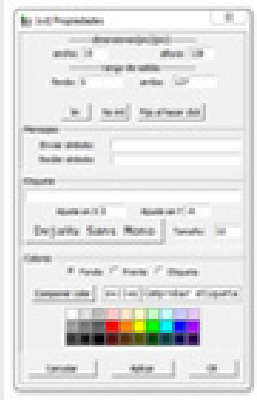
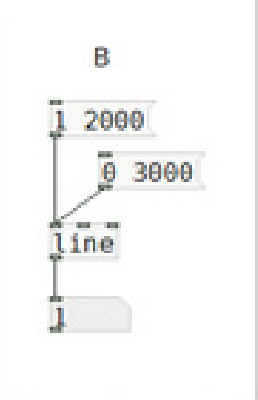
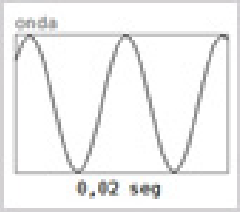
Universidad Virtual de Quilmes

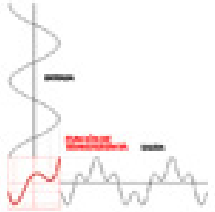
Captura
y procesamiento de sonido

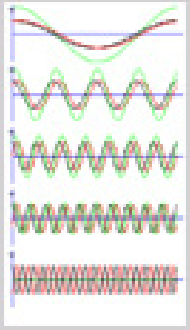
Pablo Cetta

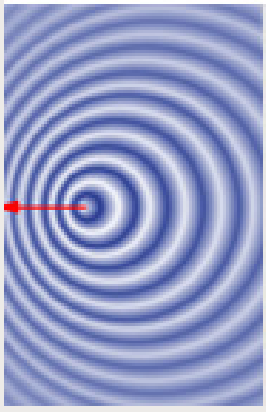


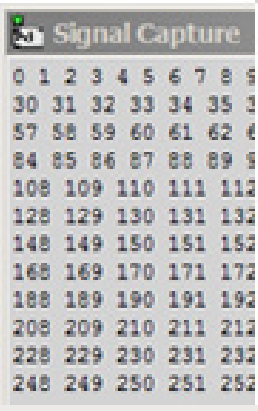


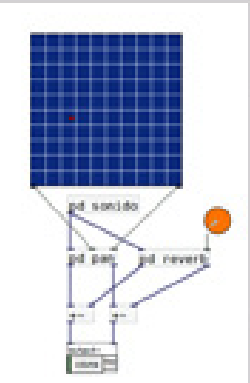


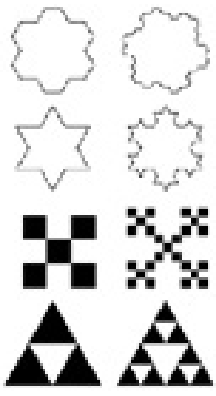












Cetta, Pablo
Captura y procesamiento de sonido. - 1a ed. - Bernal : Universidad Virtual de Quilmes, 2014.
E-Book.

ISBN 978-987-1856-82-4

1. Enseñanza de Música. 2. Tecnología Educativa. 3. Enseñanza Universitaria. I. Título.
CDD 780.285 711

Diseño instruccional y procesamiento didáctico: Bruno De Angelis y Ana Elbert

Diseño, diagramación y desarrollo web: Alejandro Jobad

Programación: Marisol Martin

Primera edición: febrero de 2014

ISBN: 978-987-1856-82-4

Roque Sáenz Peña 352, (B1876BXD) Bernal, Buenos Aires
Teléfono: (5411) 4365 7100
http://www.virtual.unq.edu.ar

Íconos

Leer con atención

Son afirmaciones, conceptos o definiciones destacadas y sustanciales que aportan claves para la comprensión del tema que se desarrolla.

Para reflexionar

Propone un diálogo con el material a través de preguntas, planteamiento de problemas, confrontaciones del tema con la realidad, ejemplos o cuestionamientos que alienten la autorreflexión.

Texto aparte

Contiene citas de autor, pasajes que contextualicen el desarrollo temático, estudio de casos, notas periodísticas, comentarios para formular aclaraciones o profundizaciones.

Pastilla

Incorpora informaciones breves, complementarias o aclaratorias de algún término o frase del texto principal. El subrayado indica los términos a propósito de los cuales se incluye esa información asociada en el margen.

Cita

Se diferencia de la palabra del autor de la Carpeta a través de la inserción de comillas, para indicar claramente que se trata de otra voz que ingresa al texto.

Ejemplo

Se utiliza para ilustrar una definición o una afirmación del texto principal, con el objetivo de que se puedan fijar mejor los conceptos.

Para ampliar

Extiende la explicación a distintos casos o textos como podrían ser los periodísticos o de otras fuentes.

Actividades

Son ejercicios, investigaciones, encuestas, elaboración de cuadros, gráficos, resolución de guías de estudio, etcétera.

Audio

Fragmentos de discursos, entrevistas, registro oral del profesor explicando algún tema, etcétera.

Audiovisual

Videos, documentales, conferencias, fragmentos de películas, entrevistas, grabaciones, etcétera.

Recurso web

Links a sitios o páginas web que resulten una referencia dentro del campo disciplinario.

Lectura obligatoria

Textos completos, capítulos de libros, artículos y papers que se encuentran digitalizados en el aula virtual.

Lectura recomendada

Bibliografía que no se considera obligatoria y a la que se puede recurrir para ampliar o profundizar algún tema.

Índice

El autor

Pablo Cetta

Pablo Cetta realizó sus estudios musicales en la Facultad de Artes y Ciencias Musicales de la Universidad Católica Argentina, donde obtuvo el Doctorado en Música en la especialidad Composición, y paralelamente, cursó estudios de composición con Gerardo Gandini. Realizó una extensa actividad docente en su país, dictando clases de grado y posgrado en la UBA, UNQ e IUNA, entre otras instituciones. Participó como investigador y director de diversos proyectos vinculados a la composición asistida y procesamiento en tiempo real de sonido. Recibió varias becas y encargos en el país y en el exterior, entre ellas de la Fundación Antorchas; LIPM y Fundación Rockefeller, en un proyecto de intercambio con la UCSD y la Universidad de Stanford; Fondo Nacional de las Artes; IMEB de Bourges; LIEM de Madrid y Ministerio de Educación y Cultura de España. Obtuvo, además, la Beca Antorchas, el Premio Municipal de Música, Segundo Premio Nacional, Premio Argentores en el género Música para Teatro, Primer Premio en el Concurso Internacional de Bourges y el Premio Euphonies d´Or, en Francia, por su obra.

Introducción

Las primeras experiencias significativas que vinculan a la tecnología electrónica con la música tienen lugar a fines de la década de 1940. Entre ellas, se destacan las creaciones sonoras producidas por Pierre Schaeffer en Francia, a partir del uso de cintas magnetofónicas para la transformación y reproducción de objetos sonoros. A este género musical, conocido como *musique concrète*, se contrapone luego el de la *elektronische musik*, con base en la síntesis de sonidos complejos a partir de la combinación de sonidos puros. La música electrónica encuentra su origen en Alemania, y cuenta con el compositor Karlheinz Stockhausen como principal exponente.

Por otro parte, surge en la década de 1950, en Estados Unidos, la generación de sonido y música mediante la utilización de las primeras computadoras. Donde Max Mathews crea en los laboratorios de la empresa Bell Telephone el lenguaje Music I, considerado actualmente como el punto de partida de todos los entornos de síntesis y procesamiento de sonido actuales utilizados en *computer music*. También en esos momentos, Lejaren Hiller alcanzó sus primeros logros en el campo de la composición algorítmica asistida por computadora, en la Universidad de Illinois. El legado de estos y otros pioneros ha guiado el desarrollo de diversos lenguajes y aplicaciones destinadas al tratamiento del sonido y la música con medios tecnológicos.

Los desarrollos continuaron, pero es a partir de la década de 1990 que las computadoras alcanzan una rapidez de cálculo tal que permiten la generación y procesamiento del sonido en tiempo real. La interacción entre instrumentos musicales tradicionales y sonidos grabados, propia de las técnicas mixtas, se ve potenciada de manera notable, dando lugar a nuevas formas de expresión.

El crecimiento sostenido de las capacidades de procesamiento de la información, sumado a las posibilidades de vinculación entre dispositivos, da lugar hoy al procesamiento conjunto de imagen y sonido en tiempo real, y a la intervención tanto de intérpretes como de espectadores en el desarrollo de la obra artística, ya sea de forma presencial o mediante redes de comunicación. Los géneros se multiplican y se plantean nuevos desafíos que requieren de la investigación en distintos campos de conocimiento para materializarse.

Problemática del campo

El desarrollo de aplicaciones informáticas para el procesamiento de sonido en tiempo real es un campo multidisciplinario en el que convergen áreas de conocimiento diversas. Entre las principales: acústica, psicoacústica, matemática, procesamiento digital de señales e informática. Si consideramos, además, las interfaces que suelen aplicarse en el control de los procesos que tienen lugar en determinadas representaciones artísticas vinculadas con la tecnología, la electrónica ocupa también un lugar de importancia.

El dominio de cada una de estas áreas exige un estudio profundo, sostenido en el tiempo, e insoslayable en el caso de aquellos interesados en cubrir todos los aspectos tecnológicos involucrados en la producción de obras multimediales que den al sonido un rol protagónico. No obstante ello, es posible hallar un punto de partida y una metodología accesible, que proponga los conocimientos necesarios en la medida de los requerimientos de cada proyecto a emprender. De este modo, iremos cubriendo diferentes aspectos del procesamiento sonoro, partiendo de la síntesis, y pasando por el uso de filtros, retardos temporales, transformaciones espectrales y localización espacial del sonido, hasta llegar a la generación de estructuras musicales simples. Una vez finalizado el curso, habremos afianzado temas indispensables que nos permitirán acceder a niveles de complejidad mayores, tanto en relación con la bibliografía especializada para cada área, como en la capacidad de imaginar y diseñar configuraciones, dispositivos y programas aplicables a la producción artística.

Reflexiones acerca del aprendizaje en un entorno virtual

Tratar temas tecnológicos a través de una plataforma educativa con base en la tecnología presenta interesantes ventajas. La posibilidad de interactuar, alternando entre los contenidos a estudiar y la puesta en práctica de los mismos, potencia sustancialmente la capacidad de aprender.

El eje central de nuestra asignatura se orienta a la programación de aplicaciones, por lo cual la computadora es sin duda la principal herramienta. No solo podremos valernos de textos digitalizados, visitas a sitios web o recursos multimediales para acceder a los conocimientos teóricos propios de la disciplina, sino que podremos contar también con un entorno de programación versátil para la práctica y el desarrollo del software de procesamiento de audio a crear, que incorporaremos a nuestra estación de trabajo.

La inclusión de técnicas hipertextuales en la escritura aportó en su momento una estructuración no lineal del texto, estableciendo jerarquías, niveles de profundidad y vínculos con temas de interés. Hoy contamos con lenguajes de programación basados en interfaces gráficas, didácticos y de fácil manejo que incluso representan sus textos de ayuda mediante programas interactivos, construidos con el mismo lenguaje.

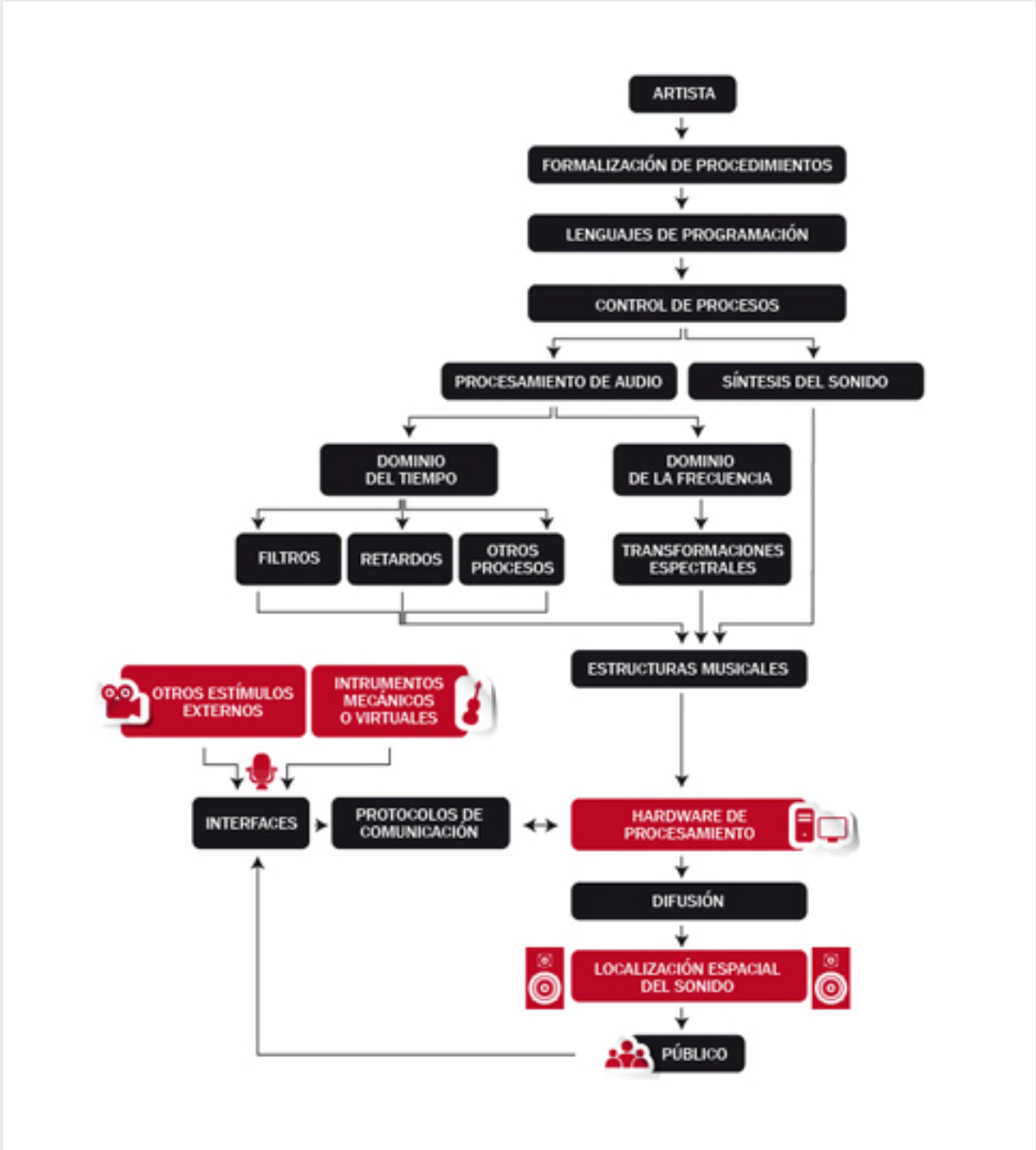
La enseñanza a través de entornos virtuales se revela como una opción atractiva, eficaz y naturalmente aplicable a las temáticas que aquí abordaremos. No obstante, los entornos constituyen simplemente un medio, y no un fin en sí mismo, por lo cual concentraremos nuestro mayor esfuerzo en brindar los contenidos que consideremos más adecuados para la asignatura, dispuestos de la forma más clara, precisa y efectiva posible. En definitiva, se trata de transmitir conocimientos, estudiar, investigar, reflexionar, crear y producir, pero a partir del aprovechamiento de los medios tecnológicos actuales, que sin duda facilitarán notablemente la tarea que nos proponemos emprender.

Objetivos del curso

Los objetivos del curso son:

- Conocer los fundamentos teóricos del procesamiento digital de señales de audio.
- Diseñar dispositivos virtuales y procedimientos que sirvan a la producción y transformación de sonido y música.
- Adquirir destreza en la programación de aplicaciones de síntesis y procesamiento de audio en tiempo real, orientadas a la creación de instalaciones sonoras o multimediales.
- Obtener los conocimientos necesarios para enfrentar con éxito la coordinación de proyectos que vinculen al lenguaje sonoro y a otras disciplinas artísticas mediante los usos tecnológicos actuales.

Mapa conceptual



1. Síntesis y procesamiento de sonido en tiempo real

Objetivos

- Acceder al entorno de programación Pure Data. Proceder a su descarga, instalación y configuración.
- Introducir al estudiante en las generalidades del lenguaje de síntesis y procesamiento de audio.

1.1. Introducción

Los sistemas de procesamiento de sonido en tiempo real que estudiaremos a lo largo de esta Carpeta, realizan transformaciones de las señales de audio que ingresan a una computadora, y difunden el resultado de tales procesos a través de parlantes. Asimismo, estos sistemas son capaces de crear o generar, por ellos mismos, las señales a reproducir a través de diversas técnicas de síntesis del sonido. Los procesos que intervienen son muy variados y los parámetros que los definen pueden ser controlados mediante distintos tipos de interfaces (pantallas sensibles al tacto, instrumentos MIDI, cámaras web, sensores de movimiento, de presión, de proximidad, etc.). Incluso, estos procesos pueden modular a partir de las cualidades mismas del sonido que es capturado para su transformación (su intensidad, altura, registro, o grado de tonicidad) a partir de un análisis del mismo.

MIDI *(Musical Instruments Digital Interface)* es un protocolo de comunicación entre instrumentos musicales electrónicos, creado en 1983. La sigla MIDI también designa a la interfaz física integrada a los instrumentos que permite la conexión entre dispositivos, aún cuando estos pertenezcan a distintos fabricantes. El protocolo MIDI será tratado en la Unidad 10.

Para tratar los aspectos teóricos y prácticos del procesamiento de señales de audio en tiempo real, y la creación de aplicaciones destinadas al tratamiento de sonido y música, vamos a utilizar el entorno gráfico de programación denominado Pure Data *(PD)*.

PD no solo está orientado al procesamiento de audio, sino que posee extensiones para el tratamiento de imágenes y video. Es un *software* libre, creado para las plataformas Windows, IRIX, GNU/Linux, BSD y MacOS, y forma parte de los lenguajes de programación visual –con base en una interfaz gráfica que facilita el manejo– cuyos primeros desarrollos aplicables al sonido comienzan en el Institut de Recherche et Coordination Acoustique/Musique (IRCAM), en Francia.

El núcleo principal de PD fue desarrollado por Miller Puckette. Actualmente existen dos distribuciones, la denominada "vanilla" que es actualizada por el mismo Puckette, y "PD-extended", una versión con múltiples librerías externas, documentación y un programa que facilita la instalación del *software*.

EL IRCAM fue fundado por el compositor y director de orquesta Pierre Boulez. Para conocer detalles del *I*nstitute de Recherche et Coordination Acoustique/Musique visite <http://www.ircam.fr/>

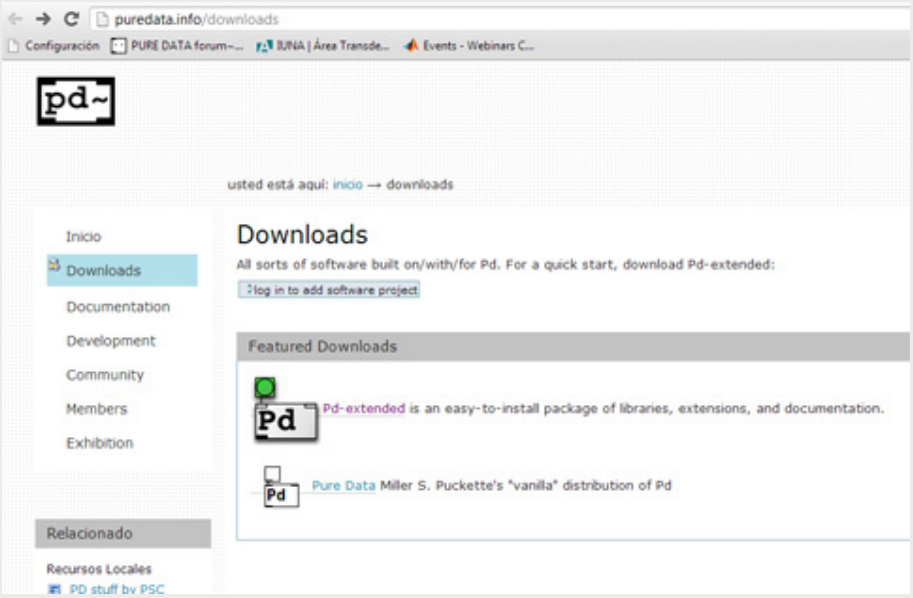
Miller Puckette es autor del programa de procesamiento de audio en tiempo real *Max-MSP*, desarrollado en el IRCAM. Posteriormente, creó Pure Data en el *Center for Research in Computing and the Arts* (CRCA), de la Universidad de California en San Diego. Para saber más sobre este autor: <http://es.wikipedia.org/wiki/Miller_Puckette>

También puede visitar la página del CRCA a través de <http://crca.ucsd.edu/>

1.2. Descarga e instalación de Pure Data

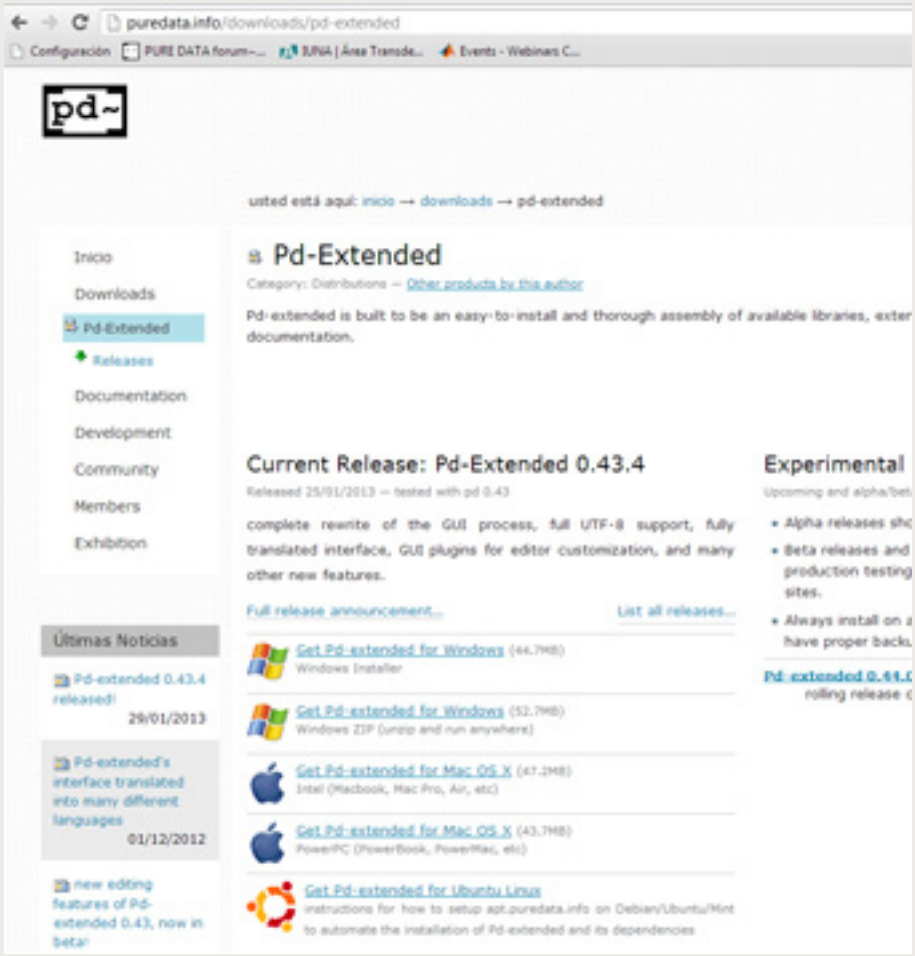
Para proceder a la descarga del instalador nos dirigimos a la dirección <http://puredata.info/downloads>, donde observaremos la siguiente pantalla:

G.1.1. Descarga de PD



De las dos distribuciones posibles, vamos a seleccionar la primera de ellas, PD-extended. Al hacer clic sobre el enlace, pasamos a la página siguiente que se reproduce a continuación:

G.1.2. Selección de la versión de PD



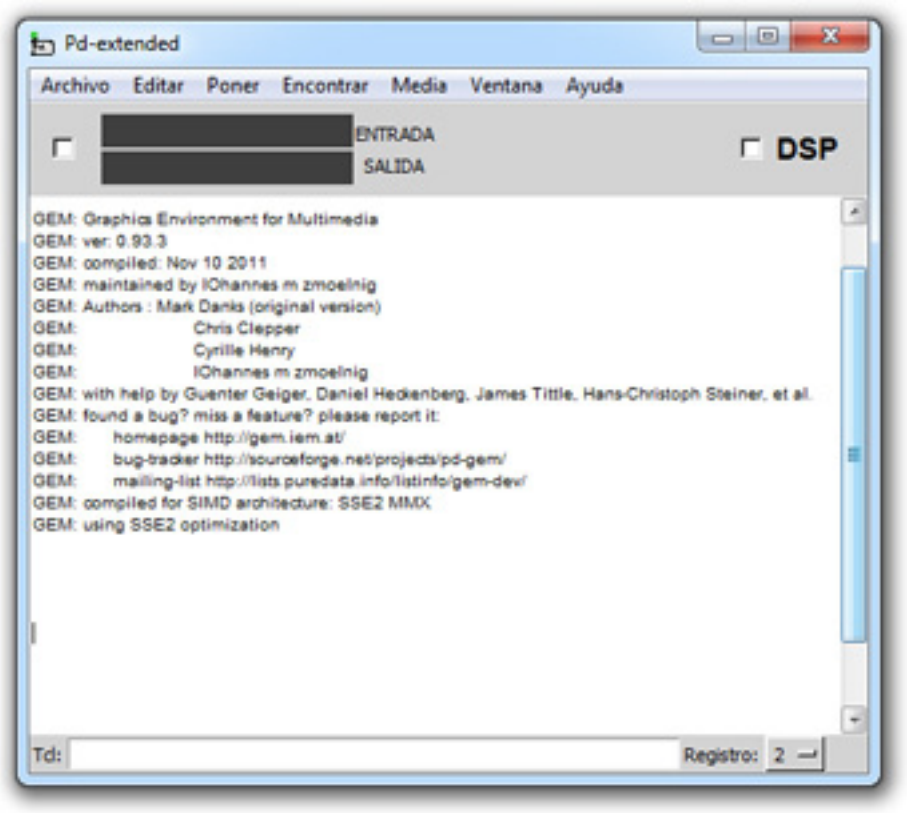
En esta página vamos a seleccionar la versión adecuada para el sistema operativo que tenemos instalado en nuestra computadora. Una vez que hagamos clic en el enlace deseado, comenzará la descarga.
Completada la descarga, ejecutaremos la aplicación de instalación.

1.3. Configuración de PD

Una vez instalado el programa procedemos a ejecutarlo. Al inicio, debe aparecer la pantalla de G.1.3., a la que vamos a denominar "ventana de PD". Esta ventana nos brinda información valiosa para la programación, como mensajes de error o resultados de ciertas acciones; y nos permite acceder a las distintas opciones del menú.

En el menú **Media**, en la opción **Preferencias de Audio**, podemos seleccionar los dispositivos de entrada y salida instalados en nuestra computadora. En caso de que dispongamos de más de una placa de sonido, elegiremos cuál de ellas va a ser utilizada por PD. En la opción **Preferencias de MIDI**, podremos hacer lo propio con los dispositivos de entrada y salida de datos MIDI (interfaces para controlar instrumentos MIDI externos, sintetizadores de sonido incorporados a la placa de audio, etc.). Si nuestra computadora dispone de una única placa de sonido, incluso las del tipo *on-board*, no se tornará indispensable acceder a estas opciones, pues PD hará la selección por nosotros. PD posee un desarrollo muy sólido y no suele presentar conflictos de configuración.

G.1.3. Ventana de PD



Dentro del menú **Media** encontraremos, además, la opción **Probar Audio** y **MIDI**; se trata de una aplicación realizada en PD que nos ayuda a probar las entradas y salidas de audio y mensajes MIDI.

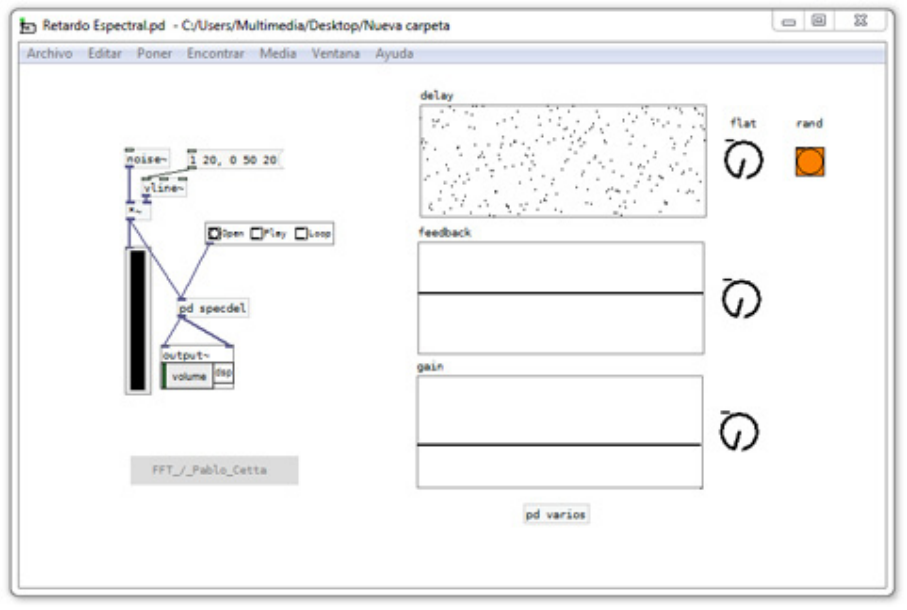
1.4. Descripción general del lenguaje

Pure Data es un lenguaje basado en una interfaz gráfica, con objetos interconectables mediante cables virtuales, que facilita la programación a usuarios no expertos y que, a la vez, sirve como soporte de librerías creadas por terceros. Algunas de estas librerías, incluso, se encuentran destinadas al procesamiento de imagen en tiempo real.

En G.1.4. observamos un programa realizado en PD que sirve para el procesamiento de retardos espectrales de señales de audio. No nos interesa por ahora aprender qué tareas realiza ni cómo fue programado, sino mostrar cómo se interrelacionan los objetos entre sí mediante conexiones virtuales, y apreciar los elementos gráficos que sirven a la comunicación entre el programa y los usuarios.

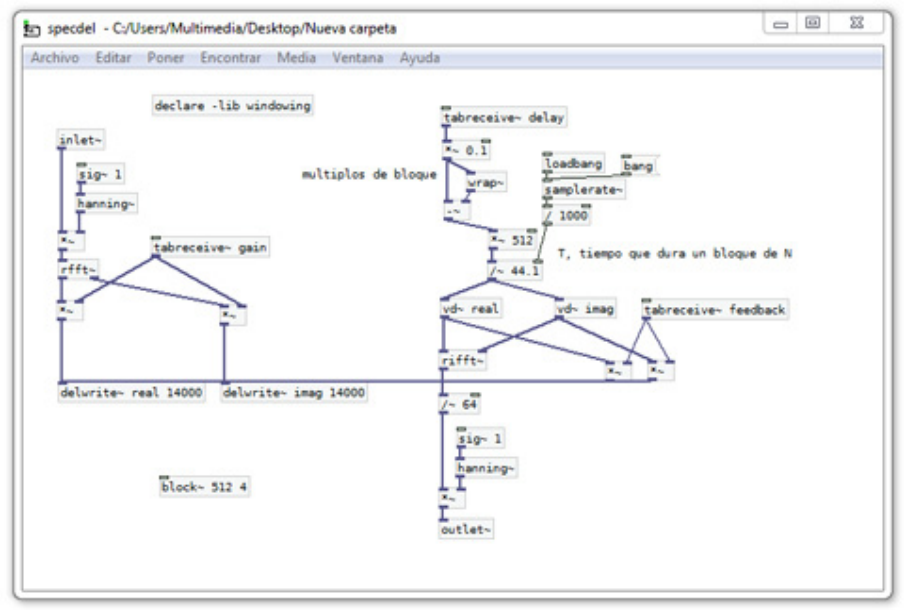
La programación se desarrolla de forma estructurada, encapsulando partes de la aplicación en nuevos objetos, con funciones específicas, creados por el programador. Al programa en general se lo suele denominar *patch* (por analogía con el modo de interconectar módulos en los primeros sintetizadores analógicos), y a los subprogamas *subpatches* o abstracciones, según como hayan sido concebidos.

G.1.4. Un *patch* de PD



G.1.5. muestra un *subpatch* del programa anterior, dedicado a una tarea específica dentro de la aplicación. De acuerdo con este paradigma, la programación se estructura modularmente, y el programa puede ser leído por capas, establecidas de acuerdo con un orden de importancia.

G.1.5. Un *subpatch* del programa de la figura anterior



Los objetos de PD contienen funciones diseñadas para realizar tareas específicas, y se dividen en tres grandes grupos: objetos de control, objetos de audio y objetos de interfaz con el usuario o GUI *(Graphic User Interface)*. Los de control, manejan información alfanumérica y están destinados principalmente al control de procesos. Los objetos de audio, por su parte, son capaces de procesar señales digitales, lo cual implica operar con un volumen importante de información. Pensemos que para producir una señal monofónica de audio es preciso generar en tiempo real una cantidad de muestras igual a la frecuencia de muestreo elegida (44.100 muestras por segundo, por ejemplo, o aún más). Visualmente, distinguimos unos de otros a través del tilde (~) que poseen los objetos de audio al final del nombre (*osc~*, que es un oscilador, o *dac~*, un conversor digital-analógico). Por último, los objetos GUI sirven para comunicarnos con el programa, y representan perillas, botones, *radio buttons, sliders,* recuadros numéricos, vúmetros o paneles.

La distribución PD-extended incorpora una gran cantidad de librerías desarrolladas por terceros, que amplían considerablemente las posibilidades del lenguaje. Si instalamos PD en el directorio por defecto (por ejemplo, en Windows, c:\Archivos de Programa\PD), encontraremos una carpeta llamada "extra", en la cual residen estas librerías (unas 80, con gran cantidad de objetos en cada una). Entre ellas, la librería pduino, para comunicarnos con el dispositivo de hardware Arduino y realizar proyectos multimediales, *vbap y iem-ambi*, para localización espacial del sonido, o GEM, para procesamiento de imágenes y video en tiempo real.

GEM *(Graphics Environment for Multimedia)* es una librería externa de PD, desarrollada originalmente por Mark Danks y luego mantenida por otros autores, destinada al tratamiento de imágenes y generación de gráficos 3D a través de OpenGL. Más información, documentación y ejemplos en <http://gem.iem.at>

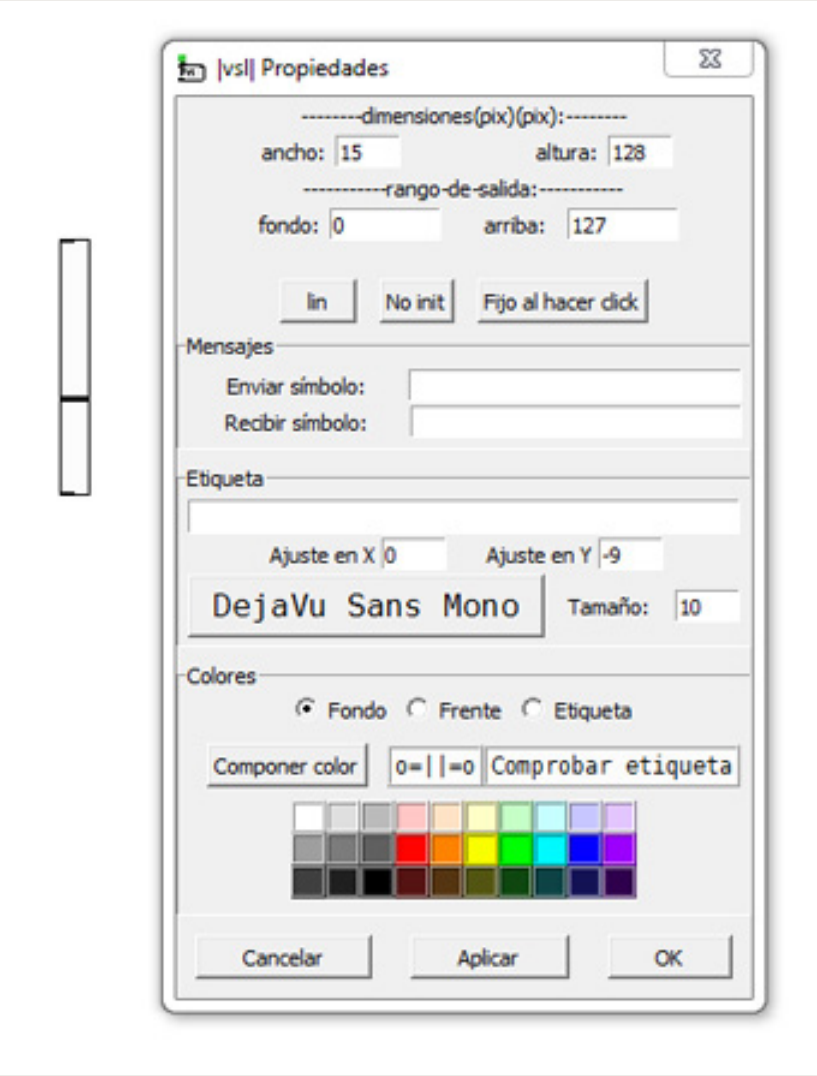
Arduino es un dispositivo electrónico capaz de recibir información de sensores, y de controlar actuadores, y otros dispositivos. Es ampliamente utilizado en la producción de obras multimediales. Para más información sobre Arduino puede consultar <http://www.arduino.cc/>

Mediante los objetos GUI configuramos la interfaz gráfica para la interacción entre el usuario y la aplicación. Cuando abrimos un programa deberíamos ver el panel de control de esa aplicación sin detalles de programación que distraigan la atención durante la operación (objetos, cables, etc.), a menos que se trate de programas sencillos o con fines didácticos.

