  ruido.patch(filtro);
  filtro.patch(out);
}

void mouseMoved() {
```

```
  filtro.setQ(50 * (1 - (float)mouseY / height));
  filtro.setFrecuencia(pow(2, 14 * (float)mouseX / (width - 1)));
}

void draw() {
  background(0);
  fill(255);
  text("frecuencia = " + filtro.frecuencia(), 10, 10);
  text("Q = " + filtro.Q(), 10, 20);
  translate(0, -height / 4);
  stroke(0, 255, 0);
  simplescopio(out.left.toArray());
  translate(0, height / 2);
  stroke(255, 0, 255);
  simplescopio(out.right.toArray());
}
```

6.5.3 C++ y PortAudio

En C++ la inicialización de los arreglos de muestras anteriores se realiza en el constructor, pero en este caso es necesario agregar un destructor para liberar la memoria reservada para los arreglos. En la función callback, se realizan algunas comprobaciones adicionales para verificar que el número de canales de la UGen de entrada coincide con el número de canales de salida. Recordemos, además, que para obtener la frecuencia de muestreo simplemente hay que llamar al método `getSampleRate()` de la misma UGen.

```
class FiltroPBR : public UGen {
protected:
    // arreglos de muestras anteriores para cada canal
    float *ynm1, *ynm2, *xnm1, *xnm2;

    // parametros principales del filtro
    float frecuencia, Q, comp;

   // parametros secundarios y coeficientes
    float omega, polo, cero, ganancia, a1, a2, b2;

public:

    /*************************************************************************/
    /*** Incluir metodos comunes aqui
    /*** Usar getSampleRate() para obtener la frecuencia de muestreo
    /*************************************************************************/

    // constructor
```

```
FiltroPBR(int canales) : UGen(canales, 1) {
    ynm1 = new float[canales];
    ynm2 = new float[canales];
    xnm1 = new float[canales];
    xnm2 = new float[canales];
    memset(ynm1, 0, canales * sizeof(float));
    memset(ynm2, 0, canales * sizeof(float));
    memset(xnm1, 0, canales * sizeof(float));
    memset(xnm2, 0, canales * sizeof(float));
    frecuencia = getSampleRate() / 2;
    Q = 1;
    comp = 0.6;
    calculaCoeficientes();
}

// destructor
~FiltroPBR() {
    if (ynm1) delete[] ynm1;
    if (ynm2) delete[] ynm2;
    if (xnm1) delete[] xnm1;
    if (xnm2) delete[] xnm2;
    ynm1 = ynm2 = xnm1 = xnm2 = NULL;
}

// funcion callback
void processBuffer() {
    int numOuts = getNumOutputs();
    UGen *ugen = getInput(0);
    if (ugen == NULL) return;
    if (ugen->getNumOutputs() != numOuts) return;
    float *in = ugen->getBuffer();
    float *out = getBuffer();
    int N = getBufferSize() * numOuts;
    float in_sample;
    for (int i = 0; i < N; ) {
        for (int c = 0; c < numOuts; c++) {
            in_sample = in[i] * ganancia;
            out[i] = a1 * ynm1[c] + a2 * ynm2[c] + in_sample + b2 * xnm2[c];
            ynm2[c] = ynm1[c];
            ynm1[c] = out[i];
            xnm2[c] = xnm1[c];
            xnm1[c] = in_sample;
            i++;
        }
    }
}
```

```
};
```

Nuevamente, la interfaz en C++ será una versión simplificada para consola. Se controlará la frecuencia de entonamiento del filtro mediante las teclas numéricas ('0' a '9'), y la resonancia mediante las teclas '+' y '-', para incrementar o decrementar, respectivamente. Solamente se graficará el canal izquierdo de la señal de salida.

```
int main() {
    UGen::setup(44100, 512);

    RuidoBlanco *ruido = new RuidoBlanco(2);
    FiltroPBR *filtro = new FiltroPBR(2);
    UGen *input = new UGen(2);
    filtro->setInput(ruido);

    PaStream *stream;
    UGen *io[2];
    io[0] = NULL; io[1] = filtro;
    initializeAudio(&stream, io);

    char c = 0;
    do {
        if (_kbhit()) {
            c = _getch();
            if (c >= '0' && c <= '9') {
                float g = (float)(c - '0') / 10;
                filtro->setFrecuencia(pow(2, 12 * g + 3));
            }
            if (c == '+') {
                filtro->setQ(filtro->getQ() * 2);
            }
            if (c == '-') {
                filtro->setQ(filtro->getQ() / 2);
            }
        }
        simplescopio(filtro, 0);
        cout << "Frecuencia = " << filtro->getFrecuencia() << endl;
        cout << "Q = " << filtro->getQ() << endl;
    } while (c != 27);

    Pa_CloseStream(stream);
    Pa_Terminate();
    delete ruido;
    delete filtro;
    delete input;
```

```
    return 0;
}
```

6.6 Evaluación y reporte de resultados

1.- Escriba el desarrollo matemático para obtener la Ecuación 6.2 a partir de la Ecuación 6.1.

2.- Explique porqué el ancho de banda del filtro pasa-banda resonante de dos polos se puede estimar como $\beta = 1 - p$, donde p es la magnitud de los polos. (Sugerencia: recuerde que este filtro proviene de un filtro pasa-bajas).

3.- Imagine que la compensación de ganancia se aplica a la salida del filtro en lugar de aplicarse a la entrada. Es decir, suponga que la salida del filtro se calcula como

$$y[n] = (1 - p)\left(a_1 y[n-1] + a_2 y[n-2] + x[n] + b_2 x[n-2]\right),$$

donde $(1-p)$ es el factor de compensación de ganancia para un filtro con ganancia máxima unitaria.

El problema con este enfoque radica en que cuando p es cercano a uno (que por lo general lo es), un cambio relativamente pequeño en p puede ocasionar un cambio abrupto en la magnitud de la salida. Por ejemplo, si inicialmente $p = 0.99$, y en algún momento se cambia a $p = 0.98$, entonces la magnitud de la salida se duplicaría repentinamente, ocasionando artefactos audibles.

Explique porqué esto no ocurre cuando la compensación de ganancia se aplica a la entrada del filtro?

6.7 Retos

Con base en las ideas discutidas al final de la Sección 6.2, implemente un filtro rechaza-banda como una nueva clase de UGen. Implemente también un generador de onda senoidal como una nueva UGen sin entradas, que cuente con métodos *set* y *get* para manipular la frecuencia del oscilador. Si lo desea, incluya un método `setSemitono(int n)` que permita establecer la frecuencia del oscilador dado el número de semitonos relativos a la nota A4 (440 Hz), y trate de utilizar el oscilador para generar melodías.

6.8 Conclusiones

Redacte en el recuadro sus conclusiones acerca de los conceptos aprendidos, las actividades realizadas y los objetivos planteados en esta práctica. Si lo desea, puede considerar las preguntas de apoyo que se muestran abajo.

Preguntas de apoyo

- Qué procesos físicos se pueden modelar mediante un filtro pasa-banda resonante?
- Desde el punto de vista de diseño, cuál es la principal diferencia entre un filtro pasa-bajas o pasa-altas, y un filtro pasa-banda.
- Cuáles son los parámetros que caracterizan a un filtro pasa-banda resonante?

Capítulo 7

Filtros de ecualización

Nombre del estudiante	Calificación

7.1 Relación con los programas de estudio

Materia	Unidad y tema
Procesamiento Digital de Señales (IE / IT / IB)	1.3 - Sistemas discretos y sus características 1.4 - Sistemas lineales e invariantes en el tiempo 1.6 - Representación de sistemas LIT mediante ecuaciones en diferencias 5.1 - Definición de Transformada Z 5.3 - Transformada Z racional 5.4 - Propiedades de la Transformada Z 5.5 - Representación de sistemas LIT en el dominio Z 6.2 - Consideraciones para el diseño de filtros 6.4 - Diseño de filtros IIR
Procesamiento de Señales de Audio (IE)	1.5 - Sistemas lineales e invariantes en el tiempo 1.7 - Sistemas LIT dados como ecuaciones en diferencias 1.8 - Filtros FIR e IIR 1.9 - Respuesta en frecuencia 1.11 - Transformada Z 2.3 - Filtros de ecualización

7.2 Introducción

En el contexto de señales de audio, un *ecualizador* es un filtro que permite realizar ajustes al balance espectral de la señal. Estos ajustes, por lo general, se realizan realzando o atenuando bandas de frecuencia específicas. A diferencia de un filtro pasa-bajas, pasa-altas, o pasa-bandas, un ecualizador no suele estar diseñado para eliminar parte del contenido espectral de la señal; mas bien, se busca que la ganancia de un ecualizador sea aproximadamente unitaria en todas las frecuencias, salvo por la banda que se desea procesar.

Los ecualizadores tienen múltiples aplicaciones en la grabación y reproducción de audio y para el audio en vivo. Un ejemplo sencillo son los controles de tono en la mayoría de los estéreos caseros y para auto, que muchas veces cuentan con controles para realzar o atenuar las frecuencias graves, medias y agudas. También se utilizan para corregir problemas de acústica en un entorno debido a las ondas estacionarias y las propiedades de absorción de distintos materiales, para evitar problemas de retroalimentación mediante la atenuación de la frecuencia de resonancia, para pre-procesar las señales que serán transmitidas por radio con el objeto de optimizar el ancho de banda del medio, o bien, pre-procesar señales antes de ser grabadas en un medio analógico como los discos de acetato o vinilo.

Un ecualizador se construye a partir de uno o más filtros, cada uno de los cuales se encarga de procesar una banda de frecuencia específica. En este sentido existen dos tipos de filtros para ecualización: los filtros tipo *shelving* y los filtros tipo *peaking*. En un filtro *shelving* se divide el espectro en dos bandas: la banda inferior y la banda superior, determinadas por una frecuencia de corte; una de las dos bandas es procesada, realzándola o atenuándola por un cierto factor de ganancia, mientras que la otra banda se deja pasar con una ganancia aproximadamente igual a uno. En contraste, un filtro *peaking* afecta únicamente a las frecuencias alrededor de una frecuencia central, ya sea realzándolas o atenuándolas, y una ganancia aproximadamente unitaria para frecuencias alejadas de la frecuencia central [10].

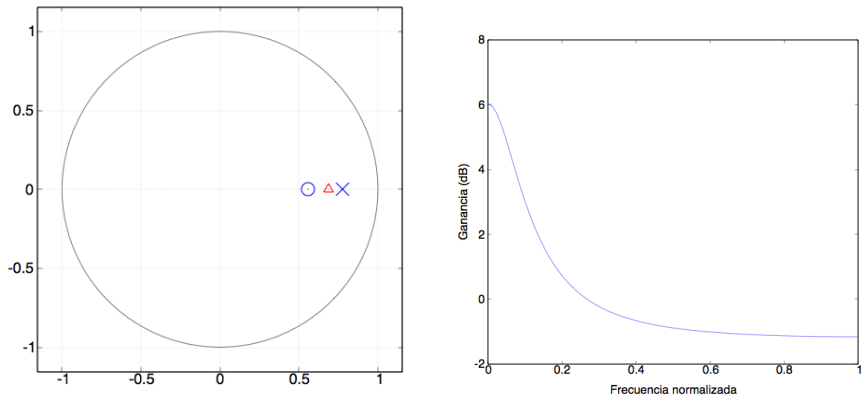
Claramente, existe una relación entre los filtros shelving y los filtros pasa-bajas y pasa-altas, así como entre los filtros peaking y los filtros pasa-banda y rechaza-banda. De hecho, es posible implementar filtros de ecualización de primer y segundo orden, similares a los que se desarrollaron en las dos prácticas anteriores, una vez que los ceros se colocan cerca de los polos.

7.2.1 Filtro de ecualización tipo shelving

Considere un filtro con un polo p y un cero c, ambos sobre la parte positiva del eje real, suficientemente cerca del círculo unitario, tal como se muestra en la Figura 7.1a. La función de transferencia de este filtro es

$$H(z) = \frac{z - c}{z - p}.$$

Suponga que la distancia $1 - p$ es el doble que la distancia $1 - c$ (tal como



(a) (b)

Figura 7.1: Filtro de ecualización tipo shelving con un ancho de banda $b = \pi/10$ y una ganancia $g = 2$: (a) Configuración de polo y cero para control de la banda inferior (frecuencias graves); el triángulo rojo se localiza en la posición $1 - b$, donde b es el ancho de la banda a procesar. (b) Respuesta en frecuencia del filtro (la frecuencia está normalizada en múltiplos de π).

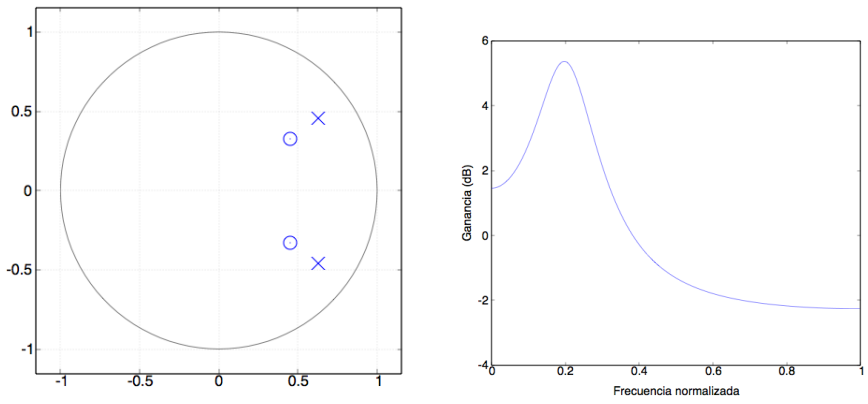
ocurre en la Figura 7.1a). En este caso, la función de transferencia evaluada en la frecuencia cero da como resultado una ganancia igual a 2 (correspondiente a 6 dB). Sin embargo, conforme se incrementa la frecuencia ω, las distancias $|z-c|$ y $|z-p|$, para un punto $z = e^{j\omega}$ en el círculo unitario, se vuelven similares, dando como resultado una ganancia aproximadamente unitaria (0 dB) para frecuencias por encima de una cierta frecuencia de corte b. En otras palabras, b representa el ancho de banda de la banda que se desea procesar.

Para el caso de un filtro pasa-bajas de un polo (ver Sección 6.2), la frecuencia de corte es aproximadamente $1 - p$, para valores altos de p ($p > 0.7$). En el caso de un filtro tipo shelving de un polo y un cero, el valor de la frecuencia de corte (o el ancho de banda) estaría entre $1 - c$ y $1 - p$. Si se desea obtener un filtro con un ancho específico dado por b, entonces se debe colocar el polo p y el cero c en lados opuestos de b, con $p > c$ si se desea realzar la banda de interés, o $p < c$ si se desea atenuar. Por ejemplo, si se desea tener una ganancia g en la frecuencia cero, uno puede elegir $p = 1 - b/\sqrt{g}$ y $c = 1 - b\sqrt{g}$. La Figura 7.1b muestra la respuesta en frecuencia de este filtro con $b = \pi/10$ y $g = 2$.

La ecuación en diferencias mediante la que se puede implementar este filtro se obtiene de manera directa multiplicando el numerador y denominador de $H(z)$ por z^{-1}, quedando como

$$y[n] = py[n-1] + x[n] - cx[n-1].$$

Si se desea procesar la banda alta en vez de la banda baja, simplemente hay



(a) (b)

Figura 7.2: Filtro de ecualización tipo peaking con una frecuencia central $\omega_0 = \pi/5$, un ancho de banda $b = \pi/10$ y una ganancia $g = 2$: (a) Configuración de polos y ceros. (b) Respuesta en frecuencia del filtro (la frecuencia está normalizada en múltiplos de π).

que invertir el polo y el cero multiplicándolos por -1.

7.2.2 Filtro de ecualización tipo peaking

Para obtener el filtro tipo peaking, partimos del filtro tipo shelving y aplicamos nuevamente un desplazamiento en frecuencia mediante la transformación $z \longrightarrow ze^{-j\omega_0}$, tal como se hizo en la práctica anterior para obtener el filtro pasa-banda a partir de un pasa-bajas. Esto da como resultado una rotación del polo y el cero a las posiciones $pe^{j\omega_0}$ y $ce^{j\omega_0}$, respectivamente. Dado que ahora el polo y el cero quedan fuera del eje real, tiene sentido agregar a sus conjugados complejos para obtener un filtro con coeficientes reales.

De esta manera, la función de transferencia para el filtro peaking queda como

$$\begin{aligned} H(z) &= \frac{(z - ce^{j\omega_0})(z - ce^{-j\omega_0})}{(z - pe^{j\omega_0})(z - pe^{-j\omega_0})} \\ &= \frac{z^2 - (2c\cos\omega_0)z + c^2}{z^2 - (2p\cos\omega_0)z + p^2} \\ &= \frac{1 - (2c\cos\omega_0)z^{-1} + c^2z^{-2}}{1 - (2p\cos\omega_0)z^{-1} + p^2z^{-2}}. \end{aligned}$$

Esto da lugar a la ecuación en diferencias

$$y[n] = a_1y[n-1] + a_2y[n-2] + x[n] + b_1x[n-1] + b_2x[n-2],$$

con $a_1 = 2p\cos\omega_0$, $a_2 = -p^2$, $b_1 = -2c\cos\omega_0$ y $b_2 = c^2$.

Nuevamente, la magnitud p de los polos y la magnitud c de los ceros pueden obtenerse a partir del ancho de banda b y ganancia g deseados, por lo que el filtro cuenta con tres parámetros: la frecuencia central ω_0, el ancho de banda b, y la ganancia g en la frecuencia central. La Figura 7.2 muestra un ejemplo de este filtro para $\omega_0 = \pi/5$, $b = \pi/10$ y $g = 2$.

7.2.3 Tipos de ecualizadores

Existen básicamente dos tipos de ecualizadores:

Gráficos.- Están constituidos por un banco de filtros entonados a frecuencias fijas y con anchos de banda fijos, de manera que el usuario solamente puede manipular la ganancia de cada filtro o banda. La mayoría de los filtros son de tipo peaking, salvo posiblemente por aquellos en los extremos del espectro. Un ecualizador gráfico profesional suele tener de 25 a 31 bandas para un control mas preciso, comúnmente espaciadas a intervalos de un tercio de octava.

Paramétricos.- Cuentan con un número reducido de bandas (comúnmente entre tres y cinco), pero es posible manipular todos los parámetros de cada banda, incluido el ancho de banda y la frecuencia central (en el caso de los filtros tipo peaking). Las bandas que representan los extremos inferior y superior del espectro suelen procesarse mediante filtros shelving, mientras que el resto de las bandas utilizan filtros peaking.

También es posible encontrar ecualizadores que combinan algunas bandas con frecuencia y/o ancho de banda fijos, junto con bandas (comúnmente las bandas centrales) variables. A este tipo de ecualizadores comúnmente se les llama *semi-paramétricos*.

7.3 Objetivos didácticos

- Comprender la naturaleza, diseño y aplicación de los filtros de ecualización
- Implementar filtros de ecualización de primer y segundo orden

7.4 Material

- Una computadora con alguna de las siguientes combinaciones de software instalado:
  - Processing y Beads
  - Processing y Minim
  - GNU C++ (e.g., MinGW) y PortAudio
- Bocinas o audífonos estéreo

7.5 Procedimiento

En esta práctica se desarrollarán dos nuevas UGens, llamadas `EQShelving` y `EQPeaking`. La implementación es muy similar a la de las clases `Filtro1P` y `FiltroPBR`, respectivamente. Cada clase contará con miembros para almacenar los parámetros fundamentales, así como los respectivos métodos *get* y *set* para manipularlos. También contarán con miembros para almacenar los coeficientes que se calculen a partir de los parámetros y un método donde se calcularán tales coeficientes. Todos estos elementos son prácticamente idénticos en las tres plataformas. Además se implementará el constructor y el método callback de cada clase, los cuales dependen de la plataforma elegida.

La clase `EQShelving` tendrá como parámetros el ancho de banda, la ganancia y una variable booleana que indicará si se procesa la banda baja o la banda alta; estos parámetros estarán representados por las variables `ancho`, `ganancia` y `bandaAlta`, respectivamente. A partir de estos parámetros se calcularán las posiciones del polo y el cero, dadas por las variables `polo` y `cero`. De esta manera, los métodos `set` y `get`, así como el método `calculaCoeficientes()` que serán comunes en todas las implementaciones pueden escribirse como:

```
float ancho() { return ancho; }
float ganancia() { return ganancia; }
boolean bandaAlta() { return bandaAlta; }

void setAncho(float a) {
  if (a < 0) a = 0;
  if (a > context.getSampleRate() / 4) a = context.getSampleRate() / 4;
  ancho = a;
  calculaCoeficientes();
}

void setGanancia(float g) {
  ganancia = g;
  calculaCoeficientes();
}

void setBandaAlta(boolean pa) {
  bandaAlta = pa;
  calculaCoeficientes();
}

void calculaCoeficientes() {
  // se asume ancho en Hz y ganancia en db
  float s = bandaAlta ? -1 : 1;
  float b = 2 * PI * ancho / context.getSampleRate();
  float g = pow(10, ganancia / 20);
  polo = (1 - b / sqrt(g)) * s;
```

```
    cero = (1 - b * sqrt(g)) * s;
  }
```

Recuerde que la obtención de la tasa de muestreo depende de la plataforma elegida. En el código anterior utiliza Beads, por lo que la tasa de muestreo se obtiene mediante `context.getSampleRate()`; mientras que en Minim y en C++ se debe llamar a `sampleRate()` y `getSampleRate()`, respectivamente.

Para el filtro `EQPeaking` se tienen como parámetros la frecuencia central, el ancho de banda y la ganancia.Para los dos últimos utilizaremos los mismos métodos `set` y `get` que para el filtro Shelving, agregando únicamente los métodos correspondientes a la frecuencia central, la cual se almacenará en la variable `frecuencia`:

```
  float frecuencia() { return frecuencia; }

  void setFrecuencia(float f) {
    if (f < 0) f = 0;
    if (f > context.getSampleRate() / 2) f = context.getSampleRate() / 2;
    frecuencia = f;
    calculaCoeficientes();
  }
```

Por otra parte, para el filtro Peaking se requiere calcular los coeficientes a_1, a_2, b_1, b_2 de la ecuación en diferencias, que se almacenarán en las variables `a1, a2, b1, b2` de la clase. Por lo tanto, la función `calculaCoeficientes()` para esta clase queda como sigue:

```
  void calculaCoeficientes() {
    // se asume frecuencia y ancho en Hz, ganancia en dB
    float omega = 2 * PI * frecuencia / context.getSampleRate();
    float b = 2 * PI * ancho / context.getSampleRate();
    float g = pow(10, ganancia / 20);
    polo = 1 - b / sqrt(g);
    cero = 1 - b * sqrt(g);
    a1 = 2 * polo * cos(omega);
    a2 = -polo * polo;
    b1 = -2 * cero * cos(omega);
    b2 = cero * cero;
  }
```

A continuación se muestra el código restante para cada una de las plataformas. Solamente se muestran las clases que definen las UGens `EQShelving` y `EQPeaking`. Se deja al alumno la elaboración de un programa de prueba que permita manipular los parámetros de los filtros mientras se procesa una señal de audio en tiempo real. Se sugiere utilizar un reproductor de música o un smartphone conectado a la entrada de audio, pero también es posible escuchar el efecto de los filtros en una señal de ruido blanco.

7.5.1 Processing y Beads

```
public class EQShelving extends UGen {
  protected float[] ynm1, xnm1;
  protected float ancho, ganancia;
  protected float polo, cero;
  protected boolean bandaAlta;

  // Agregar metodos set y get
  // asi como el metodo calculaCoeficientes

  // constructor
  public EQShelving(AudioContext context, int canales) {
    super(context, canales, canales);
    ynm1 = new float[canales];
    xnm1 = new float[canales];
    ancho = 1000;  // Hz
    ganancia = 0;  // db
    bandaAlta = false;
    calculaCoeficientes();
  }

  // funcion callback
  public void calculateBuffer() {
    for (int c = 0; c < getOuts(); c++) {
      float[] in = bufIn[c];
      float[] out = bufOut[c];
      for (int i = 0; i < bufferSize; i++) {
        out[i] = polo * ynm1[c] + in[i] - cero * xnm1[c];
        ynm1[c] = out[i];
        xnm1[c] = in[i];
      }
    }
  }
}
```

```
public class EQPeaking extends UGen {
  protected float[] ynm1, ynm2, xnm1, xnm2;
  protected float frecuencia, ancho, ganancia;
  protected float polo, cero, a1, a2, b1, b2;

  // Agregar metodos set y get
  // asi como el metodo calculaCoeficientes

  // constructor
  public EQPeaking(AudioContext context, int canales) {
    super(context, canales, canales);
```

```
    ynm1 = new float[canales];
    ynm2 = new float[canales];
    xnm1 = new float[canales];
    xnm2 = new float[canales];
    frecuencia = 1000;
    ancho = 100;
    ganancia = 0;
    calculaCoeficientes();
  }

  // funcion callback
  public void calculateBuffer() {
    for (int c = 0; c < getOuts(); c++) {
      float[] in = bufIn[c];
      float[] out = bufOut[c];
      for (int i = 0; i < bufferSize; i++) {
        out[i] = a1 * ynm1[c] + a2 * ynm2[c] + in[i] + b1 * xnm1[c] + b2 * xnm2[c];
        ynm2[c] = ynm1[c];
        ynm1[c] = out[i];
        xnm2[c] = xnm1[c];
        xnm1[c] = in[i];
      }
    }
  }
}
```

7.5.2 Processing y Minim

```
public class EQShelving extends UGen {
  protected float[] ynm1, xnm1;
  protected float ancho, ganancia;
  protected float polo, cero;
  protected boolean bandaAlta;
  UGenInput audio_in;

  // Agregar metodos set y get
  // asi como el metodo calculaCoeficientes

  public EQShelving() {
    super();
    audio_in = new UGenInput(UGen.InputType.AUDIO);
    ancho = 1000;  // Hz
    ganancia = 0;  // db
    bandaAlta = false;
    calculaCoeficientes();
  }
```

```
  protected void channelCountChanged() {
    super.channelCountChanged();
    ynm1 = new float[channelCount()];
    xnm1 = new float[channelCount()];
  }

  protected void sampleRateChanged() {
    super.sampleRateChanged();
    calculaCoeficientes();
  }

  protected void uGenerate(float[] channels) {
    int cc = channelCount();
    float[] in = audio_in.getLastValues();
    for (int c = 0; c < cc; c++) {
      channels[c] = polo * ynm1[c] + in[c] - cero * xnm1[c];
      ynm1[c] = channels[c];
      xnm1[c] = in[c];
    }
  }
}

public class EQPeaking extends UGen {
  protected float[] ynm1, ynm2, xnm1, xnm2;
  protected float frecuencia, ancho, ganancia;
  protected float polo, cero, a1, a2, b1, b2;
  UGenInput audio_in;

  // Agregar metodos set y get
  // asi como el metodo calculaCoeficientes

  public EQPeaking() {
    super();
    audio_in = new UGenInput(UGen.InputType.AUDIO);
    frecuencia = 1000;
    ancho = 100;
    ganancia = 0;
  }

  protected void channelCountChanged() {
    super.channelCountChanged();
    ynm1 = new float[channelCount()];
    ynm2 = new float[channelCount()];
    xnm1 = new float[channelCount()];
    xnm2 = new float[channelCount()];
```

```
  }

  protected void sampleRateChanged() {
    super.sampleRateChanged();
    calculaCoeficientes();
  }

  protected void uGenerate(float[] channels) {
    int cc = channelCount();
    float[] in = audio_in.getLastValues();
    for (int c = 0; c < cc; c++) {
      channels[c] = a1 * ynm1[c] + a2 * ynm2[c] + in[c] + b1 * xnm1[c] + b2 * xnm2[c];
      ynm2[c] = ynm1[c];
      ynm1[c] = channels[c];
      xnm2[c] = xnm1[c];
      xnm1[c] = in[c];
    }
  }
}
```

7.5.3 C++ y PortAudio

```
class EQShelving : public UGen {
protected:
    float *ynm1, *xnm1;
    float ancho, ganancia;
    float polo, cero;
    bool bandaAlta;

public:
    EQShelving(int canales) : UGen(canales, 1) {
        ynm1 = new float[canales];
        xnm1 = new float[canales];
        memset(ynm1, 0, canales * sizeof(float));
        memset(xnm1, 0, canales * sizeof(float));
        ancho = 1000;
        ganancia = 0;
        bandaAlta = false;
        calculaCoeficientes();
    }

    ~EQShelving() {
        if (ynm1) delete[] ynm1;
        if (xnm1) delete[] xnm1;
        ynm1 = xnm1 = NULL;
    }
```

```
  // Agregar metodos set y get
  // asi como el metodo calculaCoeficientes

    void processBuffer() {
        int numOuts = getNumOutputs();
        UGen *ugen = getInput(0);
        if (ugen == NULL) return;
        if (ugen->getNumOutputs() != numOuts) return;
        float *in = ugen->getBuffer();
        float *out = getBuffer();
        int N = getBufferSize() * numOuts;
        for (int i = 0; i < N; ) {
            for (int c = 0; c < numOuts; c++) {
                out[i] = polo * ynm1[c] + in[i] - cero * xnm1[c];
                ynm1[c] = out[i];
                xnm1[c] = in[i];
                i++;
            }
        }
    }
};

class EQPeaking : public UGen {
protected:
    float *ynm1, *ynm2, *xnm1, *xnm2;
    float frecuencia, ancho, ganancia;
    float polo, cero, a1, a2, b1, b2;

public:
    EQPeaking(int canales) : UGen(canales, 1) {
        ynm1 = new float[canales];
        ynm2 = new float[canales];
        xnm1 = new float[canales];
        xnm2 = new float[canales];
        memset(ynm1, 0, canales * sizeof(float));
        memset(ynm2, 0, canales * sizeof(float));
        memset(xnm1, 0, canales * sizeof(float));
        memset(xnm2, 0, canales * sizeof(float));
        frecuencia = 1000;
        ancho = 100;
        ganancia = 0;
        calculaCoeficientes();
    }

    ~EQPeaking() {
```

```
        if (ynm1) delete[] ynm1;
        if (ynm2) delete[] ynm2;
        if (xnm1) delete[] xnm1;
        if (xnm2) delete[] xnm2;
        ynm1 = ynm2 = xnm1 = xnm2 = NULL;
    }

   // Agregar metodos set y get
   // asi como el metodo calculaCoeficientes

    void processBuffer() {
        int numOuts = getNumOutputs();
        UGen *ugen = getInput(0);
        if (ugen == NULL) return;
        if (ugen->getNumOutputs() != numOuts) return;
        float *in = ugen->getBuffer();
        float *out = getBuffer();
        int N = getBufferSize() * numOuts;
        for (int i = 0; i < N; ) {
            for (int c = 0; c < numOuts; c++) {
                out[i] = a1 * ynm1[c] + a2 * ynm2[c] + in[i] + b1 * xnm1[c] + b2 * xnm2[c];
                ynm2[c] = ynm1[c];
                ynm1[c] = out[i];
                xnm2[c] = xnm1[c];
                xnm1[c] = in[i];
                i++;
            }
        }
    }
};
```

7.6 Evaluación y reporte de resultados

1.- Describa las principales diferencias entre un filtro de ecualización y un filtro pasa-bajas, pasa-altas o pasa-banda.

2.- Describa las diferencias entre un filtro tipo shelving y un filtro tipo peaking, así como los parámetros generales de cada tipo de filtro.

3.- Describa las diferencias entre un ecualizador gráfico y uno paramétrico.

4.- Enliste tres o más aplicaciones de los ecualizadores.

7.7 Retos

Escriba un programa que implemente un ecualizador gráfico de cinco bandas fijas utilizando las clases `EQShelving` y `EQPeaking` desarrolladas en la práctica. Las bandas deben ser aproximadamente las siguientes: Low-shelving con ancho de 100 Hz, tres bandas peaking entonadas a 300, 1000 y 4000 Hz, respectivamente, y con anchos de banda iguales a un tercio de la frecuencia central, de manera que se cubra aproximadamente todo el espectro, y una banda high-shelving con frecuencia de corte aproximadamente a 10 KHz (ojo: calcular el ancho de la banda). El programa debe tomar el audio de la entrada física, pasarlo por los cinco filtros en serie, y enviar la señal procesada a la salida de audio. Implemente una interfaz de usuario que permita manipular las ganancias de las cinco bandas en pasos de 1 db.

7.8 Conclusiones

Redacte en el recuadro sus conclusiones acerca de los conceptos aprendidos, las actividades realizadas y los objetivos planteados en esta práctica.

Capítulo 8

Estimación de amplitud

Nombre del estudiante	Calificación

8.1 Relación con los programas de estudio

Materia	Unidad y tema
Procesamiento Digital de Señales (IE / IT / IB)	1.3 - Sistemas discretos y sus características
Procesamiento de Señales de Audio (IE)	1.3 - Medidas y unidades de amplitud y frecuencia 4.6 - Procesadores de rango dinámico 5.6 - Aplicaciones a las interfaces humano-máquina

8.2 Introducción

Hasta este punto, las prácticas se han orientado principalmente al procesamiento básico de señales de audio (amplificación, filtrado y ecualización). Sin embargo, muchas aplicaciones requieren del *análisis* de las señales; es decir, la obtención de rasgos característicos de la señal los cuales, por lo general, varían a lo largo del tiempo. Una de las principales herramientas de análisis es la estimación de la amplitud de una señal. Recordemos que la amplitud está asociada con la intensidad o volumen, una de las propiedades fundamentales de los sonidos.

Si bien los valores de una señal $x[n]$ representan amplitudes (por ejemplo, la amplitud del voltaje que se tiene a la entrada de un convertidor análogo-digital), éstos no proporcionan directamente una medida de la intensidad o volumen de una señal. Mas bien, la amplitud de una señal periódica está relacionada con

el rango de valores de la señal a lo largo de uno o más periodos. Para ejemplificar esto, consideremos la señal $x[n] = A\cos(2\pi fn/f_m)$ con una frecuencia de muestreo f_m y $A \geq 0$. Aunque la señal varía en el tiempo, la intensidad de la señal está asociada al factor A, el cual representa la amplitud de las oscilaciones, y se mantiene constante. Para una señal arbitraria, la amplitud en general no permanece constante, por lo que es necesario estimarla como función del tiempo.

Por lo tanto, el objetivo de la estimación de amplitud es calcular la intensidad, a lo largo del tiempo, a partir de una señal $x[n]$ arbitraria. En esta práctica se presentarán tres técnicas para la estimación de amplitud y se implementarán dos de ellas.

8.2.1 Aplicaciones

Dado que la amplitud se corresponde con uno de los principales rasgos del sonido, existe un gran número de aplicaciones que hacen uso de esta característica. Una de las principales tareas consiste en detectar el inicio y fin de un sonido, lo cual se hace estableciendo umbrales de amplitud. De esta manera, un sistema de reconocimiento del habla puede detectar y separar los distintos fonemas para su posterior análisis. Por otra parte, la amplitud de cada fonema permite obtener información asociada con el estado emocional del hablante, cosa que no es posible obtener a partir del texto escrito.

En el campo del audio profesional es común un proceso conocido como *compresión* el cual tiene como objetivo controlar el rango dinámico de una señal. Un compresor analiza la señal de entrada para estimar su amplitud, y si la amplitud sobrepasa un cierto umbral, entonces la señal es atenuada por un factor inversamente proporcional a la amplitud; es decir, entre más intensa es la señal, mayor es la atenuación que se le aplica. La compresión es importante durante la grabación y transmisión de señales para evitar que la señal sature el medio.

En otra aplicación de la compresión, conocida como *ducking*, la señal que recibe el analizador de amplitud proviene de una entrada distinta (llamada *sidechain*) a la señal que será procesada; cuando la señal *sidechain* aumenta de intensidad, la señal procesada será atenuada. Esto se utiliza comúnmente en la radio y la televisión para atenuar automáticamente la música de fondo cuando un locutor comienza a hablar.

8.2.2 Estimación de amplitud por bloques

Una de las maneras mas comunes para estimar la amplitud A de una señal discreta $x[n]$ al tiempo n consiste en calcular el valor absoluto máximo de las muestras dentro de una ventana de tiempo centrada alrededor de n [10]:

$$A_{\text{peak}}[n] = \max_{n-W/2 \leq j \leq n+W/2} \{|x[j]|\}, \tag{8.1}$$

donde W es el tamaño de ventana. A este estimador se le conoce como la *amplitud pico* (peak amplitude). Para una señal del tipo $x[n] = A\cos(2\pi fn/f_m)$, A_{peak} es exactamente igual a A si W es lo suficientemente grande como para

abarcar por lo menos un periodo de la señal. Para una señal en general, el tamaño de ventana W está asociado con la frecuencia mas baja que uno desea analizar. Por ejemplo, considerando que el rango de frecuencias audibles (para los humanos) es de 20 Hz a 20 KHz, uno podría verse tentado a utilizar un ancho de ventana W igual al periodo de una señal de 20 Hz, es decir $50ms$, que a una frecuencia de muestreo de 44100 Hz corresponde a $W = 2205$ muestras. Esto representa un alto costo computacional (o una alta latencia), por lo que en la práctica se suelen utilizar ventanas de menor tamaño, sacrificando la sensibilidad del análisis a bajas frecuencias.

Otro estimador comúnmente utilizado es el estimador *pico a pico*, el cual mide la diferencia entre el valor máximo y el valor mínimo de la señal en la ventana de tiempo:

$$A_{\text{pp}}[n] = \max_{n-W/2 \leq j \leq n+W/2} \{x[j]\} - \min_{n-W/2 \leq j \leq n+W/2} \{x[j]\}. \tag{8.2}$$

A la amplitud pico a pico también se le conoce como el *rango dinámico* de la señal. Note que para una señal senoidal, A_{pp} es el doble de A_{peak}. Por otra parte, el estimador pico a pico tiene la ventaja de ser insensible a la componente DC que pueda presentar la señal.

Los estimadores pico y pico-a-pico son muy sensibles a ciertos artefactos como el ruido impulsivo y las discontinuidades, los cuales son frecuentes en las señales de audio; por ejemplo, el ruido que se produce al conectar o desconectar una guitarra de un amplificador cuando éste sigue encendido, es de tipo impulsivo. Una alternativa menos sensible a estos artefactos consiste en estimar la energía de la señal en la ventana de tiempo, dando lugar al estimador RMS (root mean square):

$$A_{\text{rms}}[n] = \sqrt{\frac{1}{W+1} \left(\sum_{j=n-W/2}^{n+W/2} |x[j]|^2 \right)}. \tag{8.3}$$

Al igual que A_{peak}, el estimador RMS es sensible a la componente DC presente en la señal. Para solventar esto, es recomendable primero pasar la señal por un filtro pasa-altas con una frecuencia de corte menor a 20 Hz para eliminar la componente DC, para posteriormente estimar la amplitud. Para una señal senoidal con amplitud A, la amplitud RMS es igual a $A/\sqrt{2}$ para una ventana de suficiente tamaño.

En la práctica no es posible utilizar directamente los estimadores anteriores, ya que dependen de muestras posteriores al tiempo n. Mas bien, el estimador se calcula en una ventana de tiempo desde $n - W$ hasta n siendo conscientes de que la estimación obtenida tendrá un retardo de $W/2$ muestras. Una gran ventaja del estimador RMS radica en que es relativamente sencillo de actualizar del tiempo n al tiempo $n + 1$ si uno conserva siempre el valor de la suma de los cuadrados de las muestras de la señal dentro de la ventana de tiempo, y simplemente se actualiza esta suma agregando $x[n]^2$ (la nueva muestra que entra a la ventana) y restando $x[n-W-1]^2$ (la muestra que sale de la ventana).

Es necesario, sin embargo, implementar una línea de retardo para obtener $x[n - W - 1]$. Las líneas de retardo, y sus aplicaciones, se estudiarán en las prácticas de niveles intermedio y avanzado.

8.3 Objetivos didácticos

- Conocer y comprender algunas de las técnicas para estimar la amplitud de una señal de audio.
- Implementar la estimación de amplitud para aplicaciones específicas.

8.4 Material

- Una computadora con alguna de las siguientes combinaciones de software instalado:
  - Processing y Beads
  - Processing y Minim
  - GNU C++ (e.g., MinGW) y PortAudio
- Bocinas o audífonos estéreo
- Micrófono integrado o externo, o algún dispositivo para reproducción de audio a través de la entrada auxiliar.

8.5 Procedimiento

En esta práctica se implementará una nueva UGen llamada `Amplimetro` para estimar la amplitud de una señal de entrada por bloques, ya sea estimando la amplitud pico o la amplitud RMS (la estimación de la amplitud pico a pico se deja como ejercicio al final de la práctica). Por simplicidad, la amplitud se estimará para cada bloque de procesamiento de audio, obteniendo una estimación de amplitud por bloque, en lugar de una estimación en cada instante de tiempo. Estimar la amplitud en cada instante de tiempo conlleva ciertas dificultades técnicas, sobre todo si se quiere hacer de manera eficiente. Afortundamente, en muchas aplicaciones no es necesario estimar la amplitud en cada instante, y para aquellas aplicaciones donde sí es necesario, puede ser suficiente estimar la amplitud en cada bloque e interpolar los valores de amplitud a lo largo del bloque.

En todas las plataformas, la clase `Amplimetro` contará con un arreglo de variables de punto flotante, llamado `amplitud`, donde se guardará la amplitud estimada para cada canal, así como una variable entera, llamada `tipo`, que servirá para determinar la manera en que se estimará la amplitud (pico o RMS), junto con sus correspondientes métodos *set* y *get*. La amplitud se calculará dentro de la función callback y se almacenará en el arreglo `amplitud`. Además,

se implementará un método llamado `getAmplitud()` para obtener en cualquier momento el valor actualizado de la amplitud para un canal dado.

Como aplicación de ejemplo, se implementará un espectrógrafo rudimentario a partir de un banco de filtros pasa-banda (mediante la UGen `FiltroPBR` desarrollada en la práctica 6). La señal de entrada se tomará de la entrada física de audio (micrófono o entrada auxiliar) y se enviará de manera paralela a cinco filtros pasabanda entonados a distintas frecuencias. Luego, la salida de cada filtro se enviará a una UGen `Amplimetro` para estimar y graficar la energía en cada una de las bandas.

Para cubrir un rango amplio del espectro, se entonarán los filtros por octavas a partir de 400 Hz. Es decir, las frecuencias centrales de los filtros serán 400, 800, 1600, 3200 y 6400 Hz, respectivamente. Particularmente los micrófonos integrados a las computadoras tienen poca sensibilidad fuera de este rango de frecuencia, ya que están diseñados para voz y teleconferencia. Además, utilizaremos filtros resonantes con un factor de calidad ligeramente alto ($Q \approx 5$) con compensación completa de ganancia para asegurar una ganancia unitaria en la banda de interés y una atenuación pronunciada fuera de esta banda.

En primera instancia se definirá una constante `NB` para representar el número de bandas y un arreglo `frecuencia` de punto flotante para almacenar las frecuencias centrales de los filtros. Los filtros se almacenarán en un arreglo de objetos `FiltroPBR` llamado `filtro`, así como los estimadores de amplitud en un arreglo de clase `Amplimetro` llamado `am`. También se requerirá una UGen para representar la entrada física de audio (Práctica 4). Durante la actualización de la pantalla, el programa dibujará una serie de `NB` barras cuya altura será proporcional a la amplitud estimada en cada banda.

Existe un detalle técnico que se deberá resolver de manera particular para cada plataforma. En las tres plataformas se conecta una UGen U a la salida física de audio (e.g., las bocinas). Cuando llega el momento de calcular el bloque de salida, el sistema de audio llama a la función callback de la UGen U conectada a la salida, que a su vez llama a las funciones callback de las UGen conectadas a U y así sucesivamente hasta las UGen generadoras, formando así un árbol de llamadas a las funciones callback. Sin embargo, cualquier UGen que no forme parte del árbol que se origina al procesar la salida física de audio, nunca será procesada. Esto significa que debemos buscar la manera de conectar las salidas de todos las UGen de tipo `Amplimetro` a una sola UGen que se conecte a la salida física de audio, de manera que todas las cadenas de procesamiento (correspondientes a las distintas bandas de frecuencia que se desean procesar) formen parte del árbol de llamadas. En la sección correspondiente a cada plataforma se discutirá una manera de resolver esta situación.

8.5.1 Processing y Beads

A continuación se muestra el código de la UGen `Amplimetro` para Beads. Cabe notar que esta UGen no cuenta con canales de salida (esto se aprecia en la llamada al constructor de la clase base), dado que se trata de una UGen que solamente analiza el audio.

```
public class Amplimetro extends UGen {
  static final int PEAK = 0;
  static final int RMS = 1;

  float[] amplitud;
  int tipo;

  public Amplimetro(AudioContext context, int canales) {
    super(context, canales, 0);
    amplitud = new float[canales];
    Arrays.fill(amplitud, 0);
    tipo = PEAK;
  }

  void setTipo(int t) { tipo = t; }

  int tipo() { return tipo; }

  float getAmplitud(int canal) {
    return (canal >= 0 && canal < getIns()) ? amplitud[canal] : 0;
  }

  public void calculateBuffer() {
    float max;
    for (int c = 0; c < getIns(); c++) {
      float[] in = bufIn[c];
      switch (tipo) {
        case PEAK:
          max = 0;
          for (int i = 0; i < bufferSize; i++) {
            if (abs(in[i]) > max) max = abs(in[i]);
          }
          amplitud[c] = max;
          break;

        case RMS:
          amplitud[c] = 0;
          for (int i = 0; i < bufferSize; i++) {
            amplitud[c] += in[i] * in[i];
          }
          amplitud[c] = sqrt(amplitud[c] / bufferSize);
          break;
      }
    }
  }
}
```

La inicialización que se realiza en la función `setup()` se encarga de construir e inicializar las UGen, ruteando la entrada física de audio hacia el banco de filtros, y cada uno de los filtros hacia un `Amplimetro`. En Beads, las UGen cuentan con un mecanismo para procesar UGens que no necesariamente se encuentran en la cadena de procesamiento de audio. A este tipo de UGens se les llama *dependientes*, y pueden agregarse mediante el método `addDependent()` de la clase UGen. Por otra parte, la clase `AudioContext`, que representa el sistema de audio, cuenta con un miembro llamado `out` que representa a la salida física. Para nuestra aplicación, agregaremos todos los objetos de clase `Amplimetro` como dependientes de la UGen `out`.

```
int NB = 5;
float[] frecuencia = { 400, 800, 1600, 3200, 6400 };
AudioContext ac;
FiltroPBR[] filtro;
Amplimetro[] am;
UGen audioInput;

void setup() {
  size(800, 600);

  ac = new AudioContext();
  audioInput = ac.getAudioInput();

  filtro = new FiltroPBR[NB];
  am = new Amplimetro[NB];

  for (int i = 0; i < NB; i++) {
    filtro[i] = new FiltroPBR(ac, 2);
    filtro[i].setFrecuencia(frecuencia[i]);
    filtro[i].setQ(5);
    filtro[i].setCompensacion(1);
    filtro[i].addInput(audioInput);

    am[i] = new Amplimetro(ac, 2);
    am[i].addInput(filtro[i]);
    am[i].setTipo(Amplimetro.PEAK);
    ac.out.addDependent(am[i]);
  }
  ac.start();
}
```

Finalmente, la función `draw()` actualizará la pantalla dibujando la forma de onda y superponiendo las barras que indicarán la amplitud en cada banda de frecuencia. Dado que el procesamiento se realiza en estereo (2 canales), se promediará la amplitud de ambos canales para determinar la altura de cada barra.

```
void draw() {
  background(0);
  fill(255);

  pushMatrix();
  translate(0, -height / 4);
  stroke(0, 255, 0);
  simplescopio(audioInput.getOutBuffer(0));
  translate(0, height / 2);
  stroke(255, 0, 255);
  simplescopio(audioInput.getOutBuffer(1));
  popMatrix();

  float dx = width / NB;
  float xi, y, a;
  textAlign(CENTER, CENTER);
  for (int i = 0; i < NB; i++) {
    xi = dx * i;
    a =  0.5 * (am[i].getAmplitud(0) + am[i].getAmplitud(1));
    y = a * (height - 100);
    stroke(255, 255, 0, a * 128 + 64);
    fill(200, 200, 0, a * 200);
    rect(xi + 1, height - 50 - y, dx - 2, y, 10);
    fill(255);
    text(frecuencia[i] + " Hz", xi + dx / 2, height - 50 );
  }
}
```

8.5.2 Processing y Minim

La implementación de un estimador de amplitud en Minim nos lleva rápidamente a un problema técnico: las UGen de Minim están diseñadas para procesar una muestra a la vez, por lo que dentro del método callback no contamos con acceso a todo un bloque de muestras a partir del cual podamos calcular la amplitud. Una manera de solventar esto es mediante la implementación de una UGen auxiliar que guarde cada muestra de la señal en un buffer de un cierto tamaño previamente definido. Una vez que se llene el buffer, el contenido de éste se copiará en otro arreglo, al que el programador tendrá acceso, mientras el primer buffer vuelve a llenarse. Esta técnica se conoce como buffereo doble (double buffering).

En el Apéndice C se presenta la implementación de un buffer doble como una UGen llamada `Buffer`, junto con una descripción de los miembros y métodos de la clase. En esta práctica se asumirá que se cuenta ya con esta implementación. Como aplicación adicional, se puede usar la UGen `Buffer` para graficar la forma de onda de cualquier UGen, no solamente de la salida física de audio. Utilizaremos esta técnica para visualizar la señal de entrada, ya que la señal de salida

será nula.

La clase `Amplimetro` heredará directamente de la clase `Buffer`, agregando la funcionalidad requerida para la estimación de amplitud. El cálculo de la amplitud se realiza cada vez que se llena el buffer de llenado y su contenido se transfiere al buffer de lectura `last_buffer`; esto ocurre cuando la variable `index`, que se utiliza para indexar el buffer de llenado, se reinicializa a cero dentro de la función callback `UGenerate()`.

```
public class Amplimetro extends Buffer {
  static final int PEAK = 0;
  static final int RMS = 1;

  float[] amplitud;
  int tipo;

  void setTipo(int t) { tipo = t; }

  int tipo() { return tipo; }

  float getAmplitud(int canal) {
    return (canal >= 0 && canal < channelCount()) ? amplitud[canal] : 0;
  }

  Amplimetro(int s) {
    super(s);
    tipo = PEAK;
  }

  protected void channelCountChanged() {
    super.channelCountChanged();
    amplitud = new float[channelCount()];
    Arrays.fill(amplitud, 0);
  }

  void uGenerate(float[] channels) {
    super.uGenerate(channels);

    if (index == 0) {
      float max, suma;
      for (int c = 0; c < channelCount(); c++) {
        switch (tipo) {
          case PEAK:
            max = 0;
            for (int i = 0; i < size; i++) {
              if (abs(last_buffer[c][i]) > max) max = abs(last_buffer[c][i]);
            }
```

```
            amplitud[c] = max;
            break;

          case RMS:
            suma = 0;
            for (int i = 0; i < size; i++) {
              suma += last_buffer[c][i] * last_buffer[c][i];
            }
            amplitud[c] = sqrt(suma / size);
            break;
        }
      }
    }
  }
}
```

Para lograr que múltiples cadenas de UGens sean procesadas, Minim cuenta con una UGen llamada `Sink` a la que pueden conectarse las UGens que uno desea procesar, y la cual a su vez se conecta a la salida de audio. La UGen `Sink` genera silencio, lo cual es adecuado para nuestra aplicación; sin embargo, Minim cuenta otra UGen llamada `Summer` que da como salida la suma de todas las UGens conectadas a ella.

Además, utilizaremos una UGen de clase `Buffer` para capturar bloques de la señal de entrada, de manera que sea posible visualizarla mediante la función `simplescopio()`. Para asegurar que el buffer se actualiza constantemente, conectaremos el objeto `Buffer` al la UGen `Sink`.

Aunque el tamaño de los buffers puede elegirse libremente, utilizaremos el mismo tamaño que los bloques de procesamiento del sistema de audio, el cual puede obtenerse mediante el método `bufferSize()` de la clase `AudioOutput`.

De esta manera, la inicialización del espectrógrafo puede quedar como sigue:

```
Minim minim;

AudioOutput out;
LiveInput  input;
Buffer buffer;

int NB = 5;
float[] frecuencia = { 400, 800, 1600, 3200, 6400 };
FiltroPBR[] filtro;
Amplimetro[] am;
Sink sink;

void setup() {
  size(800, 600);
```

```
  minim = new Minim(this);
  out = minim.getLineOut();

  AudioStream stream = minim.getInputStream(
    out.getFormat().getChannels(), out.bufferSize(),
    out.sampleRate(), out.getFormat().getSampleSizeInBits());

  input = new LiveInput(stream);
  sink = new Sink();
  buffer = new Buffer(out.bufferSize());
  input.patch(buffer).patch(sink).patch(out);

  filtro = new FiltroPBR[NB];
  am = new Amplimetro[NB];
  for (int i = 0; i < NB; i++) {
    filtro[i] = new FiltroPBR();

    am[i] = new Amplimetro(out.bufferSize());
    am[i].setTipo(Amplimetro.PEAK);
    input.patch(filtro[i]).patch(am[i]).patch(sink);

    filtro[i].setFrecuencia(frecuencia[i]);
    filtro[i].setQ(5);
    filtro[i].setCompensacion(1);
  }
}
```

Cabe notar que el método `patch()` de las UGen de Minim puede encadenarse para efectuar múltiples conexiones en una misma línea de código, por ejemplo:

```
input.patch(buffer).patch(sink).patch(out);
```

Esto es posible ya que `patch()` devuelve una referencia a la UGen que recibe como argumento. Por otra parte, es necesario resaltar que el orden en que se efectúan las conexiones es de suma importancia. Una UGen de Minim desconoce la frecuencia de muestreo y el número de canales con los que debe trabajar hasta que es conectada a una cadena que termina en la salida física de audio. En el caso de los filtros pasabanda resonantes (clase `FiltroPBR`), los parámetros del filtro dependen de la frecuencia de muestreo, por lo que conviene fijarlos hasta después de haber realizado todas las conexiones entre UGens.

Finalmente, la función `draw()` visualiza la señal de entrada a partir del buffer y posteriormente dibuja una serie de barras cuya altura es proporcional a la amplitud de cada una de las bandas. La amplitud para cada banda se calcula promediando las amplitudes de ambos canales (izquierdo y derecho).

```
void draw() {
  background(0);
```

```
  fill(255);

  pushMatrix();
  translate(0, -height / 4);
  stroke(0, 255, 0);
  if (buffer.getChannelBuffer(0) != null) simplescopio(buffer.getChannelBuffer(0));
  translate(0, height / 2);
  stroke(255, 0, 255);
  if (buffer.getChannelBuffer(1) != null) simplescopio(buffer.getChannelBuffer(1));
  popMatrix();

  float dx = width / NB;
  float xi, y, a;
  textAlign(CENTER, CENTER);
  for (int i = 0; i < NB; i++) {
    xi = dx * i;
    a =  0.5 * (am[i].getAmplitud(0) + am[i].getAmplitud(1));
    y = a * (height - 100);
    stroke(255, 255, 0, a * 128 + 64);
    fill(200, 200, 0, a * 200);
    rect(xi + 1, height - 50 - y, dx - 2, y, 10);
    fill(255);
    text(frecuencia[i] + " Hz", xi + dx / 2, height - 50 );
  }
}
```

8.5.3 C++ y PortAudio

La implementación de la UGen `Amplimetro` en C++ es muy similar a la de Beads. El constructor crea una UGen con una entrada y un número determinado de canales de salida (especificado como argumento del constructor). Si bien esta UGen no generará una señal de salida, es necesario especificar el número de canales de la UGen de entrada que se desean procesar. Así mismo, se tienen funciones *set* y *get* para manipular el tipo de estimación (pico o RMS), así como una función para obtener la amplitud estimada para cualquier canal de la señal.

```
class Amplimetro : public UGen {
protected:
    float *amplitud;
    int tipo;

public:
    enum { PEAK = 0, RMS = 1 };

    Amplimetro(int canales) : UGen(canales, 1) {
        amplitud = new float[canales];
```

```
    memset(amplitud, 0, canales * sizeof(float));
    tipo = PEAK;
}

void setTipo(int t) { tipo = t; }

int getTipo() { return tipo; }

float getAmplitud(int canal) {
    return amplitud[canal % getNumOutputs()];
}

void processBuffer() {
    int numOuts = getNumOutputs();
    UGen *ugen = getInput(0);
    if (ugen == NULL) return;
    if (ugen->getNumOutputs() != numOuts) return;
    float *in = ugen->getBuffer();
    float *out = getBuffer();
    int N = getBufferSize() * numOuts;

    for (int c = 0; c < numOuts; c++) amplitud[c] = 0;

    switch (tipo) {
        case PEAK:
            for (int i = 0; i < N; ) {
                for (int c = 0; c < numOuts; c++) {
                    if (fabs(in[i]) > amplitud[c]) amplitud[c] = fabs(in[i]);
                    i++;
                }
            }
            break;

        case RMS:
            for (int i = 0; i < N; ) {
                for (int c = 0; c < numOuts; c++) {
                    amplitud[c] += in[i] * in[i];
                    i++;
                }
            }
            for (int c = 0; c < numOuts; c++) {
                amplitud[c] = sqrt(amplitud[c] / getBufferSize());
            }
            break;
    }
    memset(out, 0, N * sizeof(float));
```

```
    }
};
```

A continuación se presenta la implementación del espectrograma en C++. De manera similar al caso de Processing, se utilizan dos arreglos de UGens, uno de clase `FiltroPBR` y otro de clase `Amplimetro`. La entrada de audio se conectará a cada uno de los filtros pasa-banda resonantes, cada uno de los cuales irá conectado a un amplímetro. Para asegurar que todos los amplímetros sean procesados, se conectarán a un objeto UGen genérico (llamado `sink`) con tantas entradas como el número de bandas; finalmente, este UGen se conectará a la salida física de audio.

La visualización de las amplitudes de las bandas se simplifica considerablemente para una ventana de consola. Por cada banda se imprime una línea horizontal formada por guiones, cuya longitud es proporcional a la amplitud de la banda correspondiente (o mas precisamente, al promedio de las amplitudes de los canales izquierdo y derecho). Las operaciones de entrada y salida a la consola son relativamente lentas, por lo que no se visualizará la forma de onda.

```
int main() {
    UGen::setup(44100, 512);

    const int NB = 5;
    float frecuencia[NB] = { 400, 800, 1600, 3200, 6400 };

    UGen *input = new UGen(2);
    FiltroPBR *filtro[NB];
    Amplimetro *am[NB];
    UGen *sink = new UGen(2, NB);

    for (int i = 0; i < NB; i++) {
        filtro[i] = new FiltroPBR(2);
        filtro[i]->setFrecuencia(frecuencia[i]);
        filtro[i]->setQ(5);
        filtro[i]->setCompensacion(1);
        filtro[i]->setInput(input);

        am[i] = new Amplimetro(2);
        am[i]->setInput(filtro[i]);
        am[i]->setTipo(Amplimetro::PEAK);

        sink->setInput(i, am[i]);
    }

    PaStream *stream;
    UGen *io[2];
    io[0] = input; io[1] = sink;
```

```
    initializeAudio(&stream, io);

    char c = 0;
    do {
        if (_kbhit()) {
            c = _getch();
        }

        system("cls");

        // Dibujar bandas
        for (int i = 0; i < NB; i++) {
            int a = 25.0 * (am[i]->getAmplitud(0) + am[i]->getAmplitud(1));
            if (a > 50) a = 50;
            cout << setw(6) << frecuencia[i] << " Hz  (";
            for (int j = 0; j < a; j++) cout << "-";
            cout << ")" << endl;
        }

    } while (c != 27);

    Pa_CloseStream(stream);
    Pa_Terminate();
    for (int i = 0; i < NB; i++) {
        delete am[i];
        delete filtro[i];
    }
    delete sink;
    delete input;
    return 0;
}
```

8.6 Evaluación y reporte de resultados

1.- Defina el concepto de amplitud de una señal y explique el porqué (para una señal real) se requiere considerar un segmento de la señal para poder estimar su amplitud.

2.- Considere un estimador de amplitud por bloque con un ancho de ventana $W = 256$ para un sistema de procesamiento de audio con frecuencia de muestreo de 44100 Hz. Aproximadamente a partir de qué frecuencia el estimador comenzará a perder sensibilidad?

3.- Modifique la clase `Amplimetro` para que sea posible también estimar la amplitud pico-a-pico.

8.7 Retos

Un *seguidor de envolvente* es un dispositivo que toma una señal como entrada y genera a la salida otra señal que representa las variaciones contínuas de la amplitud de la señal de entrada. Es posible implementar un seguidor de envolvente a partir de una UGen `Amplimetro` si se interpolan los valores de amplitud que se estiman bloque a bloque para obtener una señal discreta. La interpolación de los valores de amplitud puede realizarse de manera sencilla mediante interpolación lineal o un filtro pasa-bajas de primer orden. Implemente un seguidor de envolvente y grafique la envolvente de amplitud de la señal proveniente del micrófono o entrada auxiliar de la computadora.

8.8 Conclusiones

Redacte en el recuadro sus conclusiones acerca de los conceptos aprendidos, las actividades realizadas y los objetivos planteados en esta práctica.

Parte II

Prácticas de nivel intermedio

Introducción a las prácticas de nivel intermedio

Las prácticas de nivel intermedio tienen algunos cambios importantes con respecto a las prácticas de nivel básico. La primer diferencia radica en que solamente se incluyen implementaciones en la plataforma Processing con la librería Beads, la cual para el autor es la que da lugar a implementaciones mas simples y potentes. Sin embargo, con la experiencia adquirida durante las prácticas de nivel básico, el alumno no debería tener problemas para tomar la implementación en Beads y escribir su propia versión para Minim o C++. Para aquellas prácticas en las que sea necesario hacer consideraciones especiales para alguna plataforma, estas se discutirán a detalle en la sección de Procedimiento.

La segunda diferencia consiste en que las UGen que se desarrollarán serán monofónicas; es decir, constarán de un solo canal de salida. Si bien es trivial procesar múltiples canales en Beads, en la práctica es mas común utilizar bloques mono-canal mientras no sea estrictamente necesario trabajar en estéreo. La librería Beads permite conectar una UGen de un canal a la salida física estéreo, multiplexando simplemente la misma señal en ambos canales. Esta funcionalidad es fácil de incorporar en Minim o C++.

Finalmente, en estas prácticas incorporaremos el concepto de modulación, el cual es fundamental para la implementación de diversos métodos de síntesis y procesamiento de audio. Uno de los retos de la modulación radica en que algunos de los parámetros del algoritmo a implementar pueden variar a lo largo de un bloque, lo cual implica cómputos adicionales en cada muestra. Se buscará en todo momento un buen equilibrio entre sencillez de implementación y eficiencia de ejecución.

Contenido

1. Oscilador senoidal
2. Concepto de modulación y su implementación
3. Rampas y envolventes
4. Waveshaping
5. Tablas de ondas
6. Líneas de retardo
7. Análisis de Fourier
8. Estimación de frecuencia

Capítulo 9

Oscilador senoidal

Nombre del estudiante	Calificación

9.1 Relación con los programas de estudio

Materia	Unidad y tema
Procesamiento Digital de Señales (IE / IT / IB)	1.2.- Señales discretas básicas 4.2.- Teorema de muestreo de Nyquist
Procesamiento de Señales de Audio (IE)	1.2.- Señales sinusoidales y concepto de frecuencia 1.12.- Teorema de muestreo de Nyquist 3.1 - Moduladores típicos: envolventes y osciladores de baja frecuencia

9.2 Introducción

El oscilador senoidal es otro de los principales bloques que se utilizan para el procesamiento y síntesis de audio. No solamente se utiliza como generador de sonido sino también como señal moduladora en diversos procesos, muchos de los cuales se estudiarán en la sección de Nivel Avanzado. Por otra parte, es posible obtener otras formas de onda a partir de una onda senoidal mediante la aplicación de técnicas de deformación de ondas (waveshaping) y modulación, como las que se estudiarán mas adelante.

En una práctica anterior se propuso utilizar un filtro pasa-banda altamente resonante como generador de tonos senoidales, sin embargo, esta no es la mejor manera de implementar un oscilador. En realidad, implementar un oscilador

senoidal básico es relativamente simple, pero una implementación ingenua o descuidada puede ocasionar artefactos audibles en la señal.

La señal a generar está dada por

$$x[n] = \text{sen}(\omega n),$$

donde ω es la frecuencia del oscilador en radianes por muestra. Usualmente uno querrá especificar la frecuencia f del oscilador en Hertz, por lo que ω puede calcularse como $\omega = 2\pi f/f_s$ para una frecuencia de muestreo f_s.

9.2.1 Implementación ingenua de un oscilador sinusoidal

En una implementación ingenua uno podría verse tentado a utilizar directamente la ecuación anterior, conservando el valor de n (el tiempo), el cual se incrementaría en cada muestra. Por ejemplo, una implementación ingenua (con fines ilustrativos) bajo Processing y Beads podría verse como sigue:

```
public class Senoidal extends UGen {
  int tiempo;
  float frecuencia;

  public Senoidal(AudioContext context) {
    super(context, 1);
  }

  public void calculateBuffer() {
    float[] out = bufOut[0];
    float omega = 2 * PI * frecuencia / context.getSampleRate();
    for (int i = 0; i < bufferSize; i++) {
      out[i] = sin(omega * tiempo);
      tiempo++;
    }
  }
}
```

El problema con esta implementación radica en que un cambio en la frecuencia por lo general ocasiona una discontinuidad en la señal de salida. Esto se ejemplifica en la Figura 9.1 para una señal senoidal cuya frecuencia es 500 Hz durante las primeras 200 muestras, y a partir de la muestra 201 la frecuencia cambia a 600 Hz (la tasa de muestreo es 44100 Hz). Esto se debe a que la *fase* del oscilador (es decir, el argumento de la función seno), la cual está dada por ωn, puede cambiar abruptamente al variar la frecuencia ω. En el ejemplo de la Figura 9.1, la fase en la muestra 200 (justo antes de que cambie la frecuencia) es igual a 0.267 ciclos (la parte decimal de nf/f_s), mientras que en la muestra 201 la fase es 0.735 ciclos, casi medio ciclo de diferencia.

El ejemplo que se presenta es severo, pero aún las discontinuidades leves pueden llegar a ser audibles y en muchos casos dificultan la implementación de algunas técnicas mas avanzadas como la modulación en frecuencia.

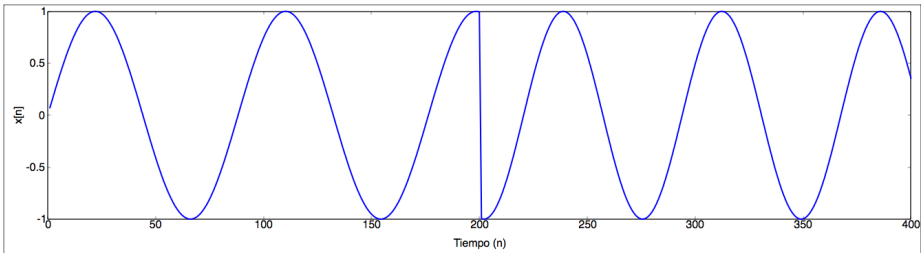


Figura 9.1: Ejemplo de discontinuidad que se presenta al cambiar la frecuencia de un oscilador sinusoidal "ingenuo". La frecuencia inicial es de 500 Hz y cambia a 600 Hz en la muestra 201.

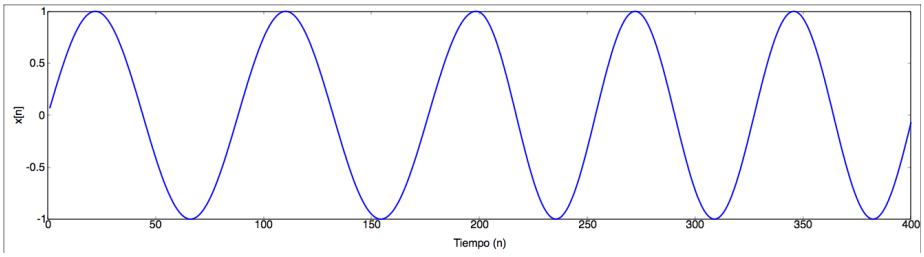


Figura 9.2: Ejemplo de señal generada por un oscilador sinusoidal con acumulación de fase. La frecuencia inicial es de 500 Hz y cambia a 600 Hz en la muestra 201. Note que no se presenta ninguna discontinuidad debido al cambio de frecuencia.

Existe otro problema con el enfoque anterior, relacionado con su implementación en un sistema digital. La variable `tiempo`, que se incrementa en cada muestra, podría desbordarse dependiendo del tipo de datos de la variable, ocasionando nuevamente un cambio abrupto en la fase, con los artefactos audibles que esto conlleva. En el caso de Processing o C++, donde el tipo `int` suele ser de 32 bits, el desbordamiento ocurriría aproximadamente cada 27 horas, lo cual no afectaría a la mayoría de las aplicaciones. Pero en plataformas donde las variables enteras son de 16 bits (e.g., algunas versiones de Arduino), el desbordamiento ocurriría aproximadamente cada 1.5 segundos.

9.2.2 Implementación mediante acumulación de fase

Para evitar cambios abruptos en la fase (y discontinuidades en la señal), es necesario conservar y actualizar la fase en lugar del tiempo. Dado que la fase es ωn, entonces en cada muestra, la fase se incrementa por ω cuando n se incrementa en 1. De esta manera, es fácil ir conservando un acumulado de la fase ϕ en lugar del tiempo; simplemente hay que hacer $\phi \longleftarrow \phi + \omega$ en cada muestra.

En esta implementación también se corre el riesgo de que se desborde la

variable en la que se acumula la fase, como se mencionó al final de la sección anterior. Sin embargo, en este caso el problema es muy fácil de resolver. Dado que sabemos que la función seno es periódica con periodo 2π, lo único que hay que hacer es tomar la fase módulo 2π después de un cierto tiempo (antes que de ocurra el desbordamiento), por ejemplo, después de procesar cada bloque de la señal. Esta acción además ayudará a mantener la precisión numérica de los cálculos dado que la fase debe almacenarse en una variable de punto flotante, las cuales son susceptibles a la acumulación de errores numéricos.

El código en Processing podría entonces verse como sigue:

```
public class Senoidal extends UGen {
  float fase;
  float frecuencia;

  public Senoidal(AudioContext context) {
    super(context, 1);
  }

  public void calculateBuffer() {
    float[] out = bufOut[0];
    float omega = 2 * PI * frecuencia / context.getSampleRate();
    for (int i = 0; i < bufferSize; i++) {
      out[i] = sin(fase);
      fase += omega;
    }

    // aplica modulo 2*PI a la fase para evitar desbordamiento y mantener precision
    fase = fase % (2 * PI);
  }
}
```

El cambio en el código es sutil, pero el resultado es completamente distinto. La Figura 9.2 muestra la señal que se obtiene para el caso de ejemplo de la sección anterior, pero utilizando acumulación de fase, donde es claro que ya no se presenta una discontinuidad.

El código mostrado anteriormente es solamente de carácter ilustrativo. En la sección de Procedimiento se presentará el código completo del oscilador, incluyendo métodos para manipular la frecuencia y reiniciar la fase del oscilador, así como una sencilla aplicación de prueba.

9.3 Objetivos didácticos

- Implementar un oscilador senoidal con control preciso de frecuencia.
- Ilustrar las diferencias entre distintas formas de implementar un oscilador.
- Elaborar una aplicación de prueba emulando un instrumento musical.

9.4 Material

- Una computadora con alguna de las siguientes combinaciones de software instalado:
  - Processing y Beads
  - Processing y Minim
  - GNU C++ (e.g., MinGW) y PortAudio
- Bocinas o audífonos estéreo

9.5 Procedimiento

En esta práctica implementaremos el oscilador sinusoidal como una UGen y lo utilizaremos para programar un instrumento musical virtual inspirado en el *theremin*. El theremin produce una oscilación audible cuya frecuencia se controla mediante un movimiento horizontal y su amplitud mediante un movimiento vertical. En un theremin real, los movimientos se realizan en el aire con las manos; en nuestra implementación utilizaremos el mouse para ambos movimientos. En el Apéndice D se presenta una breve historia y descripción de este instrumento.

El primer paso consiste en implementar la UGen `Senoidal` completa. Esta se basa en la implementación por acumulación de fase e incluye además métodos *set* y *get* para manipular la frecuencia y un método para reiniciar la fase del oscilador a cero (esto es de utilidad cuando se requiere sincronizar el oscilador con algún evento).

La clase completa queda como sigue:

```
public class Senoidal extends UGen {
  float fase;
  float frecuencia;

  public Senoidal(AudioContext context) {
    super(context, 1);
  }

  public Senoidal(AudioContext context, float f) {
    super(context, 1);
    frecuencia = f;
  }

  float frecuencia() { return frecuencia; }

  void setFrecuencia(float f) { frecuencia = f; }

  void reinicia() { fase = 0; }
```

```
  public void calculateBuffer() {
    float[] out = bufOut[0];
    float omega = 2.0 * PI * frecuencia / context.getSampleRate();
    for (int i = 0; i < bufferSize; i++) {
      out[i] = sin(fase);
      fase += omega;
    }

    // aplica modulo 2*PI a la fase para evitar desbordamiento y mantener precision
    fase = fase % (2 * PI);
  }
}
```

La aplicación principal será muy simple. Primero crearemos, en la función `setup()`, una cadena de síntesis formada por un oscilador, que pasa por un amplificador, y finalmente llega a la salida física de audio. La ganancia del amplificador estará determinada por la posición Y del mouse (incrementando la ganancia desde abajo hacia arriba), mientras que la frecuencia del oscilador estará determinada por la posición X del mouse. La relación entre la posición X del mouse y la frecuencia será exponencial, de manera que la posición del mouse corresponda linealmente con notas musicales. El rango de frecuencias cubrirá un cierto número de octavas (incialmente cuatro octavas, que es el rango típico de un theremin).

Una distancia de una octava corresponde a un factor de 2 entre frecuencias. Por lo tanto, si se toma como frecuencia de referencia f_0, la frecuencia f del oscilador estará dada por

$$f = f_0 \cdot 2^{n_i + rx/W},$$

donde n_i es el número de octava inicial (la que corresponde a la nota mas grave que se puede reproducir), r es el rango del instrumento en octavas, x es la posición X del mouse y W es el ancho de la ventana de la aplicación. Como nota de referencia tomaremos $f_0 = 55$ Hz, que corresponde a una nota La tres octavas por debajo del La central en un piano.

Además, se implementará la función `draw()` de Processing para visualizar la forma de onda y algunos parámetros en tiempo real. De esta manera, el programa queda como se muestra a continuación:

```
AudioContext ac;
Amplificador amp;
Senoidal osc;
int inicio = 1;  // octava inicial
int rango = 4;   // rango en octavas

void setup() {
  size(800, 600);
  ac = new AudioContext();
```

```
  amp = new Amplificador(ac, 1);
  osc = new Senoidal(ac, 440);
  amp.addInput(osc);
  ac.out.addInput(amp);
  amp.setGanancia(0.0);
  ac.start();
}

void mouseMoved() {
  float oct = inicio + (float)rango * mouseX / width;
  osc.setFrecuencia(55 * pow(2, oct));
  amp.setGanancia(1 - (float)mouseY / height);
}

void draw() {
  background(0);
  text("Rango = " + rango + " octavas", 10, 10);
  text("Octava inicial = " + inicio, 10, 20);
  text("Frecuencia = " + osc.frecuencia(), 10, 30);
  stroke(0, 255, 0);
  simplescopio(amp.getOutBuffer(0));
}
```

Finalmente, implementaremos algunos comandos del teclado para cambiar el rango del instrumento (mediante los caracteres numéricos) y la octava inicial (mediante las flechas izquierda y derecha). Esto permitirá ajustar el rango de frecuencias del instrumento a cualquier rango deseado, incluso por debajo o por arriba del rango audible. También se podrá utilizar la tecla 'R' para reiniciar la fase del oscilador (lo cual, por lo general, ocasionará un cambio abrupto en la fase y una discontinuidad en la señal de salida). Para agregar esta funcionalidad, implementaremos la función `keyPressed()` como sigue:

```
void keyPressed() {
  if (key >= '1' && key <= '9') rango = key - '0';
  if (keyCode == LEFT) inicio--;
  if (keyCode == RIGHT) inicio++;
  if (key == 'R' || key == 'r') osc.reinicia();
}
```

9.6 Evaluación y reporte de resultados

1.- Después de implementar y probar el programa de ejemplo, comente o elimine la línea donde se aplica el módulo 2π a la fase acumulada. Use nuevamente el programa por algunos minutos, con rangos de 1 o 2 octavas. Nota alguna diferencia? En caso de ser así, describa qué es lo que ocurre. Qué ocurre cuando reinicia la fase (usando la tecla 'R')?

2.- Mediante la implementación realizada en esta práctica, explore y describa qué ocurre cuando el oscilador se entona a frecuencias muy altas (superiores a la frecuencia de Nyquist). Puede entonar el oscilador a una frecuencia superior a Nyquist que sea audible? Explique su respuesta.

3.- Modifique la UGen `Senoidal` para implementar el oscilador de manera ingenua, agregando un contador de tiempo (una variable entera llamada `tiempo`) y cambiando `sin(fase)` por `sin(omega * tiempo)`. Pruebe el nuevo programa y escuche con cuidado. Describa las diferencias audibles entre ambas implementaciones. Simule el problema de desbordamiento de la variable `tiempo` restringiéndola a 16 bits (cuando la variable llegue a 2^{16}, reiníciela a cero) y describa lo que ocurre.

4.- Anteriormente se discutió la posibilidad de implementar un oscilador sinusoidal a partir de un filtro pasa-banda altamente resonante sin compensación. Qué tipo de artefactos no deseados podría producir este oscilador al variar su frecuencia?

9.7 Retos

La ejecución de un instrumento como el theremin requiere de mucha práctica, precisión en los movimientos y desarrollo del oido musical. Una manera de facilitar la ejecución consiste en *cuantizar* el espacio de frecuencias para que solamente puedan reproducirse frecuencias que correspondan a tonos musicales; por ejemplo, aquellas frecuencias de la forma $f_0 \cdot 2^{n/12}$ para n entero. En nuestra implementación, esto se logra simplemente asegurándose de cuantizar la variable `oct` en la función `mouseMoved()` al múltiplo de 1/12 mas cercano. Agregue además un comando del teclado en la función `keyPressed()` para activar o desactivar la cuantización. Y como reto final, dibuje en la pantalla un teclado de piano para tener una referencia visual de la nota que corresponde a cada posición (recuerde que en el extremo izquierdo siempre se encuentra una nota La).

9.8 Conclusiones

Redacte en el recuadro sus conclusiones acerca de los conceptos aprendidos, las actividades realizadas y los objetivos planteados en esta práctica.

Capítulo 10

Concepto de modulación y su implementación

Nombre del estudiante	Calificación

10.1 Relación con los programas de estudio

Materia	Unidad y tema
Procesamiento Digital de Señales (IE / IT / IB)	1.3.- Sistemas discretos y sus características 1.4.- Sistemas lineales e invariantes en el tiempo
Procesamiento de Señales de Audio (IE)	3.1.- Concepto de modulación 3.2.- Moduladores típicos: envolventes y osciladores de baja frecuencia 3.6.- Modulación de amplitud

10.2 Introducción

La mayoría de los componentes básicos a partir de los cuales se construyen los sistemas de generación y procesamiento de audio dependen de ciertos parámetros ajustables. Por ejemplo, la ganancia de un amplificador, la frecuencia de un oscilador, o la frecuencia de corte de un filtro pasa-bajas. En una aplicación específica, algunos de estos parámetros se fijan permanentemente, mientras que otros pueden variar durante la ejecución de la aplicación. La manera en que hemos implementado las distintas UGen, hasta ahora, permite variar los parámetros de la UGen fuera de la función callback donde se implementa el

proceso. Los parámetros pueden variar de un bloque de procesamiento al siguiente; sin embargo, en muchas ocasiones es necesario producir variaciones tan rápidas o tan suaves que requieran actualizar los parámetros para cada muestra. A la acción de variar algún parámetro de un sistema de manera continua se le conoce como *modulación*; en el caso de señales digitales, la modulación de un parámetro de realiza a nivel de muestra, o incluso a nivel sub-muestra (utilizando técnicas de sub-muestreo).

Si bien existen casos donde algunos parámetros pueden ser modulados de manera directa por el usuario, en la mayoría de los casos, las modulaciones se aplican de manera hasta cierto punto automatizada, donde el valor del parámetro a modular está definido por una función, o mas explícitamente, por una señal. Esta señal, a la que se le llama *señal moduladora*, es simplemente otra señal de entrada de la componente o la UGen cuyo parámetro se desea modular.

Tomemos como ejemplo el caso de un amplificador. De acuerdo a lo estudiado en la Práctica 3, la ecuación mediante la cual implementamos un amplificador es

$$y[n] = g \cdot x[n],$$

donde $x[n]$ es la señal de entrada, $y[n]$ es la señal de salida y g es el factor de ganancia. Ahora bien, para poder modular la ganancia del amplificador, es necesario considerar variaciones continuas (muestra a muestra) en el valor de la ganancia con respecto al valor fijo g. Supongamos que estas variaciones están dadas por una señal $z[n]$, entonces la salida del amplificador puede obtenerse como

$$y[n] = (g + z[n]) \cdot x[n]. \tag{10.1}$$

Para mayor versatilidad, supongamos que además deseamos controlar, de manera general, la amplitud de las modulaciones dadas por $z[n]$. Esto se puede lograr incorporando un factor adicional g_z de la manera siguiente:

$$y[n] = (g + g_z z[n]) \cdot x[n]. \tag{10.2}$$

Tanto $x[n]$ como $z[n]$ son señales de entrada al amplificador, siendo $x[n]$ la señal *portadora* y $z[n]$ la señal *moduladora*. Además, el amplificador ahora cuenta con dos parámetros: la *ganancia base* g y la profundidad o índice de modulación g_z. Simplemente variando estos dos parámetros es posible obtener resultados muy distintos para las mismas señales de entrada.

Note que el sistema resultante ya no puede considerarse como un sistema lineal e invariante en el tiempo. Si se considera a $x[n]$ como la entrada del sistema, entonces un retardo aplicado a $x[n]$ no afectaría a $z[n]$, y la señal de salida no sería una versión retardada de la salida original $y[n]$; y si se considera como entrada la señal multicanal $(x[n], z[n])$, de manera que un retardo afecte por igual a ambas señales de entrada, entonces el sistema no sería lineal ya que aparece el producto $x[n]z[n]$.

10.2.1 Sobre el espectro de la señal modulada

Consideremos ahora lo que sucede con la salida de un amplificador de ganancia variable (también llamado *VCA* - Voltage Controlled Amplifier) desde el punto de vista espectral. Para simplificar el análisis, consideremos que tanto la señal de entrada como la moduladora son sinusoidales; por ejemplo $x[n] = \cos(\omega_x n)$ y $z[n] = \cos(\omega_z n)$.

Estudiemos primero lo que ocurre al multiplicar ambas señales. Aplicando identidades trigonométricas se llega a que

$$x[n]z[n] = \cos(\omega_x n)\cos(\omega_z n) = \frac{\cos((\omega_x + \omega_z)n) + \cos((\omega_x - \omega_z)n)}{2}.$$

En otras palabras, el producto de dos sinusoidales da como resultado la superposición de dos nuevas sinusoidales (atenuadas), cuyas frecuencias son la suma y la diferencia de las frecuencias originales. A estas nuevas frecuencias se les llama *heterodinas*, y su generación es la base de diversos dispositivos, incluyendo el theremin (Apéndice D). A este fenómeno también se le conoce como *modulación en anillo* y se utiliza comúnmente en síntesis de audio para generar componentes de frecuencia inarmónicas. El mismo principio, pero de manera inversa, se utiliza también por algunos músicos para afinar sus instrumentos: se tocan simultáneamente la cuerda que desean afinar y una nota de referencia (por ejemplo, de un diapasón o de otra cuerda ya afinada); si se escuchan variaciones en la intensidad del sonido combinado (es decir, una modulación en amplitud) significa que la cuerda está desafinada (caso $\omega_z > 0$), y la frecuencia ω_z de las oscilaciones en amplitud indica qué tan desafinada está la cuerda.

Sustituyendo la ecuación anterior en la Ecuación 10.2, podemos ver que la salida del VCA cuando las entradas son sinusoidales es:

$$y[n] = g\cos(\omega_x n) + \frac{g_z}{2}\left[\cos((\omega_x + \omega_z)n) + \cos((\omega_x - \omega_z)n)\right].$$

Es decir que a la señal portadora simplemente se suman las componentes heterodinas. Por supuesto, manipulando los valores de la ganancia base g y el índice de modulación g_z, uno puede controlar el balance entre la señal original $x[n]$ y las componentes que se añaden al espectro.

Finalmente, en caso de que las señales de entrada no sean sinusoidales, entonces se considera su descomposición de Fourier como sumas de sinusoidales. Es fácil ver que el producto de ellas dará como resultado la superposición de sinusoidales cuyas frecuencias son todas las posibles sumas y diferencias entre las componentes de la señal portadora y las de la señal moduladora, dando lugar a espectros que pueden ser muy complejos y con un gran número de componentes inarmónicas.

10.2.2 Detalles sobre la implementación

El primer reto en la implementación de una UGen con modulación consiste en dotar a la UGen de múltiples entradas de audio. Esto es distinto a contar con

una sola entrada de audio multicanal, ya que en este caso las señales portadoras y moduladoras por lo general provienen de diferentes UGens. Para lograr esto, cada librería cuenta con sus propios mecanismos.

Sin importar la plataforma, es importante considerar la posibilidad de que no todas las señales de entrada de una UGen hayan sido asignadas. Por ejemplo, en el caso de un amplificador con modulación de ganancia, sería deseable que el amplificador actúe como un amplificador con ganancia fija si no se ha asignado una UGen a la entrada de modulación. Por lo general, esto requiere considerar distintos casos (mediante sentencias `if...else`) donde en algunos de ellos se asume que las señales moduladoras valen siempre cero (cuando no han sido asignadas).

En el caso de Beads, el número de canales de entrada (y de salida) se define en el constructor de la clase `UGen`. Luego se utiliza el método `addInput()` para asociar canales de salida de una UGen a los canales de entrada de otra UGen. Este método cuenta con dos formas: la primera recibe como argumento una UGen y asigna canales de la UGen fuente (la que se pasa como argumento) a la UGen destino (desde la que se llama al método) de manera consecutiva y cíclica. Es decir, el canal 0 de salida de la UGen fuente se asigna al canal 0 de entrada de la UGen destino, y así consecutivamente hasta haber asignado todos los canales de la UGen destino. Si la UGen destino tiene mas entradas que el número de salidas de la UGen fuente, entonces comienzan a asignarse nuevamente los primeros canales de la UGen fuente a los canales restantes de la UGen destino. Este es precisamente el mecanismo que permite rutear una UGen monocanal hacia cualquier UGen multicanal. La segunda forma de `addInput()` asigna solamente uno de los canales, y recibe como argumentos el número de canal de entrada a asignar, la UGen fuente y el número de canal de salida de la UGen fuente que se conectará a la entrada asignada. Esta forma es la que nos permitirá asignar salidas de distintas UGens a las entradas de una UGen con modulación. Sin embargo, es responsabilidad del programador recordar el papel de cada una de las entradas de una UGen, ya que éstas solo se distinguen mediante un subíndice. Para evitar confusiones, es recomendable encapsular las llamadas a `addInput()` dentro de métodos *set* cuyos nombres hagan alusión al papel que juega cada uno de los canales de entrada. Si no se conecta ninguna UGen a alguna de las entradas de audio, Beads automáticamente llena con ceros el buffer correspondiente a esa entrada, por lo que no es necesario realizar comprobaciones o consideraciones adicionales (como debe hacerse para la implementación en C++ que se muestra mas abajo). El siguiente código ejemplifica lo anterior:

```
public class VCA extends UGen {
  float ganancia;
  float indice_modulacion;

  public VCA(AudioContext context) {
   // crea una UGen con 2 entradas (portadora y moduladora) y 1 salida
    super(context, 2, 1);
  }
```

```
  // Asigna una UGen a la entrada de audio del amplificador (portadora)
  void setEntrada(UGen ugen) { addInput(0, ugen, 0); }

  // Asigna una UGen a la entrada de la señal moduladora
  void setModulador(UGen ugen) { addInput(1, ugen, 0); }

  public void calculateBuffer() {
    float[] in = bufIn[0];     // Obtiene el buffer de la señal de entrada (portadora)
    float[] mod = bufIn[1];    // Obtiene el buffer de la señal moduladora
    float[] out = bufOut[0];   // Obtiene el buffer de la señal de salida

    for (int i = 0; i < bufferSize; i++) {
      out[i] = (ganancia + indice_modulacion * mod[i]) * in[i];
    }
  }
}
```

Minim, por el contrario, requiere que cada una de las señales de entrada de una UGen sea declarada como un miembro de clase `UGenInput` e inicializada en el constructor de la UGen a la que pertenece. Al inicializar una `UGenInput` se debe especificar en el constructor el tipo de entrada, siendo los tipos `AUDIO` o `CONTROL`. Las entradas de tipo `AUDIO` tendrán siempre el mismo número de canales que la UGen anfitrión, mientras que las entradas de tipo `CONTROL` solo pueden tener un canal. Por lo general, son estas últimas las mas adecuadas para señales moduladoras.

La conexión de la salida de una UGen a una entrada de otra UGen se realiza mediante el método `patch()`. El método se llama desde la UGen fuente y toma como argumento el objeto destino de clase `UGen` o `UGenInput`. Esto se muestra en el siguiente código:

```
public class VCA extends UGen {
  float ganancia;
  float indice_modulacion;
  UGenInput audio_in;
  UGenInput mod_in;

  VCA() {
    super();
    audio_in = new UGenInput(UGen.InputType.AUDIO);
    mod_in = new UGenInput(UGen.InputType.CONTROL);
  }

  protected void uGenerate(float[] channels) {
    int cc = channelCount();
    float[] in = audio_in.getLastValues();
    float[] mod = mod_in.getLastValues();
```

```
    for (int c = 0; c < cc; c++) {
      channels[c] = (ganancia + indice_modulacion * mod[0]) * in[c];
    }
  }
}


void setup() {
  size(800, 600);

  Minim minim = new Minim(this);
  AudioOutput out = minim.getLineOut();

  // Inicializa las UGen involucradas
  Senoidal osc = new Senoidal(440);
  Senoidal mod = new Senoidal(4);
  VCA vca = new VCA();

  // Realiza las conexiones entre UGens
  osc.patch(vca);
  mod.patch(vca.mod_in);
  vca.patch(out);
}
```

Note que la llamada a `getLastValues()` para un objeto `UGenInput` de tipo `CONTROL` devuelve un arreglo con un solo elemento, ya que el objeto cuenta con un solo canal de salida.

Finalmente, la implementación en C++ basada en la clase `UGen` presentada en el Apéndice B es similar al caso de Beads. El número de entradas y salidas de la UGen se define en el constructor, y cada entrada consiste en un apuntador a alguna UGen, de la cual puede obtenerse su señal de salida mediante el método `getBuffer()`. La asignación de una entrada se realiza llamando al método `setInput()` de la UGen destino, a la cual se le pasan como argumentos el subíndice de la entrada a asignar y un apuntador a la UGen fuente. Al igual que en Beads, es recomendable definir métodos *set* que encapsulen las llamadas a `setInput()` de manera que no sea necesario recordar el papel asignado a cada canal. A continuación se muestra un bosquejo de cómo podría programarse un amplificador con modulación en C++:

```
class VCA : public UGen {
protected:
    float ganancia;
    float indice_modulacion;

public:
    VCA() : UGen(1, 2) {
        ganancia = 0;
```

```
        indice_modulacion = 0;
    }

    // Asigna una UGen a la entrada de audio del amplificador
    void setEntrada(UGen *ugen) { setInput(0, ugen); }

    // Asigna una UGen a la entrada de la señal moduladora
    void setModulador(UGen *ugen) { setInput(1, ugen); }

    void processBuffer() {
        UGen *ugen_in = getInput(0);
        UGen *ugen_mod = getInput(1);
        if (ugen_in == NULL) return;

        float *in = ugen_in->getBuffer();
        float *out = getBuffer();
        int n = getBufferSize();

        if (ugen_mod != NULL) {
            // caso con modulacion
            float *mod = ugen_mod->getBuffer();
            int j = 0, k = 0;
            int io = ugen_in->getNumOutputs();
            int mo = ugen_mod->getNumOutputs();
            for (int i = 0; i < n; i++) {
                out[i] = (ganancia * indice_modulacion * mod[k]) * in[j];
                j += io;
                k += mo;
            }
        }
        else {
            int io = ugen_in->getNumOutputs();
            for (int i = 0, j = 0; i < n; i++) {
                out[i] = in[j] * ganancia;
                j += io;
            }
        }
    }
};
```

En este caso, se hace la consideración de procesar solamente el primer canal de las señales portadora y moduladora, en caso de que alguna de ellas sea multi-canal. Esto se logra incrementando los subíndices de estas señales en `io` y `mo`, respectivamente, donde `io` es el número de canales de salida de la señal portadora y `mo` el de la señal moduladora.

10.3 Objetivos didácticos

- Entender el concepto de modulación cuando se aplica a sistemas de procesamiento de audio digital
- Comprender las cuestiones técnicas asociadas con la implementación de modulación.
- Implementar algunos de los bloques básicos de generación y procesamiento de audio con modulación: oscilador, filtro y amplificador.

10.4 Material

- Una computadora con alguna de las siguientes combinaciones de software instalado:
  - Processing y Beads
  - Processing y Minim
  - GNU C++ (e.g., MinGW) y PortAudio
- Bocinas o audífonos estéreo

10.5 Procedimiento

En esta práctica se implementará un amplificador con modulación de ganancia como una nueva UGen llamada `VCA`. El nombre *VCA* corresponde a las siglas de *Voltage Controlled Amplifier*, que es el nombre que se les dió a estos dispositivos en el campo del procesamiento de audio analógico [16], y por razones históricas se ha conservado. Los VCAs se utilizan ampliamente en el diseño y fabricación de consolas mezcladoras, procesadores dinámicos (compresores, compuertas de ruido, etc), y son pieza clave en la síntesis y diseño de sonidos.

El primer paso consiste en implementar la UGen `VCA`. Básicamente se tomará la implementación propuesta en la Introducción, agregando métodos `set` y `get` para manipular los parámetros de ganancia e índice de modulación. Estos métodos son los siguientes:

```
float ganancia() { return ganancia; }

void setGanancia(float g) { ganancia = g; }

float indice() { return indice_modulacion; }

void setIndice(float d) { indice_modulacion = d; }
```

Para probar el VCA, modificaremos el programa de ejemplo elaborado en la práctica anterior, reemplazando el amplificador de ganancia fija por un VCA

e incorporando un segundo oscilador senoidal como señal moduladora con una frecuencia fija de 4 Hz. Esto da lugar a un efecto conocido como *trémolo*. Para apreciar el efecto de la modulación, se controlará el índice de modulación mediante las teclas numéricas, que en la práctica anterior se utilizaban para modificar el rango del instrumento. En esta práctica, el rango se fijará a cuatro octavas.

```
AudioContext ac;
VCA vca;
Senoidal osc;
Senoidal mod;

int inicio = 1;  // octava inicial
int rango = 4;   // rango en octavas

void setup() {
  size(800, 600);

  ac = new AudioContext();
  vca = new VCA(ac);
  osc = new Senoidal(ac, 440);
  mod = new Senoidal(ac, 4);
  vca.setEntrada(osc);
  vca.setModulador(mod);
  ac.out.addInput(vca);
  ac.start();
}

void mouseMoved() {
  float oct = inicio + (float)rango * mouseX / width;
  osc.setFrecuencia(55 * pow(2, oct));
  vca.setGanancia(1 - (float)mouseY / height);
}

void keyPressed() {
  if (key >= '0' && key <= '9') vca.setIndice(vca.ganancia() * (key - '0') / 9);
  if (keyCode == LEFT) inicio--;
  if (keyCode == RIGHT) inicio++;
}

void draw() {
  background(0);
  text("Octava inicial = " + inicio, 10, 20);
  text("Frecuencia = " + osc.frecuencia(), 10, 30);
  text("Frecuencia modulacion = " + mod.frecuencia(), 10, 50);
  text("Indice modulacion = " + vca.indice(), 10, 60);
```

```
  stroke(0, 255, 0);
  simplescopio(vca.getOutBuffer(0));
}
```

Si realizó el reto de la práctica anterior, se sugiere incorporar a esta práctica la cuantización del espacio de frecuencias y dibujar un teclado de piano en la pantalla para facilitar la ejecución del instrumento.

10.6 Evaluación y reporte de resultados

1.- Explique en qué consiste la modulación y cuáles son las consideraciones técnicas que deben tomarse en cuenta para implementarla en la plataforma de su elección.

2.- Cuando la ganancia base del amplificador es cercana a uno (i.e., cuando el mouse se encuentra en la parte superior de la ventana) y el índice de modulación se fija también a uno (i.e., presionando la tecla '9'), el sonido que se escucha se vuelve mas brillante en las crestas de la señal moduladora. Puede explicar porqué ocurre esto?

3.- Una manera de medir el costo computacional de una componente de procesamiento de señales en tiempo real es contando el número de operaciones que se realizan por muestra dentro de la función callback. Distintas operaciones tienen un costo computacional diferente, por lo que las operaciones suelen contabilizarse por categorías, donde las categorías mas comunes son: sumas/restas, multiplicaciones/divisiones, asignaciones, llamadas a funciones (e.g., trigonométricas, etc), y toma de decisiones. Contabilice el número de operaciones por muestra para los siguientes procesos:

1. Amplificador con ganancia fija
2. Amplificador con modulación de ganancia (VCA)
3. Sintetizador implementado en la práctica anterior (oscilador senoidal conectado a un amplificador de ganancia fija)
4. Sintetizador implementado en esta práctica (oscilador senoidal conectado a un VCA cuya ganancia es modulada por otro oscilador senoidal)

4.- La señal moduladora aplicada en esta práctica es una senoidal a 4 Hz, lo que significa que la modulación provoca que la intensidad del sonido suba y baje cuatro veces por segundo. Pruebe qué ocurre cuando la frecuencia de modulación está en el rango de frecuencias audibles (por ejemplo, 100 Hz, 1000 Hz, etc) y describa el resultado tanto visual como audible. Qué ocurre cuando la frecuencia de modulación es proporcional a la frecuencia portadora?

10.7 Retos

El VCA implementado en esta práctica tiene una curva de control lineal; es decir, el factor de ganancia varía linealmente con respecto a la magnitud de la señal moduladora. En muchas aplicaciones es preferible usar un VCA con control exponencial, donde la ganancia expresada en decibeles varía de manera lineal con respecto a la señal de modulación. De igual manera, la ganancia base (a la cual se le suma la modulación) se expresa en decibeles. Implemente una nueva UGen VCA con control exponencial.

10.8 Conclusiones

Redacte en el recuadro sus conclusiones acerca de los conceptos aprendidos, las actividades realizadas y los objetivos planteados en esta práctica.

Capítulo 11

Rampas y envolventes

Nombre del estudiante	Calificación

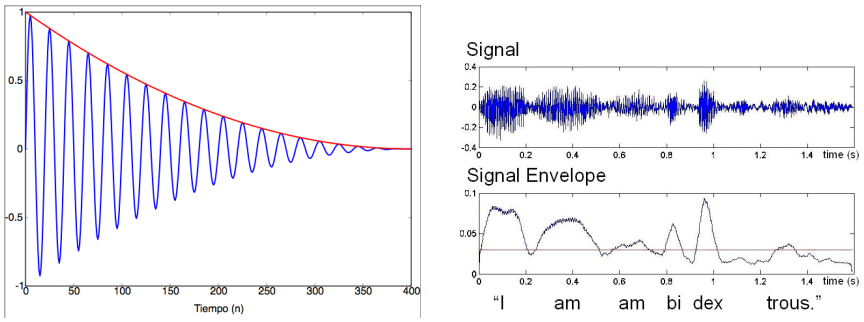
11.1 Relación con los programas de estudio

Materia	Unidad y tema
Procesamiento Digital de Señales (IE / IT / IB)	1.2.- Señales discretas básicas
Procesamiento de Señales de Audio (IE)	3.2.- Moduladores típicos: envolventes y osciladores de baja frecuencia

11.2 Introducción

En el contexto de señales, una envolvente o contorno es una función que describe la manera en que alguna propiedad de la señal cambia a lo largo del tiempo. Uno de los ejemplos mas claros es la envolvente de amplitud de una señal, la cual en casos simples puede verse como una función suave que pasa por las crestas de la señal en cuestión, como se muestra en la Figura 11.1a. Este concepto se introdujo brevemente al final de la Práctica 8, donde se pone como reto la implementación de un seguidor de envolvente mediante la interpolación de los valores de amplitud estimados en cada bloque de procesamiento. En la Figura 11.1b se muestra la envolvente de amplitud de un fragmento de voz hablada; como puede verse, la envolvente permite distinguir el inicio y final de cada fonema, por lo cual su estimación es uno de los primeros pasos para el procesamiento del habla [17].

La otra cara de la moneda consiste en generar, de manera artificial, una señal cuya envolvente de amplitud esté dada por una función o señal arbitraria. Para



(a) (b)

Figura 11.1: (a) Envolvente de amplitud (curva roja) de una señal sinusoidal (curva azul) cuya amplitud varía a lo largo del tiempo. (b) Envolvente de amplitud de una señal de voz hablada [17].

esto, se parte de una señal $x[n]$ cuya amplitud sea aproximadamente unitaria a lo largo del tiempo, y se obtiene la señal modulada $y[n]$ simplemente como

$$y[n] = e[n] \cdot x[n],$$

donde $e[n]$ representa a la envolvente que se desea aplicar. Claramente, lo anterior se puede implementar mediante un VCA con ganancia base cero y un índice de modulación positivo (e.g., unitario). El problema consiste, entonces, en modelar la función moduladora $e[n]$. Como puede verse en la Figura 11.1b, las envolventes reales pueden ser muy complejas; sin embargo, en muchas aplicaciones es posible utilizar una función definida a trozos, donde cada segmento está dado por una función simple.

Por ejemplo, la envolvente (curva roja) que se muestra en la Figura 11.1a corresponde a una caída exponencial de la forma

$$e[n] = \begin{cases} 0, & \text{si } n < n_0, \\ \exp\left(-\frac{n-n_0}{\lambda}\right), & \text{si } n \geq n_0, \end{cases} \tag{11.1}$$

donde n_0 es el tiempo de inicio de la envolvente y λ es un parámetro que se puede ajustar para definir la duración de la envolvente. Uno puede definir la duración en términos del tiempo Δn (en muestras) que tarda la envolvente en caer a la mitad de su amplitud; es decir que $e^{-\Delta n/\lambda} = 1/2$, lo que nos lleva a que $\lambda = \Delta n / \log 2$. Por otra parte, dado que $e^{-n/\lambda} = e^{-(n-1)/\lambda} \cdot e^{-1/\lambda}$, entonces uno puede calcular la envolvente de manera eficiente mediante el siguiente sistema sin entradas:

$$e[n] = g \cdot e[n-1],$$

donde $g = e^{-(\log 2)/\Delta n}$. Para iniciar o “disparar” la envolvente, simplemente se hace $e[n_0] = 1$ en el tiempo n_0 en que se desea que ocurra un evento y se

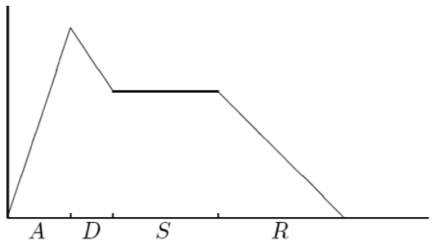


Figura 11.2: Modelo de envolvente ADSR.

calcula el factor g de acuerdo a la duración deseada. Este modelo de envolvente exponencial es útil para modelar la dinámica de sonidos percusivos a lo largo del tiempo.

Otro de los modelos de envolvente mas populares en la síntesis de sonido es la envolvente ADSR (Figura 11.2), cuyas siglas denotan las cuatro etapas de las que se compone: ataque (attack), decaimiento (decay), sostenimiento (sustain) y relajación (release). La forma de esta envolvente está definida por cuatro parámetros: la duración de las etapas de ataque, decaimiento y relajación, y el nivel de sostenimiento. La duración de la etapa de sostenimiento es variable y suele estar determinada por la duración del sonido que se desea reproducir. Pensando, por ejemplo, en un piano, el ataque de una nota inicia al pulsar la tecla correspondiente, y el sostenimiento termina (e inicia la relajación) cuando la tecla deja de pulsarse.

El uso de segmentos lineales simplifica mucho la implementación de una envolvente definida a trozos; sin embargo, existen diversos trucos a partir de los cuales pueden obtenerse envolventes no lineales a partir de envolventes lineales. Por ejemplo, uno puede definir cada segmento lineal en términos de una función lineal parametrizada $\phi(t)$ donde el parámetro t va de 0 a 1, y luego reemplazar $\phi(t)$ por $\phi(t^\alpha)$, donde $\alpha > 0$ determina la curvatura de la envolvente. Alternativamente, uno puede pasar la señal $e[n]$ por un filtro pasa-bajas para suavizar la forma de la envolvente.

Las envolventes definidas por trozos (incluyendo la ADSR) no se utilizan únicamente para controlar la amplitud de un sonido, sino también para modular cualquier otro parámetro, como la frecuencia de un oscilador o la frecuencia de corte de un filtro. Algunas de estas aplicaciones se estudiarán en las siguientes prácticas.

11.3 Objetivos didácticos

- Comprender el papel de las envolventes tanto en el análisis como en la síntesis de señales de audio.
- Implementar una envolvente lineal a trozos y modelar envolventes ADSR.

- Utilizar una envolvente para modular la amplitud de una señal de audio.

11.4 Material

- Una computadora con alguna de las siguientes combinaciones de software instalado:
  - Processing y Beads
  - Processing y Minim
  - GNU C++ (e.g., MinGW) y PortAudio
- Bocinas o audífonos estéreo

11.5 Procedimiento

En esta práctica se realizarán las siguientes tareas:

1. Implementación de una UGen que genere una envolvente lineal a trozos de manera general con transiciones esporádicas (aplicadas fuera de la función callback)
2. Implementación de cambios pre-programados en la envolvente con transiciones en cualquier momento (precisión a nivel muestra).
3. Implementación de una aplicación de prueba para emular una envolvente ADSR.

11.5.1 Envolvente lineal básica

Consideremos una UGen que da como salida un valor v, el cual permanecerá constante mientras no se le de ninguna instrucción a la UGen. En algún momento, se le pedirá a la UGen cambiar su valor de salida al valor objetivo v_o, pero no de manera inmediata y abrupta, sino gradualmente a lo largo de un intervalo de tiempo Δn (dado en muestras). Si se asume que el valor de salida cambiará linealmente, entonces la tasa de cambio es constante y está dada por $m = (v_o - v_i)/\Delta n$, donde v_i es el valor de salida al momento de recibir la instrucción. A partir de ese momento, el valor de salida puede actualizarse de la manera siguiente:

$$\begin{aligned} v &\longleftarrow v + m, \\ \Delta n &\longleftarrow \Delta n - 1, \end{aligned}$$

hasta que Δn sea igual a cero. Una vez que Δn llega a cero, la duración del segmento ha transcurrido completamente y el valor de salida nuevamente se mantiene constante. Esto da como resultado una señal lineal a trozos donde la transición de un valor a otro se tiene que realizar de manera explícita, por

ejemplo, llamando a un método de la UGen. Este diseño está inspirado en el objeto `[line~]` de PureData [10].

A continuación se presenta la primer versión de esta UGen, llamada `Rampa`, la cual tiene como miembros de datos el valor actual de la salida, la tasa de cambio y el tiempo en muestras que le resta al segmento en curso. En esta clase, el constructor simplemente llama al constructor de `UGen` para crear una UGen con un canal de salida. La función callback `calculateBuffer()` implementa el proceso descrito arriba para actualizar el valor de salida. Además, se tiene un método `cambia(float vo, float tms)` para instruír a la UGen a cambiar su valor de salida hacia el valor `vo` en un intervalo de `tms` milisegundos. Este método calcula primero la duración del segmento en muestras (variable `n`) y luego calcula la tasa de cambio. En caso de que se especifique una duración equivalente a cero muestras, se asume que el cambio debe ser inmediato, por lo que el valor de salida se actualiza en ese momento. El método `cambia()` se sobrecarga para admitir un solo argumento para el cambio inmediato del valor de salida.

```
public class Rampa extends UGen {
  float valor;    // valor actual
  float tasa;     // tasa de cambio
  int tiempo;     // tiempo restante en muestras

  public Rampa(AudioContext context) {
    super(context, 1);
  }

  float valor() { return valor; }

  Rampa cambia(float vo, float tms) {
    int n = round(tms * context.getSampleRate() / 1000.0);
    if (n > 0) {
      tasa = (vo - valor) / n;
      tiempo = n;
    } else {
      valor = vo;
      tasa = 0;
      tiempo = 0;
    }
    return this;
  }

  Rampa cambia(float vo) { return cambia(vo, 0); }

  public void calculateBuffer() {
    float[] out = bufOut[0];
    for (int i = 0; i < bufferSize; i++) {
```

```
      out[i] = valor;
      if (tiempo >= 0) {
        valor += tasa;
        tiempo--;
      }
    }
  }
}
```

Note que algunos de los métodos devuelven una referencia al objeto desde el cual se llama, permitiendo así el encadenamiento de llamadas en una misma línea de código. Por ejemplo, la siguiente línea ocasiona que el valor de salida cambie abruptamente a 1, seguido de una rampa descendente hacia cero, con una duración de un segundo.

```
rampa.cambia(1).cambia(0, 1000);
```

Finalmente, se incluyó un método *get* para obtener el valor actual de la envolvente, simplemente con fines de depuración y graficación de la envolvente.

11.5.2 Cambios programados

La UGen `Rampa`, en su primer versión, tiene el inconveniente de que cada transición debe inducirse explícitamente llamando al método `cambia()`. Sin embargo, existen ocasiones en las que es deseable definir una secuencia de segmentos para ejecutarse de manera preprogramada, sin tener que preocuparse de disparar el siguiente segmento una vez que transcurre el segmento en curso. Un claro ejemplo es la primer mitad de la envolvente ADSR, donde las etapas de ataque y decaimiento tienen duraciones predefinidas.

Una manera de mejorar la UGen `Rampa` consiste en dotarla de una lista de cambios pre-programados, para cada uno de los cuales se definirá el valor objetivo y el tiempo en el que se alcanzará dicho valor a partir del final de la etapa anterior. Para almacenar dichas listas, lo mas sencillo es utilizar la clase `FloatList` de Processing, o bien, la clase `queue<float>` de la librería estándar de plantillas de C++. Por lo tanto, agregaremos los siguientes miembros de datos a la clase `Rampa` :

```
  FloatList objetivo;
  FloatList tiempo_ms;
```

Será necesario inicializar los nuevos miembros en el constructor de la clase, agregándole las siguientes líneas:

```
    objetivo = new FloatList();
    tiempo_ms = new FloatList();
```

La idea es que cada vez que transcurre un segmento de la envolvente, la UGen verifique inmediatamente si existe algún cambio programado (es decir, si

la lista de cambios no está vacía). En caso de haberlo, se aplicará el cambio almacenado y se eliminará el primer elemento de la lista. Todo esto ocurre dentro del ciclo interno de la función callback, la cual ahora queda como sigue:

```
public void calculateBuffer() {
  float[] out = bufOut[0];
  for (int i = 0; i < bufferSize; i++) {
    out[i] = valor;
    if (tiempo >= 0) {
      valor += tasa;
      tiempo--;

      // procesar cambios programados
      if ((tiempo <= 0) && (objetivo.size() > 0)) {
        cambia(objetivo.get(0), tiempo_ms.get(0));
        objetivo.remove(0);
        tiempo_ms.remove(0);
      }
    }
  }
}
```

Note que ahora los cambios pueden ocurrir en cualquier momento, incluso dentro del bloque de procesamiento, si éstos han sido previamente programados. Para programar los cambios, se implementa el método `agrega()` de la clase `Rampa`, el cual agrega elementos nuevos al final de la lista de cambios:

```
Rampa agrega(float vo, float tms) {
  objetivo.append(vo);
  tiempo_ms.append(tms);
  return this;
}
```

Y finalmente, se agrega un método que permite eliminar todos los elementos de la lista, cancelando así cualquier cambio programado (pero sin alterar el segmento en curso):

```
Rampa cancela() {
  objetivo.clear();
  tiempo_ms.clear();
  return this;
}
```

Nuevamente, los métodos `agrega()` y `cancela()` devuelven una referencia a la UGen desde donde son llamados, permitiendo encadenar múltiples llamadas en una sola línea. Por ejemplo, para iniciar una envolvente ADSR, uno podría escribir:

```
rampa.cancela().cambia(0, 10).agrega(1, a).agrega(s, d);
```

donde las variables `a` y `d` representan los tiempos (en milisegundos) de ataque y decaimiento, y la variable `s` contiene el nivel de sostenimiento (entre cero y uno). Note que primero se reinicia la envolvente a cero (con un tiempo corto de 10 ms, para no producir cambios abruptos), mientras que las etapas de ataque y decaimiento quedan programadas para transcurrir una después de otra. La etapa de sostenimiento concluye cuando el ejecutante así lo decide, y en ese momento se puede ejecutar la etapa de relajación (con un tiempo de relajación `r`) mediante la instrucción `rampa.cambia(0, r);`

11.5.3 Aplicación de ejemplo

Nuevamente, la aplicación de ejemplo es una modificación de la aplicación elaborada en la práctica anterior. El sistema a implementar consistirá en un oscilador sinusoidal cuya amplitud será modulada (a través de una VCA) por una envolvente. La inicialización de la aplicación se muestra a continuación:

```
AudioContext ac;
VCA vca;
Senoidal osc;
Rampa env;

int inicio = 1;  // octava inicial
int rango = 4;   // rango en octavas
boolean cuantiza = true;

// parámetros de la envolvente ADSR
float A = 20;
float D = 500;
float S = 0.5;
float R = 500;

float[] envplot;
int envplotidx;

void setup() {
  size(800, 600);
  envplot = new float[300];
  envplotidx = 0;

  ac = new AudioContext();
  vca = new VCA(ac);
  osc = new Senoidal(ac, 440);
  env = new Rampa(ac);

  vca.setEntrada(osc);
```

```
  vca.setModulador(env);
  ac.out.addInput(vca);

  vca.setGanancia(0.0);
  vca.setIndice(1.0);
  ac.start();
}
```

Además de las UGens requeridas (oscilador, VCA y envolvente), se definen algunas propiedades del instrumento: octava inicial, rango de octavas, y una variable que permite activar o desactivar la cuantización de frecuencia a la escala cromática de 12 tonos. Posteriormente se definen los parámetros de la envolvente ADSR (tiempos de ataque, decaimiento y relajación en milisegundos, así como el nivel de sostenimiento). Finalmente, se declara un arreglo `envplot` que contendrá una versión submuestreada de la envolvente para fines de visualización (ver la implementación de la función `draw()` mas abajo).

Cabe resaltar que la ganancia base del VCA se mantendrá en cero, y el índice de modulación en uno. Esto significa que la ganancia del VCA estará totalmente controlada por la envolvente, de manera que no se escuche ningún sonido mientras la envolvente genere una salida nula. A diferencia de las prácticas anteriores, el usuario deberá ejecutar alguna acción (en este caso, presionar el botón del mouse) para escuchar algún sonido.

Se simulará una envolvente ADSR donde el ataque iniciará al presionar el botón del mouse, y la relajación iniciará al soltar el botón del mouse. Para esto, se implementarán las funciones callback `mousePressed()` y `mouseReleased()` de Processing. Además, mientras se escucha la nota, será posible cambiar su tono al arrastrar el mouse, lo cual puede implementarse mediante la función callback `mouseDragged()`. A continuación se muestra el código para estas funciones:

```
// Inicia el ataque y decaimiento de una nota al presionar el mouse
void mousePressed() {
  float oct = inicio + (float)rango * mouseX / width;
  float vel = 1 - (float)mouseY / height;
  if (cuantiza) oct = round(oct * 12) / 12.0;

  env.cancela().cambia(0, 10).agrega(vel, A).agrega(S * vel, D);
  osc.setFrecuencia(55 * pow(2, oct));
}

// Inicia la relajación al soltar el mouse
void mouseReleased() {
  env.cambia(0, 500);
}

void mouseDragged() {
  float oct = inicio + (float)rango * mouseX / width;
```

```
  if (cuantiza) oct = round(oct * 12) / 12.0;
  osc.setFrecuencia(55 * pow(2, oct));
}
```

Existen varios detalles en las funciones anteriores. El primero es la cuantización, la cual se implementa de manera muy simple: recordemos que la variable `oct` representa la altura de la nota en octavas a partir de una nota base, la cual en este caso es la nota La tres octavas por debajo del La central, cuya frecuencia es 55 Hz. Para cuantizar la frecuencia, simplemente hay que redondear `oct` al múltiplo de 1/12 mas cercano (ya que hay exactamente 12 notas por octava). El segundo detalle consiste en que la altura de la envolvente al concluir el ataque estará dado por la variable `vel` (por *velocidad*), la cual en este caso depende de la posición Y del mouse a la hora de presionar el botón. Esto permite controlar el volumen o intensidad de cada nota. Note que el nivel de sostenimiento se especifica en proporción a la velocidad (`S * vel`). Finalmente, en la función `mouseReleased()` se activa la etapa de relajación.

Para poder activar o desactivar la cuantización, implementamos un comando del teclado (tecla 'C') mediante la función `keyPressed()` de Processing. Si el alumno lo desea, puede agregar comandos para modificar el rango del instrumento y transportar a otras octavas, como se hizo en prácticas anteriores.

```
void keyPressed() {
  if (key == 'C' || key == 'c') cuantiza = !cuantiza;
}
```

Para concluir, se implementa la función `draw()` para mostrar la siguiente información en la ventana de la aplicación:

- Mostrar información sobre algunos parámetros del programa (cuantización, octava inicial y rango, frecuencia del oscilador, etc)
- Mostrar gráficamente la forma de onda generada por el instrumento (a la salida del VCA).
- Mostrar gráficamente la forma de la envolvente (submuestreada a partir de la UGen).
- Mostrar un bosquejo de un teclado tipo piano que proporcione una referencia de la ubicación de las notas.

La función queda como sigue:

```
void draw() {
  background(0);

  // Imprime parámetros básicos
  text("Frecuencia = " + osc.frecuencia(), 10, 30);
  text("Cuantizacion = " + (cuantiza ? "ON" : "OFF"), 10, 40);
  text("Ataque = " + A + " ms", 10, 50);
```

```
  text("Decaimiento = " + D + " ms", 10, 60);
  text("Sostenimiento = " + S, 10, 70);
  text("Relajacion = " + R + " ms", 10, 80);

  // Grafica la forma de onda
  stroke(0, 255, 0);
  simplescopio(vca.getOutBuffer(0));

  // Sub-muestrea y grafica la envolvente
  envplot[envplotidx++] = env.valor();
  if (envplotidx >= envplot.length) envplotidx = 0;
  pushMatrix();
  translate(0, 200);
  stroke(255, 0, 0);
  simplescopio(envplot);
  popMatrix();

  // Dibuja teclado
  int[] colortecla = { 1, 0, 1, 1, 0, 1, 0, 1, 1, 0, 1, 0 };
  String[] nombretecla = { "A", "", "B", "C", "", "D", "", "E", "F", "", "G", "" };
  int col, nota, notaant = -1;
  float offx = ((width / (rango * 12)) - textWidth("M")) * 0.75;
  for (int x = 0; x < width; x++) {
    nota = (int)round(12.0 * rango * x / width) % 12;
    col = colortecla[nota] * 191 + 64;
    if (nota == notaant) {
      stroke(col, 64);
    } else {
      fill(255, 128);
      text(nombretecla[nota], x + offx, height - 20);
      stroke(0, 64);
    }
    notaant = nota;
    line(x, 0, x, height);
  }
}
```

11.6 Evaluación y reporte de resultados

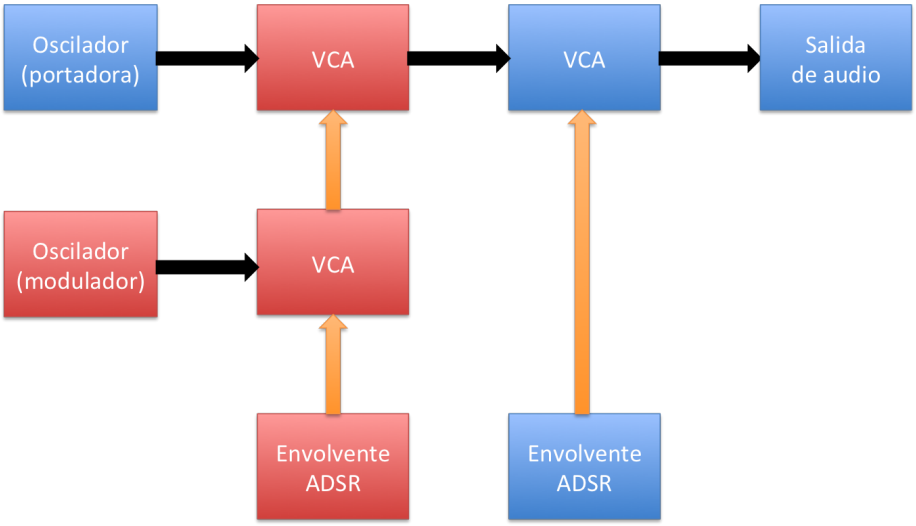


Figura 11.3: Arquitectura propuesta para el Ejercicio 2. Los bloques azules muestran los componentes utilizados en la práctica original, mientras que los rojos muestran los nuevos componentes. Las flechas anaranjadas muestran las conexiones correspondientes a señales moduladoras.

1.- Ajuste los parámetros de ataque, decaimiento, sostenimiento y relajación de la envolvente para modelar los siguientes sonidos:

a) Un sonido con un ataque largo, y que una vez transcurrido el ataque se mantenga a su máxima intensidad, pero se apague inmediatamente al soltar el botón del mouse.

b) Un sonido percusivo, de ataque rápido y corta duración, independientemente del tiempo que dure presionado el botón del mouse.

2.- Modifique la arquitectura del sistema implementado en esta práctica de acuerdo a la Figura 11.3, donde los bloques azules corresponden a las componentes ya existentes, y los bloques rojos son las componentes nuevas. Esta nueva arquitectura incorpora un segundo oscilador cuya amplitud es modulada por una segunda envolvente ADSR. Este segundo oscilador, modula a su vez la amplitud del primer oscilador, para producir efectos tipo *tremolo* (si la frecuencia de la señal moduladora es baja), o cambios en el timbre y la forma de onda del primer oscilador (si la frecuencia moduladora es alta), pero la intensidad de estos efectos es controlada por la segunda envolvente. Experimente con esta nueva arquitectura generando sonidos que le parezcan interesantes y registre los parámetros utilizados para el segundo oscilador y para ambas envolventes en una tabla.

11.7 Retos

Como se menciona en la Introducción, una manera de generar envolventes no lineales consiste en parametrizar la salida de cada segmento lineal de la envolvente mediante un parámetro $p \in [0, 1]$, lo cual da lugar a la ecuación $v = v_i + (v_o - v_i)p$, donde v es el valor de salida de la envolvente, v_i es el valor al inicio del segmento y v_o es el valor objetivo (el que se alcanzará al final del segmento). El siguiente paso consiste en hacer que p vaya de cero a uno a lo largo del segmento, lo cual se realiza fácilmente inicializando $p = 0$ al inicio del segmento, y posteriormente actualizando $p \longleftarrow p + m_p$ en cada muestra, con $m_p = 1/\Delta n$, donde Δn es la duración deseada del segmento.

El cálculo de m_p y la inicialización de p se realizarían en el método `cambia()` de la UGen, mientras que la actualización de p y el cálculo del valor de salida v se implementarían en el ciclo principal de la función callback.

Computacionalmente, esta alternativa es mucho menos eficiente, ya que se reemplaza una suma ($v \longleftarrow v + m$) por un producto y dos sumas ($v = v_i + \Delta v \cdot p$; $p \longleftarrow p + m_p$), por cada muestra de salida. Sin embargo, ahora es posible distorsionar la forma de la envolvente aplicando a p cualquier función $\phi : [0, 1] \longrightarrow [0, 1]$. Por ejemplo, uno puede ahora calcular v como $v = v_i + \Delta v \cdot p^\alpha$, con $\alpha > 0$ para obtener segmentos curvos.

Implemente y pruebe una nueva UGen llamada `RampaCurva` para generar envolventes no-lineales donde se pueda especificar el parámetro de curvatura α para cada segmento (incluso para los segmentos pre-programados).

11.8 Conclusiones

Redacte en el recuadro sus conclusiones acerca de los conceptos aprendidos, las actividades realizadas y los objetivos planteados en esta práctica.

Capítulo 12

Waveshaping

Nombre del estudiante	Calificación

12.1 Relación con los programas de estudio

Materia	Unidad y tema
Procesamiento Digital de Señales (IE / IT / IB)	1.3.- Sistemas discretos y sus características
Procesamiento de Señales de Audio (IE)	1.13.- Cuantización 3.5.- Waveshaping

12.2 Introducción

En la Práctica 3 se definieron las propiedades fundamentales de un sonido, desde un enfoque perceptual, las cuales son: **tono, intensidad, timbre y duración**. Así mismo, se mencionó que cada una de estas propiedades se relaciona con una característica física/matemática de la señal o la onda que representa el sonido. El tono está asociado con la frecuencia fundamental, la intensidad está asociada con la amplitud, el timbre está asociado con la forma de onda y con el espectro de frecuencia, y la duración con la percepción del inicio y final de un sonido. En las prácticas anteriores, hemos visto como producir sonidos con una frecuencia específica, cómo controlar la amplitud de un sonido, y cómo usar una envolvente para definir el inicio y fin de un sonido. Sin embargo, poco hemos dicho sobre el timbre, y esto se debe a que el timbre es considerablemente más complicado de analizar y modelar que las otras propiedades. De hecho, existe un gran número de técnicas para modelar y manipular el timbre de un sonido, algunas de las cuales estudiaremos a partir de esta práctica.

Armónico	Inarmónico	Ruido
$\sum_{k=1}^{\infty} a_k \cos(k\omega t + \phi_k)$	$\sum_{k=1}^{\infty} a_k \cos(\omega_k t + \phi_k)$	$\int_0^{\infty} a(\omega) \cos(\omega t + \phi(\omega))\delta\omega$

Cuadro 12.1: Descripción matemática de los distintos tipos de espectros: armónico, inarmónico y ruido.

12.2.1 Nociones sobre el timbre y el espectro

De acuerdo al teorema de Fourier, una función que describe una onda periódica con periodo T se puede descomponer en la suma de señales sinusoidales con periodos T/k, $k = 1, 2, \ldots$, cada una de las cuales tiene su propia fase y amplitud (la cual puede ser cero para algunas componentes). En otras palabras, una onda periódica con frecuencia $f = 1/T$ (llamada *frecuencia fundamental*) está compuesta por señales sinusoidales cuyas frecuencias son de la forma $f_k = kf$, llamadas *armónicos*. Así mismo, se le llama *onda armónica* a cualquier función periódica que pueda representarse como la suma discreta de una serie de señales sinusoidales cuyas frecuencias son todas múltiplos de una frecuencia fundamental.

Dada una señal armónica con frecuencia fundamental f, las frecuencias que no corresponden a múltiplos enteros de f se conocen como *frecuencias inarmónicas*. Si a una señal armónica se le agregan componentes inarmónicas, decimos que la señal resultante es también inarmónica.

Finalmente, es posible también encontrar señales cuyo espectro no puede representarse como una suma discreta de componentes sinusoidales, sino como una suma contínua (i.e., una integral) de componentes cuyas amplitudes y fases están dadas por funciones que describen la densidad espectral de la señal. A estas señales se les llama *ruido* (no en un sentido despectivo) y a la calidad acústica del ruido, dada por la función de densidad de amplitud, se le llama *color*.

El Cuadro 12.1 resume los tres tipos de espectros mencionados. La mayoría de los sonidos que escuchamos son una combinación de ondas armónicas, inarmónicas y ruido.[10]

12.2.2 Waveshapers

Una de las técnicas mas sencillas para alterar el timbre, consiste en distorsionar la forma de onda de un sonido o de un oscilador mediante una función $w(x) : \mathbb{R} \longrightarrow \mathbb{R}$ que transforme una señal de entrada $x(t)$ en una señal $y(t)$ muestra por muestra; es decir,

$$y(t) = w(x(t)).$$

A la función $w(x)$ se le conoce como un *deformador de onda* o *waveshaper*. [1] De acuerdo a las propiedades de los filtros estudiadas en la Práctica 5, un waveshaper es equivalente a un filtro *sin memoria*.

Por supuesto, siendo $w(x)$ una función arbitraria, existe un rango muy amplio de procesos y aplicaciones que pueden representarse mediante un waveshaper; más aún si éste depende de uno o más parámetros que sean modulables. Por poner un ejemplo, un amplificador es un caso particular de waveshaper, para el cual $w(x) = gx$, donde g es la ganancia del amplificador.

La aplicación que da su nombre a los waveshapers es aquella donde se busca transformar la forma de onda de un oscilador. Por ejemplo, se puede partir de un oscilador simple, con poco contenido armónico (onda senoidal o triangular), y distorsionar la forma de onda para incrementar el contenido armónico. Además, es deseable contar con un parámetro que regule el grado de distorsión, y por lo tanto, el contenido armónico de la onda resultante.

En esta práctica probaremos cuatro tipos de waveshapers diseñados para incrementar el contenido armónico de un oscilador mediante distorsión. El primero consiste en un amplificador con saturación dura, el segundo consiste en un plegador de onda, luego un saturador suave, y finalmente un cuantizador. Los cuatro diseños contarán con un parámetro modulable, de manera similar al amplificador controlado por voltaje (VCA). De hecho, para simplificar la implementación, los cuatro waveshapers se derivarán de la clase `VCA`.

12.2.3 Amplificador con saturación dura

Por lo general, las señales de audio digital representadas mediante números de punto flotante tienen un rango dinámico de -1.0 a +1.0. Valores fuera de este rango pueden causar distintos tipos de distorsión, dependiendo de la manera en que estos valores sean convertidos a números enteros, y posteriormente a voltajes analógicos por medio del convertidor análogo digital (DAC). En la mayoría de los casos, esta distorsión se da por saturación, lo cual significa que los valores mayores a +1 serán saturados a +1, y los menores a -1 serán saturados a -1. Una manera de acentuar el efecto de esta saturación consiste en pre-amplificar la señal de entrada por un factor de ganancia $g \geq 1$, tal como lo describe la siguiente función:

$$w(x) = \begin{cases} -1 & \text{si } gx < -1, \\ gx & \text{si } -1 \leq gx \leq 1, \\ 1 & \text{si } gx > 1. \end{cases} \tag{12.1}$$

A este tipo de saturación se le conoce como *saturación dura* o *clipping* (en inglés) ya que el efecto se aplica de manera binaria: o bien los valores son "recortados" (en el caso $|x| > 1$), o no lo son (cuando $|x| \leq 1$). Desde el punto de vista de la forma de onda, este saturador tiende a generar "esquinas" angulares en la señal de salida.

[1] En el área de procesamiento de imágenes, a los waveshapers se les conoce como *funciones de transferencia de tonos*.

12.2.4 Amplificador con plegamiento de onda

En algunos sistemas numéricos, las operaciones cuyos resultados se salen del rango del sistema no generan saturación, sino desbordamiento (*overflow*). El desbordamiento básicamente consiste en que el sistema no cuenta con el número suficiente de dígitos para representar el resultado de alguna operación, por lo que algunos de los dígitos mas significativos serán despreciados. Considere por ejemplo un sistema de numeración decimal con un rango de solamente tres dígitos (es decir, de 0 a 999); en este sistema, la operación $995 + 10$ daría como resultado 5, mientras que la operación $5 - 8$ da como resultado 997. Una manera de visualizar este sistema es pensar que la recta numérica donde residen los números del 0 al 999 se ha plegado para formar un círculo donde el 0 coincidiría con el 1000. Formalmente, estos sistemas numéricos se conocen como *aritmética modular entera*, siendo este ejemplo particular la aritmética módulo 1000. En términos generales, al resultado de cualquier operación en aritmética módulo Z se le debe sumar un múltiplo de Z (posiblemente negativo) que logre que el resultado se mantenga en el rango deseado.

Considere ahora una señal $x[n]$ para la cual el rango dinámico es de -1 a 1, y suponga que deseamos amplificar la señal con un factor de ganancia g, pero utilizando aritmética modular (real, ya no entera) módulo 2. Conforme se incrementa el factor de ganancia, las secciones de la señal que "salen por arriba" (i.e., que sobrepasan el valor 1.0) aparecerán ahora por la parte de abajo (cercanas al -1.0), y viceversa.

Este tipo de distorsión se conoce como *wavefolding* y la función que la implementa es:

$$w(x) = (1 + gx)_{\text{mod } 2} - 1, \tag{12.2}$$

donde la operación $z_{\text{mod } 2}$ (módulo 2) da como resultado el valor $m \in [0, 2)$ tal que $2q + m = z$ para algún entero q, y nuevamente g es un factor de ganancia que se asume mayor o igual a uno.

12.2.5 Saturador suave

Otro tipo de waveshaper está basado en un mapeo polinomial $\phi : [0, 1] \longrightarrow [0, 1]$ de la forma $\phi(x) = x^{1-\alpha}$, idéntico al propuesto en el reto de la Práctica 11. Este tipo de mapeo modifica la curvatura de la señal de entrada, de acuerdo con el parámetro $\alpha \in [0, 1]$. Note que para $\alpha = 0$ se tiene $\phi(x) = x$, por lo que la señal de entrada se mantiene intacta; por otro lado, para $\alpha = 1$ se tiene $\phi(x) = 1$, lo cual resulta en una saturación total.

Existe un detalle que debemos considerar: dado que el rango dinámico de las señales de audio va de -1 a 1, es posible que algunas muestras tomen valores negativos; sin embargo, la función potencia con bases negativas y exponentes reales en general no se encuentra definida dentro de los reales (considere, por ejemplo, el caso $(-1)^{0.5} = \sqrt{-1}$). Para solventar esto, será necesario aplicar la función potencia al valor absoluto de la muestra de entrada, y luego multiplicar

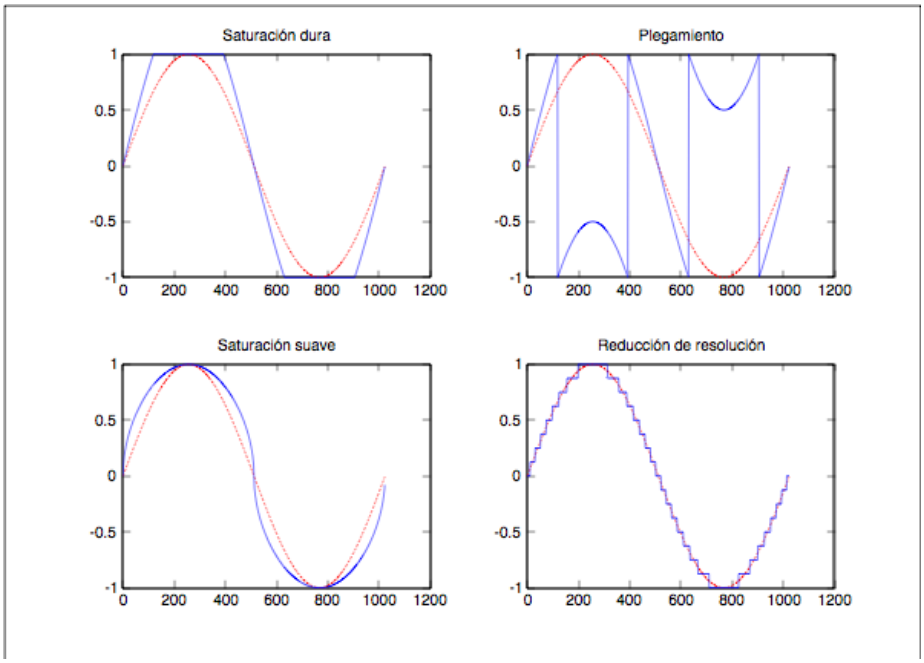


Figura 12.1: Resultado de pasar una onda senoidal por distintos waveshapers.

el resultado por el signo original de la muestra. En otras palabras:

$$w(x) = \text{sgn}(x) \cdot |x|^{1-\alpha}, \tag{12.3}$$

donde $\text{sgn}(x)$ devuelve $+1$ si $x > 0$, -1 si $x < 0$ o cero si $x = 0$.

12.2.6 Reductor de resolución

El último tipo de waveshaper que estudiaremos en esta práctica consiste en un cuantizador que reduce la resolución de los valores de la señal de entrada. Las señales digitales de audio comúnmente se representan utilizando números enteros cuya longitud puede ir desde 8 hasta 32 bits. Por ejemplo, los discos compactos (CDs) utilizan 16 bits por muestra (incluido el bit de signo), mientras que los DVDs de audio y Blu-ray soportan hasta 24 bits por muestra.

Las librerías Beads y Minim, así como la clase UGen que desarrollamos en C++, utilizan números de punto flotante para representar las muestras; sin embargo, estos números siempre son convertidos a enteros con una cierta resolución (en bits) para poder ser enviados al convertidor digital-análogo (DAC) de la interfaz de audio. Podemos reducir artificialmente el número de bits utilizados para representar cada muestra para simular el sonido de convertidores de baja calidad, como los que se encuentran en las computadoras de 8-bits y las consolas de videojuegos de la década de 1980.

El proceso es muy sencillo. Considere una variable x que toma valores contínuos entre 0 y 1. Podemos cuantizar la variable a $M + 1$ niveles multiplicando x por M, lo cual resulta en valores entre 0 y M, luego redondeando para obtener valores *enteros* entre 0 y M, y finalmente dividiendo el resultado entre M para regresar al rango original entre 0 y 1, pero dividido ahora en $M+1$ pasos discretos. Lo anterior se representa mediante la siguiente operación:

$$w(x) = \lfloor Mx + 0.5 \rfloor / M,$$

de manera que $w(x) \in \{0, 1/M, 2/M, \ldots, 1\}$.

Si ahora consideramos que x toma valores entre -1 y 1, entonces la discretización en pasos de tamaño $1/M$ da como resultado un total de $2M + 1$ niveles de cuantización.

Supongamos que en vez de definir el número de niveles de cuantización, deseamos definir el número de bits b que se utilizarán para representar la señal, entonces podemos utilizar $M = 2^{b-1}$ para tener un total de $2^b + 1$ niveles de cuantización (uno mas que los 2^b niveles que se pueden representar con b bits). Note que b y M no necesariamente deben ser enteros.

La Figura 12.1 muestra un ejemplo de la distorsión producida por los cuatro waveshapers para una onda de entrada senoidal.

12.3 Objetivos didácticos

- Presentar al alumno el concepto de filtros sin memoria o waveshapers como herramientas para modificar la forma de onda y el contenido armónico de una señal.
- Implementar y experimentar con algunos tipos de waveshapers
- Utilizar waveshapers y envolventes para producir cambios tímbricos a lo largo de la duración de un sonido

12.4 Material

- Una computadora con alguna de las siguientes combinaciones de software instalado:
  - Processing y Beads
  - Processing y Minim
  - GNU C++ (e.g., MinGW) y PortAudio
- Bocinas o audífonos estéreo

12.5 Procedimiento

Dado que los waveshapers estudiados toman una señal monocanal de entrada, así como una señal moduladora, y generan una señal monocanal a la salida, su estructura e implementación es muy similar a la de un VCA. De hecho, en dos de los cuatro waveshapers, el parámetro modulable corresponde también a un factor de ganancia. Por estos motivos, es buena idea heredar las nuevas clases a partir de la clase `VCA` en lugar de la clase `UGen`.

El primer tipo de waveshaper (saturación dura) se implementará en una UGen llamada `Clipper`. Solamente será necesario implementar el constructor (el cual únicamente llama al constructor de la clase base) y la función callback, la cual implementa la Ecuación 12.1, considerando la modulación de la ganancia mediante la señal moduladora. A continuación se muestra el código propuesto:

```
public class Clipper extends VCA {
  public Clipper(AudioContext ac) {
    super(ac);
  }

  public void calculateBuffer() {
    float[] out = bufOut[0];
    float[] in = bufIn[0];
    float[] mod = bufIn[1];
    float x;

    for (int i = 0; i < bufferSize; i++) {
      x = in[i] * (1 + ganancia + indice_modulacion * mod[i]);
      out[i] = (x > 1.0) ? 1.0 : ((x < -1.0) ? -1.0 : x);
    }
  }
}
```

Para apreciar el efecto de la saturación, es necesario que la ganancia en la etapa de pre-amplificación sea mayor o igual a uno. Por esta razón, se suma uno al factor de ganancia, de manera que cuando el parámetro de la UGen vale cero, la ganancia real es igual a uno, y la señal no se ve afectada. De esta forma, el parámetro de la UGen controla el grado de distorsión que se aplica a la señal. Cabe mencionar que Processing cuenta con la función `constrain()` que puede utilizarse para aplicar la saturación; por ejemplo, `out[i] = constrain(x, -1.0, 1.0);` sin embargo, se ha utilizado código en C para facilitar la portabilidad a otras plataformas.

La implementación del segundo waveshaper (plegamiento de onda) es muy similar al primero, con la única diferencia que en lugar de saturar se calculará el módulo-2 de la señal (ajustando el offset debidamente). De igual manera, se incrementará artificialmente la ganancia por uno, considerando que el parámetro de la UGen represente el grado de distorsión. La UGen propuesta se llama `Folder` y consta del siguiente código:

```
public class Folder extends VCA {
  public Folder(AudioContext ac) {
    super(ac);
  }

  public void calculateBuffer() {
    float[] out = bufOut[0];
    float[] in = bufIn[0];
    float[] mod = bufIn[1];
    float x;

    for (int i = 0; i < bufferSize; i++) {
      x = in[i] * (1 + ganancia + indice_modulacion * mod[i]);
      x = (x + 1.0) % 2.0;
      if (x < 0) x += 2.0;
      out[i] = x - 1.0;
    }
  }
}
```

Note que en Processing el operador módulo `%` está definido tanto para números de punto flotante como para enteros. En C++ este operador solamente está definido para enteros, por lo que habría que utilizar en su lugar la función `fmod()` de la librería `cmath`.

A continuación, tenemos el saturador suave basado en un mapeo monomial, al que llamaremos `Shaper`. En este caso, el exponente α debe ser igual a uno menos el nivel de distorsión aplicado a la señal. Así, la implementación de la UGen `Shaper` queda como sigue:

```
public class Shaper extends VCA {
  public Shaper(AudioContext ac) {
    super(ac);
  }

  public void calculateBuffer() {
    float[] in = bufIn[0];
    float[] mod = bufIn[1];
    float[] out = bufOut[0];
    float alpha;

    for (int i = 0; i < bufferSize; i++) {
      alpha = 1 - (ganancia + indice_modulacion * mod[i]);
      out[i] = (in[i] >= 0) ? pow(in[i], alpha) : -pow(-in[i], alpha);
    }
  }
}
```

Finalmente, se implementa el reductor de resolución, al que llamaremos `Crusher`. En esta UGen, el parámetro de ganancia se utilizará mas bien como el grado de reducción de la resolución de la señal, dado en bits. Cuando la ganancia sea igual a cero, la resolución será de 16 bits (15 bits mas el bit de signo), mientras que si la ganancia es uno, la resolución será de un bit. En este caso, los valores de ganancia se limitarán siempre al rango de cero a uno, y se asume que la señal de entrada toma valores entre -1 y 1.