G.1.6. Panel de control de una aplicación de síntesis aditiva

Para los objetos GUI disponemos de una ventana de **Propiedades** (accedemos desde el menú que se despliega al hacer clic derecho sobre el objeto), en la que es posible establecer sus dimensiones, colores, rango, comportamiento y modos de conexión sin utilizar cables (conexiones remotas, que veremos más adelante).

G.1.7. Propiedades de un objeto GUI *(slider)*

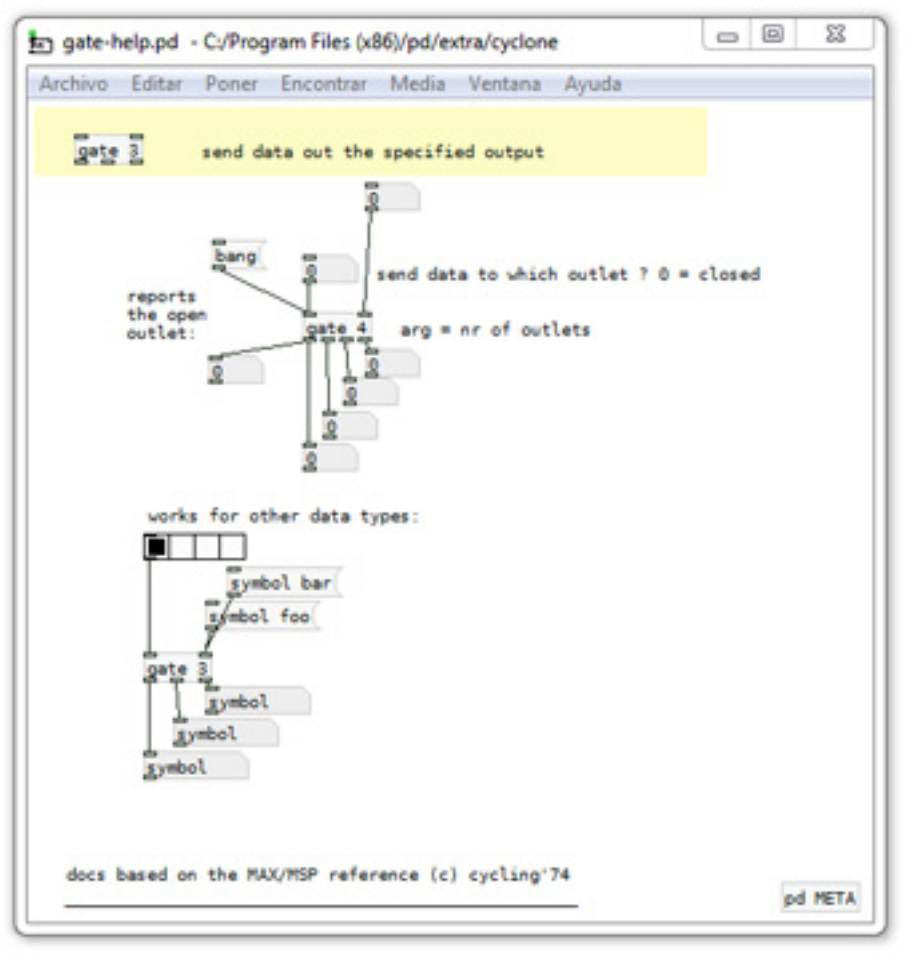


1.5. Recursos de ayuda

Otra característica importante de PD –ya existente en otros programas, como Max-MSP– es que la ayuda sobre el funcionamiento de los objetos que brinda el programa no es un mero texto, sino un patch de PD, el cual podemos operar e incluso modificar.

Al presionar el botón derecho del mouse sobre cualquier objeto se despliega un menú, y si allí elegimos la opción **Ayuda**, podremos acceder a una aplicación en PD que explica las características del objeto, sus entradas *(inlets)* y salidas *(outlets)*, así como sus argumentos (valores que se inscriben al momento de crearlo). G.1.6. muestra la ventana de ayuda para el objeto de control *gate*, incluido en la librería externa denominada Cyclone.

G.1.8. Ayuda para el objeto *gate* de la librería cyclone



Asimismo, podemos obtener ayuda sobre PD en general si vamos al menú **Ayuda** de la ventana de PD, seleccionamos el ítem **Search**, y luego elegimos el link **index.htm** (ver G.1.9.).

G.1.9. Documentación general de PD

Actividad 1

Descargue e instale el software Pure Data en su computadora. Proceda a su configuración, y explore sus características principales, abriendo archivos de ejemplos de la carpeta Patches, y accediendo a los recursos de ayuda.

2. Operaciones de control de datos

Objetivos

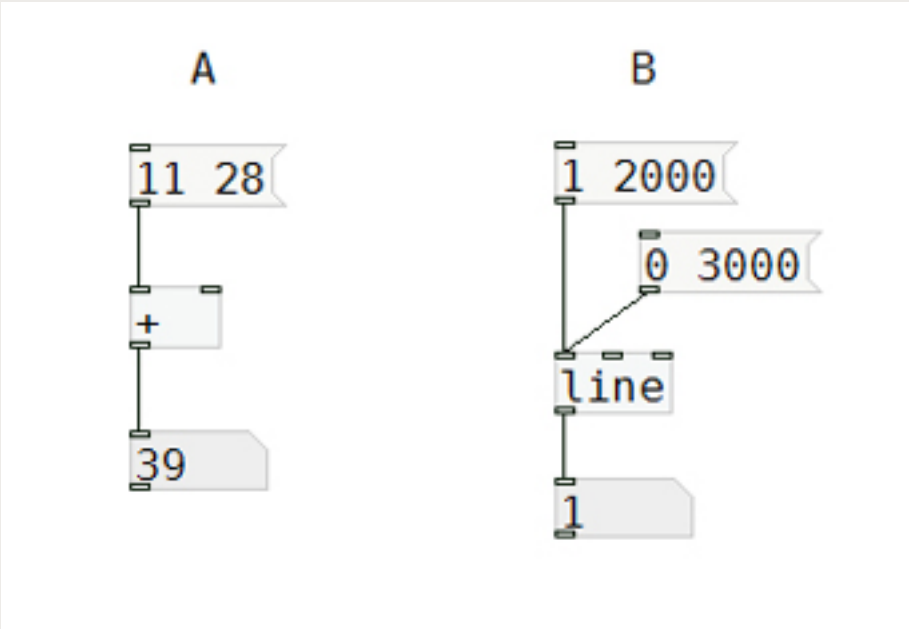
- Conocer los principales tipos de datos en PD y los principios de funcionamiento de los objetos de control.
- Aplicar objetos aritméticos y de control de datos en la programación de aplicaciones sencillas.

2.1. Objetos y operaciones aritméticas

Para comenzar con la programación en Pure Data vamos a realizar un ejemplo simple de suma de dos números. Para ello, en la ventana de PD ingresamos al menú **Archivo** y elegimos la opción **Nuevo**. Observamos que se abre una ventana, sobre la que vamos a desarrollar nuestra aplicación.

En el menú **Poner**, elegimos la opción **Objeto**, y hacemos clic sobre la ventana. Vemos que aparece un rectángulo en línea rayada, y un cursor que parpadea a la espera del nombre del objeto que queremos invocar (ver G.2.1.).

G.2.1. Suma de dos números



Escribimos el símbolo +, y posteriormente hacemos clic sobre el fondo de la ventana. De este modo, habremos creado una instancia del objeto de suma.

Debemos ahora encontrar el modo de pasarle al objeto los números que deseamos sumar. Para esto, vamos nuevamente al menú ***Poner*** y seleccionamos la opción **Número**. Si repetimos la operación dos veces más, habremos ubicado tres objetos idénticos. Dos de ellos nos permiten ingresar números en el sumador, y el tercero recibir números desde su salida para representar visualmente el resultado de la suma. Luego, conectamos las salidas con las entradas, trazando cables virtuales con el **mouse**, finalizando así nuestro primer programa (G.2.2.).

En estos momentos nos encontramos todavía en la instancia de programación, o sea, en el modo de edición. Para poder utilizar nuestro programa deberemos pasar al modo de ejecución, mediante el cual se bloqueará la posibilidad de seguir editando.

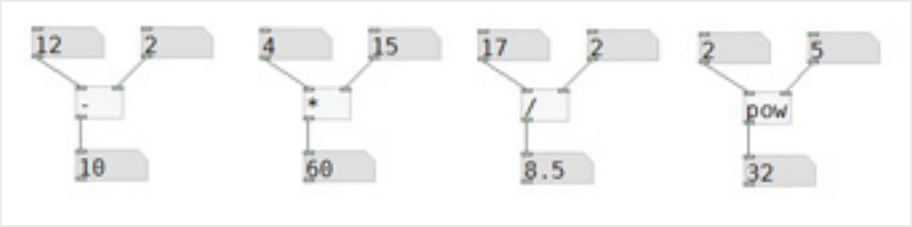
Para alternar entre el modo de edición y el modo de ejecución, simplemente presionamos la tecla **Control**, y sin soltarla, presionamos la tecla *"E"* (Ctrl+E). Estando en el modo de ejecución vemos que el puntero del **mouse** tiene ahora forma de flecha. Este modo nos permite actuar sobre los objetos **Número** sin que se muevan, haciendo **clic** y arrastrando el **mouse** sobre ellos para cambiar su contenido numérico. Probemos esa posibilidad, y también observemos qué sucede cuando movemos el **mouse** con la tecla **Shift** apretada. Vemos que también es posible representar números con decimales.

Un archivo de ejemplo del *patch* de suma que acabamos de realizar se encuentra en la carpeta *Patches*, con el nombre "01-Suma.pd". Esa carpeta contiene todos los ejercicios que veremos a lo largo del curso, nos servirán para analizar cómo están programados y para modificarlos y aprender de ello.

Si experimentamos el uso del programa creado notaremos que el objeto "+" solo arroja un resultado al cambiar los números que ingresan por la izquierda. El número que ingresa por la derecha es almacenado en el interior del objeto, a la espera del dato que entre por la izquierda y dispare la operación suma. A las entradas al objeto las denominamos **inlets**, y a las salidas, **outlets**. El objeto + posee dos *inlets*, el de la derecha es el *cold inlet*, y el de la izquierda el *hot inlet* (que dispara la suma). Esta particularidad forma parte de la sintaxis de PD, y es necesario tenerla en cuenta para evitar errores de programación, u obtener datos erróneos al utilizar los programas.

Del mismo modo que realizamos el programa de suma podemos desarrollar otros con los objetos de resta (-), multiplicación (*), división (/) y potenciación *(pow)*.

G.2.2. Otras operaciones aritméticas



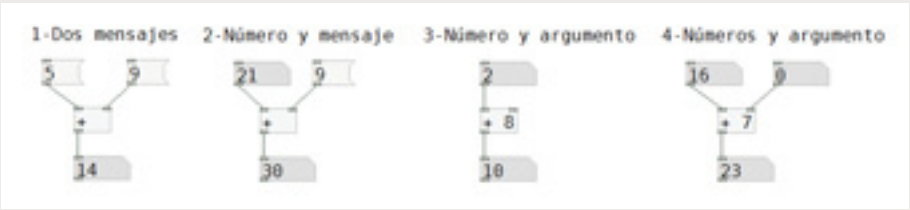
Abrir con PD el *patch* de G.2.2., denominado "02-Otras operaciones.pd", y experimentar su uso. Notar especialmente que en todos los casos el dato que ingresa por el *inlet* de la izquierda es el que dispara la operación; el que entra por la derecha solo se almacena en la memoria interna del objeto.

2.2. Mensajes y argumentos

Los mensajes pueden estar formados por números, palabras o la combinación de ambos tipos de datos. Para utilizar un mensaje vamos al menú **Poner**, y seleccionamos la opción **Mensaje (Ctrl+2)**. El objeto *Mensaje* sirve para establecer constantes, es decir, datos que no cambian durante la ejecución del programa. Los mensajes funcionan, además, como botones, pues para que su contenido llegue al objeto al que están conectados, es preciso presionarlos con el puntero del mouse.

El ejemplo "03-Mensajes.pd" emplea mensajes para determinar los números a sumar. Al operar el programa comprobamos que no es posible, estando en el modo de ejecución, modificar el contenido numérico de los mensajes. Esto se debe a que funcionan como constantes, a las que se les atribuye un valor fijo.

G.2.3. Mensajes



En el tercer ejemplo (señalado en G.2.3. como 3 - **Número y argumento**) encontramos al número 8 dentro de la caja del objeto, el cual obviamente fue escrito en el momento de creación del objeto de suma. Ese número conforma el argumento del objeto, y es tomado como valor inicial o como valor por defecto. El objeto suma siempre 8 al número que ingrese por el *inlet* izquierdo.

En el cuarto ejemplo, a lo anterior, agregamos un objeto **Número** en el *cold inlet*. Hasta acá sabemos que el número 7, establecido como argumento, es el valor que se sumará a cualquier número que ingrese por la izquierda. No obstante, si enviamos otro valor desde el objeto **Número** conectado a la derecha, el valor por defecto será sustituido por el nuevo número. Por esa razón, consideramos a los argumentos de un objeto como "valores por defecto"; su vida útil finaliza cuando es reemplazado por otro valor que ingresa al objeto desde el exterior.

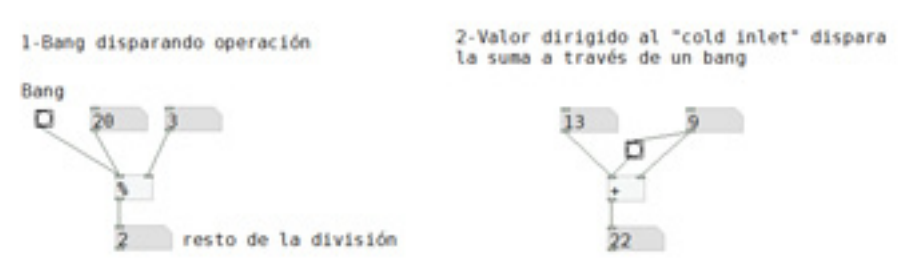
Por otra parte, en este *patch* incorporamos leyendas aclaratorias, que se denominan **Comentarios**.

Los comentarios son sumamente útiles, pues permiten agregar textos a nuestros programas, que en general explican los pasos o etapas que lo conforman. Para utilizar comentarios, una vez más nos dirigimos al menú **Poner (Ctrl+5)**.

2.3. El mensaje *bang*

El mensaje *bang* (onomatopeya de un disparo) sirve precisamente para disparar eventos. En general, podemos producirlo mediante un botón que lleva el mismo nombre que el mensaje. Encontramos ese botón en **Poner – Bang (Shift+Ctrl+b)**. El siguiente programa, en el primer ejemplo, utiliza un *bang* para disparar el cálculo del resto de una división de números enteros (mediante el objeto %). Si cambiamos el número que ingresa por la derecha, y después presionamos el *bang*, se obtendrá el resultado de la operación.

G.2.4. Uso del mensaje *bang*



En el segundo ejemplo, en cambio, utilizamos el botón *bang* para obtener el resultado de la suma cada vez que se cambia el número de la derecha. Recordemos que de por sí el dato que entra por el *cold inlet* no dispara la operación. En este caso, el botón *bang* "filtra" el contenido del objeto **Número**, y lo convierte en un mensaje *bang*, que ingresa luego por la izquierda y dispara la suma.

El *patch* "04-Mensaje *bang*.pd" contiene la programación que se observa en G.2.4.

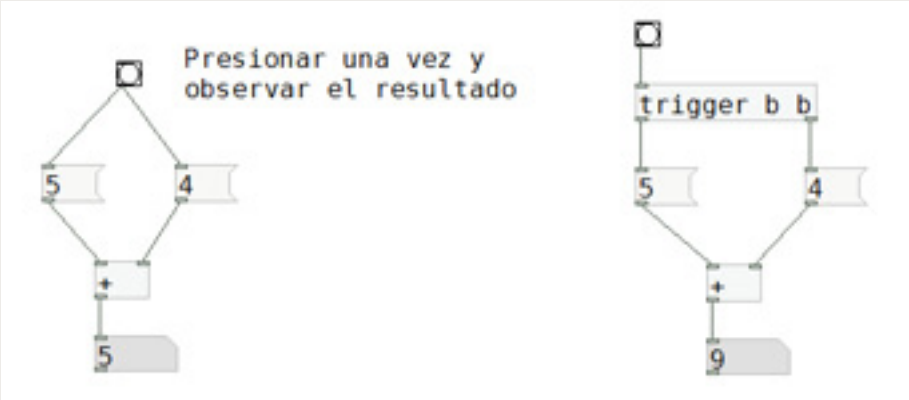
2.4. Flujo de la información

A fin de comprender cómo se establece el flujo de datos en los programas desarrollados en PD vamos a abrir el ejemplo siguiente para operar directamente sobre él.

Abrir el *patch* "05-Flujo de la información.pd" y leer atentamente el texto siguiente antes de accionarlo. Para repetir el efecto, puede cerrar el archivo y volver a abrirlo.

Observemos los dos ejemplos siguientes del *patch* 05. En el de la izquierda, cuando presionamos el botón *bang* por primera vez, vemos que el resultado de la suma da 5, lo cual es erróneo, pues debería ser 9. Solo cuando presionamos por segunda vez la respuesta es correcta.

G.2.5. Flujo erróneo y solución del problema



La razón de ello es que el número 5 del mensaje de la izquierda ingresa antes al objeto que el 4 de la derecha, y dispara la operación (5 + 0). Luego, al hacer *bang* por segunda vez, el 4 ya se encuentra en la memoria del objeto, y eso determina que la respuesta sea correcta al entrar nuevamente el 5 por la izquierda (5 + 4).

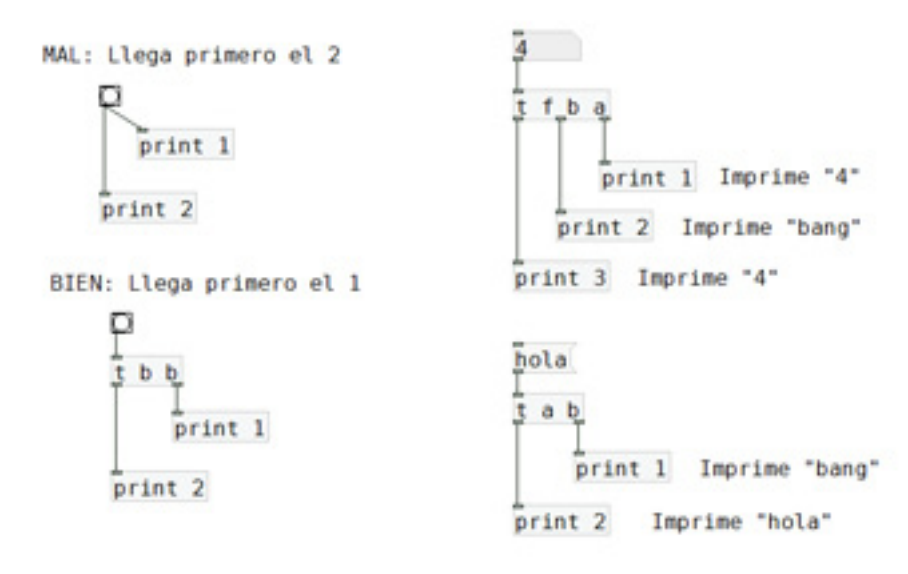
Para remediar esta situación podemos utilizar el objeto *trigger*, como se ve a la derecha de la figura anterior. Este objeto repite el *bang* de entrada a través de sus salidas, pero en estricto orden, de derecha a izquierda. De este modo, empleando *trigger* podemos garantizar el orden en el que van a entrar los datos en el objeto de destino.

El objeto *trigger* es muy usado, tanto que suele abreviarse su nombre con una simple *t*. Según vimos en G.2.5., hay dos letras *b* (de *bang*) ubicadas como argumentos. Cada letra que escribimos crea una nueva salida en el objeto, vale decir, el objeto tiene tantas salidas como letras escribamos como argumentos. Las letras permitidas son, además de la *b*, la *f, s, l, p* y *a*, y las iremos tratando en la medida que precisemos de ellas.

El criterio que guía a PD para controlar el flujo de la información se basa en el orden en que fueron conectados los cables virtuales. El cable que se conectó primero es el que primero conduce la información. Pero este criterio, si no es atendido rigurosamente, puede dar lugar a múltiples errores en la programación. Por ello, recomendamos emplear el objeto *trigger* cada vez que se presente una bifurcación que pueda dar lugar a resultados equivocados al ejecutar el programa.

Veamos otros ejemplos para afianzar estos conceptos (G.2.6.). A la izquierda imprimimos dos veces en la ventana de PD el mensaje *bang* del botón, pero notamos que no aparece en el orden buscado. Abajo, lo mismo, pero controlado con *trigger*. A la derecha de la figura utilizamos tres letras diferentes, la *a* (de *anything*) que replica cualquier dato de entrada a la salida; la *b* (de *bang*) que convierte cualquier dato a *bang* (si entra un 8 sale un *bang*); y la *f* (de *float*) que copia a la salida el contenido numérico de la entrada (si entra un 8 sale un 8). Por último, abajo a la derecha, vemos en el objeto *trigger* la *a* de *anything* y la *b* de *bang*, pero el dato de entrada es un símbolo (la palabra "hola"). Las salidas son, en orden respectivo, *bang* y *hola*.

G.2.6. Otro ejemplo de *trigger*



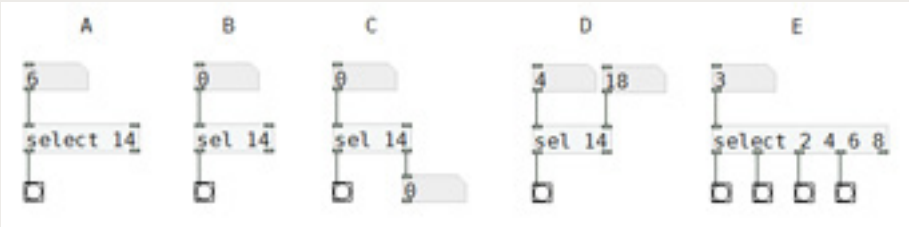
2.5. Compuertas y otros objetos de control de flujo

Pure Data cuenta con una serie de objetos que nos va a permitir controlar el flujo de datos en nuestras aplicaciones. Estos objetos sirven para seleccionar o filtrar números y para direccionar la información.

2.5.1. *Select*

Cuando deseamos saber si la salida de un objeto es un número en particular podemos valernos del objeto *selec*t. En G.2.7. observamos varios ejemplos que describen el modo de utilizarlo.

G.2.7. Selección de números mediante el objeto *select*



En *A*, el número 14 en el argumento determina el criterio de selección; cuando el número que ingresa es el 14, el objeto envía un *bang* a la salida. En *B*, vemos un alias de *select*, sel, implementado para ahorrar escritura, dado que este también es un objeto muy usado. En *C*, el *outlet* de la derecha envía los números que no cumplieron con la condición (todos menos el 14). En D, observamos que por el *inlet* derecho se puede cambiar el criterio de selección, y reemplazar el 14 del argumento por cualquier otro valor. Finalmente, en *E* vemos que un mismo objeto *select* puede reconocer más de una opción numérica, simplemente aumentando la cantidad de argumentos escritos en el objeto. Pero en este último caso, no resulta posible cambiar los argumentos por valores externos al objeto.

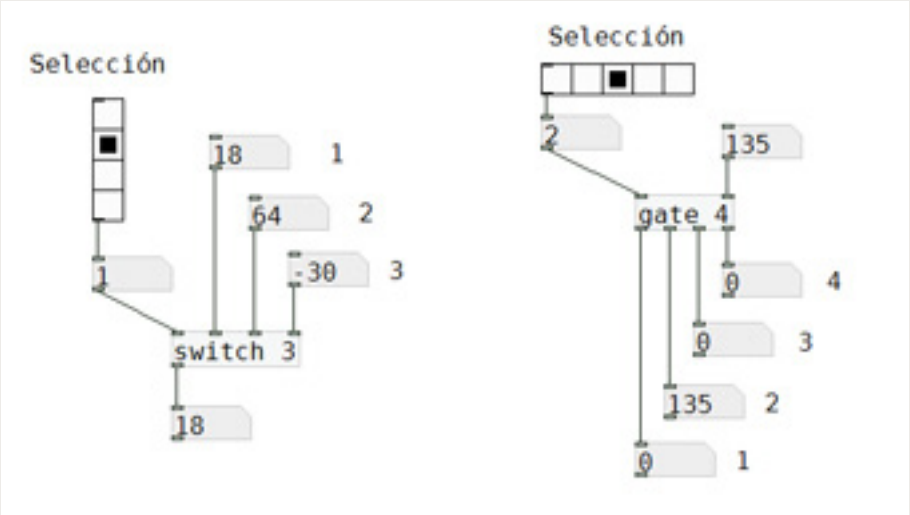
A través del *patch* "07-select.pd" podremos experimentar el uso de este objeto, ingresando valores o modificando la programación.

2.5.2. *Switch y gate*

El objeto *switch* dispone de varias entradas y una única salida. La cantidad de entradas está determinada por el argumento que escribamos al crear ese objeto. A *switch* ingresan varias líneas de datos numéricos, pero solo una –o ninguna– pasa a la salida. A través del número que entra por el primer *inlet* determinamos cuál de las entradas dejamos salir por el *outlet*. Si por el *inlet* izquierdo ingresamos un 0 (cero) ninguna entrada sale por su *outlet*; si ingresamos un 1, sale solo lo que entra por el segundo *inlet*; si entra en cambio un 2, sale lo que entra por el tercer *inlet*, y así sucesivamente.

El ejemplo "08-switch y gate.pd", cuya ventana se reproduce en G.2.8., implementa estas compuertas. A partir de su uso, comprenderemos mejor el funcionamiento y posibles aplicaciones de estos objetos de control.

G.2.8. Objetos compuerta *switch* y *gate*



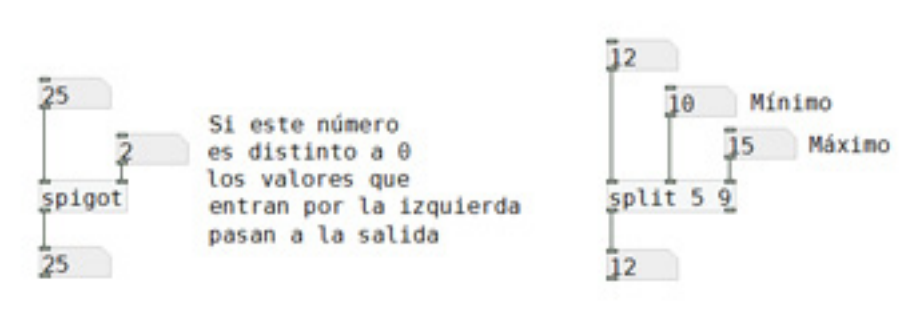
El objeto *gate*, por otra parte, funciona de manera inversa. Mediante el número que ingresa por el *inlet* izquierdo determinamos a qué salida se dirigen los números que entran por la derecha. Aquí también el valor del argumento que elijamos determina la cantidad de *outlets* del objeto. En ambos ejemplos empleamos un nuevo objeto gráfico, denominado *radio button*. Para *switch* utilizamos uno vertical, *Vradio* (que encontramos en el menú *Poner*), y para *gate* un *radio button* horizontal, denominado *Hradio*. La cantidad de opciones de selección de este objeto GUI, la especificamos en el menú de *Propiedades* del objeto (accedemos haciendo clic con el botón derecho del mouse sobre él), al igual que sus dimensiones y colores. Si presionamos el primer botón de un *radio button* obtenemos un cero (0), con el segundo botón, un uno (1) y así sucesivamente.

2.5.3. *Spigot y split*

El objeto *spigot* deja pasar a su salida cualquier número que ingresa por la izquierda, siempre y cuando su argumento, o el número que entra por la derecha, sea distinto de cero.

El objeto *split*, por su parte, deja pasar a su *outlet* izquierdo cualquier número que esté comprendido en cierto rango de números. Si el número ingresado se encuentra fuera del rango, sale por la derecha. En G.2.9., a la derecha, se observa que el rango se establece mediante dos argumentos, o bien a través del segundo *inlet* (valor mínimo del rango) y del tercero (valor máximo).

G.2.9. Objetos compuerta *spigot* y *split*



El ejemplo "09-spigot y split.pd", cuya ventana se reproduce en la figura anterior, implementa estas compuertas.

2.6. Números aleatorios, metrónomos y *timers*

Los objetos que analizaremos a continuación se destinan a la generación de eventos aleatorios, eventos periódicos y retardos temporales.

2.6.1. Generación de números al azar

Los números aleatorios resultan de interés pues pueden servir a la generación de procesos basados en el azar. Para generar números aleatorios utilizamos el objeto *random*. Cada vez que *random* recibe un *bang* por el *inlet* izquierdo genera un número al azar entre 0 y *N* - 1. El valor de *N* se especifica como argumento, o bien se ingresa desde el *inlet* derecho.

2.6.2. Generación periódica de mensajes *bang*

Cuando precisamos enviar mensajes *bang* a intervalos idénticos de tiempo, como si se tratara de un metrónomo que marca el paso del tiempo, utilizamos el objeto *metro*. Este objeto comienza a enviar los mensajes cuando ingresa un 1 por el *inlet* izquierdo, y se detiene al recibir un 0. El tiempo transcurrido entre un *bang* y el siguiente se especifica en milisegundos, a través de un argumento, o bien, ingresando el valor por la derecha.

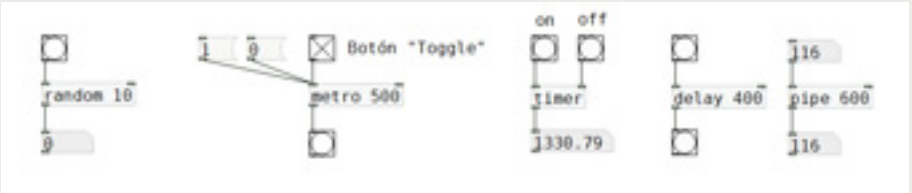
Para los objetos que actúan al recibir un 1 y se detienen al recibir un 0, como es el caso de metro, se suele utilizar un tipo de botón que al presionarlo sucesivas veces alterna entre esos dos valores. Ese botón se denomina **Toggl**e, y lo encontramos en el menú **Poner**, o bien apretando las teclas Shift+Ctr+T.

2.6.3. Medición del tiempo transcurrido y retardos temporales

A fin de medir el transcurso del tiempo entre un evento y otro nos valemos del objeto timer. La medición se inicia al enviar un *bang* al *inlet* izquierdo, y se detiene al enviar otro *bang*, pero al *inlet* derecho. El objeto devuelve el intervalo, expresado en milisegundos.

Para producir un retardo entre la entrada de un *bang* y su salida utilizamos el objeto *delay*. Expresamos el tiempo de retardo en milisegundos, como argumento del objeto, o a través del *inlet* derecho. Pero si, en cambio, lo que deseamos retrasar es la salida de un número, y no de un mensaje *bang*, empleamos el objeto *pipe*.

G.2.10. Azar y medición del tiempo

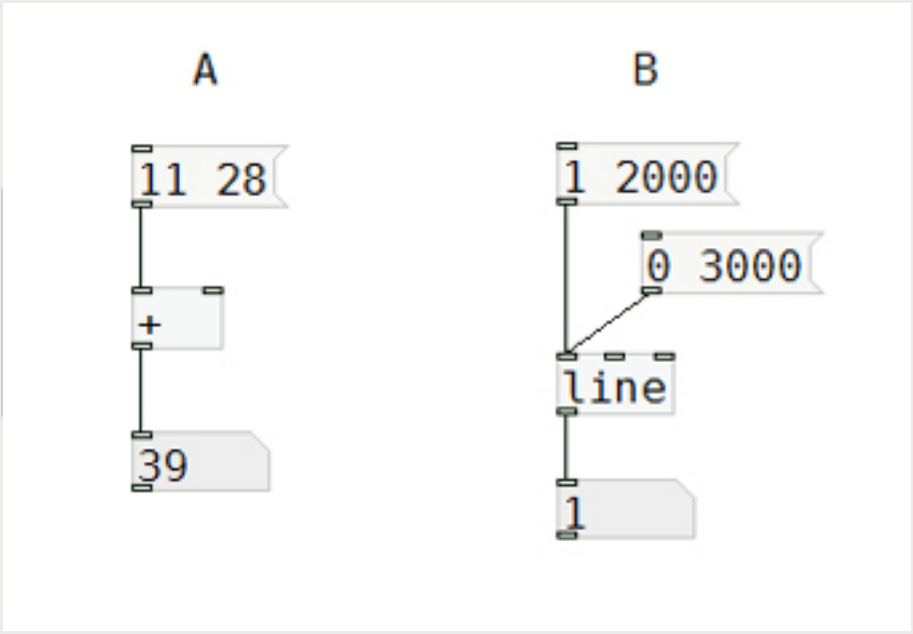


El archivo "10-random-metro-timers.pd" contiene ejemplos básicos con los objetos recién mencionados, y dos *patches* adicionales que combinan estos objetos.

2.7. Listas de datos

Los datos, sean numéricos o alfanuméricos (cadenas de caracteres) pueden agruparse formando listas. Una lista es un mensaje que contiene más de un dato en su interior. Hay objetos que utilizan listas como dato, entre ellos, uno que ya utilizamos: el objeto suma (+). Según se muestra en G.2.11. (A), una lista de dos números que ingresa por el *inlet* izquierdo también dispara la operación, y distribuye los sumandos en el objeto, de forma automática.

G.2.11. Objetos que reciben listas



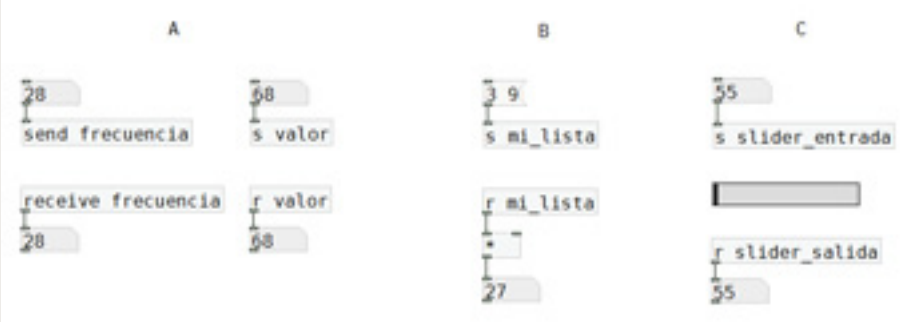
En G.2.11. (B) vemos un nuevo objeto llamado *line*. El objeto *line* genera una rampa de valores entre el valor actual y un valor de destino, en un tiempo determinado. Por ejemplo, si enviamos el mensaje con los valores "1 2000", el objeto generará una serie creciente de números entre 0 (valor actual) y 1 (valor de destino), en 2000 milisegundos (tiempo para realizarlo). Y si luego enviamos la lista "0 3000", decrecerá de 1 a 0 en 3 segundos.

Para probar los objetos **+** y **line** con listas, ejecutar el *patch* "11-listas de datos.pd".

2.8. Conexiones remotas

Es posible conectar dos o más objetos sin necesidad de utilizar cables virtuales, mediante conexiones remotas. Para efectuar una conexión inalámbrica de este tipo debemos utilizar dos objetos adicionales, uno que envía la información (como si se tratara de un transmisor) y otro que recibe (un receptor). El objeto que envía se llama *send*, y el que recibe *receive*, y son tan empleados que suelen crearse de forma abreviada mediante las letras *s* y *r*, respectivamente. Para que un objeto *receive* sepa cuál objeto *send* le envía datos, es preciso poner en ambos objetos un nombre cualquiera de conexión, como argumento. De esta forma "*send* frecuencia", por ejemplo, solo enviará información a un objeto "*receive* frecuencia"; lo importante es que tengan un nombre en común, que en el ejemplo es la palabra *frecuencia*. G.2.12. (A) ilustra lo que acabamos de explicar.

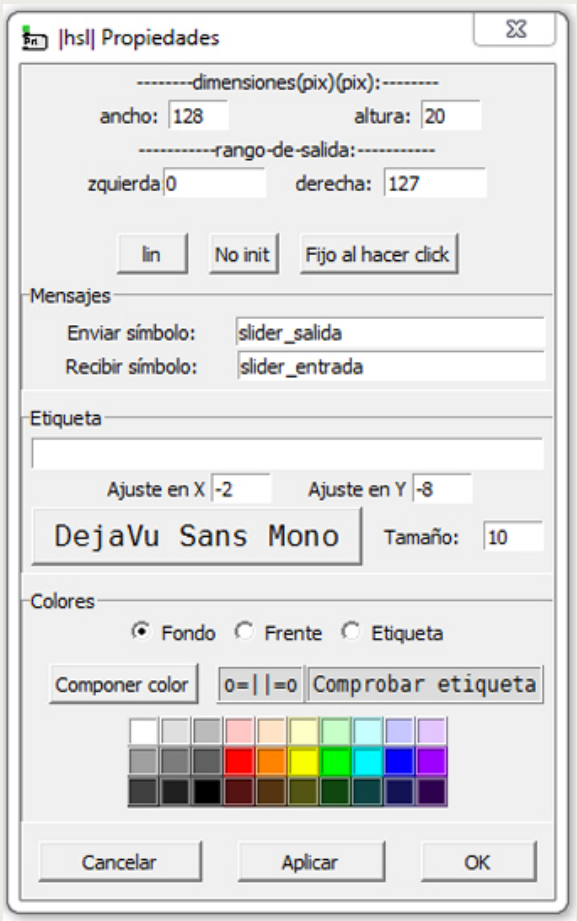
G.2.12. Conexiones remotas



G.2.12. (B) presenta algo similar, pero aquí se envía remotamente una lista de dos números a un sumador.

En el ejemplo C, por último, vemos otra manera de enviar y recibir datos de forma remota, en los casos en que utilizamos objetos de interfaz con el usuario (GUI). Para indicar los nombres de la conexión, tanto para recibir como para enviar información, accedemos a las *Propiedades* del objeto (haciendo clic sobre él con el botón derecho del mouse). Para el ejemplo empleamos un *slider* horizontal (objeto *Hslider* del menú **Poner**). La figura siguiente exhibe sus propiedades, donde se aprecian los nombres de conexión usados para enviar y recibir datos (observar los campos *Enviar símbolo* y *Recibir símbolo*, con los nombres *slider_salida* y *slider_entrada*, respectivamente).

G.2.13. Conexiones remotas en un objeto GUI



El archivo "12-conexiones remotas.pd" contiene la programación de los tres ejemplos analizados a partir de la observación de G.2.12.

Las conexiones remotas constituyen un recurso valioso para la programación, pues ayudan a lograr aplicaciones mejor estructuradas. La posibilidad de vincular los objetos gráficos que conforman el panel de operaciones de un programa sin la necesidad de utilizar cables también contribuye a una representación más clara de la información.

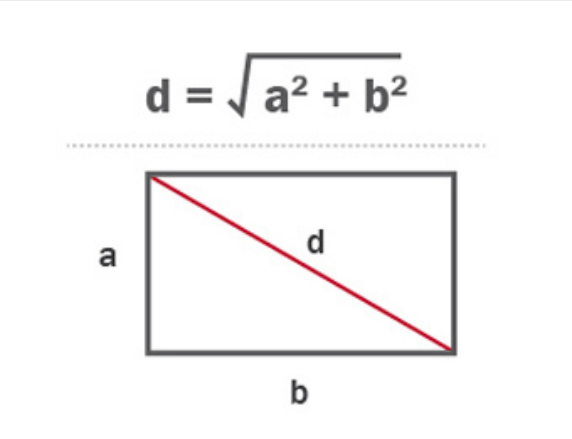
2.9. *Subpatches* y abstracciones

Una aplicación desarrollada en PD, o una parte de la misma, puede ser encapsulada en un objeto creado por el programador.

Para crear un subprograma o *subpatch* colocamos un objeto en la ventana y escribimos en él PD, un espacio, y un nombre que identifique qué función cumple, por ejemplo, "análisis" o "espacializador". Luego de escribir sobre el objeto, y al hacer clic en el fondo de pantalla, vemos que se despliega una ventana nueva. En esta ventana desarrollamos la programación de nuestro subprograma.

A modo de ejemplo, vamos a realizar un *subpatch* que calcule la diagonal de un rectángulo, aplicando el teorema de Pitágoras. G.2.14. ilustra el problema.

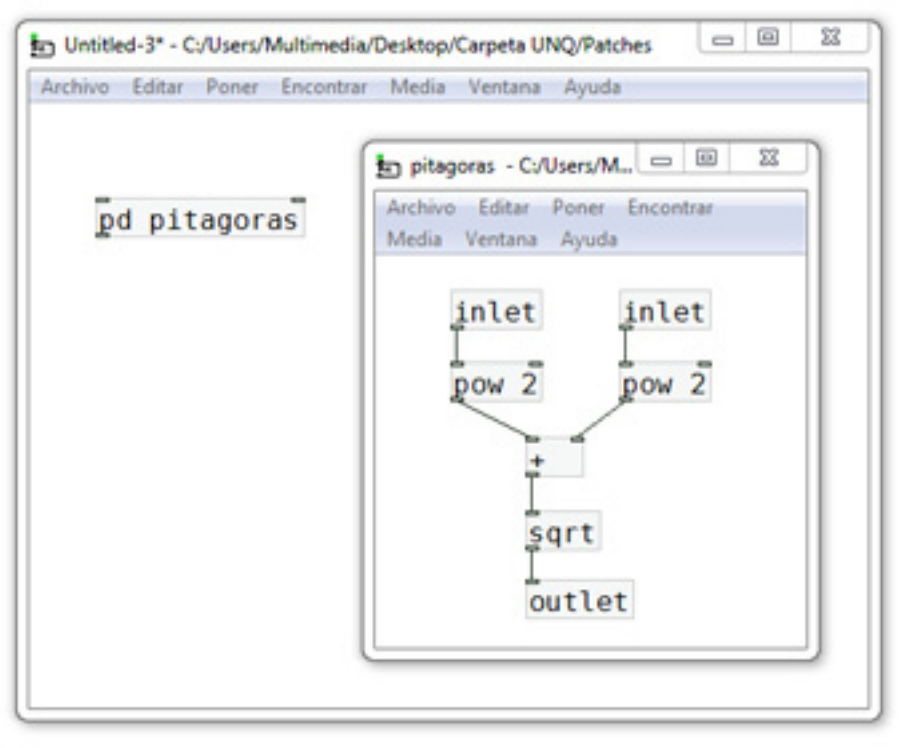
G.2.14. Teorema de Pitágoras



Ingresamos al *subpatch* dos datos, que son las longitudes de los lados *a* y *b*, y dentro de él vamos a implementar la fórmula del teorema. Para ello, precisamos crear en nuestro objeto dos entradas, y una salida que devuelva el resultado. Las entradas al objeto las generamos con el objeto *inlet*, y la salida con *outlet*.

La figura siguiente (G.2.15.) muestra el programa principal, donde creamos el objeto PD *pitagoras*, y la ventana asociada a este nuevo objeto, con la programación correspondiente. De este modo, realizamos un subprograma que calcula la longitud de la diagonal de un rectángulo, que puede ser duplicado cuantas veces deseemos, y que puede ser incorporado (copiando y pegando el objeto) en cualquier otra ventana o aplicación que desarrollemos.

G.2.15. Aplicación del Teorema de Pitágoras



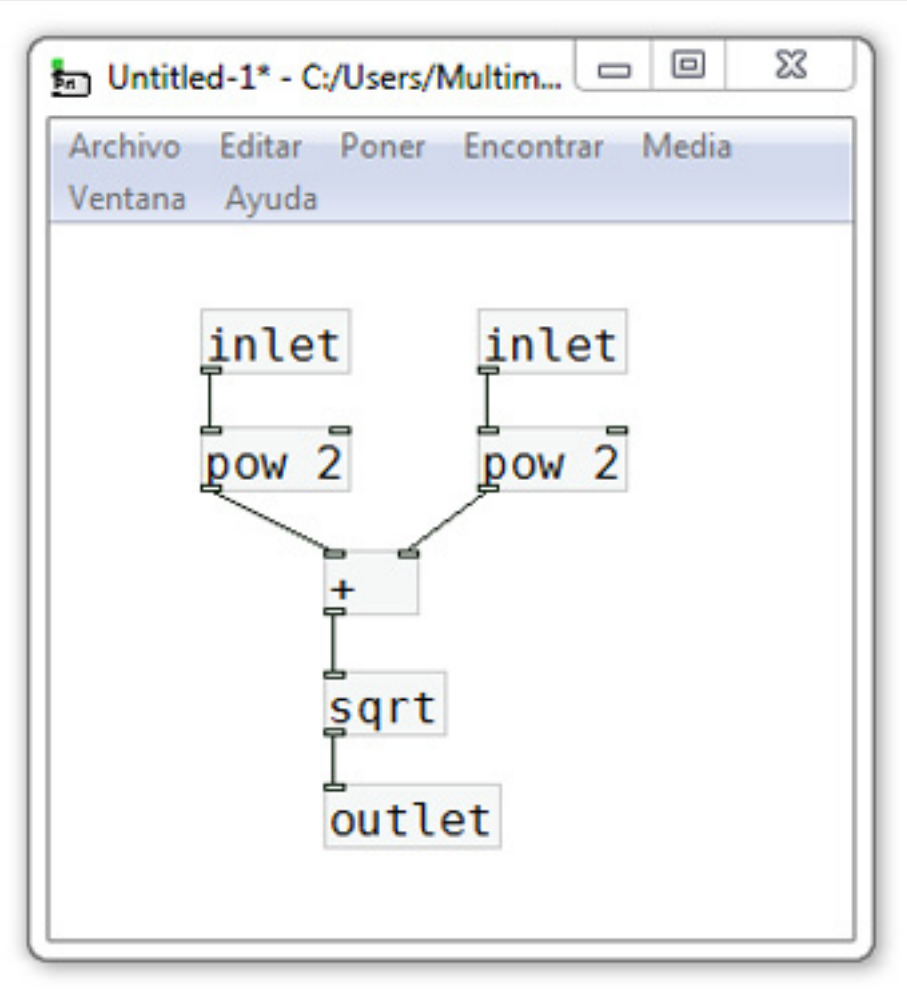
El programa *"13-subpatches.*PD" implementa el ejemplo recién visto, del cálculo de la longitud de la diagonal de un rectángulo. Para acceder a la ventana del *subpatch* "pd pitagoras" desde el modo de ejecución, simplemente hacemos clic sobre el objeto. Estando en el modo de edición, presionamos primero la tecla Ctrl y luego hacemos clic sobre el objeto.

El desarrollo de una aplicación a partir de subprogramas que cumplan tareas específicas es considerado de buena programación. Esto se debe a que el programa así creado presenta una estructura más sólida, posee diferentes niveles de lectura (de lo más importante a lo menos importante), y permite la reutilización del código para futuras aplicaciones.

Las abstracciones son *subpatches* almacenados en el disco rígido de la computadora. Para crear una abstracción, abrimos una ventana nueva y programamos en ella, del mismo modo que lo haríamos en la ventana de un *subpatch*, poniendo objetos *inlet* y *outlet* para crear las entradas y salidas del objeto a crear. Una vez finalizada la programación, guardamos el *patch* en el disco, prestando especial atención al elegir el nombre del archivo, pues va a ser el mismo nombre con el que vamos a llamar al objeto creado de este modo.

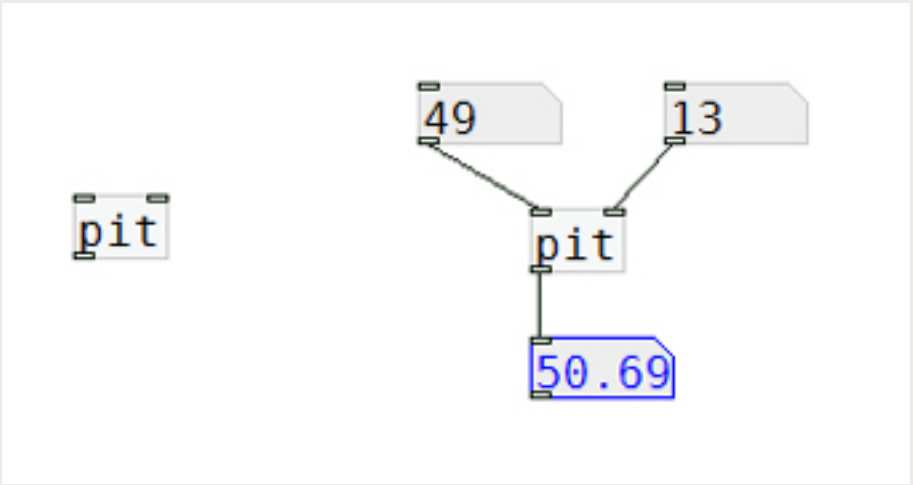
A fin de ilustrar la creación de abstracciones, vamos a convertir el ejemplo anterior, basado en el teorema de Pitágoras, en una abstracción. Para ello, abrimos una nueva ventana y programamos lo mismo que en el *subpatch* al que denominamos "pitágoras" (ver G.2.16.).

G.2.16. Creando una abstracción a partir del Teorema de Pitágoras



Se puede observar en la ventana anterior la leyenda *Untitled…*, lo cual indica que el *patch* aún no fue guardado. Procedemos, entonces, a guardar ese archivo con el nombre "pit.pd". Abrimos ahora una ventana nueva, donde vamos a invocar a esa abstracción, y la guardamos en el disco rígido como "mi_programa.pd" o cualquier otro nombre, en la misma carpeta donde ubicamos al archivo "pit.pd". En la nueva ventana colocamos un objeto vacío y escribimos dentro de él *pit*, y la abstracción ya está lista para ser usada.

G.2.17. Utilización de la abstracción



El *patch* correspondiente al tema recién tratado se encuentra en el archivo "14-abstracciones.pd". Aquí también, para acceder a la ventana de la abstracción desde el modo de ejecución, simplemente hacemos clic sobre el objeto. Estando en el modo de edición, presionamos primero la tecla Ctrl y luego hacemos clic sobre el objeto.

Es importante tener en cuenta que el archivo que contenga la abstracción debe encontrarse en el mismo directorio que el *patch* que la utiliza, pues, de otro modo, este último podría no hallarla.
No obstante, existe una manera de informar a PD en qué carpeta el usuario aloja sus abstracciones. Para ello, nos dirigimos al menú **Editar** y luego al ítem **Preferencias**. En la ventana que se despliega presionamos el botón **New**, y luego elegimos la carpeta donde almacenaremos nuestras abstracciones.

A fin de ampliar la información sobre PD y sus objetos de control, leer:
Sergi Jordà, "Capítulos 1 a 4",
en *Manual de Introducción a* PD,
publicación en línea disponible en:
<http://www.tecn.upf.es/~sjorda/PD/IntroduccionPD3.pdf>

Manual de Introducción
Este .pdf está en el servidor para su descarga.

Actividad 1

Realice un programa que, al recibir un mensaje *bang*, efectúe la suma, la resta, la multiplicación y la división de dos números generados por medio del azar. Utilice conexiones remotas para enviar los resultados de esas operaciones a los objetos *Número* correspondientes.

En el archivo "Respuesta Actividad 01.pd" encontrará la solución de este ejercicio. Compare los resultados que obtuvo con los del archivo, y anote las diferencias.

Actividad 2

a. El objeto *float* (o su forma abreviada *f*) sirve par almacenar números. Si un número ingresa por la entrada de la derecha, permanece en el objeto hasta que ingrese un *bang* por la izquierda, y envíe al número almacenado a la salida. Ubique en una ventana nueva de PD un objeto *float*, acceda al *patch* de ayuda del objeto y experimente sus características.

b. Programe un *patch* que, luego de recibir un *bang*, cuente desde 1 hasta 20, incrementando los números cada 100 milisegundos, y se detenga automáticamente al finalizar. Utilice los objetos *metro, float, selec*t, + y *Número*, y mensajes (con los números 0 y 1).

c. Cree una nueva versión del programa desarrollado que envíe los números de 1 a 10 a un objeto *Número*, y los números de 11 a 20 a otro.

d. Encapsule el programa realizado en un *subpatch*, con un *inlet* de ingreso del mensaje *bang* de inicio, y dos *outlets* para la representación de los resultados.

Las respuestas a los problemas planteados las encontrará en el archivo "Respuesta Actividad 02.pd"

3. Generación y procesamiento de señales de audio

Objetivos

- Conocer las características principales de los objetos de audio más utilizados.
- Programar aplicaciones básicas de generación y procesamiento de sonido y música en tiempo real.

3.1. Objetos de audio

Los objetos de audio, según ya mencionamos, se diferencian visualmente de los de control no solo por su nombre, sino también por el tilde que se coloca al final del nombre del objeto (*reson~*, por ejemplo, que es un filtro).

Un objeto ampliamente utilizado es *osc~*, que implementa un oscilador, o sea, un algoritmo capaz de leer un ciclo de una cosinusoide tantas veces por segundo como le sea requerido, generando así una señal de audio periódica de una frecuencia determinada. Dicho en otros términos, el objeto *osc~* produce un señal sinusoidal (sonido puro) cuya frecuencia se establece a través de un argumento, o del *inlet* izquierdo. En la entrada de la derecha se especifica la fase inicial (un número entre 0 y 1 que corresponde a un ángulo de fase inicial comprendido entre 0 y 360º), y por la salida se envía una cantidad constante de muestras de amplitud de la señal por segundo, que equivale a la frecuencia de muestreo.

El término algoritmo, según se emplea aquí, define a un método con base en un conjunto finito de instrucciones o reglas destinadas a resolver un problema particular, que se implementan de manera ordenada y precisa a través de un programa.

La señal digital generada por el oscilador posee una amplitud igual a 1, lo que significa que el valor de las muestras varía sinusoidalmente entre -1 y 1. En PD, 1 es la máxima amplitud que una señal de audio puede tener sin que se produzca una saturación del sonido resultante. Por lo tanto, podemos solamente disminuir la amplitud del oscilador (a un valor inferior a 1) y no elevarla, si deseamos evitar la distorsión de la señal que produce.

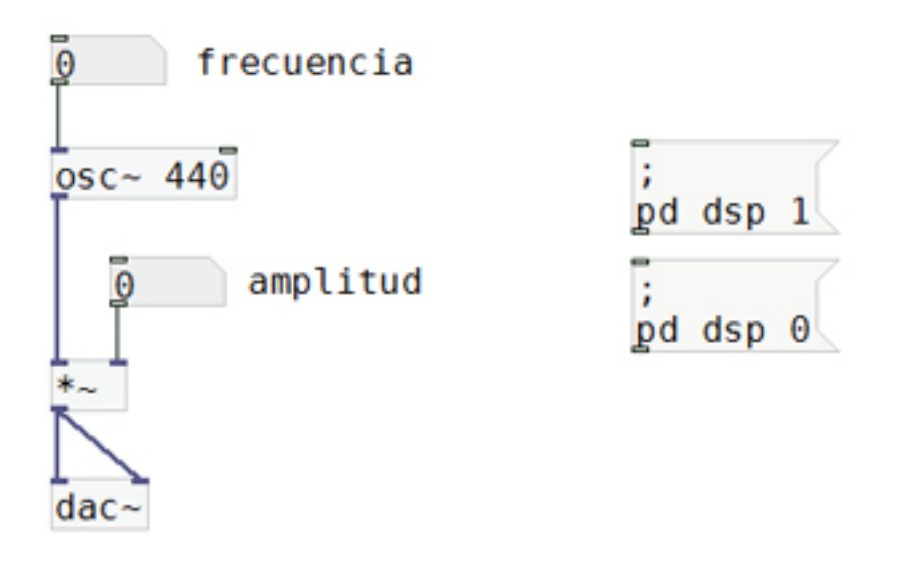
En general, para modificar la amplitud de una señal de audio, simplemente debemos multiplicar el valor de sus muestras por un número. Si el número equivale a 0.5, por ejemplo, la amplitud disminuirá a la mitad (pues multiplicar por 0.5 equivale a dividir por 2).

Por último, para enviar las muestras de la señal a la placa de sonido de nuestra computadora, podemos emplear el objeto *dac~*, cuyo nombre proviene de las iniciales de *Conversor Digital Analógico*, en inglés.

La figura siguiente muestra un programa de audio simple que consiste en un oscilador con control de amplitud. A diferencia de los *patches* anteriores, que solo empleaban objetos de control, los programas que contienen objetos de audio requieren que PD dé inicio al procesamiento de audio. Esta tarea puede lograrse de varias formas:

1) Tildando la cajita ubicada en la ventana de PD, junto a la sigla DSP.
2) Seleccionando en el menú **Media** la opción DSP encendido para iniciar el procesamiento y DSP apagado para detenerlo.
3) Accediendo a las opciones anteriores mediante los atajos Ctrl+/ y Ctrl+., respectivamente.
4) Mediante mensajes enviados a PD, de encendido y apagado, con el texto que se observa en la figura.

G.3.1. Control de amplitud de un oscilador



El *patch* "15-control amplitud.pd" contiene la programación de la figura anterior. Para encender el procesamiento de audio haga clic sobre el mensaje que se encuentra a la derecha del *patch*, y para detenerlo, sobre el mensaje de abajo. La frecuencia del oscilador se especifica como argumento (440 *Hz*), si bien puede modificarla ingresando un valor por el *inlet* izquierdo de *osc~*. También puede transformar la amplitud, y escuchar la distorsión que se produce sobre el sonido puro al superar el valor máximo 1.

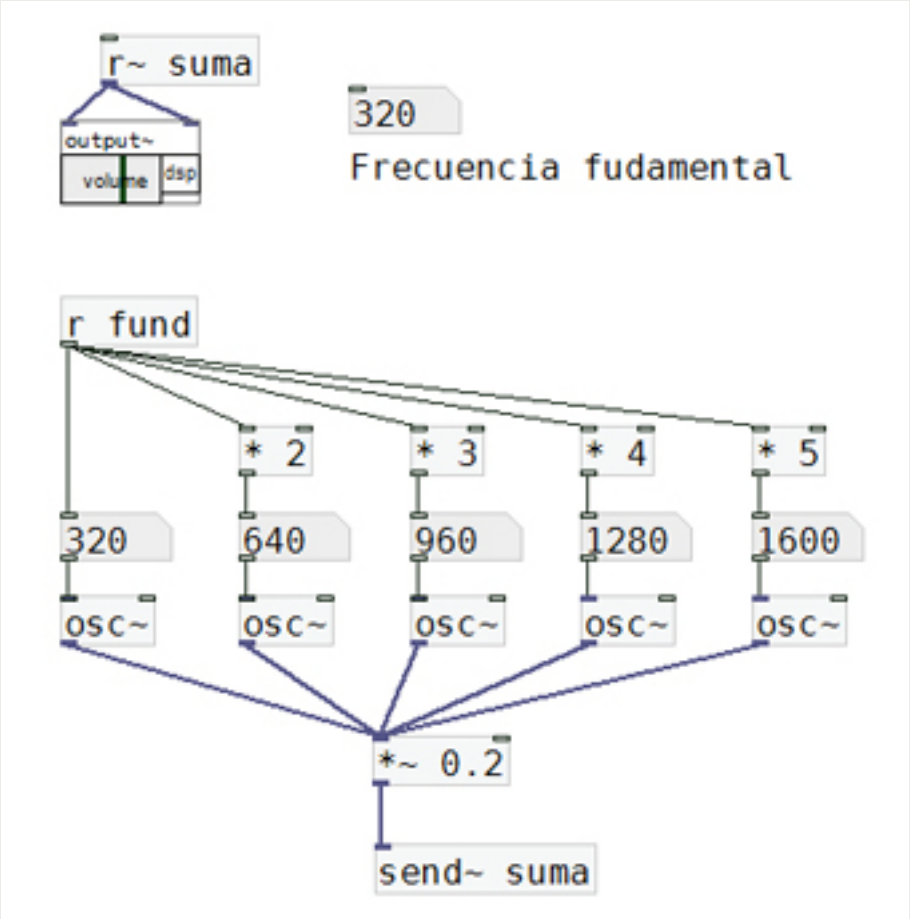
El objeto *dac~* consta inicialmente de dos *outlets*, uno para el canal izquierdo y otro para el derecho. En nuestro ejemplo, la señal de audio resultante se dirige a ambos canales. Más adelante veremos que mediante argumentos podremos aumentar la cantidad de salidas con el propósito de lograr un sistema de reproducción multicanal (cuadrafónico u octofónico, por ejemplo), en lugar del sistema estereofónico tradicional. Para ello, deberemos contar con una placa de audio multicanal que posea el número de salidas requerido en la programación.

3.2. Conexiones remotas

Al referirnos a los objetos de control vimos que era posible realizar conexiones remotas entre ellos, valiéndonos de los objetos *send* y *receive*. Esto mismo es practicable al utilizar señales de audio, pero en este caso empleamos los objetos *send~* y *receive~*, o sus versiones abreviadas *s~* y *r~*.

Para el ejemplo siguiente, que consiste en la generación de un sonido complejo mediante la adición de los cinco primeros armónicos, utilizaremos una abstracción denominada *output~*, que reemplaza al objeto *dac~*. Al operar el *patch,* veremos que resulta más práctico, dado que incluye un botón para encender el procesamiento de audio, y un control de nivel de la amplitud de salida.

G.3.2. Conexión remota de audio



La conexión entre el objeto *~ que escala la salida de los osciladores, y el objeto *output~* se realiza remotamente. Notemos también que todos los cables de salida de los osciladores se dirigen a un mismo punto, que es la entrada al multiplicador. En estos casos, cuando varias señales de audio alcanzan un mismo *inlet*, esas señales se suman. La multiplicación de las señales por 0.2 (que equivale a dividirlas por 5) evita la saturación de la resultante, si tenemos presente que la amplitud de cada componente vale 1 y que estas se suman.

En el *patch* "16-conexiones remotas de audio.pd" encontraremos la programación del ejemplo anterior. Se trata de la síntesis de un sonido complejo periódico mediante la suma de los cinco primeros armónicos de una fundamental, con amplitudes iguales a 1, aplicando conexiones remotas de control y de audio.

Cuando precisamos conectar remotamente varias salidas de audio a una única entrada, los objetos *send~* y *receive~* no resultan adecuados. Para ello, existen dos objetos especiales denominados *throw~* y *catch~*. El ejemplo siguiente, con base en el *patch* anterior, ilustra su utilización.

El *patch* "17-conexiones remotas con *throw* y catch.pd" modifica el *patch* 16, incorporando los objetos *throw~* y *catch~* para el envío de múltiples señales de audio a un mismo destino.

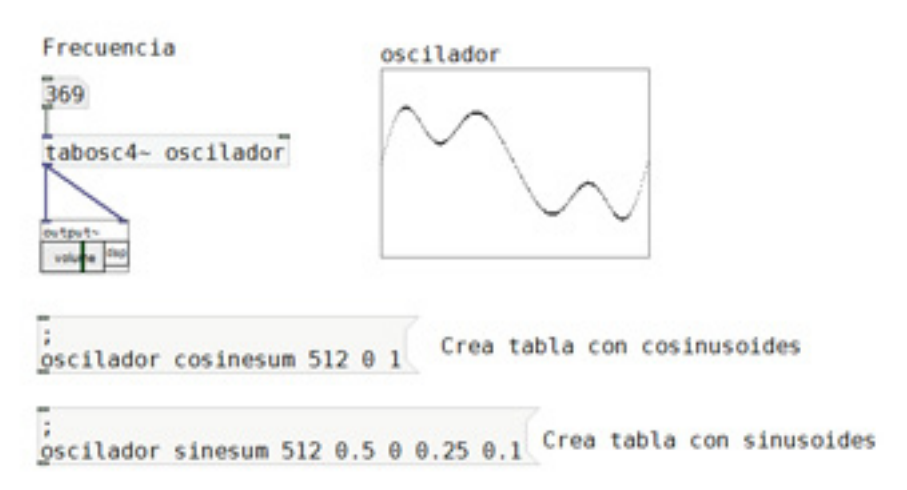
3.3. Oscilador por tabla de onda

Un oscilador por tabla de onda es un dispositivo en el cual el ciclo de la forma de onda a leer periódicamente es almacenado en una matriz de datos. Para crear una matriz en una ventana nueva de PD vamos al menú ***Poner***, y seleccionamos la opción **Matriz**.

Vemos que inmediatamente aparece un cuadro de diálogo donde establecemos sus propiedades. En el campo **Nombre** escribimos una palabra que describa su uso, por ejemplo *oscilador*. En **Tamaño** determinamos las dimensiones de la tabla, que para nuestro ejemplo deberá ser un número potencia de dos (256, 512, 1024), al cual le sumaremos tres unidades, que son destinadas a cálculos internos que PD precisa realizar (512 + 3 = 515, por ejemplo). Mientras mayor sea el tamaño, mejor definido estará el ciclo de la onda, pero también será mayor el procesamiento requerido para su empleo. Por último presionamos **OK** para crear la matriz y el gráfico donde se representan los datos que contiene. Podremos volver al menú de propiedades del objeto cuando deseemos, haciendo clic derecho sobre la tabla.

Para leer cíclicamente la tabla, a una frecuencia determinada, utilizamos el objeto *tabosc4~*, con el nombre de la tabla creada como argumento (en nuestro ejemplo, la palabra *oscilador*).

G.3.3. Oscilador por tabla de onda



En la parte inferior del *patch* de G.3.3. observamos dos mensajes destinados a nuestra tabla, denominada *oscilador*. El primero emplea el término *cosinesum* para crear una cosinusoide, o la suma de varias componentes cosinusoidales. Luego de esa palabra, colocamos el tamaño de la tabla (512 muestras), y después una sucesión de números que representa la amplitud de cada armónico, comenzando con la componente de 0 *Hz*.

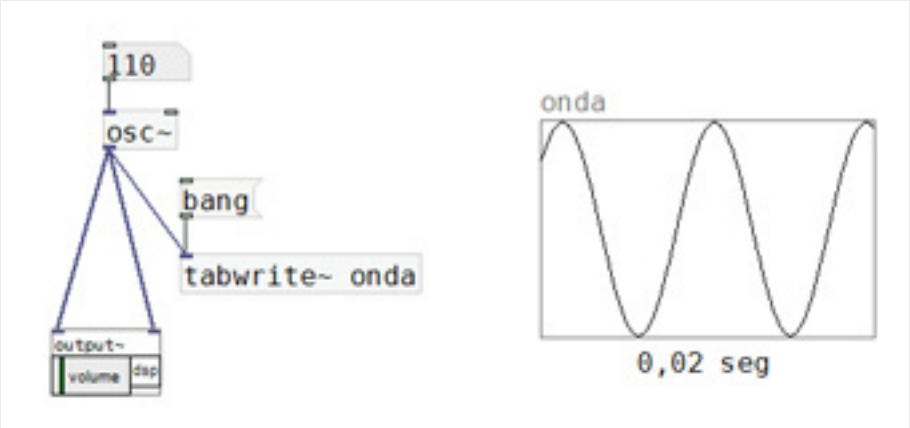
En el ejemplo fijamos una amplitud igual a cero para 0 *Hz*, y una amplitud igual a uno para el primer armónico de la onda; el resultado es un sonido puro. El segundo mensaje crea una tabla, también de 512 muestras, con componentes sinusoidales de amplitud 0.5 (primer armónico), 0 (segundo armónico), 0.25 (tercer armónico) y 0.1 (cuarto armónico).

Aquí utilizamos el término *sinesum* (suma de sinusoides, y no de cosinusoides), y que la amplitud de la componente de 0 *Hz* no se aplica en este caso.

El *patch* "18-oscilador por tabla.pd" implementa el ejemplo anterior. Según vimos, el ciclo de la señal a reproducir se determina mediante mensajes dirigidos al objeto *Matriz*. No obstante, también es posible dibujar la forma de onda, trazándola con el puntero del mouse directamente sobre el gráfico que representa la tabla.

Las tablas pueden tener diversos usos, entre ellos, pueden servir a la representación gráfica de formas de onda producidas por objetos o *patches* generadores de señales de audio. En G.3.4. observamos el gráfico de una matriz de 882 muestras, capaz de almacenar 0.02 segundos de una señal muestreada a 44 100 *Hz*. Como fuente empleamos un objeto *osc~*, cuyas muestras son almacenadas por el objeto *tabwrite~* en la tabla onda, cada vez que este recibe un *bang*.

G.3.4. Visualización de una forma de onda mediante una tabla



El *patch* "19-osciloscopio.pd" contiene la programación del ejemplo anterior. Un osciloscopio es un instrumento de medición capaz de representar gráficamente la variación de señales eléctricas en función del tiempo. De ahí el nombre dado a este ejemplo.

3.4. Envolventes dinámicas

Una envolvente dinámica es una curva que une los picos de amplitud de una forma de onda y describe la variación de la amplitud de esa señal de audio a lo largo del tiempo.

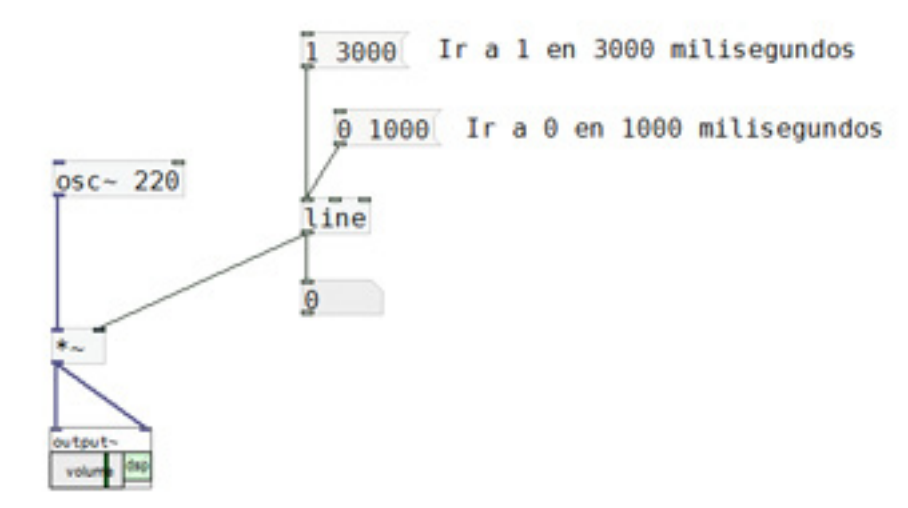
Las señales que produjimos hasta ahora son invariantes en el tiempo. Esto significa que a medida que el tiempo transcurre la amplitud de sus componentes se mantiene constante. Los sonidos producidos mecánicamente no se comportan de ese modo, sino que la amplitud de cada componente cambia con el paso del tiempo.

Si deseamos producir electrónicamente un sonido que resulte interesante a la audición, deberemos prestar especial atención a la generación de envolventes dinámicas para el sonido en su conjunto, o para cada uno de los armónicos o parciales que lo conforman.

El sonido de una campana, por ejemplo, posee un ataque rápido y fuerte, y se torna cada vez más débil hasta que desaparece. A la vez, el comportamiento de la amplitud es distinto para cada uno de sus parciales (componentes no armónicas). Su timbre comienza brillante (con muchos parciales) y se torna cada vez más pobre y opaco, dado que las componentes agudas desaparecen antes que las graves.

G.3.5. muestra un *patch* donde controlamos el ataque y la desaparición del sonido mediante envolventes. Al presionar el mensaje superior, ingresa una lista de dos números al objeto *line*, cuyos valores representan el punto de llegada y el tiempo en milisegundos para alcanzarlo, respectivamente. Este objeto calcula los valores intermedios entre el punto de partida y el de llegada, en el tiempo establecido.