```
public class Crusher extends VCA {
    public Crusher(AudioContext ac) { super(ac); }

    public void calculateBuffer() {
      float[] in = bufIn[0];
      float[] mod = bufIn[1];
      float[] out = bufOut[0];
      float bits;
      float levels;

      for (int i = 0; i < bufferSize; i++) {
        bits = 15 * constrain(1 - (ganancia + indice_modulacion * mod[i]), 0, 1);
        levels = pow(2, bits);
        out[i] = round(in[i] * levels) / levels;
      }
    }
}
```

12.5.1 Aplicación de prueba

Para probar los waveshapers, modificaremos ligeramente la aplicación de prueba de la Práctica 9, reemplazando el amplificador utilizado por cada uno de los waveshapers. Esto permitirá usar el mouse para controlar tanto la frecuencia/tono como la distorsión del sonido generado.

Al igual que con un amplificador o un VCA, un cambio abrupto en la ganancia (producido, por ejemplo, por un movimiento rápido del mouse), puede generar una discontinuidad en la señal de salida, que se traduce en artefactos audibles. Una manera de evitar que un parámetro cambie abruptamente consiste en controlarlo mediante una envolvente: la interfaz de usuario no cambiará directamente el parámetro, sino la envolvente que lo controla, pero con un tiempo de retardo suficientemente largo para lograr una transición suave, y a la vez no tan largo como para percibirse como una falta de responsividad en la interfaz. Por este motivo, agregaremos una envolvente para modular el parámetro de distorsión/ganancia del waveshaper.

A continuación se muestra la sección de inicialización de la aplicación:

```
AudioContext ac;
VCA ws;
```

```
Rampa env;
Senoidal osc;

int inicio = 1;  // octava inicial
int rango = 4;   // rango en octavas
boolean cuantiza = true;

void setup() {
  size(800, 600);

  ac = new AudioContext();

  // Para probar otros waveshapers,
  // reemplace el nombre del constructor en la siguiente línea
  ws = new Clipper(ac);

  env = new Rampa(ac); // envolvente que modula el waveshaper
  osc = new Senoidal(ac, 440);

  ws.setEntrada(osc);
  ws.setModulador(env);
  ws.setIndice(0.9);
  ac.out.addInput(ws);
  ac.start();
}
```

Note que el objeto `ws`, que representa al waveshaper, se declara de clase `VCA`. En realidad, este objeto podrá ser de clase `VCA` o cualquiera de sus clases heredadas. Esto permite cambiar el tipo de waveshaper simplemente modificando el nombre del constructor al momento de inicializar el objeto `ws`, sin tener que cambiar ninguna otra línea de código. Por supuesto, también es posible reemplazar el waveshaper por un VCA estándar, resultando en la aplicación original de la Práctica 9.

La interfaz con el usuario se limita al movimiento del mouse, en el eje horizontal para controlar el tono y en el eje vertical para controlar la distorsión. Note que la posición Y del mouse no controla directamente el parámetro del waveshaper, sino el nivel de la envolvente que modula dicho parámetro (con un tiempo de transición de 20 ms). De esta manera, los cambios abruptos en la posición del mouse no generan artefactos audibles. Para ver la diferencia, pruebe cambiando el tiempo de transición a 0 ms. Para completar la interfaz, se utilizan algunas teclas para controlar el rango y transposición del instrumento, así como la cuantización. A continuación se presentan las funciones que lidian con el movimiento del mouse y el teclado:

```
void mouseMoved() {
  float oct = inicio + (float)rango * mouseX / width;
```

```
  if (cuantiza) oct = round(oct * 12) / 12.0;
  osc.setFrecuencia(55 * pow(2, oct));
  env.cambia(1 - (float)mouseY / (height - 1), 20);
}

void keyPressed() {
  if (key >= '1' && key <= '9') rango = key - '0';
  if (keyCode == LEFT) inicio--;
  if (keyCode == RIGHT) inicio++;
  if (key == 'C' || key == 'c') cuantiza = !cuantiza;
}
```

Note que el nivel de distorsión está limitado de cero a uno por la interfaz; sin embargo, los waveshapers y el VCA permiten asignar cualquier valor al parámetro de ganancia, por lo que es posible obtener efectos aún mas drásticos.

Finalmente, la función `draw()` se utiliza nuevamente para mostrar información sobre algunos de los parámetros, graficar la forma de onda, y visualizar un teclado simple como referencia (en la práctica anterior se visualiza un teclado mas sofisticado).

```
void draw() {
  background(0);
  text("Rango = " + rango + " octavas", 10, 10);
  text("Octava inicial = " + inicio, 10, 20);
  text("Frecuencia = " + osc.frecuencia(), 10, 30);
  text("Cuantizacion = " + (cuantiza ? "ON" : "OFF"), 10, 40);
  stroke(0, 255, 0);
  simplescopio(ws.getOutBuffer(0));


  // dibuja teclado
  int[] colortecla = { 1, 0, 1, 1, 0, 1, 0, 1, 1, 0, 1, 0 };
  for (int x = 0; x < width; x++) {
    int nota = (int)round(12.0 * rango * x / width) % 12;
    stroke(colortecla[nota] * 191 + 64, 64);
    line(x, 0, x, height);
  }
}
```

12.6 Evaluación y reporte de resultados

1.- Considere un oscilador senoidal cuya salida se conecta a un waveshaper. Puede el waveshaper modificar la frecuencia fundamental del oscilador? Explique su respuesta.

2.- Los filtros de primer y segundo orden estudiados en las Prácticas 5 y 6, respectivamente, no producen resultados muy interesantes sobre ondas sinusoidales, ya que éstas carecen de contenido armónico que el filtro pueda resaltar o atenuar. Sin embargo, es posible aplicar primero un waveshaper al oscilador para generar ondas con alto contenido armónico, y posteriormente utilizar un filtro para moldear el espectro y el timbre del sonido resultante. Pruebe lo anterior insertando un filtro pasa-banda resonante de segundo orden entre el waveshaper y la salida de audio, y agregue elementos de interfaz de usuario para manipular la frecuencia de corte y calidad (Q) del filtro.

3.- Modifique la aplicación reemplazando el oscilador senoidal por la señal proveniente de la entrada física de audio y explore el efecto de los waveshapers en su propia voz. Dado que ya no se tiene el oscilador, puede utilizar la posición X del mouse para manipular otro parámetro del sistema, por ejemplo, la frecuencia de corte de un filtro pasa-banda colocado después del waveshaper.

4.- Una forma de onda $x(t)$ es simétrica alrededor de t_0 si se cumple que $x(t_0 + t) = x(t_0 - t)$ para todo t. Por ejemplo, las funciones seno y coseno son simétricas en este sentido. Demuestre que si un waveshaper recibe como entrada una onda simétrica, entonces la salida también será simétrica.

12.7 Retos

Experimente diseñando sus propios waveshapers. Los cuatro waveshapers estudiados en esta práctica solo representan un punto de partida, pero es posible utilizar cualquier función $w : \mathbb{R} \longrightarrow \mathbb{R}$ a manera de waveshaper, o incluso combinar múltiples waveshapers para crear nuevos tipos de distorsión.

12.8 Conclusiones

Redacte en el recuadro sus conclusiones acerca de los conceptos aprendidos, las actividades realizadas y los objetivos planteados en esta práctica. Si lo desea, puede considerar las preguntas de apoyo que se muestran abajo.

Preguntas de apoyo

- Qué es un waveshaper y qué propiedades tiene como sistema o filtro?
- Qué limitaciones tiene un waveshaper para modelar el timbre o forma de onda de un sonido?
- Porqué es útil poder modular los parámetros de un waveshaper mediante una envolvente?

Capítulo 13

Tablas de ondas

Nombre del estudiante	Calificación

13.1 Relación con los programas de estudio

Materia	Unidad y tema
Procesamiento Digital de Señales (IE / IT / IB)	
Procesamiento de Señales de Audio (IE)	2.6.- Tablas de ondas 2.7.- Interpolación

13.2 Introducción

Considere el oscilador senoidal basado en acumulación de fase que se implementó en la Práctica 9. Para este oscilador, se conserva el valor de la fase (en radianes) en una variable, la cual se incrementa en cada muestra en un valor proporcional a la frecuencia del oscilador y al periodo de muestreo del sistema. Además, para mantener la estabilidad numérica, se debe tener cuidado de que el valor de la fase no crezca indefinidamente, por lo que se sugirió tomar el módulo 2π después de incrementar la fase. Con esto en mente, el pseudo-código que calcula cada muestra de salida del oscilador es el siguiente:

```
salida = sin(fase);
incremento_fase = 2 * PI * frecuencia / frecuencia_muestreo;
fase = fase + incremento_fase;
fase = fase % (2 * PI);
```

En este caso, la fase está dada en radianes, pero con el objeto de generalizar la implementación a otras formas de onda, sería conveniente expresar la fase en

ciclos. Bajo esta consideración, uno puede reemplazar el pseudo-código anterior con el siguiente:

```
salida = sin(2 * PI * fase);
incremento_fase = frecuencia / frecuencia_muestreo;
fase = fase + incremento_fase;
fase = fase - floor(fase);
```

Note que se utiliza la expresión `fase - floor(fase)` como una alternativa para calcular el módulo uno de la fase (es decir, la parte fraccionaria) sin tener que hacer una división como en `fase % 1`; esta expresión funciona incluso cuando la fase se vuelve negativa, lo cual puede ocurrir al modular la frecuencia. En cualquier caso, esto asegura que el valor de la fase estará siempre entre cero y uno, e indicará en qué punto del ciclo se encuentra el oscilador.

Ahora bien, supongamos que deseamos generar una forma de onda distinta a la senoidal. Esto es sencillo de hacer reemplazando la primer línea del pseudo-código anterior, donde se calcula la salida. Por ejemplo, podemos usar alguna de las siguientes alternativas para generar las formas de onda típicas de un generador de funciones o de un oscilador analógico (diente de sierra, triangular, cuadrada, pulso):

```
// genera onda diente de sierra
salida = fase * 2 - 1;

// genera onda cuadrada
salida = (fase < 0.5) ? 1 : -1;

// genera onda de pulso con ancho de pulso pw
salida = (fase < pw) ? 1 : -1;

// genera onda triangular
salida = 4 * ((fase < 0.5) ? fase : (1 - fase)) - 1;
```

Note que todas las ondas anteriores tienen un rango de -1 a 1, pensando en que serían utilizadas en el contexto del audio digital. Verifique que se obtienen las formas de onda mencionadas graficando manualmente un periodo de la salida para cada tipo de onda (variando la fase de 0 a 1), o bien, creando una nueva UGen (heredada de `Senoidal`) para probar el código.

13.2.1 Tablas de ondas

Suponga ahora que deseamos generar cualquier forma de onda arbitraria. Una alternativa consiste en tener una representación discretizada de un periodo de la onda almacenada en un arreglo o una tabla. Supongamos que tal representación se encuentra en el arreglo `tabla[]` de tamaño `N`. Para implementar un oscilador que genere esa forma de onda simplemente hay que calcular la salida como

```
salida = tabla[fase * N];
```

Conforme la fase va de cero a uno, el subíndice `fase * N` recorre todo el arreglo, dando como salida justamente la forma de onda almacenada en el arreglo. A esta técnica se le conoce como *tabla de onda* o *wavetable*. Este tipo de osciladores son empleados en la mayoría de los sintetizadores digitales.

Note que, en general, el valor de `fase * N` no será entero; sin embargo, es utilizado como subíndice dentro de un arreglo. Algunos lenguajes de programación (e.g., C y C++) realizarán una conversión automática a entero, mientras que en otros lenguajes (como Processing y Java) se debe hacer una conversión explícita. Una mejor alternativa consiste en interpolar linealmente entre muestras adyacentes de la tabla para obtener una forma de onda mas suave. Esto requiere algunas operaciones adicionales, como se muestra a continuación:

```
indice = fase * N;   // subíndice no entero
izq = (int)indice;   // subíndice correspondiente a la muestra izquierda
der = (izq + 1) % N; // subíndice correspondiente a la muestra derecha
frac = indice - izq; // parte fraccionaria del subíndice
salida = (1 - frac) * tabla[izq] + frac * tabla[der];
```

Note que conforme `indice` varía entre un entero y el siguiente; es decir, desde `izq` hacia `der`, el valor de salida del oscilador varía de `tabla[izq]` hacia `tabla[der]`.

También es posible utilizar esquemas más sofisticados de interpolación, como interpolación polinomial (cuadrática o cúbica) o interpolación por splines [10].

13.2.2 Modulación de frecuencia

Como detalle final, vale la pena incorporar a los nuevos osciladores la posibilidad de modular su frecuencia (FM, por sus siglas en inglés) de manera continua (es decir, muestra a muestra) a través de alguna señal moduladora. Dado que la sensibilidad del oído a la frecuencia sigue una curva logarítmica, es buena idea aplicar una modulación de tipo exponencial a la frecuencia; por ejemplo, que la magnitud de la modulación esté expresada en octavas en lugar de Hertz. Esto se logra calculando la frecuencia instantánea como

$$f = f_0 \cdot 2^m,$$

donde f_0 es la frecuencia base del oscilador y m es la magnitud de la modulación en octavas. Por lo general, m se obtiene multiplicando el valor de la señal moduladora en el instante actual por un índice de modulación que controla la amplitud de la señal moduladora.

También es posible modular la frecuencia de manera lineal, calculando la frecuencia instantánea como

$$f = f_0 \cdot (1 + m),$$

donde nuevamente f_0 es la frecuencia base y m es la magnitud total de la modulación (es decir, el producto de la señal moduladora por el índice de modulación). Para este tipo de modulación es posible que se generen frecuencias negativas

(cuando $m < 1$), lo cual debe tomarse en cuenta en la implementación del oscilador para asegurarse de que la fase siempre está entre cero y uno.

Bajo estas consideraciones, el pseudo-código completo para calcular cada muestra de la salida de un oscilador basado en tabla de ondas con interpolación lineal y modulación de frecuencia es el siguiente:

```
// calcula la salida interpolando la tabla de ondas
indice = fase * N;   // subíndice no entero
izq = (int)indice;   // subíndice correspondiente a la muestra izquierda
der = (izq + 1) % N; // subíndice correspondiente a la muestra derecha
frac = indice - izq; // parte fraccionaria del subíndice
salida = (1 - frac) * tabla[izq] + frac * tabla[der];

// actualiza la fase considerando la modulación de frecuencia
if (modulacion_exponencial)
   frecuencia_instantanea = frecuencia * pow(2, indice_modulacion * señal_moduladora);
else
   frecuencia_instantanea = frecuencia * (1 + indice_modulacion * señal_moduladora);
endif
incremento_fase = frecuencia_instantanea / frecuencia_muestreo;
fase = fase + incremento_fase;
fase = fase - floor(fase);
```

Aún en el caso de implementaciones digitales, a los osciladores cuya frecuencia puede ser modulada por una señal externa se les conoce como osciladores controlados por voltaje, o simplemente VCO (voltage controlled oscillator).

13.3 Objetivos didácticos

- Conocer el funcionamiento de un oscilador basado en una tabla de ondas
- Aplicar interpolación lineal para mejorar la calidad de ciertos procesos de audio
- Implementar una interfaz de usuario que permita dibujar una forma de onda arbitraria mientras ésta se reproduce mediante un oscilador

13.4 Material

- Una computadora con alguna de las siguientes combinaciones de software instalado:
  - Processing y Beads
  - Processing y Minim
  - GNU C++ (e.g., MinGW) y PortAudio
- Bocinas o audífonos estéreo

13.5 Procedimiento

En esta práctica implementaremos una nueva UGen `Oscilador` que consistirá en un oscilador basado en tabla de ondas con interpolación y modulación de frecuencia. No solo será una de las UGen mas ambiciosas hasta este punto, sino que en prácticas futuras se le agregará mayor funcionalidad. Para probar la UGen, implementaremos una interfaz similar a la de las prácticas anteriores, a través de la cual uno pueda controlar el tono y la amplitud del oscilador mediante el mouse, pero también será posible dibujar libremente la forma de onda del oscilador.

A continuación se muestra el código de la clase `Oscilador`, seguido de una descripción de sus miembros y métodos:

```
public class Oscilador extends UGen {
  float fase;
  float frecuencia;
  float frecInst;
  float[] tabla;
  float indiceModulacion;
  boolean modulacionExponencial;

  public Oscilador(AudioContext context) {
    super(context, 1, 1);
    setSenoidal(1024);
    modulacionExponencial = true;
  }

  public Oscilador(AudioContext context, float f) {
    super(context, 1, 1);
    frecuencia = f;
    setSenoidal(1024);
  }

  float frecuenciaInst() { return frecInst; }

  float frecuencia() {
    return frecuencia;
  }

  void setFrecuencia(float f) {
    frecuencia = f;
  }

  void reinicia() {
    fase = 0;
  }
```

```
float indice() { return indice_modulacion; }

void setIndice(float g) { indice_modulacion = g; }

void setModulador(UGen ugen) { addInput(0, ugen, 0); }

void setTabla(float[] t) { arrayCopy(t, tabla); }

void setFMLineal() { modulacionExponencial = false; }

void setFMExponencial() { modulacionExponencial = true; }

void setSenoidal(int res) {
  tabla = new float[res];
  for (int i = 0; i < res; i++) {
    tabla[i] = sin(2.0 * PI * i / res);
  }
}

public void calculateBuffer() {
  float[] out = bufOut[0];
  float[] mod = bufIn[0];
  float dfase, frac, indice;
  int izq, der;
  if (tabla == null) return;

  for (int i = 0; i < bufferSize; i++) {
    // Calcula muestra de salida
    indice = fase * tabla.length;
    izq = int(indice);
    frac = indice - izq;
    der = (izq + 1) % tabla.length;
    out[i] = (1 - frac) * tabla[izq] + frac * tabla[der];

    // Actualiza la fase
    if (modulacionExponencial) {
      frecInst = frecuencia * pow(2, indiceModulacion * mod[i]); // FM exponencial
    } else {
      frecInst = frecuencia * (1.0 + indiceModulacion * mod[i]); // FM lineal
    }
    dfase = frecInst / context.getSampleRate();
    fase += dfase;
    fase = fase - floor(fase);
  }
}
```

```
}
```

Al igual que la UGen `Senoidal`, el `Oscilador` cuenta con miembros de datos para almacenar la `frecuencia` y la `fase` del oscilador, salvo que en este caso se trata de la frecuencia base (alrededor de la cual se pueden producir modulaciones) y la fase en ciclos (en lugar de radianes). Además, se agregan los miembros `frecInst` (la frecuencia instantánea, considerando el efecto de la modulación), `tabla` (el arreglo que contiene la tabla de onda) e `indice_modulacion` (el factor de ganancia aplicado a la señal moduladora). Finalmente, se incluye una variable booleana para definir el tipo de modulación: lineal o exponencial. Por defecto, se selecciona la modulación exponencial en el constructor, pero se proporcionan métodos los métodos `setFMLineal()` y `setFMExponencial()` para modificar este comportamiento.

Los arreglos en Processing son objetos que conocen su propio tamaño (dado por el miembro `length`). En el caso de C o C++, será necesario agregar a la UGen `Oscilador` un miembro adicional que represente el tamaño de la tabla de ondas.

A continuación se encuentran los constructores de `Oscilador`, que simplemente llaman al constructor de `UGen` para construir una UGen con una entrada (la señal moduladora) y una salida, inicializan la frecuencia del oscilador y la tabla de ondas. La tabla de ondas se inicializa con una representación de una onda senoidal con una resolución de 1024 muestras; esto se hace llamando al método `setSenoidal()` que se implementa después y no requiere mayor descripción.

Posteriormente se tienen múltiples funciones *set* y *get* para manipular la frecuencia base y el tipo de modulación, reinicializar la fase, conectar la UGen moduladora, manipular el índice de modulación, y modificar la tabla de ondas. El alumno es libre de agregar otras funciones, por ejemplo, para crear ondas cuadradas, triangulares o diente de sierra.

Finalmente, la función callback `calculateBuffer()` implementa el pseudocódigo descrito al final de la sección de Introducción para calcular las muestras de salida interpolando la tabla de ondas y actualizando la fase de según la frecuencia modulada.

13.5.1 Programa de prueba

El programa de prueba para la UGen `Oscilador` será similar al utilizado en la práctica anterior, con tres diferencias importantes:

1. Se empleará solamente la mitad inferior de la ventana de la aplicación para controlar el instrumento. La mitad superior de la interfaz se utilizará para dibujar y manipular la forma de onda deseada.

2. Se utilizará un VCA estándar, en lugar de un waveshaper, para controlar el volumen del instrumento. Si el alumno lo desea, puede probar cambiando el VCA por un waveshaper o por un filtro pasa-bajas para mayor control del timbre.

3. Para probar la modulación de frecuencia del oscilador, utilizaremos una envolvente para controlar la velocidad de los cambios en frecuencia (así como se utilizó una envolvente para controlar la ganancia en la práctica anterior). Este efecto se conoce como *glide* o *portamento* y hace que el instrumento sea mas fácil de controlar.

De esta forma, la sección de inicialización queda como se muestra a continuación:

```
AudioContext ac;
VCA vca;
Rampa env_amp;
Rampa env_frec;
Oscilador osc;

int inicio = 1;
int rango = 4;
float portamento = 100;
boolean cuantiza = true;

void setup() {
  size(800, 600);

  ac = new AudioContext();
  vca = new VCA(ac);
  env_amp = new Rampa(ac);
  env_frec = new Rampa(ac);
  osc = new Oscilador(ac, 440);
  osc.setSenoidal(128);

  osc.setModulador(env_frec);
  osc.setIndice(1);
  osc.setFrecuencia(55);
  vca.setEntrada(osc);
  vca.setModulador(env_amp);
  vca.setIndice(0.9);
  ac.out.addInput(vca);
  ac.start();
}
```

Note que ahora la arquitectura del sistema incluye dos envolventes: `env_amp` y `env_frec`, para modular, respectivamente, la amplitud y la frecuencia. Además, se incluye una nueva variable global `portamento` que representa el tiempo que tarda la frecuencia en llegar al valor objetivo cuando se selecciona una nota al mover el mouse horizontalmente. Dado que ahora la frecuencia del instrumento se controlará mediante la envolvente `env_frec`, la frecuencia base del oscilador se fijará en 55 Hz (la nota La tres octavas por debajo del La central)

y el índice de modulación de frecuencia se fija en uno. La tabla de ondas se inicializa con una representación de una onda senoidal con 128 muestras de resolución. Intencionalmente se utiliza una resolución baja (128 muestras) para resaltar el efecto de la interpolación lineal.

A continuación implementaremos la función `draw()` en la cual se grafica la forma de onda del oscilador (es decir, la tabla de onda) en la parte superior de la ventana, y la salida del sistema (específicamente, la salida del VCA) en la parte inferior. Así mismo, la parte inferior se empleará como interfaz para controlar el tono y volumen del sintetizador, para lo cual se bosqueja un teclado musical, y finalmente se despliegan algunos de los parámetros controlables.

```
void draw() {
  background(0);

  pushMatrix();
  translate(0, -height / 4);
  stroke(64);
  line(0, height / 2, width, height / 2);
  stroke(200);
  simplescopio(osc.tabla);
  stroke(0, 255, 0);
  translate(0, height / 2);
  simplescopio(vca.getOutBuffer(0));
  popMatrix();

  // dibuja teclado
  int[] colortecla = { 1, 0, 1, 1, 0, 1, 0, 1, 1, 0, 1, 0 };
  for (int x = 0; x < width; x++) {
    int nota = (int)round(12.0 * rango * x / width) % 12;
    stroke(colortecla[nota] * 191 + 64, 64);
    line(x, height / 2, x, height);
  }

  // despliega parámetros
  int y = height - 100;
  text("Rango (1-9) = " + rango + " octavas", 10, y + 10);
  text("Octava inicial (</>) = " + inicio, 10, y + 20);
  text("Frecuencia = " + osc.frecuenciaInst(), 10, y + 30);
  text("Cuantizacion = " + (cuantiza ? "ON" : "OFF"), 10, y + 40);
  text("Ganancia / Distorsion = " + env_amp.valor(), 10, y + 50);
  text("Portamento (v/^) = " + portamento + " ms", 10, y + 60);
}
```

Finalmente implementaremos la interfaz de usuario por medio del mouse y del teclado. Para esto, emplearemos tres funciones callback de Processing. La función `mouseMoved()` responderá a los movimientos del mouse sobre la mitad

inferior de la ventana, para controlar el tono del oscilador y la ganancia del VCA (a través de las respectivas envolventes), de manera similar a como se ha hecho en prácticas anteriores, pero ajustando el rango de ganancia a la mitad inferior de la ventana:

```
void mouseMoved() {
  float y = height / 2;
  if (mouseY > y) {
    float oct = inicio + (float)rango * mouseX / width;
    if (cuantiza) oct = round(oct * 12) / 12.0;
    env_frec.cambia(oct, portamento);
    env_amp.cambia(1 - (mouseY - y) / (y - 1), 20);
  }
}
```

En seguida, utilizaremos la función `mouseDragged()` para permitir al usuario modificar la forma de onda que se muestra en la parte superior de la ventana al arrastrar el mouse sobre la misma. El único detalle importante es asegurarse de no acceder a subíndices de la tabla fuera de rango:

```
void mouseDragged() {
  float y = height / 2;
  if (mouseY < y) {
    int x = osc.tabla.length * mouseX / width;
    if (x < 0) x = 0;
    if (x >= osc.tabla.length) x = osc.tabla.length - 1;
    osc.tabla[x] = (1 - 2 * mouseY / y);
  }
}
```

Finalmente, agregaremos algunos comandos del teclado mediante la función `keyPressed()`. Además de los comandos ya utilizados para manipular el rango en octavas (dígitos del 1 al 9), la transposición del instrumento (flechas laterales) y la cuantización (letra 'C'), se podrá modificar el tiempo de portamento (flechas arriba y abajo) y reinicializar la forma de onda a una senoidal (letra 'S'). El código que implementa estas funciones es el siguiente:

```
void keyPressed() {
  if (key >= '1' && key <= '9') rango = key - '0';
  if (keyCode == LEFT) inicio--;
  if (keyCode == RIGHT) inicio++;
  if (keyCode == UP) portamento += 100;
  if (keyCode == DOWN && portamento >= 100) portamento -= 100;
  if (key == 'C' || key == 'c') cuantiza = !cuantiza;
  if (key == 'S' || key == 's') osc.setSenoidal(128);
}
```

13.6 Evaluación y reporte de resultados

1.- Uno podría preguntarse si la interpolación lineal es realmente necesaria. Una manera fácil de verificarlo es eliminar la interpolación lineal y dar como salida simplemente `out[i] = tabla[izq]` en la función callback de la clase `Oscilador`. Pruebe esta modificación asignando al oscilador una forma de onda senoidal (la onda por defecto) en una tabla de tamaño 128. Describa las diferencias tanto visuales como audibles entre usar y no usar interpolación lineal.

2.- Agregue a la clase `Oscilador` métodos similares a `setSenoidal()` para generar ondas clásicas: triangular, cuadrada y diente de sierra. Agregue comandos de teclado a la interfaz del programa de prueba para seleccionar las distintas formas de onda.

3.- La interfaz del programa de prueba tiene el inconveniente de que si se arrastra el mouse demasiado rápido al dibujar la forma de onda, aparecerán discontinuidades en la onda. Esto se debe a que se producen cambios grandes en la posición del mouse. Utilice las variables de sistema `pmouseX` y `pmouseY` dentro de la función `mouseDragged()` junto con interpolación lineal para modificar adecuadamente todas las muestras de la tabla de ondas que corresponden al segmento recorrido por el mouse.

13.7 Retos

Si bien el oscilador implementado en esta práctica puede generar prácticamente cualquier forma de onda, no deja ésta de ser una onda estática, sin la posibilidad de modular el timbre a lo largo del tiempo. Una alternativa consiste en incorporar al oscilador dos tablas de ondas e interpolar bilinealmente entre ellas mediante un parámetro adicional. El siguiente pseudo-código ilustra la idea anterior:

```
// calcula la salida interpolando la tabla de ondas
indice = fase * N;   // subíndice no entero
izq = (int)indice;   // subíndice correspondiente a la muestra izquierda
der = (izq + 1) % N; // subíndice correspondiente a la muestra derecha
frac = indice - izq; // parte fraccionaria del subíndice
onda1 = (1 - frac) * tabla1[izq] + frac * tabla1[der];
onda2 = (1 - frac) * tabla2[izq] + frac * tabla2[der];
salida = (1 - forma) * onda1 + forma * onda2;
```

donde `tabla1` y `tabla2` son las tablas de onda entre las que se desea interpolar y `forma` es el parámetro de interpolación (número de punto flotante entre 0 y 1). Implemente un nuevo oscilador que incorpore dos tablas de onda y permita interpolar entre ellas. El oscilador debe incorporar la modulación de la forma de onda mediante una señal externa.

13.8 Conclusiones

Redacte en el recuadro sus conclusiones acerca de los conceptos aprendidos, las actividades realizadas y los objetivos planteados en esta práctica.

Capítulo 14

Líneas de retardo

Nombre del estudiante	Calificación

14.1 Relación con los programas de estudio

Materia	Unidad y tema
Procesamiento Digital de Señales (IE / IT / IB)	1.3.- Sistemas discretos y sus características 1.4.- Sistemas lineales e invariantes en el tiempo
Procesamiento de Señales de Audio (IE)	2.5.- Líneas de retardo 2.7.- Interpolación 2.8.- Líneas de retardo fraccional y variable 4.4.- Corrimiento en frecuencia y vibrato

14.2 Introducción

Un gran número de procesos de audio requieren el uso de retardos, los cuales corresponden a sistemas de la forma

$$y[n] = x[n - d],$$

donde $d \geq 0$ es el tiempo de retardo.

Por sí solo, un retardo puede considerarse como un filtro *pasa-todo*. Claramente, la función de transferencia del retardo es simplemente $H(z) = z^{-d}$, por lo cual $|H(e^{j\omega})| = |e^{-j\omega d}| = 1$. Sin embargo, cuando se combina la salida de un retardo con la señal original, o cuando se combinan las señales de múltiples retardos, los desfasamientos entre las distintas señales retardadas producirán

interferencia, que para algunas frecuencias será constructiva y para otras destructiva, dando lugar a distintas curvas de respuesta en frecuencia.

Un claro ejemplo son los filtros lineales e invariantes en el tiempo, que de manera general se escriben utilizando ecuaciones en diferencias que involucran términos de la forma $x[n]$, $x[n-1]$, $x[n-2]$, etc., así como términos de la forma $y[n-1]$, $y[n-2]$, etc., en el caso de filtros recursivos. En la mayoría de los casos, los tiempos de retardo involucrados en este tipo de filtros son fijos y limitados por el orden del filtro, por lo que las muestras anteriores a n de las señales x y y pueden almacenarse en variables individuales. Esto se hizo en las prácticas 5, 6 y 7, donde se calculaba la salida del filtro y posteriormente se transferían los valores de las variables a manera de cascada para la siguiente iteración. El siguiente pseudo-código ilustra lo anterior para el caso general de un filtro de segundo orden:

```
xn = entrada
yn = a1 * ynm1 + a2 * ynm2 + b0 * xn + b1 * xnm1 + b2 * xnm2;
ynm2 = ynm1;
ynm1 = yn;
xnm2 = xnm1;
xnm1 = xn;
salida = yn;
```

Existen también muchos procesos que requieren retardos donde los tiempos de retardo son considerablemente más largos que unas cuantas muestras. Por ejemplo, se requerien retardos largos para la generación de ecos y primeras reflexiones de la reverberación. En estos casos se vuelve impráctico almacenar las muestras anteriores de la señal en variables individuales, y es aún menos práctico transferir en cascada los valores (imagine el caso de un proceso que requiera mantener $x[n-100]$ o $x[n-10000]$). Una manera eficiente de lograr esto consiste en utilizar un buffer circular para guardar las muestras anteriores a $x[n]$, desde $x[n-1]$ hasta $x[n-N]$, donde N es el tamaño del buffer. Dado que se tiene acceso a las N muestras anteriores a $x[n]$, las líneas de retardo también proporcionan un mecanismo ideal para implementar filtros FIR causales arbitrarios.

Por otra parte, existen procesos donde es imprescindible utilizar retardos con tiempos de retardo no enteros. Imagine, por ejemplo, que por alguna razón necesita un retardo cuyo tiempo de retardo es precisamente el periodo de una nota musical específica. Por ejemplo, la nota La = 440 Hz tiene un periodo de 1/440 segundos, equivalente a 100.2273 muestras a una frecuencia de muestreo de 44100 Hz. Para estimar una señal con este tiempo de retardo, uno podría interpolar entre dos muestras con subíndices enteros (e.g., entre $x[n-101]$ y $x[n-100]$).

Finalmente, se tiene el caso de procesos donde se requiere un retardo con un tiempo de retardo variable; es decir, donde se aplica modulación al tiempo de retardo para producir distintos tipos de efectos, como el efecto Doppler.

Los procesos mencionados arriba serán estudiados en las prácticas de nivel avanzado, pero todos ellos requieren el uso de retardos con tiempos largos,

fraccionarios y/o variables. Es por eso que el objetivo de esta práctica consiste en escribir una UGen que implemente este tipo de retardos.

14.3 Objetivos didácticos

- Comprender las diferencias entre retardos fijos, variables, continuamente variables y fraccionales, así como los retos técnicos que conlleva la implementación digital de cada uno de ellos.
- Entender cualitativamente la relación entre un retardo continuamente variable y el efecto Doppler.
- Implementar distintos tipos de retardos y analizar su efecto en una señal de prueba.

14.4 Material

- Una computadora con alguna de las siguientes combinaciones de software instalado:
  - Processing y Beads
  - Processing y Minim
  - GNU C++ (e.g., MinGW) y PortAudio
- Bocinas o audífonos estéreo

14.5 Procedimiento

Para implementar la línea de retardo utilizaremos una estructura de datos conocida como *buffer circular* o *cola circular*. Este es un arreglo de tamaño fijo N donde el último elemento está "conectado" con el primero, en el sentido de que al ir escribiendo o leyendo secuencialmente el arreglo, después de recorrer el último elemento se llega nuevamente al primero. Esto se logra simplemente haciendo que las variables mediante las que se indexa o apunta a los elementos del arreglo se incrementen con módulo N, donde N es el tamaño del buffer. De esta manera, es posible escribir y leer del arreglo infinitamente, siempre y cuando el apuntador de lectura nunca alcance al de escritura.

En una línea de retardo, el apuntador de lectura se encuentra exactamente d posiciones detrás del de lectura, donde d es el tiempo de retardo. Para que el buffer circular funcione correctamente, es necesario que el tiempo de retardo sea estrictamente menor que el tamaño del buffer; es decir, $d < N$. En una computadora moderna uno puede simplemente elegir un N suficientemente grande para asegurar que esta condición se cumple, pero en arquitecturas limitadas se debe tener cuidado en elegir el N óptimo para la aplicación.

El pseudo-código para implementar el buffer circular (de tamaño N) para una línea de retardo con tiempo de retardo d es muy simple. Suponer que las variables `indiceEscritura` e `indiceLectura` representan las posiciones de escritura y lectura, respectivamente, del buffer circular, y que la posición de lectura se encuentra exactamente d muestras antes de la posición de escritura (módulo N). Esto puede lograrse calculando la posición de lectura como

```
indiceLectura = (indiceEscritura - d + N) % N;
```

Para calcular las muestras de salida, primero se guarda la muestra de entrada en el buffer (en la posición indicada por `indiceEscrtitura`), luego se obtiene la muestra correspondiente a la posición de lectura para asignarla como salida, y finalmente, se incrementan ambos apuntadores (escritura y lectura) para procesar la siguiente muestra. El siguiente pseudo-código ilustra lo anterior:

```
buffer[indiceEscritura] = entrada;
salida = buffer[indiceLectura];
indiceEscritura = (indiceEscritura + 1) % N;
indiceLectura = (indiceLectura + 1) % N;
```

Note que los incrementos y decrementos a los apuntadores del buffer circular se aplican con módulo N.

14.5.1 Linea de retardo simple

Con base en lo anterior, podemos implementar una primer versión de la línea de retardo en su forma mas simple. A continuación se presenta el código de la clase seguido de una explicación de los miembros y métodos:

```
public class RetardoSimple extends UGen {
  float[] buffer;
  int indiceEscritura;
  float indiceLectura;

  public RetardoSimple(AudioContext ac, float tiempoMax) {
    super(ac, 2, 1);
    int tamanio = ceil(context.getSampleRate() * tiempoMax / 1000.0);
    buffer = new float[tamanio];
  }

  void setTiempoMuestras(float tiempo) {
    if (tiempo < 0) tiempo = 0;
    if (tiempo > (buffer.length - 1)) tiempo = buffer.length - 1;
    indiceLectura = (indiceEscritura - tiempo + buffer.length) % buffer.length;
  }

  float getTiempoMuestras() {
    return (indiceEscritura - indiceLectura + buffer.length) % buffer.length;
```

```
  }

  void setTiempo(float ms) {
    setTiempoMuestras(context.getSampleRate() * ms / 1000.0);
  }

  float getTiempo() {
     return getTiempoMuestras() * 1000.0 / context.getSampleRate();
  }

  void reinicia() {
    Arrays.fill(buffer, 0);
  }

  public void calculateBuffer() {
    float[] in = bufIn[0];
    float[] out = bufOut[0];
    if (buffer == null) return;

    for (int i = 0; i < bufferSize; i++) {
      // Almacena muestra de entrada en el buffer
      buffer[indiceEscritura] = in[i];

      // Calcula la muestra de salida interpolando el buffer
      out[i] = buffer[(int)indiceLectura];

      // Actualiza los indices de escritura y lectura
      indiceEscritura = (indiceEscritura + 1) % buffer.length;
      indiceLectura = (indiceLectura + 1) % buffer.length;
    }
  }
}
```

La clase cuenta con tres miembros de datos: el arreglo `buffer`, donde se almacena el buffer circular, y los apuntadores de escritura y lectura (`indiceEscritura` y `indiceLectura`). Enseguida se define el constructor de la clase, el cual llama al constructor de la clase base `UGen` para crear una UGen con dos entradas y una salida y crear el buffer; por ahora no se utilizará la segunda entrada, la cual está pensada para una señal moduladora, pero es importante considerarla desde este momento.

A continuación se define una serie de funciones *set* y *get* para manipular el tiempo de retardo. Los métodos `setTiempoMuestras()` y `getTiempoMuestras()` se utilizan para fijar y obtener el tiempo de retardo dado en muestras; cabe notar que el tiempo de retardo no se almacena como miembro de la clase, sino que se obtiene de manera indirecta mediante la diferencia (módulo N) entre los índices de escritura y lectura. En la mayoría de las aplicaciones es más común

utilizar segundos o milisegundos como unidades de tiempo, por lo cual se definen otros dos métodos `setTiempo()` y `getTiempo()` donde el tiempo se especifica en milisegundos. Además, se incorpora a la clase un método `reinicia` para limpiar el buffer, haciendo que éste "olvide" la señal que tenía almacenada.

Finalmente se tiene el método callback, que simplemente implementa el pseudo-código del buffer circular dentro del ciclo que procesa las muestras de cada bloque. Note que dentro del ciclo se utiliza el valor redondeado a entero del tiempo de retardo (variable `t`), de manera que el apuntador de lectura sea también entero.

Si al alumno lo consume la curiosidad, puede adelantarse a implementar la aplicación de ejemplo para probar la UGen `RetardoSimple` y posteriormente regresar a la siguiente sección para implementar algunas mejoras a la clase, las cuales no son necesarias para la aplicación de ejemplo.

14.5.2 Implementación de filtros FIR

En la mayoría de las aplicaciones de procesamiento (de audio) en tiempo real, los filtros utilizados son filtros de respuesta infinita al impulso (IIR), debido a que pueden implementarse de manera muy eficiente mediante retroalimentación. Sin embargo, existen ocasiones donde un filtro de respuesta finita al impulso (FIR) puede ser mas adecuado. Los filtros FIR tienen varias características que pueden ser ventajosas: siempre son estables, su respuesta en fase es siempre lineal, y son mas fáciles de diseñar si se requiere un filtro muy selectivo en frecuencia. Sin embargo, también presentan algunas desventajas, principalmente en términos del costo computacional y de la dificultad para variar o modular sus parámetros de manera continua (muestra a muestra).

Una línea de retardo simple ya es en sí un filtro FIR, pero también es posible utilizar la línea de retardo para implementar cualquier (en teoría) filtro FIR causal a través de la convolución $y = x * h$. La salida del filtro está definida como

$$y[n] = \sum_{k=0}^{N-1} h_k x[n-k], \tag{14.1}$$

donde los coeficientes $h_k = h[k]$ constituyen la respuesta al impulso del filtro (también llamada *núcleo* o *kernel*), y N es la longitud (en muestras) de la respuesta al impulso. Algunos autores también le llaman a N el *orden* del filtro, pero no debe confundirse con el orden de un filtro IIR, el cual representa el número de polos (los filtros FIR no tienen polos).

En general, aún cuando la señal de entrada x es real, el kernel h puede ser complejo y dar como resultado una salida y también compleja. Dado que la convolución es un operador lineal, podemos ver que

$$y = x * h = x * (h_r + ih_i) = x * h_r + i(x * h_i),$$

donde h_r y h_i son las partes real e imaginaria de h, respectivamente. Por lo tanto, considerando que la señal de entrada es real, podemos implementar la

convolución asumiendo un kernel real y simplemente calcular dos convoluciones reales cuando sea necesario utilizar un kernel complejo.

A manera de ejemplo, se muestra a continuación una UGen derivada de `RetardoSimple` para implementar filtros FIR a partir de una línea de retardo:

```
class FiltroFIR extends RetardoSimple {
  float[] kernel;

  public FiltroFIR(AudioContext ac, int maxKernel) {
    super(ac, 1000 * maxKernel / ac.getSampleRate());
  }

  void setKernel(float[] k) { arrayCopy(kernel, k); }

  public void calculateBuffer() {
    int i, k, n;
    float[] in = bufIn[0];
    float[] out = bufOut[0];
    float suma;
    if (buffer == null || kernel == null) return;

    n = min(buffer.length, kernel.length);

    for (i = 0; i < bufferSize; i++) {
      // Almacena muestra de entrada en el buffer
      buffer[indiceEscritura] = in[i];

      // Calcula convolucion
      suma = 0;
      for (k = 0; k < n; k++) {
        suma += kernel[k] * buffer[(indiceEscritura - k + buffer.length) % buffer.length];
      }

      out[i] = suma;
      indiceEscritura = (indiceEscritura + 1) % buffer.length;
    }
  }
}
```

La nueva clase incorpora como miembro un arreglo `float[] kernel` para almacenar el kernel y un método `setKernel()` para definirlo. El constructor de la clase toma como argumento la longitud máxima del kernel (en muestras), lo cual viene a ser el tamaño del buffer de retardo. Cualquier kernel de mayor longitud será truncado durante el procesamiento de la señal. Finalmente, la función callback `calculateBuffer()` implementa la convolución tomando las muestras de la señal de entrada almacenadas en el buffer, no sin antes almacenar la muestra actual.

Este manual no cubre el diseño y uso de filtros FIR, por lo cual la UGen `FiltroFIR` se presenta solo como un prototipo, pero no se utilizará en las prácticas. El diseño de filtros FIR requiere balancear el costo computacional (determinado por la longitud del kernel), la selectividad en frecuencia (anchura de las bandas de transición) y los artefactos que puedan producirse (rizado en los extremos de las bandas de paso y rechazo).

14.5.3 Tiempos de retardo fraccionarios

Existen aplicaciones en las cuales el tiempo de retardo se calcula de alguna manera a partir de la cual resultan valores fraccionarios (algunas de estas aplicaciones se estudiarán mas adelante). Por supuesto, uno podría simplemente redondear el tiempo de retardo al entero mas cercano (como lo hace la UGen `RetardoSimple`), pero las imprecisiones resultantes pueden conducir a artefactos serios. Por ejemplo, algunos métodos de síntesis basados en modelado físico utilizan retardos "entonados" a la nota que se desea reproducir; es decir, donde el tiempo de retardo debe ser igual al periodo fundamental de la nota. Si se redondea este tiempo a un número entero de muestras, entonces la línea de retardo estará desafinada con respecto a la nota deseada, generando entonces una frecuencia distinta. Esto será aún mas evidente para las notas agudas, cuyo periodo será relativamente pequeño y entonces el redondeo reducirá seriamente el número de dígitos significativos. En estos casos, la diferencia entre la frecuencia deseada y la obtenida podría ser incluso del orden de un semitono.

Una solución consiste en estimar, mediante interpolación, los valores de la señal retardada para tiempos de retardo d no necesariamente enteros. En este caso utilizaremos interpolación lineal, tal como hicimos para el oscilador basado en tablas de ondas. Sin embargo, también es posible utilizar esquemas de interpolación mas sofisticados para mejorar la calidad.

El siguiente pseudo-código ilustra cómo calcular cada muestra de salida mediante interpolación lineal:

```
buffer[indiceEscritura] = entrada;

izq = int(indiceLectura);
frac = indiceLectura - izq;
der = (izq + 1) % N;
salida = (1 - frac) * buffer[izq] + frac * buffer[der];

indiceEscritura = (indiceEscritura + 1) % N
indiceLectura = (indiceLectura + 1) % N;
```

14.5.4 Modulación del tiempo de retardo

El último paso consiste en agregar modulación al tiempo de retardo para obtener una línea de retardo *continuamente variable*. Al igual que en el caso de los osciladores, debe tenerse cuidado de que al variar el tiempo de retardo no se

produzcan cambios abruptos en el apuntador de lectura que resulten en artefactos o discontinuidades en la señal de salida. Por este motivo, la modulación no se aplica directamente al tiempo de retardo (el cual ni siquiera conservamos como miembro de la clase), sino a la manera en que se incrementa el índice de lectura.

Cuando el tiempo de retardo es fijo, el apuntador de lectura se incrementa de la manera siguiente:

```
indiceLectura = (indiceLectura + 1) % N;
```

Lo que haremos es agregar un parámetro g que representa la tasa de crecimiento del tiempo de retardo (en muestras por muestra), y reemplazamos la instrucción anterior por la siguiente:

```
indiceLectura = (indiceLectura + 1 - g + N) % N;
```

Cuando $g = 0$, tendremos nuevamente un retardo de tiempo fijo. Si $g > 0$, entonces el tiempo de retardo crece a razón de g muestras por muestra, mientras que si $g < 0$ el tiempo de retardo disminuye a razón de $|g|$ muestras por muestra (o segundos por segundo - note que g no depende de las unidades de tiempo utilizadas) [18]. Por supuesto, no es posible mantener $g > 0$ (o $g < 0$) por un tiempo indefinido, ya que llegará el momento en que el tiempo de retardo se volverá mayor que la longitud del buffer o el índice de lectura alcanzará al de escritura. Sin embargo, es posible mantener $g \neq 0$ por un cierto tiempo hasta alcanzar el tiempo de retardo deseado, y luego volver a fijar $g = 0$, por ejemplo, mediante una UGen `Rampa`; o bien, hacer que g oscile de manera que su valor promedio a lo largo del tiempo sea cero.

Desde un punto de vista físico se puede pensar que un valor $g > 0$ equivale a un observador que se aleja de la fuente de audio: conforme pasa el tiempo, el sonido tarda cada vez más en llegar al oído del observador. De la misma manera, para un observador que se acerca a la fuente de audio tendríamos $g < 0$. De hecho, se puede estimar g como $g = -v/c$, donde v es la velocidad del observador (hacia la fuente) y c la velocidad del sonido. Es así que una línea de retardo continuamente variable puede utilizarse para simular el efecto Doppler, corrimientos de frecuencia y efectos de coro y vibrato [19].

Para implementar un retardo fraccionario y continuamente variable escribiremos una nueva UGen llamada `RetardoVariable` heredada de `RetardoSimple`. La razón de lo anterior es que un retardo fraccionario y continuamente variable tiene un costo computacional considerablemente mas elevado que el retardo simple, y no todas las aplicaciones lo requieren, por lo que vale la pena implementar ambas clases.

La nueva clase agrega un nuevo miembro: el índice de modulación, mediante el cual se controla la amplitud de la señal moduladora. Además, incluye métodos *set* y *get* para manipular el índice de modulación, conectar la UGen moduladora, y por supuesto, una nueva función callback que integra las mejoras discutidas:

```
public class RetardoVariable extends RetardoSimple {
  float indiceModulacion;
```

```
  public RetardoVariable(AudioContext ac, float tiempoMax) {
    super(ac, tiempoMax);
  }

  float indice() {
    return indiceModulacion;
  }

  void setIndice(float g) {
    indiceModulacion = g;
  }

  void setEntrada(UGen ugen) { addInput(0, ugen, 0); }

  void setModulador(UGen ugen) { addInput(1, ugen, 0); }

  public void calculateBuffer() {
    float[] in = bufIn[0];
    float[] mod = bufIn[1];
    float[] out = bufOut[0];
    float frac, g;
    int izq, der;
    if (buffer == null) return;

    for (int i = 0; i < bufferSize; i++) {
      // Almacena muestra de entrada en el buffer
      buffer[indiceEscritura] = in[i];

      // Calcula la muestra de salida interpolando el buffer
      izq = int(indiceLectura);
      frac = indiceLectura - izq;
      der = (izq + 1) % buffer.length;
      out[i] = (1 - frac) * buffer[izq] + frac * buffer[der];

      // Actualiza los indices de escritura y lectura
      g = indiceModulacion * mod[i];
      indiceEscritura = (indiceEscritura + 1) % buffer.length;
      indiceLectura = (indiceLectura + 1 - g + buffer.length) % buffer.length;
    }
  }
}
```

14.5.5 Aplicación de ejemplo

La aplicación de ejemplo es una modificación sencilla de la aplicación de la práctica anterior, donde únicamente se modifica la arquitectura del sistema propuesto. Por este motivo, solamente modificaremos la sección de inicialización, mientras que las funciones de interfaz y visualización, así como las declaraciones de variable globales, quedan como en la práctica anterior.

La arquitectura anterior consiste en un oscilador de tabla de ondas que pasa por un amplificador variable (VCA), donde la frecuencia del oscilador y la ganancia del amplificador son controladas por envolventes para lograr transiciones suaves. Ahora, agregaremos un retardo simple después del VCA, y mezclaremos la señal del VCA con la del retardo en la salida de audio.

```
// La siguiente variable se agrega a las variables globales ya existentes
RetardoSimple ret;

void setup() {
  size(800, 600);

  ac = new AudioContext();
  vca = new VCA(ac);
  env_amp = new Rampa(ac);
  env_frec = new Rampa(ac);
  osc = new Oscilador(ac, 440);
  osc.setSenoidal(128);
  ret = new RetardoSimple(ac, 5000);

  osc.setModulador(env_frec);
  osc.setIndice(1);
  osc.setFrecuencia(55);

  vca.setEntrada(osc);
  vca.setModulador(env_amp);
  vca.setIndice(0.4);

  ret.addInput(vca);
  ret.setTiempo(1000);

  ac.out.addInput(vca);
  ac.out.addInput(ret);
  ac.start();
}
```

Note que se conecta tanto el VCA como el retardo a la salida física de audio. Beads tiene la ventaja de poder conectar múltiples señales a una misma entrada, sumando automáticamente las señales conectadas. En el caso de Minim y C++, será necesario utilizar una UGen que permita mezclar o combinar dos

o mas señales; por ejemplo, puede utilizar el mezclador propuesto como reto en la Práctica 3 (Sección 3.7). El retardo tiene un tiempo fijo de un segundo, lo que permite distinguir fácilmente entre la señal original (proveniente del VCA) y la señal retardada.

14.6 Evaluación y reporte de resultados

1.- El programa de ejemplo utiliza un retardo con un tiempo muy largo (1 segundo) dando como resultado un efecto de eco. Verifique qué ocurre cuando se utilizan tiempos de retardo relativamente cortos (del orden de decenas de milisegundos). En particular, describa y explique lo que ocurre con la amplitud de la señal de salida conforme varía la frecuencia del oscilador.

2.- Reemplace el retardo fijo por un retardo continuamente variable modulado por un oscilador senoidal de baja frecuencia (alrededor de 4 Hz). Pruebe con distintos valores para el índice de modulación, así como con retardos largos (1 segundo) y cortos (20-50 ms). Describa los resultados.

3.- Considere la línea de retardo continuamente variable donde g es la tasa de cambio del tiempo de retardo (en muestras por muestra, segundos por segundo, o ms por ms). Suponga que se mantiene $g = g_0$ constante durante un tiempo t, y luego se hace inmediatamente $g = 0$. Esto haría que el tiempo de retardo cambiara en $g_0 t$ unidades. Verifique experimentalmente lo anterior utilizando una UGen `Rampa` para generar una señal moduladora que tome el valor g_0 durante un tiempo t y luego se haga cero. Implemente un comando de teclado para disparar la envolvente que modula el tiempo de retardo. Describa las diferencias en la frecuencia del oscilador cuando $g = 0$ y cuando $g = g_0$, y cómo se relaciona esta diferencia con g_0.

14.7 Retos

El efecto que resulta de pasar una señal de audio por una línea de retardo continuamente variable con un tiempo de retardo relativamente corto (≈ 20 ms) el cual es modulado por una onda periódica (e.g., sinusoidal) es conocido como *vibrato*. El objetivo del vibrato es producir ligeras variaciones cuasi-periódicas en el tono, sin que se escuche demasiado artificial (como una sirena de patrulla). Para esto, se suelen usar señales moduladoras con frecuencias alrededor de los 4 Hz, y un índice de modulación relativamente pequeño (≈ 0.02). Es importante en este caso NO combinar la señal original con la señal retardada, ya que solo se desea el efecto de variación en frecuencia, mas no en amplitud. Aplique este efecto a la señal proveniente de la entrada física de audio, utilizando el mouse para controlar tanto la frecuencia de la señal moduladora, como la profundidad del efecto (dada por el índice de modulación).

14.8 Conclusiones

Redacte en el recuadro sus conclusiones acerca de los conceptos aprendidos, las actividades realizadas y los objetivos planteados en esta práctica.

Capítulo 15

Análisis de Fourier

Nombre del estudiante	Calificación

15.1 Relación con los programas de estudio

Materia	Unidad y tema
Procesamiento Digital de Señales (IE / IT / IB)	2.2.- Transformada de Fourier 3.2.- Transformada Discreta de Fourier 3.5.- Transformada Rápida de Fourier
Procesamiento de Señales de Audio (IE)	1.10.- Transformada discreta de Fourier 2.9.- Análisis de Fourier en ventanas de tiempo corto 5.6.- Aplicaciones a las interfaces humano-máquina

15.2 Introducción

Para una señal periódica compleja $x[n] \in \mathbb{C}$ con periodo N, la Transformada Discreta de Fourier (DFT - por sus siglas en inglés) $X[k] = X(\omega_k)$ se define como

$$X[k] = X(\omega_k) = \sum_{n=0}^{N-1} x[n] \exp\{-j2\pi kn/N\}, \tag{15.1}$$

para $k = 0, \ldots, N-1$.

La DFT es una versión discretizada de la Transformada de Fourier continua $X(\omega)$ de $x[n]$, muestreada a intervalos de $2\pi/N$, de manera que $\omega_k = 2\pi k/N$ es

la k-ésima frecuencia muestreada (también llamada k-ésimo bin de frecuencia o k-ésima frecuencia de Fourier).

Inspeccionando la ecuación anterior, es fácil ver que $X[k]$ mide la correlación (con lag cero) entre la señal $x[n]$ y la sinusoidal compleja $\exp\{-j\omega_k n\}$. En otras palabras, $X[k]$ es una medida de la amplitud y fase con la que la sinusoidal con frecuencia ω_k está presente en la señal $x[k]$ [13, 6, 20]. Debido a la periodicidad de $x[n]$, solamente las frecuencias ω_k pueden estar presentes en la señal. En la práctica, $x[n]$ suele ser una señal finita de longitud N, pero siempre debe tenerse en cuenta que el cómputo de la DFT asume que $x[n]$ es periódica.

La DFT tiene un gran número de aplicaciones en el análisis y procesamiento de audio. No solo es posible obtener el espectro de una señal con una alta resolución en frecuencia (aunque sacrificando resolución en tiempo o eficiencia computacional), sino que también permite estimar con una gran precisión la frecuencia fundamental de una señal de entrada monofónica, así como las frecuencias secundarias (harmónicas o inharmónicas), Además, junto con la transformada inversa y algún esquema adecuado de interpolación, es posible realizar otras tareas como filtrado, síntesis aditiva y resíntesis, en tiempo real. Algunas de estas aplicaciones se estudiarán mas adelante.

Dado que pretendemos utilizar la DFT en tiempo real, será necesario calcular la DFT por bloques. Por simplicidad, éstos bloques serán precisamente los buffers que genera el sub-sistema de audio y que se procesan en las funciones callback. Por lo tanto, la resolución en frecuencia del análisis de Fourier estará determinada por el tamaño de buffer del sistema de audio.

Además, existen otras dos cuestiones técnicas importantes que se deben considerar. El primer problema es que la DFT, dada por la ecuación 15.1, tiene una complejidad computacional muy alta (de orden N^2) que limita seriamente su aplicación en tiempo real. La solución consiste en utilizar un algoritmo de Transformada Rápida de Fourier (FFT - por sus siglas en inglés). Tanto Beads como Minim cuentan con implementaciones de la FFT; sin embargo, éstas utilizan un mecanismo distinto a las UGens, y por lo tanto no pueden insertarse de manera simple en una cadena de UGens. Existen también implementaciones muy eficientes en C/C++, como la FFTW (Fastest Fourier Transform in the West) [21]; pero éstas no están disponibles para Java o Processing. Una solución consiste en implementar nuestra propia versión de la FFT que sea lo suficientemente eficiente para las aplicaciones que nos interesan, y lo suficientemente sencilla de implementar y portar a otros lenguajes. Para esto, he optado por un algoritmo de FFT de libro de texto; se trata del algoritmo basado en decimado en frecuencia radix-2 descrito por Oppenheim y Schafer [13]. Este método se ha encapsulado en una clase llamada `FourierAnalysis` que se presenta en el apéndice E, la cual puede incorporarse en cualquier UGen que requiera de la FFT. La única limitación es que se requiere que la longitud de las señales a procesar sea una potencia de 2, por lo que buscaremos que el tamaño de buffer del sistema de audio cumpla con esta restricción.

La segunda cuestión tiene que ver con el hecho de que el audio se dividirá en bloques, y se obtendrá la DFT de cada bloque durante la función callback. Recordemos que al calcular la DFT se hace la suposición de que $x[n]$ es periódica;

sin embargo, al extender $x[n]$ para hacerla periódica (pegando una copia de $x[n]$ tras otra) es muy posible que se introduzcan discontinuidades entre la última muestra de una copia y la primer muestra de la siguiente copia. Estas discontinuidades, a su vez, introducen artefactos en la DFT en forma de rizado, desparramando la energía espectral a lo largo de todas las frecuencias. Una manera de reducir estos artefactos consiste en multiplicar la señal de entrada por una función de ventana $w[n]$ para atenuar la señal en ambos extremos y forzar la continuidad de la extensión periódica de la señal. Existen múltiples funciones de ventana en la literatura. En este caso, utilizaremos la ventana de Blackman, la cual está dada por

$$w[n] = 0.42 - 0.5\cos(2\pi n/(N-1)) + 0.08\cos(4\pi n/(N-1)),$$

para $n = 0, \ldots, N-1$.

Un análisis y discusión sobre la ventana de Blackman y otras funciones de ventana se puede encontrar en [13] y [20].

15.3 Objetivos didácticos

- Conocer la Transformada Discreta de Fourier y sus aplicaciones en el análisis y procesamiento de señales de audio.
- Comprender las cuestiones técnicas que se originan de una implementación de la FFT para procesamiento en tiempo real.
- Implementar una UGen para realizar análisis de Fourier en tiempo real y obtener el espectrograma de una señal.

15.4 Material

- Una computadora con alguna de las siguientes combinaciones de software instalado:
  - Processing y Beads
  - Processing y Minim
  - GNU C++ (e.g., MinGW) y PortAudio
- Bocinas o audífonos estéreo

15.5 Procedimiento

En esta práctica se implementará una UGen para obtener la Transformada Discreta de Fourier de la señal de entrada. Esta UGen servirá como clase base para implementar otras UGens que requieran del análisis de Fourier. Como ejemplo, se implementará una UGen para calcular el espectro de energía de una señal. Finalmente, se desarrollará una aplicación de prueba para mostrar el espectro de una señal en tiempo real.

15.5.1 Cálculo de la DFT

La clase base que se implementará para el análisis de frecuencia se llama `Fourier`, y consiste en una UGen con un canal de entrada y uno de salida. La función callback se encargará de multiplicar la señal de entrada por la ventana de Blackman, calcular la FFT de la señal con ventana y guardarla en dos arreglos llamados `fft_real` y `fft_imag`, ya que FFT da como resultado una señal compleja, y finalmente llamar a *otra* función callback que se implementará en las clases derivadas para realizar distintos tipos de análisis frecuenciales. En la clase base, la función callback únicamente copiará la señal original de entrada en el buffer de salida. De esta manera, podremos insertar la UGen en cualquier parte de la cadena de procesamiento, sin que sea necesario agregar a la UGen como dependiente o utilizar una UGen tipo `Sink`, como se hizo en la Práctica 8.

El código de esta clase es el siguiente:

```
public class Fourier extends UGen {
  float[] fft_real, fft_imag, blackman;
  FourierAnalysis fourier;
  boolean ventana;

  public Fourier(AudioContext ac) {
    super(ac, 1, 1);
    int i, M = ceil(log(bufferSize) / log(2));
    float a;

    fourier = new FourierAnalysis(M);
    fft_real = new float[bufferSize];
    fft_imag = new float[bufferSize];
    blackman = new float[bufferSize];

    // pre-calcula ventana de Blackman
    for (i = 0; i < bufferSize; i++) {
      a = 2 * PI * i / (bufferSize - 1);
      blackman[i] = 0.42 - 0.5 * cos(a) + 0.08 * cos(2 * a);
    }

    ventana = true;
  }

  boolean ventana() { return ventana; }
  void setVentana(boolean v) { ventana = v; }

  public void calculateBuffer() {
    arrayCopy(bufIn[0], fft_real);
    Arrays.fill(fft_imag, 0);
    if (ventana) {
      for (int i = 0; i < bufferSize; i++) { fft_real[i] *= blackman[i]; }
```

```
    }
    fourier.fft(fft_real, fft_imag);
    generaSalida();
  }

  public void generaSalida() {
    arrayCopy(bufIn[0], bufOut[0]);
  }
}
```

La clase tiene como miembros dos arreglos de tipo `float`, llamados `fft_real` y `fft_imag` donde se almacenará la DFT de la señal de entrada, así como un arreglo `blackman` donde se almacenará la ventana de Blackman precalculada (para no tener que calcularla en cada llamada a la función callback). También cuenta con un miembro de clase `FourierAnalysis`, que es la clase que implementa la FFT y se presenta en el Apéndice E. Finalmente, se tiene un miembro booleano llamado `ventana` que determina si se aplicará o no la ventana de Blackman.

Como métodos, la clase tiene un constructor que realiza varias tareas de inicialización: primero crea una UGen con un canal de entrada y uno de salida llamando al constructor de la clase base, luego calcula el logaritmo base 2 del tamaño de bloque para así inicializar el objeto `FourierAnalysis`, así como los arreglos que contendrán la DFT y la ventana de Blackman. Finalmente, se pre-calcula la ventana de Blackman y se activa el uso de ésta por defecto (`ventana = true;`). Se cuenta también con funciones *set* y *get* para activar o desactivar el uso de la función de ventana. Si el alumno lo desea, puede cambiar la variable booleana y las funciones `set` y `get` para seleccionar distintos tipos de ventanas (Hanning, Hamming, Triangular, etc.). Por último se tiene la función callback, la cual copia la señal de entrada en los arreglos `fft_real` y `fft_imag` (este último se inicializa con ceros ya que la señal de entrada es real) y, en su caso, aplica la función de ventana; posteriormente se calcula la FFT y se llama a una nueva función callback llamada `generaSalida()` cuyo propósito es realizar la tarea deseada en el dominio de la frecuencia, y posteriormente calcular la salida de la UGen. Por ahora, la función `generaSalida()` simplemente copia la señal de entrada original en el buffer de salida. La intención de introducir una nueva función callback es que no tengamos que implementar nuevamente todo el proceso en las clases derivadas de `Fourier`.

Note que en el caso de Minim, será necesario heredar la clase `Fourier` de la clase `Buffer` presentada en el Apéndice C, y no directamente de `UGen`, ya que el análisis de Fourier debe realizarse por bloques. Revise la implementación de la clase `Amplimetro` para Minim (Sección 8) para recordar las consideraciones necesarias para esta implementación.

15.5.2 Cálculo del espectro de energía

Para ejemplificar el uso de la clase `Fourier`, implementaremos una subclase con una aplicación específica: calcular el espectro de energía de la señal de entrada, la cual se obtiene a partir de la magnitud de la DFT, multiplicada por un factor de normalización, que por lo general es el inverso de la longitud de la señal. El espectro de una señal real es simétrico, por lo que es suficiente con calcular solamente $N/2$ coeficientes; es decir, hasta la frecuencia de Nyquist. Por la misma razón, la energía de la señal de entrada se reparte equitativamente entre las frecuencias positivas y negativas del espectro; para compensar esto, multiplicaremos por dos el espectro resultante. Más aún, el multiplicar la señal por la ventana de Blackman disminuye también la energía de la señal en aproximadamente un 50%, lo cual compensaremos con una ganancia adicional de 2.

Con base en lo anterior, podemos implementar una UGen `Espectro` heredada de `Fourier` como se muestra a continuación:

```
class Espectro extends Fourier {
  float[] espectro;

  public Espectro(AudioContext ac) {
    super(ac);
    espectro = new float[bufferSize / 2];
  }

  public void generaSalida() {
    float ganancia = (ventana ? 4.0 : 2.0) / bufferSize;
    for (int i = 0; i < bufferSize / 2; i++) {
      espectro[i] = ganancia * sqrt(fft_real[i] * fft_real[i] + fft_imag[i] * fft_imag[i]);
    }
    super.generaSalida();
  }
}
```

Note que la clase `Espectro` es muy compacta, ya que aprovecha la funcionalidad ya implementada en `Fourier`. De hecho, no es necesario implementar la función callback `calculateBuffer()`, sino únicamente la nueva función callback `generaSalida()`, la cual calcula el espectro, lo guarda en el arreglo `espectro`, y luego llama a la función `generaSalida()` de la clase base.

15.5.3 Aplicación de prueba

Para la aplicación de prueba, obtendremos el espectro de un oscilador cuya frecuencia y amplitud serán controladas por el mouse. Esta es básicamente la misma arquitectura que la utilizada en la Práctica 9, pero utilizando el oscilador basado en tabla de ondas y un VCA, en lugar del oscilador sinusoidal y el amplificador de ganancia fija. El espectro se obtendrá de la salida del VCA.

El siguiente código corresponde a las etapas de inicialización y la interfaz con el usuario de la aplicación de prueba:

```
AudioContext ac;
VCA vca;
Oscilador osc;
Espectro fft;

int inicio = 1;
int rango = 7;

void setup() {
  size(800, 600);

  ac = new AudioContext(512);
  vca = new VCA(ac);
  osc = new Oscilador(ac, 440);
  fft = new Espectro(ac);

  vca.setEntrada(osc);
  fft.addInput(vca);
  ac.out.addInput(fft);
  ac.start();
}

void mouseMoved() {
  float oct = inicio + (float)rango * mouseX / width;
  osc.setFrecuencia(55 * pow(2, oct));
  vca.setGanancia(1 - (float)mouseY / height);
}

void keyPressed() {
  if (key >= '1' && key <= '9') rango = key - '0';
  if (keyCode == LEFT) inicio--;
  if (keyCode == RIGHT) inicio++;
  if (key == 'V' || key == 'v') fft.setVentana(!fft.ventana());
}
```

Cabe notar que ahora en el constructor de `AudioContext` se especifica un tamaño de bloque de 512 muestras, para asegurar que éste sea una potencia de dos. Además, se ha agregado un comando de teclado (letra 'V') para activar o desactivar el uso de la ventana de Blackman en la estimación del espectro.

Finalmente, se implementará la función `draw()` para visualizar tanto la señal generada como su espectro:

```
void draw() {
  background(0);
```

```
  pushMatrix();
  translate(0, -height / 4);
  stroke(64);
  line(0, height / 2, width, height / 2);
  stroke(0, 255, 0);
  simplescopio(ac.out.getOutBuffer(0));
  translate(0, height / 2);
  stroke(64);
  line(0, height / 2, width, height / 2);
  stroke(0, 255, 255);
  simplescopio(fft.espectro);
  popMatrix();

  // despliega parámetros
  int y = height - 100;
  text("Rango (1-9) = " + rango + " octavas", 10, y + 10);
  text("Octava inicial (</>) = " + inicio, 10, y + 20);
  text("Frecuencia = " + osc.frecuencia(), 10, y + 30);
  text("Análisis FFT", 10, y + 50);
  text("Ventana: " + (fft.ventana() ? "Blackman" : "Rectangular"), 10, y + 60);
}
```

Dado que la señal generada es una senoidal, se espera que el espectro muestre solamente un pico centrado alrededor de la frecuencia del oscilador.

15.6 Evaluación y reporte de resultados

1.- Agregue etiquetas a lo largo del eje X del espectro cada 1000 Hz (para mayor legibilidad, se aconseja que las etiquetas sean "1K", "2K", etc., en lugar de "1000", "2000", etc.). Apóyese en las etiquetas para verificar que el pico en el espectro se observa en la frecuencia correcta.

2.- Observe los espectros de otras formas de onda distinta a la senoidal. Puede obtener distintas formas de onda mediante los métodos que haya agregado a la clase `Oscilador` para generar otras formas, o bien reemplazando el VCA con distintos waveshapers. Describa lo que ocurre en el espectro para una frecuencia f del oscilador (pruebe, por ejemplo, fijando $f = 1000$ Hz). Observe qué ocurre conforme incrementa la frecuencia del oscilador.

3.- Qué pasa con el espectro cuando se desactiva el uso de la ventana de Blackman; es decir, cuando se reemplaza la ventana de Blackman por una ventana rectangular?

4.- Modifique el programa de prueba para visualizar el espectro de la señal proveniente de la entrada física de audio. Tome en cuenta que esta señal es estéreo, por lo que habría que visualizar el espectro de cada canal por separado, o del promedio de ambos canales.

15.7 Retos

Modifique la clase `Fourier` para que la variable `ventana` en lugar de ser booleana sea un selector del tipo de ventana, e incorpore la funcionalidad necesaria en la función `setVentana()` para crear distintos tipos de ventana, incluidas Rectangular, Triangular (Bartlett), Hann, Hamming y Blackman.

15.8 Conclusiones

Redacte en el recuadro sus conclusiones acerca de los conceptos aprendidos, las actividades realizadas y los objetivos planteados en esta práctica. Si lo desea, puede considerar las preguntas de apoyo que se muestran abajo.

Preguntas de apoyo

- Qué aplicaciones tiene el análisis de Fourier de señales de audio?
- Cuáles son las dificultades técnicas que se presentan al implementar análisis de Fourier en tiempo real?

Capítulo 16

Estimación de frecuencia

16.1 Relación con los programas de estudio

Materia	Unidad y tema
Procesamiento Digital de Señales (IE / IT / IB)	1.7.- Correlación y autocorrelación 3.2.- Transformada Discreta de Fourier 3.5.- Transformada Rápida de Fourier 6.1.- Principio de Incertidumbre de Heisenberg 6.3.- Diseño de filtros mediante enventanado
Procesamiento de Señales de Audio (IE)	1.3.- Medidas y unidades de amplitud y frecuencia 1.10.- Transformada discreta de Fourier 2.9.- Análisis de Fourier en ventanas de tiempo corto 5.4.- Detección de tonos 5.6.- Aplicaciones a las interfaces humano-máquina

16.2 Introducción

Como se ha mencionado anteriormente, una de las características principales de un sonido es su altura o tono; es decir, qué tan grave o agudo es el sonido. Esta característica está directamente relacionada con la frecuencia fundamental de la señal que representa el sonido, y de manera mas general con el contenido espectral del mismo.

Un problema interesante consiste en estimar la frecuencia fundamental de

una señal de audio en tiempo real. Existen una gran variedad de métodos para realizar esta tarea, cada uno con sus ventajas y desventajas. De manera muy general, estos métodos pueden clasificarse en tres tipos, los que trabajan en el dominio del tiempo, los que trabajan en el dominio de la frecuencia, y los que combinan ambos dominios. Hasta ahora, no existe un algoritmo que funcione bien para todos los casos [22].

Por lo general, los métodos de estimación de frecuencia fundamental asumen que la señal de entrada es armónica; es decir, que su espectro está muy concentrado en la frecuencia fundamental y en frecuencias que son múltiplos de la fundamental. Además, se asume también que la señal de entrada es monofónica; es decir, que consta de un solo tono que se mantiene constante durante un cierto tiempo. Encontrar la frecuencia fundamental de una señal con alto contenido inarmónico y/o ruido, o más aún, encontrar las múltiples frecuencias fundamentales en una señal polifónica, es una tarea complicada que va mas allá de los alcances de este manual.

En esta práctica se implementarán dos algoritmos simples para la estimación de frecuencia: uno en el dominio del tiempo y otro en frecuencia. Aún cuando éstos no son muy sofisticados, funcionan bien para señales simples en las que la mayor parte de la energía está concentrada alrededor de la frecuencia fundamental.

La estimación de frecuencia tiene aplicaciones en áreas como la detección de emociones en el habla, análisis de vibraciones, monitoreo de señales, y aplicaciones musicales como afinadores y correctores de tono.

16.2.1 Método 1: Conteo de cruces por cero

Consideremos una señal armónica que conste únicamente de la frecuencia fundamental. Por supuesto, la forma de onda de esta señal será una senoidal, como se muestra en la Figura 16.1. Claramente, esta señal cruza el eje horizontal dos veces por cada periodo de la señal. Si la señal tiene una frecuencia de f ciclos por segundo (Hertz), entonces durante un intervalo de tiempo Δt ocurrirán exactamente $f\Delta t$ periodos y $2f\Delta t$ cruces por cero [23].

Por lo tanto, si logramos contar el número de cruces n_c a lo largo de un intervalo de tiempo Δt, entonces podremos estimar la frecuencia de la señal como

$$f \approx \frac{n_c}{2\Delta t}.$$

Para contar el número de cruces por cero n_c podemos simplemente detectar cambios de signo entre muestras consecutivas en un intervalo de tiempo predefinido. Este intervalo de tiempo debe ser lo suficientemente grande como para contener al menos unos cuantos ciclos de la frecuencia mas baja de interés, digamos 20 Hz. Ya que el periodo correspondiente a una onda de 20 Hz es de 50 ms, entonces se sugiere que el intervalo de tiempo requerido para estimar la frecuencia sea de por lo menos 100 ms. Este intervalo es mayor que el tamaño de bloque de procesamiento del sistema de audio, por lo que será necesario mantener la cuenta a lo largo de múltiples bloques; es decir, a lo largo de múltiples

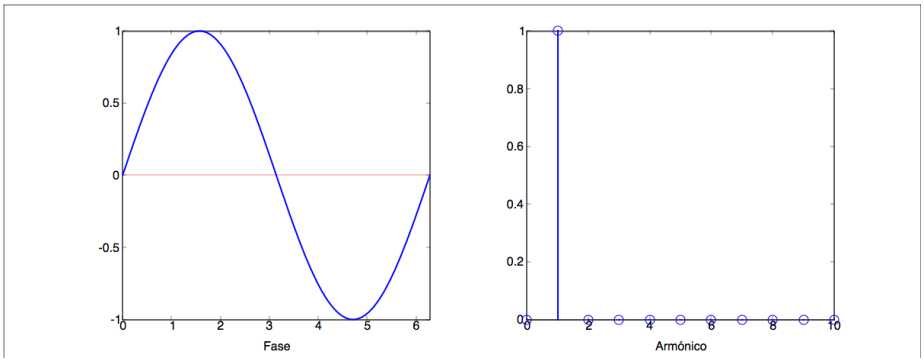


Figura 16.1: Señal armónica que consiste únicamente de la frecuencia fundamental. El panel izquierdo muestra la forma de onda correspondiente (en este caso, una senoidal), mientras que el panel derecho muestra la contribución o peso de cada armónico (donde el primer armónico corresponde a la frecuencia fundamental).

llamadas consecutivas a la función callback. El siguiente pseudo-algoritmo ilustra una manera de hacerlo:

```
for (i = 1; i < bufferSize; i++) {
     if (in[i] * in[i-1] < 0) cuenta_cruces++;
     cuenta_muestras++;

     if (cuenta_muestras == intervalo * getSampleRate()) {
        frecuencia = cuenta_cruces / (2 * intervalo);
        cuenta_cruces = 0;
        cuenta_muestras = 0;
     }
}
```

donde `in[]` es el buffer de entrada de tamaño `bufferSize`, `intervalo` es el intervalo de tiempo Δt (en segundos) sobre el cual se estima la frecuencia y `frecuencia` es la frecuencia estimada. Las variables `cuenta_cruces` y `cuenta_muestras` son contadores cuyo valor debe mantenerse entre llamadas a la función callback, por lo que lo mas sencillo es declararlos como miembros de la UGen e inicializarlos a cero.

Otra alternativa es forzar a que el intervalo sea un múltiplo del tamaño de bloque. De esa manera se puede optimizar la función callback moviendo la segunda sentencia `if` fuera del ciclo que procesa cada muestra.

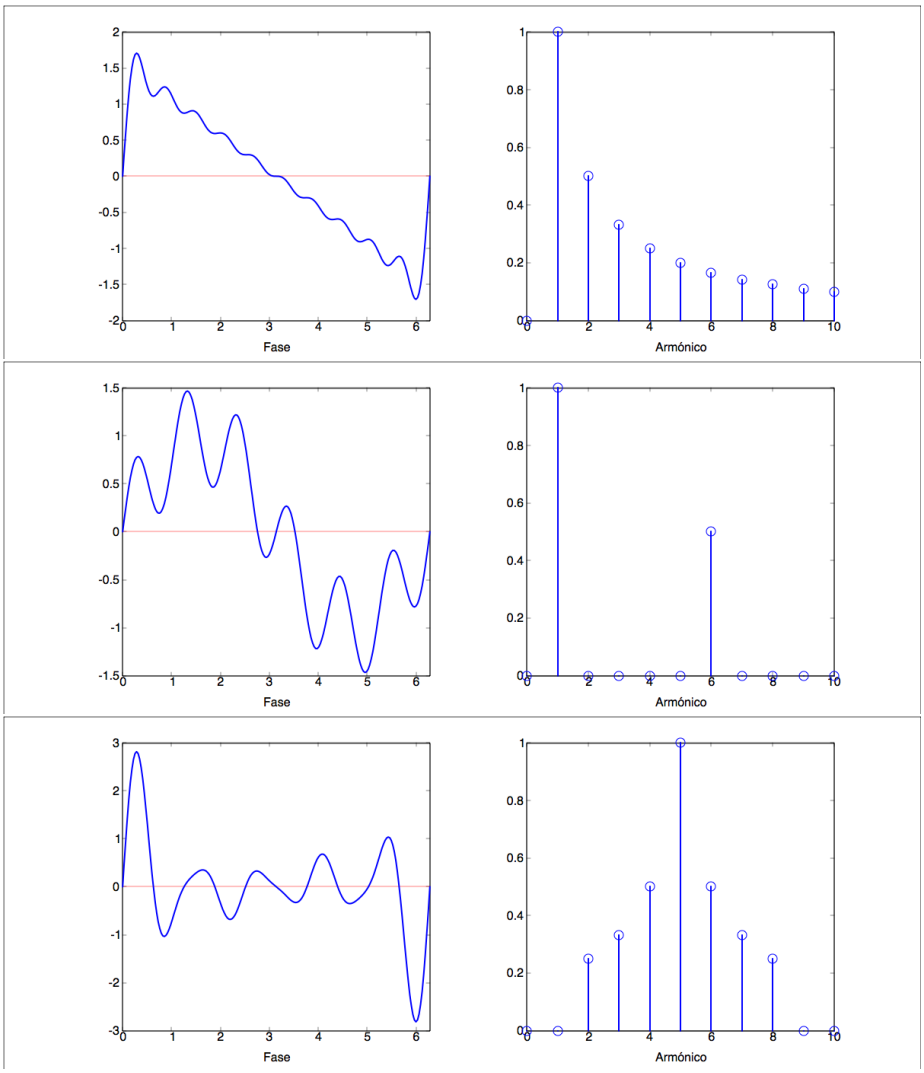


Figura 16.2: Ejemplos de señal armónica con distintas distribuciones de pesos. Renglón superior: aproximación a una diente de sierra con diez armónicos. Renglón intermedio: alta contribución del sexto armónico. Renglón inferior: contribución nula de la fundamental.

16.2.2 Limitaciones del método de conteo de cruces por cero

Lo anterior funciona bien para señales armónicas cuyo espectro está concentrado alrededor del primer armónico (la frecuencia fundamental), ya que conforme los armónicos superiores tienen mayor energía, la forma de onda presentará un mayor número de ondulaciones rápidas que podrían generar cruces adicionales por cero, ocasionando que la frecuencia estimada sea un múltiplo de la fundamental. En cualquier caso, el número de cruces por cero es siempre par. La Figura 16.2 muestra algunos ejemplos de ondas con mayor contenido armónico. En el primer caso (renglón superior), se muestra una aproximación a una onda diente de sierra con diez armónicos, para la cual se observan dos cruces por cero, por lo que la frecuencia fundamental sería estimada correctamente. El segundo ejemplo (renglón intermedio) muestra una onda donde el sexto armónico tiene un peso considerable, generando dos cruces por cero adicionales, lo cual ocasionará que la frecuencia estimada sea el doble de la fundamental. Finalmente, el tercer ejemplo (renglón inferior) muestra un caso mas extremo donde el primer armónico tiene una contribución nula, y solamente existen los armónicos superiores; esto produce un gran número de cruces por cero (diez por periodo), por lo que la frecuencia estimada sería cinco veces la fundamental.

El problema es aún mayor cuando existe ruido, ya que éste puede inducir cualquier número de cruces por cero (incluso un número impar) y no se repite de un bloque de procesamiento a otro, lo cual genera inestabilidad en la estimación.

16.2.3 Método 2: Centroide espectral

El segundo método que se propone se basa en encontrar el pico máximo en el espectro de la señal y estimar el centro del pico mediante algún método de interpolación. Esto nuevamente supone que la señal de entrada es armónica y tiene su energía concentrada alrededor de la frecuencia fundamental; sin embargo, es suficiente con que la contribución del primer armónico sea mayor que la de cualquier otro armónico, por lo que este método es algo mas robusto que el conteo de cruces por cero. Por ejemplo, este método es capaz de detectar correctamente la frecuencia fundamental de la onda correspondiente al segundo caso de la Figura 16.2 (renglón central), pero tendría las mismas dificultades con el tercer caso (renglón inferior).

El primer paso de este método consiste en estimar el espectro de energía de la señal. Esto puede hacerse mediante a través de la UGen `Espectro` desarrollada en la Práctica 15. Una vez hecho esto, localizaremos la posición p del máximo, la cual está dada por:

$$p = \arg \max_{k=0,\ldots,N/2-1} \{|X[k]|\},$$

donde $X[k]$ es la Transformada Discreta de Fourier del buffer de entrada y N es el tamaño de buffer.

Para una frecuencia de muestreo f_S, la frecuencia (en Hertz) que corresponde al bin indexado por p está dada por:

$$f = pf_s/N;$$

sin embargo, esta frecuencia es una estimación muy burda de la frecuencia fundamental, ya que si la frecuencia fundamental no coincide exactamente con una de las frecuencias de Fourier, entonces su energía se esparcirá a lo largo de varios bins consecutivos, alrededor de p.

Entonces, una manera de mejorar la estimación consiste en tomar en cuenta un rango de bins alrededor de p y utilizar la información de estos bins en la estimación. Por ejemplo, uno podría estimar la frecuencia como el centroide del espectro en una ventana centrada en p, el cual está dado por

$$f \approx \frac{f_s}{N} \cdot \frac{\sum_{k=p-w}^{p+w} k|X[k]|}{\sum_{k=p-w}^{p+w} |X[k]|},$$

donde w es el semi-tamaño de la ventana. Para esta aplicación, se obtienen buenos resultados con valores de w entre 1 y 5.

16.2.4 Otras alternativas

Algunos métodos más sofisticados permiten estimar la frecuencia fundamental de una señal armónica aún cuando su espectro no se concentra en el primer armónico. Mencionaremos brevemente tres de ellos.

Autocorrelación: La autocorrelación permite detectar tendencias cíclicas en una señal y estimar el periodo con el que éstas tendencias ocurren. Esta se puede calcular en el dominio del tiempo mediante la ecuación

$$R[d] = \frac{1}{N-d} \sum_{n=d}^{N-1} x[n]x[n-d],$$

donde d es el retardo o *lag* entre la señal original y una copia retrasada de la misma. Por lo general, se estima $R[d]$ para múltiples valores de $d > 0$ y se localiza el valor de d que maximiza $R[d]$. Este valor corresponde al periodo (en muestras) de la señal cuya frecuencia se desea estimar. Este método es computacionalmente demandante ya que se requiere que N sea lo suficientemente grande como para abarcar por lo menos dos periodos a la menor frecuencia de interés, y su complejidad es de orden N^2. También es posible estimar de manera mas eficiente la autocorrelación en el dominio de la frecuencia.

Distancia mínima entre armónicos: En lugar de estimar únicamente la posición del pico mas prominente en el espectro, se estima la posición de múltiples picos, y luego se utiliza esta información para calcular las distancias entre picos consecutivos, las cuales deben ser múltiplos de la frecuencia fundamental. La menor de estas distancias es un buen estimador de la frecuencia

fundamental. También es posible estimar algún estadístico utilizando la información de todas las distancias para tener una estimación mas robusta, por ejemplo, la mediana.

Cepstrum: El espectro de una onda armónica muestra una serie de picos equiespaciados, los cuales pueden interpretarse como otra señal periódica, cuyo periodo es precisamente la frecuencia fundamental. Por lo tanto, si se calcula el espectro de esta nueva señal (es decir, el espectro del espectro - aunque en la práctica se calcula el espectro del log-espectro), se observará un pico localizado en el periodo fundamental de la señal original. Esta técnica funciona mejor con señales con alto contenido armónico.

16.3 Objetivos didácticos

- Conocer el concepto de estimación de frecuencia fundamental así como sus aplicaciones.
- Comprender algunos de los métodos básicos para la estimación de frecuencia fundamental y sus limitaciones.
- Implementar y probar Unidades Generadoras para la estimación de frecuencia fundamental.

16.4 Material

- Una computadora con alguna de las siguientes combinaciones de software instalado:
  - Processing y Beads
  - Processing y Minim
  - GNU C++ (e.g., MinGW) y PortAudio
- Bocinas o audífonos estéreo

16.5 Procedimiento

Para esta práctica se implementarán dos UGens para la estimación de la frecuencia fundamental. La primera de ellas está basada en el conteo de cruces por cero y su código es el siguiente:

```
class CrucesPorCero extends UGen {
  float intervalo;
  int intervalo_muestras;
  int cuenta;
```

```
  int cuenta_cruces;
  int cuenta_muestras;

  public CrucesPorCero(AudioContext ac, float i) {
    super(ac, 1, 1);
    setIntervalo(i);
    cuenta = 0;
  }

  void setIntervalo(float i) {
    intervalo = i;
    intervalo_muestras = round(i * context.getSampleRate() / 1000.0);
    cuenta_cruces = 0;
    cuenta_muestras = 0;
  }

  float intervalo() { return intervalo; }

  int cuenta() { return cuenta; }

  float frecuencia() { return cuenta * 1000.0 / intervalo / 2; }

  public void calculateBuffer() {
    float[] in = bufIn[0];
    for (int i = 1; i < bufferSize; i++) {
       if (in[i] * in[i - 1] < 0) cuenta_cruces++;
       cuenta_muestras++;
       if (cuenta_muestras == intervalo_muestras) {
         cuenta = cuenta_cruces;
         cuenta_cruces = 0;
         cuenta_muestras = 0;
       }
    }
    arrayCopy(in, bufOut[0]);
  }
}
```

El intervalo de tiempo, dado en milisegundos, se especifica en el constructor, y puede cambiarse posteriormente mediante el método `setIntervalo()`. La UGen cuenta también con métodos *get* para obtener el intervalo, la cuenta de cruces por cero mas reciente, y la frecuencia estimada. Finalmente, la función callback `calculateBuffer()` implementa el pseudo-código propuesto anteriormente y copia la señal de entrada a la salida, de manera que la UGen pueda insertarse en cualquier punto de la cadena de procesamiento.

El segundo método a implementar es el cálculo del centroide alrededor del pico máximo del espectro. Para esto, aprovecharemos la UGen `Espectro` im-

plementada en la Práctica 15, heredando de ella la nueva UGen:

```
class Frecuenciometro extends Espectro {
  float frecuencia;
  int w;

  public Frecuenciometro(AudioContext ac, int _w) {
    super(ac);
    w = _w;
  }

  public Frecuenciometro(AudioContext ac) {
    super(ac);
    w = 3;
  }

  float frecuencia() { return frecuencia; }

  public void generaSalida() {
    int i, a, b, pmax = 0, n = bufferSize / 2;
    float num = 0, den = 0, max = 0;

    super.generaSalida();

    // detecta pico maximo
    for (i = 0; i < n; i++) {
      if (espectro[i] > max) { max = espectro[i]; pmax = i; }
    }

    a = pmax - w; if (a < 0) a = 0;
    b = pmax + w; if (b > n-1) b = n - 1;

    for (i = a; i <= b; i++) {
      num += (float)i * espectro[i];
      den += espectro[i];
    }

    frecuencia = (den > 0) ? (context.getSampleRate() * num / den / bufferSize) : 0;
  }
}
```

Esta clase cuenta con dos constructores, uno de los cuales permite establecer el tamaño de la ventana utilizada para calcular el centroide, y otro que simplemente lo fija a $w = 3$. Si el alumno lo desea, puede agregar funciones *set* y *get* para manipular este parámetro, aunque en la práctica no suele ser necesario. También se tiene una función *get* para obtener el valor mas reciente de la frecuencia estimada. Por último, ya que esta clase se deriva de `Espectro`, la cual

a su vez se deriva de `Fourier`, no es necesario implementar la función callback original `calculateBuffer()`, sino la función callback `generaSalida()`, la cual se llama después de haber calculado la FFT de la señal de entrada. Para este caso, la función `generaSalida()` llama primero a la misma función para la clase base, en la cual se calcula el espectro, y posteriormente encuentra la posición del pico principal del espectro y su centroide, para finalmente estimar la frecuencia correspondiente a este pico.

16.5.1 Aplicación de prueba

Una vez implementados los detectores de frecuencia, el alumno debe desarrollar una pequeña aplicación de prueba. Es posible utilizar ambos detectores simultáneamente al conectarlos en cascada, ya que cada uno de ellos deja pasar la señal de entrada intacta. Posiblemente lo mas sencillo es tomar alguna de los simuladores de theremin elaborados en las prácticas anteriores, pero también sería interesante utilizar la entrada de audio, por ejemplo para conectar algún instrumento musical. Se puede utilizar también la voz, a través del micrófono, pero ésta tiene un espectro mas complejo.

16.6 Evaluación y reporte de resultados

1.- Utilice un oscilador entonado a una frecuencia conocida como entrada para los detectores de frecuencia. Compare la frecuencia estimada por los detectores con la frecuencia real del oscilador y mida el error de estimación (la diferencia absoluta entre la frecuencia real y la estimada, normalizada por la frecuencia real). Cuál de los detectores tiene mayor exactitud? Cómo se comporta el error en frecuencias bajas, medias y altas?

2.- Siguiendo la línea de pruebas de la pregunta anterior, compare el error del estimador basado en el centroide espectral utilizando diferentes ventana (e.g., Rectangular vs. Blackman). Si implementó otras funciones de ventana, pruébelas también. Con cuál ventana se obtiene mayor exactitud?

3.- Un problema que se tiene con los dos detectores de frecuencia propuestos es la presencia de una componente de DC que sea lo suficientemente grande como para que toda la señal sea positiva (o toda negativa), lo cual eliminaría los cruces por cero y produciría un pico espectral de gran magnitud en la frecuencia cero. Esto ocasionaría que ambos métodos fallaran en la estimación de frecuencia. Proponga una solución sencilla a este problema y pruébela utilizando un oscilador de tabla de ondas donde los valores de la tabla de onda sean todos positivos; por ejemplo, $t[i] = 0.5 + 0.5 * \text{sen}(2\pi i/N)$, donde N es el tamaño de la tabla.

16.7 Retos

Utilice el detectores de frecuencia de mayor precisión implementado en esta práctica para desarrollar una afinador cromático para instrumento. La aplicación debe detectar la frecuencia de la señal proveniente de la entrada física de audio, convertirla a semitonos e indicar visualmente qué tan lejos se está del semitono mas cercano.

16.8 Conclusiones

Redacte en el recuadro sus conclusiones acerca de los conceptos aprendidos, las actividades realizadas y los objetivos planteados en esta práctica. Si lo desea, puede considerar las preguntas de apoyo que se muestran abajo.

Preguntas de apoyo

- Qué es un sonido armónico y cómo se define la frecuencia fundamental?
- Qué técnicas existen para la detección de la frecuencia fundamental?
- Qué dificultades o limitaciones se presentan en la detección de la frecuencia fundamental?

Parte III

Prácticas de nivel avanzado

Introducción a las prácticas de nivel avanzado

A lo largo de las prácticas de niveles básico e intermedio, hemos desarrollado muchos de los bloques básicos en los que se basan los procesos de audio más interesantes. Estos bloques (UGens) pueden dividirse en las siguientes categorías:

- **Generadores:** `RuidoBlanco`, `Senoidal`, `Oscilador`, `Rampa`
- **Filtros:** `Filtro1P`, `FiltroPBR`, `EQShelving`, `EQPeaking`
- **Waveshapers:** `Amplificador`, `VCA`, `Clipper`, `Folder`, `Shaper`, `Crusher`
- **Retardos:** `RetardoSimple`, `RetardoVariable`
- **Análisis:** `Amplimetro`, `Fourier`, `Espectro`, `CrucesPorCero`, `Frecuenciometro`

Además, es posible que el alumno haya implementado algunas UGens adicionales, como el sumador `Suma2` y el mezclador que se proponen al final de la Práctica 3 (llamémosle `Mezclador`), algunos filtros adicionales (e.g., un rechaza-banda o el filtro FIR), osciladores adicionales, o waveshapers adicionales. No sería mala idea organizar todas estas clases en una librería con múltiples archivos fuente (y tener un par de respaldos).

Pasaremos ahora a las prácticas de nivel avanzado, las cuales nos acercan aún mas a las aplicaciones reales. Algunas de estas prácticas podrán implementarse combinando las UGens que ya se tienen, mientras que otras prácticas requerirán la creación de nuevas UGens, ya sea modificando y/o heredando las UGens existentes. En cualquier caso, nos apoyaremos mucho en el trabajo ya realizado, por lo que ya no será necesario incluir el código completo de los programas elaborados. Se incluirá pseudo-código, y el código que sea crítico para la implementación de nuevas UGens, pero en la mayoría de los casos se omite el código de las aplicaciones de prueba.

Otra novedad para estas prácticas será la inclusión de interfaces de usuario más elaboradas. Esto será necesario ya que los sistemas que implementaremos serán mas elaborados y contarán con un mayor número de parámetros. Para no complicar demasiado las cosas, se recomienda el uso de una librería externa de Processing para agregar elementos de GUI (Graphical User Interface) como deslizadores, selectores de opciones y botones (por ejemplo, ControlP5 o Interfascia). Esto, nuevamente, con la intención de acercarnos más al desarrollo de una aplicación completa. Los usuarios de C++ tienen disponibles un gran número de librerías y herramientas para elaborar GUIs, pero algunas de éstas dependen del sistema operativo, por lo que la elección se deja libre al alumno.

Contenido

1. Síntesis aditiva
2. Síntesis mediante modulación de frecuencia

3. Síntesis sustractiva
4. Retroalimentación: Parte II
5. Efectos basados en retardos
6. Síntesis por modelado físico
7. Filtros pasa-todo y phasers
8. Reverberación artificial

Capítulo 17

Síntesis aditiva

Nombre del estudiante	Calificación

17.1 Relación con los programas de estudio

Materia	Unidad y tema
Procesamiento Digital de Señales (IE / IT / IB)	2.3 - Propiedades de la Transformada de Fourier 3.1.- Periodicidad en tiempo discreto 3.2.- Transformada discreta de Fourier 3.4.- Propiedades de la Transformada Discreta de Fourier 4.1.- Muestreo de señales en tiempo contínuo 4.2.- Teorema de muestreo de Nyquist 4.3.- Reconstrucción de señales de banda limitada
Procesamiento de Señales de Audio (IE)	2.6.- Tablas de ondas 3.3 - Síntesis aditiva 5.5.- Aplicaciones musicales en arquitecturas diversas 5.6.- Aplicaciones a las interfaces hombre-máquina 5.7.- Representación auditiva de datos científicos

17.2 Introducción

En la actualidad existe un gran número de técnicas de síntesis de sonido, la mayoría de las cuales son implementadas digitalmente. Estas técnicas tienen como objetivo tanto simular sonidos que se producen de manera acústica o eléctrica, como crear sonidos completamente artificiales.

Las técnicas de síntesis proporcionan diversas maneras de modelar el espectro de una señal, con sus respectivas variaciones a lo largo del tiempo, permitiendo así manipular el timbre del sonido resultante. Dada la relación entre timbre y espectro, una de las primeras técnicas de síntesis que fueron estudiadas es la llamada *síntesis aditiva*, la cual no es mas que una aplicación directa del teorema de Fourier, el cual establece, a muy grandes rasgos, que cualquier función periódica puede descomponerse como una suma de un cierto número (posiblemente infinito) de funciones oscilatorias simples, ya sea senos y cosenos o exponenciales complejas, con distintas frecuencias. El espectro de una señal no es mas que una representación de la contribución de cada una de las componentes sinusoidales (también llamadas *parciales*) en función de su frecuencia.

En la Práctica 12 se mencionó que el espectro de un sonido puede clasificarse como armónico, inarmónico o ruido. Un espectro armónico está compuesto de una suma discreta de parciales equiespaciadas en frecuencia; específicamente en múltiplos de la frecuencia fundamental. Matemáticamente, el espectro de una señal armónica con frecuencia fundamental f se representa como sigue:

$$x(t) = \sum_{k=0}^{\infty} X_k(t) \exp\{j2\pi k f t\},$$

donde f_m es la frecuencia de muestreo y $X_k(t) \in \mathbb{C},\ k = 0, \ldots$ son los coeficientes complejos que definen la amplitud y fase de cada una de las parciales o armónicos, y que posiblemente varían a lo largo del tiempo (de ahí la dependencia con respecto a t). La componente correspondiente a $k = 0$ representa el DC de la señal, y por lo general tiene amplitud nula (i.e., $X_0 = 0$).

En la práctica, uno evita los cálculos con números complejos al utilizar solamente funciones seno o coseno en lugar de exponenciales complejas. Por otra parte, la fase de las parciales tiene un efecto importante en la forma de onda de la señal resultante; sin embargo, el oído es poco sensible a la fase de las componentes. En efecto, al sumar dos sinusoidales con distinta frecuencia, la forma de onda cambia drásticamente al variar la fase de uno o ambos osciladores, pero no cambia el timbre que se percibe. Por este motivo, podemos considerar que, para efectos audibles, todas las parciales corresponden a funciones seno (o bien, todas son cosenos). Finalmente, desde un punto de vista práctico podemos considerar también que el número de parciales es finito, ya que para algún entero K ocurrirá que las frecuencias mayores a Kf son inaudibles, o en el caso de las señales en tiempo discreto, se alcanza la frecuencia de Nyquist.

A partir de las consideraciones anteriores, se pueden proponer los siguientes modelos para sintetizar una forma de onda armónica (periódica) con frecuencia

fundamental f en tiempo discreto:

$$x[n] = \sum_{k=1}^{K} a_k[n] \cos(2\pi k f n / f_m), \qquad x[n] = \sum_{k=1}^{K} a_k[n] \operatorname{sen}(2\pi k f n / f_m), \tag{17.1}$$

donde K es el número de parciales a sumar y $a_k[n] \in \mathbb{R},\ k = 1, \ldots, K$ son las contribuciones o pesos de las respectivas parciales. Dados los mismos parámetros a_k, cualquiera de los dos modelos genera aproximadamente el mismo timbre, pero con una forma de onda distinta.

17.2.1 Aplicación directa y mediante tablas de ondas

Es posible implementar un sintetizador aditivo aplicando directamente la Ecuación 17.1 (cualquiera de ellas). En este caso, se requiere contar con un arreglo de K osciladores sinusoidales, para un valor de K que por lo general es fijo pero suficientemente grande. Es común utilizar desde $K = 16$ hasta $K = 128$ parciales. Dado que es necesario controlar o modular individualmente la amplitud de cada oscilador, se requiere también un arreglo de K amplificadores o VCAs, cada uno de ellos conectado a la salida de un oscilador, y finalmente un bloque mezclador que combine la salida de los amplificadores. En caso de utilizar amplificadores de ganancia fija, el sonido resultante tendrá un timbre estático mientras las ganancias no se modifiquen. Para generar timbres que varíen a lo largo del tiempo, se utilizan VCAs y una envolvente por cada parcial para modular su amplitud; es decir, K envolventes en total. Claramente, aún para un número pequeño de parciales K, la implementación de un sintetizador aditivo requiere un gran número de bloques y requiere fijar un gran número de parámetros. Por este motivo, los sintetizadores aditivos comerciales son poco comunes y por lo general difíciles de programar[1].

Otra alternativa, que es mas utilizada en la práctica, consiste en utilizar síntesis aditiva para generar un solo periodo de una onda periódica, el cual se almacenará en una tabla de ondas. Si se cuenta con un oscilador que permita interpolar entre dos o más tablas de ondas (ver el reto propuesto al final de la Práctica 13), entonces se obtiene un resultado similar al de múltiples envolventes modulando las amplitudes de las parciales. Dados los pesos $a_k,\ k = 1, \ldots, K$ de las parciales, se puede sintetizar un periodo de una onda mediante alguna de las siguientes ecuaciones:

$$w[n] = \sum_{k=1}^{K} a_k \cos(2\pi k n / N), \qquad w[n] = \sum_{k=1}^{K} a_k \operatorname{sen}(2\pi k n / N), \tag{17.2}$$

donde N es el tamaño de la tabla de onda $w[n]$. Por lo general se puede asumir que N es mucho mas grande que el número de parciales K. Por ejemplo, para tener una buena resolución a bajas frecuencias (cercanas a 20 Hz) con una frecuencia de muestreo f_m, se puede elegir un tamaño de tabla alrededor de $f_m/20$. Si se elige un tamaño que sea potencia de dos, las opciones son $N = 2048$ ó $N = 4096$, mientras que el número de parciales K podría ser 32, 64 o 128.

[1] Al acto de manipular los parámetros de un sintetizador para modelar un sonido específico, se le conoce como *programar* el sintetizador.

Se puede demostrar que al utilizar una suma de cosenos, la forma de onda resultante será simétrica; es decir que $w[n] = w[N-n]$. Por otra parte, una suma de senos da como resultado una forma de onda *antisimétrica*, donde $w[n] = -w[N-n]$. Cualquier forma de onda arbitraria (discretizada a N muestras) se puede obtener como la suma de una parte simétrica y una antisimétrica [13, 10].

También es posible demostrar que si una forma de onda cumple que $w[n] = w[n+N/2]$, entonces ésta contiene únicamente armónicos pares; es decir, a_k es distinto de cero solamente cuando k es par. Esto es fácil de ver, ya que si $w[n] = w[n+N/2]$, entonces w es periódica con periodo $N/2$, y por lo tanto su frecuencia fundamental es en realidad el doble. De la misma manera, si se cumple que $w[n] = -w[n+N/2]$, entonces la forma de onda contiene solamente armónicos impares.

Una vez calculada la tabla de ondas, es común normalizarla para que ésta tenga una amplitud máxima unitaria; esto se logra haciendo $w \longleftarrow w/\max_n\{w[n]\}$.

17.2.2 Síntesis aditiva de formas de onda clásicas

Una aplicación directa de lo anterior consiste en sintetizar aditivamente las formas de onda clásicas que genera un oscilador analógico, tales como diente de sierra, cuadrada y triangular [10]. Para esto, consideremos primero una señal periódica $x[n]$ con periodo N y su diferencial de primer orden $d_x[n] = x[n] - x[n-1]$. Aplicando las propiedades de linealidad y desplazamiento en tiempo de la Transformada de Fourier, es fácil ver que $D_x[k] = (1 - e^{-j2\pi k/N})X[k]$. Dado que solamente nos interesa estimar los primeros K coeficientes de Fourier para $K << N$, podemos tomar ventaja de la aproximación $e^{j\phi} \approx (1 + j\phi)$ cuando ϕ es cercano a cero. De esta manera se llega a que

$$D_x[k] \approx (2\pi jk/N)X[k]. \tag{17.3}$$

Supongamos ahora que $s[n]$ es una señal tipo diente de sierra con periodo N; por ejemplo $s[n] = (n \bmod N)/N - D$, donde $D = (N-1)/2N$ es una constante que elimina la componente DC. Entonces la diferencial se puede escribir como

$$d_s[n] = \frac{1}{N} + \begin{cases} -1 & \text{si } n = 0, \\ 0 & \text{si } n \neq 0. \end{cases}$$

En otras palabras, $d_s[n] = \frac{1}{N} - \delta[n]$, donde $\delta[n]$ es el impulso discreto (delta de Kronecker). Despreciando el término $1/N$ (que puede considerarse como parte de la componente DC $D_s[0]$), llegamos a que $D_s[k] = -1$ para $k \neq 0$. Aplicando la Ecuación 17.3 se llega entonces a que

$$S[k] \approx \frac{D_s[k]}{2\pi jk/N} = \frac{-N}{2\pi jk} = \frac{jN}{2\pi} \cdot \frac{1}{k},$$

para $0 < k \leq 0$. El factor constante $jN/2\pi$ puede también ser despreciado, sobre todo si se piensa generar una tabla de ondas y normalizarla posteriormente.

Por último, es necesario notar que la señal $s[n]$ es antisimétrica, por lo que puede reconstruirse a partir de una suma de senos, cuyos pesos son inversamente proporcionales al número de parcial k. Mas explícitamente,

$$s[n] \propto \sum_{k=1}^{K} \frac{\text{sen}(2\pi kn/N)}{k}.$$

Retomemos la señal periódica $x[n]$ con periodo N, y consideremos ahora una versión de esta señal desplazada medio ciclo, digamos $y[n] = x[n + N/2]$. Aplicando nuevamente las propiedades de desplazamiento en tiempo de la DFT, se observa que $Y[k] = e^{j\pi k}X[k] = (-1)^k X[k]$. Si ahora sumamos ambas señales, $z[n] = x[n] + y[n]$, es fácil ver que el espectro resultante $Z[k]$ contendrá únicamente armónicos pares (i.e., aquellos con k par), mientras que si tomamos la diferencia $x[n] - y[n]$, el espectro resultante contendrá solamente armónicos impares.

Aplicando lo anterior a la señal diente de sierra $s[n]$, es fácil ver que si tomamos la diferencia entre $s[n]$ y una versión desplazada por medio ciclo de $s[n]$, el resultado es una onda cuadrada $c[n]$; es decir, $c[n] = s[n] - s[n + N/2]$. El espectro $C[k]$ de esta señal está dado por

$$C[k] = S[k] - (-1)^k S[k] = \begin{cases} 2S[k] & \text{si } k \text{ es impar ,} \\ 0 & \text{si } k \text{ es par.} \end{cases}$$

Dado que la onda cuadrada es también antisimétrica, puede aproximarse mediante una suma de senos que incluya únicamente las parciales impares, cuyo peso es también inversamente proporcional al número de parcial:

$$c[n] \propto \sum_{k=0}^{K/2-1} \frac{\text{sen}(2\pi(2k+1)n/N)}{2k+1}.$$

Finalmente, para obtener los coeficientes de Fourier de una onda triangular emplearemos un enfoque mas intuitivo. Partimos de que una onda cuadrada se puede obtener como la derivada de una onda triangular, ya que en el segmento ascendente de la onda triangular la pendiente es constante y positiva, mientras que en el segmento descendente la pendiente es constante y negativa. Llamémosle $t[n]$ a la señal triangular con periodo N, de manera que $t'[n] = c[n]$. Por otra parte, las propiedades de diferenciación de la Transformada de Fourier establecen que $C(\omega) = j\omega T(\omega)$ [20], lo cual nos lleva a que

$$T[k] = \frac{N}{2\pi j} \cdot \frac{1}{k} C[k] \propto \begin{cases} 1/k^2 & \text{si } k \text{ es impar ,} \\ 0 & \text{si } k \text{ es par.} \end{cases}$$

Sin embargo, la onda triangular es simétrica (no antisimétrica), por lo que debe reconstruirse a partir de una suma de cosenos, como se muestra a continuación:

$$t[n] \propto \sum_{k=0}^{K/2-1} \frac{\cos(2\pi(2k+1)n/N)}{(2k+1)^2}.$$

La Figura 17.1 muestra las formas de onda que resultan de sumar 4, 8 y 16 parciales para cada una de las tres ondas clásicas.

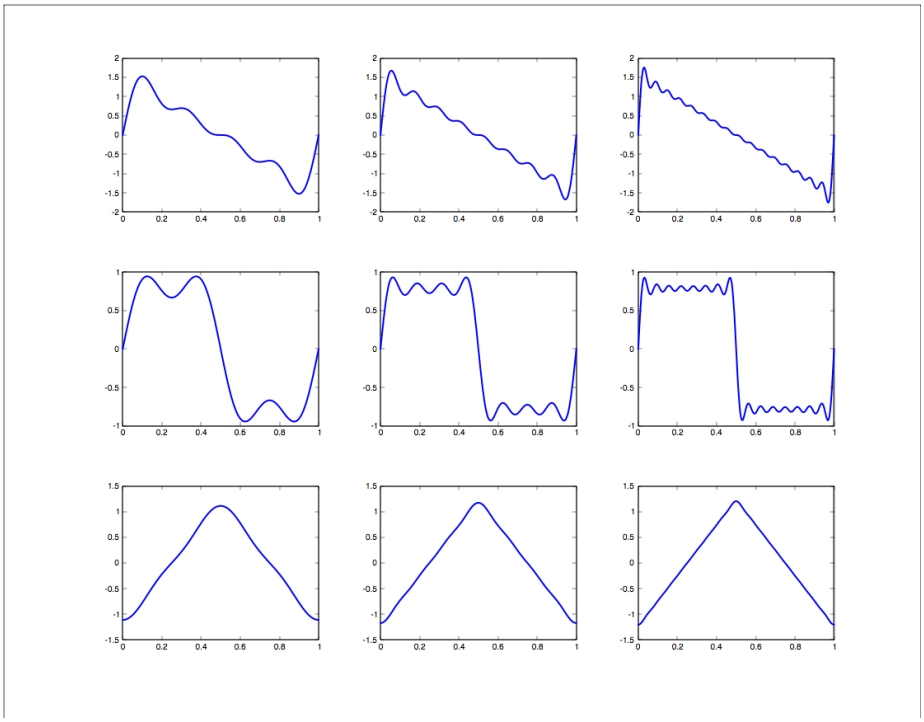


Figura 17.1: Formas de onda clásicas (diente de sierra, cuadrada y triangular) aproximadas mediante síntesis aditiva usando 4 parciales (columna izquierda), 8 parciales (columna central) y 16 parciales (columna derecha).

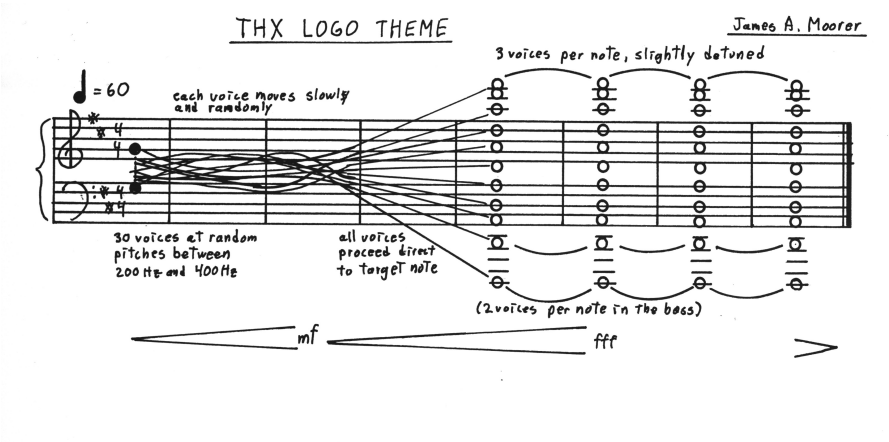


Figura 17.2: Partitura original del tema *Deep Note*, creado mediante síntesis aditiva.

17.2.3 Síntesis de espectros inarmónicos

Matemáticamente, un espectro inarmónico (onda aperiódica) se obtiene a partir de una suma discreta (posiblemente infinita) de parciales cuyas frecuencias no son necesariamente armónicas. De manera general, esto se puede expresar como sigue:

$$x[n] = \sum_{k=0}^{\infty} X_k[n] \exp\{j2\pi f_k n/f_m\}, \tag{17.4}$$

donde f_k y $X_k[n]$ son, respectivamente, la frecuencia y el coeficiente complejo (magnitud y fase) correspondientes a la k-ésima parcial del espectro.

Un modelo mas práctico es el siguiente, donde el número de parciales es finito y éstas corresponden a las frecuencias armónicas, pero con una posible desviación que permita obtener componentes inarmónicas. Además, se reemplazan las exponenciales complejas por funciones seno o coseno. La siguiente ecuación ejemplifica este modelo:

$$x[n] = \sum_{k=1}^{K} X_k[n] \cos\left(2\pi(k + d_k[n])fn/f_m\right),$$

donde f es la frecuencia fundamental y $d_k[n]$ es la desviación en frecuencia de la k-ésima parcial con respecto a su frecuencia base kf. Básicamente, d_k es una señal, por lo general de baja amplitud, que modula la frecuencia de la k-ésima componente. Es común utilizar osciladores de baja frecuencia (LFOs) o ruido filtrado para producir modulaciones cíclicas o aleatorias en el espectro, donde la amplitud de las señales moduladoras está asociada con el grado de inarmonicidad del sonido resultante.

Un ejemplo clásico de síntesis aditiva inarmónica es el tema *Deep Note* de la empresa THX (fundada en 1983 por George Lucas), el cual es un paisaje sónico creado a partir de 30 osciladores que comienzan con frecuencias aleatorias en el rango de 200 Hz a 400 Hz, y después de varios segundos cada oscilador se aproxima a una frecuencia predefinida para concluir en un sonido completamente armónico (un acorde de Re Mayor). La Figura 17.2 muestra la partitura original del compositor James Moorer.

17.3 Objetivos didácticos

- Conocer los fundamentos de la síntesis aditiva, así como su aplicación en la práctica.
- Incorporar la generación de formas de onda mediante síntesis aditiva en un oscilador basado en tablas de onda.
- Experimentar emulando algunos paradigmas basados en síntesis aditiva.

17.4 Material

- Una computadora con alguna de las siguientes combinaciones de software instalado:
  - Processing y Beads
  - Processing y Minim
  - GNU C++ (e.g., MinGW) y PortAudio
- Bocinas o audífonos estéreo

17.5 Procedimiento

En esta práctica se utilizará la síntesis aditiva para incrementar la funcionalidad de nuestro oscilador basado en tablas de ondas (UGen `Oscilador`), agregando métodos para crear tablas de ondas a partir de sumas de senos o cosenos.

Una vez hecho esto, se implementará un programa de prueba que permita definir gráficamente los pesos de las parciales para dos osciladores distintos en un esquema de *crossfade* que permita interpolar entre los dos espectros utilizando el mouse. De esta forma emularemos un esquema de síntesis aditiva donde las ganancias de las parciales pueden variar a lo largo del tiempo.

17.5.1 Funcionalidad aditiva para el oscilador basado en tabla de ondas

Para dotar a la UGen `Oscilador` de capacidades aditivas (para la síntesis de timbres armónicos), implementaremos las dos formas de la Ecuación 17.2 como

métodos de la clase. Es posible crear una nueva clase heredada de `Oscilador` a la que se le agreguen los nuevos métodos, pero lo mejor es simplemente agregarlos a la clase original.

Además del cálculo de la tabla de onda mediante sumas de senos o cosenos, será útil agregar un método que permita normalizar la tabla de ondas para asegurar que sus valores estén en el rango de -1 a 1. Los tres métodos se muestran a continuación:

```
void normalizaTabla(float[] t) {
  if (t == null) t = tabla;
  if (t == null) return;
  float suma = 0, max = 0;

  // Calcula DC
  for (int i = 0; i < t.length; i++) { suma += t[i]; }
  suma /= t.length;

  // Elimina DC y busca el máximo absoluto
  for (int i = 0; i < t.length; i++) {
    t[i] -= suma;
    if (abs(t[i]) > max) { max = abs(t[i]); }
  }
  if (max == 0) return;

  // Normaliza la tabla
  for (int i = 0; i < t.length; i++) { t[i] /= max; }
}

void setSumaSenos(int res, float[] w) {
  float[] t = new float[res];
  for (int k = 0; k < w.length; k++) {
    for (int i = 0; i < res; i++) {
      t[i] += w[k] * sin(2.0 * PI * (k + 1) * i / res);
    }
  }
  normalizaTabla(t);
  tabla = t;
}

void setSumaCosenos(int res, float[] w) {
  float[] t = new float[res];
  for (int k = 0; k < w.length; k++) {
    for (int i = 0; i < res; i++) {
      t[i] += w[k] * cos(2 * PI * (k + 1) * i / res);
    }
  }
```

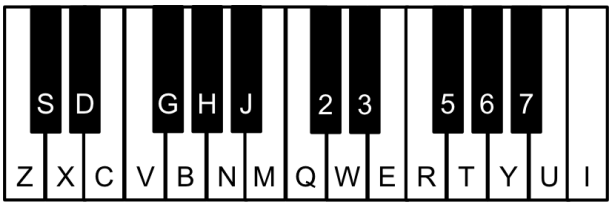


Figura 17.3: Simulación de un teclado de piano utilizando un teclado de computadora

```
    normalizaTabla(t);
    tabla = t;
  }
```

El primer método es el que se utiliza para normalizar una tabla de ondas, ya sea una tabla externa dada como argumento, o bien la tabla interna del oscilador cuando el argumento es `null`. Esta función primero calcula y elimina el DC de la tabla, para luego normalizar con respecto a la magnitud máxima de los valores de la tabla. A continuación, los siguientes métodos calculan una nueva tabla (reemplazando a la anterior) basada en sumas de senos o cosenos, respectivamente. Los argumentos de estas funciones son el tamaño de la tabla `res` (N en la Ecuación 17.2) y un arreglo `w[]` que contenga los pesos a_k de las parciales; el tamaño del arreglo será igual al número de parciales K. Es importante considerar que los arreglos están indexados a partir de cero, por lo que el elemento `w[k]` corresponde a la $(k+1)$-ésima parcial. Una vez calculada la suma de senos o cosenos, se llama al método que normaliza la tabla. Note que el cálculo de la tabla se realiza en un buffer temporal `t[]` el cual hasta el final se asigna a la tabla interna del oscilador. Esto se hace para evitar artefactos ya que el oscilador continuará generando una salida mientras se calcula la nueva forma de onda.

17.5.2 Implementación de un sintetizador aditivo

La arquitectura propuesta consiste de dos osciladores de tablas de ondas (que llamaremos `osc1` y `osc2`), cada uno de los cuales está conectado a un VCA (`vca1` y `vca2`, respectivamente). La idea es interpolar entre las formas de onda de ambos osciladores, por lo que uno de los VCA tendrá una ganancia g, mientras que la ganancia del otro será $1-g$, con $0 \leq g \leq 1$. Al modular el parámetro g, lograremos una transición suave entre los timbres de ambos osciladores. Este parámetro se controlará mediante la coordenada X del mouse. Finalmente, las salidas de ambos VCA serán ruteadas hacia un tercer VCA (llamado `amp`) que permitirá controlar la intensidad final del sonido mediante la coordenada Y del mouse. Además, utilizaremos envolventes para modular tanto la frecuencia de ambos osciladores como la ganancia de los tres VCA, de manera que se produzcan cambios suaves para evitar artefactos.

La interfaz gráfica se dividirá en dos partes: en la parte superior de la ventana se mostrarán dos gráficas con los pesos asociados a las parciales de cada uno de los osciladores. Estos pesos se almacenarán en dos arreglos globales, llamados `pesos1` y `pesos2`. En la parte inferior de la ventana se mostrará la salida del sintetizador usando la función `simplescopio()`, como hemos hecho anteriormente.

La interacción con el sintetizador se realizará de varias maneras. La primera es a través del teclado, el cual usaremos para emular un teclado de piano con dos octavas, como se muestra en la Figura 17.3. Al presionar una de estas teclas, la frecuencia de ambos osciladores se actualizará a la frecuencia de la nota correspondiente. Por otra parte, dotaremos al programa de un modo de edición, en el cual será posible modificar los pesos de las parciales simplemente arrastrando el mouse sobre las gráficas correspondientes. Una variable booleana `modoEdicion` permitirá saber en qué modo se está ejecutando el programa, y utilizaremos la barra espaciadora para alternar entre ambos modos.

Dicho lo anterior, se muestra a continuación la etapa de inicialización del programa:

```
AudioContext ac;
Oscilador osc1, osc2;
VCA vca1, vca2, amp;
Rampa envFrec, envVCA1, envVCA2, envAmp;

final int numParciales = 16;
float[] pesos1, pesos2;

boolean modoEdicion = false;

void setup() {
  size(800, 600);

  pesos1 = new float[numParciales];
  pesos2 = new float[numParciales];

  for (int k = 0; k < numParciales; k++) {
    pesos1[k] = 1.0 / (k + 1);
    pesos2[k] = (k % 2 == 0) ? (1.0 / (k + 1)) : 0;
  }

  // crear cadena de audio
  ac = new AudioContext();
  osc1 = new Oscilador(ac, 110);
  osc2 = new Oscilador(ac, 110);
  vca1 = new VCA(ac);
  vca2 = new VCA(ac);
  amp = new VCA(ac);
```

```
  envFrec = new Rampa(ac);
  envVCA1 = new Rampa(ac);
  envVCA2 = new Rampa(ac);
  envAmp = new Rampa(ac);

  osc1.setModulador(envFrec);
  osc1.setIndice(1);
  osc1.setSumaSen(4096, pesos1);
  osc2.setModulador(envFrec);
  osc2.setIndice(1);
  osc2.setSumaSen(4096, pesos2);

  vca1.setEntrada(osc1);
  vca1.setModulador(envVCA1);
  vca1.setIndice(1);
  vca2.setEntrada(osc2);
  vca2.setModulador(envVCA2);
  vca2.setIndice(1);

  amp.setEntrada(vca1);
  amp.setEntrada(vca2);
  amp.setModulador(envAmp);
  amp.setIndice(1);

  ac.out.addInput(amp);
  ac.start();
}
```

La función `setup()` crea los arreglos para almacenar los pesos de las parciales de ambos osciladores y los inicializa con los pesos que definen una onda diente de sierra para el primer oscilador, y una onda cuadrada para el segundo oscilador. A continuación se crean las UGen y se realizan las conexiones entre ellas. La frecuencia base de los osciladores se inicializa a 110 Hz, la cual se usará como frecuencia de referencia, ya que los cambios en frecuencia se realizarán a través de la envolvente `envFrec`; también se inicializan las tablas de onda de los osciladores con base en los pesos iniciales (utilizando, por supuesto, los métodos implementados en la sección anterior).

A continuación presentamos las funciones utilizadas para desplegar la interfaz gráfica. El dibujo de las gráficas de pesos se ha delegado a una función llamada `dibujaParciales()`, mientras que el resto tiene lugar en la función `draw()`:

```
void dibujaParciales() {
  float x, y;
  float m = 20;
  float w = float(width) / 2 - 2 * m;
  float d = w / numParciales;
```

```
  float h = float(height) / 2 - 2 * m;

  pushStyle();

  stroke(255, 0, 0);
  fill(200);
  rect(m - 2, m - 2, w + 4, h + 4);
  rect(m + width / 2 - 2, m - 2, w + 4, h + 4);

  stroke(0);
  strokeWeight(3);
  for (int k = 0; k < numParciales; k++) {
    x = m + k * d;
    y = m + h - pesos1[k] * h;
    line(x, y, x + d, y);
    x += width / 2;
    y = m + h - pesos2[k] * h;
    line(x, y, x + d, y);
  }

  popStyle();
}

void draw() {
  background(0,0,32);
  if (modoEdicion) text("Modo de Edicion", 10, 10);

  dibujaParciales();

  pushMatrix();
  stroke(0, 255, 0);
  translate(0, height / 4);
  simplescopio(amp.getOutBuffer(0));
  popMatrix();
}
```

La variable `m` define el márgen (en pixeles) que habrá entre las orillas de la ventana y las gráficas. Con base en este margen y el tamaño de la ventana se calcula el tamaño de las gráficas (`w` por `h`), así como el ancho `d` de las barras que corresponden a cada peso. Estos mismos valores se utilizarán para relacionar la posición del mouse con el peso que se desea manipular en el modo de edición.

El control del mouse se realiza mediante tres funciones. La primera de ellas es una función auxiliar llamada `crossfade()` que será encargada de modular las ganancias de los VCAs conectados a los osciladores con base en el parámetro de interpolación g. Por otra parte, se implementará la función callback `mouseDragged()`, la cual tendrá efecto únicamente en el modo de edición y

permitirá manipular los pesos de los espectros. Si el mouse se encuentra en la mitad izquierda de la ventana, se manipularán los pesos correspondientes al primer oscilador; de otra manera, se modificarán los pesos del segundo oscilador. Una vez actualizados los pesos, es necesario recalcular la tabla de ondas del oscilador correspondiente. Por último, la función callback `mouseMoved()` responderá a los movimientos del mouse (sin arrastre) para manipular el crossfade entre ambos osciladores y la ganancia final mediante las coordenadas X y Y del mouse, respectivamente. En modo de edición, el movimiento del mouse simplemente permitirá alternar entre el sonido de un oscilador y otro (asignando $g = 0$ o $g = 1$ al parámetro de crossfade) dependiendo de en qué lado de la pantalla se encuentre el mouse.