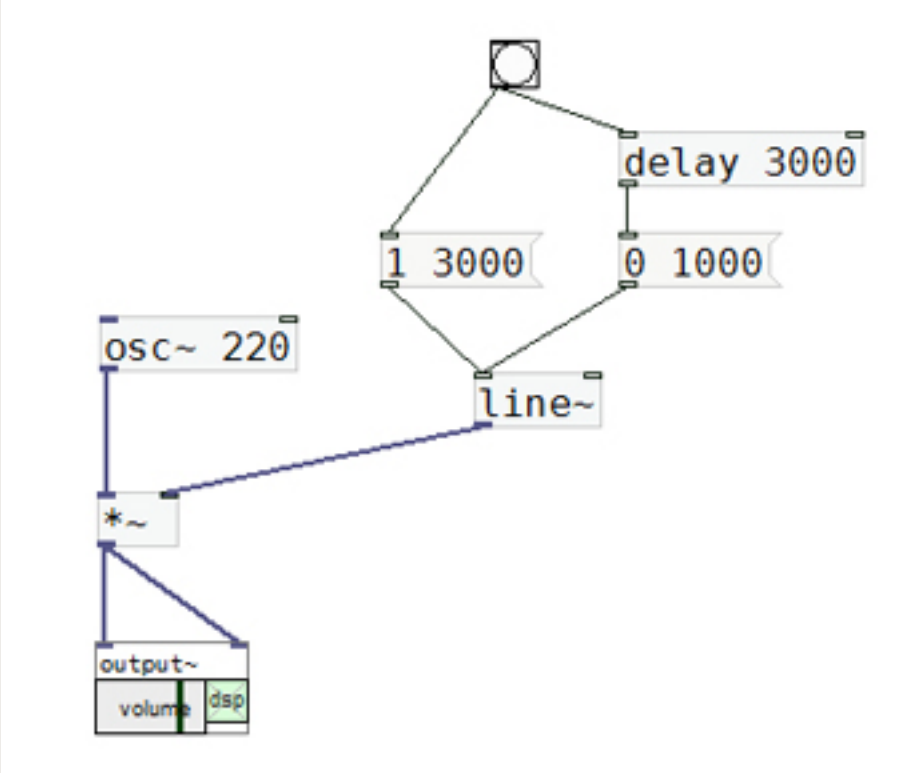
G.3.5. Envolvente dinámica realizada con el objeto *line*



Los valores producidos por *line* multiplican a la señal del oscilador, cambiándole la amplitud paulatinamente. Para volver a cero, simplemente presionamos el mensaje inferior, y la amplitud del sonido decrece linealmente durante un segundo. En el ejemplo anterior, si quisiéramos que el decaimiento de la amplitud comenzara inmediatamente después de finalizado el ataque, deberíamos presionar rápidamente el segundo mensaje. Una forma de automatizar esta operación es empleando el objeto *delay*, que retrasa la salida de un *bang* entrante, durante un tiempo especificado en milisegundos. G.3.6. lo ilustra.

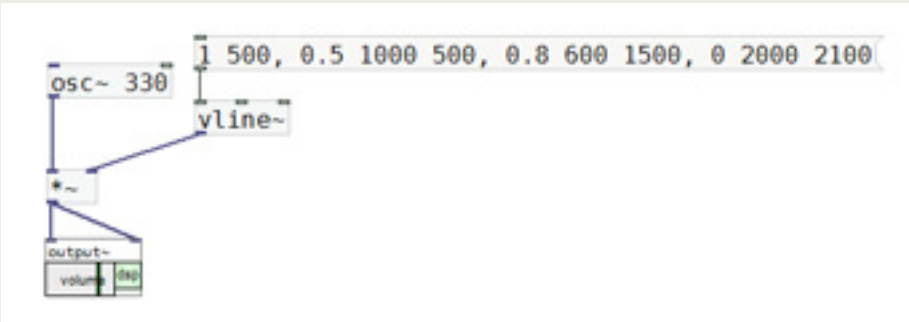
Nótese, además, que el objeto *line* es reemplazado por su versión de audio *line~*. Se trata de un objeto que funciona de igual modo, pero que produce la envolvente con una resolución mucho mayor, equivalente a la frecuencia de muestreo.

G.3.6. Ataque y decaimiento de una envolvente con line~



Según observamos en el ejemplo anterior, parece necesario crear un mensaje con un retardo inicial para cada paso de la envolvente a generar. Sin embargo, esta tarea se simplifica si recurrimos al objeto *vline~*, para el cual todos los pasos de la envolvente pueden estar definidos dentro de un único mensaje, sin que se requiera el uso de objetos de retardo.

G.3.7. Envolvente de 4 pasos con *vline~*



La sintaxis de los mensajes enviados a *vline~* consta de dos valores iniciales, seguidos por tríos de datos, separados por comas. Los dos primeros números expresan "a dónde voy" y "en cuánto tiempo", respectivamente. Luego siguen ternas de números que indican "a dónde voy", "en cuánto tiempo" y "cuánto debo esperar para realizar este paso". Este último valor cumple la misma función que el argumento del objeto *delay* visto anteriormente: esperar el tiempo que duran los pasos anteriores de la envolvente antes de disparar el paso actual (ver ejemplo en G.3.7.).

El *patch* "20-envolventes.pd" implementa los tres ejemplos recién analizados. Intente generar variantes, o diseñar distintas envolventes y llevarlas a la práctica.

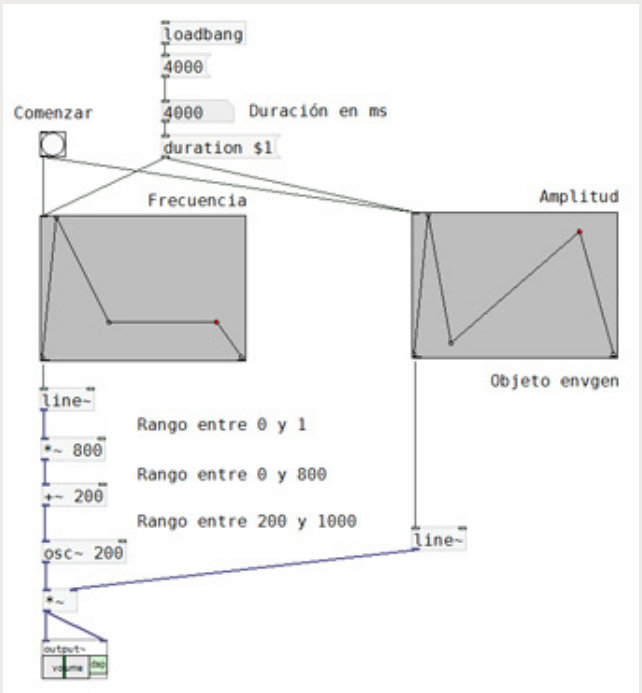
A partir de los ejemplos vistos podríamos pensar que la utilización de envolventes dinámicas en nuestros programas puede resultar tediosa, y estaríamos en lo cierto. Afortunadamente, existen objetos gráficos en las librerías externas que pueden facilitarnos esta tarea. Entre estos objetos contamos con *envgen*, de la librería *Ggee*, y *Breakpoints*, de la librería *Tof*. Para utilizar el objeto *envgen* creamos un objeto vacío y escribimos el nombre del objeto. A continuación podemos agregar una serie de argumentos, que establecen sus dimensiones, los valores máximos de los ejes *x* e *y*, y las etiquetas para recibir y enviar datos de forma remota. Si no declaramos ningún argumento, el objeto se construirá con un valor máximo de *x* de 1000 milisegundos, y un rango de *y* comprendido entre 0 y 1. La sintaxis es la siguiente:

envgen tamañoX tamañoY Xmáx Ymáx símbolo_enviar símbolo_recibir

Una vez creado, conectamos el *outlet* izquierdo del objeto a un objeto *line~*, que recibirá los datos de la envolvente gráfica y realizará las rampas necesarias para producir la envolvente.

En el ejemplo siguiente observamos dos envolventes gráficas, una destinada a controlar la amplitud de un oscilador (rango entre 0 y 1), y otra empleada para controlar su frecuencia (rango entre 0 y 1, pero escalado luego entre 200 y 1000).

G.3.8. Envolventes gráficas de frecuencia y amplitud



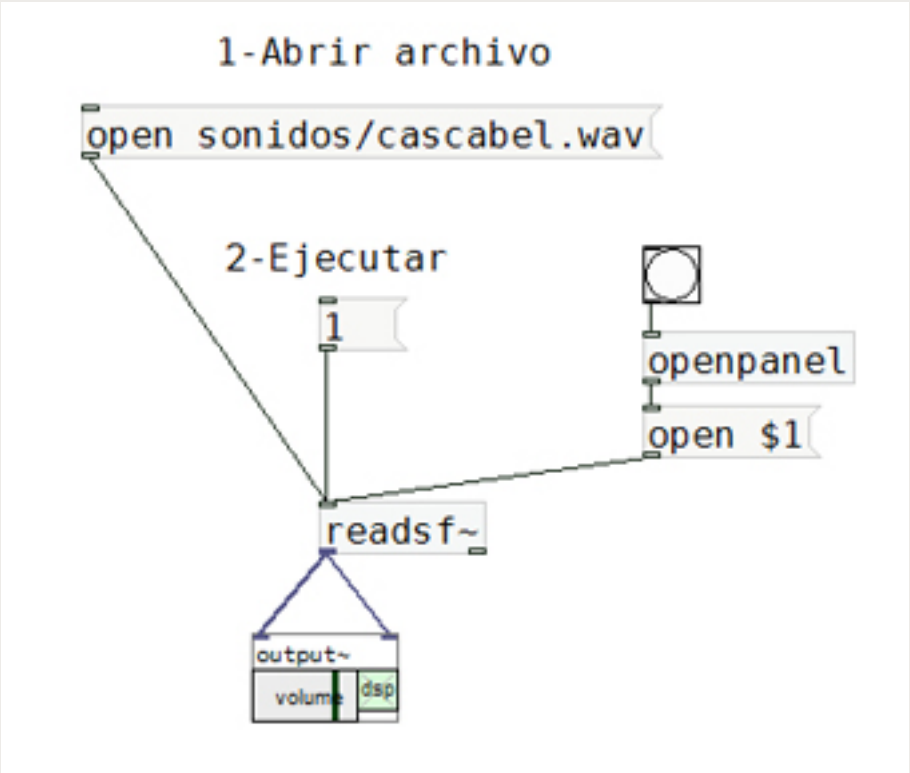
Nótese que la duración (valor máximo del eje x) se establece mediante un mensaje con la palabra clave *duration* y el tiempo en milisegundos. Sin embargo, el tiempo no está establecido como una constante dentro del mensaje, sino como una variable, que toma el valor del objeto *Number* conectado por encima. En PD, según vemos, los nombres de variables están formados por el signo "$" seguido de un número. En el ejemplo inicialmente la variable "$1" adopta el valor 4000.

El *patch* "21-envolventes gráficas.pd" incluye la programación de G.3.8. Aquí puede practicar fácilmente el diseño de envolventes de amplitud y de frecuencia, y percibir inmediatamente los resultados.

3.5. Lectura de un archivo de audio

Para ejecutar un archivo de sonido, almacenado en el disco de nuestra computadora, podemos utilizar el objeto *readsf~*. Este objeto recibe un mensaje *open* con el nombre del archivo y la ruta donde se encuentra, y al recibir los mensajes 1 o 0 comienza la ejecución o la detiene. A fin de hallar un archivo utilizando un explorador, podemos también utilizar el objeto *openpanel*, como se aprecia en la figura siguiente.

G.3.9. Lectura de archivo de audio



No obstante, si programamos el ejemplo anterior, notaremos que luego de ejecutarlo es preciso cargar el archivo nuevamente antes de poder volver a reproducirlo. Este problema es fácilmente subsanable cuando usamos el mensaje *open* seguido de la ruta y nombre del archivo (sin *openpanel*), pues podríamos emplear un objeto *trigger* para enviar primero el mensaje *open* y luego el "1" que da inicio a la ejecución.

El *patch* "22-archivo de sonido.pd" implementa la programación de G.3.9., y remedia la dificultad antes mencionada.

Sin embargo, al utilizar *openpanel* la dificultad persiste. Por esta razón, hemos programado una abstracción, denominada *sfplay~*, que nos permitirá buscar un archivo en un directorio, ejecutarlo, e incluso generar un sinfín (*loop*) para que se reproduzca cíclicamente.

El *patch* "23-archivo de sonido con abstracción.pd" muestra el uso de la abstracción *sfplay~*, que hemos creado para facilitar la operación de lectura de archivos de sonido.

Otro modo de disponer de la información de un archivo de sonido para su manipulación es a través del objeto *soundfiler*. Este objeto recibe un mensaje read con la ruta y nombre del archivo, y con el nombre de la tabla donde va a alojarlo (en nuestro ejemplo de G.3.10. esa matriz se llama "archivo").

G.3.10. Lectura de un archivo en tabla

La lectura de la tabla la realizamos con el objeto *tabread4~*, que recibe números de muestra de la tabla y devuelve los valores de amplitud de la onda almacenada. El sonido del archivo "raspado.wav" dura 2 segundos, o sea que ocupa 88 200 muestras, a una frecuencia de muestreo de 44 100 *Hz*. Mediante *line~* barremos con una recta desde la muestra 0 hasta la 88 200 en 2 segundos (hacia adelante), o bien desde la 88 200 hasta la 0 (hacia atrás). A través del *slider*, con valores entre 0 y 88 200, especificados en sus propiedades, podemos recorrer manualmente el archivo a distintas velocidades.

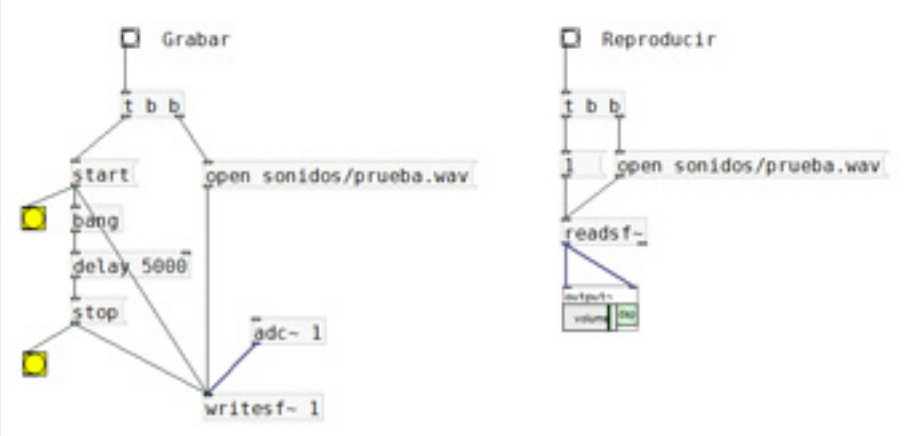
El *patch* "24-soundfiler.pd" contiene la programación de G.3.10. Nótese el uso de la variable $1 dentro del último de los mensajes que preceden al objeto *line~*.

3.6. Grabación de un archivo de audio

Para grabar una señal de audio en un archivo utilizamos el objeto *writesf~*. Este objeto acepta como argumento la cantidad de canales de audio que se va a registrar en el archivo (1 = mono, 2 = estéreo, 4 = cuadrafónico, etc.). A través de un mensaje *open* informamos al objeto cómo se llama el archivo que vamos a crear o sobrescribir con datos de audio. Luego, con *start* comenzamos a grabar, y con stop detenemos la grabación.

En G.3.11. se observa un dispositivo para grabar utilizando el micrófono de la computadora, y reproducir luego el archivo registrado. La señal del micrófono la capturamos mediante el objeto *adc~* (*analog to digital converter*). Obsérvese en el ejemplo que una vez que presionamos el botón *bang* para comenzar a grabar, ponemos en acción un *delay* de 5 segundos que detiene automáticamente la grabación.

G.3.11. Grabación y reproducción de un archivo de audio



El *patch* "25-grabar audio.pd" contiene la programación de G.3.11.

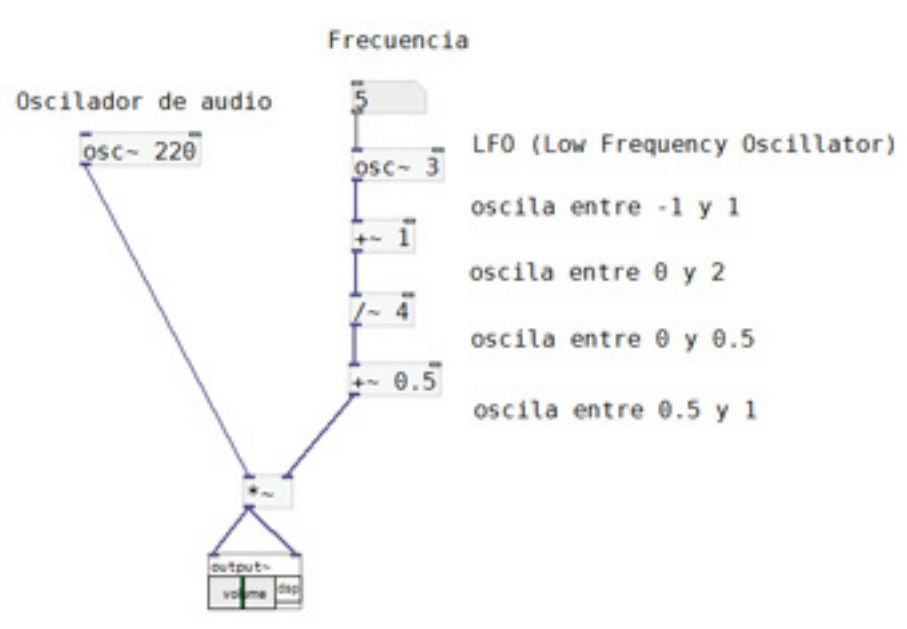
3.7. Osciladores de baja frecuencia

Un oscilador de baja frecuencia es un dispositivo empleado para controlar procesos, en general periódicos, destinados a transformar las cualidades acústicas de una señal de audio, como su frecuencia, su amplitud, o la riqueza de su espectro.

La salida de un oscilador de baja frecuencia, que produce menos de 10 o 20 ciclos por segundo, no es enviada en ningún caso a los parlantes, pues por su frecuencia no puede convertirse en sensación sonora. Asimismo, puede tener una amplitud que supere ampliamente al valor 1, exigido por PD para que la señal no se distorsione, pues su función no es "sonar", sino brindar datos a un proceso determinado.

G.3.12. muestra un oscilador de audio que produce una sinusoide de 220 *Hz*, cuya amplitud es modulada (transformada) mediante la salida de un oscilador con baja frecuencia (3 *Hz*, inicialmente).

G.3.12. Modulación en amplitud, a baja frecuencia

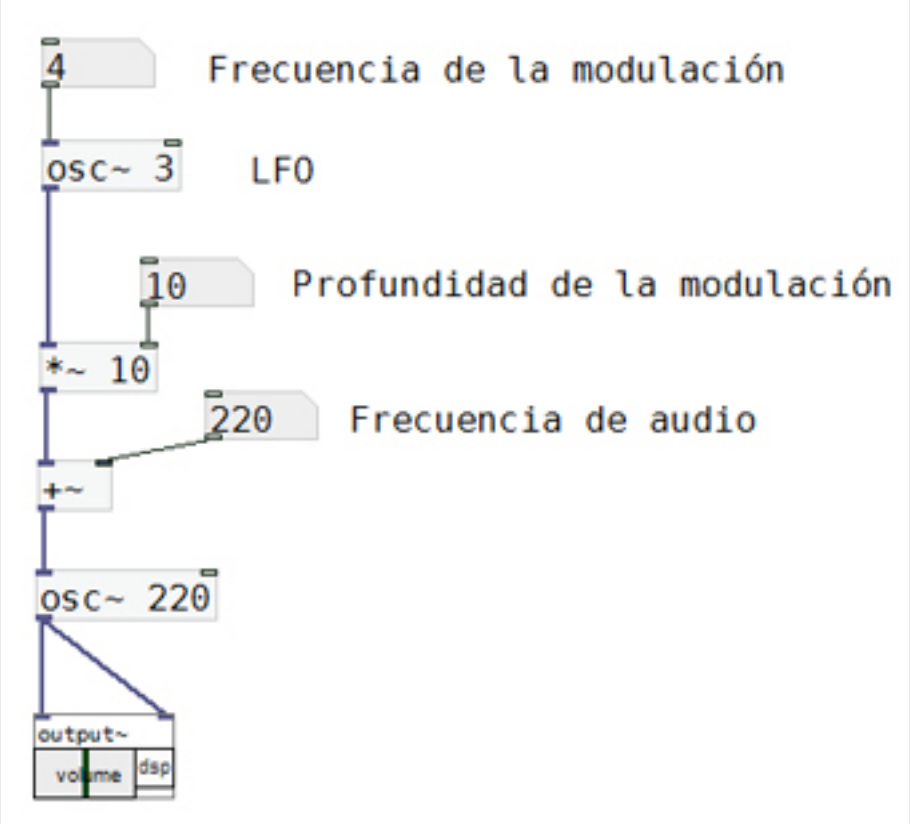


En G.3.12. vemos que, mediante operaciones aritméticas, limitamos el rango de oscilación entre 0.5 y 1, con el propósito de utilizar esos datos en el control de amplitud del oscilador de audio de la izquierda. A este proceso se lo denomina modulación de amplitud en baja frecuencia, y al efecto sonoro que produce se lo conoce en la música electrónica como "trémolo".

El *patch* "26-modulación en amplitud.pd" contiene la programación de G.3.12.

Del mismo modo, podemos transformar cíclicamente la frecuencia de un oscilador de audio. El efecto es muy similar al que se emplea en el canto, o en los instrumentos de cuerda, con el nombre "vibrato". G.3.13. ilustra este proceso.

G.3.13. Modulación en frecuencia, a baja frecuencia



En este caso, el valor de amplitud de las muestras positivas y negativas que salen del oscilador de baja frecuencia es sumado a la frecuencia del oscilador de audio (220 *Hz* en nuestro ejemplo), desviando la altura del sonido por encima y por debajo de la altura normal. A mayor amplitud del oscilador de baja frecuencia, mayor desviación de la frecuencia del oscilador de audio.

El *patch* "27-modulación en frecuencia.pd" contiene la programación de G.3.13. Aquí podremos comprobar los resultados que surgen de modificar la frecuencia del oscilador modulador y su amplitud (profundidad de la modulación).

3.8. Modulación en anillo

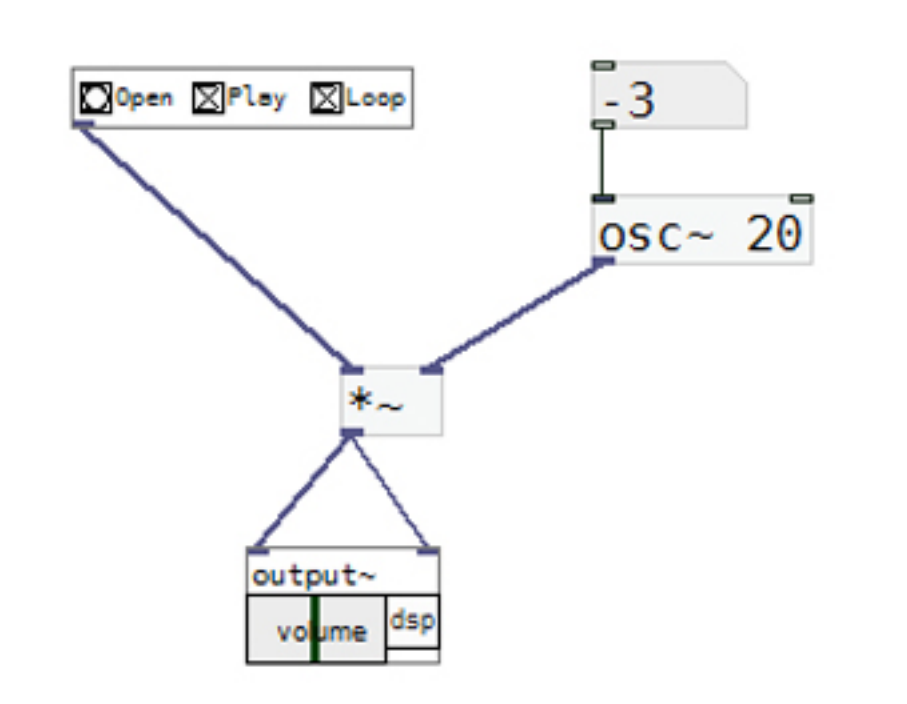
Modulación en anillo es el nombre que recibe un proceso ampliamente utilizado en los antiguos sintetizadores de sonido analógicos. Actualmente se emplea un efecto similar en los medios digitales e informalmente suele conservarse el viejo nombre para denominarlo, si bien se considera correcto llamarlo "convolución espectral".

Mediante este proceso entre dos señales se produce la suma y la resta de cada componente del espectro de la primera señal con cada componente del espectro de la segunda señal. Supongamos que la primera tiene tres armónicos de 100, 200 y 300 *Hz* respectivamente, y que la segunda es un sonido puro de 20 *Hz*. La señal resultante de la modulación posee un espectro cuyos parciales tienen 80, 120, 280, 220, 280 y 320 *Hz* respectivamente. Según se observa, ninguna de las componentes participantes forma parte del resultado, sino solo la suma y la resta de las componentes de ambas señales, generándose una onda compleja inarmónica (las componentes no son múltiplos de la de menor frecuencia).

La incorporación de la modulación en anillo en los sintetizadores analógicos permitió la generación de señales inarmónicas, o sea, sonidos de altura indefinida, partiendo de osciladores que solo producían señales armónicas (sinusoides, onda cuadrada, diente de sierra). Así, se tornaba posible imitar el sonido de instrumentos de percusión (campanas, cencerros, etcétera).

En los medios digitales la modulación en anillo se realiza multiplicando dos formas de onda. En G.3.14. observamos una modulación en anillo entre un sonido almacenado en un archivo de audio y una sinusoide. La frecuencia de la sinusoide se suma y se resta a la frecuencia de cada componente del sonido complejo del archivo, produciéndose un corrimiento de sus parciales, con la consiguiente deformación del timbre.

G.3.14. Modulación en anillo



El *patch* "28-modulación en anillo.pd" contiene la programación de G.3.14. Abra archivos de audio con fragmentos musicales vocales o instrumentales, y experimente la audición de la modulación cambiando la frecuencia del oscilador.

Cetta, P. (2005), "Procesamiento en tiempo real de sonido e imagen con PD-GEM", en: *Revista de Investigación Multimedia* Nº 1. ATAM-IUNA, Buenos Aires, pp. 28-34.

Actividad 1

La librería "zexy" de PD-extended cuenta con un objeto llamado *multiplex~*, el cual permite ingresar varias señales de audio y seleccionar cuál de ellas se dirige a su *outlet*.

Realice un *patch* donde genere una señal sinusoidal, una señal compleja (sumando señales sinusoidales de distinta frecuencia) y el resultado de una modulación en anillo, y donde pueda escuchar cada una de ellas a voluntad (utilizando *multiplex~*).

En el archivo "Respuesta Actividad 03.pd" encontrará una posible solución de este ejercicio. Compare los resultados que obtuvo con los del archivo y anote las diferencias.

Actividad 2

a. Programe un *patch* donde la señal proveniente de un archivo de sonido sea modulada en amplitud por una señal compleja. Genere la señal moduladora compleja sumando tres señales sinusoidales de distinta frecuencia.

b. En el mismo *patch*, agregue un oscilador modulado en frecuencia cuya amplitud esté controlada por una envolvente gráfica.

c. En el archivo "Respuesta Actividad 04.pd" encontrará una posible solución de este ejercicio. Compare los resultados que obtuvo con los del archivo y anote las diferencias.

4. Técnicas de síntesis del sonido

Objetivos

- Conocer las principales técnicas de síntesis de sonido.
- Aplicar los conocimientos teóricos aprendidos en la realización de programas de síntesis sonora.

4.1. Introducción a la síntesis sonora

Entendemos por *síntesis* ir de lo simple a lo complejo, y en nuestro caso el término se aplica perfectamente, pues mediante la combinación de elementos simples (sonidos puros, por ejemplo) podremos arribar a resultantes de un alto grado de complejidad.

Las técnicas de síntesis suelen ser agrupadas en dos categorías: las técnicas lineales y las no lineales. Básicamente, las primeras son la síntesis aditiva y la sustractiva, mientras que el resto de las técnicas pertenecen a la categoría no lineal. Una técnica se define como lineal si la cantidad de elementos simples que se ponen en juego está en relación lineal con los resultados que se obtienen. En el caso de la síntesis aditiva resulta obvio, dado que por cada nueva componente que deseo sumarle a un sonido complejo debo agregar un nuevo oscilador al proceso de síntesis. En las técnicas no lineales esta relación no se mantiene, pues con pocos elementos simples, pero conectados de otras formas, puedo obtener resultados muy complejos, como en el caso de la síntesis por frecuencia modulada, que luego trataremos.

4.2. Síntesis aditiva

La síntesis aditiva consiste en la suma de componentes sinusoidales de frecuencias y amplitudes determinadas. A través de ella pretendemos generar sonidos complejos que imiten a los instrumentos acústicos, o bien, creen nuevos instrumentos.

Esta técnica se fundamenta en el teorema de Fourier que, llevado al campo que nos ocupa, expresa que toda forma de onda compleja periódica puede descomponerse en la suma de formas de onda sinusoidales de frecuencias, amplitudes y fases adecuadas, a las que se denomina *armónicos*.

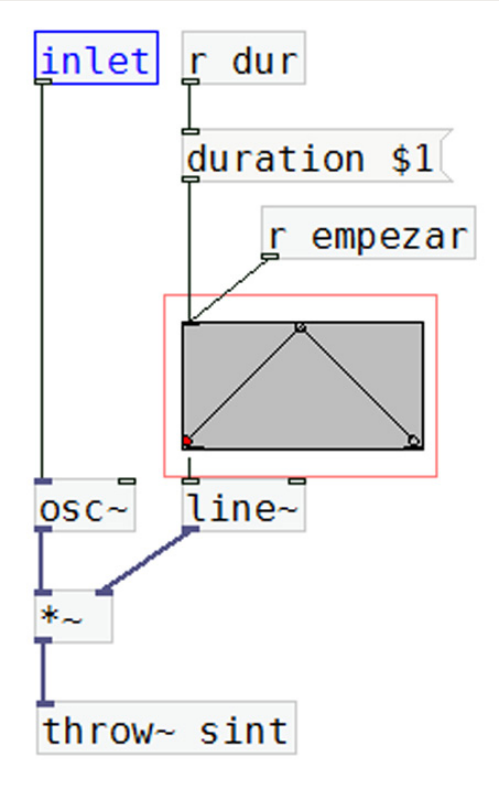
Jean-Baptiste Joseph Fourier (1768 -1830) fue un matemático y físico francés, autor de la Transformada que lleva su nombre, cuya aplicación en el campo del procesamiento de señales nos permite hallar el espectro asociado a una forma de onda.

Actualmente, sabemos que los sonidos tónicos (aquellos que definen claramente una altura, como el piano, por ejemplo) poseen componentes armónicas, es decir, están compuestos por sonidos puros cuyas frecuencias son múltiplos de la frecuencia más baja, denominada fundamental. Y que los sonidos no tónicos (aquellos que no definen una altura, como algunas campanas, o un platillo) poseen componentes no armónicas (no son múltiplos de una fundamental) a las que se denomina parciales.

Mediante la síntesis aditiva podemos crear sonidos tónicos o no tónicos, ya sea a través de la suma de armónicos o de parciales, respectivamente. Cada componente es producida por un oscilador, y su amplitud es afectada por una envolvente dinámica que describe el comportamiento de esa componente en el tiempo.

G.4.1 muestra un *subpatch* destinado a producir una componente de la síntesis. Observamos un *inlet* por donde ingresa el valor de frecuencia, el oscilador, y una envolvente dinámica que recibe la duración remotamente, al igual que el bang para disparar esa envolvente. La señal de audio generada es enviada remotamente a la salida a través del objeto *throw~*.

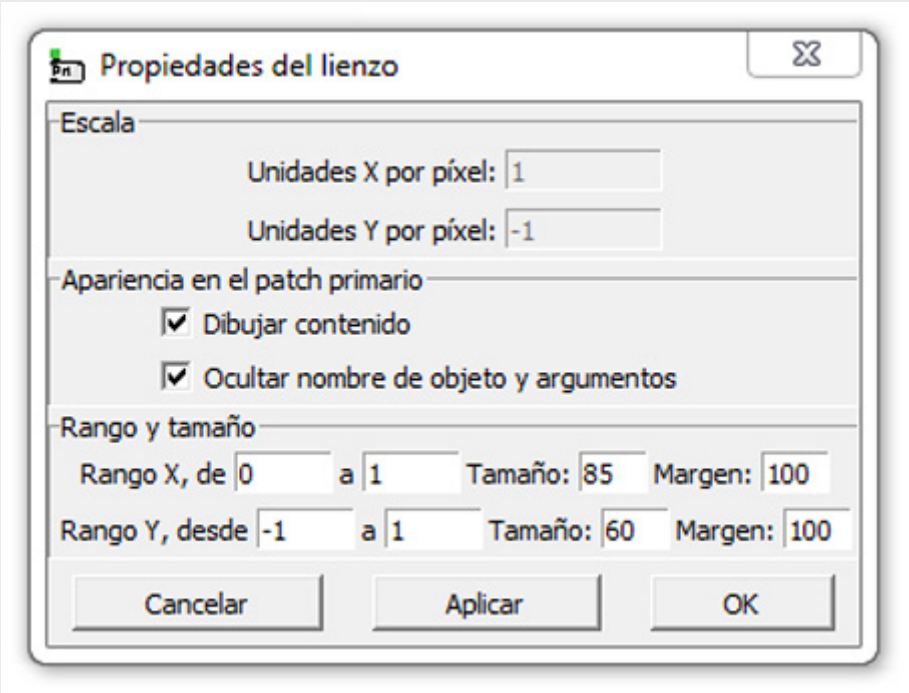
G.4.1. Oscilador en un *subpatch* que dibuja contenido



Encapsulamos la programación en un objeto al que denominamos *pd oscil*. Hasta aquí todo nos resulta conocido, excepto que el *subpatch* creado tiene la capacidad de mostrar el objeto gráfico que ubicamos en su interior (el objeto *envgen*, que utilizamos para dibujar envolventes).

Para crear un *subpatch*, o incluso una abstracción, que muestre los objetos GUI ubicados en su interior, hacemos clic con el botón derecho del mouse en el fondo de la ventana de ese *subpatch* y al desplegarse el menú elegimos la opción **Propiedades**. Cuando se abre el cuadro de diálogo, tildamos la opción **Dibujar contenido**. Al presionar el botón **Aplicar** veremos que en la ventana aparece un rectángulo rojo, que podemos trasladar o redimensionar cambiando los números de **Tamaño** y **Margen**, respectivamente (ver G.4.2.).

G.4.2. Propiedades del fondo de ventana

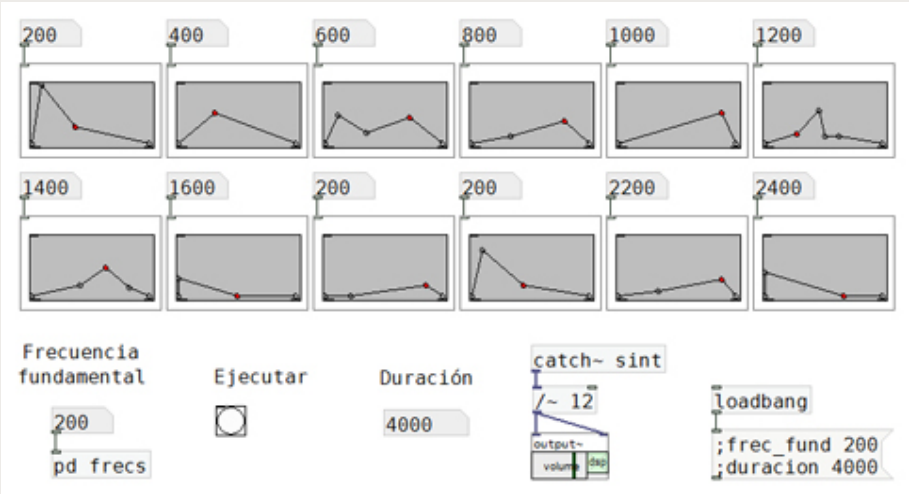


Para abrir un *subpatch* creado de este modo, hacemos clic derecho sobre él y, al desplegarse el menú, elegimos la opción **Abrir**.

Vamos a utilizar el *subpatch pd oscil* en un programa que sintetice sonidos complejos a partir de sonidos puros. G.4.3. muestra el resultado de la programación, donde observamos 12 instancias del objeto, cada una con una frecuencia distinta.

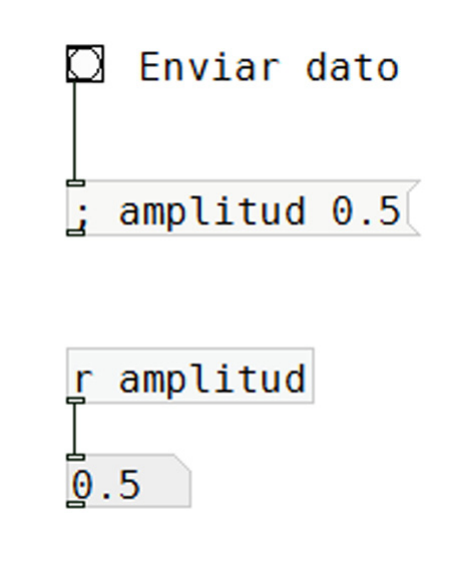
Al comenzar, las frecuencias se corresponden con los 12 primeros armónicos de una frecuencia fundamental preestablecida (200 *Hz*). Pero es posible cambiar, tanto la fundamental desde el objeto *Number* asociado, como transformar la frecuencia de cada componente individualmente, con la intención de convertir en inarmónico el sonido a sintetizar.