```
void crossfade(float g) {
  g = constrain(g, 0, 1);
  envVCA1.cambia(1 - g, 50);
  envVCA2.cambia(g, 50);
}

void mouseDragged() {
  if (modoEdicion) {
    float m = 20;
    float hw = float(width) / 2;
    float w = hw - 2 * m;
    float d = w / numParciales;
    float h = float(height) / 2 - 2 * m;
    float x = (mouseX < hw) ? mouseX : (mouseX - hw);
    float y = constrain((m + h - mouseY) / h, 0, 1);
    int k = (int)constrain((x - m) / d, 0, numParciales - 1);
    crossfade((mouseX < hw) ? 0 : 1);

    float[] p = (mouseX < hw) ? pesos1 : pesos2;
    Oscilador o = (mouseX < hw) ? osc1 : osc2;
    p[k] = y;
    o.setSumaSen(4096, p);
  }
}

void mouseMoved() {
  if (modoEdicion) {
    float hw = float(width) / 2;
    crossfade((mouseX < hw) ? 0 : 1);
  } else {
    crossfade(float(mouseX) / (width - 1));
    envAmp.cambia(1 - (float(mouseY) / height), 50);
  }
}
```

Finalmente, se implementará la función `keyPressed()` para responder a comandos del teclado y emular el teclado de piano de acuerdo a la asignación mostrada en la Figura 17.3. También se responderá a la barra espaciadora, la cual servirá para alternar entre ambos modos de ejecución:

```
void triggerNote(int n) {
  float oct = (float)(n + 9) / 12;
  envFrec.cambia(oct, 50);
}

void keyPressed() {
  char[] low = { 'z', 's', 'x', 'd', 'c', 'v', 'g', 'b', 'h', 'n', 'j', 'm', ','};
  char[] high = { 'q', '2', 'w', '3', 'e', 'r', '5', 't', '6', 'y', '7', 'u', 'i'};
  if (key == ' ') { modoEdicion = !modoEdicion; return; }
  int k = Character.toLowerCase(key);
  for (int i = 0; i < 13; i++) {
    if (k == low[i]) triggerNote(i);
    if (k == high[i]) triggerNote(i + 12);
  }
}
```

La función auxiliar `triggerNote()` simplemente modula la frecuencia de los osciladores para alcanzar la nota deseada, dada por el argumento `n` en semitonos a partir de una nota Do.

17.6 Evaluación y reporte de resultados

1.- Dada la frecuencia de muestreo f_m, calcule el número de parciales que requieren sumarse para sintetizar aditivamente una onda diente de sierra a una frecuencia fundamental f con la mayor fidelidad posible, pero evitando a toda costa el aliasing. Cuántas parciales se requieren para sintetizar la primera y última teclas de un piano, correspondientes a las notas A0 = 27.5 Hz y C8 = 4186.009 Hz, con $f_m = 44.1$ KHz? Discuta los resultados.

2.- Incorpore una función al programa de prueba que permita cambiar entre sumas de senos o sumas de cosenos para generar las tablas de ondas. Verifique que una suma de senos y una suma de cosenos con los mismos pesos dan como resultado un timbre similar (con una amplitud posiblemente distinta, debido a la normalización), pero con una forma de onda diferente. Qué ocurre si se toma la suma de ambos, senos y cosenos?

3.- Uno de los primeros instrumentos basados en síntesis aditiva fue el órgano Hammond, cuyo sonido era generado a partir de nueve osciladores electromecánicos (llamados *ruedas tonales* o *tonewheels*) para cada nota. Cada rueda tonal generaba una forma de onda aproximadamente pura (sinusoidal) con una frecuencia que era un múltiplo específico de la frecuencia fundamental, y cuyas intensidades podían ajustarse en nueve pasos discretos (del 0 al 8) mediante jaladores (ver Figura 17.4). Los múltiplos correspondientes a las nueve parciales son 1/2, 3/2, 1, 2, 3, 4, 5, 6, 8. Las primeras dos parciales (jaladores de color café) son en realidad subarmónicos de la frecuencia fundamental y del tercer armónico. Los jaladores de color blanco corresponden a la fundamental y a sus tres octavas siguientes, mientras que los jaladores de color negro corresponden a armónicos ligeramente disonantes, que añaden textura al sonido. Modifique el programa de prueba para emular el sonido de un órgano Hammond.

Figura 17.4: Deslizadores utilizados para controlar el volumen de cada parcial en una emulación de órgano Hammond (la imagen corresponde a un órgano Nord Electro 4D).

17.7 Retos

Haga su propia versión del tema *Deep Note* de THX (Figura 17.2) sumando la salida de 30 osciladores cuyas frecuencias son moduladas por envolventes. Las envolventes deben lograr que la frecuencia vaya del rango de 200 a 400 Hz hasta una frecuencia predefinida para formar un acorde de Re Mayor (notas D, F y A) a lo largo de tres octavas, en un periodo de aproximadamente 20 segundos, y mantener el acorde por otros 5 segundos. Envíe la suma de los 30 osciladores a través de un VCA para controlar la intensidad del sonido mediante otra envolvente, incrementando la intensidad hasta el máximo durante los primeros 20 segundos y relajándose en el último segundo.

17.8 Conclusiones

Redacte en el recuadro sus conclusiones acerca de los conceptos aprendidos, las actividades realizadas y los objetivos planteados en esta práctica.

Capítulo 18

Síntesis por modulación de frecuencia

Nombre del estudiante	Calificación

18.1 Relación con los programas de estudio

Materia	Unidad y tema
Procesamiento Digital de Señales (IE / IT / IB)	2.2.- Transformada de Fourier 2.3 - Propiedades de la Transformada de Fourier 3.2.- Transformada discreta de Fourier 3.4.- Propiedades de la Transformada Discreta de Fourier
Procesamiento de Señales de Audio (IE)	3.7 - Modulación de frecuencia 5.5.- Aplicaciones musicales en arquitecturas diversas 5.6.- Aplicaciones a las interfaces hombre-máquina 5.7.- Representación auditiva de datos científicos

18.2 Introducción

La modulación de frecuencia (FM) tiene sus orígenes en la década de 1930, cuando comenzó a utilizarse como una alternativa a la modulación en amplitud (AM) para la transmisión de información por radio [24]. Esencialmente se implementa mediante un oscilador de alta frecuencia (señal portadora) cuya

frecuencia es modulada por la señal que se desea transmitir (señal moduladora). Para recuperar la señal moduladora en el receptor (un proceso conocido como *demodulación*) existen varias técnicas; por ejemplo, se puede utilizar un filtro pasabanda entonado a la frecuencia superior de la banda de transmisión, de manera que la amplitud de la salida del filtro siga los cambios en la frecuencia de la señal de entrada; o bien, uno puede utilizar estimadores de frecuencia, como los estudiados en la Práctica 16.

Comparada con la transmisión por AM, la transmisión por FM tiene una mejor tasa señal-a-ruido y es menos propensa a interferencia, pero también tiene un alcance mas reducido debido a que las señales portadoras de FM tienen frecuencias mayores (decenas de MHz) comparadas con AM (cientos de KHz).

Un principio similar al de la transmisión por radio FM se ha utilizado para la transmisión de información digital, donde los dígitos se representan mediante un conjunto discreto de frecuencias. Por ejemplo, para información codificada en binario, se entona el oscilador a una frecuencia f_0 para representar un cero, o a una frecuencia f_1 para representar un uno. Esta técnica se conoce como frequency-shift keying (FSK), y puede implementarse incluso cuando la señal portadora es una señal de audio (Audio FSK ó AFSK), como en los primeros modems que transmitían a través de la línea telefónica.

Matemáticamente, una señal modulada en frecuencia se puede modelar a partir de la siguiente ecuación:

$$y[n] = \cos\left(\omega_c n + I\cos(\omega_m n)\right), \tag{18.1}$$

donde ω_c es la frecuencia de la señal portadora, ω_m es la frecuencia de la moduladora e I es el índice de modulación. Utilizando la identidad trigonométrica para el coseno de la suma de dos ángulos, se llega a

$$y[n] = \cos(\omega_c n)\cos(I\cos(\omega_m n)) - \operatorname{sen}(\omega_c n)\operatorname{sen}(I\cos(\omega_m n)). \tag{18.2}$$

Por otra parte, la composición de funciones trigonométricas puede expresarse en términos de las funciones de Bessel J_α del primer tipo:

$$\cos(I\cos(x)) = J_0(I) + 2\sum_{k=1}^{\infty}(-1)^k J_{2k}(I)\cos(2kx) \tag{18.3}$$

$$\operatorname{sen}(I\cos(x)) = 2\sum_{k=0}^{\infty}(-1)^k J_{2k+1}(I)\cos((2k+1)x) \tag{18.4}$$

Sustituyendo las expresiones anteriores en la Ecuación 18.2 y simplificando los productos de funciones trigonométricas (utilizando nuevamente las identidades de sumas y restas de ángulos), se llega a la siguiente expresión [10]:

$$y[n] = J_0(I)\cos(\omega_c n) + \sum_{k=1}^{\infty} J_k(I)\left\{\cos\left((\omega_c + k\omega_m)n + \frac{k\pi}{2}\right) + \cos\left((\omega_c - k\omega_m)n + \frac{k\pi}{2}\right)\right\}.$$

De lo anterior se puede deducir que la modulación en frecuencia distribuye la energía de la señal portadora en bandas laterales (*sidebands*) equiespaciadas (en frecuencia), ubicadas a ambos lados de la frecuencia portadora, cuyo espaciamiento es igual a la frecuencia moduladora. La amplitud o intensidad de las bandas laterales está regida por las funciones de Bessel $J_k(I)$, las cuales dependen del índice de modulación. Aunque las funciones de Bessel son oscilatorias, estas tienden a decaer con respecto al orden k, por lo que la amplitud de las bandas laterales también decae conforme se alejan de la frecuencia portadora.

Para la aplicación de transmisión por radio, la onda portadora suele tener una frecuencia mucho mayor que el ancho de banda de la señal a transmitir (la moduladora), por lo que las bandas laterales se ubican en un ancho de banda relativamente pequeño alrededor de la portadora, y así no interfieren con otras bandas de transmisión.

En 1973, John Chowning propuso como aplicación la síntesis de timbres audibles con espectros complejos mediante un esquema de modulación de frecuencia donde tanto la señal portadora como la moduladora tuvieran frecuencias en el rango audible [25]. Considere, por ejemplo, el caso donde ambas frecuencias son iguales; es decir, $\omega_m = \omega_p$. En este caso, las bandas laterales estarían ubicadas en múltiplos de ω_p, dando como resultado un espectro armónico. El índice de modulación I podría usarse entonces para controlar la redistribución de la energía hacia los armónicos superiores, manipulando así la brillantez del sonido resultante.

De manera mas general, podemos caracterizar el espectro resultante mediante dos parámetros: el índice de modulación I, y la razón entre la frecuencia moduladora y la portadora $R = f_m/f_p$. Si R es una fracción simple; es decir, si puede expresarse como la razón de dos números enteros pequeños, entonces el espectro resultante será armónico De otra forma, se obtendrá un espectro inarmónico. Para el caso de un espectro armónico, donde R puede expresarse como una fracción reducida $R = a/b$ con a, b enteros, se pueden hacer las siguientes consideraciones adicionales:

- Si $b = 1$ (es decir, si R es entero), entonces la frecuencia fundamental del espectro resultante es f_p.
- En general, f_p será la frecuencia del b-ésimo armónico del espectro resultante.
- Si $a = 1$, entonces la frecuencia fundamental del espectro resultante es f_m y el espectro contendrá a todos los armónicos (f_m, $2f_m$, $3f_m$, etc.).
- Si $a = 2$, entonces el espectro contendrá únicamente los armónicos impares, dando lugar a una forma de onda simétrica.

En esta práctica se implementará el esquema básico de síntesis por frecuencia modulada basado en dos operadores (portadora y modulador), explorando el rango de timbres que este esquema puede producir y comparándolo con esquemas estudiados anteriormente.

18.3 Objetivos didácticos

- Conocer las aplicaciones de la modulación de frecuencia.
- Implementar un esquema básico de síntesis de sonido basado en modulación de frecuencia.
- Experimentar con los timbres obtenidos mediante la modulación de frecuencia con dos operadores.

18.4 Material

- Una computadora con alguna de las siguientes combinaciones de software instalado:
  - Processing y Beads
  - Processing y Minim
  - GNU C++ (e.g., MinGW) y PortAudio
- Bocinas o audífonos estéreo

18.5 Procedimiento

Para esta práctica no es necesario implementar nuevas UGens ya que toda la funcionalidad requerida para implementar un sintetizador FM reside en las UGens `Oscilador`, `VCA` y `Rampa` implementadas en prácticas anteriores.

Como se mencionó en la Introducción, los parámetros básicos para la síntesis FM son el índice de modulación I y la razón R entre las frecuencias moduladoras y portadora. En particular, el índice de modulación permite controlar la brillantez del sonido, la cual suele variar a lo largo de la duración de un sonido. Para tener un mayor control sobre la amplitud de la modulación, podemos insertar un VCA entre la señal moduladora y la portadora. Por lo general, la ganancia de este VCA será modulada a su vez por una envolvente, de manera que podamos controlar el contenido espectral a lo largo de la duración de cada sonido. Así mismo, el oscilador que genera la señal portadora se pasará por un VCA modulado por una envolvente para controlar la amplitud final del sonido generado.

En otras palabras, la señal de cada uno de los osciladores involucrados (portadora y moduladora) se pasará a través de un VCA modulado por una envolvente. Esta combinación de VCO, VCA y envolvente se conoce como un *operador* (Figura 18.1a) en la terminología adoptada por la empresa Yamaha, que fue la primera en comercializar sintetizadores basados en FM a principios de la década de 1980. El esquema básico de síntesis FM que hemos descrito requiere dos operadores (Figura 18.1b), pero existen implementaciones comerciales con cuatro, seis y ocho operadores que pueden inter-modularse de distintas maneras, siendo incluso posible que un operador se module a sí mismo.

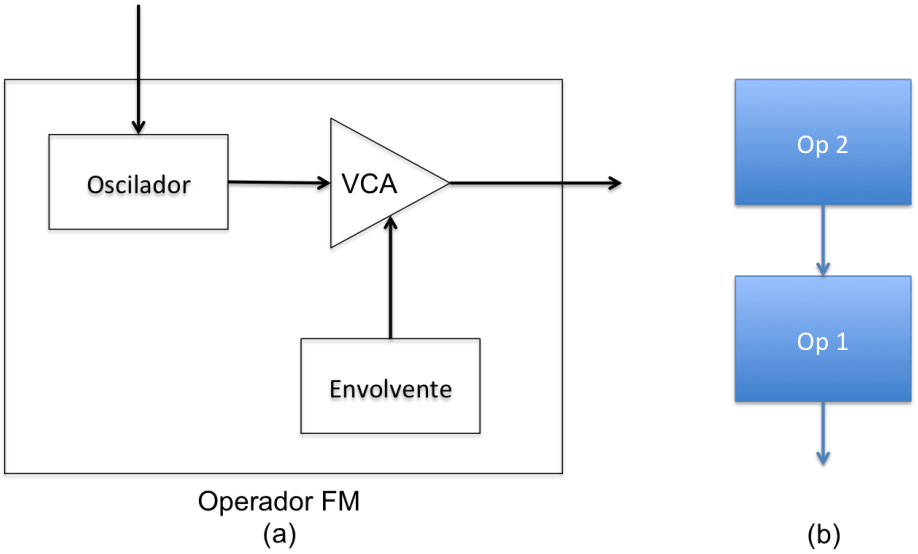


Figura 18.1: (a) Arquitectura típica de un operador en un sintetizador FM. El operador recibe como entrada la señal que modula la frecuencia del oscilador, y como salida la señal del oscilador cuya amplitud es controlada por una envolvente. (b) Configuración típica con dos operadores. El caso mas simple de síntesis FM.

18.5.1 Implementación

La implementación de un sintetizador FM puede realizarse a partir de las UGens con las que ya se cuenta. Cada operador consta de un `Oscilador` que pasa por un `VCA` el cual es modulado por una `Rampa`. La salida del operador es la salida del `VCA`, mientras que la entrada del operador es la entrada de modulación del oscilador.

El siguiente código ejemplifica la etapa de construcción del sintetizador en Processing y Beads:

```
// Variables globales y UGens requeridas
AudioContext ac;
Oscilador osc1, osc2;
VCA vca1, vca2;
Rampa env1, env2;

void setup() {
  size(800, 600);

  // Crear las UGens
  ac = new AudioContext();
  osc1 = new Oscilador(ac, 440);
  osc2 = new Oscilador(ac, 440);
  vca1 = new VCA(ac);
  vca2 = new VCA(ac);
  env1 = new Rampa(ac);
  env2 = new Rampa(ac);

  // Inicializa los osciladores
  osc1.setSenoidal(4096);
  osc1.setFMLineal();
  osc2.setSenoidal(4096);
  osc2.setFMLineal();

  // Conecta las UGens del Operador 1 (portador)
  vca1.setEntrada(osc1);
  vca1.setModulador(env1);
  vca1.setIndice(0.8);

  // Conecta las UGens del Operador 2 (modulador)
  vca2.setEntrada(osc2);
  vca2.setModulador(env2);
  vca2.setIndice(0);

  // Conecta la salida del Operador 2 a la entrada del Operador 1
  osc1.setModulador(vca2);
  osc1.setIndice(1);
```

```
  // Conecta la salida del Operador 1 a la salida física de audio
  ac.out.addInput(vca1);
  ac.start();
}
```

Note que el tipo de modulación debe ser lineal, de manera acorde con la Ecuación 18.1. Así mismo, se utiliza un tamaño de 4096 para las tabla de ondas de los osciladores. Esto debido a que la menor frecuencia de interés es de 20 Hz, lo cual corresponde a un periodo de 2205 muestras (a una frecuencia de muestreo de 44.1 KHz), por lo que se recomienda utilizar una tabla de onda con un tamaño de por lo menos 2205 muestras para reducir los artefactos en bajas frecuencias debido a la discretización de la forma de onda.

Para reproducir una nota con una frecuencia f, es necesario entonar el operador 1 a la frecuencia f y el operador 2 a la frecuencia Rf, donde R es la razón de frecuencias moduladora/portadora. Esto se puede realizar a la hora de "disparar" una nota. Por ejemplo, la siguiente función realiza esta tarea:

```
// Parametros del sintetizador
int octave; // transposicion por octavas
float ratio; // razón de frecuencias moduladora/portadora
float attack1, decay1; // parametros de la envolvente 1
float attack2, decay2; // parametros de la envolvente 2

void disparaNota(int n) {
  float freq = 55 * pow(2, octave + (float)(n + 9) / 12);
  op1.setFrecuencia(freq);
  op2.setFrecuencia(freq * ratio);
  env1.cancela().cambia(0,4).cambia(1, attack1).agrega(0, decay1);
  env2.cancela().cambia(0,4).cambia(1, attack2).agrega(0, decay2);
}
```

En este ejemplo, las envolventes son de dos etapas: ataque y decaimiento. Los tiempos de cada etapa se almacenan en las variables globales `attack1`, `decay1`, `attack2` y `decay2`. La razón de frecuencias se almacena también como una variable global llamada `ratio`; sin embargo, el índice de modulación se controla llamando directamente al método `vca2.setGanancia()`, o bien al método `osc1.setIndice()`. El efecto es distinto: modificar la ganancia de `vca2` tiene un efecto aditivo sobre el índice de modulación, mientras que modificar el índice en `osc1` tiene un efecto multiplicativo. Es interesante experimentar con ambos enfoques, aunque en este ejemplo utilizaremos el enfoque aditivo.

Una manera simple (aunque no muy precisa en términos de ejecución) de disparar los sonidos es simulando un teclado de piano con el teclado de la computadora. Esto puede hacerse como sigue:

```
void keyPressed() {
  char[] high = { 'q', '2', 'w', '3', 'e', 'r', '5', 't', '6', 'y', '7', 'u', 'i'};
```

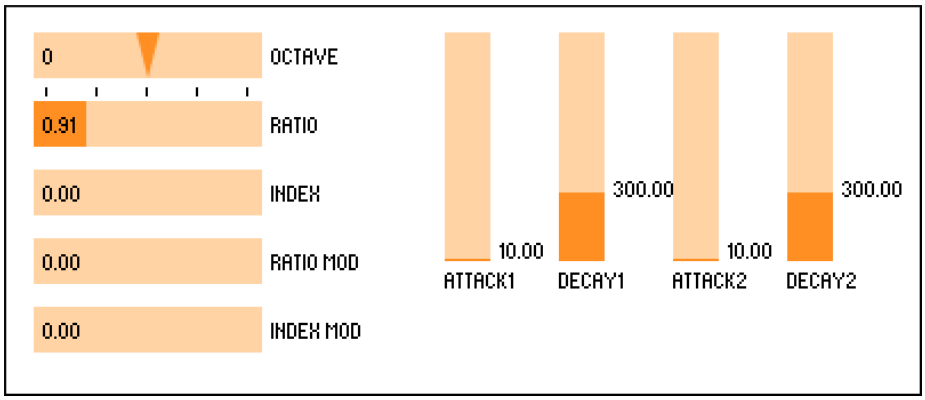


Figura 18.2: Ejemplo de interfaz de usuario para el control de los parámetros de un sintetizador FM de dos operadores

```
  char[] low = { 'z', 's', 'x', 'd', 'c', 'v', 'g', 'b', 'h', 'n', 'j', 'm', ','};
  for (int i = 0; i < 13; i++) {
    if (key == low[i]) disparaNota(i);
    if (key == high[i]) disparaNota(i + 12);
  }
}
```

18.5.2 Interfaz gráfica de usuario

Queda pendiente la elaboración de una interfaz de usuario que permita manipular todos los parámetros de síntesis. En este caso, los parámetros son los siguientes:

- Transposición por octavas (variable `octave`), con un rango entre -2 y +2.
- Razón de frecuencias (variable `ratio`) (un buen rango es entre 0 y 4).
- Indice de modulación base (ganancia base de `vca2`), con un rango de 0 a 10.
- Efecto de la envolvente en el índice de modulación (índice de modulación de `vca2`), con rango de 0 a 10.
- Tiempos de cada etapa de las envolventes (variables `attack1`, `decay1`, `attack2` y `decay2`) con rangos desde 10 ms a varios segundos.

También es posible modular la razón de frecuencias mediante alguna de las envolventes, lo cual da lugar a efectos especiales interesantes. Para lograr esto, uno puede simplemente asignar una de las envolventes como modulador del `osc2` (e.g., `osc2.setModulador(env2)`) dentro de la función `setup()`, y posteriormente controlar el grado de modulación llamando al método `osc2.setIndice()`.

Hablamos entonces de 8 o 9 parámetros que deben ser controlados por el usuario mediante una interfaz gráfica. En la Figura 18.2 se muestra un ejemplo de interfaz elaborada en Processing con la librería ControlP5, la cual permite incorporar diversos elementos de interfaz gráfica como botones y deslizadores. Esta librería puede instalarse directamente desde el entorno de Processing. Los detalles de la implementación de una interfaz se dejan como ejercicio para el alumno.

Por último, se recomienda ampliamente agregar a la interfaz una sección donde se despliegue el espectro y la forma de onda del sonido generado por el sintetizador. Esto ayudará a visualizar el efecto del índice de modulación y la razón de frecuencias.

18.6 Evaluación y reporte de resultados

1.- Note que las ecuaciones 18.3 y 18.4 corresponden a pasar una señal senoidal por waveshapers de la forma $f(x) = \cos(x)$ y $f(x) = \text{sen}(x)$, respectivamente. El primer waveshaper genera armónicos pares mientras que el segundo genera armónicos impares. En este sentido, cada uno de los términos de la Ecuación 18.2 puede verse como el resultado de modular la amplitud de la señal portadora por la salida de un waveshaper sinusoidal aplicado a la señal moduladora. Pruebe experimentalmente este enfoque implementando los waveshapers senoidal y cosenoidal, y reemplazando la arquitectura basada en operadores por dos parejas de osciladores, cada una de las cuales pasa por un VCA (es decir, implementando directamente la Ecuación 18.2).

2.- Explore lo que ocurre cuando se utilizan otras formas de onda distintas a la senoidal como señales portadora y/o moduladora. Algunos sintetizadores FM con múltiples operadores utilizan dos de ellos para generar una onda compleja que después será utilizada como señal moduladora de un tercer operador. En nuestro caso, podemos continuar utilizando dos operadores y generar distintas formas de onda mediante la tabla de ondas o mediante waveshapers; sin embargo, usando este enfoque no es posible generar una señal moduladora que no sea armónica.

18.7 Retos

Simule un canal de transmisión/recepción FM entre dos computadoras usando señales acústicas. Utilice una señal senoidal a una frecuencia audible (digamos, alrededor de 1000 Hz) como portadora, y una señal moduladora (i.e., la señal a transmitir) con un ancho de banda considerablemente menor a la frecuencia de la portadora (digamos, alrededor de 100 Hz). Una computadora funcionará como emisor produciendo la señal modulada a través de la salida física de audio, mientras que la otra computadora funcionará como receptor utilizando la entrada física de audio. El receptor debe implementar algún esquema de demodulación (e.g., filtrado y/o estimación de frecuencia) para recuperar la señal transmitida.

18.8 Conclusiones

Redacte en el recuadro sus conclusiones acerca de los conceptos aprendidos, las actividades realizadas y los objetivos planteados en esta práctica.

Capítulo 19

Síntesis sustractiva

Nombre del estudiante	Calificación

19.1 Relación con los programas de estudio

Materia	Unidad y tema
Procesamiento Digital de Señales (IE / IT / IB)	6.4.- Diseño de filtros IIR
Procesamiento de Señales de Audio (IE)	3.4 - Síntesis sustractiva 5.5.- Aplicaciones musicales en arquitecturas diversas 5.6.- Aplicaciones a las interfaces hombre-máquina 5.7.- Representación auditiva de datos científicos

19.2 Introducción

Las técnicas de síntesis de sonido estudiadas hasta ahora (waveshaping, aditiva, modulación en amplitud, modulación en frecuencia), consisten en tomar una o más señales espectralmente simples (e.g., sinusoidales) y procesarlas para obtener señales con espectros mas complejos mediante superposición, distorsión y/o modulación.

La síntesis sustractiva utiliza un enfoque opuesto. Parte de una señal armónicamente rica, tal como una onda diente de sierra o cuadrada, y utiliza filtros para atenuar los armónicos no deseados, y de esa manera controlar el timbre del sonido resultante. Esta técnica fue popularizada por Robert Moog en las décadas de 1960 y 1970, quien desarrolló los primeros osciladores, filtros y amplificadores controlados por voltaje que sentaron las bases de la síntesis

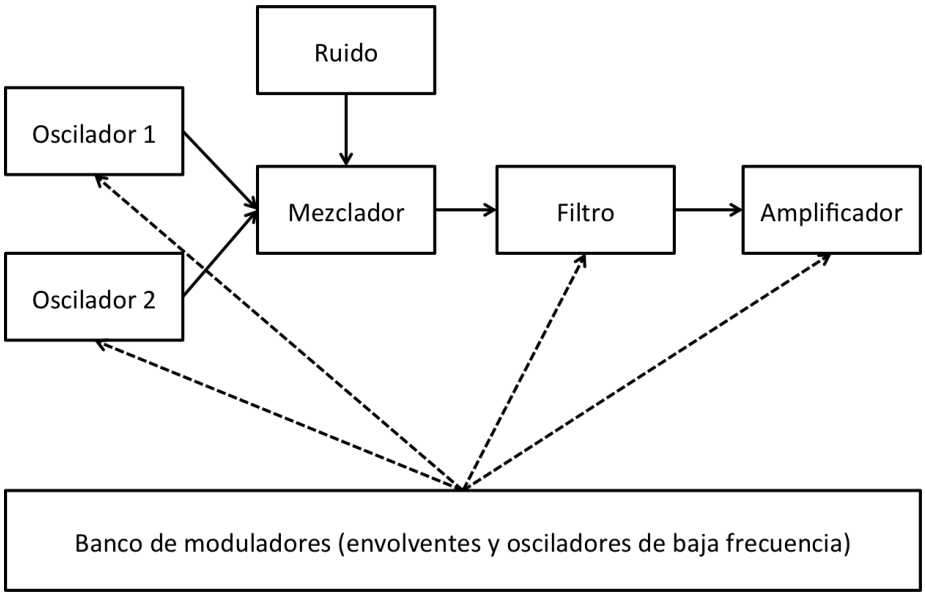


Figura 19.1: Diagrama esquemático de un sintetizador sustractivo típico. Las líneas punteadas representan las conexiones de señales moduladoras.

sustractiva analógica [16]. Estos principios continúan vigentes en la actualidad, en especial durante el resurgimiento de sintetizadores analógicos que ha tenido lugar en las últimas dos décadas.

La clave de la síntesis sustractiva reside en el filtro, el cual usualmente es un pasa-bajas de segundo o cuarto orden, aunque también se utilizan filtros pasa-banda y pasa-altas. El filtro pasa-bajas permite modelar la forma en que la energía del espectro es absorbida una superficie, donde las frecuencias mas altas sufren una mayor absorción de energía y son por lo tanto atenuadas en mayor grado, simulando de esta manera la caja resonante de un instrumento musical. Similarmente, los filtros pasa-banda pueden utilizarse para simular las formantes (frecuencias de resonancia que aparecen como picos en el espectro) que se producen al hablar o cantar, y que cambian al variar la forma de la boca y la posición de la lengua y el paladar. En particular, las formantes cambian al emitir distintas vocales como 'a', 'e', 'o', etc. [11]

Dado que el contenido espectral puede variar a lo largo de la duración de un sonido, es importante que el filtro cuente con la capacidad de modular algunos de sus parámetros mediante una señal externa, en particular la frecuencia de corte o frecuencia central del filtro. La señal de modulación suele ser una envolvente con una forma predefinida, o un oscilador de baja frecuencia (LFO) para producir variaciones cíclicas (un efecto conocido como *wah*). También es común el combinar múltiples señales moduladoras (por ejemplo, una envolvente y un LFO) para lograr distintos efectos. La Figura 19.1 ilustra una arquitectura

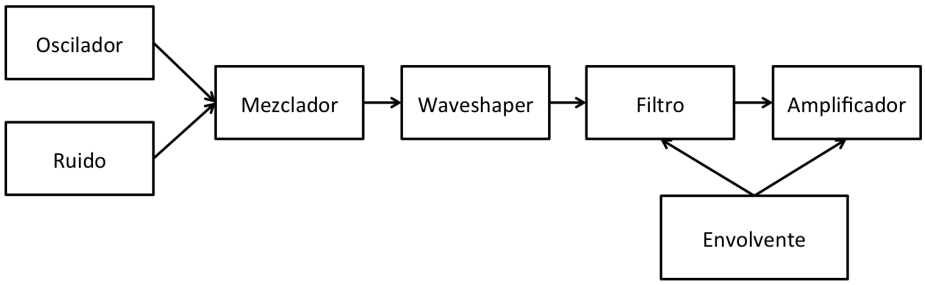


Figura 19.2: Diagrama esquemático del sintetizador sustractivo a implementar.

típica de un sintetizador sustractivo con dos osciladores y una fuente de ruido cuyas señales se combinan antes de pasar por el filtro y finalmente por un amplificador que controlará la intensidad del sonido final; además, se tiene un cierto número de fuentes de modulación (generadores de envolvente y osciladores de baja frecuencia) que permiten modular el tono de los osciladores, el ancho de pulso (en caso de que el oscilador pueda generar señales con ancho de pulso variable), la frecuencia de corte del filtro y la ganancia del amplificador.

En esta práctica se implementará un sintetizador sustractivo muy básico que permita explorar el rango de timbres que se pueden obtener a partir de un filtro resonante y de la modulación de la frecuencia de corte.

19.3 Objetivos didácticos

- Conocer los principios técnicos de la síntesis sustractiva y sus diferencias con respecto a otros métodos de síntesis de sonidos.
- Implementar un sintetizador sustractivo básico y explorar el rango de timbres que este puede producir.

19.4 Material

- Una computadora con alguna de las siguientes combinaciones de software instalado:
  - Processing y Beads
  - Processing y Minim
  - GNU C++ (e.g., MinGW) y PortAudio
- Bocinas o audífonos estéreo

19.5 Procedimiento

El sintetizador a implementar consistirá de un oscilador y una fuente de ruido que se combinarán a través de un mezclador. El oscilador debe ser capaz de generar ondas con un alto contenido armónico, por lo que utilizaremos un oscilador de tabla de ondas. La salida del mezclador pasará por un waveshaper para aplicar distorsión armónica e incrementar aún mas el contenido espectral. La señal resultante pasará entonces por un filtro resonante y finalmente por un amplificador para controlar la curva de volumen final. Una envolvente sencilla se utilizará para modular tanto la ganancia del amplificador como la frecuencia central del filtro. Esta arquitectura se ilustra en la Figura 19.2.

La arquitectura propuesta requiere de algunas funciones que tal vez no hayan sido implementadas aún en prácticas anteriores. Lo primero es implementar un filtro resonante con modulación de la frecuencia central. Lo mas sencillo, en este caso, es hacer una nueva implementación del filtro pasa-banda resonante (la clase `FiltroPBR`) que funcione para un solo canal de salida, y que incorpore un canal de entrada adicional para la señal moduladora. Dentro del ciclo principal de la función callback, será necesario calcular la frecuencia en cada muestra considerando una modulación exponencial; es decir, multiplicando la frecuencia base por 2^m, donde m es el producto de la señal moduladora por el índice de modulación. Una vez calculada la frecuencia, será necesario recalcular todos los coeficientes del filtro, para posteriormente calcular el valor de la muestra de salida. Un filtro cuya frecuencia de corte o frecuencia central puede ser modulada mediante una señal externa, se conoce como un *filtro controlado por voltaje* o *VCF*.

Además del VCF, será necesario contar con una UGen que permita combinar dos o mas señales con distintas ganancias; por ejemplo, la clase `Mezclador` que se propone como reto al final de la Práctica 3; o bien, pasar las señales a combinar por amplificadores de ganancia fija y luego combinar las salidas de los amplificadores enviándolas a la misma entrada (en Beads), o mediante un objeto `Summer` (en Minim).

Por último, será útil que el oscilador sea capaz de generar las distintas formas de onda "clásicas" de un sintetizador analógico, como diente de sierra, triangular y cuadrada. Estas funciones se proponen como un ejercicio en la Práctica 13, aunque también se pueden generar aditivamente como sumas de senos y/o cosenos.

19.5.1 Interfaz gráfica de usuario

Para la interfaz de usuario, pueden considerarse elementos que permitan manipular los siguientes parámetros:

- Transposición por octavas (rango de -2 a +2).
- Nivel de ruido (ganancia aplicada al ruido en el mezclador), con un rango de 0 a 1. En general, es común agregar un control de ganancia por cada

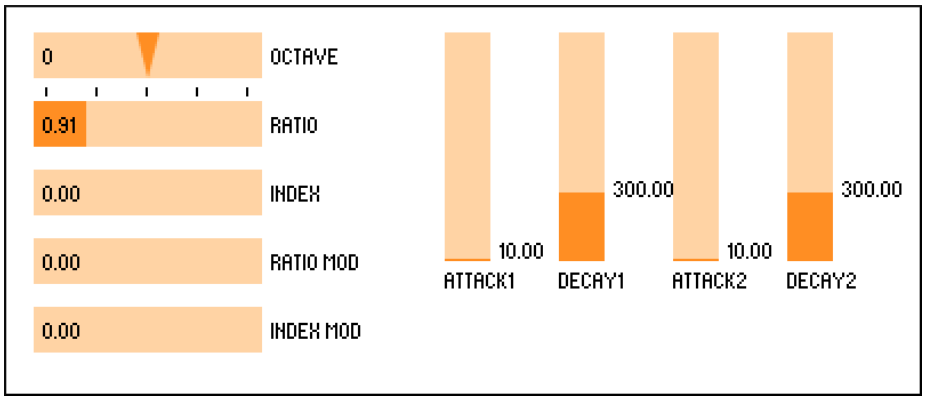


Figura 19.3: Ejemplo de interfaz de usuario para el control de los parámetros de un sintetizador FM de dos operadores

canal de entrada en el mezclador (osciladores, ruido, entrada de audio externa, etc).

- Nivel de distorsión (ganancia del waveshaper), con un rango de 0 a 1.
- Frecuencia de corte base del filtro. Por lo general se utiliza todo el rango audible, es decir, de 20 a 20000 Hz, pero con una escala logarítmica para tener mayor precisión. Por ejemplo, se puede calcular la frecuencia base como $f_c = 20 * 10^x$, donde x es el valor del control en un rango de 0 a 3.
- Parámetros de la envolvente. En este caso se utiliza una envolvente simple con etapas de ataque y decaimiento.
- Indice de modulación de la frecuencia de corte con respecto a la envolvente. También es posible modular la ganancia del waveshaper mediante la misma envolvente.

En la Figura 19.3 se presenta un ejemplo de interfaz gráfica para la arquitectura descrita, utilizando la librería ControlP5 para Processing.

19.6 Evaluación y reporte de resultados

1.- Explore los timbres que pueden obtenerse mediante la síntesis sustractiva, en particular cuando se modula la frecuencia de corte del filtro mediante la envolvente (es posible que sea necesario disminuir la frecuencia de corte base para que la envolvente tenga algún efecto). Compare estos timbres con los que se pueden obtener mediante síntesis FM. Trate de obtener sonidos similares con ambos tipos de síntesis, así como sonidos que son particulares a cada tipo.

2.- Incorpore un oscilador de baja frecuencia (digamos, con un rango de 0.1 a 20 Hz) como modulador adicional para la frecuencia de corte, la ganancia del amplificador y la frecuencia del oscilador. Agregue elementos de interfaz para controlar la frecuencia del LFO y el grado de modulación para cada uno de los tres destinos. Es posible que necesite utilizar amplificadores adicionales y/o mezcladores para combinar las múltiples señales moduladoras del LFO y de la envolvente. El efecto que se obtiene al modular alguno de los parámetros mediante un LFO recibe diferentes nombres, según el parámetro modulado: vibrato (modulación de tono o frecuencia del oscilador), trémolo (modulación de amplitud) o wah (modulación de frecuencia de corte del filtro resonante).

19.7 Retos

A lo largo de la historia de la síntesis sustractiva se han desarrollado diversos tipos de filtros con distintas propiedades (respuesta en frecuencia, resonancia, distorsiones y otras propiedades no lineales) que imparten un carácter particular a cada diseño. Aunque muchos de los diseños originales se realizaron mediante electrónica analógica, en las últimas dos décadas se ha buscado modelar o aproximar estos filtros en el dominio digital. El filtro pasa-banda resonante utilizado en esta práctica no corresponde a ninguno de estos diseños, por lo que su aplicación hacia la síntesis de sonido es limitada. Se sugiere al alumno buscar y probar otros diseños de filtros, sobre todo aquellos basados en el filtro pasa-bajas de Moog. Se recomienda visitar los sitios `www.musicdsp.org`, `https://github.com/ddiakopoulos/MoogLadders`, así como las referencias [26, 27, 28, 29].

19.8 Conclusiones

Redacte en el recuadro sus conclusiones acerca de los conceptos aprendidos, las actividades realizadas y los objetivos planteados en esta práctica.

Capítulo 20

Retroalimentación: Parte II

Nombre del estudiante	Calificación

20.1 Relación con los programas de estudio

Materia	Unidad y tema
Procesamiento Digital de Señales (IE / IT / IB)	6.2.- Consideraciones para el diseño de filtros 6.4.- Diseño de filtros IIR
Procesamiento de Señales de Audio (IE)	2.4.- Filtros de orden superior 4.1.- Eco y retardos 5.5.- Aplicaciones musicales en arquitecturas diversas

20.2 Introducción

La retroalimentación en un sistema de procesamiento se produce cuando parte de la salida del sistema se suma a la señal de entrada. En la Práctica 4 se experimentó brevemente con retroalimentación al utilizar el micrófono y las bocinas de la computadora como entrada y salida de audio. Por lo general, uno buscará eliminar este tipo de retroalimentación para que no se contamine la señal de entrada, pero el efecto resultante puede llegar a ser de utilidad, sobre todo si uno puede controlar, de alguna manera, el grado de distorsión y las frecuencias de resonancia que se generan.

También es posible que la retroalimentación se produzca solamente en algunos bloques del sistema. Esto lo hemos aplicado en la implementación de filtros basados en ecuaciones en diferencias. Por ejemplo, recordemos el filtro pasa-bajas de un polo, dado por la siguiente ecuación:

$$y[n] = (1 - p)x[n] + py[n - 1], \tag{20.1}$$

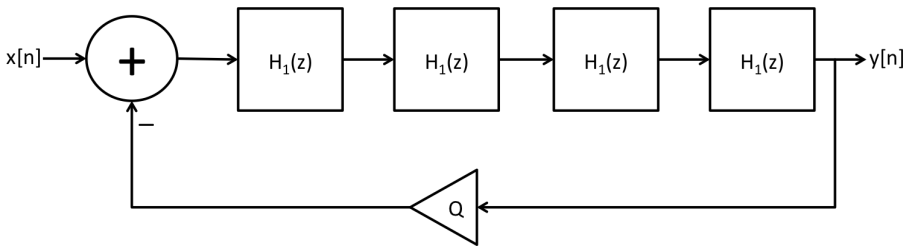


Figura 20.1: Diagrama esquemático del filtro en cascada de Moog.

donde p es la posición del polo, la cual se puede aproximar como $p \approx 1 - \omega_c$ para una frecuencia de corte ω_c en radianes por muestra (ver Prácticas 5 y 6). La ecuación anterior se puede reescribir como

$$y[n] = (1-p)\left(x[n] + \frac{p}{1-p}y[n-1]\right),$$

lo cual representa a un sistema amplificador (con factor de ganancia $1-p$) donde la salida $y[n-1]$ se retroalimenta hacia la entrada con un factor de ganancia $p/(1-p)$. Note que, dado que el sistema se implementa en tiempo discreto, es necesario que la salida retroalimentada tenga un retardo de por lo menos una muestra.

En las siguientes prácticas, exploraremos algunas otras aplicaciones de la retroalimentación. Específicamente, se incorporará la retroalimentación en algunos de los bloques de procesamiento ya implementados y se estudiarán los resultados. Comenzaremos en esta práctica con la implementación de un filtro pasa-bajas resonante y una línea de retardo retroalimentada para obtener efectos de eco.

20.2.1 Filtro pasa-bajas resonante

En la Práctica 19 se mencionó que el componente fundamental de la síntesis sustractiva es el filtro, y se utilizó como ejemplo el filtro pasa-bandas resonante diseñado en la Práctica 6. Sin embargo, la mayoría de los sintetizadores sustractivos cuentan mas bien con un filtro pasa-bajas resonante, ya que éste modela de manera mas adecuada la absorción acústica y resonancia en la mayoría de los instrumentos musicales.

Uno de los diseños clásicos de filtro pasa-bajas resonante es el filtro Moog [16], el cual esencialmente consiste en cuatro filtros pasa-bajas de un polo conectados en cascada, donde la salida del último filtro se retroalimenta hacia la entrada del primero [26]. Esto se ilustra en la Figura 20.1. Cada bloque rectangular representa un filtro pasa-bajas de un polo con función de transferencia $H_1(z)$, la salida $y[n]$ del último bloque se retroalimenta *negativamente* hacia la entrada con un factor de ganancia Q. Suponiendo que implementamos cada etapa mediante la Eq. 20.1, entonces cada etapa puede implementarse como

```
y = (1 - p) * x + p * ya;
ya = y;
```

donde `x` representa la muestra de entrada al bloque, `ya` representa la salida anterior de ese bloque (la cual debe conservarse en todo momento) y `p` es la posición del polo. Dado que la salida de un bloque se convierte en la entrada del siguiente bloque, podemos implementar las cuatro etapas como sigue:

```
y = x;
y = (1 - p) * y + p * ya1;  ya1 = y;
y = (1 - p) * y + p * ya2;  ya2 = y;
y = (1 - p) * y + p * ya3;  ya3 = y;
y = (1 - p) * y + p * ya4;  ya4 = y;
```

donde `ya1, ya2, ya3, ya4` almacenan la salida anterior para cada uno de los bloques. Estas variables deben conservar su valor en todo momento, lo cual significa que serán miembros de la clase que implementa el filtro.

Para implementar la retroalimentación, es necesario sumar la salida anterior de todo el sistema (almacenada en `ya4`) a la entrada del sistema, pero con un factor de ganancia negativo. Esto se hace simplemente reemplazando la primer línea del pseudo-código anterior con la siguiente:

```
y = x - Q * p * ya4;
```

donde `Q` es el factor de retroalimentación, también llamado *resonancia*. En este caso, la resonancia debe estar entre 0 y 4. Valores mayores a 4 harán que el filtro se vuelva inestable rápidamente.

Para concluir la implementación, solo resta calcular la posición del polo p a partir de la frecuencia de corte base del filtro ω_c y tomar en cuenta la modulación de la frecuencia de corte (de manera exponencial, por octavas). Partiendo del análisis realizado en la Práctica 6, la posición del polo se puede aproximar por $p = \max\{1 - \omega, 0\}$ con $\omega = \omega_c \cdot 2^m$, donde m es el producto de la señal moduladora por el índice de modulación.

Finalmente, hay dos detalles importantes que es necesario aclarar. El primero es que, como ya se ha mencionado, la retroalimentación es negativa; esto es debido a que, para el filtro de un polo, la respuesta en fase alrededor de la frecuencia de corte es aproximadamente $-\pi/4$ en el rango donde el filtro tiene un mayor efecto ($p > 0.8$). Al encadenar los cuatro filtros de un polo, se tendrá una respuesta en fase total aproximadamente igual a $-\pi$, por lo que las componentes de la señal de salida alrededor de la frecuencia de corte estarán invertidas en fase con respecto a la señal de entrada. Al retroalimentar la salida con un factor de ganancia negativo, estaremos enfatizando las frecuencias alrededor de la frecuencia de corte, generando así una resonancia. El segundo detalle es que conforme la frecuencia de corte aumenta (y p se acerca a cero), la señal de salida del filtro conserva cada vez mas energía, por lo que al retroalimentarla hacia la entrada el filtro se vuelve inestable si la ganancia de retroalimentación es mayor o igual a uno. Existen varias maneras de compensar el factor de atenuación para evitar la inestabilidad; en este caso, simplemente multiplicamos el factor

de retroalimentación Q por la posición p del polo, lo cual es algo drástico pero funciona bien si no deseamos un filtro altamente resonante o auto-oscilante.

20.2.2 Línea de retardo con retroalimentación

Otro caso interesante de estudio es el de una línea de retardo con retroalimentación, lo cual tiene un gran número de aplicaciones en audio. Matemáticamente, se puede expresar mediante el siguiente sistema:

$$y[n] = w[n - d], \tag{20.2}$$

donde d es el tiempo de retardo y $w[n]$ es la suma de la señal de entrada $x[n]$ mas la salida anterior del sistema $y[n]$ multiplicada por un factor de ganancia R; es decir, $w[n] = x[n] + Ry[n]$. De esta manera, la Ecuación 20.2 se puede expander como

$$\begin{aligned} y[n] &= x[n-d] + Ry[n-d] \\ &= x[n-d] + Rx[n-2d] + R^2 y[n-2d] \\ &\vdots \\ y[n] &= \sum_{k=0}^{\infty} R^k x[n-(k+1)d]. \end{aligned}$$

En otras palabras, la retroalimentación en una línea de retardo simple genera un número infinito de repeticiones de la señal de entrada equiespaciadas en tiempo (donde el espaciamiento es igual al tiempo de retardo d) y cuya amplitud crece o decrece exponencialmente de acuerdo al factor de retroalimentación R. Por supuesto, por lo general se utilizan factores $|R| < 1$ para evitar que la amplitud de salida crezca indefinidamente.

La implementación es muy sencilla y no requiere mas que agregar una línea de código al pseudo-código que se presentó en la Práctica 14 para sumar la salida retroalimentada a la muestra de entrada que se almacena en el buffer del retardo. El siguiente pseudo-código ilustra lo anterior para el caso de un retardo simple:

```
buffer[indiceEscritura] = entrada;
salida = buffer[indiceLectura];
buffer[indiceEscritura] += R * salida;   //  <-- Aqui se aplica la retroalimentacion
indiceEscritura = (indiceEscritura + 1) % N;
indiceLectura = (indiceLectura + 1) % N;
```

Alternativamente, uno podría almacenar la salida anterior en una variable que mantenga su valor y aplicar la retroalimentación directamente en la primer línea del pseudo-código (con un retardo adicional de una muestra).

Una de las aplicaciones de las líneas de retardo retroalimentadas es la implementación de diversos efectos de audio, tales como eco, coro (chorus) y flanger. En esta práctica implementaremos un efecto de eco, mientras que los efectos de coro y flanger se estudiarán en la Práctica 21.

20.3 Objetivos didácticos

- Conocer algunas de las aplicaciones de la retroalimentación controlada
- Implementar la retroalimentación en bloques de procesamiento de aplicación específica
- Incorporar los bloques implementados en un sintetizador sustractivo

20.4 Material

- Una computadora con alguna de las siguientes combinaciones de software instalado:
  - Processing y Beads
  - Processing y Minim
  - GNU C++ (e.g., MinGW) y PortAudio
- Bocinas o audífonos estéreo

20.5 Procedimiento

En esta práctica se implementarán los dos bloques estudiados arriba: un filtro pasa-bajas resonante de cuarto orden, y una línea de retardo simple con retroalimentación. La retroalimentación siempre debe implementarse dentro del ciclo principal en la función callback que realiza el procesamiento, para evitar introducir retardos adicionales.

El filtro se implementará como una nueva UGen heredada de la clase `VCF`, aprovechando muchos de los métodos y miembros de datos ya existentes; de esta manera, solamente será necesario implementar la función callback y el constructor para el nuevo filtro. Para la línea de retardo retroalimentada, lo mejor será modificar directamente la clase `RetardoSimple` para agregar la nueva funcionalidad, lo cual solamente requiere unas cuantas líneas de código adicionales.

Para probar lo anterior, incorporaremos las nuevas UGens al sintetizador sustractivo elaborado en la Práctica 19, reemplazando el filtro pasa-bandas resonante `VCF` por el nuevo filtro pasa-bajas, y agregando un retardo retroalimentado al final de la cadena de audio (después del VCA) para añadir efectos de eco al sintetizador.

20.5.1 Implementación del filtro pasa-bajas resonante

Como se mencionó anteriormente, la clase `VCF` implementada en la Práctica 19 cuenta ya con todos los parámetros requeridos para el filtro pasa-bajas resonante (frecuencia de corte, resonancia, señal moduladora e índice de modulación), así como las funciones *get* y *set* correspondientes. Por lo tanto, lo mas sencillo será derivar la nueva UGen a partir de `VCF`. Dado que el filtro está inspirado en

el diseño de Moog, llamaremos a la nueva UGen `FiltroMoog`. El código es el siguiente:

```
public class FiltroMoog extends VCF {
  protected float ya1, ya2, ya3, ya4;

  public FiltroMoog(AudioContext context) {
    super(context);
  }

  public void calculateBuffer() {
    float w, y, a, b, omega;
    float[] in = bufIn[0];
    float[] out = bufOut[0];
    float[] mod = bufIn[1];
    omega = 2 * PI * frecuencia / context.getSampleRate();
    for (int i = 0; i < bufferSize; i++) {
      w = omega * pow(2, indiceModulacion * mod[i]);
      a = 1 - w;
      if (a < 0) a = 0;
      b = 1 - a;

      y = in[i] - Q * a * ya4;
      y = b * y + a * ya1; ya1 = y;
      y = b * y + a * ya2; ya2 = y;
      y = b * y + a * ya3; ya3 = y;
      y = b * y + a * ya4; ya4 = y;
      out[i] = y;
    }
  }
}
```

20.5.2 Implementación de la línea de retardo retroalimentada

Para incorporar la retroalimentación en la línea de retardo simple, agregaremos a la clase `RetardoSimple` un miembro de datos `float feedback` para almacenar el factor de retroalimentación R, así como sus correspondientes métodos *set* y *get*:

```
float feedback;

void setFeedback(float fb) {  feedback = fb; }

float feedback() { return feedback; }
```

Note que estos cambios también tendrán efecto en la clase derivada `RetardoVariable`, por lo que no será necesario agregar explícitamente estos miembros a esa clase.

También se debe modificar la función callback para agregar la línea que aplica la retroalimentación justo después de calcular la salida del retardo:

```
public void calculateBuffer() {
    float[] in = bufIn[0];
    float[] out = bufOut[0];
    if (buffer == null) return;

    for (int i = 0; i < bufferSize; i++) {
      // Almacena muestra de entrada en el buffer
      buffer[indiceEscritura] = in[i];

      // Calcula la muestra de salida interpolando el buffer
      out[i] = buffer[(int)indiceLectura];

      // Aplica retroalimentación (AGREGAR LA SIGUIENTE LINEA)
      buffer[indiceEscritura] += feedback * out[i];

      // Actualiza los indices de escritura y lectura
      indiceEscritura = (indiceEscritura + 1) % buffer.length;
      indiceLectura = (indiceLectura + 1) % buffer.length;
    }
  }
```

20.5.3 Incorporación de los nuevos elementos al sintetizador sustractivo

Una manera sencilla de probar las UGen implementadas es incorporándolas al sintetizador sustractivo elaborado en la Práctica 19. En el caso del filtro, no hay mas que reemplazar la UGen `VCF` de la práctica anterior con el nuevo filtro `FiltroMoog`. Para el efecto de eco, se requiere agregar elementos nuevos tanto a la arquitectura del sintetizador como a la interfaz de usuario.

20.5.4 Efectos de eco

El eco se produce cuando un sonido emitido por una fuente rebota contra alguna superficie lejana y regresa hacia la fuente para luego rebotar contra otra superficie y volver a hacer lo mismo. Cada vez que el sonido rebota contra una de las superficies, parte de su energía es absorbida por la superficie. Supongamos que nos encontramos cerca de la fuente de sonido; entonces escucharemos el sonido emitido por la fuente prácticamente al instante de ser emitido, y posteriormente escucharemos las repeticiones (ecos) de la señal original, donde cada repetición está cada vez mas degradada con respecto a la anterior. Si la distancia entre las superficies sobre las cuales rebota el sonido es suficientemente grande, el tiempo

entre repeticiones será lo suficientemente largo como para distinguir cada una de ellas como un sonido aparte, dando lugar al efecto de eco. Por el contrario, si la distancia entre las superficies es corta, las repeticiones estarán tan juntas en el tiempo que se confundirán con el sonido original, cambiando su timbre de diversas maneras. Estos últimos efectos se estudiarán mas a fondo en prácticas siguientes.

Un efecto simple de eco puede implementarse mediante una línea de retardo retroalimentada, a cuya salida se le suma la señal original sin retardo. El tiempo de retardo determina el tiempo entre las repeticiones o ecos del sonido generado, el cual está relacionado con el tamaño del espacio donde se generan los ecos (e.g., la distancia entre las superficies de rebote). Por lo general, los ecos tendrán una intensidad menor que la señal original, por lo que es común pasar la salida del retardo por un amplificador (usualmente de ganancia fija) que permita controlar, de manera general, la intensidad de los ecos, mientras que el factor de retroalimentación de la línea de retardo controla la rapidez con la que decae la intensidad de los ecos (o mejor dicho, la persistencia de éstos), y de manera indirecta controla el número de ecos producidos.

La siguiente arquitectura ejemplifica cómo aplicar un efecto de eco a la salida de una UGen llamada `fuente`, y rutear el sonido con efecto de eco hacia otra UGen llamada `destino`:

```
RetardoSimple retardo;
Amplificador ampRetardo;

retardo.addInput(fuente);
ampRetardo.addInput(retardo);
destino.addInput(fuente);
destino.addInput(ampRetardo);

retardo.setTiempo(1000); // fija el tiempo de retardo
retardo.setFeedback(0.5); // fija el coeficiente de retroalimentacion
ampRetardo.setGanancia(0.5); // fija la intensidad de los ecos
```

En este caso utilizamos la capacidad de Beads de conectar múltiples señales a una sola entrada, ya que Beads sumará automáticamente todas las señales conectadas, en este caso, a la UGen `destino`. En caso de usar Minim o C++, será necesario primero enviar las salidas de `fuente` y `ampRetardo` una UGen `Summer` o un mezclador, y conectar la salida del mezclador a la entrada de `destino`. En caso de usar un mezclador, uno podría omitir la UGen `ampRetardo` y controlar la intensidad de los ecos directamente en el mezclador.

20.6 Evaluación y reporte de resultados

1.- Graficar la respuesta en fase del filtro pasabajas de un polo $\measuredangle H_1(\omega_c)$ en función de la frecuencia de corte ω_C, o en función del polo p. Verifique que para frecuencias de corte bajas ($p > 0.8$), la respuesta en fase es aproximadamente $-\pi/4$, mientras que para frecuencias de corte altas la respuesta en fase se acerca a cero.

2.- Agregue un efecto de eco al sintetizador FM elaborado en la Práctica 18.

3.- Agregue retroalimentación al retardo fraccional variable (UGen `RetardoVariable`). Note que, dado que la clase `RetardoVariable` hereda de `RetardoSimple`, solamente será necesario modificar la función callback de `RetardoVariable`.

20.7 Retos

Una arquitectura interesante para un procesador de efectos de audio es aplicar primero algún tipo de distorsión (lo cual incrementará el contenido armónico de la señal), seguido de un filtro pasa-bajas resonante, y por último un efecto de eco. Aplique esa cadena de procesamiento a la entrada física de audio (micrófono o entrada auxiliar) y elabore una interfaz gráfica para controlar todos los parámetros de los efectos (ganancia del waveshaper; frecuencia de corte y resonancia del filtro; tiempo de retardo, retroalimentación e intensidad del eco). Si se cuenta con una interfaz de audio adecuada, es posible incluso aplicar los efectos a una guitarra u otro instrumento. Para un reto aún mayor, agregue uno o más LFOs para modular algunos de los parámetros como la frecuencia de corte y el tiempo de retardo.

20.8 Conclusiones

Redacte en el recuadro sus conclusiones acerca de los conceptos aprendidos, las actividades realizadas y los objetivos planteados en esta práctica.

Capítulo 21

Efectos basados en retardos

Nombre del estudiante	Calificación

21.1 Relación con los programas de estudio

Materia	Unidad y tema
Procesamiento Digital de Señales (IE / IT / IB)	2.3.- Propiedades de la Transformada de Fourier 6.2.- Consideraciones para el diseño de filtros 6.4.- Diseño de filtros IIR
Procesamiento de Señales de Audio (IE)	2.4.- Filtros de orden superior 2.10.- Generación de ruido con color 4.1.- Eco y retardos 4.2.- Efectos de coro y flanger 4.4.- Corrimiento en frecuencia y vibrato 5.5.- Aplicaciones musicales en arquitecturas diversas

21.2 Introducción

Varios efectos comúnmente utilizados en audio pueden ser implementados mediante líneas de retardo. En la Práctica 14 se describe el efecto de introducir modulación al tiempo de retardo; o mas precisamente, al incremento en el apuntador de lectura del buffer. Esta modulación da lugar a un efecto tipo Doppler donde la frecuencia de la señal procesada aumenta conforme el tiempo de retardo disminuye, y viceversa. Al final de esa práctica, se propuso como reto implementar un efecto de *vibrato* (modulación cíclica de la frecuencia de un

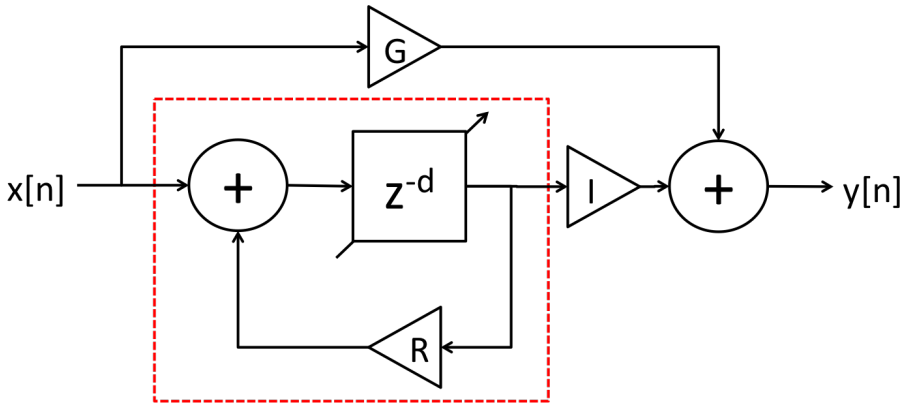


Figura 21.1: Diagrama general de un efecto basado en una línea de retardo variable. La sección dentro de la línea roja punteada representa una línea de retardo variable retroalimentada, que corresponde a la UGen `RetardoVariable` (una vez que se le incorpora la retroalimentación).

sonido) aplicando una modulación cíclica al tiempo de retardo. Por supuesto, en un sintetizador es posible (y mas sencillo) aplicar vibrato modulando directamente la frecuencia de los osciladores; sin embargo, la arquitectura basada en una línea de retardo permite aplicar vibrato a cualquier señal.

La salida de un efecto de vibrato puede considerarse como una versión ligeramente desafinada de la señal original. Si combinamos ambas señales, obtendremos un efecto de *coro*, como cuando dos cantantes cantan el mismo pasaje simultáneamente, pero con ligeras diferencias en su entonación. Por supuesto, uno podría simular mas de dos voces añadiendo más líneas de retardo, pero también es posible obtener un efecto similar retroalimentando el retardo [29].

En la Práctica 20 se implementó un efecto de eco basado en una línea de retardo con retroalimentación. Para este efecto, a la señal original se le suma la señal retardada para simular la reflexión de una onda acústica sobre una superficie, y se retroalimenta la salida del retardo para simular las reflexiones subsecuentes. Cuando el tiempo de retardo es lo suficientemente grande, el sonido original y sus reflexiones se perciben como sonidos separados, lo cual corresponde a la simulación del eco en espacios muy amplios. Sin embargo, conforme el tiempo de retardo se hace pequeño, digamos del mismo orden que el periodo fundamental de la señal procesada, se simulan espacios muy pequeños como las cajas de resonancia de los instrumentos acústicos. En este caso, las reflexiones ya no se distinguen del sonido original sino que modifican sus propiedades.

21.2.1 Análisis espectral de una línea de retardo

Para entender mejor el efecto que imparte una línea de retardo retroalimentada en el timbre de un sonido, consideremos el sistema mostrado en la Figura 21.1, al que llamaremos *sistema general de retardo*. La salida de este sistema consiste en

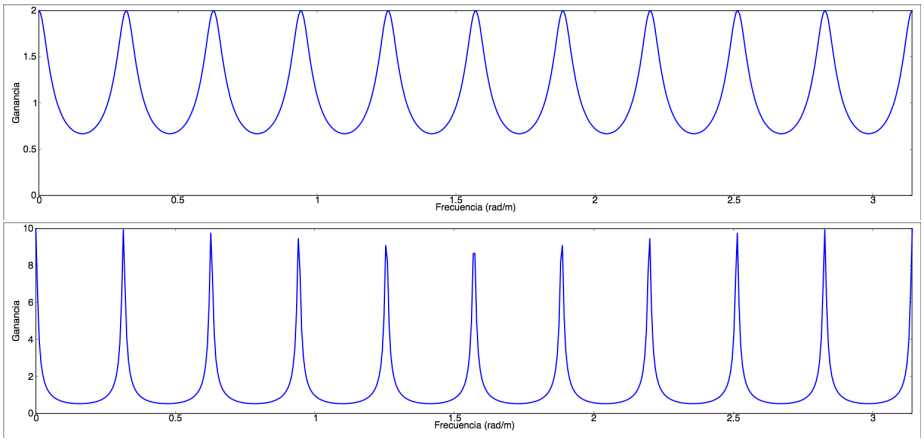


Figura 21.2: Respuesta en frecuencia de un sistema de retardo retroalimentado (filtro peine) con un tiempo de retardo fijo de $d = 20$ muestras y factores de retroalimentación de $R = 0.5$ (arriba) y $R = 0.9$ (abajo).

la suma de la señal de entrada (atenuada por un factor G) y la salida (también atenuada, por un factor I) de una línea de retardo retroalimentada con tiempo de retardo d y factor de retroalimentación R. La línea de retardo es fraccional y variable (lo cual se indica con una flecha diagonal en el bloque de retardo), por lo que el tiempo de retardo puede ser modulado mediante una señal externa. Este sistema es muy similar al efecto de eco implementado en la Práctica 20, excepto que en aquél sistema se utilizó un retardo de tiempo fijo y entero, y la ganancia G de la señal de entrada era igual a uno.

Consideremos primero la señal de entrada al retardo (bloque rojo en la Figura 21.1). Llamémosle $v[n]$ y $w[n]$ a la entrada y salida del retardo, respectivamente; por lo tanto, $v[n] = x[n] + Rw[n]$ y $w[n] = v[n - d]$. Entonces:

$$\begin{aligned} w[n] &= v[n-d] \\ &= x[n-d] + Rw[n-d] \\ &= x[n-d] + Rv[n-2d] \\ &= x[n-d] + R(x[n-2d] + Rw[n-2d]) \\ &= x[n-d] + Rx[n-2d] + R^2 w[n-2d] \\ &\vdots \\ &= \sum_{k=1}^{\infty} R^{k-1} x[n-kd] \end{aligned}$$

Por lo tanto, podemos expresar el sistema general de retardo como

$$y[n] = Gx[n] + I \sum_{k=1}^{\infty} R^{k-1} x[n-kd]. \tag{21.1}$$

Por simplicidad, consideremos el caso particular cuando $I = R$ y $G = 1$, para el cual tenemos que

$$y[n] = \sum_{k=0}^{\infty} R^k x[n - kd].$$

La respuesta al impulso $h[n]$ de este sistema consiste en un tren de impulsos equiespaciados por una distancia d y atenuados exponencialmente:

$$h[n] = \sum_{k=0}^{\infty} R^k \delta[n - kd].$$

Finalmente, aplicando algunas propiedades de la Transformada de Fourier se llega a que la respuesta en frecuencia $H(\omega)$ del sistema está dada por

$$H(\omega) = \sum_{k=0}^{\infty} R^k \exp\{-j\omega kd\}. \tag{21.2}$$

La magnitud de la respuesta en frecuencia $|H(\omega)|$ de este sistema se ilustra en la Figura 21.2 para el caso con un tiempo de retardo de $d = 20$ muestras y factores de retroalimentación $R = 0.5$ y $R = 0.9$. Como se aprecia en la figura, la respuesta en frecuencia consiste de una serie de picos angostos equiespaciados por una distancia $2\pi/d$, cuya altura está en relación directa al factor de retroalimentación. Debido a la forma de la respuesta en frecuencia, a este sistema se le conoce también como un *filtro peine* (*comb filter*, en inglés).

Para este caso, donde $I = R$, la ganancia en los picos puede llegar a ser considerablemente mayor que uno, lo que se traduce en resonancias. De manera más general, uno puede utilizar I para compensar la alta ganancia y evitar distorsiones o problemas de estabilidad.

Cuando d es relativamente pequeño, como en la Figura 21.2, el espaciamiento entre los picos del filtro peine es suficientemente grande como para que las crestas y valles de $|H(\omega)|$ estén bien definidas y el sistema actúe realmente como un filtro, alterando el timbre de la señal de entrada. Por otra parte, si d es grande (digamos, del mismo orden que la frecuencia de muestreo) entonces el espaciamiento entre los picos del peine será demasiado pequeño, por lo que se pierde la distinción entre crestas y valles, y el timbre de la señal procesada no se ve afectado.

21.2.2 Retardos naturales

Cuando una onda acústica choca con una superficie, parte de su energía es reflejada y otra parte es absorbida por la superficie. Por lo general, las frecuencias altas son absorbidas en mayor medida que las frecuencias bajas, por lo que cada vez que la onda es reflejada, pierde contenido armónico; esto al margen de las frecuencias que pudieran enfatizarse debido a la resonancia. Se dice entonces que el espacio en el cual el sonido es reflejado le imparte un cierto *color* al sonido.

Podemos simular este fenómeno fácilmente colocando un filtro pasa-bajas de un polo justo a la salida del retardo en la Figura 21.1, de manera que el filtro afecte tanto al lazo de retroalimentación como a la señal que se dirige a la salida. La implementación no requiere mas que agregar un par de líneas de código a la función callback, y agregar métodos *set* y *get* para manipular el polo del filtro. A continuación se presenta el pseudo-código para procesar cada muestra en una línea de retardo variable con retroalimentación, modulación del tiempo de retardo y filtrado:

```
// Almacena muestra de entrada en el buffer
buffer[indiceEscritura] = in[i];

// Calcula la muestra de salida interpolando el buffer
izq = int(indiceLectura);
frac = indiceLectura - izq;
der = (izq + 1) % buffer.length;
out[i] = (1 - frac) * buffer[izq] + frac * buffer[der];

// Aplica filtro pasa-bajas a la salida del retardo
out[i] = (1 - polo) * out[i] + polo * anterior;
anterior = out[i];

// Aplica retroalimentación a la entrada
buffer[indiceEscritura] += R * out[i];