G.4.3. Programa de síntesis aditiva



En la parte inferior derecha del *patch* puede verse un mensaje que se envía automáticamente, cuando se abre el archivo, a través del objeto *loadbang*. Este mensaje tiene nombres de conexiones remotas, precedidos por un punto y coma, y seguidos de valores numéricos. Este tipo de mensajes permite enviar valores a objetos *receive* que comparten el mismo nombre. Por ejemplo, si escribimos en un mensaje la expresión "; amplitud 0.5", el número 0.5 llegará a cualquier objeto *receive* que tenga como argumento la palabra "amplitud" (ver G.4.4.).

G.4.4. Envío remoto mediante mensaje



El *patch* de G.4.3 se encuentra en el archivo "29-Síntesis aditiva.pd". Abra el archivo y analice en el menú de **Propiedades** de cada objeto *Number* y del botón *bang*, los nombres de las conexiones remotas utilizadas. Modifique también las envolventes y las frecuencias de cada componente, y escuche atentamente los resultados.

4.3. Síntesis sustractiva

La síntesis sustractiva es la técnica que utilizaban la mayoría de los sintetizadores analógicos. Se basa en el tratamiento de señales ricas en componentes (onda diente de sierra, onda cuadrada o incluso ruido blanco), a las cuales se modela mediante el uso de filtros.

Los filtros poseen la capacidad de retener ciertas partes del espectro de un sonido y dejan pasar otras. En la Unidad 5 nos detendremos especialmente en su estudio.

4.4. Síntesis por frecuencia modulada

En la Unidad 3 tratamos la utilización de osciladores de baja frecuencia para el control cíclico de la amplitud o la frecuencia de señales de audio.

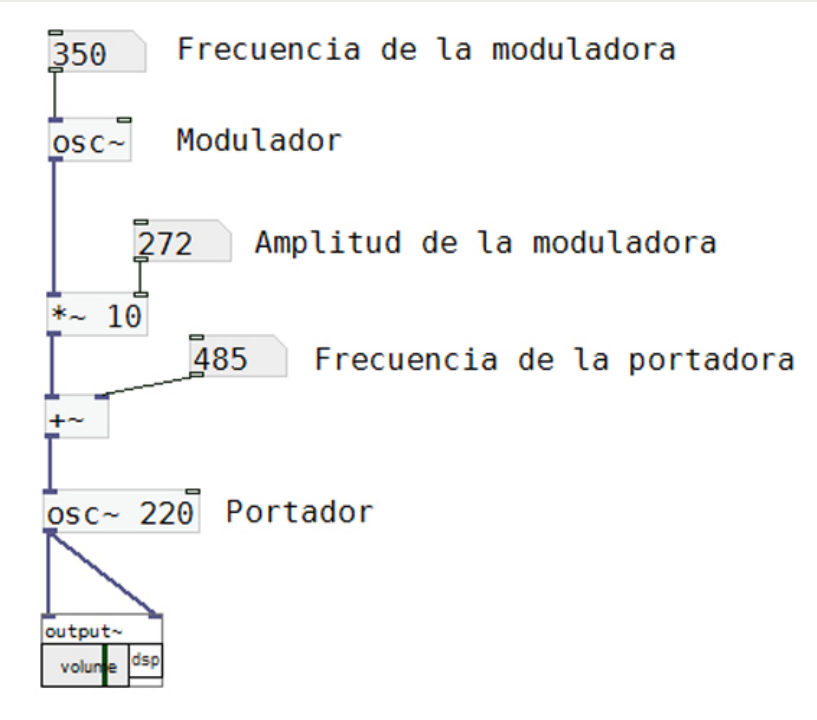
La técnica de síntesis por frecuencia modulada emplea una configuración similar a la de la modulación en baja frecuencia vista anteriormente, pero la diferencia principal reside en que al aumentar la frecuencia de la señal moduladora por encima de 20 *Hz*, comienza a deformarse la señal de la resultante de la modulación, dando lugar a la producción de sonidos complejos.

A la modulación en baja frecuencia la relacionamos con el efecto instrumental denominado *vibrato*, en el cual, a la altura producida por el instrumento o la voz, se le suma una desviación periódica. En el caso de un violín, por ejemplo, el efecto se realiza acortando y alargando ligeramente la longitud de la cuerda pisada mediante una oscilación del dedo. Suponiendo que el violinista pudiera desplazar el dedo con el que pisa la cuerda unas 100 veces por segundo, y a una amplitud considerable, el sonido del violín se tornaría muchísimo más complejo, pues ese rápido cambio de frecuencia causaría una deformación de la onda que naturalmente produce la cuerda.

La técnica de síntesis por frecuencia modulada fue ideada por John Chowning, y publicada por primera vez en 1973. Posteriormente, fue implementada en el más exitoso de los sintetizadores de sonido comerciales, el *Yamaha DX7*.

La figura siguiente muestra un *patch* de FM simple, que consta de dos osciladores sinusoidales. El primero de ellos genera la señal moduladora, mientras que el segundo genera la señal que recibe la modulación, denominada portadora. La acción de estos osciladores, si se tratara de una modulación en baja frecuencia, podría compararse con el movimiento que realiza el dedo de la mano izquierda del violinista al producir el *vibrato* (moduladora), y con la oscilación propia que caracteriza a la cuerda al ser frotada por el arco (portadora). No obstante, para que se generen sonidos complejos a partir de señales sinusoidales, es preciso que tanto la frecuencia como la amplitud de la moduladora superen ampliamente los valores de un *vibrato*.

G.4.5. FM simple



La programación que se observa en G.4.5. se encuentra en el *patch* "30-FM simple.pd". Experimente los sonidos que se producen al cambiar la frecuencia de la moduladora, la frecuencia de la portadora y la amplitud de la moduladora.

A partir de una exploración intuitiva del *patch* "30-FM simple.pd" observamos que el timbre que resulta depende de la modificación de los tres parámetros analizados: la frecuencia de la moduladora, la frecuencia de la portadora y la amplitud de la moduladora. De los dos primeros interesa particularmente la relación que existe entre ellos, o sea, el cociente f_p / f_m. También observamos que al aumentar la amplitud de la moduladora, aumenta la cantidad de componentes del sonido (armónicos o parciales, según sea el caso). Cuando esa amplitud vale cero solo escuchamos la señal sinusoidal producida por el oscilador portador a la frecuencia de la portadora, pero si aumentamos la amplitud gradualmente, el timbre se torna cada vez más rico.

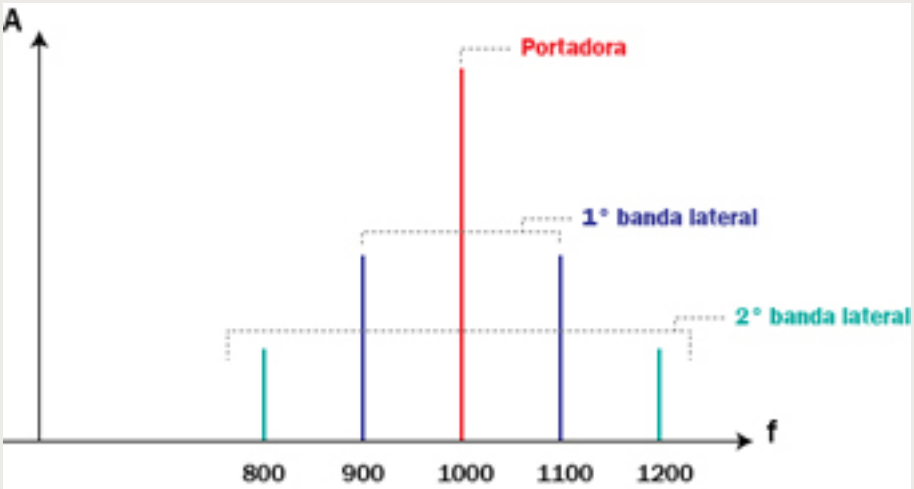
A través de la relación entre la frecuencia de la portadora y de la moduladora podemos predecir cuáles serán las componentes presentes en el espectro. Las componentes se producen simétricamente de a pares alrededor de la frecuencia de la portadora, y a cada par de componentes la denominamos "banda lateral". La cantidad de bandas laterales depende de la amplitud de la moduladora mientras mayor es la amplitud, mayor es la cantidad de bandas que aparecen.

La siguiente expresión permite calcular las frecuencias de cada banda lateral; si *k* es igual a 1 nos referimos a la primera banda lateral, si *k* es igual a 2 a la segunda, y así sucesivamente:

$$f_k = f_n \pm k.f_m$$

Suponiendo que la frecuencia de la portadora es 1000 *Hz*, y que la frecuencia de la moduladora es 100 *Hz*, las componentes aparecerán, siguiendo la ecuación anterior, a 900 y 1100 *Hz* (primera banda lateral, con $k = 1$), 800 y 1200 *Hz* (segunda banda lateral, con $k = 2$), etc. El espectro quedaría conformado de la siguiente manera:

G.4.6. Espectro producido por FM



Supongamos ahora que f_p = 100 *Hz* y f_m = 100 *Hz*. Las bandas laterales aparecerán a ambos lados de la portadora con 0 y 200 *Hz* (*k* = 1), -100 y 300 *Hz* (*k* = 2), -200 y 400 *Hz* (*k* = 3), etc. En este caso, las frecuencias negativas se rebaten sobre el lado positivo del espectro, quedando 100, 200, 300 y 400 *Hz*.

Pero si f_p = 1000 *Hz* y f_m = 1000 *Hz*, nos queda un espectro con 1000, 2000, 3000 y 4000 *Hz*, que es similar al anterior pues tiene armónicos consecutivos. Siguiendo este razonamiento, vemos que lo que interesa particularmente es la relación f_p / f_m, y cuáles armónicos o parciales se hacen presentes. En el ejemplo anterior la relación en ambos casos es 1/1, lo cual genera armónicos de una fundamental.

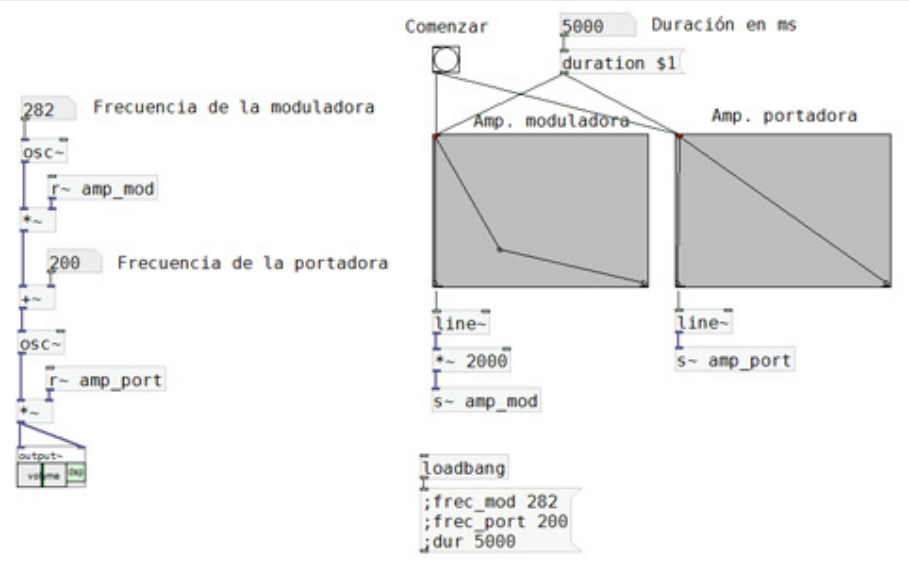
Veamos otro ejemplo. Si f_p = 100 *Hz* y f_m = 200 *Hz*, aplicando la fórmula vista obtenemos -100 y 300 *Hz* (*k* = 1), -300 y 500 *Hz* (*k* = 2), -500 y 700 *Hz* (*k* = 3), etc. Si rebatimos las frecuencia positivas sobre el eje positivo de frecuencias, nos queda 100, 300, 500 y 700 *Hz*. Observamos, para este caso, que la relación f_p / f_m = 1/2 genera los armónicos impares.

Por último, si f_p = 100 *Hz* y f_m = 141 *Hz* obtenemos -41 y 241 *Hz* (*k* = 1), ---182 y 382 *Hz* (*k* = 2), -323 y 523 *Hz* (*k* = 3), etc., que sobre el eje positivo dan 41, 100, 182, 241, 323, 382 y 523 *Hz*. Obtuvimos, entonces, un espectro con parciales, dado que las frecuencias no son múltiplos de la más baja o fundamental, y por lo tanto el sonido sintetizado no resulta tónico.

En la técnica de síntesis por frecuencia modulada, la frecuencia de las componentes del espectro resultante depende de la relación entre la frecuencia portadora y la moduladora. La relación f_p / f_m determina, además, si el espectro posee armónicos o parciales, y en consecuencia, si el sonido generado es tónico o no. Por otra parte, la riqueza del espectro está en relación directa con la amplitud de la moduladora: a mayor amplitud, mayor cantidad de bandas laterales se hacen presentes.

A fin de lograr resultados más interesantes podemos utilizar envolventes para controlar la amplitud de la moduladora y la amplitud de la resultante. La primera de ellas incidirá sobre la riqueza del timbre, mientras que la segunda actuará como una envolvente dinámica, dando forma al sonido. Obsérvese en G.4.7. el valor máximo dado a la amplitud de la moduladora a través de la multiplicación a la salida del objeto *line~*. Una amplitud igual a 2000 es impensable para un oscilador de audio, pero en este caso, la salida de la moduladora se utiliza únicamente para modificar la frecuencia de la portadora, y no para ser escuchada.

G.4.7. FM con envolventes



La programación de G.4.7. se encuentra en el *patch* "31-FM simple con envolventes.pd".

Resulta notable la complejidad de los sonidos que pueden lograrse con la frecuencia modulada, sobre todo si consideramos que solamente intervienen dos osciladores sinusoidales en el proceso de síntesis. Pero no terminan aquí las posibilidades de la técnica, ya que el algoritmo de la FM simple puede combinarse de diversas formas para generar sonidos aún más complejos.

Podemos, por ejemplo, utilizar dos o más FM simples distintas y sumar sus resultados, o bien lograr que el resultado de una FM simple actúe como señal moduladora de una nueva portadora, o sea, que una portadora sea modulada mediante una señal compleja. A este último tipo de modulación se lo denomina "en cascada", por la forma en que se disponen los osciladores en el *patch*.

4.5. Síntesis granular

Esta técnica se basa en la combinación de eventos sonoros de muy corta duración (menos de 50 milisegundos), denominados "granos", que en conjunto son percibidos como sonidos complejos.

El concepto de grano sonoro proviene de la idea de *quantum* de sonido introducida por Dennis Gabor, y expuesta en su artículo "Acoustical Quanta and the Theory of Hearing". Gabor construyó, en 1946, un granulador sonoro, con base en un sistema de registro óptico que empleaba un proyector de cine de 16 mm. Mediante este mecanismo podía obtener pequeñas muestras de un sonido a intervalos regulares, y luego reproducirlas de modo tal que cambiara la duración del sonido sin que se modifique la altura, y viceversa.

Dennis Gabor (1900-1979) fue Premio Nobel de Física e inventor de la holografía.

El primer músico que creó una teoría compositiva con base en granos sonoros fue Iannis Xenakis, partiendo de la idea que todos los sonidos existentes pueden ser descriptos como un conjunto de partículas sonoras.
Desde 1971, muchos compositores, entre ellos Curtis Roads (1978) y Barry Truax (1988), han utilizado diferentes técnicas para la síntesis del sonido a partir de combinar estos breves eventos acústicos. Los granos a emplear pueden provenir de alguna otra técnica de síntesis (aditiva o frecuencia modulada, por ejemplo) o bien ser tomados de la fragmentación de un archivo de sonido. También suelen ser modificados en amplitud a través de pequeñas envolventes, que mejoran la transición entre eventos.
Existen numerosos y variados métodos que implementan la síntesis granular. Aquí trataremos los principales aspectos de esta técnica, realizando ejemplos en Pure Data.

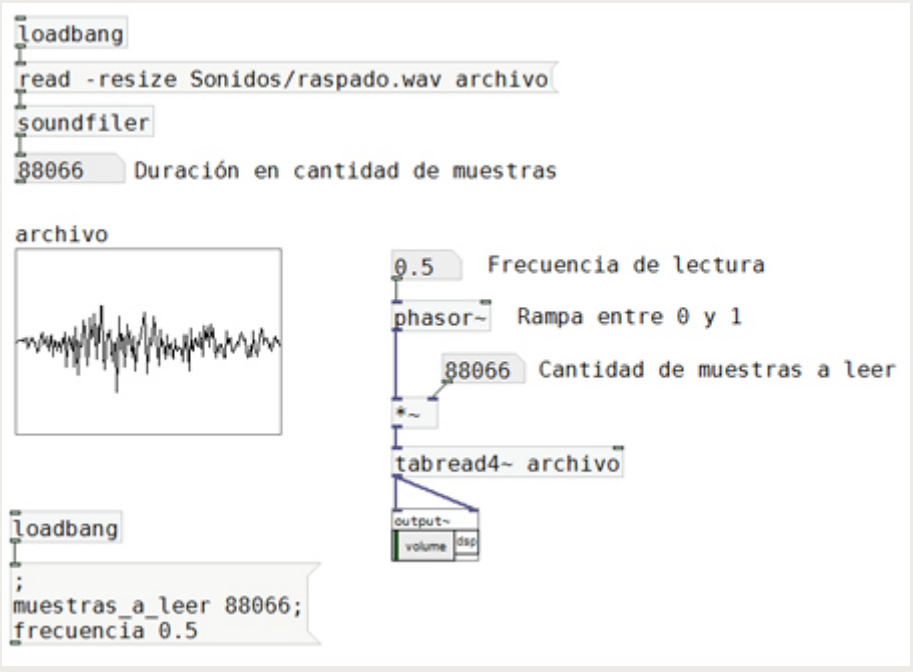
Iannis Xenakis (1922-2001) fue un compositor y arquitecto rumano, de ascendencia griega, considerado uno de los más importantes del siglo XX. Propuso una estética con base en la creación de obras a través de masas de sonidos y texturas, combinando marcos teóricos provenientes de la probabilidad y la estadística.

4.5.1. Lectura de un archivo de audio con *phasor~*

Al analizar el *patch* "24-soundfiler.pd" vimos que era posible alojar la información de un archivo de audio en una tabla, y luego ejecutarla a diferentes velocidades. Comenzaremos con algo similar, pero en lugar de leer las muestras almacenadas en la tabla a través de *line~*, lo haremos con un objeto llamado *phasor~*, que genera una rampa ascendente entre 0 y 1 a una frecuencia especificada. Multiplicando la salida de *phasor~* por la cantidad de muestras del archivo podremos leerlo cíclicamente, y cambiar la frecuencia de lectura.

G.4.8. ilustra este procedimiento. El archivo de sonido es cargado en una tabla mediante el objeto *soundfiler*, desde cuyo *outlet* obtenemos la cantidad de muestras de audio que contiene. En el mensaje *read*, donde especificamos el nombre del archivo a abrir, y el nombre de la tabla donde alojarlo, hemos agregado la inscripción *–resize*, con el propósito de que el tamaño de la tabla se adapte automáticamente a la longitud del archivo. Esto evitará tener que redimensionar la tabla cada vez que cambiemos de archivo.

G.4.8. Lectura de archivo mediante *phasor~*



Según mencionamos antes, el objeto *phasor~* genera linealmente muestras entre 0 y 1 a una frecuencia dada. Si a esos números los multiplicamos por la cantidad de muestras de la tabla, obtendremos una rampa entre 0 y ese valor, que nos permitirá leer cíclicamente la totalidad del archivo. Sin embargo, si multiplicamos por un número menor, podremos leer una porción del archivo, desde la muestra 0 hasta el número en cuestión.

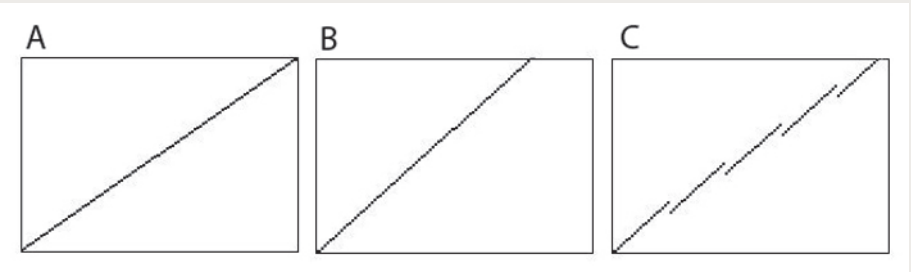
Para determinar con exactitud cuál debería ser la frecuencia adecuada de *phasor~* que permita leer el archivo sin acortarlo o alargarlo, es decir, a la velocidad correcta, podemos dividir la frecuencia de muestreo por la cantidad de muestras a leer (frecuencia = 44 100 / 88 066 = 0,50076 *Hz*).

El *patch* de G.4.8. se encuentra en el archivo "32-lectura de archivo con *phasor~*.pd". A través de este ejemplo podrá experimentar la lectura cíclica de un archivo de audio a distintas velocidades. Observe, además, el cambio de altura que se produce en el sonido, en relación con el cambio de velocidad en la lectura de las muestras, al modificar la frecuencia del oscilador. El *patch* "33-cálculo de frecuencia de *phasor~*" es similar al anterior, pero implementa el cálculo necesario para que la frecuencia del objeto *phasor~* permita la lectura cíclica del archivo a velocidad normal.

Una forma de lograr un cambio de altura del sonido sin que cambie su duración es, según mencionamos antes, implementando la técnica granular. Para ello, vamos a modificar el programa de lectura de un archivo de audio con *phasor~*, agregándole lo necesario para transportarlo sin que el sonido se acorte o alargue.

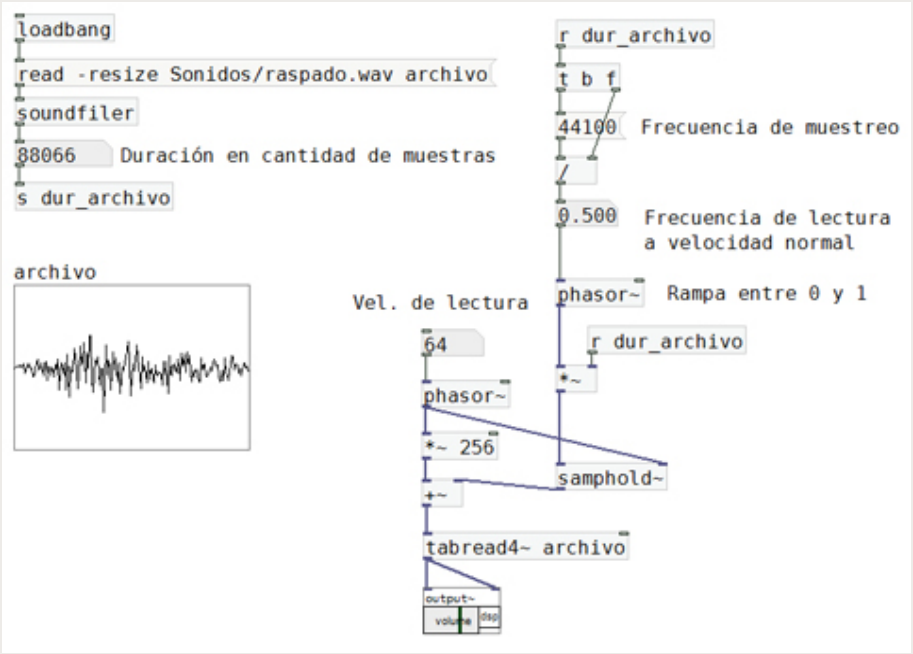
La figura siguiente (G.4.9.) muestra de qué modo realizamos la lectura. En *A* observamos un ciclo de *phasor~* cuya frecuencia es tal que permite leer la totalidad de las muestras del archivo a velocidad normal. En *B*, en cambio, aumentamos la frecuencia, por lo cual el archivo es leído más velozmente y la altura se incrementa en consecuencia. Por último, en *C*, vemos de qué manera podríamos leer el archivo para que la altura se incremente sin que varíe la duración, dividiendo a la señal en partes, donde cada segmento de recta conforma un "grano" o "quantum" sonoro. La pendiente de cada segmento de *C* equivale a la pendiente de *B*, sin embargo, los segmentos en su totalidad ocupan la misma cantidad de muestras que en *A*.

G.4.9. Modos de lectura de archivo



Para llevar a la práctica esta idea utilizaremos el objeto *samphold~*. Este objeto emplea un procedimiento denominado en inglés *sample and hold*, que consiste en tomar una muestra de una señal y repetirla hasta recibir la orden de tomar una nueva muestra. En el objeto *samphold~* ingresa por el *inlet* izquierdo la señal a muestrear, y por el derecho, una señal de control que dispara un nuevo muestreo cada vez que sus valores de amplitud descienden. Para generar esa señal de control usamos otro objeto *phasor~*, y a través de su frecuencia modificamos la altura del sonido almacenado.

G.4.10. Síntesis granular

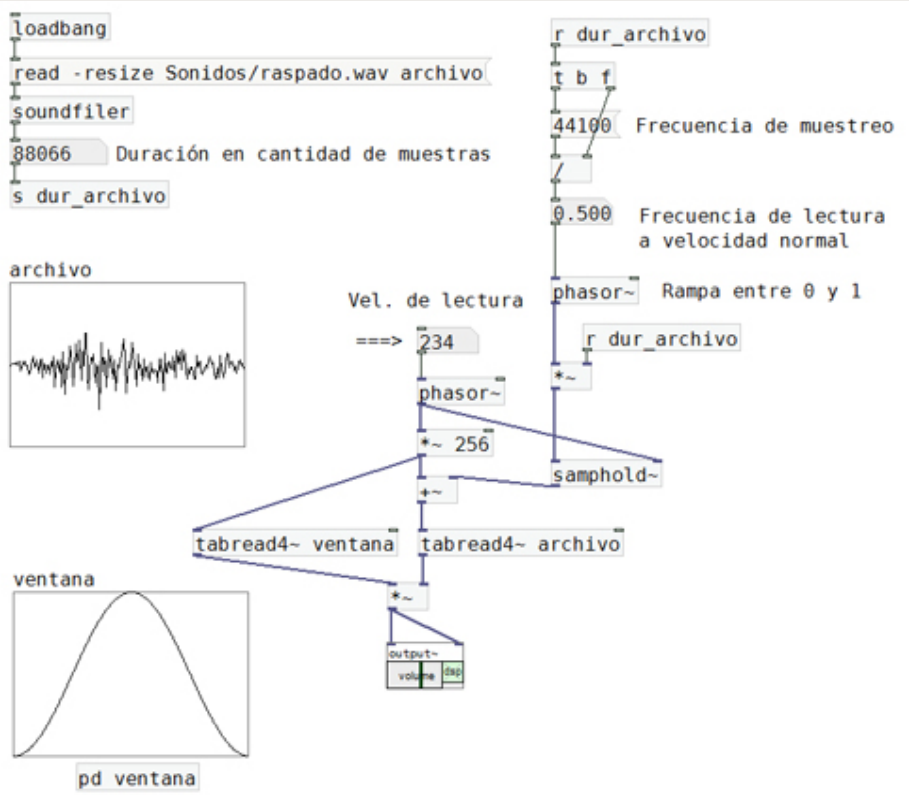


En G.4.10. se observa una implementación de lo expuesto anteriormente. La duración de los gránulos se establece en 256 muestras, y con el objeto *phasor~* de la izquierda controlamos la velocidad de lectura. La lectura de cada grano comienza en el tiempo que el objeto *phasor~* de la derecha determina, lográndose así una lectura fragmentada del archivo que lleva a aumentar o disminuir la altura del sonido sin que se modifique su duración.

El *patch* "34-síntesis granular.pd" contiene la programación de G.4.10.

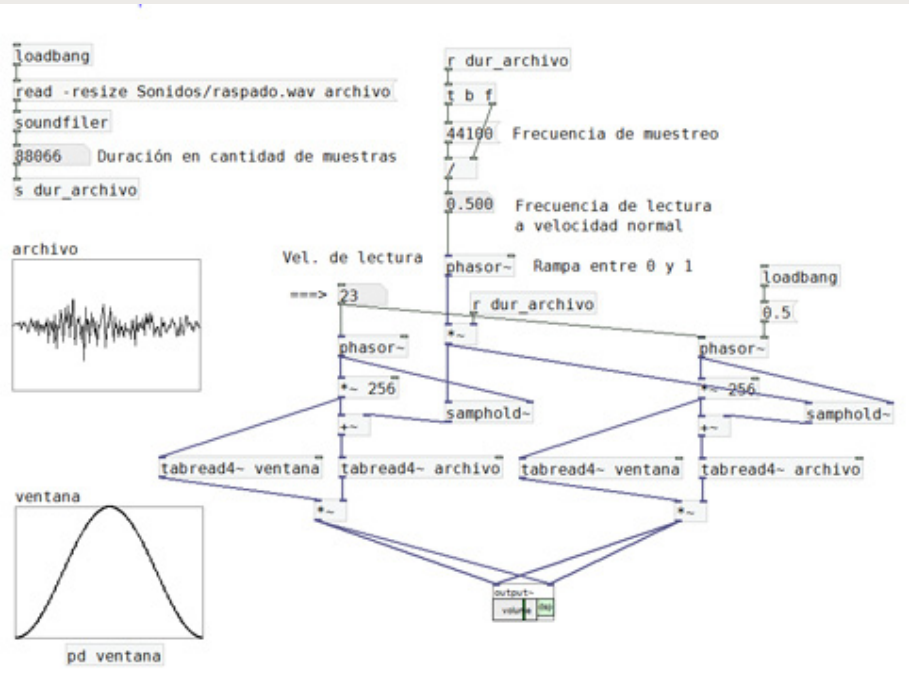
El *patch* que hemos realizado presenta un problema, pues produce saltos de amplitud entre grano y grano, los cuales se traducen en ruido. Podemos mejorar la transición entre los granos aplicando una pequeña envolvente que suavice las discontinuidades. La figura siguiente ilustra este procedimiento.

G.4.11. Síntesis granular con ventana



Multiplicamos cada grano de 256 muestras por una envolvente del mismo tamaño, eliminando así las discontinuidades de amplitud entre granos sucesivos. Sin embargo, aparecen pequeños silencios entre ellos, lo cual debería ser subsanado. Una manera de evitar las diferencias de amplitud producidas por el encadenamiento de envolventes es sumando otra lectura idéntica de los granos con sus envolventes, pero desfasada 128 muestras, lográndose de esta forma cubrir los huecos y que la amplitud se mantenga constante. Para generar el desfase en las lecturas empleamos dos objetos *phasor~* distintos, y a uno de ellos le modificamos la fase inicial 180º (introduciendo el valor 0.5 en el *inlet* derecho).

G.4.12. Síntesis granular con doble ventana



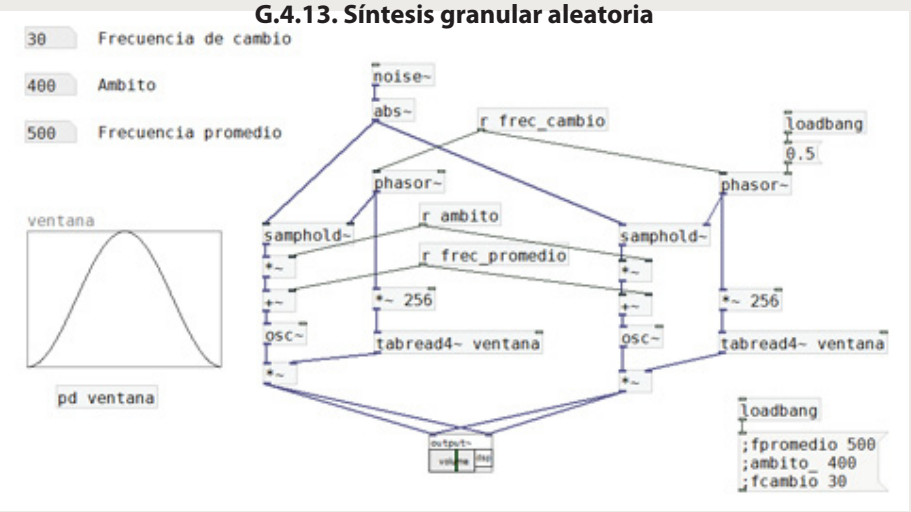
Los *patches* "35-síntesis granular con ventana.pd" y "36-síntesis granular con doble ventana.pd" contienen la programación de las G.4.11 y G.4.12.

En el próximo ejemplo vamos a producir una síntesis a partir de la generación de sonidos puros muy breves, de distinta frecuencia. Esas frecuencias van a oscilar en torno a una frecuencia central, desviándose de ella en un determinado ámbito.

Según se observa en G.4.13., empleamos los objetos *noise~* y *abs~* (que convierte muestras negativas en positivas) para obtener números al azar entre 0 y 1. Cada vez que finaliza un ciclo de *phasor~* el objeto *samphold~* toma uno de esos números al azar y lo repite hasta que se cumpla un nuevo ciclo. Posteriormente, se multiplica ese número para ampliar el ámbito, se le suma otro para centrar el rango en torno a él, y se lo ingresa al *inlet* de frecuencia de un oscilador. Además, se multiplica cada grano por una envolvente, al igual que en el ejemplo anterior, y se le superpone una versión desfasada, a fin de cubrir los silencios que la envolvente produce en los extremos del grano.

Mediante la modificación de los parámetros "frecuencia de cambio", "ámbito" y "frecuencia central" es posible crear "nubes" de alturas, ubicándolas en el registro y cambiándoles su espesor y densidad.

G.4.13. Síntesis granular aleatoria



El *patch* "37-síntesis granular aleatoria.pd" contiene la programación de G.4.13.

4.6. Distorsión no lineal

La distorsión no lineal –también conocida como *waveshaping*– utiliza los valores de salida de un oscilador de audio como índice de lectura de una tabla. Esa tabla almacena una función de transformación de la sinusoide del oscilador, a la que se denomina *función de transferencia*.

La técnica fue desarrollada por J. C. Risset, y luego perfeccionada por Arfib y Le Brun en 1979.

Jean-Claude Risset es un compositor nacido en 1938 en Francia, y pionero de la informática musical. Es autor de decenas de obras instrumentales, mixtas y electroacústicas, y publicó diversos escritos sobre acústica, psicoacústica y síntesis del sonido.

En G.4.13. observamos la sinusoide de entrada, la función de transferencia y la forma de onda de salida.
La función de transferencia puede imaginarse como un espejo que deforma la imagen, distorsionándola.

Si convertimos la función de transferencia en una recta a 45º veremos que no existe transformación y, en ese caso, podríamos comparar esa recta con un espejo plano, que simplemente cambia la dirección de la imagen sin modificarla.

G.4.14. Distorsión no lineal

En principio, podríamos usar cualquier curva como función de transferencia, no obstante, resulta necesario poder predecir los resultados y tener cierto control sobre la técnica. Con el propósito de generar componentes armónicas a partir de la distorsión de la señal sinusoidal de entrada suelen emplearse los denominados "polinomios de Chebyshev de primer tipo". Cada polinomio deforma la sinusoide de manera tal que se genera una nueva sinusoide, cuya frecuencia es múltiplo de la frecuencia de la señal de entrada. Dicho de otro modo, ingresa un ciclo de una sinusoide y salen dos ciclos, o tres, o cuatro, o más, dependiendo del polinomio utilizado. A través del escalamiento y la suma de varios polinomios es posible generar un sonido complejo con los armónicos deseados, y la amplitud requerida para cada uno de ellos.

Los primeros ocho polinomios de Chebyshev son los siguientes:
$T_1(x) = x$
$T_2(x) = 2x^2 - 1$
$T_3(x) = 4x^3 - 3x$
$T_4(x) = 8x^4 - 8x^2 + 1$
$T_5(x) = 16x^5 - 20x^3 + 5x$
$T_6(x) = 32x^6 - 48x^4 + 18x^2 - 1$
$T_7(x) = 64x^7 - 112x^5 + 56x^3 - 7x$
$T_8(x) = 128x^8 - 256x^6 + 160x^4 - 32x^2 + 1$, etcétera.