// Actualiza los indices de escritura y lectura (con modulacion)
indiceEscritura = (indiceEscritura + 1) % buffer.length;

g = indiceModulacion * mod[i];
indiceLectura = (indiceLectura + 1 - g + buffer.length) % buffer.length;
```

En este ejemplo, las variables `polo` y `anterior` son miembros de la clase, por lo que conservan su valor entre llamadas a la función callback. El valor del polo p debe estar entre 0 y 1, y puede interpretarse como el grado de atenuación o absorción de altas frecuencias (comúnmente llamado *damping*).

21.2.3 Efecto basado en retardos

El sistema general de retardo (Figura 21.1) es capaz de producir algunos efectos populares, como los que se describen a continuación. Algunos requieren el uso de una señal moduladora para modular el tiempo de retardo, que por lo general es un oscilador de baja frecuencia (LFO) o ruido filtrado por un pasa-bajas de un polo con frecuencia de corte en el rango infrasónico (< 20 Hz); en estos casos asumiremos que la señal moduladora varía entre -1 y 1, y la amplitud de modulación está determinada completamente por el índice de modulación.

- *Vibrato.-* Como se ha mencionado, el vibrato consiste en aplicar una ligera modulación en frecuencia a la señal procesada. Esto se logra pasando la

señal por un retardo variable no retroalimentado ($R = 0$) cuyo tiempo de retardo base es alrededor de $d = 20$ms, y es modulado de manera cíclica, por ejemplo mediante un oscilador de baja frecuencia ($\sim$ 4 Hz), con un índice de modulación alrededor de 0.02 - 0.05. Dado que solo se desea escuchar la salida del retardo, se elige $G = 0$, $I = 1$ y $p = 0$ (sin absorción de altas frecuencias).

- *Coro (Chorus).-* El efecto de coro se obtiene básicamente combinando la salida del vibrato con la señal original con la señal, por lo que se pueden utilizar los mismos parámetros del vibrato, excepto G e I, los cuales toman ambos el mismo valor entre 0.5 y 0.7. Para este efecto es preferible utilizar el ruido filtrado como señal moduladora, para obtener un resultado mas orgánico. El efecto se puede enfatizar agregando retroalimentación positiva o negativa, pero no demasiada ($|R| < 0.7$).

- *Resonador.-* La resonancia se obtiene utilizando un tiempo de retardo fijo (sin modulación) y corto ($d < 20$ ms), con un coeficiente de retroalimentación alto ($R > 0.9$). Incluso puede utilizarse $R = 1$ si el sistema incorpora absorción de altas frecuencias ($p > 0.5$). De hecho, p es de gran utilidad para controlar el color del sonido resonante. Los valores de G e I pueden utilizarse en una configuración cross-fade ($G = 1 - I$) para variar entre la señal de entrada y el sonido resonante.

- *Flanger.-* Este efecto es similar al efecto de coro, pero utilizando tiempos de retardo mas cortos ($d < 20$) y una mayor retroalimentación ($R > 0.7$) para generar resonancias. Al igual que en el efecto de coro, se aplica modulación del tiempo de retardo (por lo general usando un LFO con frecuencia menor a 1 Hz), lo que ocasiona que las frecuencias resonantes varíen con el tiempo.

- *Eco.-* El eco se logra utilizando un tiempo fijo de retardo (sin modulación) y considerablemente mas largo ($d > 100$ ms) de manera que el oído es capaz de distinguir las distintas copias de la señal. El factor de retroalimentación R controla el número de ecos, mientras que I controla la intensidad general de los ecos con respecto a la señal original. La absorción de altas frecuencias ($p > 0$) da como resultado un sonido mas natural, ocasionando que los ecos se apaguen mas rápidamente.

21.3 Objetivos didácticos

- Comprender el comportamiento en frecuencia de una línea de retardo retroalimentada

- Explorar las aplicaciones de las líneas de retardo como procesadores de efectos

- Implementar un procesador de efectos basado en una línea de retardo para procesar sonidos externos

21.4 Material

- Una computadora con alguna de las siguientes combinaciones de software instalado:
  - Processing y Beads
  - Processing y Minim
  - GNU C++ (e.g., MinGW) y PortAudio
- Bocinas o audífonos estéreo
- Micrófono o dispositivo para reproducir audio (smartphone, reproductor de MP3, etc)
- Cable de audio para conectar dispositivo a la entrada auxiliar de la computadora (usualmente un cable estereo con plugs de 1/8").

21.5 Procedimiento

En esta práctica se tiene como objetivo implementar un procesador de efectos basado en el sistema general de retardo para procesar señales externas (desde la entrada física de audio) y escuchar los resultados a través de las bocinas de la computadora o audífonos (en caso de que la entrada externa sea el micrófono). El sistema incluirá la generación de dos señales de baja frecuencia para modular el tiempo de retardo: un oscilador y ruido filtrado. Finalmente, se implementará una interfaz gráfica para manipular los parámetros del procesador de efectos, con el objetivo de lograr los efectos descritos arriba.

21.5.1 Implementación del retardo natural

Se sugiere implementar un bloque de retardo variable con retroalimentación y absorción de altas frecuencias como una nueva UGen, posiblemente llamada `RetardoNatural`, heredada de `RetardoVariable`. A la nueva clase habrá que agregar un miembro `float polo` para conservar la posición del polo, y un miembro `float anterior` para conservar el valor de la muestra de salida anterior del filtro. Además será útil incorporar métodos `setPolo()` y `getPolo()` para manipular el polo, y será necesario reimplementar la función callback, copiando la de `RetardoVariable` y agregando el código necesario para aplicar el filtrado a la salida del retardo.

21.5.2 Arquitectura del procesador de efectos

El siguiente paso es implementar el sistema general de retardo (Figura 21.1) con sus respectivos moduladores. Para tener mayor control sobre el efecto de los moduladores, la señal de cada uno de ellos se pasará por un amplificador de ganancia fija, y las salidas de estos amplificadores se conectarán a la entrada de

modulación del retardo. De esta manera incluso será posible combinar la acción de ambos moduladores.

El sistema se compone de los siguientes módulos (UGens), los cuales serán declarados de manera global:

```
UGen entrada;
Amplificador promedio;
Amplificador ampEntrada;
RetardoNatural retardo;
Amplificador ampRetardo;

Senoidal lfo;
Amplificador ampLFO;
RuidoBlanco ruido;
Filtro1P filtroRuido;
Amplificador ampRuido;
```

Las primeras cinco UGens conforman el sistema general de retardo, donde la señal de entrada proviene de la entrada física de audio. Dado que la entrada es estéreo, utilizaremos un amplificador de ganancia fija (UGen `promedio`) para promediar los dos canales de la entrada y obtener una señal mono-canal; simplemente conectaremos ambos canales a la entrada del amplificador `promedio` y le asignaremos una ganancia fija de 0.5. Los amplificadores `ampEntrada` y `ampRetardo` son los que permitirán atenuar, respectivamente, la señales de entrada y la salida del retardo con sus respectivos factores G e I. Las salidas de estos amplificadores se conectarán directamente a la salida física de audio.

El siguiente bloque de UGens corresponde a las señales de modulación para el tiempo de retardo. Estas consisten en un oscilador senoidal simple `lfo` cuya amplitud es controlada por `ampLFO`, y una fuente de ruido `ruido` la cual será procesada por un filtro de un polo `filtroRuido` y atenuada por `ampRuido`. Las salidas de `ampLFO` y `ampRuido` se conectarán a la entrada de modulación del retardo.

Note que la mitad de los módulos requeridos son amplificadores. Ya sea de ganancia fija o variable, los amplificadores son una de las herramientas mas útiles ya que permiten controlar la intensidad tanto de los sonidos como de las modulaciones que se aplican.

A continuación se muestra el código para construir e interconectar las UGens, el cual debe ir dentro de la función `setup()`:

```
  // crear UGens
  ac = new AudioContext();
  promedio = new Amplificador(ac, 1);
  ampEntrada = new Amplificador(ac, 1);
  retardo = new RetardoNatural(ac, 10000);
  ampRetardo = new Amplificador(ac, 1);
  lfo = new Senoidal(ac, 1);
  ampLFO = new Amplificador(ac, 1);
```

```
ruido = new RuidoBlanco(ac, 1);
filtroRuido = new Filtro1P(ac, 1);
ampRuido = new Amplificador(ac, 1);

// Realizar conexiones para el sistema general de retardo
entrada = ac.getAudioInput();
promedio.addInput(0, entrada, 0);
promedio.addInput(0, entrada, 1);
promedio.setGanancia(0.5);
ampEntrada.addInput(promedio);
retardo.setEntrada(promedio);
ampRetardo.addInput(retardo);

// Realizar conexiones para modulación del tiempo de retardo
ampLFO.addInput(lfo);
filtroRuido.addInput(ruido);
ampRuido.addInput(filtroRuido);
retardo.setModulador(ampLFO);
retardo.setModulador(ampRuido);
retardo.setIndice(0.1);

ac.out.addInput(ampEntrada);
ac.out.addInput(ampRetardo);
ac.start();
```

21.5.3 Interfaz gráfica

Para poder manipular libremente los parámetros del procesador de efectos, se sugiere implementar una interfaz gráfica como la que se muestra en la Figura 21.3. Los controles que presenta la interfaz son los siguientes:

- *Entrada* - Ganancia G de la señal de entrada a la salida del sistema (rango de 0 a 1).
- *Retardo* - Ganancia I de la señal retardada a la salida del sistema (rango de 0 a 1).
- *Rango* - Selector de rango para el tiempo de retardo (valores posibles: 0, 1, 2).
- *Tiempo* - Tiempo de retardo, dentro del rango especificado (de 0 a 100).
- *Retroalimentación* - Factor de retroalimentación R (rango de -1 a +1).
- *Absorción* - Posición p del polo para el filtro pasa-bajas (rango de 0 a 1).
- *LFO Frecuencia* - Frecuencia del LFO (rango de 0 a 4 Hz).

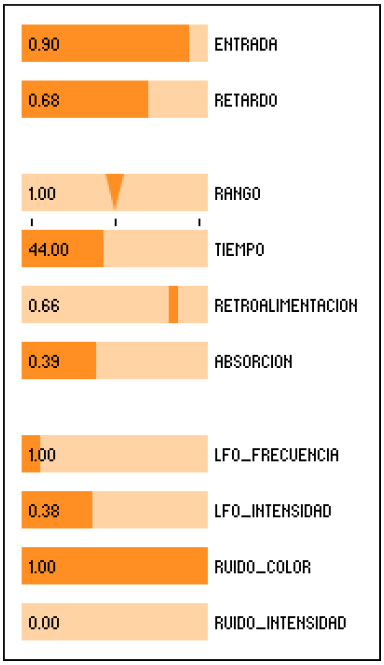


Figura 21.3: Interfaz gráfica del procesador de efectos basado en el sistema general de retardo.

- *LFO Intensidad* - Amplitud del LFO (rango de 0 a 1, pero este valor será multiplicado por el índice de modulación del retardo, cuyo valor se fija en 0.1).

- *Ruido Color* - Ancho de banda del ruido de baja frecuencia. El rango es de 0 a 1, pero este valor se multiplica por 0.001 y se le suma 0.999, para tener frecuencias de corte en el rango infrasónico.

- *Ruido Intensidad* - Amplitud del ruido filtrado (rango de 0 a 1, pero este valor será multiplicado por el índice de modulación del retardo, cuyo valor se fija en 0.1).

Aunque son nueve parámetros, la interfaz presenta diez controles debido a que dos de ellos (rango y tiempo) controlan el tiempo de retardo. Esto es debido a que el tiempo de retardo tiene un rango muy amplio que va desde pocos milisegundos hasta mas de un segundo, por lo que es difícil hacer ajustes finos en el rango completo. La solución aquí propuesta es tener un selector de rango con tres opciones: rango 0 (de 0 a 100 ms), rango 1 (de 0 a 1000 ms) y rango 2 (de 0 a 10 s). De esta manera, el tiempo de retardo está dado por

$$d = \text{tiempo} \cdot 10^{\text{rango}}.$$

Una vez implementado el programa, pruebe aplicando los distintos efectos a su propia voz (a través del micrófono) y escuchando el resultado con audífonos.

21.6 Evaluación y reporte de resultados

1.- Nombre tres tipos de efectos que pueden implementarse utilizando líneas de retardo y para cada uno de ellos describa tanto su efecto acústico como las cuestiones técnicas en su implementación.

2.-Considere el análisis espectral del sistema general de retardo presentado en la Introducción. Por simplicidad, este análisis se realizó para el caso con $G = 1$ e $I = R$. Cómo queda la respuesta en frecuencia $H(\omega)$ del sistema para el caso general (G e I arbitrarias)? Qué papel juegan G e I en el espectro de la señal procesada?

3.- La manera en que hemos aplicado modulación al tiempo de retardo, modulando mas bien la velocidad de lectura del buffer, tiene un inconveniente: después de un cierto tiempo, el tiempo de retardo (la distancia entre el apuntador de escritura y el de lectura) comienza a desviarse de su valor esperado debido a la acumulación de pequeños errores. Proponga y pruebe un esquema para resolver este problema.

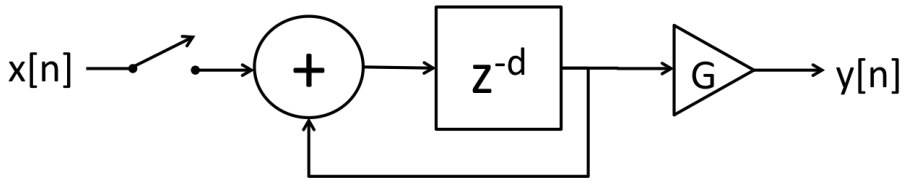


Figura 21.4: Arquitectura básica de un looper. El interruptor que deja pasar la señal de entrada hacia el retardo se puede implementar mediante un amplificador cuya ganancia es cero o uno, para respectivamente abrir o cerrar el interruptor.

21.7 Retos

Un *looper* es un dispositivo que puede grabar un fragmento de una señal de audio en un buffer, y posteriormente reproducir el fragmento una y otra vez, de manera cíclica. Mas aún, mientras el *loop* se reproduce, es posible continuar grabando la señal de entrada, sumándola a la señal ya existente en el buffer. Esto permite ir agregando capas de sonido nuevas al *loop*, sin perder las que ya se han grabado. La longitud del *loop* se determina una vez hecha la primer grabación.

Es posible implementar un looper utilizando una línea de retardo simple retroalimentada (UGen `RetardoSimple`), como se muestra en la Figura 21.4. El retardo tiene un factor de retroalimentación fijo $R = 1$, de manera que la señal de salida vuelve siempre a grabarse en el buffer, y un interruptor a la entrada que permite o evita que se grabe la señal de entrada. Este interruptor puede emularse con un amplificador con ganancia igual a cero o uno. A la salida del retardo se coloca otro amplificador de ganancia fija simplemente para controlar el volumen de salida y evitar posibles saturaciones.

El control del looper es muy simple y cuenta solamente con dos controles: un botón que activa o desactiva el modo de grabación (alternando el interruptor), y un control para manipular la ganancia a la salida del retardo. Es importante que cuando se realiza la primer grabación, se fije el tiempo de retardo al tiempo transcurrido entre la activación y la desactivación del interruptor. Finalmente, se puede agregar otro botón para reinicializar el looper, borrando el buffer existente y reestableciendo la próxima grabación como la primera.

21.8 Conclusiones

Redacte sus conclusiones acerca de los conceptos aprendidos, las actividades realizadas y los objetivos planteados en esta práctica.

Capítulo 22

Síntesis por modelado físico

Nombre del estudiante	Calificación

22.1 Relación con los programas de estudio

Materia	Unidad y tema
Procesamiento Digital de Señales (IE / IT / IB)	6.4.- Diseño de filtros IIR
Procesamiento de Señales de Audio (IE)	2.4.- Filtros de orden superior 3.8.- Modelado físico 5.5.- Aplicaciones musicales en arquitecturas diversas

22.2 Introducción

Una línea de retardo puede simular la manera en que se propaga una onda acústica a una distancia d de la fuente, siendo $d = ct$, donde t es el tiempo de retardo y c es la velocidad del sonido. Si la fuente de sonido está ubicada en un espacio cerrado, entonces la onda se reflejará en las superficies rebotando una y otra vez, lo cual puede modelarse como un lazo de retroalimentación en la línea de retardo. Dado que parte de la energía de la onda es absorbida por las superficies, entonces suele aplicarse un factor de retroalimentación con magnitud menor a uno. Mas aún, es posible modelar una absorción de energía dependiente de la frecuencia insertando un filtro en el lazo de retroalimentación. A partir de estas nociones, fue posible modelar algunos efectos como ecos y vibrato (una aplicación del efecto Doppler) a partir de una descripción *física* de los mismos.

Similarmente, existe un rango muy amplio de métodos de síntesis de sonido basados en modelos computacionales obtenidos a partir de descripciones físicas de los instrumentos que se pretenden emular, entre los que se incluyen modelos

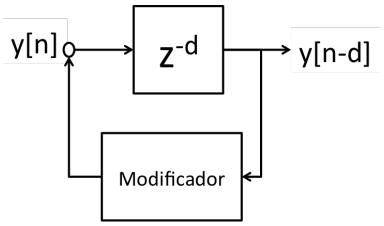


Figura 22.1: Modelo digital de Karplus-Strong de una cuerda tensa vibrando libremente con pérdida de energía.

de cuerdas pulsadas, cuerdas frotadas, instrumentos de viento, percusiones y otros. De manera general, a estos métodos se les denomina métodos de *síntesis por modelado físico*, aunque los modelos pueden diferir mucho unos de otros. Esto es en contraste con los métodos "clásicos" como la síntesis aditiva, sustractiva y de frecuencia modulada, los cuales se consideran mas bien como *métodos de modelado espectral* [19].

En esta práctica estudiaremos uno de los primeros y mas básicos métodos de síntesis por modelado físico, el cual modela el sonido de cuerdas pulsadas (e.g., guitarra o arpa) con mucho mayor realismo que los métodos de síntesis clásicos.

22.2.1 Algoritmo de Karplus-Strong

Considere una cuerda tensa entre dos extremos fijos y un punto p sobre la cuerda. Suponga que la cuerda es excitada, por ejemplo, pulsándola en el punto p y soltándola. El punto p comenzará a oscilar con una frecuencia fundamental f, desplazándose de manera perpendicular al eje de la cuerda a una posición $y_p(t)$ (donde t denota el tiempo) con respecto a su posición de reposo. Debido a que los extremos de la cuerda están fijos y a que la cuerda tiene un cierto grado de elasticidad, el punto p solo puede desplazarse una cierta distancia antes de "rebotar" en la dirección opuesta, de manera similar a como una onda acústica rebota en un espacio cerrado. Si la energía se conserva totalmente, uno puede escribir $y_p(t) = y_p(t - T)$, donde $T = 1/f$ es el periodo de oscilación. Esto da como resultado un sistema sin entradas que puede funcionar como oscilador cuando el buffer de retardo contiene ya alguna señal almacenada, similar al oscilador basado en tabla de ondas. Sin embargo, el sonido que resulta de este oscilador es totalmente estático, ya que mantiene su timbre, intensidad y frecuencia a lo largo del tiempo.

En 1983, Kevin Karplus y Alex Strong propusieron un nuevo modelo en el cual se agrega un modificador a la señal retardada, lo cual puede expresarse como $y_p(t) = M(y_p(t - T))$, donde M representa al bloque modificador. Este sistema se puede construir a partir de una línea de retardo retroalimentada como la que se muestra en la Figura 22.1, donde el tiempo de retardo d equivale al periodo de oscilación dado en muestras, por lo que $d = f_m/f$ para una frecuencia de muestreo f_m, y el modificador se inserta en el lazo de retroalimentación. En

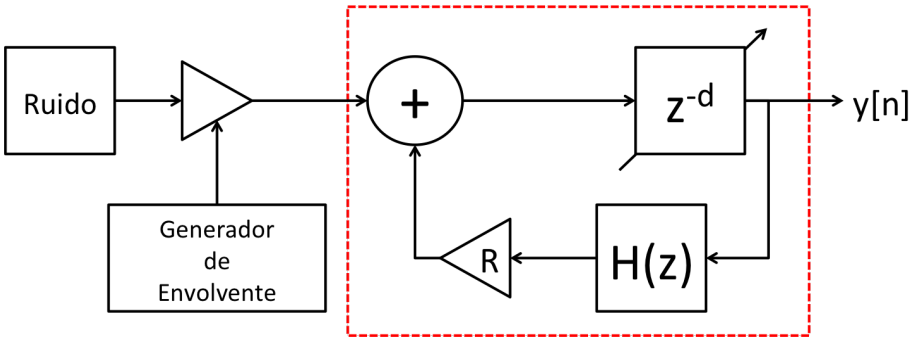


Figura 22.2: Modelo basado en el algoritmo de Karplus-Strong modificado con un filtro IIR de un polo y un atenuador en el lazo de retroalimentación, así como una fuente de ruido y un VCA controlado por una envolvente como señal de entrada para excitar el sistema.

la propuesta original de Karplus y Strong, el bloque modificador consiste de un simple filtro FIR cuyo kernel es $[0.5, 0.5]$; es decir, $y[n] = (y[n-d]+t[n-d-1])/2$, cuya implementación era extremadamente sencilla y eficiente. Este filtro FIR corresponde a un filtro pasa-bajas, lo cual ocasionaba que la salida del oscilador perdiera contenido armónico y se atenuara gradualmente. El sonido resultante se asemeja al de una cuerda vibrando libremente, pero con pérdida gradual de energía [30].

La velocidad de las computadoras actuales permite experimentar con otros tipos de modificadores bajo el esquema Karplus-Strong. Por ejemplo, podemos reemplazar el filtro FIR con coeficientes constantes por un filtro IIR de un polo con el que tengamos control de la frecuencia de corte, y por lo tanto de la velocidad con la que decaen los armónicos del oscilador. Podemos también agregar una etapa de amplificación para tener un mayor control de la ganancia de retroalimentación, ya que una ganancia unitaria está justo al borde de la inestabilidad.

Como segunda mejora importante, podemos reemplazar la línea de retardo fija por una línea de retardo fraccional y variable. El manejo de tiempos de retardo fraccionarios permitirá entonar el oscilador a cualquier frecuencia deseada, ya que el tiempo de retardo $d = f_m/f$ rara vez será entero. Por otra parte, si se desea modular la frecuencia f del oscilador, esto se traducirá en un tiempo de retardo variable.

Finalmente, queda la cuestión de cómo modelar la excitación inicial del sistema; es decir, cómo introducir una señal inicial en el buffer del retardo para producir las oscilaciones a la salida del sistema. Karplus y Strong propusieron inicializar el buffer con ruido al comienzo de cada nota, ya que el ruido presenta aproximadamente la misma energía en todos los armónicos. En la actualidad es mas común excitar el sistema enviando una ráfaga de ruido de corta duración hacia la entrada de la línea de retardo.

Con estas consideraciones, se llega al sistema que se muestra en la Figura 22.2, y que implementaremos en esta práctica. Dentro de la línea roja punteada se tiene una línea de retardo fraccional variable retroalimentada y con absorción de altas frecuencias (mediante el filtro $H(z)$), que es prácticamente la que se implementó en la Práctica 21 como la UGen `RetardoNatural`. Sin embargo, la modulación en `RetardoNatural` se aplica directamente al tiempo de retardo, simulando el efecto Doppler; mientras que en una implementación de Karplus-Strong, tiene mas sentido aplicar la modulación a la frecuencia del oscilador, de manera exponencial. Por otra parte, la excitación del sistema (elementos fuera del rectángulo rojo) se produce mediante una fuente de ruido cuya amplitud es modulada por una envolvente, la cual por lo general consta solamente de una etapa de decaimiento con corta duración.

22.3 Objetivos didácticos

- Visualizar las líneas de retardo como un mecanismo para modelar la propagación de ondas en diversos contextos.
- Comprender el modelo Karplus-Strong para la simulación de cuerdas pulsadas.
- Implementar el algoritmo Karplus-Strong y explorar algunas de sus modificaciones.

22.4 Material

- Una computadora con alguna de las siguientes combinaciones de software instalado:
  - Processing y Beads
  - Processing y Minim
  - GNU C++ (e.g., MinGW) y PortAudio
- Bocinas o audífonos estéreo

22.5 Procedimiento

Aunque técnicamente contamos ya con todos los elementos para programar el algoritmo Karplus-Strong (fuente de ruido, VCA, generador de envolvente y línea de retardo), en una implementación mas práctica es deseable incorporar la modulación exponencial de frecuencia. Para lograr esto, el camino mas corto consiste en crear una nueva UGen, heredada de `RetardoNatural`, que incorpore la frecuencia como parámetro fundamental, junto con sus respectivas funciones *set* y *get*, y donde se modifique la función callback para estimar la frecuencia

instantánea (tomando en cuenta la modulación exponencial) y el tiempo de retardo correspondiente.

Esto se ilustra con siguiente código:

```
public class KarplusStrong extends RetardoNatural {
  float frecuencia;

  float frecuencia() { return frecuencia; }
  void setFrecuencia(float f) { frecuencia = f; }

  public KarplusStrong(AudioContext ac) {
    super(ac, 16384);
    frecuencia = 440;
  }

  public void calculateBuffer() {
    float[] in = bufIn[0];
    float[] mod = bufIn[1];
    float[] out = bufOut[0];
    float frac, f, d = 0;
    int izq, der;
    if (buffer == null) return;

    for (int i = 0; i < bufferSize; i++) {
      // Almacena muestra de entrada en el buffer
      buffer[indiceEscritura] = in[i];

      // Calcula frecuencia instantanea y tiempo de retardo
      f = frecuencia * pow(2, indiceModulacion * mod[i]);
      d = (f > 0) ? context.getSampleRate() / f : 0;
      indiceLectura = (indiceEscritura - d + buffer.length) % buffer.length;

      // Calcula la muestra de salida interpolando el buffer
      izq = int(indiceLectura);
      frac = indiceLectura - izq;
      der = (izq + 1) % buffer.length;
      out[i] = (1 - frac) * buffer[izq] + frac * buffer[der];
      out[i] = (1 - polo) * out[i] + polo * anterior;
      anterior = out[i];

      // Aplica retroalimentación
      buffer[indiceEscritura] += feedback * out[i];

      // Actualiza el indices de escritura
      indiceEscritura = (indiceEscritura + 1) % buffer.length;
    }
```

```
  }
}
```

Note que el constructor de `KarplusStrong` establece un tiempo máximo de retardo de 16384 muestras, lo cual significa que la frecuencia mas baja que puede generar el oscilador es de 2.7 Hz para una frecuencia de muestreo de 44100 Hz. Esto es más que suficiente para cubrir el rango audible, incluso considerando los efectos de la modulación.

Para completar el sistema, incorporamos los módulos que generarán la ráfaga de ruido que excitará al modelo de cuerda:

```
AudioContext ac;
RuidoBlanco ruido;
VCA vca;
Rampa env;
KarplusStrong ks;

void setup() {
  size(800, 600);

  ac = new AudioContext();

  ruido = new RuidoBlanco(ac, 1);
  vca = new VCA(ac);
  env = new Rampa(ac);
  ks = new KarplusStrong(ac);

  vca.setEntrada(ruido);
  vca.setModulador(env);
  vca.setIndice(1);
  ks.setEntrada(vca);
  ks.setFeedback(0.99);

  ac.out.addInput(ks);
  ac.start();
}
```

Para disparar una nota, se debe fijar la frecuencia del oscilador, y posteriormente disparar la envolvente que controla la amplitud de la ráfaga de ruido. Por ejemplo:

```
void disparaNota(int n, float d) {
  float frec = 55 * pow(2, octave + (float)(n + 9) / 12);
  ks.setFrecuencia(frec);
  env.cambia(1).cambia(0, d);
}
```

donde `n` es el número de nota (basado en el estandard MIDI) y d es la duración de la ráfaga de ruido, la cual suele ser muy corta (unos cuantos milisegundos). La duración de la nota emitida no depende tanto de la duración de la ráfaga, sino del coeficiente de retroalimentación y el valor del polo en la línea de retardo. Por lo general, se utiliza un coeficiente de retroalimentación alto (entre 0.7 y 1.0), mientras que el valor del polo debe estar entre cero y uno (a mayor valor, mayor absorción de altas frecuencias).

Para finalizar el programa de prueba, se sugiere incorporar una manera de disparar las notas (por ejemplo, mediante el teclado), así como una interfaz gráfica que permita manipular los siguientes parámetros de síntesis:

- Duración de la ráfaga de ruido (controla la duración del chasquido inicial al pulsar la cuerda)
- Factor de retroalimentación (controla el tiempo de relajación de la cuerda; es decir, la duración de cada nota)
- Posición del polo (controla la absorción de altas frecuencias, la cual se relaciona con el color del sonido)

22.6 Evaluación y reporte de resultados

1.- Reemplace la fuente de ruido por un oscilador basado en tabla de ondas, inicializando la tabla con valores aleatorios entre -1 y 1. Describa las diferencias tímbricas entre ambos enfoques.

2.- Con el objetivo de probar el comportamiento de la modulación de frecuencia de la UGen `KarplusStrong`, agregue un oscilador de baja frecuencia (LFO) al sistema implementado para simular *vibrato*. Agregue también los elementos de interfaz gráfica necesarios para controlar la frecuencia y amplitud de la modulación.

3.- Una manera rudimentaria de simular el sonido de cuerdas frotadas consiste en mantener un nivel de excitación externa sobre el sistema. Esto se puede lograr manipulando la ganancia base del VCA que controla la intensidad del ruido. Agregue un elemento de interfaz gráfica al programa de prueba para controlar esta ganancia y experimente con este parámetro. Discuta sus resultados en clase.

22.7 Retos

Considere una arquitectura de cómputo limitada, por ejemplo, el Arduino Uno, el cual cuenta con 2 Kb de memoria RAM y la capacidad de reproducir alrededor de 10,000 muestras por segundo utilizando un DAC de 8 bits. Suponga que desea implementar el algoritmo Karplus-Strong original, utilizando tiempos de retardo fijos y enteros, así como el filtro FIR basado en el promedio de muestras consecutivas. Escriba el pseudo-código para procesar cada muestra de la manera mas eficiente que le sea posible (e.g., usando aritmética entera o de punto fijo). Qué limitaciones imponen estas especificaciones en términos de la frecuencia fundamental de los sonidos que pueden reproducirse?

22.8 Conclusiones

Redacte en el recuadro sus conclusiones acerca de los conceptos aprendidos, las actividades realizadas y los objetivos planteados en esta práctica.

Capítulo 23

Filtros pasa-todo y phasers

Nombre del estudiante	Calificación

23.1 Relación con los programas de estudio

Materia	Unidad y tema
Procesamiento Digital de Señales (IE / IT / IB)	6.2.- Consideraciones para el diseño de filtros 6.4.- Diseño de filtros IIR
Procesamiento de Señales de Audio (IE)	1.7.- Sistemas LIT dados como ecuaciones en diferencias 1.8.- Filtros FIR e IIR 1.9.- Respuesta en frecuencia 2.1.- Filtros de primer orden 2.4.- Filtros de orden superior 4.5.- Filtros pasa-todo y phasers 5.5.- Aplicaciones musicales en arquitecturas diversas

23.2 Introducción

En la Práctica 5 se mencionó que los filtros pueden clasificarse con respecto a su respuesta en frecuencia, usualmente como pasa-bajas, pasa-altas, pasa-bandas, etc. Una de estas categorías corresponde a los filtros *pasa-todo*, los cuales se caracterizan por tener una respuesta en frecuencia unitaria para todas las frecuencias (es decir, $|H(e^{j\omega})| = 1$); sin embargo, su respuesta *en fase* no es necesariamente homogénea. [19]

Cualquier sistema LIT que conserve la totalidad de la energía de la señal, durante un periodo suficientemente largo de tiempo, puede considerarse un fil-

tro pasa-todo. Específicamente, si $x[n]$ es la señal de entrada y $y[n]$ es la señal de salida de un sistema donde la energía se conserva, entonces para un N suficientemente grande se cumple que

$$\sum_{n=0}^{N-1} |x[n]|^2 = \sum_{n=0}^{N-1} |y[n]|^2. \tag{23.1}$$

Supongamos que se ingresa como señal de entrada una exponencial compleja con frecuencia ω, es decir, $x[n] = \exp\{j\omega n\}$, a un sistema LIT que conserva la energía, en el sentido descrito por la Ecuación 23.1. Dado que se trata de un sistema LIT, la salida del sistema será la misma exponencial compleja multiplicada por un factor complejo que depende de ω, es decir $y[n] = H(\omega)x[n]$. El factor $H(\omega)$ es precisamente la respuesta en frecuencia del sistema y la conservación de la energía implica que $|H(\omega)| = 1$, por lo cual el sistema es un filtro pasa-todo. [10]

23.2.1 Ejemplos de filtros pasa-todo

Consideremos, en primera instancia, el retardo simple implementado en la Práctica 14, dado por la ecuación $y[n] = x[n-d]$. Dada una señal de entrada finita de longitud M, la energía a la salida claramente se conserva (tomando $N = M+d$); por lo tanto, *cualquier retardo simple es un filtro pasa-todo.*

En segunda instancia, recordemos el sistema amplificador $y[n] = g \cdot x[n]$, implementado en la Práctica 3. Este sistema conserva la energía cuando $|g| = 1$, lo cual no parece ser muy interesante si consideramos que g es real, ya que o bien el sistema no hace nada, o solamente invierte la señal de entrada. Sin embargo, el panorama se amplía cuando consideramos que g (al igual que $x[n]$ y $y[n]$) puede ser complejo. Ya que $|g| = 1$, podemos expresar la ganancia como $g = \exp\{j\phi\}$, de manera que la salida $y[n]$ es igual a la entrada $x[n]$ rotada por un ángulo ϕ en el plano complejo. Expresando las partes real y compleja de las señales de manera separada, este sistema (al que llamaremos *rotación*) puede representarse también como:

$$\begin{aligned} y_r[n] &= cx_r[n] - sx_i[n], \\ y_i[n] &= sx_r[n] + cx_i[n], \end{aligned}$$

donde $c = \cos(\phi)$ y $s = \text{sen}(\phi)$. [10]

Este sistema tiene una respuesta en fase constante e igual a ϕ, pero es de utilidad para obtener combinaciones lineales de dos señales sin pérdida de energía total. Por ejemplo, es común implementar un *crossfade* entre dos señales de entrada $x_r[n]$ y $x_i[n]$ tomando la salida $y_i[n]$ del sistema de rotación y variando el ángulo entre 0 y $\pi/2$.

Otro caso especial es cuando se utiliza el ángulo $\phi = \pi/4$, de manera que $s = c = \sqrt{1/2} \approx 0.7071$. La salida para este caso consiste en la diferencia y la suma de las señales de entrada, atenuadas por un factor que mantiene la energía total.

Finalmente, consideremos el caso de un filtro IIR con un polo y un cero. En general, este filtro tendría una función de transferencia dada por el cociente de dos polinomios de grado uno; es decir,

$$H(z) = \frac{b_0 z + b_1}{z + a_1} = \frac{b_0 + b_1 z^{-1}}{1 + a_1 z^{-1}}.$$

Factorizando el coeficiente b_1 del numerador, obtenemos

$$H(z) = b_1 \frac{b + z^{-1}}{1 + a z^{-1}},$$

con $b = b_0/b_1$.

Para diseñar un filtro pasa-todo se requiere que $|H(z)| = 1$ cuando $|z| = 1$. Esto se puede lograr haciendo $b_1 = 1$ y $b = \bar{a}$ (donde $\bar{a}$ representa el complejo conjugado de a). De esta manera, obtenemos la función de transferencia del *filtro pasa-todo de primer orden*:

$$H(z) = \frac{\bar{a} + z^{-1}}{1 + a z^{-1}}, \tag{23.2}$$

con un polo en $-a$ y un cero en $-\bar{a}^{-1}$. [19]

La ecuación en diferencias que describe a este filtro es

$$y[n] = \bar{a}x[n] + x[n-1] - a y[n-1],$$

cuya implementación puede simplificarse enormemente si el parámetro a se limita a los números reales entre -1 y 1.

Este filtro presenta una respuesta en fase no lineal que varía desde cero para $\omega = 0$ hasta $-\pi$ para $\omega = \pi$. La forma de la curva se caracteriza por la *frecuencia de quiebre* ω_q, que es la frecuencia para la cual la respuesta en fase es exactamente $-\pi/2$ y depende del valor del parámetro a (ver Figura 23.1). Dada una frecuencia de quiebre deseada ω_q (en radianes por muestra), el valor de a requerido está dado por

$$a = \tan \omega_q - \frac{1}{\cos \omega_q}.$$

23.2.2 Propiedades de los filtros pasa-todo

Los filtros pasa-todo tienen algunas propiedades que son de gran utilidad para el diseño de filtros mas elaborados (e.g., de mayor orden). Considere dos filtros pasa-todo con funciones de transferencia $H_1(z)$ y $H_2(z)$. Por definición, sabemos que $|H_1(z)| = |H_2(z)| = 1$ cuando $|z| = 1$ (es decir, cuando $z = e^{j\omega}$).

Propiedad 1 (concatenación).- La concatenación en serie de dos filtros pasa-todo da como resultado otro filtro pasa-todo. Claramente, si $H_{12}(z)$ es la función de transferencia de la concatenación de ambos filtros, entonces $H_{12}(z) = H_1(z)H_2(z)$, y por lo tanto, $|H_{12}(z)| = 1$ cuando $|z| = 1$.

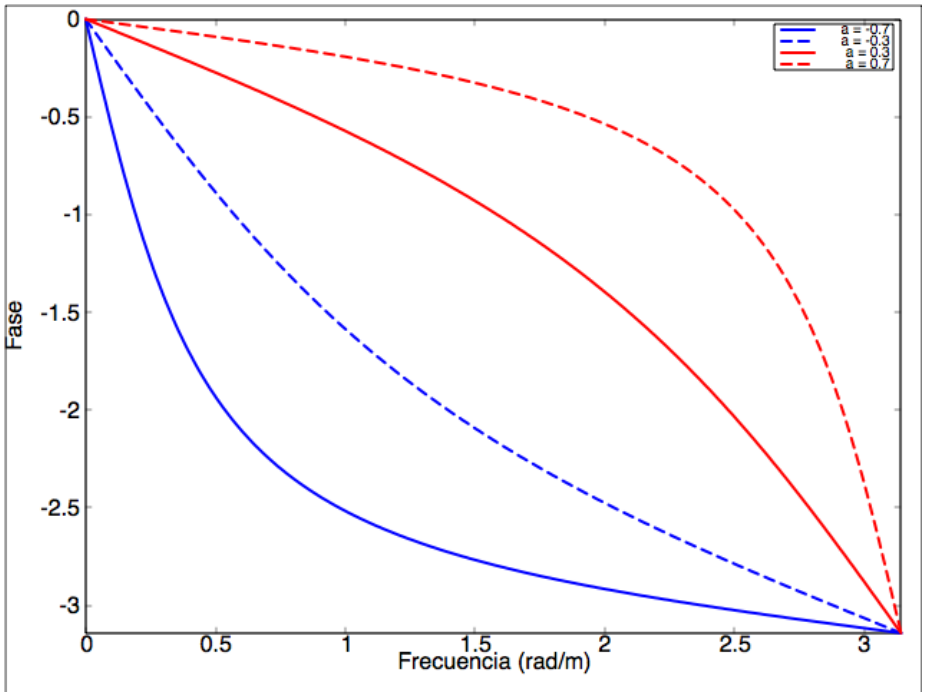


Figura 23.1: Respuesta en fase de un filtro pasa-todo de primer orden para varios valores del parámetro a (el negativo del polo).

Propiedad 2 (anidamiento).- La composición de funciones de transferencia de filtros pasa-todo da como resultado la función de transferencia de otro filtro pasa-todo. Claramente, $|z| = 1$ implica que $|H_1(z)| = 1$, lo cual a la vez implica que $|H_2(H_1(z))| = 1$. Así mismo, las funciones de transferencia $H_1(H_1(z))$, $H_1(H_2(z))$ y $H_2(H_2(z))$ corresponden todas a distintos filtros pasa-todo.

23.2.3 Filtros de orden superior y efecto phaser

Los filtros pasa-todo rara vez se utilizan de manera aislada, ya que no afectan el espectro de las señales que procesan sino solamente la fase (y como mencionamos anteriormente, el oído es hasta cierto punto insensible a la fase). Una excepción son los retardos largos (> 20 ms), los cuales no modifican el espectro pero sí la percepción del tiempo en el que ocurre un evento.

Sin embargo, cuando se combina la salida de un filtro pasa-todo con la señal original, los desfasamientos entre ambas señales dan lugar a interferencia, la cual es constructiva para algunas frecuencias y destructiva para otras. Entonces, el espectro de la suma de ambas señales ya no es necesariamente unitario, sino que puede mostrar crestas, valles y/o tendencias. Tomemos, por ejemplo, la suma de la salida del filtro pasa-todo de primer orden con parámetro a real y la señal original de entrada. La función de transferencia de este sistema está dada por

$$H(z) = 1 + \frac{az+1}{z+a} = (1+a)\frac{z+1}{z+a} = (1-p)\frac{z+1}{z-p},$$

con $p = -a$. Este sistema corresponde a un filtro con un polo en $z = p$ y un cero en $z = -1$, lo cual es claramente un filtro pasa-bajas.

Un filtro interesante, que se utiliza de manera común en la música, es el *phaser*. Este consiste en un filtro pasa-todo de orden par (usualmente entre 4 y 12) formado a partir de la concatenación en serie de múltiples filtros pasa-todo de bajo orden (1 ó 2). A la salida del filtro se le suma la señal original (posiblemente atenuada por un factor G), como se muestra en la Figura 23.2. Las frecuencias de quiebre de cada una de las etapas se encuentran espaciadas a lo largo de una cierta banda de frecuencia, por lo general de manera exponencial (e.g., por octavas), por lo que la respuesta en frecuencia del sistema completo consta de una serie de picos invertidos o muescas, como la que se muestra en la Figura 23.3. Esto se debe a lo siguiente: considere dos filtros pasa-todo de primer orden, con frecuencias de quiebre ω_1 y ω_2 relativamente cercanas (digamos, a una octava de distancia, de manera que $\omega_2 = 2\omega_1$). Para frecuencias suficientemente menores a ω_1, la respuesta en fase de ambos filtros será cercana a cero, por lo que el desfasamiento total (con respecto a la señal original) será también cercano a cero. Para frecuencias suficientemente mayores a ω_2, la respuesta en fase de cada filtro será cercana a π, por lo que el desfasamiento total será aproximadamente 2π, lo cual nuevamente equivale a un desfasamiento cercano a cero. Por lo tanto, lejos de la banda que va de ω_1 a ω_2, la salida del filtro estará aproximadamente en fase con la señal original, por lo que habrá interferencia constructiva (con una ganancia máxima $1 + G$). Sin embargo, dentro de la banda entre ω_1 a ω_2, cada filtro producirá un desfasamiento cercano a $-\pi/2$, por lo que el desfasamiento

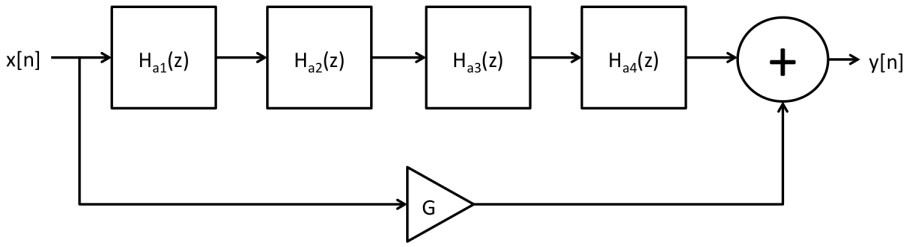


Figura 23.2: Diagrama de un phaser simple de cuatro etapas, donde cada etapa es un filtro pasa-todo de primer orden.

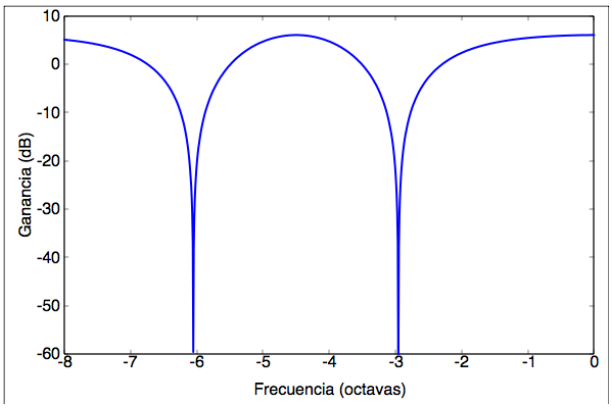


Figura 23.3: Respuesta en frecuencia de un phaser de cuatro etapas (dos muescas) con frecuencias de quiebre en $\pi/64$, $\pi/32$, $\pi/16$ y $\pi/8$. El eje de la frecuencia está en octavas a partir de la frecuencia de Nyquist.

total será cercano a $-\pi$, dando como resultado interferencia destructiva y una atenuación (con ganancia mínima de $1 - G$) de las frecuencias en esta banda. Por esta razón, un phaser formado por filtros pasa-todo de primer orden requiere de dos etapas por cada muesca que se desee.

El efecto phaser típico se obtiene cuando se modulan las frecuencias de quiebre de manera cíclica, por lo general mediante un oscilador de baja frecuencia (frecuencias entre 0.1 y 4 Hz).

23.3 Objetivos didácticos

- Conocer la definición y principales propiedades de los filtros pasa-todo
- Comprender el diseño de algunos filtros pasa-todo de bajo orden y algunas de sus aplicaciones
- Implementar un efecto de phaser utilizando filtros pasa-todo de primer

orden

23.4 Material

- Una computadora con alguna de las siguientes combinaciones de software instalado:
  - Processing y Beads
  - Processing y Minim
  - GNU C++ (e.g., MinGW) y PortAudio
- Bocinas o audífonos estéreo

23.5 Procedimiento

La manera mas sencilla de implementar el phaser que se muestra en la Figura 23.2 consiste en implementar un filtro pasa-todo de primer orden como una nueva UGen, y posteriormente construir la cadena de procesamiento utilizando cuatro (o más) de estos filtros en serie. La implementación de esta UGen, para una frecuencia de quiebre fija, es muy sencilla:

```
public class PasaTodo extends UGen {
  protected float xant, yant;
  protected float polo;

  public PasaTodo(AudioContext context) { super(context, 1, 1); }

  float polo() { return polo; }

  void setPolo(float p) { polo = constrain(p, -1, 1); }

  public void calculateBuffer() {
    float[] in = bufIn[0];
    float[] out = bufOut[0];
    float a = -polo;
    for (int i = 0; i < bufferSize; i++) {
      out[i] = a * in[i] + xant - a * yant;
      xant = in[i];
      yant = out[i];
    }
  }
}
```

Sin embargo, para una implementación mas práctica de un phaser, será necesario incorporar los siguientes aspectos:

1. Proporcionar las frecuencias de quiebre en Hertz, en lugar de la posición del polo

2. Modulación exponencial de la frecuencia de quiebre por una señal externa

3. Variar el número de etapas o muescas del filtro phaser sin necesidad de manipular la cadena de UGens

4. Permitir la posibilidad de implementar retroalimentación en el filtro phaser

Por estos motivos, será mas conveniente implementar una UGen que represente una cadena de filtros pasa-todo (de primer orden) dispuestos en serie. Las frecuencias de quiebre de las distintas etapas estarán separadas por octavas; es decir, la frecuencia de quiebre de la k-ésima etapa será el doble de la frecuencia de quiebre de la etapa $k-1$. Esto simplificará la implementación en dos sentidos: primero, no será necesario aplicar una modulación independiente a la frecuencia de quiebre de cada etapa, sino que la modulación se aplicará únicamente a la frecuencia base (la de la primer etapa); en segundo lugar, esta relación entre frecuencias de quiebre permitirá calcular de manera mas eficiente el valor de los polos.

Sea ω_k la frecuencia de quiebre correspondiente a la k-ésima etapa, y sea $\omega_{k+1} = 2\omega_k$. El parámetro del k-ésimo filtro pasa-todo está dado por $a_k = t_k - c_k^{-1}$, donde $t_k = \tan\omega_k$ y $c_k = \cos\omega_k$. Utilizando identidades trigonométricas se pueden calcular fácilmente t_{k+1} y c_{k+1} como

$$t_{k+1} = 2t_k/(1-t_k^2), \tag{23.3}$$
$$c_{k+1} = 2c_k^2 - 1, \tag{23.4}$$

a partir de los cuales se puede calcular a_{k+1}. De esta manera, solo es necesario llamar a las funciones coseno y tangente una vez por cada muestra, para calcular t_1 y c_1; además de llamar una vez a la función potencia para calcular el factor de modulación de la frecuencia base ω_1. Por supuesto, existen diversas maneras de optimizar el código para evitar por completo las llamadas a funciones matemáticas y reducir el número de operaciones; una de ellas se propone como reto al final de la práctica.

A continuación se presenta el código fuente del filtro phaser, el cual únicamente comprende la concatenación de filtros pasa-todo de primer orden, y por lo tanto representa también un filtro pasa-todo (cuyo orden es igual al número de etapas). El sistema completo para implementar el efecto de phaser se discute mas abajo.

```
public class Phaser extends VCF {
    int etapas;
    float[] xant, yant;

    public Phaser(AudioContext ac, int e) {
      super(ac);
      setEtapas(e);
```

```
    }

    int etapas() { return etapas; }

    void setEtapas(int e) {
      etapas = constrain(e, 1, 6);
      xant = new float[etapas];
      yant = new float[etapas];
    }

    public void calculateBuffer() {
      float w, tw, cw, a, x, y ;
      float[] in = bufIn[0];
      float[] out = bufOut[0];
      float[] mod = bufIn[1];
      float omega = 2 * PI * frecuencia / context.getSampleRate();

      for (int i = 0; i < bufferSize; i++) {
        w = omega * pow(2, indiceModulacion * mod[i]);
        tw = tan(w);
        cw = cos(w);
        x = in[i];
        for (int s = 0; s < etapas; s++) {
          a = constrain(tw - 1 / cw, -1, 1);
          y = a * x + xant[s] - a * yant[s];
          xant[s] = x;
          yant[s] = y;
          x = y;
          tw = 2 * tw / (1 - tw * tw);
          cw = 2 * cw * cw - 1;
        }
        out[i] = x;
      }
    }
}
```

La clase se deriva directamente de la UGen `VCF`, lo cual nos proporcionará miembros y métodos para manipular la frecuencia (que ahora será frecuencia de quiebre), la señal moduladora y el índice de modulación. Además, agregamos un miembro `int etapas` para guardar el número de etapas (que estará limitado entre 1 y 6) y las funciones *set* y *get* correspondientes. También se tienen como miembros dos arreglos `xant` y `yant` para guardar los valores anteriores (es decir, al tiempo $n-1$) de la señales de entrada y de salida en cada etapa de filtrado. Finalmente, la función callback `calculateBuffer()` calcula primero la frecuencia base `omega` en radianes por muestra (a partir del valor en Hertz dado por el miembro `frecuencia`) y a continuación entra al ciclo que procesa

las muestras. Dentro de este ciclo se calcula la frecuencia modulada `w`, así como el valor su tangente y coseno, para la primer etapa. Un segundo ciclo anidado calcula la salida de cada etapa y actualiza tanto las variables `xant` y `yant` como los valores de la tangente y coseno de la frecuencia de quiebre para la siguiente etapa.

23.5.1 Implementación del sistema phaser

Como etapa final de la práctica, se implementará una aplicación para aplicar el efecto phaser a una señal de entrada, la cual puede provenir de la entrada física de audio, o de una señal de prueba como ruido blanco (lo cual usaremos en este ejemplo). También es interesante integrar el phaser a alguno de los sintetizadores elaborados en prácticas anteriores.

A manera de ejemplo, utilizaremos los siguientes módulos para construir un phaser:

```
RuidoBlanco ruido;
Amplificador atenuador;
Amplificador ampEntrada;
Phaser phaser;
Senoidal lfo;
```

La señal de prueba provendrá de la UGen `ruido`, aunque se sugiere al alumno a probar con otras señales. Ya que el phaser puede tener una ganancia máxima de 2 para algunas frecuencias, utilizaremos un amplificador de ganancia fija (llamado `atenuador`) para atenuar la señal de prueba por un factor de 1/2 y evitar saturación. El amplificador `ampEntrada` controlará el nivel de la señal de prueba (ya atenuada) a la salida del sistema (parámetro G en la Figura 23.2). La señal de prueba atenuada se conectará tanto a este amplificador como al `phaser`, y las salidas de estos dos módulos se conectarán a la salida física de audio. Por último, se declara un oscilador de baja frecuencia `lfo`, el cual se usará para modular la posición de las muescas del phaser.

La creación de la cadena de procesamiento se ejemplifica en la siguiente función `setup()`:

```
void setup() {
  int x, y, dy;
  size(800, 600);

  // crear cadena de audio
  ac = new AudioContext();
  ruido = new RuidoBlanco(ac, 1);
  atenuador = new Amplificador(ac, 1);
  ampEntrada = new Amplificador(ac, 1);
  phaser = new Phaser(ac, 4);
  lfo = new Senoidal(ac, 1);
```

```
  atenuador.addInput(ruido);
  atenuador.setGanancia(0.5);
  ampEntrada.addInput(atenuador);
  ampEntrada.setGanancia(1);
  phaser.setEntrada(atenuador);

  phaser.setModulador(lfo);
  phaser.setIndice(1);

  ac.out.addInput(ampEntrada);
  ac.out.addInput(phaser);

  ac.start();
}
```

Se deja al alumno el ejercicio de elaborar una interfaz gráfica para manipular los parámetros del sistema, que son los siguientes:

- **Etapas.-** Número de etapas del phaser (2, 4 o 6).
- **Profundidad del efecto.-** Ganancia del amplificador `ampEntrada` (parámetro G de la Fig. 23.2). Su valor debe estar entre 0 y 1.
- **Frecuencia base.-** Frecuencia de quiebre del primer filtro pasa-todo. Un rango de cuatro o cinco octavas partiendo del límite auditivo inferior suele funcionar bien.
- **Frecuencia de modulación.-** Frecuencia de la señal moduladora, entre 0.1 y 4 Hz.
- **Profundidad de modulación.-** Indice de modulación de la frecuencia base del phaser, con valor entre 0 y 1.

23.6 Evaluación y reporte de resultados

1.- Demuestre que la Ecuación 23.2 corresponde realmente a la función de transferencia de un filtro pasa-todo.

2.- Demuestre que para obtener una frecuencia de quiebre ω_q en un filtro pasa-todo de primer orden (Ecuación 23.2), el valor del parámetro a debe ser $a = \tan \omega_b - \cos^{-1} \omega_b$.

3.- Agregue al phaser un lazo de retroalimentación desde la salida del último filtro pasa-todo hacia la entrada del sistema, con un factor de retroalimentación Q (aprovechando el miembro `Q` de la clase `VCF`). Añada a la interfaz del programa de prueba un control para manipular Q. Explore y describa el tipo de sonidos y timbres que pueden obtenerse con este phaser.

4.- Una aproximación lineal de $a = \tan \omega_b - \cos^{-1} \omega_b$ está dada por

$$\tilde{a} = \frac{2\omega}{\pi} - 1.$$

Grafique ambas funciones y calcule el error máximo entre las frecuencias representadas por a y $\tilde{a}$

23.7 Retos

Una limitación del phaser implementado en esta práctica radica en que las frecuencias de quiebre están espaciadas en intervalos de una octava, por lo que al incrementar el número de etapas se puede llegar rápidamente a la frecuencia de Nyquist, ocasionando artefactos e inestabilidad numérica. Diseñe un nuevo phaser en el que uno pueda definir la separación entre las frecuencias de quiebre mediante un parámetro adicional `separacion` dado en octavas (el caso `separacion = 1` correspondería al phaser ya implementado). Evite el uso de funciones coseno y tangente utilizando la aproximación $a \approx 2\omega/\pi - 1$.

23.8 Conclusiones

Redacte en el recuadro sus conclusiones acerca de los conceptos aprendidos, las actividades realizadas y los objetivos planteados en esta práctica.

Capítulo 24

Reverberación artificial

Nombre del estudiante	Calificación

24.1 Relación con los programas de estudio

Materia	Unidad y tema
Procesamiento Digital de Señales (IE / IT / IB)	6.2.- Consideraciones para el diseño de filtros 6.4.- Diseño de filtros IIR
Procesamiento de Señales de Audio (IE)	2.1.- Filtros de primer orden 2.4.- Filtros de orden superior 4.3.- Reverberación 5.2.- Espacialización binaural y localización estéreo 5.5.- Aplicaciones musicales en arquitecturas diversas 5.6.- Aplicaciones a las interfaces humano-máquina 5.7.- Representación auditiva de datos científicos

24.2 Introducción

La reverberación natural se produce en espacios cerrados a partir de un patrón muy complejo de reflexiones y absorciones que ocurren a lo largo del tiempo a partir de la emisión de un sonido. Estas reflexiones ocasionan que el sonido *persista* por un cierto tiempo aún después de que la fuente de sonido original ha cesado. Es común escuchar reverberación en espacios cerrados relativamente

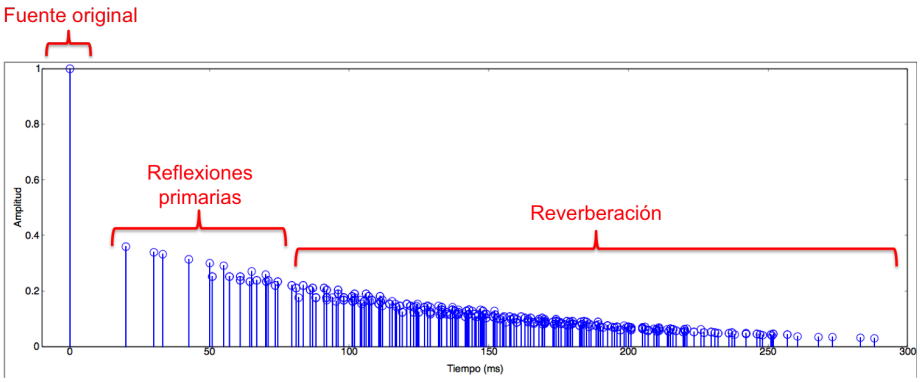


Figura 24.1: Patrón de reflexiones que se produce en un espacio cerrado a lo largo del tiempo dando lugar a la reverberación.

amplios como iglesias, teatros o auditorios, en particular cuando éstos se encuentran casi vacíos.

Desde un punto de vista psicoacústico, la reverberación proporciona una sensación del espacio donde se ubica una fuente de sonido, así como de la posición relativa entre la fuente y el receptor. El sonido de la reverberación tiene aproximadamente la misma intensidad en cualquier punto del espacio donde ésta se produce, pero la intensidad del sonido producido por la fuente decae de manera proporcional al cuadrado de la distancia entre la fuente y el receptor; por lo tanto, la diferencia entre la intensidad del sonido original y la de la reverberación producida nos proporciona un claro indicio acerca de la distancia a la que se encuentra la fuente. [9, 31]

En prácticamente cualquier aplicación relacionada con la producción de audio (música, cine, videojuegos, etc.), es común agregar reverberación de manera artificial para dar un mayor realismo a los distintos sonidos, simular la ubicación de los sonidos dentro de un cierto espacio, o añadir cohesión a un grupo de sonidos generados mediante distintas fuentes.

24.2.1 Descripción del fenómeno

Considere una fuente de sonido ubicada dentro de una habitación cerrada y vacía (salvo por la fuente y el receptor). Suponga que la fuente emite una señal que es aproximadamente un impulso (por ejemplo, un sonido muy corto y percusivo, como un chasquido o un aplauso). El sonido se propagará en todas direcciones en una onda esférica, cuya energía total se conserva en condiciones ideales pero distribuida en el área de la esfera. Por lo tanto, la energía por unidad de área disminuye a razón de $1/r^2$, donde r es la distancia entre la fuente y el receptor. Dado que la energía es proporcional al cuadrado de la amplitud de la onda, entonces la amplitud decrece a razón de $1/r$ [19]. De esta manera, la trayectoria entre la fuente original del sonido y el receptor puede modelarse mediante un

retardo simple (donde el tiempo de retardo es proporcional a la distancia r entre fuente y receptor), seguido de un amplificador con ganancia proporcional a $1/r$.

En cada dirección de propagación, la onda se topará eventualmente con una pared de la habitación. Parte de la energía de la onda será absorbida y otra parte será reflejada de nuevo hacia el receptor. Estas ondas se conocen como reflexiones primarias, ya que en su trayectoria se han encontrado con alguna de las paredes en solo una ocasión, y su amplitud estará en función de la distancia recorrida y de las propiedades de absorción del material de las paredes. De hecho, cuando la onda choca con una superficie, ésta generalmente no se refleja en una única dirección sino que se dispersa en múltiples direcciones y posiblemente rebota varias veces en diferentes superficies antes de alcanzar al receptor. Algunas de estas reflexiones pueden llegar mas o menos al mismo tiempo que algunas de las (últimas) reflexiones primarias, pero la mayoría llegarán un cierto tiempo después. Por supuesto, las reflexiones mas tardías son aquellas que han rebotado un mayor número de veces y recorrido una mayor distancia, por lo que su amplitud se ve atenuada en buena medida. La Figura 24.1 ilustra este fenómeno a través de un *ecograma*, donde cada línea vertical (en color azul) representa una versión retardada y atenuada del sonido original. Note que después de las reflexiones primarias, la densidad de reflexiones aumenta exponencialmente a la vez que su intensidad disminuye también exponencialmente. La densidad llega a ser del orden de miles de ecos por segundo, por lo que no es posible distinguir los ecos individuales.

Una de las técnicas mas comunes para producir reverberación artificial mediante procesamiento digital consiste en modelar por separado las reflexiones primarias y la reflexiones posteriores, utilizando principalmente retardos y filtros pasa-todo. Existe un gran número de algoritmos de reverberación, pero en esta práctica estudiaremos una variante del algoritmo descrito por Miller Puckette en [10].

24.2.2 Modelo de reflexiones primarias

Consideremos el sistema con dos canales de entrada y dos canales de salida que se presenta en la Figura 24.2a, el cual comienza con una rotación (filtro pasa-todo) de la forma

$$\begin{aligned} y_1[n] &= \cos(\phi)x_1[n] - \text{sen}(\phi)x_2[n], \\ y_2[n] &= \text{sen}(\phi)x_1[n] + \cos(\phi)x_2[n], \end{aligned}$$

donde por simplicidad elegiremos $\phi = \pi/4$ de manera que $\cos(\phi) = \text{sen}(\phi) = c = \sqrt{1/2}$. Este filtro recombina las dos señales de entrada, calculando su diferencia y suma, pero conservando la energía total del sistema. A continuación, a una de las salidas de la rotación se le aplica un retardo simple de $d = d_1$ muestras. A la salida de este sistema estarán presentes las dos señales originales (recombinadas), así como las dos señales originales retardadas (también recombinadas), simulando así la percepción simultánea de la señal original como de una primer reflexión. Note que la energía total del sistema se conserva, ya que el retardo es también un filtro pasa-todo.

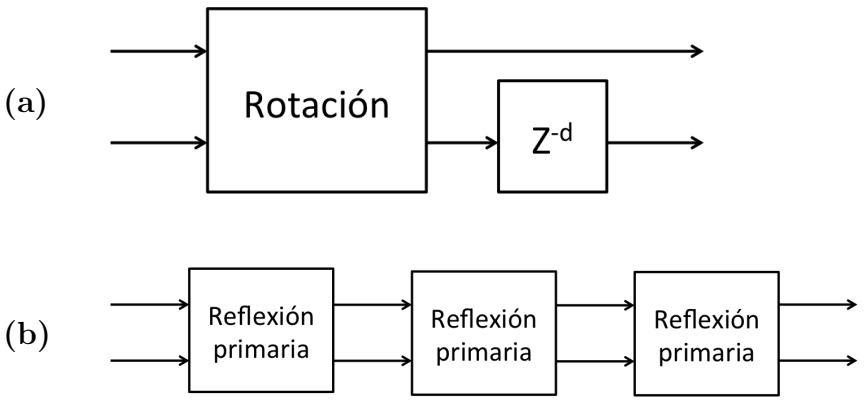


Figura 24.2: (a) Bloque para generar una reflexión primaria y combinarla con la señal original. (b) Aplicación del bloque anterior en serie para generar reflexiones primarias, secundarias y ternarias.

Matemáticamente, la salida del sistema se puede expresar como el vector

$$\begin{bmatrix} c(x_1[n] - x_2[n]) \\ c(x_1[n-d_1] + x_2[n-d_1]) \end{bmatrix}.$$

Suponga que tomamos la salida del sistema anterior y la ingresamos como entrada a un sistema idéntico, salvo por el tiempo de retardo, el cual para el segundo sistema es d_2. Entonces las señales originales serán ahora recombinadas con las señales retardadas por un segundo filtro de rotación, y una de las combinaciones pasará por un segundo retardo. La salida del segundo sistema estará dada por:

$$\begin{bmatrix} c^2(x_1[n] - x_2[n] - x_1[n-d_1] - x_2[n-d_1]) \\ c^2(x_1[n-d_2] - x_2[n-d_2] + x_1[n-d_1-d_2] + x_2[n-d_1-d_2]) \end{bmatrix}.$$

Podemos ver que ahora la salida contiene (de manera recombinada) a las señales originales $x_1[n]$ y $x_2[n]$, así como las señales correspondientes a las reflexiones primarias $x_1[n-d_1]$, $x_2[n-d_1]$, $x_1[n-d_2]$ y $x_2[n-d_2]$, donde los distintos tiempos de retardo simulan el que las señales hayan sido reflejadas en diferentes superficies. Finalmente, se encuentran presentes también las señales retardadas por $d_1 + d_2$ muestras; estas pueden considerarse como reflexiones primarias contra una superficie que está aún mas alejada, o como reflexiones secundarias que resultan de rebotar primero en una superficie y luego en otra.

Continuando este proceso, podemos pasar la salida del segundo sistema por un tercer sistema idéntico, pero ahora con tiempo de retardo d_3, para obtener un sistema como el que se muestra en la Figura 24.2b. A la salida del tercer bloque tendremos una combinación de las señales originales, las reflexiones primarias

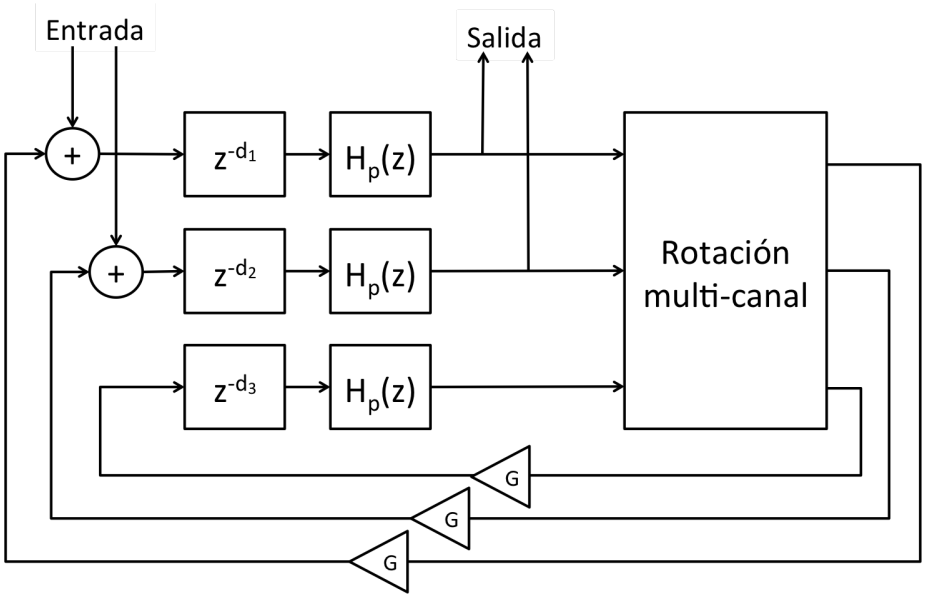


Figura 24.3: Sistema de reverberación propuesto, el cual consiste en un arreglo paralelo de múltiples líneas de retardo cuyas salidas son filtrada (mediante filtros pasa-bajas de primer orden), recombinadas mediante rotación, y retroalimentadas.

(con tiempos de retardo d_1, d_2 y d_3), secundarias (con tiempos de retardo d_1+d_2, d_1+d_3 y d_2+d_3) y ternarias (con tiempo de retardo $d_1+d_2+d_3$).

Es posible encadenar mas bloques para producir un mayor número de reflexiones primarias. Por cada bloque que se agrega en serie, el número de reflexiones se duplica (con retardos cada vez mas largos) pero la energía total se conserva, distribuyéndose entre todas las reflexiones. Es común utilizar de 6 a 8 bloques para obtener una densidad entre 64 y 256 reflexiones en un periodo de tiempo igual a la suma de todos los tiempos de retardo.

Respecto a los tiempos de retardo, es importante cuidar que estos no sean demasiado largos como para que el oído detecte las reflexiones como ecos individuales, ni demasiado cortos como para que produzcan coloración en la señal de salida por el efecto de filtro peine (como ocurre, por ejemplo, con un flanger). Por lo general se busca que el tiempo total de retardo esté entre los 30 y 80 milisegundos, aunque si la densidad de reflexiones es suficiente, el tiempo total puede ser aún mayor.

24.2.3 Modelo de reverberación

La reverberación que sucede a las reflexiones primarias puede tener una duración que va desde algunas décimas de segundo hasta varios segundos, con densidades del orden de miles de reflexiones por segundo. Una manera de generar

tal número de reflexiones es mediante un arreglo paralelo de líneas de retardo retroalimentadas, pero agregando un esquema de recombinación en el lazo de retroalimentación que permita que parte de la salida de una línea de retardo se transfiera a las demás. Este diseño se ilustra en la Figura 24.3 para el caso de tres líneas de retardo en paralelo (en la práctica pueden utilizarse mas líneas para incrementar la densidad).

Consideremos las líneas de retardo de la Figura 24.3, con tiempos de retardo d_1, d_2 y d_3. Si se omite el esquema de recombinación (dado por el bloque de rotación multi-canal) y cada línea se retroalimenta a sí misma, entonces se producirán reflexiones en los tiempos kd_1, kd_2 y kd_3 para $k = 0, 1, \ldots, \infty$, con una amplitud proporcional a G^k (donde G es el ganancia de retroalimentación). En otras palabras, los ecos estarán equiespaciados dando lugar a un efecto de filtro peine que sonará artificial y producirá una coloración no deseada. Al incorporar la recombinación, los ecos se generaran en tiempos de la forma $k_1d_1 + k_2d_2 + k_3d_3$ con $k_1, k_2, k_3 = 0, 1, \ldots, \infty$, simulando la dispersión de una onda en múltiples direcciones al chocar contra una o más superficies. Los tiempos de retardo suelen elegirse primos entre sí, cubriendo uno o dos órdenes de magnitud.

El esquema de recombinación consiste en un filtro pasa-todo multicanal que recombine n entradas en n salidas. Existen diversas maneras de lograr esto, pero en esta práctica proponemos un esquema muy sencillo, el cual consiste en rotar primero los canales 1 y 2, luego rotar los canales 2 y 3, y así hasta los canales $n-1$ y n.

Además, podemos simular una mayor absorción de frecuencias altas agregando un filtro pasa-bajas de primer orden a la salida de cada retardo, con un parámetro $p \in [0, 1]$ que a la vez representa la posición del polo y el grado de absorción de altas frecuencias.

Finalmente, la entrada y salida del sistema pueden conectarse de manera casi arbitraria a cualquier parte del lazo de las distintas líneas de retardo.

El tiempo de reverberación está determinado principalmente por el factor de retroalimentación G, así como por los tiempos de retardo individuales d_k, y de manera secundaria por el parámetro de absorción de altas frecuencias p. Es difícil determinar la duración de la reverberación de una manera precisa, pero se puede estimar estadísticamente. Suponga que la duración promedio de los retardos es d, de manera que después de un tiempo t la señal ha recirculado aproximadamente t/d veces, en cada una de las cuales ha sido atenuada en un factor G. Por lo tanto, la atenuación total que sufre la señal después de un tiempo t es aproximadamente $G^{t/d}$. Una medida estandarizada del tiempo de reverberación es el tiempo que se requiere para lograr una atenuación de 60db (llamado RT60); es decir, el tiempo t_{RT60} para el cual

$$20 \log_{10} \left(G^{t_{\mathrm{RT60}}/d} \right) = -60.$$

Despejando G de la ecuación anterior, podemos ver que el factor de retroalimentación requerido para obtener una reverberación con una duración t_{RT60} está dado por:

$$G = 10^{-3d/t_{\mathrm{RT60}}}. \tag{24.1}$$

24.3 Objetivos didácticos

- Comprender el fenómeno de la reverberación y una manera de modelarlo digitalmente
- Diseñar e implementar un algoritmo de reverberación

24.4 Material

- Una computadora con alguna de las siguientes combinaciones de software instalado:
  - Processing y Beads
  - Processing y Minim
  - GNU C++ (e.g., MinGW) y PortAudio
- Bocinas o audífonos estéreo

24.5 Procedimiento

Para esta práctica se dividirá el procedimiento en dos partes. En la primera de ellas se implementará y probará una UGen para generar reflexiones primarias. La segunda parte se enfocará en agregar la reverberación mediante líneas de retardo retroalimentadas.

24.5.1 Implementación de reflexiones primarias

Como primer paso, implementaremos el bloque que produce las reflexiones primarias (Figura 24.2a) como una nueva UGen. La implementación es muy similar a la del `RetardoSimple`; sin embargo, se requieren dos canales de entrada y dos de salida. Además, utilizaremos un tamaño de buffer fijo, el cual será una potencia de dos (en este caso, $2^{13} = 8192$) suficiente para producir retardos de hasta 180 ms a una frecuencia de muestreo de 44100 Hz. El código de la clase es el siguiente:

```
public class ReflexionPrimaria extends UGen {
  float[] buffer;
  int indiceEscritura;
  int indiceLectura;
  final int mask = 0x1FFF; // 2^n - 1

  public ReflexionPrimaria(AudioContext ac) {
    super(ac, 2, 2);
    buffer = new float[mask + 1];
  }
```

```
  void setTiempo(float ms) {
    int tiempo = constrain(round(context.getSampleRate() * ms / 1000.0), 0, mask);
    indiceLectura = (indiceEscritura - tiempo + buffer.length) & mask;
  }

  public void calculateBuffer() {
    float[] in1 = bufIn[0];
    float[] in2 = bufIn[1];
    float[] out1 = bufOut[0];
    float[] out2 = bufOut[1];
    float y1, y2;
    float c = 0.707106781186548;
    if (buffer == null) return;

    for (int i = 0; i < bufferSize; i++) {
      // Rotacion
      y1 = c * (in1[i] - in2[i]);
      y2 = c * (in1[i] + in2[i]);

      // Almacena muestra de entrada en el buffer
      buffer[indiceEscritura] = y2;

      // Calcula la muestra de salida interpolando el buffer
      out1[i] = y1;
      out2[i] = buffer[(int)indiceLectura];

      // Actualiza los indices de escritura y lectura
      indiceEscritura = (indiceEscritura + 1) & mask;
      indiceLectura = (indiceLectura + 1) & mask;
    }
  }
}
```

La clase contiene los mismos miembros que la UGen `RetardoSimple`; es decir, el buffer de retardo y los apuntadores de lectura y escritura. Además, se incluye un miembro constante `mask` que contiene una máscara binaria para aislar los bits que se usarán para los apuntadores (en este caso, 13 bits). De esta manera se pueden reemplazar las operaciones `x % buffer.length` por `x & mask`, y evitar una división. Dado que el tamaño del buffer es fijo, el constructor no requiere parámetros adicionales. Agregamos también un método `setTiempo()` para fijar el tiempo de retardo (si el alumno lo desea, puede agregar el método `getTiempo()` de la clase `RetardoSimple`). Finalmente, la función callback procesa las muestras aplicando primero una rotación entre ambos canales de entrada, y posteriormente el retardo al segundo canal.

Una excelente manera de probar esta nueva UGen consiste en agregar varias etapas de reflexiones primarias a la señal proveniente de la entrada física de

audio y escuchar el resultado a través de audífonos. Para esto, se puede definir un arreglo de UGens `ReflexionPrimaria` para manipularlas dentro de un ciclo:

```
AudioContext ac;
UGen entrada;
ReflexionPrimaria[] refPrim;
Amplificador ampReverb;

float[] tiemposRP = { 5, 2, 7, 5.5, 3, 6.5, 4.25, 3.3333 };

void setTiempoRP(float mult) {
  if (refPrim == null) return;
  for (int i = 0; i < refPrim.length; i++) {
    refPrim[i].setTiempo(mult * tiemposRP[i]);
  }
}

void setup() {
  int x, y, dy;
  size(800, 600);

  // crear cadena de audio
  ac = new AudioContext();
  entrada = ac.getAudioInput();
  refPrim = new ReflexionPrimaria[4]; // tamaño máximo = 8
  ampReverb = new Amplificador(ac, 2);

  for (int i = 0; i < ref.length; i++) {
    refPrim[i] = new ReflexionPrimaria(ac);
    if (i == 0) refPrim[i].addInput(entrada);
    else refPrim[i].addInput(refPrim[i-1]);
  }
  ampReverb.addInput(refPrim[refPrim.length - 1]);

  ac.out.addInput(ampReverb);
  ac.out.addInput(entrada);

  setTiempoRP(1);
  ac.start();
}
```

En este ejemplo, se crea un arreglo de cuatro bloques de reflexión primaria, los cuales se conectan en serie. La entrada de audio se conecta al primer bloque, y la salida del último bloque se conecta a un amplificador de ganancia fija que controlará la intensidad de la reverberación con respecto a la intensidad de la señal de entrada. Tanto la salida de este amplificador como la señal de entrada

se envían a la salida física de audio. Además, se incluye una función llamada `setTiempoRP` para manipular los tiempos de retardo de las reflexiones primarias y controlar así el tiempo de reverberación. Los tiempos se definen en el arreglo `tiemposRP`, multiplicados por un factor que recibe la función `setTiempoRP` como argumento. De esta manera, contamos hasta ahora con dos parámetros para modelar la reverberación: el factor de duración de las reflexiones primarias (para el cual se sugiere un rango de 1 a 10), y la intensidad de la reverberación (dada por la ganancia del amplificador `ampReverb`). Esto permite probar rápidamente las reflexiones primarias mediante una interfaz de usuario simple; por ejemplo, mapeando las coordenadas X y Y del mouse a los parámetros de duración e intensidad, respectivamente.

Los tiempos de reverberación serán relativamente cortos ya que solamente se modela un número relativamente pequeño de reflexiones. Para reverberaciones mas largas y densas, será necesario agregar el sistema de reverberación retroalimentada.

24.5.2 Implementación de la reverberación

Para implementar el sistema mostrado en la Figura 24.3 crearemos una nueva UGen que encapsulará múltiples líneas de retardo retroalimentadas, utilizando un arreglo de buffers (es decir, un arreglo bidimensional), así como arreglos para mantener los apuntadores de lectura y escritura para cada buffer, ya que cada buffer tiene un tiempo de retardo independiente. Además, la salida de cada retardo pasa por un filtro de primer orden, para lo cual se requiere conservar la salida anterior del filtro, y posteriormente se recombinan todas las salidas en un esquema de rotación. Otros parámetros importantes que deben conservarse como miembros de la clase son el número de líneas de retardo que conforman el sistema, el factor de retroalimentación y la posición del polo que controla la absorción de altas frecuencias; estos últimos dos parámetros serán iguales para todas las líneas de retardo.

A continuación se presenta el código de la clase:

```
public class Reverberador extends UGen {
  int numRetardos;
  float feedback;
  float polo;
  float[][] buffer;
  int[] indiceEscritura;
  int[] indiceLectura;
  float[] salida, anterior;
  final int mask = 0x1FFF; // 2^n - 1

  public Reverberador(AudioContext ac, int n) {
    super(ac, 2, 2);
    numRetardos = constrain(n + 2, 2, 8);
    buffer = new float[numRetardos][mask + 1];
```

```
  indiceEscritura = new int[numRetardos];
  indiceLectura = new int[numRetardos];
  salida = new float[numRetardos];
  anterior = new float[numRetardos];
}

int getNumRetardos() { return numRetardos; }

void setTiempo(int i, float ms) {
  if (i < 0 || i >= numRetardos) return;
  int tiempo = constrain(round(context.getSampleRate() * ms / 1000.0), 0, mask);
  indiceLectura[i] = (indiceEscritura[i] - tiempo + buffer[i].length) & mask;
}

void setFeedback(float fb) { feedback = constrain(fb, -1, 1); }

void setPolo(float p) { polo = constrain(p, 0, 1); }

public void calculateBuffer() {
  int i, c, c2;
  float temp1, temp2;
  float g = 0.707106781186548;

  for (i = 0; i < bufferSize; i++) {
    for (c = 0; c < numRetardos; c++) {
      // almacena entrada al retardo (con retroalimentacion)
      buffer[c][indiceEscritura[c]] = feedback * salida[c];

      // suma la señal de entrada a la entrada de las primeras líneas de retardo
      buffer[c][indiceEscritura[c]] += ((c < 2) ? bufIn[c][i] : 0);

      // calcula salida del retardo con filtro pasa-bajas
      salida[c] = (1 - polo) * buffer[c][(int)indiceLectura[c]] + polo * anterior[c];
      anterior[c] = salida[c];

      // Actualiza los indices de escritura y lectura
      indiceEscritura[c] = (indiceEscritura[c] + 1) & mask;
      indiceLectura[c] = (indiceLectura[c] + 1) & mask;

      // obtener salidas de las primeras dos lineas de retardo
      if (c < 2) bufOut[c][i] = salida[c];
    }

    // aplicar rotacion entre salidas de los filtros
    c2 = 0;
    for (c = 1; c < numRetardos; c++) {
```

```
        temp1 = salida[c] - salida[c2];
        temp2 = salida[c] + salida[c2];
        salida[c] = g * temp1;
        salida[c2] = g * temp2;
        c2 = c;
      }
    }
  }
}
```

Para probar esta clase, simplemente hay que insertar una UGen `Reverberador` entre el último bloque de reflexiones primarias y el amplificador `ampReverb`. El número de líneas de retardo utilizadas se establece en el constructor de `Reverberador` y no puede ser modificado en tiempo de ejecución (aunque no es complicado añadir esta funcionalidad). Una vez que se ha creado el objeto `Reverberador`, uno puede fijar los tiempos de retardo para cada línea de retardo mediante el método `setTiempo()`; sin embargo, ahora el tiempo de reverberación puede controlarse mediante el factor de retroalimentación usando el método `setFeedback()`.

Por lo tanto, el sistema completo de reverberación cuenta con cuatro parámetros, que son los siguientes:

- Dispersión de las reflexiones primarias (dado por el factor que multiplica los tiempos de retardo de las reflexiones). De manera indirecta, este parámetro también está asociado con el tamaño del espacio que se pretende simular.
- Tiempo de reverberación (dado por el factor de retroalimentación del reverberador). Note que se puede generar una reverberación infinita haciendo $G = 1$. Esto puede aprovecharse para la creación de algunos efectos interesantes u osciladores no armónicos.
- Absorción de altas frecuencias (dado por la posición del polo de los filtros del reverberador).
- Intensidad de la reverberación (dado por la ganancia del amplificador `ampReverb`).

Agregue una interfaz gráfica de usuario para manipular los cuatro parámetros y experimente con distintas combinaciones. Trate de encontrar los tiempos de retardo adecuados para que el sonido de la reverberación sea lo más uniforme posible; es decir, que no se perciban ecos o modulaciones cíclicas en intensidad, particularmente cuando el tiempo de reverberación es relativamente largo.

24.6 Evaluación y reporte de resultados

1.- Considere que el sistema de reflexiones primarias mostrado en la Figura 24.2 se aplica a una señal de entrada $x[n]$ de un solo canal, haciendo $x_1[n] = x[n]$ y $x_2[n] = 0$. Muestre que el sistema es equivalente a un filtro FIR encontrando una expresión matemática para la respuesta al impulso del filtro FIR correspondiente al caso para tres bloques de reflexión primaria en serie (Figura 24.2b).

2.- Agregue a la UGen `Reverb` un método para establecer el factor de retroalimentación en términos de la duración RT60 de la reverberación dada en segundos como argumento.

3.- Obtenga una expresión para el tiempo de reverberación RT60 del sistema mostrado en la Figura 24.3, en función del tiempo promedio d de los retardos, la ganancia G de retroalimentación y la frecuencia ω, tomando en cuenta el efecto del filtro pasa-bajas.

4.- Un efecto interesante se produce cuando la ganancia de retroalimentación del reverberador se fija a $G = 1$, obteniendo así una reverberación infinita. En este caso, si la señal de entrada mantiene una cierta intensidad, muy posiblemente saturará las líneas de retardo, creando un muro sónico poco legible. Sin embargo, si solo se deja pasar un fragmento corto de la señal de entrada (alrededor de medio segundo), el reverberador "capturará" el sonido y lo mantendrá por un tiempo indefinido. Pruebe a implementar este efecto, a partir del programa de prueba ya elaborado, insertando un VCA entre la entrada de audio y el reverberador para controlar la intensidad de la señal que llega al reverberador. Utilice una envolvente para controlar la ganancia del VCA, de manera que al presionar un botón o la barra espaciadora la envolvente se abra por una fracción de tiempo (e.g., 500 ms). Mantenga la retroalimentación al 100% ($G = 1$) para congelar el sonido, y manipule el valor del polo para permitir que el sonido se atenúe y desaparezca gradualmente.

24.7 Retos

Agregue al reverberador (Figura 24.3) la capacidad de modular los tiempos de retardo mediante una señal externa. En este caso, por simplicidad todos los tiempos de retardo serán modulados por la misma señal, pero el índice de modulación puede ser distinto para cada línea de retardo (incluso puede ser cero para algunos retardos). Agregue al sistema un LFO o ruido de baja frecuencia para modular los tiempos de retardo, junto con los elementos de interfaz de usuario para manipular todos los parámetros. Experimente con el sistema y describa el tipo de sonidos que puede producir.

24.8 Conclusiones

Redacte en el recuadro sus conclusiones acerca de los conceptos aprendidos, las actividades realizadas y los objetivos planteados en esta práctica.

Parte IV

Proyectos

Capítulo 25

Aplicaciones biomédicas

En el área de la Ingeniería Biomédica, el procesamiento de señales e imágenes digitales ha tenido un gran impacto en áreas como el desarrollo de sistemas de apoyo al diagnóstico médico, el estudio y caracterización del funcionamiento de diversos sistemas biológicos, el diseño de instrumentos y dispositivos médicos para mejorar la calidad de vida de personas, etc. En la actualidad es común el uso de señales electrofisiológicas, como el electrocardiograma o el electroencefalograma, así como de imágenes de tomografía computarizada o resonancia magnética, para el diagnóstico y seguimiento de enfermedades. Sin embargo, en algunos casos es también posible utilizar señales de audio, lo cual tiene importantes ventajas ya que no se requiere de equipo especializado para su adquisición, y en muchos casos se pueden adquirir de manera no invasiva ni obstructiva.

A continuación se describirán algunas de estas aplicaciones.

25.1 Análisis de voz

El tracto vocal humano tiene la capacidad de producir y articular un rango muy amplio de sonidos, tanto vocales como consonantes, y además es capaz de modular la voz de diversas maneras, por ejemplo modulando la altura o tono de la voz, su intensidad y la velocidad del habla. Estas modulaciones permiten a la voz acarrear información adicional (además del habla misma) que puede revelar la edad y sexo de una persona, y sugerir emociones como calma, tristeza, enojo, o sorpresa.

Existen también diversas condiciones médicas que pueden afectar la voz, ya sea por problemas o lesiones en algún elemento del tracto vocal (cuerdas vocales, laringe, lengua, paladar, etc), o por desórdenes psicofisiológicos como la enfermedad de Parkinson, el síndrome de Tourette o bipolaridad. De esta manera, el análisis de voz puede tener aplicaciones en la detección de emociones, así como ayudar al diagnóstico de las patologías mencionadas.

En este tipo de aplicaciones, el procesamiento de señales se utiliza por lo general para extraer un conjunto reducido de rasgos que puedan después ser uti-

lizados para diseñar un sistema de clasificación. En esta sección nos enfocaremos a la extracción de algunos rasgos de uso común. El diseño e implementación de clasificadores es un tema que queda fuera del alcance de este documento.

Es importante tener en cuenta que las propiedades de la voz y el rango de modulaciones varía mucho de una persona a otra. Por ejemplo, la frecuencia fundamental de los sonidos vocales suele tener un rango de aproximadamente cuatro octavas, pero el rango individual para cada persona es de aproximadamente dos octavas, y dependerá del sexo, edad y condiciones genéticas de la persona. Por este motivo, en algunas aplicaciones y para algunos rasgos puede ser conveniente normalizar el rasgo; es decir, establecer una norma individual a partir de un estado neutro, y calcular rasgos relativos a esta norma.

25.1.1 Segmentación

Las propiedades de la voz son muy dinámicas y pueden presentar cambios drásticos a lo largo del tiempo. Estos cambios pueden ocurrir de manera gradual o de manera abrupta. Por esta razón es común estimar los rasgos en segmentos de corta duración. Estos segmentos pueden ser de duración fija o de duración variable. Para el caso de segmentos de duración fija, la duración determina la resolución del análisis y suele estar entre 10 y 50 ms. En un sistema de procesamiento en tiempo real puede fijarse el tamaño de buffer a la resolución deseada (por ejemplo, 1024 muestras para una frecuencia de muestreo de 44.1 KHz equivale a 23 ms), y calcular un vector de rasgos por cada bloque de procesamiento.

En caso de utilizar segmentos de duración variable, lo mas común es tratar de segmentar los fonemas que componen la señal. La manera mas sencilla de hacer esto consiste en calcular la envolvente de amplitud y establecer un umbral de amplitud para definir los segmentos.

Antes de proceder a calcular los rasgos puede ser necesario pre-procesar la señal de entrada, por ejemplo para eliminar la componente DC y el exceso de ruido (mediante filtros adecuados) y en algunos casos normalizar la amplitud de la señal para reducir la influencia de la distancia entre el sujeto y el micrófono, y de la ganancia del pre-amplificador del micrófono.

25.1.2 Rasgos estadísticos

Entre los rasgos mas comunes a utilizar se encuentran la amplitud o energía promedio de la señal, así como la variabilidad de la misma a lo largo del tiempo (*shimmer*); la frecuencia fundamental $F0$, la cual por lo general se estima mediante técnicas basadas en auto-correlación, el periodo fundamental correspondiente $T0$, y su variación a lo largo del tiempo (*jitter*). En caso de segmentar por fonemas, la duración de los fonemas puede también ser de utilidad.

La variabilidad en la amplitud y en el periodo fundamental son de gran utilidad para caracterizar la estabilidad en el habla, por ejemplo durante emisiones de vocales largas. Estos rasgos suelen calcularse a partir de las diferencias hacia atrás de la amplitud y periodo fundamental en segmentos consecutivos, o bien

midiendo el error de ajuste de un modelo polinomial ajustado a los rasgos de varios segmentos consecutivos.

Otro rasgo interesante es el número de cruces por cero. En el caso de la voz, cuyo timbre es muy complejo, el número de cruces por cero no es útil para estimar la frecuencia fundamental, pero sí está correlacionado con el tipo de sonido o fonema que se emite. Un número relativamente alto de cruces por cero suele asociarse con fonemas sordos (e.g., /p/, /t/, /s/, /f/), mientras que los fonemas sonoros producen un menor número de cruces por cero [23].

Finalmente, dependiendo de la aplicación puede ser necesario obtener estadísticos (promedio, máximo, mínimo, desviación estándar, etc.) de los rasgos a lo largo de múltiples segmentos; por ejemplo, para caracterizar la estabilidad del habla en pacientes con Parkinson, uno puede pedir al paciente que emita una vocal durante un periodo relativamente largo, digamos tres segundos, posteriormente se divide la señal en segmentos cortos y se calcula el periodo fundamental para cada segmento y su variabilidad; al final, se estiman los rasgos promedio durante los tres segundos que dura el episodio completo.

25.1.3 Rasgos espectrales

El tracto vocal actúa como una cámara de resonancia que enfatiza frecuencias específicas. Estas frecuencias son llamadas *formantes* y varían dependiendo de la forma y dimensiones del tracto vocal, lo cual incluye la forma y posición de la boca, lengua y paladar, por lo cual las formantes pueden modularse de manera consciente para emitir sonidos vocales. Las frecuencias formantes suelen denotarse en orden creciente por $F1$, $F2$, etc. Por lo general se consideran dos o tres formantes, y en la literatura se pueden encontrar tablas con las frecuencias promedio asociadas con las formantes de las distintas vocales [23, 11].

La obtención de las formantes puede realizarse mediante técnicas similares a las utilizadas para obtener la frecuencia fundamental $F0$; éstas incluyen la autocorrelación y la estimación de la densidad de energía espectral, o bien mediante codificación predictiva lineal, la cual consiste a grandes rasgos en ajustar a la señal un modelo autoregresivo variante en el tiempo, el cual representa un filtro IIR cuyos polos corresponden a las frecuencias resonantes del sistema.

Otro método de análisis espectral comúnmente utilizado en señales de audio se conoce como *cepstrum* (palabra que se obtiene al invertir la primer mitad de *spectrum*). El cepstrum se obtiene típicamente como la transformada inversa de Fourier del logaritmo del espectro de energía de una señal. Matemáticamente, se expresa como

$$\text{cepstrum}_x(q) = \mathcal{F}^{-1}\left\{\log\left|\mathcal{F}\{x(t)\}\right|^2\right|\right\},$$

donde $x(t)$ es la señal de entrada, $\mathcal{F}$ representa la transformada de Fourier y $\mathcal{F}^{-1}$ su inversa. La potencia del cepstrum se define entonces como el cuadrado de la magnitud $|\text{cepstrum}_x(q)|^2$. En la práctica, el cepstrum se calcula en segmentos de tiempo cortos, aplicando primero una función de ventana a la señal de entrada x (e.g., Hamming).

Note que la variable independiente q del cepstrum corresponde al tiempo, al igual que la variable t; sin embargo, q se interpreta como un periodo, de manera que si el cepstrum presenta un pico en $q = q_0$, entonces $x(t)$ contiene un tono cuya frecuencia es $f_0 = f_m/q_0$, donde f_m es la frecuencia de muestreo. La motivación radica en que al existir un tono con frecuencia f_0 en $x(t)$, entonces el espectro de x mostrará picos en f_0 y en sus armónicos $2f_0$, $3f_0$, etc., lo cual corresponde a una oscilación periódica (en el espectro) con periodo f_0. Al tomar la transformada (inversa) de Fourier del espectro, podremos entonces observar un pico en el periodo fundamental del tono. Por supuesto, esta técnica puede utilizarse para estimar la frecuencia fundamental $F0$, mas no necesariamente las formantes ya que estas no presentan armónicos.

25.2 Análisis de fonocardiogramas

El sistema cardíaco produce sonidos audibles a partir de la apertura y cierre de las válvulas cardíacas, la vibración de los músculos cardíacos, y el flujo sanguíneo. Estos sonidos suelen escucharse mediante un estetoscopio como parte del proceso estándar de auscultación clínica, durante el cual el médico escucha cuidadosamente para detectar anormalidades en el patrón de sonidos [32].

En la actualidad, existen estetoscopios electrónicos que no solo permiten al médico escuchar los sonidos cardíacos, sino que son capaces de adquirir y procesar la señal de audio digitalmente para proporcionar información adicional. A la adquisición y procesamiento de las señales acústicas producidas por el sistema cardíaco se les conoce como *fonocardiografía* (FCG).

En sus inicios, las aplicaciones clínicas de la fonocardiografía se vieron rápidamente superadas por la electrocardiografía (ECG), ya que el análisis se realizaba de manera visual y la señal de ECG es considerablemente mas simple. Sin embargo, los avances tecnológicos en cómputo y procesamiento digital de señales abren la posibilidad de que la fonocardiografía llegue a tener un impacto clínico en el futuro. En los últimos años, se ha estudiado la aplicación del FCG para el monitoreo fetal [33, 34], así como para la detección de anormalidades [35].

El análisis de fonocardiogramas se realiza por lo general utilizando técnicas clásicas de procesamiento de señales. En primera instancia, las señales son preprocesadas mediante filtros para eliminar el DC, eliminar el hum producido por la fuente de voltaje (usualmente mediante un filtro rechaza-banda entonado a 50 o 60 Hz), y en general reducir ruido y limitar el ancho de banda al que corresponde a los sonidos cardíacos, el cual va de 20 Hz hasta alrededor de 1 KHz [32], aunque en varias aplicaciones la información de interés está por debajo de los 300 Hz [35].

El siguiente paso consiste en segmentar la señal en los diferentes sonidos que conforman el fonocardiograma. De acuerdo con Abbas y Bassam, durante el ciclo cardíaco normal se generan cuatro sonidos. El primero de ellos (S1) ocurre al inicio de la sístole (contracción) ventricular y se caracteriza por tener amplitud y duración mayores que la de los otros sonidos; su duración promedio está entre

los 100 y 200 ms con un ancho de banda entre 10 y 200 Hz. El segundo sonido (S2) ocurre una vez que comienza la diástole (relajación) ventricular y coincide con la finalización de la onda T en el electrocardiograma. S2 contiene dos componentes de alta frecuencia (usualmente de mayor frecuencia que las componentes de S1) separadas por un breve periodo de tiempo (menor a 30 ms). En casos normales, estos dos sonidos (S1 y S2) son audibles a través del estetoscopio y ofrecen rasgos importantes para el diagnóstico, por lo cual la mayoría de las aplicaciones de FCG se enfocan en ellos. El tercer y cuarto sonidos (S3 y S4, respectivamente) corresponden a vibraciones de baja frecuencia de las paredes ventriculares, y están asociados con condiciones muy variables, tanto normales como patológicas [32].

La segmentación de los sonidos se realiza generalmente a través de una descomposición tiempo-frecuencia donde se revelan las características espectrales y de energía de los sonidos. Una vez segmentados los sonidos, puede calcularse su duración y utilizarla como un rasgo adicional para clasificar cada sonido como uno de los cuatro posibles (S1, S2, S3 y S4), tomando en cuenta la dinámica con la cual pueden ocurrir unos después de otros. Para la descomposición tiempo-frecuencia pueden utilizarse diversas herramientas como: la transformada de Fourier en tiempo corto, bancos de filtros pasa-banda, análisis de ondeletas, modelos autoregresivos variantes en el tiempo, etc.

Una vez segmentados los sonidos se reconstruye la dinámica de los ciclos cardíacos. A partir de esta dinámica pueden estimarse, por ejemplo, la frecuencia cardíaca, o rasgos que permitan discriminar entre casos normales y patológicos. Por lo general estos rasgos pueden obtenerse a partir de la misma descomposición tiempo-frecuencia ya realizada, por ejemplo, la energía en distintas bandas de frecuencia, la duración e intensidad de los distintos sonidos que conforman el FCG, y otros.

El repositorio público Physionet contiene varias bases de datos de sonidos cardíacos, incluyendo FCG fetal y casos patológicos [36, 37, 38].

25.3 Análisis de ronquidos

Una aplicación interesante en la actualidad es la detección y clasificación de ronquidos y jadeos durante el sueño, lo cual puede apoyar en el diagnóstico y seguimiento de la calidad del sueño y algunas patologías del sueño como el síndrome de apnea e hipoapnea del sueño (SAHS). El estudio clínico del sueño se realiza comúnmente a partir de registros polisomnográficos, los cuales incluyen el registro del electroencefalograma (EEG), electrocardiograma (ECG), electrooculograma (EOG), electromiograma (EMG), señales respiratorias y otras. Un estudio polisomnográfico requiere de equipos especializados por lo que debe realizarse en una clínica especializada con personal experto, lo cual conlleva un gasto significativo para el paciente.

Una manera de reducir los costos e inconvenientes asociados al diagnóstico de SAHS, consiste en detectar y clasificar sonidos de ronquidos a partir de señales de audio de la respiración del paciente, adquiridas durante la noche.

Estas señales pueden adquirirse mediante micrófonos convencionales para voz, incluyendo aquellos que se encuentran integrados en dispositivos como teléfonos inteligentes, tabletas y computadoras portátiles, por lo que es posible implementar sistemas para la adquisición y análisis de ronquidos que sean prácticamente ubicuos.

Ya que los ronquidos y otros sonidos respiratorios se producen en el tracto vocal, el análisis y clasificación de los mismos se realiza mediante las mismas herramientas que se utilizan para el análisis de voz. En el dominio temporal es necesario segmentar los sonidos individuales, para luego estimar rasgos para cada sonido, tales como la duración, intensidad, variabilidad de la intensidad (shimmer), etc. En el dominio de la frecuencia se suelen estimar la frecuencia fundamental, las formantes, las frecuencias mínima, máxima y central, jitter, etc. Estos rasgos se obtienen a partir de técnicas como la transformada discreta de Fourier, el cepstrum, y el ajuste de modelos autoregresivos, entre otras [39, 40, 41].

El ronquido es uno de los principales síntomas del SAHS; sin embargo, la existencia de ronquidos no implica la existencia de la patología. Ronquidos constantes y estables, con pocas interrupciones, pueden ser normales; por otro lado, ronquidos irregulares son característicos del reinicio de la respiración después de un episodio de apnea. Es importante que el sistema de análisis sea capaz de distinguir entre ambos casos, y para esto se han realizado estudios donde se evalúan y describen las diferencias espectrales entre ambos tipos de ronquidos [42, 43, 44, 45]. Aún cuando existe un número considerable de trabajos previos con resultados prometedores, el diseño y validación a gran escala de un sistema de diagnóstico clínico basado en el análisis de ronquidos continúa siendo un problema abierto [46].

25.4 Interfaces hombre-máquina basadas en audio

En décadas recientes, el estudio de la interacción entre personas y máquinas (principalmente computadoras) ha cobrado un auge impresionante y se ha convertido en un campo de estudio multidisciplinar que involucra áreas como Diseño, Psicología, Ciencias de la Computación, Lingüística, Neurociencias y otras. Muchos dispositivos de uso común como automóviles, teléfonos inteligentes, computadoras, consolas de videojuegos e incluso algunos electrodomésticos, ya cuentan con la capacidad de emitir información mediante una voz sintetizada, recibir comandos por medio de la voz del usuario, o por medio de pantallas táctiles. Estos avances representan un largo camino desde las interfaces basadas en la línea de comandos de una terminal y las primeras interfaces gráficas basadas en íconos y sistemas de menús, las cuales, siguen en uso.

El sonido siempre ha sido una parte importante de las interfaces humano-computadora (HCIs). Incluso antes de que las computadoras contaran con capacidades multimedia, el *click* audible que se produce al pulsar una tecla en un

teclado o un botón en un mouse proporcionan una retroalimentación al usuario que confirma la acción realizada. Tan importantes son estos sonidos para algunos usuarios que los sistemas operativos modernos para dispositivos táctiles tienen la opción de emitir un click artificial cuando se pulsa una tecla del teclado virtual en la pantalla táctil. En la actualidad, estos dispositivos son capaces de emitir *íconos auditivos* (también llamados *earcons*) para indicar distintos eventos, como la recepción de un correo electrónico o mensaje de texto, una señal de baja batería, etc. Comúnmente, muchos de estos íconos auditivos consisten en sonidos pregrabados y suelen ser muy estáticos. Sin embargo, las computadoras actuales son lo suficientemente rápidas como para sintetizar sonidos en tiempo real, lo cual abre el panorama para el diseño de íconos auditivos interactivos cuyas propiedades varíen de acuerdo a diversos parámetros [47].

Para algunas aplicaciones, el uso de sonido en una HCI puede jugar un papel mas importante, o incluso primordial. Por ejemplo,

- **Diseño de interfaces y sistemas de guía para personas invidentes.** Por ejemplo, lectores de pantalla, traductores de Braille, sistemas de realidad aumentada y espacialización de sonido [48, 49].

- **Interacción conductor-vehículo.** Actualmente, muchos automóviles ya incorporan tecnología de reconocimiento de voz, que proporciona al conductor un medio para seleccionar destinos, reproducir música o estaciones de radio, realizar ajustes y otros comandos, y al mismo tiempo mantener la atención en el camino; por otra parte, la síntesis de voz permite al sistema comunicarse verbalmente con el usuario para proporcionarle instrucciones y notificaciones. En un futuro próximo, el sistema podría ser capaz de reconocer emociones en la voz del conductor y ajustar los parámetros de conducción de manera acorde, así como impartir emociones en la síntesis de voz para enfatizar las instrucciones al conductor [50, 51].

- **Interfaces basadas en audio para personas con discapacidades motrices.** Existe una gran cantidad de interfaces para personas cuyo movimiento de las extremidades es limitado o nulo. Algunas interfaces están basadas en señales electroencefalográficas (interfaces cerebro-computadora o BCIs), pero requieren equipo muy especializado y son difíciles de controlar por lo que su impacto ha sido limitado. Otras se basan en el movimiento de los ojos; también requieren equipo especializado pero son mas fáciles de controlar. Para aquellas personas capaces de articular sonidos, las interfaces basadas en audio son en general mas prácticas y fáciles de utilizar, y solamente requieren de tecnología ubicua [52].

Capítulo 26

Aplicaciones de síntesis de sonido

26.1 Introducción

En la actualidad, las aplicaciones multimedia se pueden encontrar en diversos ámbitos como el entretenimiento, el arte, la educación, la publicidad, y el modelado y simulación científicos. El desarrollo de aplicaciones multimedia usualmente involucra la colaboración de expertos en diversas áreas, tanto técnicas como humanísticas y artísticas [53].

Una aplicación multimedia presenta información al usuario utilizando múltiples medios, los cuales pueden incluir texto, imágenes, video, animaciones, sonidos y música. En años recientes se ha incluido información táctil por medio de dispositivos hápticos, movimiento, y efectos ambientales como viento, lluvia, humo, etc., particularmente en aplicaciones de realidad virtual, realidad aumentada y cinematografía. A su vez, las aplicaciones multimedia pueden ser interactivas o no interactivas. Para aquellas que son interactivas, la interacción puede realizarse también a través de distintos medios, entre los que se incluyen controladores táctiles (teclados, mouse, gamepads, joysticks, etc.), sistemas de visión computacional (cámaras, sensores de profundidad, seguidores de mirada), y audio (voz, gestos audibles, etc.).

El audio es posiblemente el segundo elemento mas importante en una aplicación multimedia, después de los medios visuales como imágenes y video. Los jóvenes entre 8 y 18 años pasan en promedio de dos a tres horas al día escuchando música y otros medios audibles, y esta cifra se ha incrementado gradualmente gracias a la proliferación de dispositivos móviles que permiten reproducir medios digitales [54]. A continuación se enumeran algunas de las ventajas de utilizar música y sonidos en una presentación multimedia:

- Proporciona una experiencia más inmersiva
- Permite evocar distintos tipos de emociones en el espectador

- Permite comunicar información al espectador mediante lenguaje natural
- Permite transmitir información sin requerir de atención visual (esto es de particular importancia en aplicaciones para niños pequeños o personas con cierto tipo de discapacidades).

En un buen número de aplicaciones multimedia, el audio se produce o se graba en archivos digitales antes de la ejecución de la aplicación. La aplicación se encarga de reproducir los fragmentos de audio en el momento adecuado, aplicando, si acaso, algunos procesos básicos como control de amplitud, mezcla, espacialización y reverberación artificial. Otras aplicaciones utilizan síntesis de sonido en tiempo real para generar el audio y posteriormente espacializar y mezclar los distintos sonidos que ocurren. Cada uno de estos enfoques tiene sus ventajas y desventajas; por ejemplo, el uso de sonidos pre-grabados permite utilizar sonidos más realistas, pero que sin embargo son estáticos; en contraste, el uso de síntesis de sonido permite controlar cada aspecto de la generación del mismo, permitiendo que el sonido se modifique de acuerdo a las condiciones del objeto o acción que representa. Por supuesto, es posible combinar ambos enfoques en una misma aplicación [55, 56].

26.2 Sintetizadores virtuales

Una buena manera de familiarizarse con las diferentes técnicas de síntesis de sonido, así como con su uso para el diseño de sonidos, es implementando aplicaciones de sintetizadores virtuales. Estas aplicaciones implementan un motor de síntesis razonablemente versátil, así como los elementos de interfaz de usuario para manipular los parámetros y crear distintos sonidos. Los sonidos pueden ser eventualmente grabados como archivos digitales para su uso en otras aplicaciones multimedia, o bien el motor de síntesis puede implementarse dentro de una aplicación para generar y manipular los sonidos en tiempo real.

Existe una gran cantidad de diseños y arquitecturas de sintetizadores que han sido utilizados una y otra vez convirtiéndose en estándares probados. A continuación se describen brevemente algunas de estas arquitecturas. La manera mas sencilla de encontrar los detalles para un modelo específico es consultando el sitio Web del fabricante y/o los manuales correspondientes.

- **Sintetizadores sustractivos.-** La arquitectura mas común consta de dos o tres osciladores entonados de manera independiente, los cuales se combinan en un mezclador junto con una fuente de ruido. La salida del mezclador pasa por un filtro pasa-bajas o multi-modo (un filtro multimodo permite seleccionar una salida pasa-bajas, pasa-altas o pasa-banda), y posteriormente por un amplificador para controlar la ganancia final. Algunos sintetizadores permiten la interacción entre osciladores de manera que uno de ellos pueda modular la frecuencia o amplitud de otro, sincronizarse entre sí, o bien, pasando ambos osciladores por un modulador en anillo cuya salida alimenta también al mezclador. Además, el sintetizador suele contar con por lo menos dos envolventes, una para modular

la amplitud y otra para modular la frecuencia de corte del filtro (aunque también es posible modular la frecuencia base de los osciladores), así como uno o dos osciladores de baja frecuencia (LFOs) cuya salida se puede rutear a distintos destinos de modulación.

- **Sintetizadores basados en modulación de frecuencia.-** Los sintetizadores FM comerciales suelen tener 4, 6 u 8 operadores por voz. Recordemos que un operador es un bloque formado por un oscilador (cuya frecuencia puede ser modulada por la salida de otro operador) que pasa por un amplificador controlado por una envolvente. Los operadores pueden interconectarse en configuraciones predefinidas llamadas *algoritmos*. Por ejemplo, en la Figura 26.1 se muestran los ocho algoritmos que se implementan en varios modelos de Yamaha, donde el flujo de las señales va hacia abajo. Note que el algoritmo #5 corresponde a la superposición de dos parejas modulador-portador, es decir que se producen dos sonidos simples de manera simultánea; por otro lado, el algoritmo #8 se asemeja más a un sintetizador aditivo. De esta manera, los sintetizadores FM son capaces de producir sonidos muy complejos y variados.

- **Sintetizadores basados en muestras y tablas de ondas.-** Una alternativa computacionalmente eficiente para generar sonidos realistas consiste en reemplazar los osciladores por bloques que reproduzcan sonidos previamente digitalizados (conocidos como *muestras*) cuya salida suele pasar por una arquitectura sustractiva; es decir, filtros y amplificadores modulados por envolventes y/o LFOs, además de efectos como retardos y reverberación. Los osciladores basados en muestras son capaces de variar la velocidad con la que se reproduce una muestra para alterar su tono. Por ejemplo, si se tiene una grabación de la nota Do4 de un piano, al reproducirla al doble de velocidad se obtendría el Do5, mientras que a la mitad de la velocidad se obtiene un Do3. Por supuesto, esto altera también la duración de la nota; para solventar esto, se suelen usar diversos trucos, por ejemplo, reproducir de manera cíclica una sección de la muestra durante la etapa de sostenimiento, o bien, utilizar una muestra distinta para cada nota o intervalo de notas.

26.2.1 Modulación

Un aspecto fundamental de la síntesis de sonido es la modulación, ya que es ésta la que permite crear sonidos no estáticos, cuyas propiedades van cambiando a lo largo del tiempo. La implementación de modulación conlleva dos aspectos: el primero consiste en agregar bloques que generen señales moduladoras (envolventes, osciladores de baja frecuencia, secuenciadores, etc), mientras que el segundo consiste en implementar las conexiones que permiten rutear la salida de un modulador hacia un destino específico.

Como mínimo, se acostumbra incluir un par de envolventes para modular la amplitud final del sonido, así como el contenido armónico del mismo (lo cual se

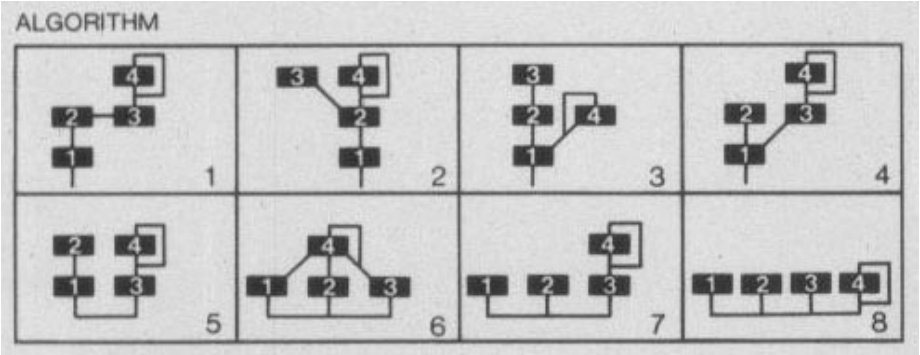


Figura 26.1: Algoritmos para síntesis FM de 4 operadores implementados en varios sintetizadores Yamaha. Fuente: Manual de referencia del sintetizador Yamaha TX81z.

percibe como la brillantez del sonido). En el caso de un sintetizador sustractivo, el contenido armónico se controla modulando la frecuencia de corte del filtro; para un sintetizador FM, lo que debe modularse es la amplitud de la salida de aquellos operadores que fungen como moduladores. También es común incorporar por lo menos un oscilador de baja frecuencia (LFO) que pueda modular el tono de los osciladores principales, la amplitud o el contenido armónico, para producir efectos de vibrato, trémolo o wah, respectivamente.

Sintetizadores mas sofisticados pueden incorporar una amplia selección de moduladores, entre los que se incluyen tres o cuatro envolventes, tres o cuatro LFOs, controladores contínuos como perillas, deslizadores, ruedas de pitch y modulación, joysticks, pedales de expresión, sensores de presión y otros, controladores discretos como botones, interruptores y pedales de switch, así como información que se produce al inicio de cada sonido como el número de nota o la fuerza con la que se presiona una tecla o botón. Cualquiera de estos moduladores puede rutearse hacia uno o más destinos utilizando lo que se denomina una *matriz de modulación*, la cual consta de un cierto número de registros, cada uno de los cuales define una conexión de una fuente (un modulador) hacia un destino (parámetro a modular), con un cierto índice de modulación (profundidad del efecto de modulación). Los destinos son todos los parámetros modulables, por ejemplo el tono (pitch) del sonido o de cada oscilador individual, la ganancia de cada amplificador o de las entradas de un mezclador, las frecuencias de corte de los filtros (y en algunos casos, también el nivel de resonancia), los parámetros de los efectos, e incluso las mismas propiedades de los moduladores (amplitudes, frecuencia de un LFO, tiempos de una envolvente, etc). La capacidad de modular las propiedades de las señales moduladoras que a su vez están modulando a otros parámetros, incrementa enormemente la versatilidad de un sintetizador, pero también su complejidad de operación, ya que se requiere incorporar elementos de interfaz de usuario para manipular todos los parámetros involucrados [57].

26.2.2 Control externo

Un aspecto importante que debe considerarse al diseñar un sintetizador de uso general es la manera en que el usuario podrá interactuar físicamente con el sintetizador para disparar los sonidos y manipular los parámetros de síntesis. Los dispositivos mas convencionales incluyen teclados estilo piano y pads para disparar distintas notas o sonidos, así como controladores basados en perillas o deslizadores para manipular los parámetros. Existe un gran número de controladores en el mercado que operan bajo el estándar MIDI (Musical Instrument Digital Interface), el cual define un protocolo de comunicación que permite enviar instrucciones al sintetizador para activar una nota específica o variar algún parámetro. El Apéndice F describe mas a fondo el protocolo MIDI y cómo implementarlo en Processing.

En algunas aplicaciones, por ejemplo aquellas orientadas a la interpretación experimental y a las instalaciones sonoras, puede ser interesante buscar otras formas de control; por ejemplo, el uso de señales electrofisiológicas como la electromiografía (EMG) o electroencefalografía (EEG), el uso de sensores de movimiento (los cuales pueden ser manipulados por el intérprete, o bien, colocarse en objetos en libre movimiento), el uso de sistemas de visión computacional, y otros más. En estos casos, puede implementarse un subsistema dedicado a adquirir y pre-procesar las señales de control para posteriormente enviarlas al sintetizador (o convertirlas en mensajes MIDI). Este subsistema puede implementarse como parte del software que realiza la síntesis, o como un dispositivo aparte (por ejemplo, basado en Arduino). El pre-procesamiento de las señales de control también se realiza en tiempo real y usualmente involucra filtrado y posiblemente la detección de ciertos eventos específicos.

26.2.3 Secuenciadores

En su forma mas simple, un secuenciador genera una señal de control constante a trozos, cuya salida puede utilizarse para modular algún parámetro de síntesis. El secuenciador cuenta además con una entrada que recibe una señal de pulsos (llamada *reloj*). Cada vez que el secuenciador recibe un pulso, el valor de salida del secuenciador cambia y se mantiene constante hasta recibir el siguiente pulso de reloj. Los valores de salida del secuenciador se encuentran almacenados de manera secuencial, digamos $s_1, s_2, \ldots, s_n$, donde n es el número de *pasos* con los que cuenta el secuenciador. Si la salida actual es s_k, entonces al recibir un pulso de reloj la salida se actualizará a s_{k+1}, y así de manera cíclica; es decir, después de emitir el valor s_n, el secuenciador regresará a s_1. Por lo general, el usuario puede manipular los valores s_k para obtener la secuencia deseada.

Un uso común de los secuenciadores consiste en usarlos para modular el tono de los osciladores principales, para así construir una melodía simple o paisaje sonoro. Un segundo secuenciador (sincronizado con el primero) puede utilizarse para enviar pulsos que disparen las envolventes. En arquitecturas mas versátiles, el sintetizador puede tener múltiples secuenciadores, cuya salida puede rutearse a cualquier destino, al igual que las demás señales moduladoras.

Secuenciadores mas sofisticados pueden contar con funciones adicionales, por ejemplo, incorporar una señal de entrada que al recibir un pulso reinicie el secuenciador al primer paso, contar con distintas formas de recorrer los pasos (hacia adelante, hacia atrás, de manera aleatoria, etc.), incorporar divisores de reloj para variar su velocidad con respecto a otros secuenciadores, o cuantizar la señal de salida. Algunas de estas funciones pueden implementarse también mediante UGens externas al secuenciador.

26.2.4 Polifonía y multitimbralidad

Un sintetizador *polifónico* es aquél que puede producir más de una nota o sonido de manera simultánea. La polifonía permite reproducir acordes (grupos de notas simultáneas) o reproducir un sonido nuevo antes de que concluya la etapa de relajación de un sonido previo. A cada uno de los sonidos que se escuchan simultáneamente en una composición se le conoce como *voz*. La manera mas simple de implementar polifonía en un sintetizador digital consiste en construir una cadena idéntica de UGens por cada voz (por ejemplo, utilizando arreglos), e implementar un sistema de asignación de voces que asigne una de las voces disponibles (es decir, una voz que no esté sonando en ese momento) al siguiente sonido a reproducir. En caso de que todas las voces estén ocupadas, el esquema de asignación decidirá si reemplaza alguna de las voces en curso por el nuevo sonido. Un esquema sencillo de asignación consiste en asignar voces de manera consecutiva y cíclica, comenzando por la voz 1 hasta la voz N (donde N es el número máximo de voces), y posteriormente asignar nuevamente la voz 1. Algunos sintetizadores cuentan con esquemas de asignación dinámica, donde las conexiones entre las UGen se realizan en tiempo de ejecución según lo requiera el parche[1]. Por ejemplo, si un oscilador no está siendo utilizado, entonces puede desconectarse de la cadena de UGens para que éste no consuma tiempo de procesamiento. En este tipo de sintetizadores, el número de voces disponibles puede variar dependiendo de la complejidad del parche.

Un aspecto interesante en algunos sintetizadores polifónicos es la capacidad de producir sonidos o timbres distintos de manera simultánea. Por ejemplo, en un videojuego se puede implementar un sintetizador con ocho voces, donde cuatro de ellas se utilicen para la música de fondo, y las otras cuatro para generar distintos efectos de sonido como pasos, disparos, explosiones, o el sonido del personaje al saltar. A esta característica se le conoce como *multitimbralidad*, y básicamente significa que cada grupo de voces cuenta con sus propios parámetros de síntesis; en el caso extremo, cada voz puede tener distintos parámetros.

26.2.5 Efectos

Es posible utilizar efectos para realzar, alterar o proporcionar una sensación de localización espacial a los sonidos generados. Sin embargo, algunos efectos tienen un costo computacional relativamente alto (en particular la rever-

[1] En la jerga de los sintetizadores, un *parche* (patch) o *programa* se refiere a la configuración de parámetros de síntesis que generan un sonido específico.

beración). Por esta razón, es común el uso de un solo bloque de efectos para todas las voces, utilizando mezcladores para controlar el nivel de cada voz que llega al procesador de efectos. De esta manera, el sonido de una explosión en un videojuego puede tener mucha reverberación, mientras que al sonido de los pasos del personaje se le asigna poca reverberación.

26.3 Composición algorítmica

La composición algoritmica (también llamada *música generativa*) combina las matemáticas y las herramientas computacionales con la teoría de la música y de la armonía para diseñar algoritmos y heurísticas que generen secuencias de notas (melodías y armonías) que al ser reproducidas por un instrumento o por un sintetizador), el resultado sea interesante y/o agradable al oído; es decir, que desde un punto de vista estético, el resultado pueda considerarse como música. En las últimas décadas se han empleado todo tipo de métodos matemáticos y computacionales para la generación de secuencias musicales, entre los que se incluyen: generadores pseudo-aleatorios, sistemas caóticos, fractales, gramáticas formales, cadenas de Markov, algoritmos evolutivos, y muchos otros [58, 59, 7, 60].

Algunos sistemas de composición algorítmica consideran las notas musicales como los objetos nativos con los que trabaja el algoritmo. Por ejemplo, uno puede definir una cadena de Markov donde las notas son los estados, y para cada una de estas notas se definen las probabilidades de transición hacia otras notas (incluida ella misma) [58]. Otros sistemas parten de la idea de generar secuencias de números utilizando diversas funciones matemáticas (por ejemplo, funciones caóticas), y posteriormente se discretizan estos números y se mapean a las distintas notas o frases musicales que el compositor desea utilizar [59]. También es posible tomar secuencias ya existentes y transformarlas o combinarlas para obtener nuevas secuencias, por ejemplo mediante algoritmos genéticos [60]. Por otra parte, es posible modelar diversos patrones rítmicos y de acentuación, así como manipularlos y transformarlos [61].

Al igual que muchos otros fenómenos, la música contiene patrones recurrentes que dotan a una composición de cierta estructura, pero también contiene variaciones y elementos sorpresivos (al menos la primera vez que uno los escucha) que mantienen el interés de la audiencia [59]. El éxito de un método de composición algorítmica depende en buena parte de encontrar un balance adecuado entre lo familiar y lo impredecible [7].

26.4 Síntesis de audio para aplicaciones multimedia

En algunas aplicaciones multimedia como videojuegos, presentaciones, demostraciones, simulaciones y arte digital, suelen utilizarse modelos procedurales para generar contenido multimedia en tiempo de ejecución, o incluso en tiempo

real. En el aspecto gráfico, es común el uso de fractales para generar texturas, terrenos, montañas, nubes, árboles y otros tipos de objetos; también es común el uso de sistemas de partículas para simular explosiones, fuego, fluidos, humo y otros efectos visuales. La generación procedural de datos no se limita únicamente a la multimedia; en la ciencia también es común generar datos sintéticos a partir de modelos para probar alguna hipótesis.

Las principales ventajas de la generación procedural de datos son las siguientes:

- Se puede generar una gran cantidad de datos sin que se requiera almacenarlos, evitando así las limitaciones de almacenamiento o transferencia del sistema.
- Es posible introducir variaciones controladas y/o aleatorias en los datos que se generan, evitando la aparición espuria de patrones repetitivos.
- Los parámetros de generación pueden variarse de acuerdo al contexto de la aplicación.
- Se pueden ahorrar costos al no requerir la producción de audio en un estudio profesional.

Así mismo, la generación procedural de datos tiene como principales desventajas las siguientes:

- Dado un modelo, los datos generados a partir de este pueden no ser suficientemente realistas.
- Puede ser que no se conozca un modelo para generar cierto tipo de datos, o que el modelo sea demasiado complicado o costoso de implementar.
- El costo computacional, principalmente en términos de uso de CPU, depende del modelo.

Así como es posible generar elementos gráficos mediante modelos procedurales, también es posible utilizar métodos síntesis de sonido para generar parte o la totalidad de los elementos audibles de una aplicación multimedia. Esto puede hacerse ya sea tratando de emular los sonidos mediante un sintetizador con alguna arquitectura convencional (sustractiva, FM, etc.), o bien, modelando (a veces heurísticamente) los aspectos físicos que originan el sonido (colisiones, fricción, resonancia, etc.) [56]. La generación procedural de audio ha encontrado su principal aplicación en los videojuegos [62, 63, 64], así como en demos (demostraciones audiovisuales generadas en tiempo real) donde el tamaño del archivo ejecutable es limitado (usualmente 64 kb o menos) [65, 66], y en simulaciones físicas [67].

26.5 Sonidos retro (breve historia del audio en computadoras)

Curiosamente, los primeros videojuegos utilizaban audio procedural debido a las limitaciones técnicas que impedían la reproducción de audio digital. A estos sonidos se les conoce ahora como *chip sounds* o *sonidos de 8-bits*[2]. Las primeras consolas de videojuegos, así como las primeras computadoras personales, contaban con chips dedicados que producían el audio (entre otras funciones) utilizando métodos de síntesis relativamente rudimentarios y un número limitado de voces. Algunos ejemplos de estos chips son los siguientes:

- 1977: Television Interface Adapter (TIA). Este chip era utilizado en las consolas Atari 2600 y era capaz de generar dos voces, cada una con diferentes tipos de ondas pulso y ruido. El audio se generaba utilizando un divisor de frecuencia de 5 bits (con un reloj de 30 kHz) y con un control de volumen de 4 bits (16 niveles) por cada voz.

- 1979: Pot Keyboard Integrated Circuit (POKEY). Este chip se utilizaba en las computadoras caseras Atari de 8 bits y en algunos videojuegos de arcade. El chip cuenta con cuatro canales de audio, cada uno de los cuales genera una onda a partir de divisores de frecuencia de 8 bits. Pares de canales consecutivos pueden combinarse en un solo canal con un divisor de frecuencia de 16 bits para tener mayor precisión en frecuencia. Además, cada canal es capaz de producir tonos o ruido armónico, y cuenta con un control de volumen de 4 bits.

- 1980: Texas Instruments SN76489. Usado en múltiples consolas de arcade, así como en consolas caseras como ColecoVision y Sega Genesis, este chip cuenta con tres generadores de onda cuadrada y un generador de ruido. Los generadores de tono operan mediante divisores de frecuencia de 10 bits, por lo que tienen una buena precisión en frecuencia, mientras que el generador de ruido utiliza un generador de números aleatorios con retroalimentación, lo que permite obtener ruido con distintas coloraciones. Además, cada voz cuenta con un control de volumen de 4 bits.

- 1982: MOS Technology 6582 / 8580 Sound Interface Device (SID). Utilizado en computadoras Commodore 64 y Commodore 128, este chip presentaba capacidades revolucionarias, ya que es prácticamente un sintetizador completo en un chip. Cuenta con tres osciladores independientes con un rango de 8 octavas y cuatro formas de onda (diente de sierra, triangular, pulso y ruido). Cada oscilador cuenta con un amplificador controlado por una envolvente ADSR independiente, además de un control de ganancia de 4 bits. La salida combinada de los tres osciladores

[2]El término *sonidos de 8-bits* se refiere mas que nada a sonidos que provienen de sistemas basados en arquitecturas de 8 bits, y no a que los sonidos estuvieran muestrados a 8 bits; exceptuando el chip Paula de las computadoras Commodore Amiga, el cual efectivamente genera audio digital con 8 bits por muestra.

pasa por un filtro multi-modo resonante con salidas pasa-bajas, pasa-altas o pasa-banda. Además, los osciladores pueden sincronizarse entre sí (reseteando la fase del oscilador esclavo cuando el oscilador maestro completa un ciclo), o modularse en anillo. El chip SID es uno de los contribuyentes a la generación de escenas de arte underground, y hasta la fecha se fabrican instrumentos y emulaciones, y se crean composiciones basadas en él.

- 1982: Ricoh RP2A03 / RP2A07. Estos son los CPUs utilizados en las consolas Nintendo Entertainment System (NES). Cuentan con cinco canales de audio: dos de ellos generan ondas de pulso con cuatro posibles anchos de pulso (12.5%, 25%, 50% y 75%) y 16 niveles de amplitud; un canal que genera una onda triangular, un canal que genera ruido, y un canal capaz de reproducir audio digital de baja calidad (7 bits). Este último canal era rara vez utilizado debido a que la reproducción de audio digital requería una cantidad considerable de memoria.

- 1984: Yamaha OPL / OPN - Yamaha fabricó una gran cantidad de chips que implementaban síntesis FM de dos y/o cuatro operadores. Cada operador produce una onda senoidal que puede ser modificada mediante rectificación de media onda ú onda completa, y cuya amplitud es modulada por una envolvente ADSR. Estos chips fueron utilizados en diversos productos como videojuegos de arcade, expansiones de audio para consolas de videojuegos y computadoras caseras (principalmente MSX), teclados portátiles (series PSS y PSR de Yamaha), y tarjetas de audio para PC (Creative Labs Sound Blaster, 1989). Algunos modelos mas recientes (1987 en adelante) incorporaban también reproducción de audio digital basada en muestras y tablas de onda.

- 1985: MOS Technology Paula (Ports, Audio, UART and Logic). Este chip formaba parte de las computadoras Commodore Amiga, una plataforma de 16-bits que fue muy popular para videojuegos y aplicaciones creativas, así como en las escenas de arte digital underground. El chip Paula era capaz de producir cuatro canales de audio digital PCM de 8 bits, cada uno con control de frecuencia así como un control de volumen de 6 bits (64 niveles).

A principios de la década de 1990, las computadoras IBM PC y compatibles dominaban el mercado, mientras que computadoras caseras como la Commodore 64, Commodore Amiga, Atari 65XE y Atari ST dejaban de producirse. Empresas como Creative Labs, Media Vision y Advanced Gravis Technologies fabricaban tarjetas de audio para PC capaces de reproducir audio digital de calidad media/alta (8 bits y 16 bits) en sistemas cuya memoria ya no era tan limitada, además de incorporar síntesis FM mediante chips Yamaha OPL2 y OPL3. Eventualmente, los chips de síntesis FM de las tarjetas de audio fueron reemplazados por chips de síntesis basada en muestras y memoria RAM incorporada en la misma tarjeta. En la actualidad, prácticamente cualquier computadora moderna cuenta con una interfaz de audio integrada en la tarjeta madre, con la

capacidad de grabar y reproducir simultáneamente dos o más canales de audio de alta calidad (desde 16 bits / 44.1 kHz hasta 24 bits / 96 kHz).

Dos décadas después de que el audio digital se volviera un componente común en las computadoras personales, el incremento en la velocidad de los CPUs así como en el número de núcleos de procesamiento, permite nuevamente incorporar síntesis de sonido en tiempo real a las aplicaciones multimedia e incluso utilizar sintetizadores virtuales ya existentes en forma de plugins [62].

Apéndice A

Fundamentos de Procesamiento Digital de Señales

En este apéndice se presentan, de manera muy concisa, los conceptos básicos relacionados con el Procesamiento Digital de Señales, enfocándose en el análisis y propiedades de los sistemas lineales e invariantes en el tiempo (LIT), a partir de los cuales se construye gran parte de los módulos (UGens) y sistemas que se presentan en este libro.

Para un estudio mas profundo de los conceptos que aquí se describen, se recomienda consultar los textos de Oppenheim y Schafer, así como de Proakis y Manolakis [13, 6].

A.1 Definiciones básicas

A.1.1 Señales

Una *señal* representa una medición que cambia con respecto a una o más variables independientes. Comúnmente, la variable independiente representa el tiempo, pero también podría representar una posición espacial, la frecuencia, o alguna otra cantidad. Las señales se denotan por medio de funciones matemáticas. Por ejemplo, $T(t)$ podría representar la temperatura de un objeto a lo largo del tiempo, mientras que $I(x, y)$ podría denotar la intensidad de luz en un punto de una imagen.

Las señales pueden clasificarse dependiendo del número y tipo de variables tanto dependientes como independientes. Con respecto a las variables independientes, cuando se tiene una sola de ellas, se dice que la señal es *univariada* o *unidimensional*; de lo contrario es *multivariada* o *multidimensional*. Si las variables independientes pueden tomar valores reales dentro de un cierto intervalo, entonces se trata de una señal *analógica* o *en tiempo continuo*. Por el contrario,

si las variables independientes solo pueden tomar valores en un conjunto discreto, se dice que la señal es *en tiempo discreto* o simplemente *discreta*. Por lo general, se asume que las variables independientes de una señal discreta toman valores enteros. Para distinguir a una señal discreta de una continua, varios autores reemplazan los paréntesis con corchetes (paréntesis rectangulares), por ejemplo, $x[n]$ con $n \in \mathbb{Z}$. Una señal en tiempo discreto cuya variable dependiente toma valores de un conjunto discreto se conoce como una señal *digital*.

Es posible tener un conjunto de señales que dependan de la misma variable independiente, por ejemplo, al tomar mediciones simultáneas $x_1(t)$, ..., $x_n(t)$. En estos casos, uno puede agrupar las señales en un vector $X(t) = [x_1(t), \ldots, x_n(t)]'$ para obtener una señal *multicanal*.

Finalmente, decimos que una señal x es *finita* si su soporte es finito; es decir, si existe un número real T tal que $x(t) = 0$ para $|t| > T$; o bien, en el caso de las señales discretas, si existe un entero N tal que $x[n] = 0$ para $|n| > N$.

A.1.2 Muestreo de señales

En general, las señales en tiempo discreto se obtienen a partir del *muestreo* de señales continuas. Este proceso consiste en registrar el valor de una señal continua en valores específicos (y usualmente equiespaciados) de tiempo. En otras palabras, se puede obtener una señal discreta $x_d[n]$ a partir de una señal analógica $x_a(t)$ tomando $x_d[n] = x_a(nT)$ para valores n enteros, donde a T se le conoce como el *intervalo de muestreo*. Al recíproco del intervalo de muestreo, $1/T$, se le llama *frecuencia de muestreo*.

A.1.3 Sistemas discretos

Un *sistema discreto* es un operador o funcional T que al aplicarse a una señal discreta x, da como salida otra señal discreta y; es decir, $y = T\{x\}$. La señal de salida y depende totalmente de la señal de entrada x y el operador T. Gran parte del área de procesamiento digital de señales se enfoca en el estudio de sistemas discretos, ya que son éstos los que describen la manera en que una señal puede transformarse en otra, con el fin de llevar a cabo alguna aplicación.

Algunas de las principales propiedades que puede tener un sistema discreto son:

- **Linealidad:** Un sistema T es lineal si cumple que $T\{ax+by\} = aT\{x\}+bT\{y\}$ para cualesquiera señales x y y, y escalares a y b.
- **Invarianza en el tiempo:** Un sistema $y = T\{x\}$ es invariante en el tiempo si un retardo a la señal de entrada produce el mismo retardo en la señal de salida; es decir, si $T\{x[n-d]\} = y[n-d]$, para todo d entero.
- **Estabilidad:** Un sistema T es estable si y solo si una señal de entrada acotada produce una salida acotada; es decir, si la existencia de un M positivo tal que $|x[n]| < M$ para todo n implica que existe un N positivo tal que $|T\{x[n]\}| < N$.

- **Causalidad:** Un sistema $y = T\{x\}$ es causal si la salida $y[n_0]$ depende solamente de valores de entrada $x[n]$ con $n \leq n_0$.
- **Sin memoria:** Un sistema $y = T\{x\}$ es sin memoria si la salida $y[n_0]$ depende únicamente del valor de entrada $x[n_0]$.

Las propiedades de linealidad e invarianza en el tiempo (LIT) son de gran importancia, ya que permiten diversas maneras de caracterizar e implementar este tipo de sistemas. Más aún, es posible implementar ciertos sistemas no lineales y/o variantes en el tiempo modelándolos a partir de sistemas LIT.

A.1.4 Señales discretas básicas

Las siguientes señales juegan un papel especial en el analisis de sistemas; en particular de sistemas lineales e invariantes en el tiempo.

- **Impulso discreto:** se denota por la letra δ y se define como $\delta[0] = 1$, y $\delta[n] = 0$ para $n \neq 0$. También se le conoce como *delta de Kronecker*.
- **Sinusoidal:** $x[n] = A\cos(\omega n + \phi)$, o bien $x[n] = A\,\mathrm{sen}(\omega n + \phi)$, donde A es la amplitud, ω es la frecuencia (en radianes por muestra) y ϕ es la fase inicial.
- **Sinusoidal compleja:** $x[n] = A\exp\{j\omega n + \phi\}$. Aplicando la fórmula de Euler, es fácil ver que una sinusoidal real equivale a la suma de dos sinusoidales complejas, y viceversa. Es fácil ver que una sinusoidal con frecuencia ω es idéntica a una sinusoidal con frecuencia $\omega + 2\pi$; por otra parte, una sinusoidal compleja con frecuencia $-\omega$ es simplemente el conjugado de una sinusoidal con frecuencia ω. Por lo tanto, las frecuencias mas altas que pueden representarse en una señal discreta son π y $-\pi$ radianes por muestra.
- **Exponencial real:** $x[n] = A\alpha^n$ con $A \in \mathbb{R}$. Esta señal crece (en valor absoluto) para $|\alpha| > 1$ y decrece cuando $|\alpha| < 1$. Si α es negativo, la señal alterna su signo en cada muestra.
- **Exponencial compleja:** $x[n] = Az^n$ con $z \in \mathbb{C}$. Representando z en forma polar como $z = |z|e^{j\omega}$, es fácil ver que una exponencial compleja es igual al producto de una sinusoidal compleja y una envolvente exponencial. En particular, si $|z| = 1$ se obtiene simplemente una sinusoidal compleja.

A.2 Convolución

Note que cualquier señal discreta se puede representar como una suma de impulsos desplazados en el tiempo:

$$x[n] = \sum_{k=-\infty}^{\infty} x[k]\delta[n-k].$$

Tomando esto en cuenta, podemos ver que la acción de un sistema discreto LIT T sobre una señal x se puede escribir como

$$y[n] = T\{x[n]\} = T\left\{\sum_{k=-\infty}^{\infty} x[k]\delta[n-k]\right\} = \sum_{k=-\infty}^{\infty} x[k]T\{\delta[n-k]\}.$$

Si definimos $h = T\{\delta\}$, entonces por ser T invariante en el tiempo tenemos que

$$y[n] = \sum_{k=-\infty}^{\infty} x[k]h[n-k].$$

La operación anterior se conoce como *convolución* y se denota mediante el operador $*$; es decir, $y = x * h$. La señal h se conoce como la *respuesta al impulso* o *kernel* del sistema, y caracteriza completamente la acción del sistema sobre cualquier señal de entrada.

La convolución tiene varias propiedades importantes: es conmutativa, asociativa, y se distribuye en la suma.

A.2.1 Estabilidad y causalidad en un sistema LIT

Considere un sistema LIT con respuesta al impulso $h[n]$, y una señal de entrada x tal que $|x[n]| \leq M$ para algún $M > 0$. Se puede ver entonces que la salida del sistema está acotada de la siguiente manera:

$$|y[n]| = \left|\sum_{k=-\infty}^{\infty} x[k]h[n-k]\right| \leq \sum_{k=-\infty}^{\infty} |x[k]||h[n-k]| \leq M \sum_{k=-\infty}^{\infty} |h[n-k]|.$$

Por lo tanto, para que un sistema LIT sea estable se requiere que su respuesta al impulso sea *absolutamente sumable*; es decir, que $\sum_{n=-\infty}^{\infty} |h[n]| < \infty$.

También es fácil ver que un sistema LIT es causal si y solo si $h[n] = 0$ para todo $n < 0$.

A.2.2 Sistemas FIR e IIR

Aquellos sistemas LIT para los cuales su respuesta al impulso h es finita se denominan sistemas de respuesta finita al impulso (FIR - finite impulse response). Los sistemas con respuesta al impulso infinita se denominan IIR (infinite impulse response). La distinción es importante ya que las técnicas para analizar e implementar sistemas FIR son distintas a las utilizadas para sistemas IIR. Por ejemplo, es posible implementar un sistema FIR mediante convolución, mas no así un sistema IIR, ya que esto requeriría un número infinito de operaciones.

A.2.3 Correlación y autocorrelación

Una manera de estimar la similitud entre dos señales x y y es mediante la *correlación cruzada*, la cual se define como la señal

$$R_{xy}[l] = \sum_{n=-\infty}^{\infty} x[n]y^*[n-l],$$

donde y^* denota el complejo conjugado de y, y l es un parámetro que especifica el retardo (llamado *lag*) de la señal y con respecto a x.

La correlación cruzada se puede calcular mediante la convolución $R_{xy} = x * \overleftarrow{y}^*$, donde $\overleftarrow{y}^*$ representa a la señal y invertida en el tiempo y conjugada.

Un caso particular de la correlación cruzada se da cuando las señales x y y son la misma. La señal resultante se conoce como *autocorrelación* y se define como

$$R_{xx}[l] = \sum_{n=-\infty}^{\infty} x[n]x^*[n-l].$$

Dado que $|R_{xx}[l]| \leq |R_{xx}[0]|$, donde $R_{xx}[0] = \sum_n |x[n]|^2$ es la potencia de la señal, es posible definir una autocorrelación normalizada como $\rho_{xx}[l] = R_{xx}[l]/R_{xx}[0]$, cuya magnitud está entre cero y uno.

A.3 Respuesta en frecuencia

Considere un sistema LIT con respuesta al impulso h, y una señal sinusoidal compleja como entrada, digamos $x[n] = \exp\{j\omega n\}$. La salida del sistema puede escribirse como $y = x * h = h * x$, lo cual se expande a

$$y[n] = \sum_{k=-\infty}^{\infty} h[k]e^{j\omega(n-k)} = e^{j\omega n} \sum_{k=-\infty}^{\infty} h[k]e^{-j\omega k}.$$

Entonces, si definimos la *respuesta en frecuencia* del sistema $H(e^{j\omega})$ como

$$H(e^{j\omega}) = \sum_{n=-\infty}^{\infty} h[n]e^{-j\omega n},$$

podemos escribir la salida del sistema (para una entrada sinusoidal compleja) como

$$y[n] = H(e^{j\omega})e^{j\omega n}.$$

Por lo tanto, cuando la entrada a un sistema LIT es una sinusoidal compleja, la salida del sistema será igual a la misma sinusoidal compleja multiplicada por un factor $H(e^{j\omega})$ que depende únicamente de la frecuencia ω. En general, este factor es complejo, por lo que puede alterar tanto la amplitud como la fase de la sinusoidal compleja. Además, es fácil ver que la respuesta en frecuencia de un sistema discreto es una función periódica de la frecuencia ω, con periodo 2π,

ya que una sinusoidal discreta con frecuencia ω es idéntica a una con frecuencia $\omega + 2\pi$.

Al igual que la respuesta al impulso, la respuesta en frecuencia de un sistema permite caracterizar por complejo la acción del sistema sobre cualquier señal de entrada. Por ejemplo, suponga que una señal $x[n]$ puede descomponerse como la suma de sinusoidales complejas, cada una de ellas con su propia amplitud, frecuencia y fase inicial, de manera que $x[n] = \sum_k z_k \exp\{j\omega_k n\}$. Entonces, la salida de un sistema LIT con respuesta en frecuencia H está dada por

$$y[n] = \sum_k H(e^{j\omega_k}) z_k e^{j\omega_k n}.$$

A.4 Transformada de Fourier

Muchas secuencias discretas $x[n]$ se pueden descomponer en la suma de funciones sinusoidales mediante una integral de Fourier:

$$x[n] = \frac{1}{2\pi} \int_{-\pi}^{\pi} X(e^{j\omega}) e^{j\omega n} d\omega,$$

donde

$$X(e^{j\omega}) = \sum_{n=-\infty}^{\infty} x[n] e^{-j\omega n}.$$

Las dos ecuaciones anteriores se conocen, respectivamente, como la *transformada inversa de Fourier* y la *transformada de Fourier*. En otras palabras, $X(e^{j\omega})$ es la transformada de Fourier de $x[n]$, mientras que $x[n]$ puede reconstruirse a partir de la transformada inversa de Fourier de $X(e^{j\omega})$.

Note que la transformada de Fourier $X(e^{j\omega})$ puede interpretarse como la correlación cruzada entre la señal $x[n]$ y una sinusoidal compleja con frecuencia ω.

Por otra parte, es claro que la respuesta en frecuencia de un sistema $H(e^{j\omega})$ es igual a la transformada de Fourier de la respuesta al impulso $h[n]$; así como la respuesta al impulso es igual a la transformada inversa de Fourier de la respuesta en frecuencia.

La transformada inversa de Fourier de $X(e^{j\omega})$ se puede calcular solamente si $|X(e^{j\omega})|$ es finito para todo ω. Es fácil ver $|X(e^{j\omega})| \leq \sum_n |x[n]|$; por lo tanto, una condición necesaria para que una señal $x[n]$ pueda descomponerse mediante una integral de Fourier es que $x[n]$ sea absolutamente sumable. Un corolario de lo anterior es que todo sistema LIT estable tiene una respuesta en frecuencia continua y con magnitud finita.

La transformada de Fourier proporciona una representación de una señal $x[n]$ en el *dominio de la frecuencia*, donde los coeficientes $X(e^{j\omega})$ indican la contribución y fase de la frecuencia ω en la señal.

A.4.1 Propiedades de la Transformada de Fourier

El siguiente cuadro resume algunas de las propiedades mas importantes de la transformada de Fourier. En esta tabla, x y y son dos señales absolutamente sumables, cuyas transformadas de Fourier son, respectivamente, X y Y.

Propiedad	**La señal...**	**...y su transformada**
Linealidad	$ax + by$	$aX + bY$
Corrimiento en tiempo	$x[n-d], /d \in \mathbb{Z}$	$e^{-j\omega d}X(e^{j\omega})$
Corrimiento en frecuencia	$e^{j\omega_0 n}x[n]$	$X(e^{j(\omega-\omega_0)})$
Reflexión temporal	$x[-n]$	$X(e^{-j\omega})$
Diferenciación en frecuencia	$nx[n]$	$j\frac{dX(e^{j\omega})}{d\omega}$
Convolución en tiempo	$x[n] * y[n]$	$X(e^{j\omega})Y(e^{j\omega})$
Producto en tiempo	$x[n]y[n]$	$\frac{1}{2\pi}\int_{-\pi}^{\pi} X(e^{j\theta})Y(e^{j(\omega-\theta)})d\theta$

Estas propiedades son de gran utilidad para el análisis y diseño de sistemas de mayor complejidad.

A.5 Teorema de muestreo de Nyquist

Como se mencionó anteriormente, una señal discreta por lo general se obtiene al muestrear una señal o función continua. Bajo ciertas condiciones, es posible recuperar la señal continua de manera exacta a partir de las muestras. Considere una señal continua $x_a(t)$, y su versión discreta $x_d[n] = x_a(nT)$, donde T es el periodo de muestreo y $\Omega_m = 1/T$ es la frecuencia de muestreo.

Sea $X_a(\Omega)$ la transformada de Fourier de la señal continua x_a (con Ω en las mismas unidades que Ω_m). Decimos que x_a es una señal de *banda limitada* si existe una frecuencia $\Omega_X > 0$ tal que $X_a(\Omega) = 0$ para $|\Omega| > \Omega_X$. El teorema de Nyquist establece que la señal de banda limitada x_a puede recuperarse de manera única a partir de las muestras $x_d[n]$ si se cumple que $\Omega_s \geq 2\Omega_X$. En otras palabras, la señal analógica puede recuperarse sin artefactos, si la frecuencia de muestreo es por lo menos el doble que el ancho de banda de la señal.

En la práctica, muchos sistemas tienen una frecuencia de muestreo fija o limitada, por lo que no es posible ajustarla para asegurar que una señal arbitraria pueda ser muestrada sin artefactos. Sin embargo, sabemos que los artefactos se producirán debido a aquellas componentes con frecuencia mayor a la mitad de la frecuencia de muestreo; es decir, $\Omega_s/2$. A esta frecuencia se le conoce como la *frecuencia de Nyquist*.

La discretización de una señal analógica da lugar a la periodización de la representación en el dominio de la frecuencia, donde el periodo es precisamente la frecuencia de muestreo. Cuando la señal a muestrear contiene componentes con frecuencias mayores (en valor absoluto) a la de Nyquist, éstas componentes se traslapan con frecuencias menores, dando lugar a un fenómeno conocido como *aliasing*, en el cual componentes de alta frecuencia en la señal analógica se traducen en componentes de menor frecuencia en la señal discreta. En general, dada una señal discreta no hay manera de saber si sus componentes de frecuencia

corresponden a componentes que son parte de la señal analógica original, o si son el resultado de aliasing.

A.6 Transformada Discreta de Fourier

Una señal discreta $x[n]$ es periódica si existe un entero $N > 0$ tal que $x[n] = x[n + N]$ para todo n. En este caso, decimos que x tiene periodo N. Es importante notar que, en general, no cualquier sinusoidal discreta es periódica. De hecho, una sinusoidal discreta con frecuencia ω es periódica con periodo N si y sólo si $\omega = 2\pi k/N$ para algún entero $k > 0$.

Debido a esto, una señal discreta periódica con periodo N solo puede tener componentes de frecuencia de la forma $\exp\{j\omega_k n\}$ con $\omega_k = 2\pi k/N$. Más aún, solamente se pueden distinguir N frecuencias de este tipo, ya que la componente con frecuencia $N + k$ es idéntica a la componente con frecuencia k.

Por lo tanto, una señal discreta periódica $x[n]$ con periodo N se puede representar mediante la siguiente sumatoria de Fourier:

$$x[n] = \frac{1}{N} \sum_{k=0}^{N-1} X[k] \exp\left(j \frac{2\pi k}{N} n\right),$$

donde los coeficientes $X[k]$ se obtienen como

$$X[k] = \sum_{n=0}^{N-1} x[n] \exp\left(-j \frac{2\pi k}{N} n\right).$$

Note que la secuencia de coeficientes $X[k]$ puede verse en sí como otra señal discreta con periodo N. Las dos ecuaciones anteriores se conocen, respectivamente, como la *transformada discreta inversa de Fourier* (TDIF) y la *transformada discreta de Fourier* (TDF).

En la práctica, las señales que utilizamos no son necesariamente periódicas, pero suelen ser finitas. Por lo tanto, podemos considerar que una señal $x[n]$ discreta y finita de longitud N (definida para $n = 0, \ldots, N - 1$) representa un periodo de una señal subyacente $\tilde{x}[n]$ periódica con periodo N. De esta manera, si calculamos la TDF $X[k]$ de $x[n]$, en realidad estaremos calculando la TDF de la señal periódica subyacente $\tilde{x}[n]$. Para eliminar el efecto de la discontinuidad espuria introducida por la periodización de $x[n]$, uno puede pre-multiplicar x por una función de ventana $w[n]$ que atenúe la señal al inicio y al final. En la literatura se definen diversas funciones de ventana como las de Hann, Hamming y Blackman, las cuales presentan distintas características.

La transformada discreta de Fourier tiene propiedades equivalentes a las de la transformada de Fourier continua, pero con las consideraciones de que las frecuencias se encuentran discretizadas y que las señales se asumen periódicas. Además, existen diversos algoritmos para calcular la TDF de manera muy eficiente, conocidos en general como Transformada Rápida de Fourier (FFT - Fast Fourier Transform). Se recomienda consultar la literatura para conocer mas sobre estos algoritmos [13, 6].

A.7 Ecuaciones en diferencias

Otra manera de describir un sistema LIT es mediante una ecuación en diferencias con coeficientes constantes, de la forma

$$\sum_{k=0}^{N} a_k y[n-k] = \sum_{k=0}^{M} b_k x[n-k],$$

donde a N se le conoce como el *orden* del sistema.

Sin pérdida de generalidad, se puede asumir que $a_0 = 1$ y reescribir el sistema de la manera siguiente:

$$y[n] = \sum_{k=0}^{M} b_k x[n-k] - \sum_{k=1}^{N} a_k y[n-k],$$

donde la primer sumatoria representa una convolución con un kernel causal, y la segunda sumatoria representa la recursividad o retroalimentación de las salidas anteriores del sistema.

Para mantener las propiedades de linealidad e invarianza temporal, es necesario que el sistema esté *inicialmente en reposo*, lo cual significa que si $x[n] = 0$ para todo $n < n_0$, entonces $y[n]$ debe ser también cero para $n < n_0$.

Las ecuaciones en diferencias permiten implementar algunos sistemas LIT de manera mucho mas eficiente que la convolución. Por ejemplo, considere el sistema acumulador, el cual puede definirse como $y[n] = \sum_{k=-\infty}^{n} x[n]$, lo cual corresponde a una convolución con un kernel infinito (IIR); o bien, el sistema puede definirse como $y[n] = x[n] + y[n-1]$, lo cual corresponde a un sistema de primer orden para el que solamente se requiere calcular una suma por muestra.

A.8 Transformada Z

La transformada Z $X(z)$ de una señal $x[n]$ se define como

$$X(z) = \mathcal{Z}\{x[n]\} = \sum_{n=-\infty}^{\infty} x[n] z^{-n},$$

para todo $z \in \mathbb{C}$.

Note que para $z = e^{j\omega}$, entonces la transformada Z es igual a la transformada de Fourier. Entonces se puede considerar a la transformada Z como una generalización de la transformada de Fourier, donde la transformada de Fourier es la transformada Z evaluada en el círculo unitario complejo.

Al igual que la transformada de Fourier, la transformada Z tiene varias propiedades importantes que son de utilidad para el análisis e implementación de sistemas LIT. El siguiente cuadro resume algunas de estas propiedades:

Propiedad	Señal	Transformada Z
Linealidad	$ax[n] + by[n]$	$aX(z) + bY(z)$
Corrimiento en tiempo	$x[n-d],\ d \in \mathbb{Z}$	$z^{-d}X(z)$
Escalamiento en frecuencia	$z_0^n x[n]$	$X(z/z_0)$
Reflexión temporal	$x[-n]$	$X(1/z)$
Diferenciación en frecuencia	$nx[n]$	$-z\frac{dX(z)}{dz}$
Convolución en tiempo	$x[n] * y[n]$	$X(z)Y(z)$

A.8.1 Representación de sistemas LIT en el dominio Z

Considere un sistema LIT dado por medio de una ecuación en diferencias en su forma general:

$$\sum_{k=0}^{N} a_k y[n-k] = \sum_{k=0}^{M} b_k x[n-k].$$

Aplicando la transformada Z a ambos lados de la ecuación anterior, y utilizando las propiedades de linealidad y corrimiento en tiempo se puede llegar a que

$$\frac{Y(z)}{X(z)} = \frac{b_0 + b_1 z^{-1} + \ldots + b_M z^{-M}}{a_0 + a_1 z^{-1} + \ldots + a_N z^{-N}}.$$

Por otra parte, si la respuesta al impulso del sistema es $h[n]$, entonces $y[n] = x[n] * h[n]$, por lo tanto $Y(z) = X(z)H(z)$. Combinando los dos resultados anteriores, obtenemos que

$$H(z) = \frac{Y(z)}{X(z)} = \frac{b_0 + b_1 z^{-1} + \ldots + b_M z^{-M}}{a_0 + a_1 z^{-1} + \ldots + a_N z^{-N}}.$$

A $H(z)$ se le conoce como la *función de transferencia* del sistema. Esta función, evaluada en el círculo unitario, proporciona la respuesta en frecuencia de un sistema IIR.

Por otra parte, es interesante notar que $H(z)$ tiene una forma racional; es decir que es el cociente de dos polinomios cuyos coeficientes son, precisamente, los coeficientes de la ecuación en diferencias a partir de la cual se implementa el sistema. Las raíces de estos polinomios son de particular interés para el análisis y diseño de filtros. A las raíces del numerador se les conoce como los *ceros* del sistema, ya que corresponden a puntos donde la respuesta del sistema es nula. Por otra parte, a las raíces del denominador se les conoce como *polos*, y corresponden a puntos alrededor de los cuales la respuesta crece desmesuradamente, causando inestabilidad. Los ceros pueden aparecer en cualquier punto del plano complejo; sin embargo, para que el sistema sea estable es necesario que todos los polos tengan magnitud menor a uno (es decir, que estén dentro del círculo unitario).

Apéndice B

Clase UGen para C++

PortAudio carece de una arquitectura similar a las UGen (unidades generadoras) de las librerías Beads y Minim. Sin embargo, no es muy difícil implementar una clase en C++ que tenga funcionalidad similar a las UGen de Beads, de manera que podamos utilizar un enfoque similar en todas las prácticas, independientemente del lenguaje con el que se implementen. A continuación se muestran los listados para los archivos de encabezado (`ugen.h`) e implementación (`ugen.cpp`) de la clase `UGen` que usaremos para los ejemplos en C++. Este archivo puede incluirse en un programa simple (compuesto de un solo archivo fuente) mediante la directiva `#include "ugen.cpp"`, o bien dentro de un proyecto compuesto por varios archivos fuente incluyendo el archivo de encabezado `ugen.h` y agregando el archivo fuente `ugen.cpp` al proyecto.

B.1 Archivo de encabezado