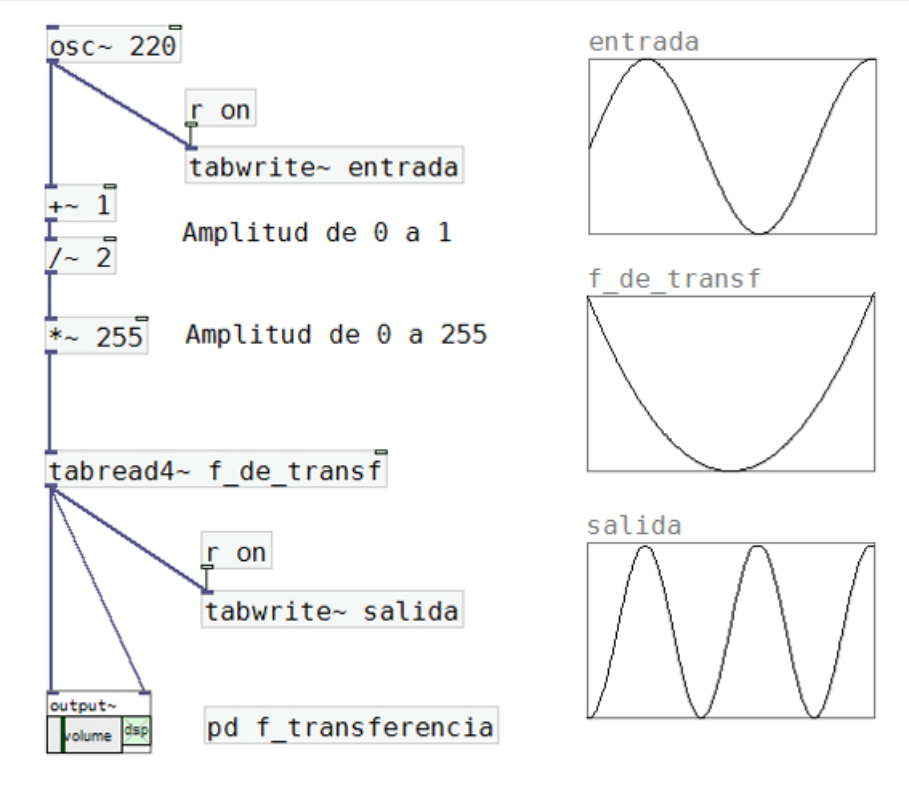
El primero de ellos no afecta a la sinusoide de entrada; el segundo, produce el segundo armónico; el tercero, el tercer armónico, y así sucesivamente.

La síntesis se realiza sumando los polinomios (uno por cada componente) y multiplicando a cada uno de ellos por el valor de amplitud deseado, por ejemplo:

$$f(x) = T_1(x) + 0.5\ T_2(x) + 0.33\ T_3(x) + \ldots$$

El *patch* de G.4.15. calcula la función de transferencia a partir del segundo polinomio, y la almacena en una tabla. Los valores de amplitud de la sinusoide de entrada se escalan de acuerdo con el tamaño de esa tabla (256 muestras) y se usan como puntero de lectura de la función de transferencia. El resultado es una sinusoide con el doble de frecuencia que la señal de entrada.

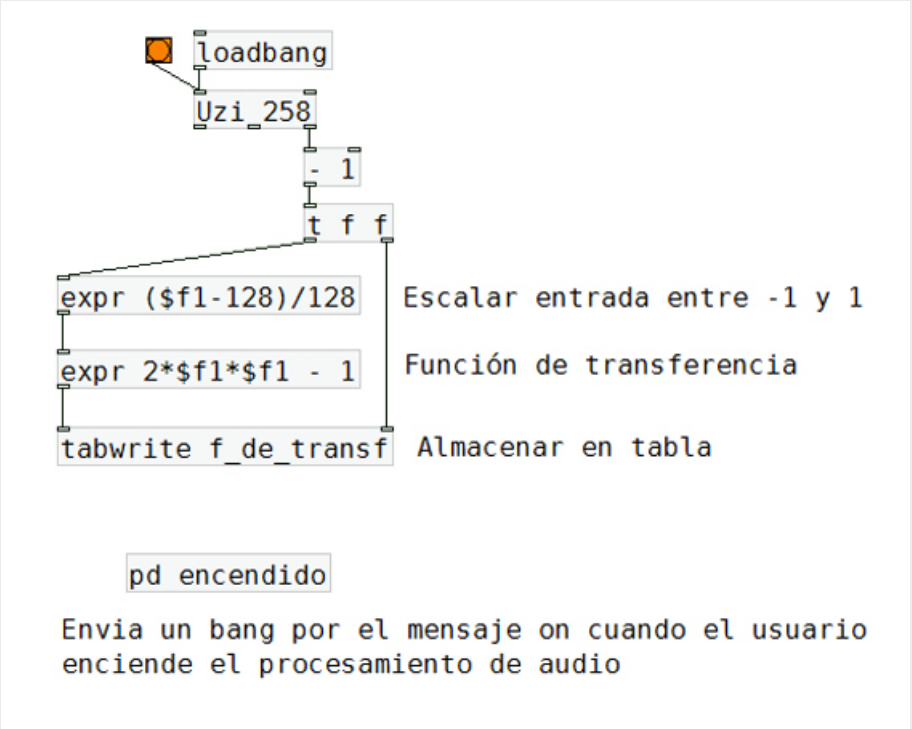
G.4.15. Waveshaping



En G.4.15. observamos un *subpatch* (PD *f_transferencia)*, que calcula la función de transferencia a partir del segundo polinomio de Chebyshev, $T_2(x) = 2x^2 - 1$.

En su interior (ver G.4.16.) un objeto *Uzi* envía velozmente a su salida todos los enteros comprendidos entre 1 y 258 que, convenientemente escalados entre -1 y 1, dan valores a la variable *x* del polinomio. Luego, los resultados de evaluar la función para cada x son almacenados en una tabla.

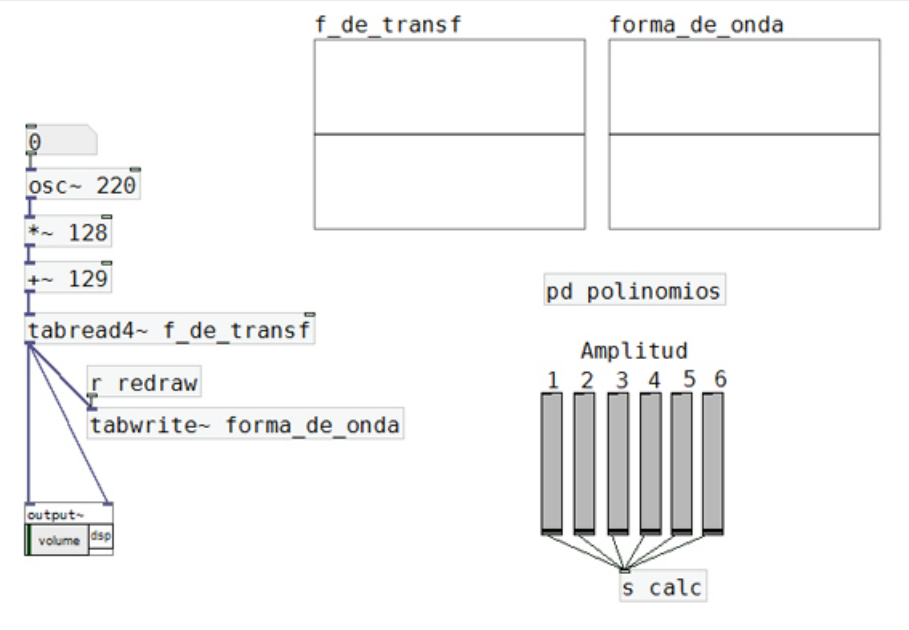
G.4.16. *Subpatch* para generar función de transferencia



El *patch* "38-*waveshaping* 1.pd" contiene la programación de G.4.15.

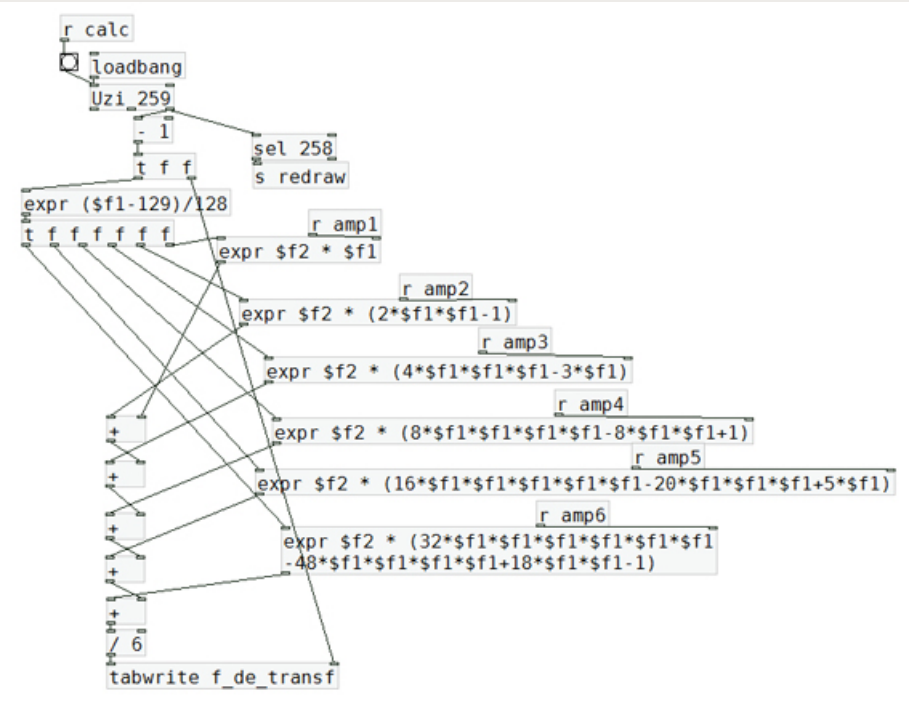
A continuación procederemos a vincular los polinomios, sumándolos y multiplicando a cada uno de ellos por un factor de amplitud, para crear así las componentes armónicas del sonido a sintetizar. En el *patch* de G.4.17. observamos un circuito similar al empleado anteriormente, pero aquí se combinan los primeros seis armónicos, dando a cada uno una amplitud relativa a través de los *sliders* que se muestran en G.4.17.

G.4.17. Waveshaping



El *subpatch* denominado PD *polinomios* contiene las ecuaciones de los polinomios, descriptas a través de objetos *expr*. Estos objetos reciben los valores de amplitud de los *sliders*, los valores de x para calcular las funciones, y sus salidas se suman y se dividen por la cantidad de componentes –seis en este caso– para dar forma a la función de transferencia.

G.4.18. Combinación de polinomios



El *patch* "39-*waveshaping* 2.pd" contiene la programación de G.4.17.

4.7. Modelado físico

La técnica de síntesis por modelado físico se basa en modelos matemáticos que simulan el comportamiento físico de los instrumentos musicales.

Como parte de estas técnicas, nos detendremos en la simulación de las cuerdas punteadas –como la guitarra– para lo cual utilizaremos un desarrollo conocido como algoritmo de Karplus-Strong.

Alexander Strong fue el inventor del algoritmo, y Kevin Karplus formalizó el modelo matemático sobre el cual se basa su funcionamiento. El instrumento así creado fue denominado *digitar* (por *digital guitar*) por los autores, y consiste en una breve banda de ruido a la cual se le aplica un retardo seguido de un filtrado de agudos. La salida de la señal retardada y filtrada reingresa nuevamente en el circuito, de forma periódica, dando lugar a un sonido de altura definida, similar al de una cuerda pellizcada.

La primera aplicación musical del algoritmo aparece en la obra *All your children be acrobats* (1981) de David Jaffe. Este autor, junto a Julius Smith, extendió el algoritmo con el propósito de implementarlo en sintetizadores comerciales.

A fin de implementarlo, y según lo hemos descrito, deberemos recurrir a operaciones de retardo y de filtrado que desarrollaremos en detalle posteriormente (en las Unidad 5 y Unidad 6, respectivamente). No obstante ello, emplearemos los objetos de Pure Data que realizan estas tareas, adelantándonos al tratamiento más profundo de estos temas.

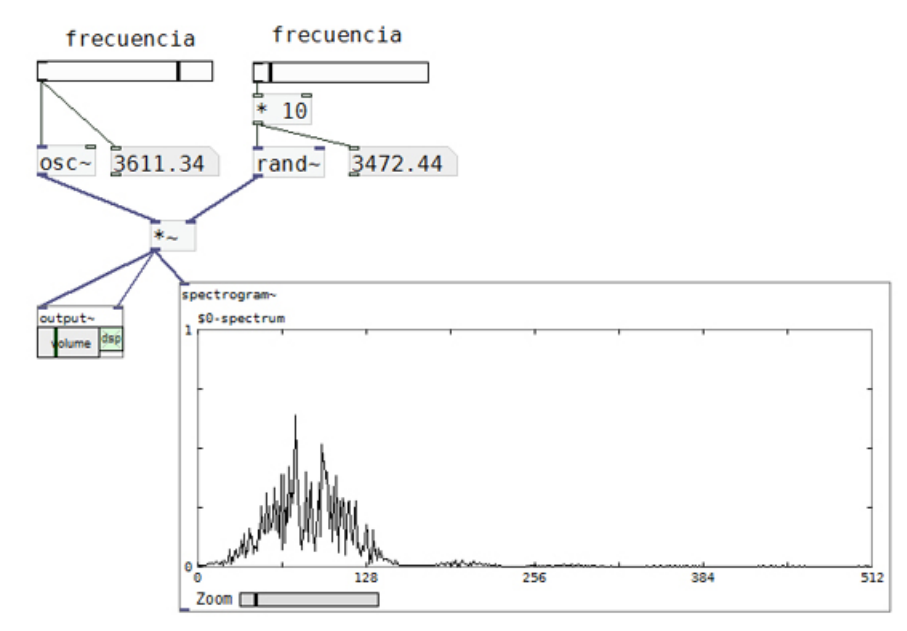
4.7.1. Generación de bandas de ruido centradas en torno a una frecuencia dada

La fuente de excitación del algoritmo, según mencionamos antes, está dada por una banda de ruido. La posibilidad de modificar el ancho y la ubicación de la banda en el registro nos permite simular diferentes comportamientos del instrumento, derivados del modo de ejecución. Una guitarra, por ejemplo, suena de forma diferente si pulsamos la cuerda cerca del puente (elemento de sujeción de las cuerdas sobre la caja), o cerca del orificio de la caja de resonancia. En el primer caso, el sonido producido es brillante y, bastante más opaco, en el segundo.

Para imitar este comportamiento vamos a partir del objeto *rand~* que genera una señal de audio a partir de números aleatorios. Por su *inlet* recibe la cantidad de números a generar por segundo y traza rampas entre ellos para calcular los valores de las muestras intermedias. Mientras mayor es la cantidad de números al azar, más se aproxima el sonido al ruido blanco. Si, en cambio, el número se hace más pequeño, el sonido resultante se asemeja a un ruido cada vez más filtrado en agudos.

Cuando tratamos la modulación en anillo vimos que al multiplicar dos señales de audio se producía la convolución de sus espectros. Podemos utilizar ese principio para continuar modelando la banda de ruido generada por *rand~*, desplazándola en el registro de audio mediante una multiplicación con una señal sinusoidal. A través de este proceso, las frecuencias de las componentes del ruido se suman y se restan a la frecuencia de la sinusoide, centrándose en torno a ella. G.4.19. muestra la programación necesaria para alcanzar este propósito.

G.4.19. Bandas de ruido centradas en una frecuencia dada



Según se aprecia en G.4.19., utilizamos el objeto denominado *spectrogram~* para representar gráficamente el espectro de la señal resultante.

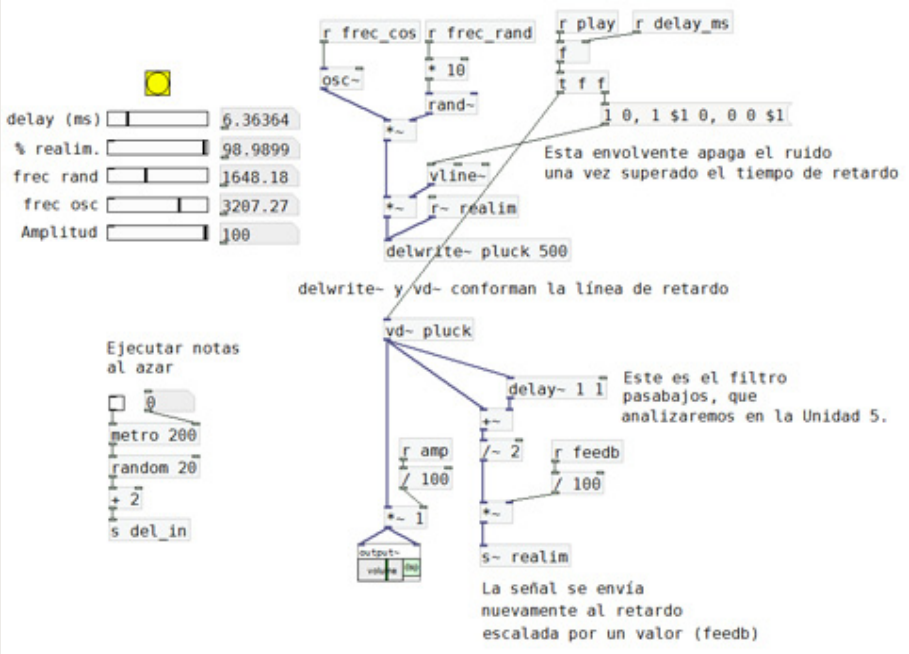
El *patch* "40-rand.pd" contiene la programación de G.4.19. Experimente la generación de bandas de ruido, modificando la frecuencia del objeto *rand~* y la frecuencia de la sinusoide producida por el oscilador.

El algoritmo de Karplus-Strong

El ruido que sirve de excitación –y que se compara con la acción de pulsar la cuerda– debe durar un tiempo idéntico al del retardo a aplicar. Para ello, una envolvente generada con *vline~* deja pasar la señal de ruido y la interrumpe luego de una cantidad de milisegundos preestablecida. La señal retardada vuelve a ingresar en la línea de retardo repetidamente, pero multiplicada cada vez por un valor inferior a 1, por lo cual, pasado cierto tiempo se extingue. En las sucesivas repeticiones produce una señal periódica, que es percibida como un sonido tónico, y que decrece en amplitud a medida que el tiempo transcurre. El tiempo de retardo equivale al período de la señal sintetizada y, por lo tanto, es la inversa de la frecuencia ($T = 1 / f$, donde T es el tiempo que dura un ciclo, expresado en segundos, y f la frecuencia del sonido).

Nótese, además, que luego del retardo, ubicamos un filtro pasabajos (este filtro elimina las componentes de mayor frecuencia y deja pasar las más bajas), que actúa cada vez que la señal abandona la línea de retardo. Mediante este recurso la señal no solo decrece en amplitud, sino que se torna más y más opaca con el paso del tiempo.

G.4.20. Algoritmo de Karplus-Strong



El *patch* "41-Karplus Strong básico.pd" contiene la programación de G.4.20. Es conveniente que luego de estudiar las unidades 5 y 6 vuelva a analizarlo, para una mejor comprensión del tema.

Serra, J. "Perspectivas actuales en la síntesis digital de sonidos musicales", [EN LÍNEA]. En: *Formats. Revista de Comunicación Audiovisual. Barcelona*. Universidad Pompeu Fabra. 1997. Disponible en: <http://www.iua.upf.edu/formats/formats1/a07et.htm> [Consulta: 23 de julio de 2013].

Actividad 5

a. Programar un *patch* que produzca sonidos no tónicos por síntesis aditiva cuyas frecuencias se generen aleatoriamente.
b. Aplicar a cada componente una envolvente de dos pasos que también se genere por medio del azar.
En el archivo "Respuesta Actividad 05.pd" encontrará una posible solución de este ejercicio. Compare los resultados que obtuvo con los del archivo y anote las diferencias.

Actividad 6

Programar un *patch* de síntesis de sonido por frecuencia modulada, empleando tres osciladores en cascada. Los dos primeros osciladores generan una FM simple, y la salida se utiliza para modular a un tercer oscilador que produce la señal portadora.

En el archivo "Respuesta Actividad 06.pd" encontrará una posible solución de este ejercicio. Compare los resultados que obtuvo con los del archivo y anote las diferencias.

5. Filtros digitales

Objetivos

- Conocer los distintos tipos de filtros empleados en el procesamiento de audio digital.
- Utilizar los filtros digitales en el tratamiento de señales de audio.

5.1. Tipos de filtros

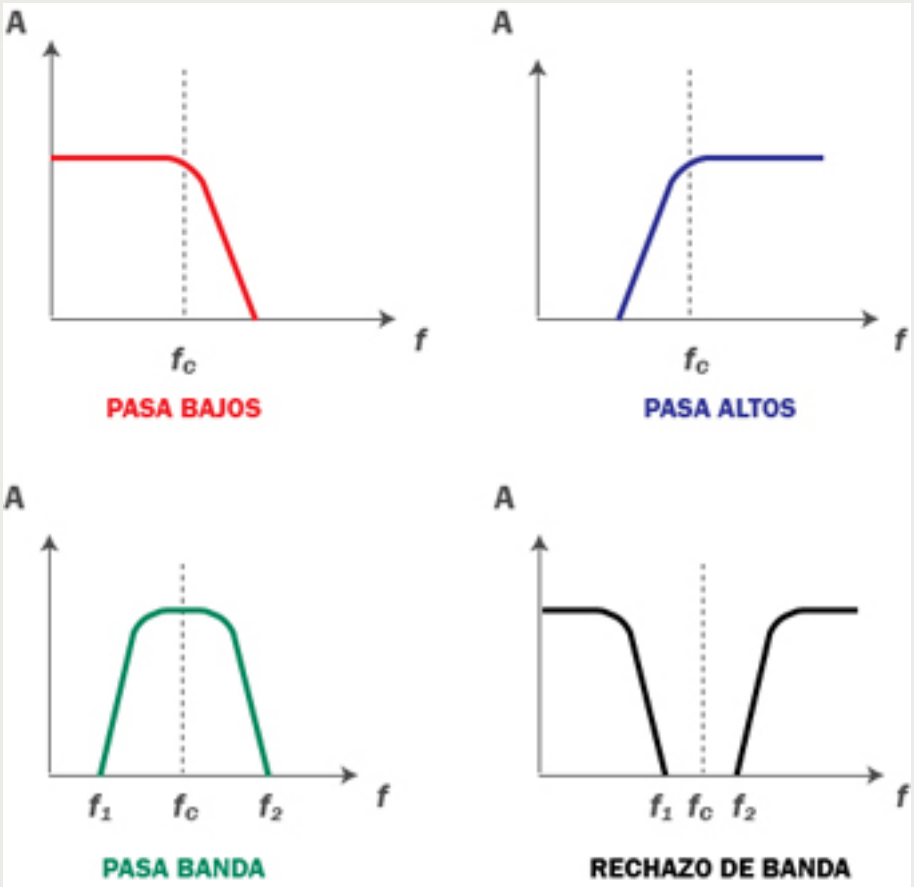
Los filtros son dispositivos capaces de retener determinadas porciones del espectro de una señal de audio, y de dejar pasar otras, y pueden clasificarse de diversas formas. Una clasificación posible se basa en las regiones espectrales que no son retenidas por el filtro, y los divide en: "pasa bajos", "pasa altos", "pasa banda" y "rechazo de banda", por oposición al anterior.

Un filtro pasa bajos deja pasar las componentes de menor frecuencia (sonidos graves) y retiene a las de mayor frecuencia (sonidos agudos). Un pasa altos funciona al revés, deja pasar las altas frecuencias y retiene a las bajas. Un pasa banda deja pasar una banda determinada de frecuencia y retiene al resto y, por último, un rechazo de banda produce el efecto contrario. En los dos últimos casos, a la diferencia entre f_2 y f_1 la denominamos **ancho de banda**.

En todos los casos, es posible especificar una frecuencia de corte, que en un pasa bajos o un pasa altos es el límite entre lo que pasa y lo que es retenido. En los dos filtros restantes solemos denominar a esta frecuencia como *frecuencia central*, y establecemos, además, un ancho de banda, que es la porción pasante o la retenida, respectivamente, que se centra alrededor de la frecuencia central.

Los gráficos de G.5.1. ilustran el comportamiento de estos cuatro tipos de filtros. El área contenida bajo cada curva es la región pasante. A estos gráficos, que representan la atenuación o incremento de la amplitud en función de la frecuencia, los denominamos gráficos de "respuesta en amplitud" del filtro.

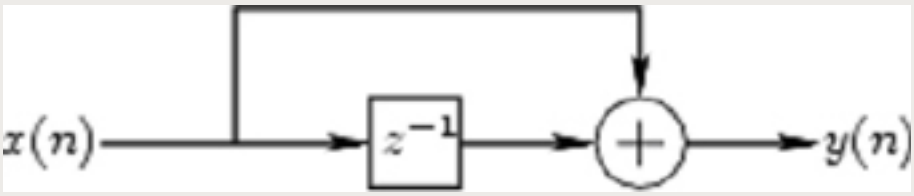
G.5.1. Respuesta en amplitud de los filtros



5.2. El filtro pasa bajos básico

Su versión más elemental se obtiene sumando a una señal digital una copia de sí misma, retrasada en una muestra. Este proceso se representa a través del siguiente diagrama:

G.5.2. Diagrama de filtro pasa bajos elemental



La señal de entrada se representa como *x(n)* –se lee *x* de *n*– que es el modo de referirnos a una forma de onda cualquiera, cuyos valores de amplitud para cada muestra dependen del número de muestra considerado *(n)*. Este tipo de notación no dice nada sobre el contenido de la señal, pero nos permite nombrarla y diferenciar, por ejemplo, una señal de otra. Siguiendo este criterio, denominamos *y(n)* a la señal de salida filtrada, que es distinta a *x(n)*.

La caja con la inscripción z^{-1} se denomina *línea de retardo* y representa un dispositivo capaz de retrasar en una muestra a la señal de entrada. Cuando ingresa a la línea la primera muestra, no sale nada (en realidad, sale un cero); cuando ingresa la segunda, sale la primera, y así sucesivamente. La cantidad de muestras que se retrasa la señal se indica con el exponente negativo (-1 en el ejemplo), el cual puede asumir cualquier valor (z^{-24} significa un retraso de 24 muestras).

Siguiendo el diagrama y resumiendo, ingresa al filtro una señal *x(n)* que se suma a una versión retrasada en una muestra de sí misma, y da como resultado otra señal filtrada *y(n)*.

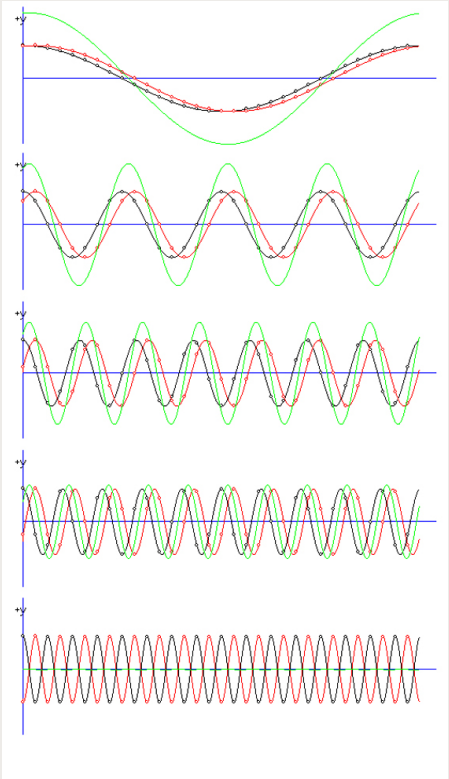
Lo mismo puede representarse en notación simbólica, de este modo:

$$y(n) = x(n) + x(n-1)$$

Al valor de la muestra n de la función x se le suma el valor de la muestra $n - 1$ (la anterior) de la misma función. Este sería el procedimiento a aplicar en un programa de computación para lograr el filtro en cuestión. A este tipo de expresión se lo denomina ecuación en *diferencias del filtro*.

Resulta difícil imaginar cómo es posible que el resultado obtenido a través de esta operación sea el de un filtro pasa bajos. Para tratar de comprenderlo vamos a hacer pasar señales sinusoidales de distintas frecuencias por nuestro filtro básico y observar qué sucede. En G.5.3. vemos la señal original [*x(n)*, roja] y la retrasada [*x(n - 1)*, negra] que sumadas, dan la resultante [*y(n)*, verde].

G.5.3. Suma de sinusoides, una de ellas retrasada en una muestra

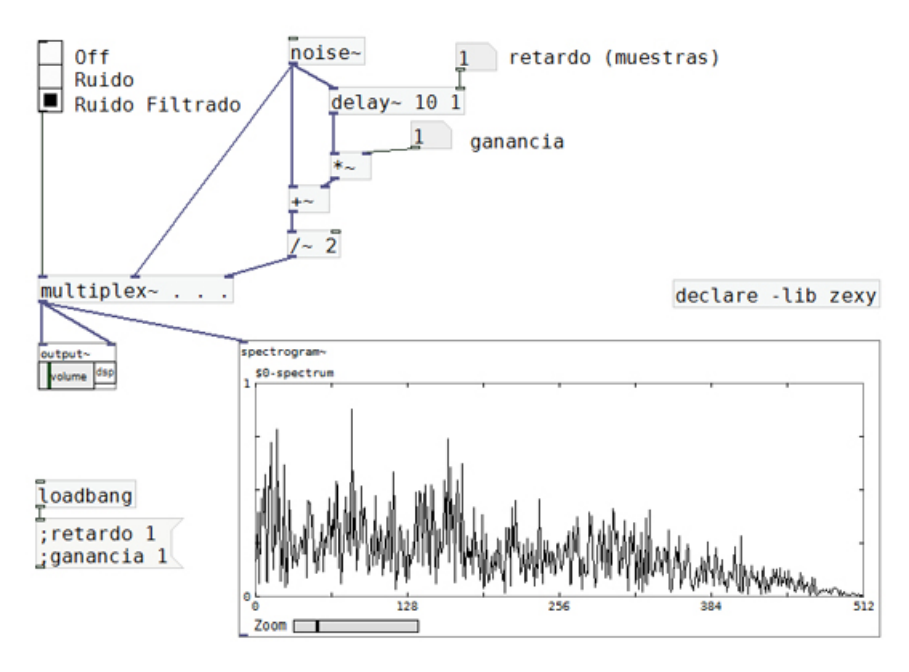


Si observamos detenidamente los gráficos, notamos que a medida que la frecuencia es mayor, se produce una mayor diferencia de fase entre la señal original y la retardada, si lo que las separa es siempre una muestra. Este desfase origina una disminución gradual de la amplitud a medida que aumenta la frecuencia, y se vuelve cada vez más notable; pues, al llegar a la mitad de la frecuencia de muestreo, retrasar una muestra equivale a una oposición de fase.

Si trazamos una curva que muestre los valores de amplitud de la resultante en función de la frecuencia de cada señal sinusoidal que ingresa al filtro, obtenemos la respuesta en amplitud.

G.5.4. muestra un *patch* con la programación del filtro básico explicado.

G.5.4. Filtro básico



Al ruido blanco producido por el objeto *noise~* le sumamos una copia retrasada en una muestra. Finalmente, el resultado se divide por 2, pues para bajas frecuencias estamos duplicando la amplitud de la señal de entrada, al sumarle una copia de sí misma ligeramente desfasada.

En el *patch* vemos, además, que la salida de la copia retardada está multiplicada por un factor de ganancia. Si ese factor es -1 ocurre lo mismo que si restáramos las dos señales, y el filtro se comporta como un pasa altos. Si, en cambio, retrasamos la señal en dos muestras da un rechazo de banda si la ganancia es 1, y un pasa banda si la ganancia es -1. Vale decir que hemos logrado los cuatro tipos de filtros cambiando el signo de la ganancia y la cantidad de muestras de retraso. El objeto *spectrogram~*, que dibuja el espectro de la señal resultante, nos permite apreciar la respuesta en amplitud de cada filtro, considerando que la señal de entrada es un ruido blanco, y posee todas las componentes del registro de audio.

El *patch* "42-filtro básico.pd" contiene la programación de G.5.4. Experimente la conformación de los cuatro tipos de filtros básicos cambiando la ganancia a -1 o +1, y la cantidad de muestras de retardo, de 1 a 2.

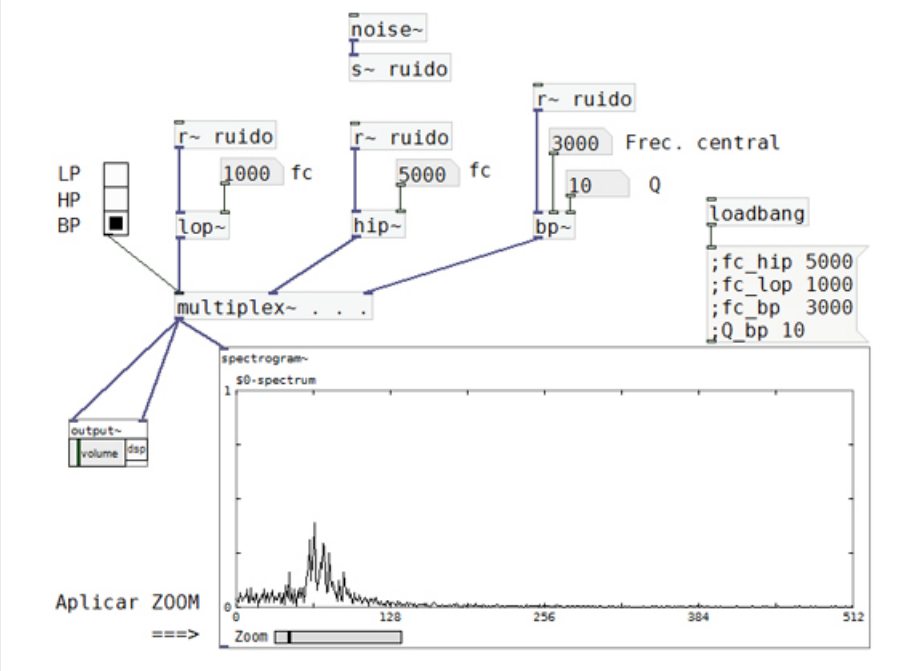
De la observación del filtro básico habrán notado que ya hemos hecho uso de él en el apartado 4.7, al estudiar la síntesis por modelado físico. Como se trata de un filtro muy elemental que corta muy paulatinamente desde 0 *Hz* hasta la mitad de la frecuencia de muestreo, resulta ideal para el empleo que quisimos darle.

Pero, en rigor, los filtros que usamos en la práctica suelen ser más complejos, si bien el principio de funcionamiento es el mismo para todos, pues en todos los casos operan aplicando retardos a la señales de entrada o salida.

5.3. Filtros usados en PD

El siguiente gráfico muestra un *patch* con los filtros más comúnmente usados en PD.

G.5.5. Filtros en PD



Los objetos *lop~* (pasa bajos) y *hip~* (pasa altos) reciben solamente valores de frecuencia de corte en *Hertz*, mientras que *bp~* (pasa banda) recibe la frecuencia central en *Hz* y el ancho de la banda pasante, expresado como Q.

En los filtros pasa banda puede expresarse el ancho de banda de dos formas distintas. La primera en una cantidad de *Hz* constante, que no depende de la frecuencia central. El problema de esta forma de especificarlo reside en que una banda de 100 *Hz* en el registro grave no se percibe igual que una de 100 *Hz* en el agudo, sino mucho más pequeña. Para entender esto basta con pensar que entre 100 y 200 *Hz* hay una octava musical (el doble de frecuencia es percibido como una octava), pero en el agudo una octava no se encuentra entre 1000 y 1100 *Hz*, sino entre 1000 y 2000 *Hz*. Por ello, mantener el ancho de banda constante en *Hertz* no se vincula con la percepción. El otro modo de especificar un ancho de banda es mediante *Q*. Ese valor depende de la frecuencia de corte, del siguiente modo:

Q = frecuencia central / ancho de banda en *Hz*

Esto significa que si mantenemos *Q* constante, el ancho de banda debe crecer en cantidad de *Hz* a medida que la frecuencia central se incrementa, pues, de otro modo, la igualdad de la fórmula dejaría de existir. Utilizando *Q* constante puede percibir el mismo ancho de banda para cualquier frecuencia central.

El *patch* "43-filtros usuales en PD.pd" contiene la programación de G.5.5.

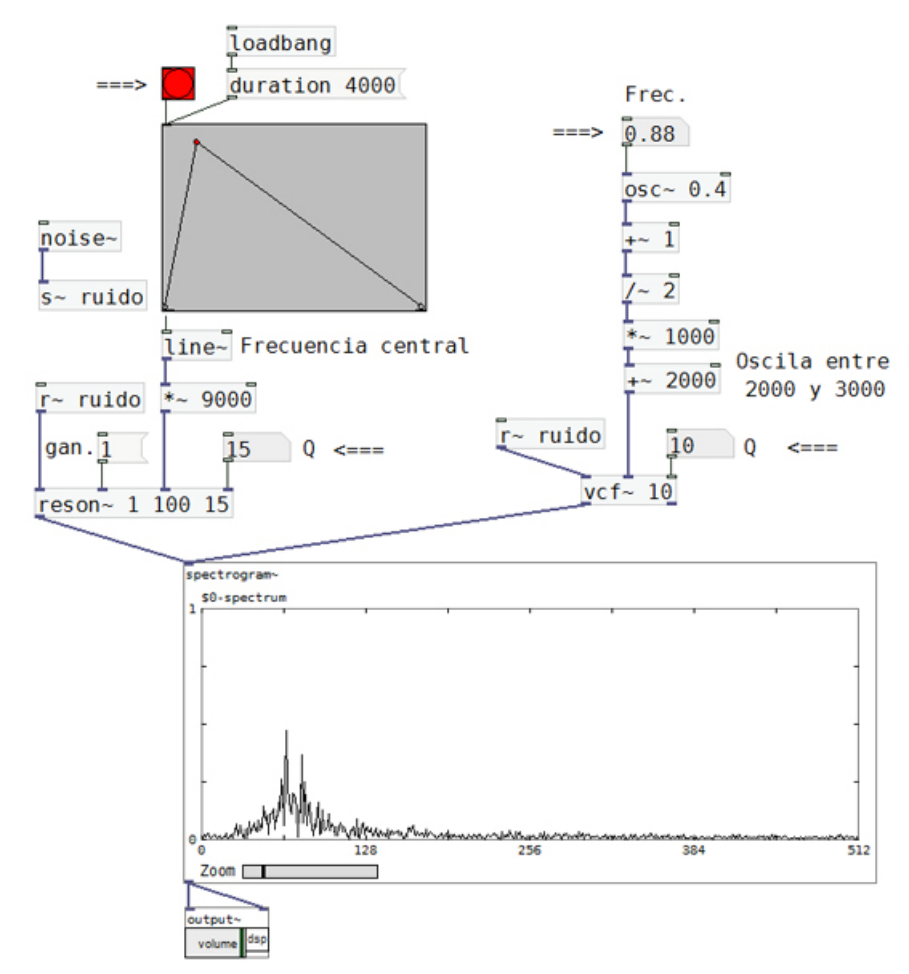
5.3.1. Filtros controlados por medio de señales de audio

El filtro pasa banda *reson~*, de la librería Cyclone, posee cuatro *inlets* por donde ingresan, de izquierda a derecha, la señal a filtrar, la ganancia (amplitud de salida), la frecuencia central y el ancho de banda expresado en valores de Q. El *inlet* de frecuencia central admite la conexión de señales de audio para el control de este parámetro.

A la izquierda de G.5.6. observamos un *patch* en el que la frecuencia es controlada por una envolvente de audio (*line~*) cuya salida está escalada en un rango comprendido entre 0 y 9000. A la derecha, vemos otro filtro pasa banda, denominado *vcf~* y similar a *bp~*, que también admite control a audio *rate*, pero en este caso, la frecuencia central del filtro es controlada por la salida de un oscilador. La amplitud del oscilador está escalada en un rango entre 2000 y 3000, y eso logra una variación sinusoidal de la frecuencia del filtro en ese rango.

Para ambos ejemplos es posible apreciar el espectro resultante.

G.5.6. Filtros controlados por señales de audio



El *patch* "44-filtros controlados por audio.pd" contiene la programación de G.5.6.

5.4. Algunos procesos que emplean filtros

A continuación analizaremos procesos de audio que involucran filtros y resultan de interés en el tratamiento del sonido.

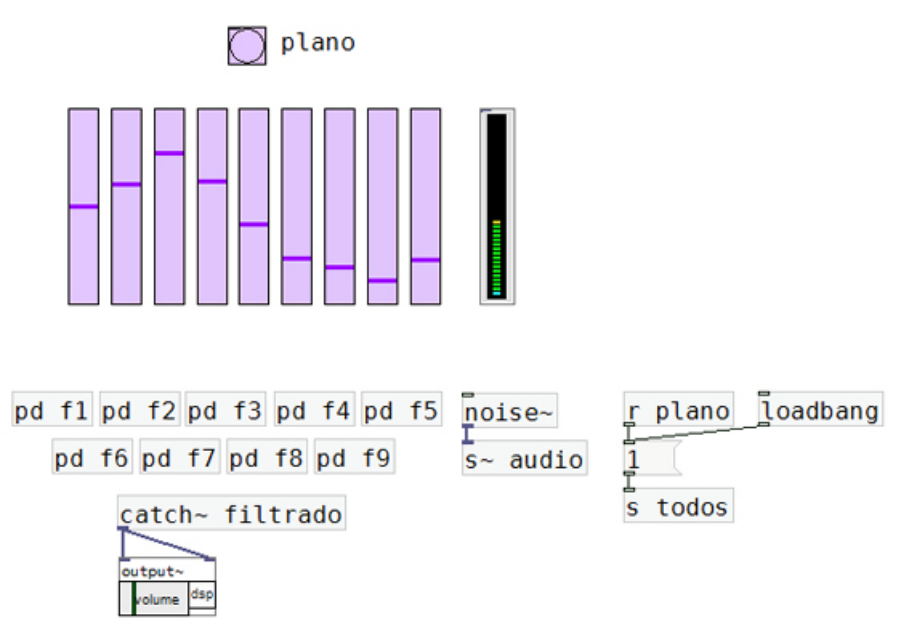
5.4.1. Ecualizador gráfico

Un ecualizador gráfico es un banco de filtros pasa banda, cuyas frecuencias centrales y ancho de banda están ajustados de manera tal que las regiones son percibidas como subdivisiones iguales en todo el registro de audio. Los ecualizadores gráficos varían en cuanto a cantidad de bandas, y suelen denominarse de acuerdo con el intervalo musical que comprende cada banda. Según este criterio de clasificación, encontramos ecualizadores: por octavas, por quintas, por cuartas, por terceras o por segundas.
A fin de simplificar la programación, vamos a desarrollar un ecualizador por octavas, lo cual significa que la frecuencia central de cada filtro resulta de multiplicar por dos la frecuencia del filtro inmediato anterior.
De este modo, si partimos de una frecuencia de 55 *Hz* (nota *LA* 1) la frecuencia central del segundo filtro será 110 *Hz*, la del tercero 220 *Hz*, etcétera.
La lista de frecuencias que resulta es:

55 - 110 - 220 - 440 - 880 - 1760 - 3520 - 7040 - 14080

Respecto al ancho de banda de cada filtro, lo expresamos a través de *Q*, y el valor que corresponde a una octava es *Q* = 1.

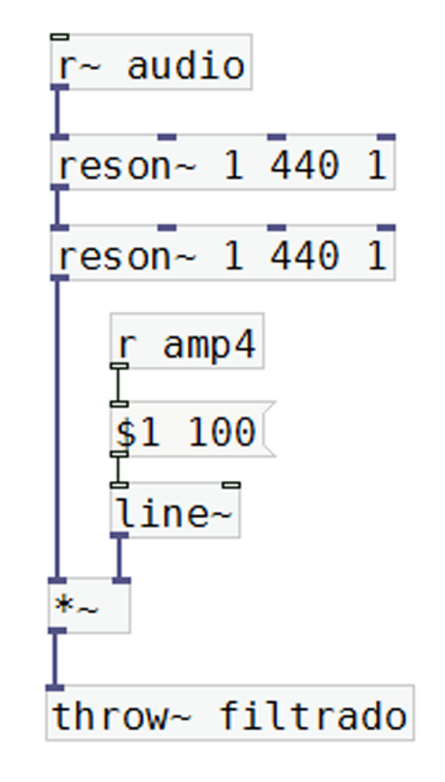
G.5.7. Ecualizador gráfico de 9 bandas



Observamos en G.5.7. la ventana principal del ecualizador, donde se aprecian los *sliders* que controlan la amplitud de cada banda. Mediante el botón con la leyenda "plano" podemos moverlos todos a la posición cero.

Para cada banda creamos un *subpatch*, que se aprecia en G.5.8. La señal de audio a procesar se recibe remotamente, al igual que los valores de amplitud. Para que los cambios bruscos de amplitud no generen clics, utilizamos un objeto *line~*, que efectúa una rampa de 100 milisegundos entre el valor anterior y el actual. También se observa que ubicamos dos filtros pasa banda idénticos, en serie, con los mismos parámetros, lo cual mejora la acción del filtro.

G.5.8. *Subpatch* de una de las bandas del ecualizador



El *patch* "45-ecualizador gráfico.pd" contiene la programación de G.5.7. Aquí podrá experimentar la audición de cada una de las bandas por separado, y modificar la programación para ingresar señales desde un micrófono o bien desde un archivo de audio.

5.4.2. Vocoder

El Vocoder (término derivado de *voice encoder*, codificador de voz) es un dispositivo que analiza una señal de entrada mediante un banco de filtros pasa banda, y la resintetiza a partir de un sonido rico en armónicos o un ruido blanco, filtrado con otro banco de filtros idéntico al utilizado para el análisis.

El origen del Vocoder se remonta a fines de la década de 1930, y fue originalmente desarrollado para codificar transmisiones durante la Segunda Guerra Mundial. Tiempo después comenzó a utilizarse en aplicaciones musicales, particularmente en la composición de música electrónica.

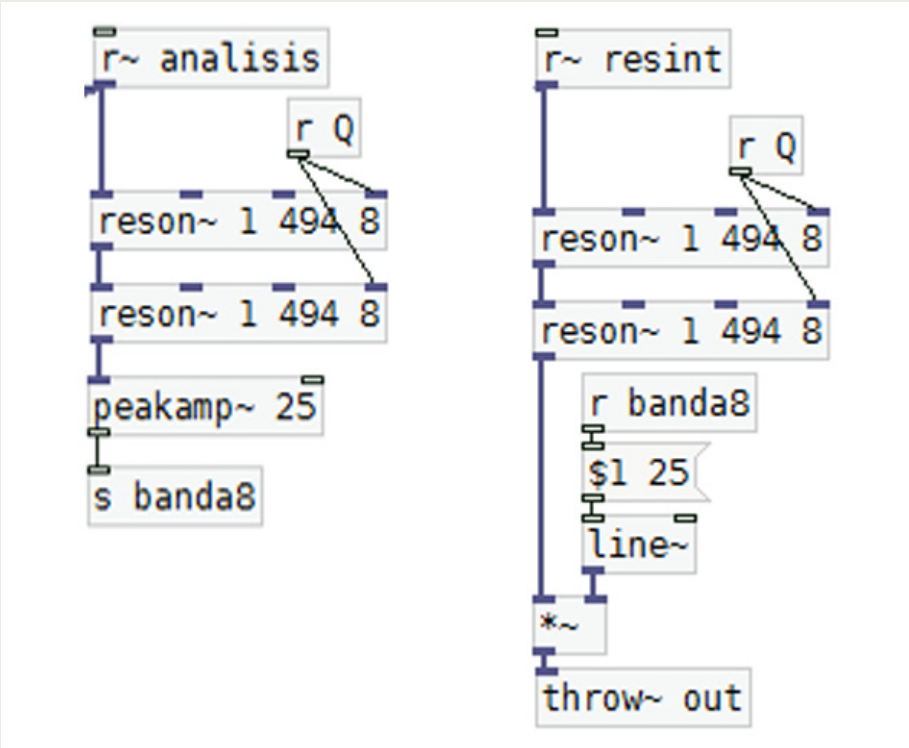
Vamos a programar un Vocoder de 24 bandas, afinadas por el intervalo musical de segunda mayor (la distancia entre las notas DO y RE, por ejemplo). Para ello, tomando una frecuencia de partida, debemos determinar cómo calcular las frecuencias centrales de los filtros. En un piano, por ejemplo, la distancia menor entre dos notas es el semitono (intervalo de segunda menor). Partiendo de una frecuencia, si deseo saber la frecuencia del semitono siguiente, debo multiplicar por 1,059463, que es la raíz doceava de 2. En nuestro caso, dado que una segunda mayor contiene dos semitonos, debo multiplicar la frecuencia de partida dos veces por ese valor, o sea por 1,12246204. Partiendo de 220 *Hz* y multiplicando por ese número, obtengo la frecuencia central de la segunda banda. Luego, ese resultado nuevamente por el número, y así, hasta alcanzar la banda 24.

G.5.9. Ventana principal del Vocoder

Cada filtro pasa banda de la etapa de análisis extrae los valores de amplitud de la señal de entrada que corresponden a esa banda en particular. La forma de obtener las variaciones de amplitud en el tiempo es mediante un objeto denominado *peakamp~*. Este objeto lleva como argumento un tiempo expresado en milisegundos, y reporta cíclicamente la amplitud máxima registrada en ese período. Por este medio, es posible extraer la envolvente dinámica de un sonido para, por ejemplo, aplicársela a otro sonido.

Según se observa en G.5.10., los resultados de la etapa de análisis son enviados remotamente, banda por banda. Los filtros utilizados son los mismos que usamos en el ecualizador gráfico, dispuestos en cascada para mejorar el corte de frecuencias.

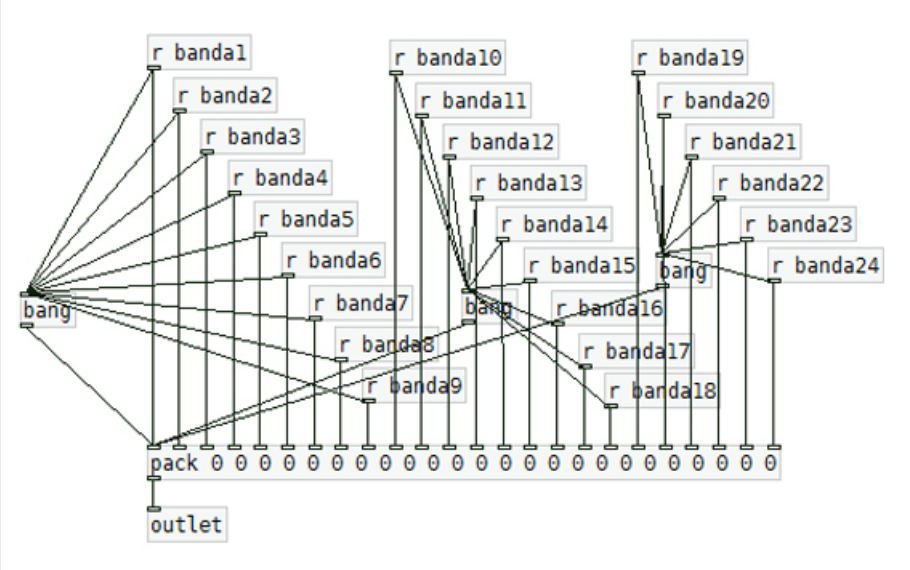
G.5.10. Banda de análisis y banda de resíntesis



Cada banda de la resíntesis filtra una porción de ruido blanco y le aplica la envolvente dinámica obtenida por un filtro idéntico en el análisis. Para aplicar la envolvente a la banda de ruido, simplemente la multiplicamos, pues le estamos modificando la amplitud a la señal, pero realizando previamente una rampa con *line~* para suavizar los cambios.

El valor de *Q* para la segunda mayor, calculada con la fórmula antes vista, es aproximadamente 8 (se ve como tercer argumento de los filtros, en el gráfico). No obstante, el usuario puede modificar este valor a voluntad y disminuir el ancho de cada banda, produciendo un efecto característico de este procesador.

G.5.11. *Subpatch* para crear listas con la amplitud de las bandas



Por último, para visualizar la amplitud de cada una de las 24 bandas, utilizamos un objeto *multi_slider* de la librería *mapping*. Para emplearlo creamos un objeto nuevo y escribimos *multi_slider*, seguido de un espacio y un número igual a la cantidad de *sliders* deseados.

Este objeto recibe una lista con los valores destinados a cada *slider*. Para crear la lista a partir de los valores de amplitud enviados por cada banda de análisis, utilizamos el objeto *pack*, que recibe los datos individuales y los agrupa en una lista (ver G.5.11.). Dado que *pack* solo envía la lista cuando ingresa un dato por el primer *inlet*, utilizamos objetos *bang* para forzar el envío, cualquiera sea la banda recibida.

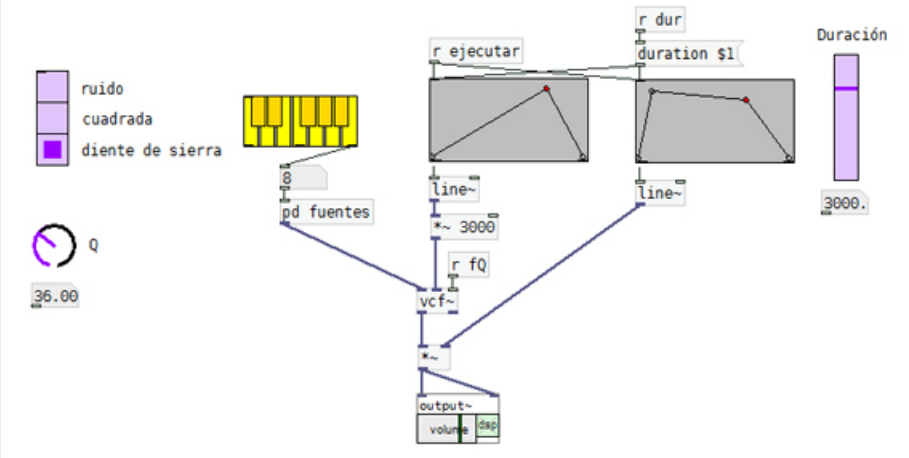
El *patch* "46-vocoder.pd" contiene la programación de G.5.9. Mediante la abstracción *sfplay~* ejecute archivos de sonido distintos, y emplee grabaciones de voz hablada y cantada. Puede, además, incorporar la entrada de un micrófono y probar el sonido de su propia voz.

5.5. Simulación de un sintetizador analógico

Al referirnos a las técnicas de síntesis del sonido, mencionamos la síntesis sustractiva, que se basa en la utilización de filtros para eliminar determinadas bandas de señales complejas, ricas en armónicos o parciales, o ruido blanco. Mediante esta técnica es posible modelar el sonido, sobre todo, teniendo en cuenta que los parámetros de los filtros pueden variarse dinámicamente, a través de procedimientos de control o de audio.

En la disposición modular de los sintetizadores analógicos podían identificarse tres partes principales: la generación de señales de audio, los filtros y la etapa de amplificación, que podían ser controlados por envolventes o, bien, por el mismo teclado. También solían contar con osciladores de baja frecuencia para producir trémolo y *vibrato*, modulación en anillo y reverberación.

G.5.12. Simulación de un sintetizador analógico

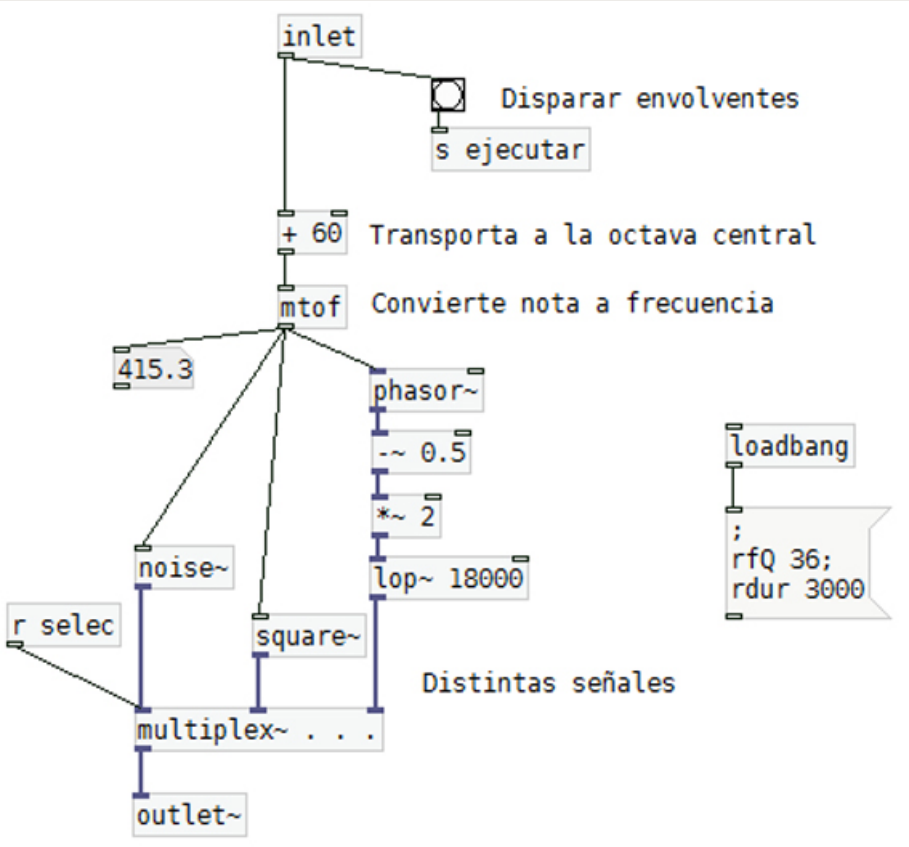


G.5.12. muestra un *patch* básico de simulación de un sintetizador analógico, donde es posible seleccionar la fuente (ruido, onda cuadrada, onda diente de sierra), establecer la frecuencia mediante un teclado, ingresar las señales a un filtro pasa banda controlado por una envolvente y aplicar una envolvente dinámica al resultado.

El objeto que simula el teclado es *moonlib/gamme*. Por su tercer *outlet* devuelve un número entre 0 y 11, que representa a la tecla presionada. A ese número le sumamos 60 para que las notas queden numeradas de acuerdo con la octava central del piano, según se especifica en la norma MIDI. Posteriormente, convertimos los números de nota (60 a 71) en frecuencias, empleando el objeto *mtof*, que realiza la conversión.

Si bien el *patch* es muy sencillo, sirve de punto de partida para seguir ampliándolo. Podríamos agregarle más osciladores y más filtros, o lograr incluso que fuera polifónico, es decir, que permitiera ejecutar varias notas simultáneamente.

G.5.13. *Subpatch* del sintetizador virtual



El *patch* "47-simil analógico.pd" contiene la programación de G.5.12.

Cetta, P. (2010), "Filtros Digitales. Primera parte", en: *Apuntes de Procesamiento Digital de Señales.* Sin publicar, Buenos Aires, pp. 1-8.

Actividad 7

a. Programar un *patch* que genere cuatro bandas angostas de ruido simultáneamente, con frecuencias centrales al azar, comprendidas cada una entre 200-1000 *Hz*, 1000-2000 *Hz*, 2000-3000 *Hz* y 3000-4000 *Hz*, y con envolventes breves, distintas para cada banda.

b. A partir de esto, generar automáticamente nubes de eventos, extrayendo dos bandas por el canal izquierdo y las restantes por el derecho. Los ataques de los eventos deben ser irregulares en el tiempo.

En el archivo "Respuesta Actividad 07.pd" encontrará una posible solución de este ejercicio. Compare los resultados que obtuvo con los del archivo y anote las diferencias.

Actividad 8

Considerando que para calcular dos frecuencias que formen un intervalo de quinta es preciso multiplicar a la menor de ellas por 1.5, y que el *Q* que corresponde a ese intervalo es aproximadamente 2, programar un ecualizador gráfico por quintas, similar al del ejemplo "45-ecualizador gráfico.pd".

6. Retardos

Objetivos

- Aprender los fundamentos de las líneas de retardo aplicadas al procesamiento digital de audio.
- Realizar programas de procesamiento de sonido y música en tiempo real que contemplen el uso de líneas de retardo.

6.1. Objetos relacionados con el retardo de señales

Las líneas de retardo permiten demorar la salida de señales de audio que ingresan a ellas durante un tiempo dado. Existen diversos procesos vinculados con retrasos de una señal y en esta Unidad vamos a estudiar algunos de ellos.

Al tratar los filtros digitales (Unidad 5), vimos que estaban constituidos por pequeños retardos, y que el resultado de su aplicación se relacionaba con una transformación tímbrica del sonido. Y aprendimos, además, que para la construcción de los filtros básicos recurríamos al objeto *delay~*, en el que expresábamos el tiempo de retardo en cantidad de muestras.

Pero ahora nos referiremos a tiempos de retardo mucho mayores, incluso de varios segundos, donde el efecto que se produce se relaciona más con fenómenos temporales que tímbricos.

Las líneas de retardo, cualquiera sea el lenguaje utilizado, suelen estar compuestas por dos objetos. El primero, almacena la señal entrante en la memoria de la computadora, mientras que el segundo, lee la información guardada, pasado cierto tiempo.

A fin de poder almacenar la señal que ingresa al objeto de escritura es preciso realizar una reserva previa de memoria, pues se trata de un recurso que es compartido con el sistema operativo y otras aplicaciones que se estén ejecutando. Los objetos de PD que realizan la escritura y lectura de datos para generar retardos se denominan *delwrite~* y *delread~*, respectivamente. Para operar en conjunto precisan tener un nombre en común, establecido como argumento. El objeto *delwrite~* puede llevar como segundo argumento la cantidad de memoria a reservar, expresada en milisegundos de señal a almacenar, mientras que *delread~*, el retardo a aplicar, también expresado en milisegundos.

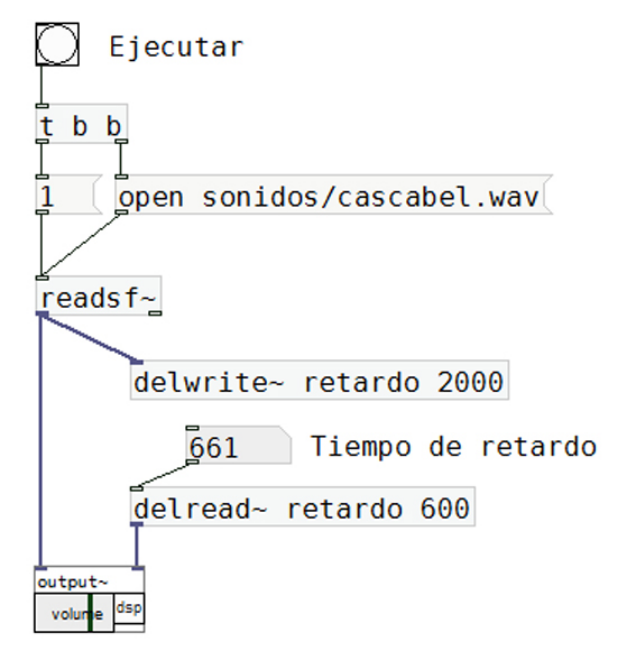
Si se desea modificar dinámicamente el tiempo de retardo, es decir, cambiarlo mientras la señal está siendo procesada, debe tenerse en cuenta que los objetos antes mencionados suelen producir discontinuidades en la forma de onda, que dan lugar a la producción de clics. En estos casos, el objeto *delread~* es reemplazado por un objeto de retardo variable, denominado *vd~*. El tiempo de retardo, utilizando este objeto, puede ser transformado mediante señales de audio, generadas a través de envolventes o por medio de osciladores.

6.2. Modelado físico del eco

G.6.1. muestra la utilización de los objetos *delwrite~* y *delread~*, que retrasan la señal almacenada en un archivo de audio. Nótese que no hay cables que los conecten entre sí, sino un término común como primer argumento que, en este caso, es la palabra "retardo".

Para *delwrite~* se reservó memoria suficiente para almacenar dos segundos de señal, por lo cual, el tiempo efectivo de retardo no debería superar el valor 2000.

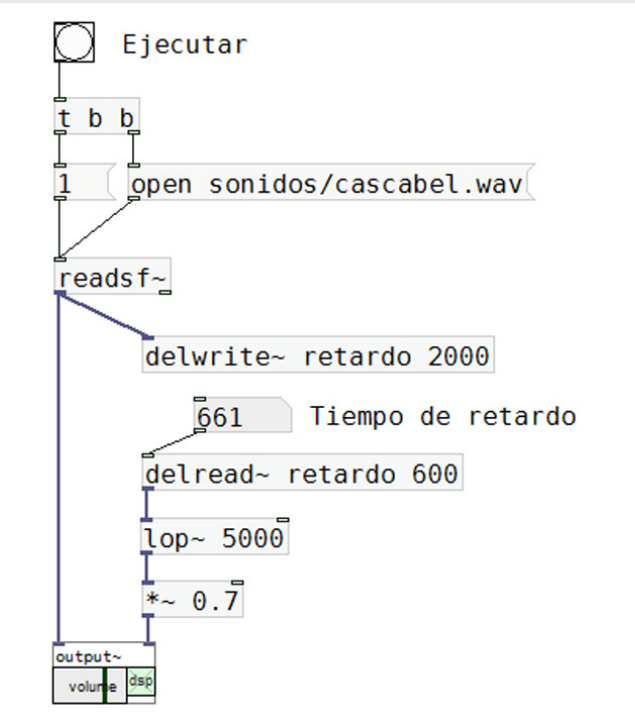
G.6.1. Modelado físico del eco



Una forma de mejorar el modelado físico del eco consiste en disminuir los agudos de la señal retrasada –para simular la absorción de las altas frecuencias que se produce en la reflexión del sonido– y disminuir también su amplitud –para recrear la caída de intensidad generada por la distancia.

Para llevar a cabo estas mejoras, agregamos al *patch* anterior un filtro pasa bajos (objeto *lop~*), y un multiplicador para atenuar la amplitud, tal como se observa en G.6.2.

G.6.2. Modelado físico del eco, mejorado



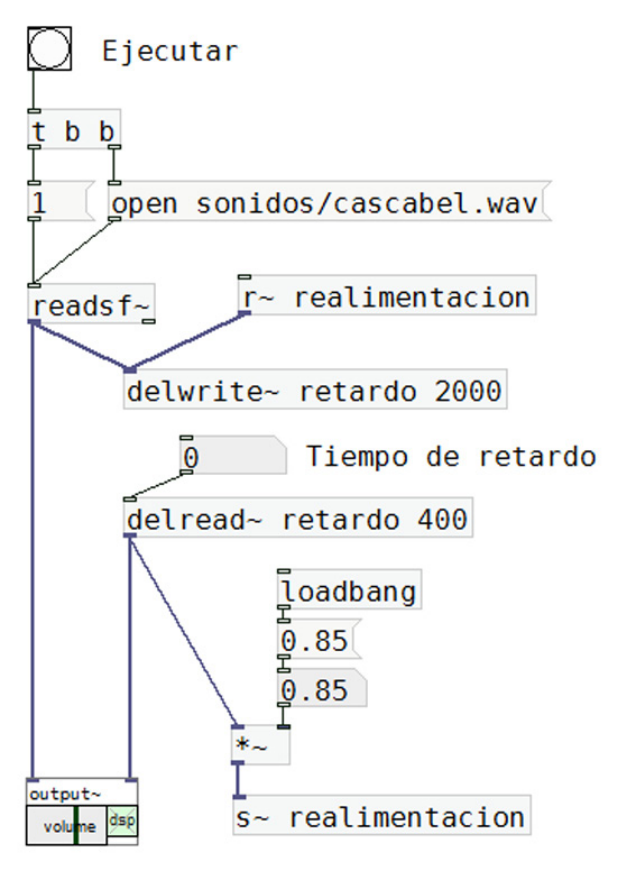
El *patch* "48-eco.pd" contiene la programación de G.6.1. Observe que el sonido directo (el que sale del objeto *readsf~*) se reproduce en el canal izquierdo, mientras que la señal retardada se reproduce en el derecho, generando así la sensación de un eco. La versión mejorada de G.6.2 se encuentra en el archivo "49-eco con filtro.pd"

6.2.1. Eco con realimentación

En el eco con realimentación, la señal retrasada reingresa una y otra vez en la línea de retardo, repitiéndose a intervalos iguales, equivalentes al tiempo de retardo. Si no existe atenuación, la señal se repite infinitamente, pero si es multiplicada por un valor inferior a 1, se atenúa gradualmente, hasta volverse nula. Suponiendo que ingresa una señal de amplitud 1, y que se multiplica a la señal retardada por 0.8, la primera vez que la escuchemos tendrá una amplitud igual a 1 (sonido directo), luego igual a 0.8, a continuación a 0.64 (0.8 por 0.8), después a 0.51 (0.64 por 0.8), etcétera.

En el ejemplo siguiente (G.6.3.) vemos la programación de un eco con realimentación, donde la señal retrasada es enviada remotamente al objeto *delwrite~*.

G.6.3. Eco con realimentación



Este *patch* seguramente nos resulta familiar, pues al referirnos a la síntesis por modelado físico utilizamos una línea de retardo con realimentación, para simular el comportamiento de una cuerda punteada. Como la duración de la señal de entrada era idéntica al tiempo de retardo, no se producía silencio entre las distintas versiones retardadas. En esa ocasión, además, incluimos un filtro pasa bajos básico para simular la caída de componentes agudas con el paso del tiempo.

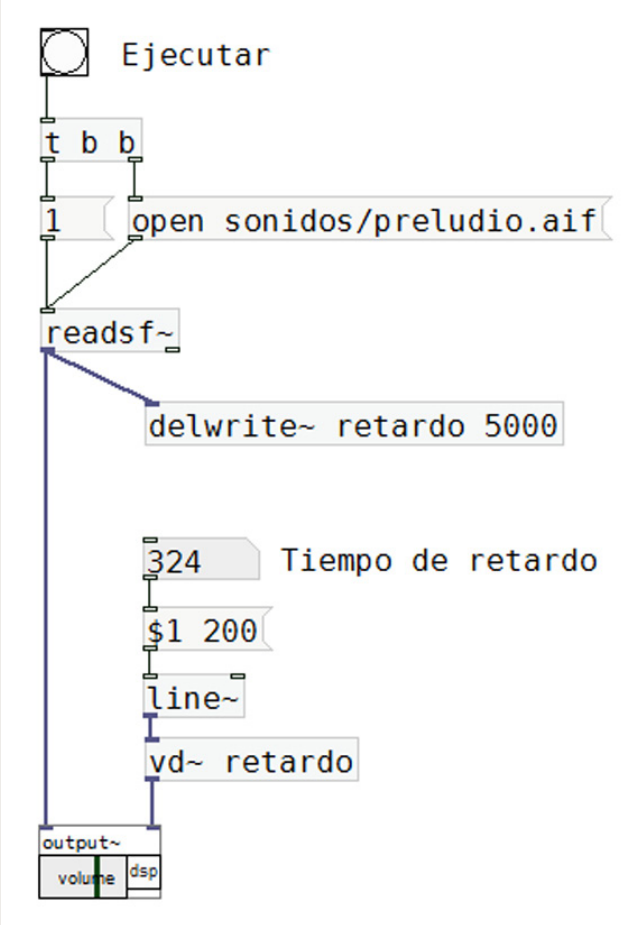
El *patch* "50-eco con realimentación.pd" contiene la programación de G.6.3. Puede editar el ejemplo y agregar un filtro pasa bajos en el circuito de realimentación, para simular la absorción de agudos producida por las sucesivas reflexiones.

6.3. Retardos variables

Según mencionamos antes, si deseamos modificar el tiempo de retardo durante el procesamiento de la señal, se torna necesario reemplazar el objeto de lectura *delread~* por su versión mejorada, el objeto *vd~*.

En el *patch* de G.6.4. vemos implementado un retardo variable. El tiempo de retardo es introducido en el objeto mediante una señal de audio generada por el objeto *line~*, que suaviza los cambios, con una rampa de 100 milisegundos.

G.6.4. Retardo variable

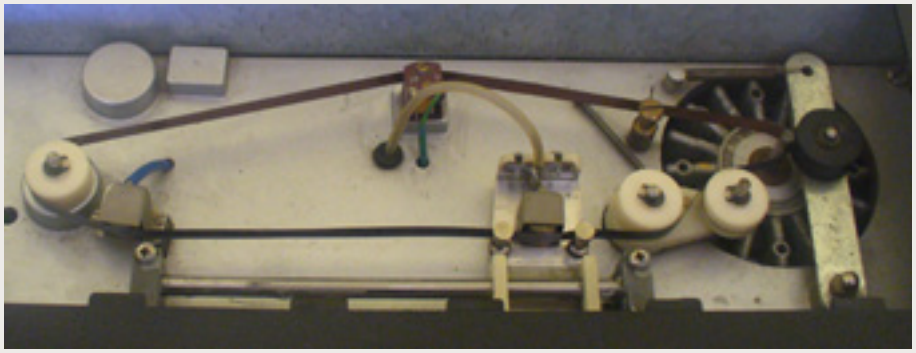


El *patch* "51-retardo variable.pd" contiene la programación de G.6.4. Escuche atentamente qué es lo que sucede cuando cambia el tiempo de retardo e intente describir la sensación que experimenta.

Habrá notado que al modificar el tiempo de retardo mientras escucha el sonido, se produce una transformación de la altura. Cuando aumentamos el tiempo de retardo, el sonido se hace más grave, y cuando disminuimos el retardo, el tiempo se vuelve más agudo. Ese fenómeno de transposición es inseparable de la modificación dinámica de un retardo, y si bien no podemos prescindir de él, puede hasta resultar útil en algunos casos.