```
//////////////////////////////
//
// ugen.h
//
//////////////////////////////

#ifndef _UGEN_HEADER_INCLUDED_
#define _UGEN_HEADER_INCLUDED_

class UGen {
private:
    static double sampleRate;
    static int bufferSize;
    static unsigned long time;

private:
```

```
    unsigned long updateTime;
    unsigned int numInputs;
    unsigned int numOutputs;

    UGen **input;
    float *outBuffer;

public:
    UGen(int outs, int ins = 0);
    virtual ~UGen();

    static void setup(double sr, int bs);
    static double getSampleRate();
    static int getBufferSize();
    static unsigned long getTime();
    static void tick();

    void update();

    int getNumInputs();
    int getNumOutputs();
    unsigned long getUpdateTime();
    void setInput(int in, UGen *ugen);
    void setInput(UGen *ugen);
    UGen *getInput(int in);
    float *getBuffer();
    virtual void processBuffer();
};

#endif
```

B.2 Archivo fuente

```
//////////////////////////////
//
// ugen.cpp
//
//////////////////////////////

#include <cstring>
#include "ugen.h"

double UGen::sampleRate;
int UGen::bufferSize;
unsigned long UGen::time;
```

```
UGen::UGen(int outs, int ins) {
    numInputs = 0;
    numOutputs = 0;
    input = NULL;
    outBuffer = NULL;
    updateTime = 0;

    if (ins > 0) {
        input = new UGen *[ins];
        if (input != NULL) numInputs = ins;
        for (int i = 0; i < ins; i++) input[i] = NULL;
    }

    if (outs > 0) {
        outBuffer = new float[bufferSize * outs];
        if (outBuffer != NULL) numOutputs = outs;
    }
}

UGen::~UGen() {
    if (input) delete[] input;
    if (outBuffer) delete[] outBuffer;
    input = NULL;
    outBuffer = NULL;
}


void UGen::setup(double sr, int bs) {
    sampleRate = (sr > 0) ? sr : 44100;
    bufferSize = (bs > 0) ? bs : 512;
    time = 0;
}

double UGen::getSampleRate() {
    return sampleRate;
}

int UGen::getBufferSize() {
    return bufferSize;
}

unsigned long UGen::getTime() {
    return time;
}
```

```
void UGen::tick() {
    time += (unsigned long)bufferSize;
}

void UGen::update() {
    if (time > updateTime) {
        updateTime = time;
        for (int i = 0; i < numInputs; i++) {
            if (input[i] != NULL) input[i]->update();
        }
        processBuffer();
    }
}

int UGen::getNumInputs() {
    return numInputs;
}

int UGen::getNumOutputs() {
    return numOutputs;
}

unsigned long UGen::getUpdateTime() {
    return updateTime;
}

void UGen::setInput(int in, UGen *ugen) {
    if (in >= 0 && in < numInputs) input[in] = ugen;
}

void UGen::setInput(UGen *ugen) {
    setInput(0, ugen);
}

UGen *UGen::getInput(int in) {
    if (in >= 0 && in < numInputs) {
        return input[in];
    }
    else return NULL;
}

float *UGen::getBuffer() {
    return outBuffer;
}

void UGen::processBuffer() {
```

```
memset(getBuffer(), 0, getBufferSize() * getNumOutputs());
}
```

B.3 Miembros de datos estáticos

La clase `UGen` cuenta con los siguientes miembros de datos estáticos; es decir, que son compartidos por todas las instancias de clase `UGen`:

- `double sampleRate` - Frecuencia de muestreo.
- `unsigned long bufferSize` - Tamaño del buffer de salida (muestras por canal).
- `unsigned long time` - Tiempo en el que inicia el procesamiento del bloque actual (en muestras).

B.4 Funciones estáticas

Así mismo, la clase incluye algunas funciones estáticas para manipular los datos estáticos. Estas funciones deben llamarse directamente a través de la clase (e.g., `UGen::setup()`) y no a través de un objeto de la clase:

- `void setup(double sr, int bs)` - inicializa la frecuencia de muestreo y el tamaño de bloque. Debe llamarse una única vez antes de instanciar cualquier objeto de clase `UGen`.
- `double getSampleRate()` - devuelve la frecuencia de muestreo.
- `int getBufferSize()` - devuelve el tamaño del buffer.
- `unsigned long getTime()` - devuelve el tiempo global (en muestras) correspondiente a la siguiente actualización. Este tiempo determina si una UGen debe actualizar su buffer de salida o no.
- `void tick()` - incrementa el tiempo (miembro `time`) de acuerdo al tamaño del buffer. Debe llamarse una sola vez después de haber calculado por completo todos los bloques actuales.

B.5 Miembros de datos particulares

Los miembros de datos que son particulares a cada objeto de la clase son los siguientes:

- `unsigned long updateTime` - tiempo correspondiente a la última actualización del buffer de salida. Permite determinar si el buffer ha sido actualizado o no.

- `int numInputs` - número de entradas. Cada entrada está conectada con la salida de otro UGen.
- `int numOutputs` - número de canales de salida. Las señales de salida se guardan en un buffer de tipo `float` en el que las muestras están intercaladas por canales.
- `UGen **input` - arreglo de apuntadores a los UGen que proporcionan las entradas. Es posible que algunos de estos apuntadores tengan el valor `NULL`, indicando que ningún UGen está conectado a esa entrada.
- `float *outBuffer` - arreglo que contiene el buffer de salida.

B.6 Métodos

Finalmente, los métodos de la clase `UGen` son los siguientes:

- `UGen(int outs, int ins = 0)` - constructor que recibe como entrada el número de salidas y (opcionalmente) el número de entradas.
- `~UGen()` - destructor.
- `int getNumInputs()` - devuelve el número de entradas (establecido en el constructor).
- `int getNumOutputs()` - devuelve el número de canales de salida (establecido en el constructor).
- `unsigned long getUpdateTime()` - devuelve el tiempo (en muestras) correspondiente a la última actualización del buffer de salida para este UGen.
- `void setInput(int in, UGen *ugen)` - asigna la salida de un UGen a una entrada determinada de este UGen.
- `void setInput(UGen *ugen)` - asigna la salida de un UGen a la primer entrada de este UGen. Muchas UGen contarán solamente con una entrada.
- `UGen *getInput(int in)` - obtiene el UGen asignado a una de las entradas de este UGen.
- `float *getBuffer()` - devuelve el apuntador al buffer de salida.
- `void update()` - actualiza, en caso de ser necesario, el buffer de salida, asegurándose primero de que los buffers de los UGen de entrada estén actualizados.
- `void processBuffer()` - función que realiza el procesamiento para un UGen particular. Esta función debe implementarse para toda clase derivada de `UGen`. La implementación por defecto simplemente genera silencio en todos los canales de salida.

Cabe resaltar que la clase `UGen` presentada aquí, así como las clases derivadas de ella, funcionan de manera independiente a la librería PortAudio, por lo que puede ser utilizada con cualquier otra librería que proporcione acceso a la entrada y salida de audio, o incluso para procesar audio de manera off-line sin requerir ninguna librería adicional.

Las clases heredadas de `UGen` deben de implementar, como mínimo, su propio constructor y su propio método `processBuffer()`. Dentro de `processBuffer()` uno puede llamar a `getInput()` para obtener un apuntador a los UGen asociados con las entradas, y posteriormente llamar al método `getBuffer()` de estos UGens para obtener sus señales de salida, hacer el cálculo necesario, y llenar el buffer para este UGen (el cual se obtiene llamando a su propio método `getBuffer()`). El acceso a los miembros de datos solo puede realizarse a través de las funciones *get* correspondientes, ya que todos los miembros de datos son privados; por ejemplo, para obtener la frecuencia de muestreo debe usarse `getSampleRate()`.

Apéndice C

Clase Buffer para Minim

Una ventaja y desventaja de Minim radica en que en la función callback de las UGen solo procesa una muestra (por canal) a la vez. Esto simplifica considerablemente la implementación de muchos procesos, sobre todo aquellos que actúan muestra a muestra o utilizan recursividad de bajo orden. Sin embargo, este esquema es computacionalmente menos eficiente ya que ocasiona llamadas a funciones por cada muestra, y por otro lado, dificulta la implementación de procesos que actúan por bloques o ventanas corredizas, como lo son la transformada de Fourier, los retardos y la estimación de amplitud.

Es posible recuperar la funcionalidad por bloques implementando una UGen que guarde la señal de entrada, una muestra a la vez, en un arreglo o buffer. Para esto, la UGen debe contar con una variable indicadora que contenga el subíndice del elemento del arreglo donde se almacenará la siguiente muestra de la señal. Dentro de la función callback, se almacenarán la muestra (para cada canal) y se incrementará la variable indicadora. Una vez que ésta alcance el tamaño del arreglo se habrá llenado el buffer, y entonces la variable indicadora se reiniciará a cero para comenzar un nuevo ciclo de llenado; al mismo tiempo, será necesario copiar el contenido del buffer en otro arreglo, ya que el primer buffer será sobreescrito con nuevos datos. Este proceso es conocido como *double buffering* y elimina artefactos ocasionados por acceder al buffer cuando contiene parte de la señal nueva y parte del bloque anterior.

Por simplicidad, el tamaño del buffer se definirá en el constructor y no podrá ser modificado. Se proporcionarán métodos para acceder al segundo buffer (el cual no presenta artefactos en ningún momento) y para recuperar el tamaño del buffer.

En las siguientes secciones se presenta el código de la clase y una descripción breve de sus miembros y métodos.

C.1 Archivo fuente

```
public class Buffer extends UGen {
```

```
protected int size;
protected float[][] buffer;
protected float[][] last_buffer;
protected int index;

UGenInput audio_in;

Buffer(int s) {
  super();
  audio_in = new UGenInput(UGen.InputType.AUDIO);
  size = s;
  index = 0;
}

protected void channelCountChanged() {
  super.channelCountChanged();
  buffer = new float[channelCount()][size];
  last_buffer = new float[channelCount()][size];
}

int size() { return size; }

float[][] getBuffer() {
  return last_buffer;
}

float[] getChannelBuffer(int c) {
  if (channelCount() <= 0) return null;
  return last_buffer[c % channelCount()];
}

void uGenerate(float[] channels) {
  int cc = channelCount();
  float[] in = audio_in.getLastValues();
  for (int c = 0; c < cc; c++) {
    buffer[c][index] = in[c];
  }
  index++;

  if (index == size) {
    index = 0;
    for (int c = 0; c < channelCount(); c++) {
      arrayCopy(buffer[c], last_buffer[c]);
    }
  }
```

```
    arrayCopy(in, channels);
  }
}
```

C.2 Miembros de datos

La clase `Buffer` cuenta con los siguientes miembros de datos:

- `int size` - Tamaño del buffer. Se especifica en el constructor y no puede ser modificado.
- `float[][] buffer` - Arreglo bidimensional que contiene el buffer de llenado (un arreglo unidimensional por cada canal).
- `float[][] last_buffer` - Arreglo bidimensional que contiene el buffer de lectura. Este se actualiza una vez que se llena el buffer de llenado.
- `int index` - Subíndice del elemento en el arreglo de llenado donde se almacenará la siguiente muestra. Se incrementa en cada llamada a la función callback y una vez que alcanza el tamaño del arreglo se reinicia a cero. En una clase heredada de `Buffer`, uno puede verificar la condición `index == 0` dentro de la función callback para sincronizar algún proceso con el llenado del buffer.
- `UGenInput audio_in` - UGen que contiene la señal de entrada al buffer.

C.3 Métodos

Finalmente, los métodos de la clase `Buffer` son los siguientes:

- `Buffer(int s)` - constructor que recibe como entrada el tamaño del buffer.
- `int size()` - devuelve el tamaño del buffer especificado en el constructor.
- `float[][] getBuffer()` - devuelve una referencia al buffer de lectura como un arreglo bidimensional. No se devuelve una copia del buffer, por lo que se debe tener cuidado al modificar los datos del arreglo devuelto.
- `float getChannelBuffer(int c)` - devuelve el buffer de lectura correspondiente al canal `c` como un arreglo unidimensional. Se devuelve una referencia al arreglo, no una copia.

Apéndice D

El Theremin

El *theremin* es un instrumento musical electrónico inventado por el ruso Lev Sergeyevich Termen (después conocido como Léon Theremin) en 1920, y tiene la característica peculiar de ser un instrumento que se ejecuta sin hacer contacto físico con él (es decir, se toca sin tocarlo), como se ilustra en la Figura D.1a. El instrumento cuenta con dos antenas, una vertical que controla el tono (pitch) y otra horizontal para controlar el volumen. Cada una de las manos del ejecutante actúa como una de las placas de un capacitor variable, siendo cada antena la otra placa del capacitor correspondiente. La antena vertical controla la frecuencia de un oscilador, mientras que la antena horizontal controla la ganancia de un amplificador. De esta manera, el movimiento de una mano controla el tono mientras que la otra controla el volumen [57].

Al igual que muchos otros inventos, el theremin fue un producto derivado de investigaciones militares, en este caso sobre sensores de proximidad y osciladores de alta frecuencia para comunicación por radio. Decepcionado de la milicia, Léon Theremin consiguió trabajo en el Instituto Técnico-Físico de Petrogrado donde comenzó a adaptar sus dispositivos para la generación de tonos audibles. Su experiencia con el violocello eventualmente lo llevó a diseñar un instrumento con sonido y características similares, al que llamó *etherphone*, haciendo alusión a la manera de ejecutarlo sin tocarlo. A finales de 1927 se mudó a los Estados Unidos, donde patentó su invento en 1928. En los años siguientes, el theremin se convirtió en una novedad electrónica, mas que en un instrumento considerado por músicos profesionales. En muchos casos, el theremin se vendía en forma de un kit electrónico para armar como pasatiempo [68]. En la década de 1950, un joven entusiasta llamado Robert Moog, se dedicaba a fabricar y vender kits de theremin mientras estudiaba un posgrado, formando una pequeña empresa. A los pocos años, Moog comenzó a diseñar módulos para síntesis de sonido que incorporaban control por voltaje (toda una novedad en ese entonces), convirtiéndose eventualmente en uno de los pioneros de la síntesis de sonido moderna.

El sonido de un theremin se produce a partir de dos osciladores de radiofrecuencia mediante el principio de generación de frecuencias heterodinas: cuando

(a) (b)

Figura D.1: (a) Artista (Lydia Kavina) ejecutando una pieza en un theremin. (b) Moog Theremini, un ejemplo de theremin moderno que utiliza un motor de audio digital con múltiples formas de onda y efectos adicionales.

dos señales sinusoidales con frecuencias f_1 y f_2 son multiplicadas entre sí, el resultado es la suma de dos nuevas sinusoidales cuyas frecuencias son, respectivamente, $f_1 + f_2$ y $f_1 - f_2$. Esto se aprecia claramente a través de la siguiente identidad trigonométrica:

$$2\cos\theta\cos\phi = \cos(\theta - \phi) + \cos(\theta + \phi).$$

La generación heterodina también ocurre tambien al aplicar modulación en amplitud o modulación en anillo para generar componentes de frecuencia armónicas o inarmónicas a partir del producto de dos señales. Electrónicamente, el producto de dos señales se puede implementar mediante un amplificador de cuatro cuadrantes (es decir, que pueda responder a ganancias negativas). En el dominio digital simplemente se utiliza un amplificador cuya ganancia es controlada por una señal moduladora.

En el caso del theremin, las frecuencias f_1 y f_2 son radiofrecuencias en el orden de cientos de kilohertz, muy por encima de las frecuencias audibles, siendo f_1 una frecuencia fija y f_2 una frecuencia variable (determinada por la proximidad de la mano a la antena vertical), pero cercana a f_1. Al multiplicar las señales de ambos osciladores, se generan las frecuencias $f_1 + f_2$ (la cual es inaudible), y $f_1 - f_2$ la cual es audible. Esta señal pasa por un waveshaper para distorsionar la forma de onda y generar armónicos adicionales, luego por un filtro pasabajas para controlar la brillantez del sonido, y finalmente por un amplificador cuya ganancia es controlada por la antena de volumen.

Aunque en la actualidad se siguen fabricando theremins analógicos con osciladores heterodinos, existen también algunos diseños donde se reemplazan los osciladores heterodinos por un solo oscilador en el rango audible. En particular, en una emulación digital del theremin es difícil usar los osciladores heterodinos ya que estos requieren oscilar a frecuencias mucho mayores a las frecuencias de

muestreo que se utilizan para audio. Por otra parte, el uso de un oscilador digital permite generar otras formas de onda y agregar efectos adicionales, como ocurre con el Moog Theremini (Figura D.1b).

El rango de frecuencias audibles que produce un theremin es de aproximadamente cuatro octavas (similar al del cello). Algunos diseños de theremin incorporan elementos adicionales para linearizar el tono con respecto a la distancia entre la mano del ejecutante y la antena vertical, lo cual hace mas sencilla la ejecución del instrumento. Diseños modernos pueden incluso incorporar una etapa de cuantización para forzar a que las frecuencias producidas correspondan a tonos en alguna escala musical.

Apéndice E

Transformada Rápida de Fourier

Tanto la librería Beads como Minim proporcionan clases para calcular la Transformada Rápida de Fourier (FFT), así como algunas herramientas de análisis en frecuencia, como el cálculo del espectro de energía y la detección de frecuencia. Sin embargo, estas clases no están programadas como Unidades Generadoras (UGens) y utilizan mecanismos distintos en cada librería. Por otra parte, la implementación que hemos desarrollado en C++ cuenta solamente con el mecanismo basado en UGens para generar, analizar y procesar señales de audio.

Por estos motivos, no está de más implementar nuestra propia versión de la FFT para utilizarla como parte de una UGen que pueda insertarse fácilmente dentro de una cadena de procesamiento, sin requerir de otro tipo de mecanismos. La implementación se presenta para Processing o Java; sin embargo, su traducción a C++ no debería suponer más que algunos cambios mínimos y la inclusión de un destructor para liberar la memoria reservada para las tablas precalculadas.

La implementación que aquí se presenta está basada en el algoritmo de cálculo de la FFT en línea mediante decimado en frecuencia que presentan Oppenheim y Schafer [13]. Si bien está lejos de ser rápida, comparada con el estado del arte, será suficiente para el desarrollo de las prácticas y aplicaciones que se presentan a lo largo de este manual. Además, nuestra implementación estará limitada a señales cuya longitud sea una potencia de 2, y por lo general se aplicará a los bloques de procesamiento del sub-sistema de audio, por lo que habrá que asegurarse de que el tamaño de bloque sea también una potencia de 2.

En las siguientes secciones se presenta el código de la clase, así como una descripción breve de sus miembros y métodos. Note que la clase que aquí se presenta permite calcular la FFT y la IFFT de una señal compleja en general, y de manera independiente al mecanismo de las UGens. En la Práctica 15 se presenta la UGen que encapsulará este proceso.

E.1 Archivo fuente

```
public class FourierAnalysis {
  int N;  // length of input/output signals (must be power of 2)
  int M;  // log2(N)
  float[] st;  // pre-computed sin table
  float[] ct;  // pre-computed cos table

  public FourierAnalysis(int m) {
    float a;
    M = m;
    N = 1 << M;
    st = new float[N];
    ct = new float[N];
    for (int k = 0; k < N; k++) {
      a = -2 * PI * k / N;
      st[k] = sin(a);
      ct[k] = cos(a);
    }
  }

  public void fft(float[] xr, float[] xi) {
    int i, j, k, m, ng, nb, d, g, b;
    float tr, ti;

    if ((xr.length != N) || (xi.length != N)) return;

    // bit reversal
    for (i = 1; i < N - 1; i++) {
      k = i;
      j = 0;
      for (m = 0; m < M; m++) {
        j = (j << 1) + (k & 1);
        k >>= 1;
      }
      if (i < j) {
        tr = xr[i]; ti = xi[i];
        xr[i] = xr[j]; xi[i] = xi[j];
        xr[j] = tr; xi[j] = ti;
      }
    }

    // fft computation
    ng = N >> 1;  // number of groups at each stage
    nb = 1;       // number of butterflies in each group
    d = 1;        // separation between butterflies
```

```
  for (m = 0; m < M; m++) {
    for (g = 0; g < ng; g++) {
      i = g * (nb << 1);
      j = i + d;
      k = 0;

      for (b = 0; b < nb; b++) {
        // compute butterfly between elements i and j
        // y[i] = x[i] + W[k] * x[j]
        // y[j] = x[i] - W[k] * x[j]

        tr = xr[j] * ct[k] - xi[j] * st[k];
        ti = xr[j] * st[k] + xi[j] * ct[k];
        xr[j] = xr[i] - tr; xi[j] = xi[i] - ti;
        xr[i] += tr; xi[i] += ti;

        i++;
        j++;
        k += ng;
      }
    }
    ng >>= 1;
    nb <<= 1;
    d <<= 1;
  }
}

public void ifft(float[] xr, float[] xi) {
  int k = N - 1, n = 0, halfN = N >> 1;
  float tr, ti;
  fft(xr, xi);
  while (n < halfN) {
    tr = xr[n]; xr[n] = xr[k]; xr[k] = tr;
    ti = xi[n]; xi[n] = xi[k]; xi[k] = ti;
    n++;
    k--;
  }
  for (n = 0; n < N; n++) {
    xr[n] /= N; xi[n] /= N;
  }

  tr = xr[N - 1]; ti = xi[N - 1];
  for (n = N - 1; n > 0; n--) {
    xr[n] = xr[n - 1];
    xi[n] = xi[n - 1];
```

```
    }
    xr[0] = tr; xi[0] = ti;
  }
}
```

E.2 Miembros de datos

La clase `FourierAnalysis` cuenta con los siguientes miembros de datos:

- `int M` - Este es el logaritmo base 2 de la longitud de la señal a procesar. En otras palabras, las señales son de longitud 2^M. El valor de M se pasa al constructor de la clase y determina también la resolución del análisis en frecuencia.
- `int N` - Tamaño de las señales a procesar, donde $N = 2^M$. Se calcula en el constructor y no puede ser modificado. Determina la resolución en frecuencia del análisis, donde las frecuencias analizadas son $\omega_k = 2\pi k/N$ para $k = 0, \ldots, N-1$.
- `float[] st` - Tabla pre-calculada con los valores del seno para los ángulos correspondientes a las frecuencias de análisis. Se calcula en el constructor y no debe ser modificado.
- `float[] ct` - Tabla pre-calculada con los valores del coseno para los ángulos correspondientes a las frecuencias de análisis. Se calcula en el constructor y no debe ser modificado.

E.3 Métodos

Finalmente, los métodos de la clase `FourierAnalysis` son los siguientes:

- `FourierAnalysis(int m)` - constructor que recibe como entrada el logaritmo base-2 del tamaño de las señales a procesar.
- `void fft(float[] xr, float[] xi)` - calcula la FFT en línea para la señal compleja de entrada, donde la parte real se almacena en el arreglo `xr` y la parte imaginaria en el arreglo `xi`. El cómputo se realiza en línea, lo cual significa que la señal contenida en los arreglos `xr` y `xi` es reemplazada por su transformada discreta de Fourier. Para una explicación del algoritmo utilizado, consultar [13].
- `void ifft(float[] xr, float[] xi)` - calcula la IFFT (transformada inversa rápida de Fourier) en línea, reemplazando el contenido de los arreglos.

Apéndice F

MIDI

El estándar MIDI (Musical Instrument Digital Interface) describe un protocolo de comunicación, así como las especificaciones electrónicas, para interconectar y/o sincronizar una gran variedad de instrumentos electrónicos, equipos de iluminación, computadoras, secuenciadores y controladores físicos. MIDI fue desarrollado a inicios de la década de 1980 por un grupo formado por los principales fabricantes de instrumentos musicales electrónicos de aquella época, con la participación de compañías como Roland, Yamaha, Korg, Sequential Circuits y Oberheim Electronics. Los primeros sintetizadores equipados con MIDI surgieron en 1982, y en 1983 se publicó la primer versión de las especificaciones MIDI, permitiendo así que cualquier fabricante pudiera incorporar esta tecnología a sus equipos. Desde entonces, se han realizado numerosas mejoras al protocolo, pero respetando la compatibilidad hacia atrás. Esto ha permitido que los nuevos instrumentos puedan mantener la comunicación con instrumentos de hace treinta años o más. En el año 2013, los principales inventores de MIDI, Ikutaro Kakehashi de Roland y Dave Smith de Sequential Circuits, recibieron el Grammy Technical Award por su contribución. La especificación completa y otros documentos complementarios pueden consultarse en el sitio Web de la Asociación MIDI [69].

Desde un punto de vista electrónico, MIDI es una interfaz de comunicación serial asíncrona con una tasa de 31.25 baudios (bits por segundo), a través de la cual se envían palabras de 8 bits. Cada palabra va precedida de un bit de inicio y seguida de un bit de parada. Los conectores estándar utilizados para MIDI son del tipo DIN de 5 pines, con conectores hembra en los dispositivos y macho para los cables de interconexión. Por lo general se utilizan conectores distintos para datos de entrada y salida, los cuales deben estar claramente etiquetados en el dispositivo como MIDI IN y MIDI OUT, respectivamente. No todos los dispositivos presentan ambas conexiones, ya que algunos dispositivos solamente pueden enviar información, mientras que algunos otros solo pueden recibirla. Algunos dispositivos presentan una tercer conexión llamada MIDI THRU, la cual envía la información recibida a través del puerto MIDI IN en cascada hacia otros dispositivos. En la actualidad, es común encontrar disposi-

tivos que reemplazan los conectores típicos tipo DIN con un solo conector USB (llamado USB MIDI); así mismo, existen interfaces que permiten interconectar dispositivos modernos con MIDI USB.

Cada conexión MIDI física permite enviar información en hasta 16 canales lógicos de transmisión. Los datos enviados contienen el número de canal correspondiente, de manera que solamente actuarán aquellos dispositivos receptores programados para responder a ese número de canal.

F.1 Mensajes MIDI

La información MIDI consiste en *mensajes* que se envían de un dispositivo a otro. Un mensaje es una secuencia de bytes (palabras de 8 bits) que puede ser o no reconocida por el receptor. Si un receptor no reconoce un mensaje, simplemente debe ignorarlo. De esta manera se puede agregar funcionalidad al protocolo y mantener la compatibilidad hacia atrás.

Los mensajes comunes tienen una longitud de dos o tres bytes. Estos son mensajes que la mayoría de los dispositivos pueden enviar o reconocer, tales como reproducir una nota o sonido específico, o cambiar el valor de un parámetro específico. En todos los mensajes, el primer byte del mensaje (llamado byte de estado) debe tener activo el bit mas significativo, mientras que para los bytes restantes del mensaje el octavo bit siempre vale cero. Por lo tanto, el primer byte de un mensaje toma valores entre 128 y 255, mientras que los bytes restantes toman valores entre 0 y 127.

La categoría mas importante de mensajes, conocidos como *mensajes de canal de voz*, son aquellos que pueden enviarse a través de uno de los 16 canales lógicos en una conexión, y usualmente indican al dispositivo receptor que debe reproducir algún sonido o alterar los que están siendo reproducidos. Todos los mensajes de canal de voz contienen el número de canal (entre 0 y 15) en los cuatro bits menos significativos del byte de estado (el primer byte del mensaje), mientras que los cuatro bits mas significativos codifican el tipo de mensaje. Ya que el bit mas significativo del byte de estado siempre vale uno, entonces puede haber solamente ocho tipos de mensajes de voces, de los cuales siete están definidos en el estándar MIDI. Estos se muestran en el Cuadro F.1 y se describen a continuación, utilizando el subíndice 2 para indicar números en base binaria.

Note Off - 1000_2

Este mensaje indica a un generador de sonidos que una nota debe ser liberada. Esto no implica necesariamente que el sonido asociado a la nota debe dejar de escucharse, sino que debe entrar en su fase de relajación, por ejemplo cuando se levanta el dedo de la tecla de un piano, o cuando se deja de frotar el arco en un instrumento de cuerdas. Este tipo de mensajes tienen una longitud de tres bytes, por lo tanto se esperan dos bytes más después del byte de estado. El primero de ellos contiene el número de nota. En el estándar MIDI existen 128 notas, numeradas de 0 a 127, donde 60 corresponde al Do central (C4) de

Bits mas significativos del byte de estado	Tipo de mensaje
1000_2	Note Off
1001_2	Note On
1010_2	Presión por tecla
1011_2	Controlador continuo
1100_2	Cambio de programa
1101_2	Presión por canal
1110_2	Pitch bend

Cuadro F.1: Tipos de mensajes MIDI de canal de voz.

un piano. El segundo byte corresponde a la *velocidad de liberación*, que indica qué tan rápido se dejó de pulsar la tecla correspondiente. La mayoría de los instrumentos MIDI no utilizan esta información y simplemente envían el valor 64 como velocidad de liberación.

Note On - 1001_2

Indica que se debe tocar una determinada nota o sonido. Propiamente hablando, el instrumento que responde a este mensaje da inicio a la fase de ataque para la nota correspondiente. Este mensaje también cuenta con tres bytes; es decir, dos bytes adicionales al byte de estado. El primero de ellos contiene el número de nota y el segundo la *velocidad* con la que se dispara la nota. La mayoría de los instrumentos MIDI son sensibles y responden a la velocidad, permitiendo una mayor expresividad en la ejecución de un instrumento. Por otra parte, si la velocidad es cero, entonces el mensaje es equivalente a un mensaje Note Off, ocasionando que la nota sea liberada. Es importante que todo mensaje Note On sea eventualmente seguido de un mensaje Note Off (o bien, Note On con velocidad cero) para la misma nota y mismo canal.

Ejemplo: el mensaje $[10011000_2, 60, 100]$ indicará al dispositivo receptor que reproduzca la nota Do central con velocidad 100 a través del canal 9 (los canales se numeran a partir de 1). Eventualmente, debe enviarse un mensaje Note Off correspondiente para liberar la nota, ya sea $[10001000_2, 60, 64]$, o bien $[10011000_2, 60, 0]$.

Presión por tecla - 1010_2

Algunos controladores cuentan con sensores de presión por cada tecla o pad, los cuales pueden detectar la fuerza con la que se mantiene presionada la tecla. Mientras la tecla se encuentra oprimida, el usuario puede variar la presión para modificar el sonido en tiempo real y añadir mayor expresión a la ejecución. Existen dos tipos de sensibilidad a la presión (también llamada *aftertouch*): uno donde la presión se aplica de manera independiente a cada nota (e.g., cada tecla tiene un sensor independiente de presión), y otro donde la presión total afecta a todas las notas activas en un canal determinado. La presión por tecla

se codifica en mensajes de tres bytes, donde los dos bytes que siguen al byte de estado contienen, respectivamente, el número de nota afectada y la cantidad de presión aplicada a dicha nota.

Controlador continuo - 1011_2

Un controlador continuo (CC) puede hacer referencia a cualquier parámetro variable que tenga alguna función en el dispositivo receptor (por ejemplo, los parámetros de síntesis en un sintetizador). Los mensajes de CC permiten manipular estos parámetros en tiempo real desde un dispositivo remoto. Por cada canal, un dispositivo MIDI puede tener hasta 128 controladores asociados a parámetros, los cuales se numeran del 0 al 127. Algunos números de controlador se encuentran ya asignados en el estándar MIDI (por ejemplo, el CC #1 corresponde a la rueda de modulación de un teclado MIDI convencional, mientras que el CC #7 corresponde al volumen). Los mensajes de CC contienen dos bytes después del byte de estado: el primero de ellos indica el número de controlador que se desea cambiar y el segundo contiene el valor que se desea asignar al controlador o parámetro.

Cambio de programa - 1100_2

En un sintetizador, un *programa* es el conjunto de valores de los parámetros de síntesis que producen un timbre específico. Muchos sintetizadores cuentan con memorias para guardar diversos programas, de manera que puedan ser recuperados en cualquier momento. Los programas en un dispositivo MIDI están numerados de 0 a 127, y es posible enviar un mensaje de cambio de programa para seleccionar cualquiera de ellos. Este tipo de mensaje solo tiene dos bytes: el byte de estado y el número de programa que desea seleccionar para el canal indicado.

Muchos sintetizadores permiten almacenar mas de 128 programas, para lo cual se agrupan los programas en *bancos*. En este caso, uno puede seleccionar un banco de programas mediante el CC #0 (ver arriba), y posteriormente seleccionar un programa dentro de ese banco. De esta manera, un dispositivo MIDI puede direccionar hasta $2^{14} = 16384$ programas distintos.

Presión de canal - 1101_2

Los mensajes de presión de canal permiten especificar la cantidad de presión total aplicada en un controlador, por cada canal lógico. La presión aplicada afectará a todas las notas activas en el canal correspondiente. Este mensaje consta de dos bytes: el byte de estado y el valor de presión.

Pitch bend - 1110_2

Los mensajes de *pitch bend* suelen utilizarse para producir variaciones continuas en la altura o tono de las notas activas en un canal determinado. La mayoría de los teclados MIDI convencionales utilizan una rueda o un joystick para aplicar

pitch bend a los sonidos internos o enviar los mensajes correspondientes a dispositivos externos. Estos mensajes constan de tres bytes; el primero es el byte de estado, mientras que los dos restantes se combinan para formar un valor de 14 bits. El primer byte de datos contiene los siete bits menos significativos, mientras que el segundo byte contiene los siete bits mas significativos. El número resultante de 14 bits toma valores entre 0 y 16383, donde el valor 8192 indica que el control de tono se encuentra centrado, por lo que la altura original de las notas activas no se ve alterada. Valores mayores a 8192 incrementan la altura, mientras que valores menores la decrementan. Por lo general, el efecto de pitch bend es exponencial; es decir, se mapea linealmente a semitonos o a octavas. El rango de transposición depende del dispositivo receptor.

F.2 La librería MidiBus en Processing

MidiBus es una librería para Processing que permite el envío y recepción de mensajes MIDI de una manera sencilla. Cabe mencionar que Java (y por lo tanto, Processing) cuenta ya con una librería estándar para el manejo de información MIDI; sin embargo, MidiBus encapsula la funcionalidad básica para simplificar la implementación.

En una aplicación simple, se puede llamar primero al método estático `list()` de la clase `MidiBus` para obtener una lista de los puertos MIDI disponibles. Usando esa información, se puede entonces crear un objeto de clase `MidiBus` asociado a puertos de entrada y salida específicos, y utilizar los métodos `sendNoteOn()`, `sendNoteOff()` y `sendControllerChange()` para enviar mensajes Note On, Note Off y CC al dispositivo de salida seleccionado. Así mismo, se pueden implementar las funciones callback `noteOn()`, `noteOff()` y `controllerChange()` para hacer que el programa responda a los mensajes respectivos que se reciban desde el exterior.

Para enviar otros tipos de mensajes (distintos a Note On, Note Off y CC) se puede utilizar el método `sendMessage()` del objeto de clase `MidiBus`, mientras que para recibir estos mensajes se puede implementar una clase derivada de `StandardMidiListener` o `RawMidiListener`, las cuales cuentan con funciones callback para recibir mensajes arbitrarios, y asociar un objeto de la clase heredada al objeto `MidiBus` deseado mediante el método `addMidiListener()`.

Referencias

[1] Casey Reas and Ben Fry. *Getting Started with Processing: A Hands-On Introduction to Making Interactive Graphics*. Maker Media, Inc., 2015.

[2] Daniel Shiffman. *Learning Processing: a beginner's guide to programming images, animation, and interaction*. Morgan Kaufmann, 2009.

[3] Oliver Bown. Experiments in modular design for the creative composition of live algorithms. *Computer Music Journal*, 35(3):73–85, 2011.

[4] Evan X Merz. *Sonifying Processing: The Beads Tutorial*. 2011.

[5] Miller S Puckette. Pure data: another integrated computer music environment. *Proceedings of the second intercollege computer music concerts*, pages 37–41, 1996.

[6] John G Proakis and Dimitris K Manolakis. *Digital Signal Processing*. Pearson, 2006.

[7] Gareth Loy. *Musimathics: the mathematical foundations of music*, volume 1. MIT press, 2006.

[8] D Poeppel, T Overath, A N Popper, and R R (Eds.) Fay. *The Human Auditory Cortex*. Springer, 2012.

[9] Jens Blauert. *Spatial hearing: the psychophysics of human sound localization*. MIT press, 1997.

[10] Miller Puckette. *The theory and technique of electronic music*. World Scientific Publishing Co Inc, 2007.

[11] Ian McLoughlin. *Applied speech and audio processing: with Matlab examples*. Cambridge University Press, 2009.

[12] Julius O Smith. *Introduction to digital filters: with audio applications*. Julius Smith, 2008.

[13] Alan V Oppenheim and W Schafer, Ronald. *Discrete-time signal processing*. Prentice Hall, 2009.

[14] Rusty Allred. *Digital filters for everyone*. Creative Arts & Sciences House, 2013.

[15] Steven W Smith. The scientist and engineer's guide to digital signal processing. 1997.

[16] Robert A Moog. Voltage controlled electronic music modules. *Journal of the Audio Engineering Society*, 13(3):200–206, 1965.

[17] Manfred R Schroeder. *Computer speech: recognition, compression, synthesis*, volume 35. Springer Science & Business Media, 2013.

[18] Julius Smith, Stefania Serafin, Jonathan Abel, and David Berners. Doppler simulation and the leslie. In *Proceeding of the Workshop on Digital Audio Effects, DAFx-02, Hamburg, Germany*, pages 188–191, 2002.

[19] Julius O Smith. *Physical audio signal processing: For virtual musical instruments and audio effects*. W3K Publishing, 2010.

[20] Julius O Smith. *Mathematics of the discrete Fourier transform (DFT): with audio applications*. Julius Smith, 2007.

[21] Matteo Frigo and Steven G Johnson. The design and implementation of fftw3. *Proceedings of the IEEE*, 93(2):216–231, 2005.

[22] David Gerhard. *Pitch extraction and fundamental frequency: History and current techniques*. Technical Report TR-CS 2003-06, Department of Computer Science, University of Regina, 2003.

[23] Lawrence R Rabiner and Ronald W Schafer. *Digital processing of speech signals*. Pearson Education, 1993.

[24] Edwin H Armstrong. A method of reducing disturbances in radio signaling by a system of frequency modulation. *proceedings of the Institute of Radio Engineers*, 24(5):689–740, 1936.

[25] John M Chowning. The synthesis of complex audio spectra by means of frequency modulation. *Journal of the audio engineering society*, 21(7):526–534, 1973.

[26] Tim Stilson and Julius Smith. Analyzing the moog vcf with considerations for digital implementation. In *Proceedings of the 1996 International Computer Music Conference, Hong Kong, Computer Music Association*, 1996.

[27] Antti Huovilainen. Non-linear digital implementation of the moog ladder filter. In *Proceedings of the International Conference on Digital Audio Effects (DAFx-04)*, pages 61–64, 2004.

[28] Vesa Välimäki and Antti Huovilainen. Oscillator and filter algorithms for virtual analog synthesis. *Computer Music Journal*, 30(2):19–31, 2006.

[29] Udo Zölzer. *DAFX: digital audio effects*. John Wiley & Sons, 2011.

[30] Kevin Karplus and Alex Strong. Digital synthesis of plucked-string and drum timbres. *Computer Music Journal*, 7(2):43–55, 1983.

[31] Fritz Winckel. *Music, sound and sensation: A modern exposition*. Courier Corporation, 1967.

[32] Abbas K Abbas and Rasha Bassam. *Phonocardiography signal processing*. Number 31. Morgan & Claypool Publishers, 2009.

[33] Ahsan Khandoker, Emad Ibrahim, Sayaka Oshio, and Yoshitaka Kimura. Validation of beat by beat fetal heart signals acquired from four-channel fetal phonocardiogram with fetal electrocardiogram in healthy late pregnancy. *Scientific reports*, 8(1):13635, 2018.

[34] Jakub Kolarik, Matej Golembiovsky, Tomas Docekal, Radana Kahankova, Radek Martinek, and Michal Prauzek. A low-cost device for fetal heart rate measurement. *IFAC-PapersOnLine*, 51(6):426–431, 2018.

[35] Madhava Vishwanath Shervegar and Ganesh V Bhat. Heart sound classification using gaussian mixture model. *Porto Biomedical Journal*, 3(1):e4, 2018.

[36] Ary L Goldberger, Luis AN Amaral, Leon Glass, Jeffrey M Hausdorff, Plamen Ch Ivanov, Roger G Mark, Joseph E Mietus, George B Moody, Chung-Kang Peng, and H Eugene Stanley. Physiobank, physiotoolkit, and physionet: components of a new research resource for complex physiologic signals. *Circulation*, 101(23):e215–e220, 2000.

[37] Mario Cesarelli, Mariano Ruffo, Maria Romano, and Paolo Bifulco. Simulation of foetal phonocardiographic recordings for testing of fhr extraction algorithms. *Computer methods and programs in biomedicine*, 107(3):513–523, 2012.

[38] Chengyu Liu, David Springer, Qiao Li, Benjamin Moody, Ricardo Abad Juan, Francisco J Chorro, Francisco Castells, José Millet Roig, Ikaro Silva, Alistair EW Johnson, et al. An open access database for the evaluation of heart sound algorithms. *Physiological Measurement*, 37(12):2181, 2016.

[39] Udantha Ranjith Abeyratne, CKK Patabandi, and Kathiravelu Puvanendran. Pitch-jitter analysis of snoring sounds for the diagnosis of sleep apnea. In *Engineering in Medicine and Biology Society, 2001. Proceedings of the 23rd Annual International Conference of the IEEE*, volume 2, pages 2072–2075. IEEE, 2001.

[40] Azadeh Yadollahi and Zahra Moussavi. Formant analysis of breath and snore sounds. In *Engineering in Medicine and Biology Society, 2009. EMBC 2009. Annual International Conference of the IEEE*, pages 2563–2566. IEEE, 2009.

[41] Barbara Calabrese, Franco Pucci, Miriam Sturniolo, Pietro Hiram Guzzi, Pierangelo Veltri, Antonio Gambardella, and Mario Cannataro. A system for the analysis of snore signals. *Procedia Computer Science*, 4:1101–1108, 2011.

[42] J R Perez-Padilla, Slawinski E, L M Difrancesco, R R Feige, J E Remmers, and W A Whitelaw. Characteristics of the snoring noise in patients with and without occlusive sleep apnea. *Am Rev Respir Dis*, 147:635–644, 1993.

[43] JA Fiz, J Abad, R Jane, M Riera, MA Mananas, P Caminal, D Rodenstein, and J Morera. Acoustic analysis of snoring sound in patients with simple snoring and obstructive sleep apnoea. *European Respiratory Journal*, 9(11):2365–2370, 1996.

[44] Andrew K Ng, TS Koh, Eugene Baey, and K Puvanendran. Speech-like analysis of snore signals for the detection of obstructive sleep apnea. In *Biomedical and Pharmaceutical Engineering, 2006. ICBPE 2006. International Conference on*, pages 99–103. IEEE, 2006.

[45] Jordi Sola-Soler, Raimon Jane, Jose Antonio Fiz, and Jose Morera. Formant frequencies of normal breath sounds of snorers may indicate the risk of obstructive sleep apnea syndrome. In *Engineering in Medicine and Biology Society, 2008. EMBS 2008. 30th Annual International Conference of the IEEE*, pages 3500–3503. IEEE, 2008.

[46] Hui Jin, Li-Ang Lee, Lijuan Song, Yanmei Li, Jianxin Peng, Nanshan Zhong, Hsueh-Yu Li, and Xiaowen Zhang. Acoustic analysis of snoring in the diagnosis of obstructive sleep apnea syndrome: a call for more rigorous studies. *Journal of Clinical Sleep Medicine*, 11(07):765–771, 2015.

[47] Thomas Hermann, Andy Hunt, and John G Neuhoff. *The sonification handbook*. Logos Verlag Berlin, 2011.

[48] Brian FG Katz and Lorenzo Picinali. Spatial audio applied to research with the blind. In *Advances in sound localization*. InTech, 2011.

[49] Jeffrey R Blum, Mathieu Bouchard, and Jeremy R Cooperstock. Spatialized audio environmental awareness for blind users with a smartphone. *Mobile Networks and Applications*, 18(3):295–309, 2013.

[50] Sergio Damiani, Enrica Deregibus, and Luisa Andreone. Driver-vehicle interfaces and interaction: where are they going? *European transport research review*, 1(2):87–96, 2009.

[51] Sergej Truschin, Michael Schermann, Suparna Goswami, and Helmut Krcmar. Designing interfaces for multiple-goal environments: Experimental insights from in-vehicle speech interfaces. *ACM Transactions on Computer-Human Interaction (TOCHI)*, 21(1):7, 2014.

[52] Susumu Harada, James A Landay, Jonathan Malkin, Xiao Li, and Jeff A Bilmes. The vocal joystick: evaluation of voice-based cursor control techniques for assistive technology. *Disability and Rehabilitation: Assistive Technology*, 3(1-2):22–34, 2008.

[53] Siu-Lan Tan, Annabel J Cohen, Scott D Lipscomb, and Roger A Kendall. *The psychology of music in multimedia*. Oxford University Press, 2013.

[54] Sandra L Calvert. *Children's media: The role of music and audio features*, pages 267–288. OUP Oxford, 2013.

[55] Eduardo Reck Miranda. *Digital Sound Synthesis for Multimedia Audio*, pages 948–958. Wiley-Interscience, 2009.

[56] Andy Farnell. *Designing sound*. MIT Press, 2010.

[57] Mark Vail. *The Synthesizer*. Oxford University Press, 2014.

[58] Iannis Xenakis. *Formalized music: thought and mathematics in composition*. Pendragon Press, 1992.

[59] Alfonso Alba. Generating music by fractals and grammars. Master's thesis, Instituto de Investigación en Comunicación Óptica, Universidad Autónoma de San Luis Potosí, Agosto 2001.

[60] Eduardo Reck Miranda and John Al Biles. *Evolutionary computer music*. Springer, 2007.

[61] Godfried T Toussaint. *The Geometry of Musical Rhythm: What Makes a" Good" Rhythm Good?* Chapman and Hall/CRC, 2016.

[62] Andy Farnell. An introduction to procedural audio and its application in computer games. In *Audio mostly conference*, volume 23, 2007.

[63] Nikunj Raghuvanshi, Christian Lauterbach, Anish Chandak, Dinesh Manocha, and Ming C Lin. Real-time sound synthesis and propagation for games. *Communications of the ACM*, 50, 2007.

[64] Karen Collins. An introduction to procedural music in video games. *Contemporary Music Review*, 28(1):5–15, 2009.

[65] George Borzyskowski. " the hacker demo scene and it's cultural artefacts. In *Cybermind Conference 1996*, pages 1–23, 1996.

[66] Markku Reunanen. Computer demos–what makes them tick, April 2010.

[67] Nikunj Raghuvanshi and Ming C Lin. Interactive sound synthesis for large scale environments. In *Proceedings of the 2006 symposium on Interactive 3D graphics and games*, pages 101–108. ACM, 2006.

[68] Albert Glinsky. *Theremin: ether music and espionage*. University of Illinois Press, 2000.

[69] The MIDI Association. The midi association website. http://www.midi.org, 2018.