La razón por la cual el sonido cambia de altura es fácilmente observable en los medios analógicos. Antiguamente, para realizar un retardo podía utilizarse un dispositivo que registrara la señal de audio en una cinta mediante un cabezal de grabación fijo, y reprodujera mediante otro cabezal de reproducción móvil, de manera tal que al separarlos una cierta distancia se produjera un retardo en la lectura de la información. La distancia entre cabezales estaba en relación directa con el tiempo de retardo. Pero ¿qué sucedía si se cambiaba la distancia mientras el sonido ocurría? Cambiaba la velocidad de lectura, pues aumentaba si se acercaba un cabezal al otro, o disminuía si se los alejaba, transportándose la altura del sonido en consecuencia. La figura siguiente muestra un dispositivo de este tipo, donde se aprecia el sistema móvil del cabezal de reproducción.

G.6.5. Mecanismo de una unidad de retardo a cinta

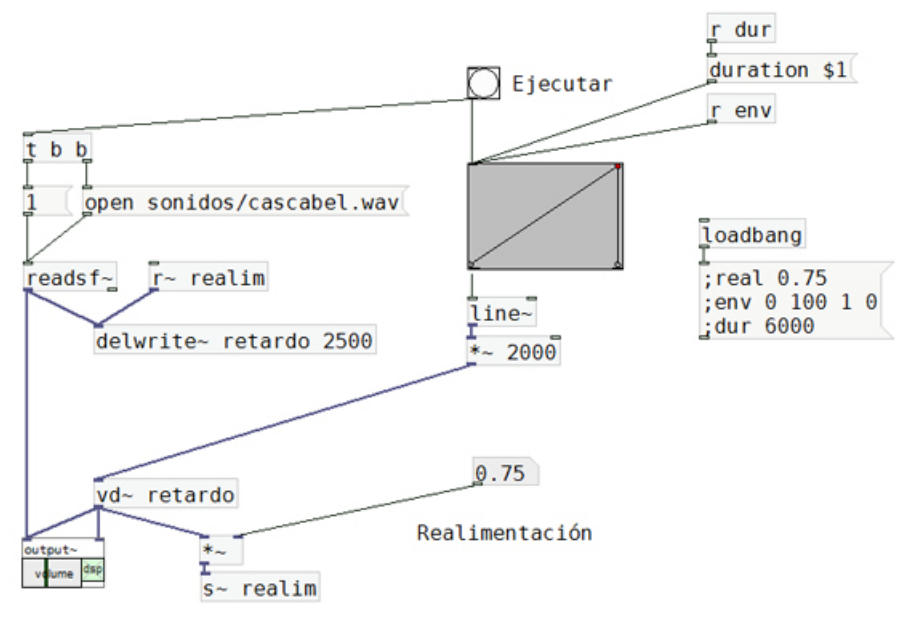


Un efecto similar podía lograrse cambiando la velocidad de la cinta, en lugar de mover el cabezal de reproducción; a mayor velocidad el tiempo para recorrer la distancia entre cabezales era menor, y viceversa.

6.3.1. Retardo variable con realimentación

En el circuito de retardo variable, al igual que ocurre con el fijo, es posible realimentar la línea de retardo y generar una repetición periódica de los eventos sonoros. G.6.6. muestra un *patch* de retardo variable con realimentación en el que el tiempo puede ser controlado mediante una envolvente.

G.6.6. Retardo variable con realimentación



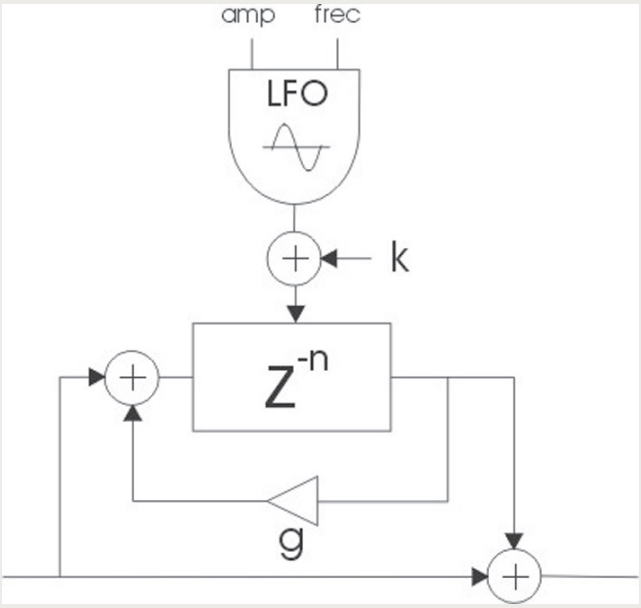
El *patch* "52-retardo variable con realimentación.pd" contiene la programación de G.6.6. Experimente dibujando distintas envolventes, relacionándolas auditivamente con la transposición de altura que producen.

6.4. *Flanger*

El *Flanger* es un efecto extensamente utilizado en la música popular. Proviene de un procedimiento utilizado en la década de 1960, que consiste en reproducir la misma música en dos bandejas giradiscos o grabadores a cinta. A uno de ellos se le cambia la velocidad periódicamente, y ese cambio de velocidad produce un ligero retraso periódico entre las copias, que da lugar a la generación del efecto.

G.6.7. muestra el diagrama de un *Flanger*. El tiempo de retardo variable se produce a través de un oscilador de baja frecuencia, cuya salida está escalada de acuerdo con el rango buscado. La línea de retardo (z^{-n}) retrasa la señal de entrada unos pocos milisegundos, y la versión original y la retardada se suman. También se observa que el efecto cuenta con realimentación, controlada por el factor *g* en el gráfico.

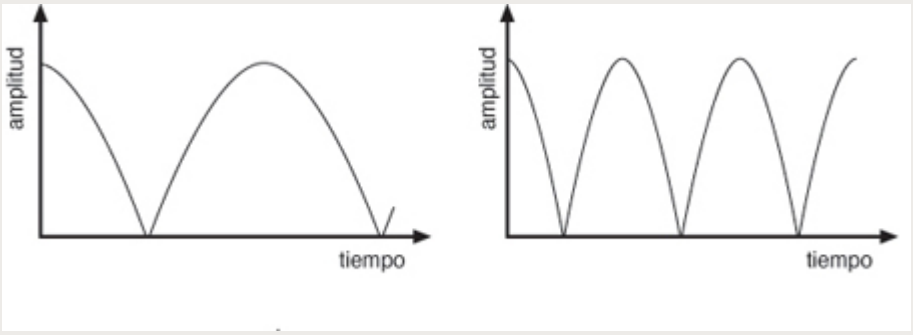
G.6.7. Diagrama de un *Flanger*



Si prescindimos del oscilador de baja frecuencia, y especificamos un valor fijo de retardo, el diagrama coincide con un tipo de filtro que se denomina filtro peine o filtro comb. Su nombre proviene de la forma particular de su respuesta en amplitud, que se asemeja a los dientes de un peine. A medida que el tiempo de retardo aumenta, crece la cantidad de picos de amplitud (ver G.6.8.).

Observe que el diagrama de G.6.7. (sin el oscilador) también representa un retardo con realimentación. Para tiempos de retardo altos el proceso produce ecos repetidos que se extinguen, pero para tiempos pequeños (menos de 40 ms), actúa sobre el espectro como un filtro.

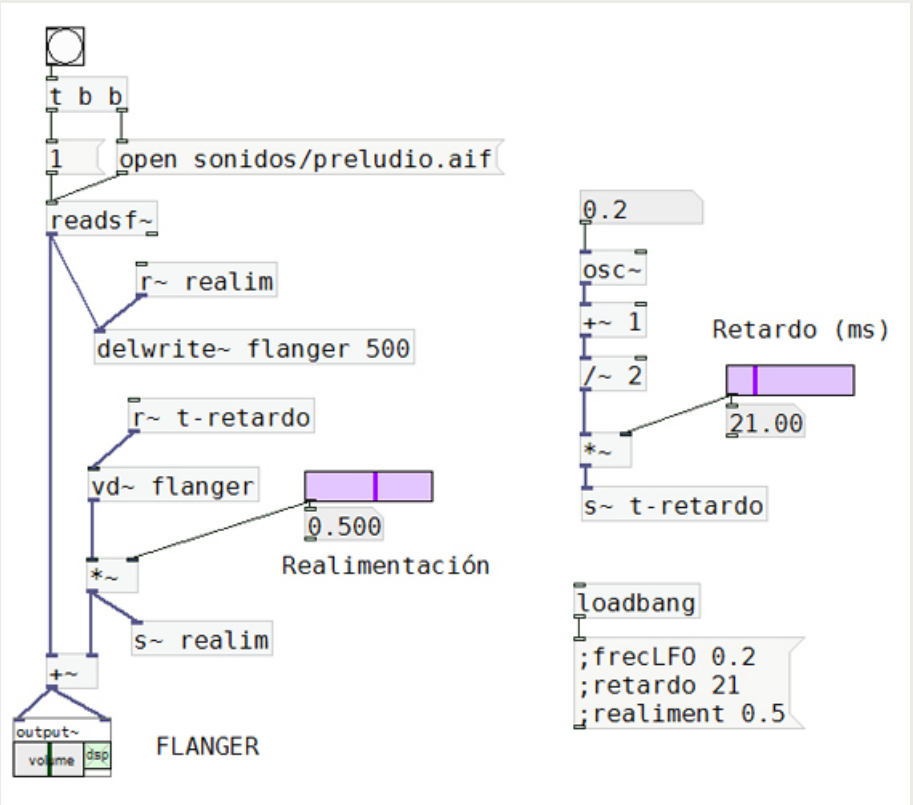
G.6.8. Respuesta en amplitud del filtro *comb*



Dado que el *Flanger* varía el tiempo de retardo mediante un oscilador, la respuesta en amplitud del filtro *comb* cambia cíclicamente –como un fuelle que se abre y se cierra– y produce un barrido sobre el espectro de la señal de entrada que resulta muy característico.

G.6.9. muestra la programación en Pure Data de un *Flanger*.

G.6.9. *Flanger*



El *patch* "53-Flanger.pd" contiene la programación de G.6.9. Puede incorporar una abstracción *sfplay~* al *patch* y experimentar el efecto sobre sus propios archivos de audio.

6.5. *Chorus*

Cuando un coro canta, aun al unísono, percibimos que se trata de diferentes voces debido a las pequeñas variaciones que, en mayor o menor grado, cada cantante imprime a la totalidad. Esas variaciones se deben a diferencias temporales en el ataque de cada evento sonoro, y a afinaciones e intensidades distintas en la interpretación de una misma nota.

Mediante el efecto denominado *Chorus* es posible realizar copias de una misma voz o instrumento, y aplicarle las diferencias mencionadas, con el objeto de multiplicarlas y que den la sensación de un coro o ensamble.

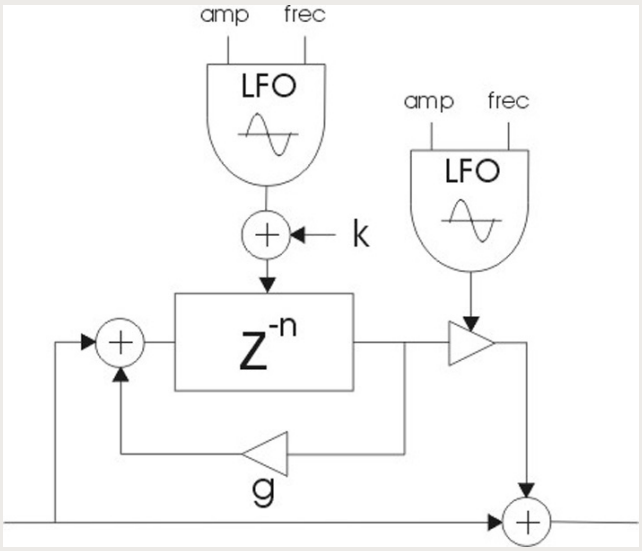
Para programar un *Chorus* podemos partir del *patch* utilizado para el *Flanger,* considerando que los tiempos de retardo deben ser mayores en este caso, pues no se intenta producir una transformación espectral, sino una verdadera desafinación. Según vimos, un cambio dinámico del tiempo de retardo trae aparejada una modificación en la altura del sonido, y como dijimos, en algunos procesos eso podría resultar beneficioso.

Por otra parte, las diferencias temporales en los ataques también aparecerán al usar la línea de retardo, dado que es lo que naturalmente ocurre al emplear retardos.

Solo resta imitar las variaciones de intensidad entre las voces, para lo cual podemos hacer una modulación en amplitud a baja frecuencia, de modo tal que la amplitud cambie lentamente en el tiempo.

La figura siguiente (G.6.10.) muestra un diagrama de un *Chorus*.

G.6.10. Diagrama de un *Chorus*

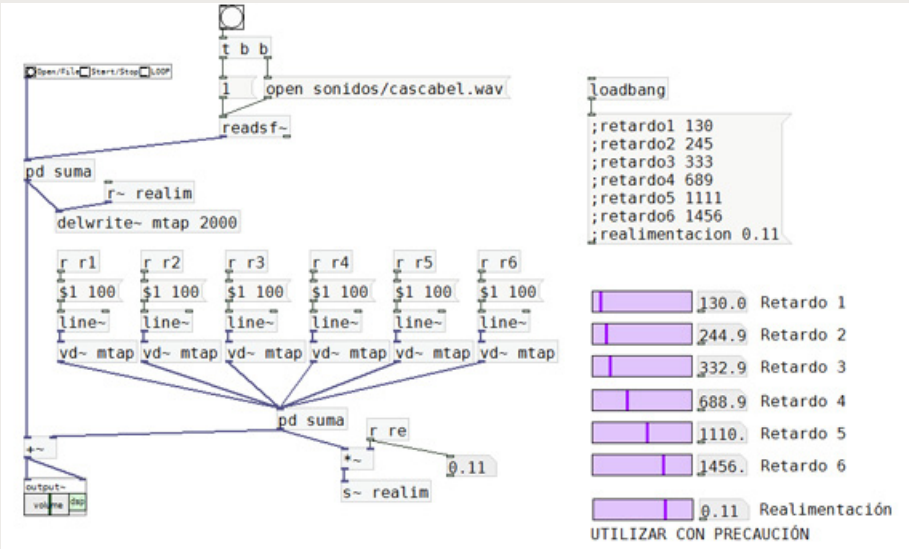


Lo que aquí vemos se denomina una "voz" del coro, y el efecto puede contar con 4, 8, o más copias de este proceso, cuyas salidas se suman.

6.6. *Multitap*

Un *multitap delay* es un efecto que consta de varias líneas de retardo sumadas y con realimentación, cada una con un tiempo de retardo diferente. Al programar un *multitap* entendemos por qué se emplean dos objetos, uno que escribe y otro que lee, en las líneas de retardo. Se debe a que vamos a utilizar un solo objeto para recibir la señal a procesar y guardarla en memoria (*delwrite~*), pero varios objetos van a leer esa información en tiempos distintos (*delread~*). Y esta división de funciones permite ahorrar una cantidad considerable de memoria, pues de no ser así, cada unidad de retardo debería guardar una copia de la señal de entrada.

G.6.11. *Multitap*



Según se observa en G.6.11., utilizamos para el ejemplo 6 retardos, cada uno a un tiempo diferente, donde los cambios están suavizados por objetos *line~*.

Cuando experimente el uso del *multitap* preste especial atención al factor de realimentación, dado que aun con valores muy bajos, la amplitud puede crecer por encima del valor unitario muy rápidamente, y producir intensidades dañinas para su oído o para los parlantes.

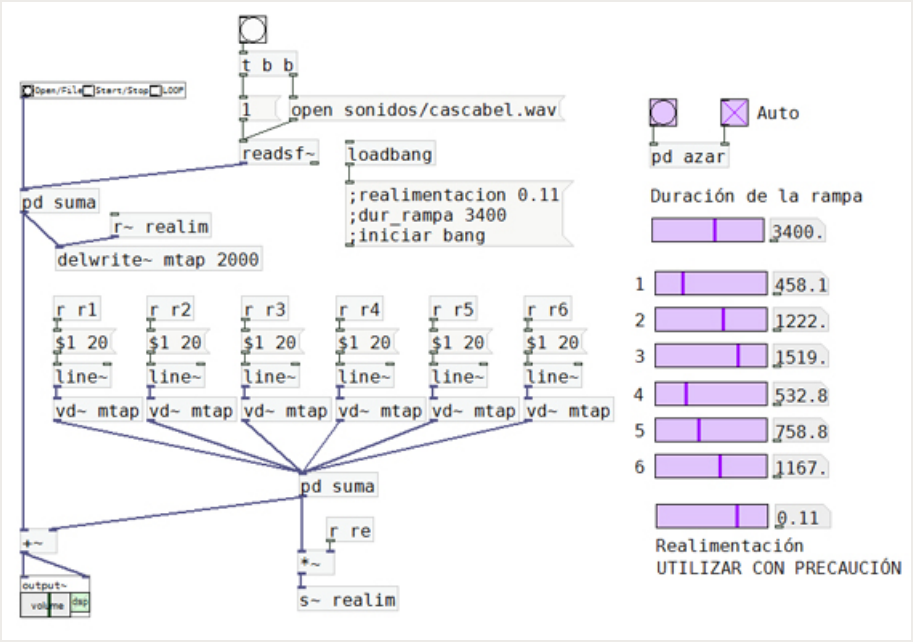
El *patch* "54-Multitap.pd" contiene la programación de G.6.11. Experimente sobre sus propios archivos de audio.

6.6.1. *Multitap* variable

Con base en el ejemplo anterior vamos a incorporar algunas variables aleatorias, no solo con el propósito de automatizar el proceso, sino de producir cambios más o menos lentos en los tiempos de retardo que van a traer aparejados cambios de altura diferentes para cada línea.

G.6.12. muestra la pantalla principal del *multitap* variable. Según puede verse, no hay cambios importantes, salvo que agregamos un *subpatch* para la generación de tiempos de retardo aleatorios, y un *slider* para determinar la duración de la rampa de transición entre un valor al azar y el siguiente, especificada en milisegundos

G.6.12. *Multitap* variable

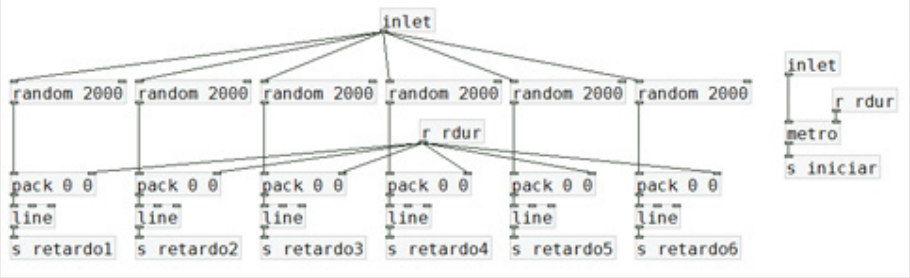


6.6.2. Generación de retardos aleatorios

El *subpatch* PD azar se observa en G.6.13. Generamos tiempos al azar para cada línea de retardo, comprendidos entre 0 y 2000, y mediante un objeto *line* obtenemos los valores intermedios entre el tiempo anterior y el nuevo, utilizando una rampa que los une en una determinada duración.

Sabemos que al objeto *line* deben ingresar dos valores en una lista: dónde quiero ir, en cuánto tiempo deseo llegar, respectivamente. Pero en este caso, ambos valores son variables, pues el tiempo de retardo se genera al azar, y el tiempo que dura la rampa es controlable por el usuario. Como no es posible cargar dos variables en un mensaje al mismo tiempo ($1 y $2) si no es a través de una lista, debemos recurrir a un objeto *pack*, que reciba los valores y genere esa lista con ellos. La lista así creada, puede ingresar al objeto *line* directamente.

G.6.13. *Multitap* variable. *Subpatch* de aleatoriedad



Observe, además, que ubicamos un objeto *metro*, cuyo período es el mismo tiempo que duran las rampas, para automatizar el cambio aleatorio de valores de retardo.

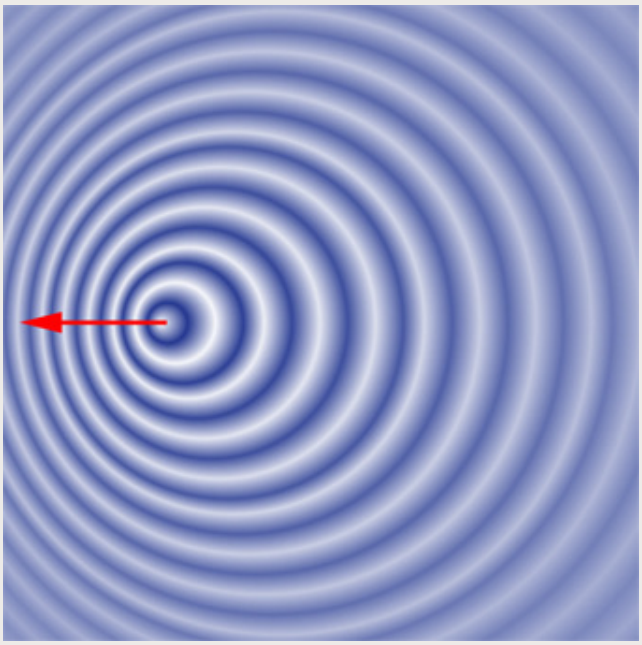
El *patch* "55-Multitap variable.pd" contiene la programación de G.6.12.

6.7. Simulación del efecto Doppler

Cuando una fuente sonora se desplaza en una determinada dirección, los frentes de onda que avanzan en el mismo sentido experimentan una compresión y, por lo tanto, se produce una disminución en la longitud de la onda. Este fenómeno se observa en G.6.14., donde también se aprecia que los frentes de onda que avanzan en sentido contrario se expanden, por lo cual, la longitud de onda disminuye en esta zona.

Dado que la frecuencia está en relación inversa con la longitud de onda, cuando esta última disminuye, la frecuencia aumenta, y viceversa. Por esta razón, un sujeto que observa a la fuente acercarse percibe una mayor altura en el sonido que emite, y una altura menor si la fuente se aleja.

G.6.14. Compresión del frente de onda en una fuente móvil

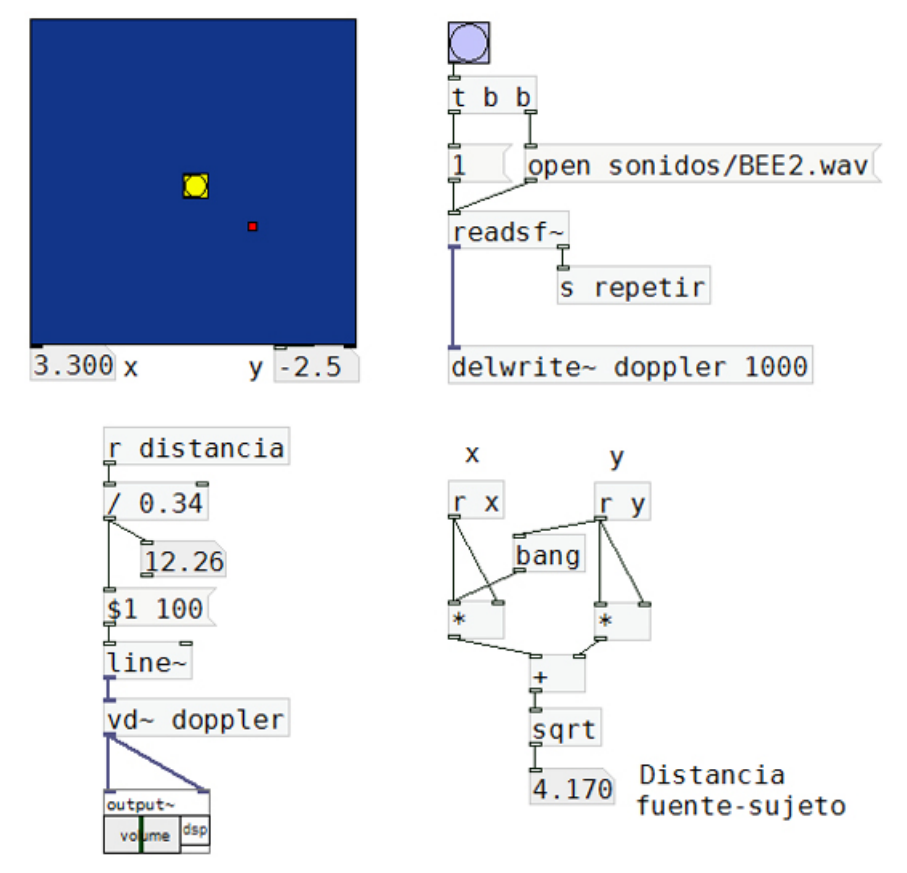


6.7.1. Implementación del efecto Doppler en PD

G.6.15. muestra un *patch* que implementa el efecto Doppler. Mediante el objeto *grid*, de la librería *unauthorized*, podemos movernos sobre una grilla con el puntero del mouse, mientras obtenemos las coordenadas de su posición. A partir de las coordenadas, calculamos la distancia al centro de la grilla utilizando el teorema de Pitágoras. Ese valor representa la distancia fuente-sujeto, medida en metros.

Posteriormente, calculamos el tiempo que tarda en llegar el sonido al sujeto, para cada posición de la fuente. Para ello, debemos dividir la distancia por la velocidad del sonido (340 m/s), y multiplicarla por 1000 para obtener el resultado en milisegundos. En el programa, a la distancia la dividimos directamente por 0.34, evitando así cuentas innecesarias. Finalmente, con el tiempo calculado, retardamos el sonido original de la fuente. Dado que el tiempo de retardo va cambiando según la posición de la fuente, el sonido retardado cambia de altura en consecuencia. Ese cambio coincide exactamente con la modificación de la altura que percibiríamos en una situación real, debida al efecto Doppler.

G.6.15. Efecto Doppler



El *patch* "56-efecto Doppler.pd" contiene la programación de G.6.15.

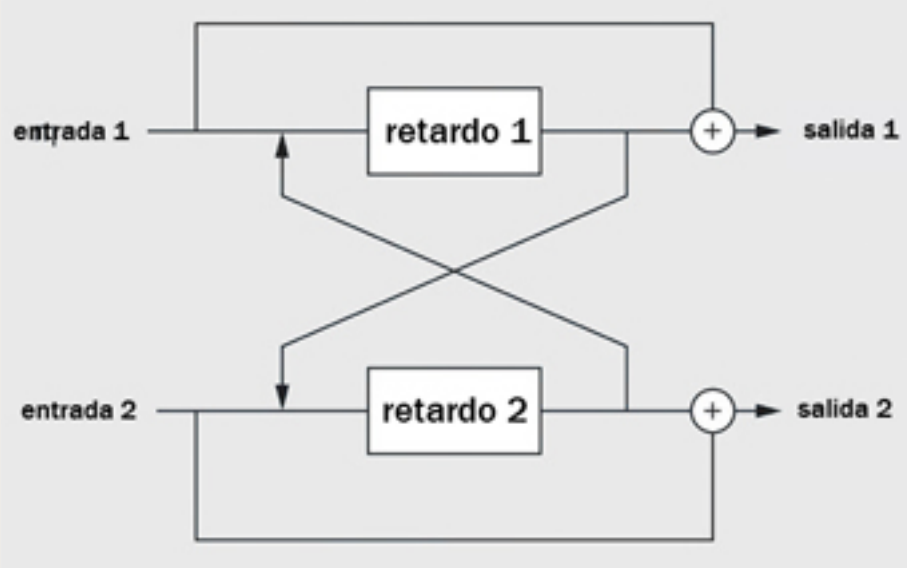
Gómez Gutiérrez, E. "Efectos digitales básicos", [EN LÍNEA]. En: *Apuntes de síntesis y procesamiento de sonido*. Departamento de Sonología. ESMUC. 2009. Disponible en: <http://www.dtic.upf.edu/~egomez/teaching/sintesi/SPS1/Tema10-EfectosDigitales.pdf> [Consulta: 25 de julio de 2013].

Romero Costas, M. (2011), "Técnicas de síntesis y procesamiento de sonido y su aplicación en tiempo real" en: *Revista de Investigación Multimedia* Nro. 3, IUNA, Buenos Aires, 69-83.

Actividad 9

G.6.16. muestra un efecto estereofónico denominado retardo "ping pong". Realice la programación en PD, de modo tal que ingresen simultáneamente las señales de dos archivos de sonido por las entradas 1 y 2.

G.6.16. Retardo "ping pong"



En el archivo "Respuesta Actividad 09.pd" encontrará la solución de este ejercicio. Compare los resultados que obtuvo con los del archivo y anote las diferencias.

Actividad 10

A partir de la explicación dada en el apartado 6.5. programe un efecto *Chorus*, de 4 voces.

En el archivo "Respuesta Actividad 10.pd" encontrará la solución de este ejercicio. Compare los resultados que obtuvo con los del archivo y anote las diferencias.

7. Transformaciones espectrales

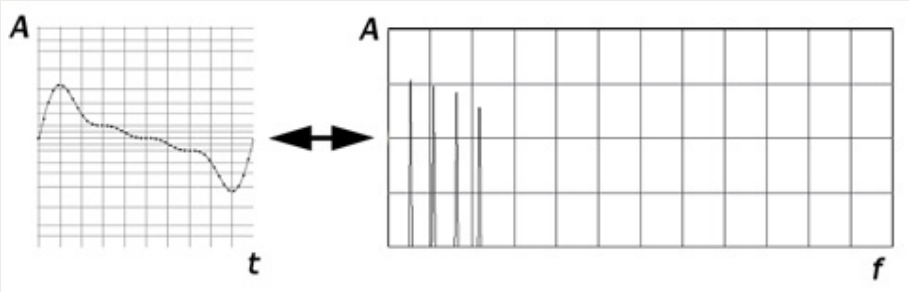
Objetivos

- Aprender los principios de la *Transformada Discreta de Fourier* y sus aplicaciones.
- Realizar programas de procesamiento de señales de audio en el dominio de la frecuencia.

7.1. Aspectos teóricos de la Transformada Discreta de Fourier

La Transformada Discreta de Fourier es una herramienta matemática que, aplicada al campo de conocimiento que aquí tratamos. Nos permite calcular el espectro de una forma de onda. Contrariamente, utilizando su inversa, podemos obtener la forma de onda si conocemos el espectro (ver figura G.7.1.).

G.7.1. Un ciclo de una forma de onda y su correspondiente espectro



Esta herramienta resulta sumamente útil, pues nos sirve para analizar espectralmente una señal de audio, y realizar procesos y transformaciones que no serían posibles operando directamente sobre las muestras de la forma de onda. Para entender el principio de funcionamiento de la Transformada Discreta de Fourier vamos a comenzar tratando de extraer la amplitud de un ciclo de una señal conocida, por medios matemáticos.

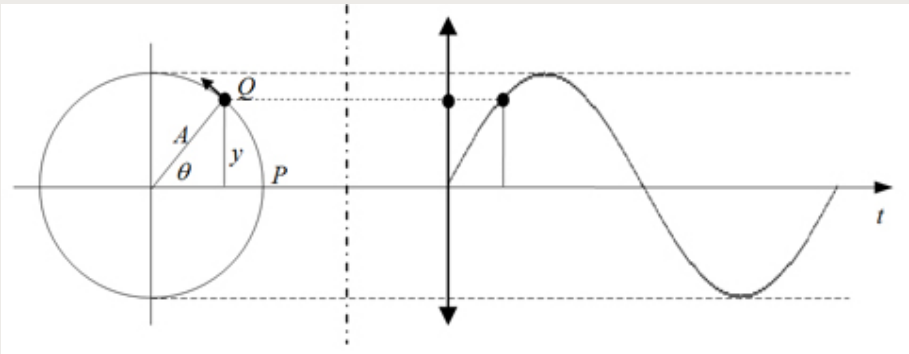
7.1.1. Relación entre el movimiento oscilatorio armónico y el movimiento circular uniforme

Sabemos que experimentamos la sensación sonora a partir de movimientos oscilatorios, y que la complejidad del movimiento determina la complejidad del sonido escuchado. El sonido más simple es el sonido puro, y lo determina un movimiento oscilatorio simple, que es un movimiento de ida y vuelta alrededor de un punto de equilibrio (similar al que realiza un péndulo).

Para poder predecir la amplitud que tendrá un movimiento oscilatorio simple en un momento determinado, podemos recurrir a un sistema análogo, formado por un móvil que se desplaza en una trayectoria circular a velocidad constante.

Si ubicamos un objeto en el borde de una bandeja giradiscos, por ejemplo, e iluminamos sobre un costado, podremos observar la proyección (sombra) del objeto sobre la pared. El movimiento oscilatorio proyectado, desarrollado en el tiempo, puede representarse a través de una sinusoide (ver G.7.2.).

G.7.2. Relación entre el movimiento oscilatorio simple y el circular uniforme



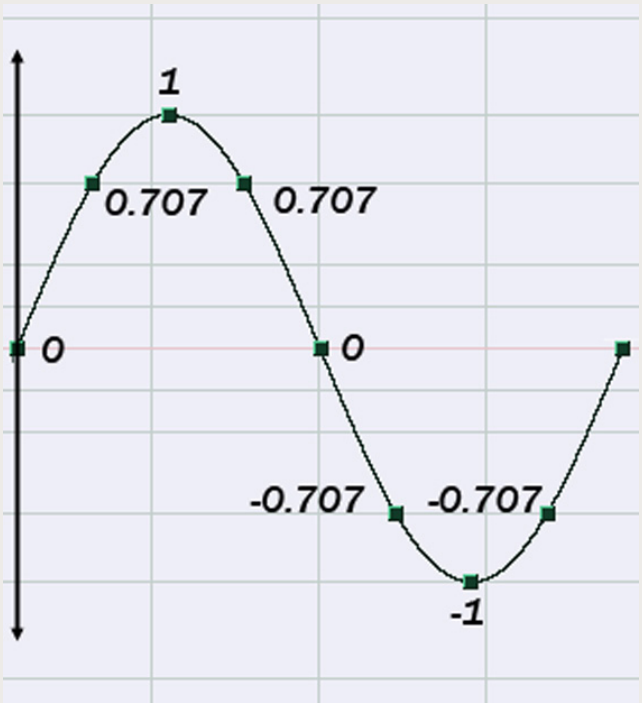
Esta analogía resulta útil para predecir la amplitud del movimiento oscilatorio en un momento determinado. Para conocerlo, basta calcular el valor de *y*, utilizando la definición de la función seno.

$$\sin \theta = y/A$$
$$y = A.\ \sin \theta$$

Debemos tener presente que en los medios digitales el tiempo no es una variable continua, sino discreta, y está representado a través de muestras.

Si consideramos, por ejemplo, una señal sinusoidal periódica x(n) con amplitud máxima 1, cuyo ciclo está representado por 8 muestras, el valor de amplitud de cada muestra es 0, 0.707, 1, 0.707, 0, -0.707, -1 y 0.707 (G.7.3.).

G.7.3. Ciclo de una sinusoide representada con 8 muestras



Para 0º en el sistema análogo, *y* vale 0 de amplitud, para 45º vale 0.707, para 90º vale 1, etc. Pero para aquellos ángulos cuyo seno vale 0.707, o -0.707, vamos a utilizar una expresión más adecuada, debido a que el seno de 45º tiene infinitos decimales.

$$\sin 45^\circ = 0.707\ldots = 2/2 = 1/2$$

7.1.2. Obtención de la amplitud de un ciclo de una sinusoide

Supongamos ahora que no conocemos la amplitud máxima de nuestra sinusoide, pero deseamos averiguarla realizando un cálculo sobre sus muestras que podríamos representar así:

$$x(n) = 0, \frac{1}{2}A, A, \frac{1}{2}A, 0, -\frac{1}{2}A, -A, -\frac{1}{2}A$$

Podríamos comenzar por sumarlas, para ver si obtenemos algún resultado.

$$0 + \frac{1}{2}A + A + \frac{1}{2}A + 0 - \frac{1}{2}A - A - \frac{1}{2}A = 0$$

Pero vemos que la suma da 0, lo cual es bastante lógico debido a que ambos hemiciclos son opuestos, tienen forma similar, pero signo contrario.
Veamos qué sucede si, en cambio, elevamos el valor de cada muestra al cuadrado y luego las sumamos:

$$0 + \frac{A}{2} + A + \frac{A}{2} + 0 + \frac{A}{2} + A + \frac{A}{2} = 4A = N\frac{A}{2}$$

Al sumar, notamos que nos da 4 veces la amplitud, lo cual parece incorrecto. Pero si realizamos la misma operación sobre otras sinusoides, con distinta cantidad de muestras por ciclo, vemos que la amplitud siempre está multiplicada por el número de muestras *(N)*, y divido por 2. De esta manera, hemos podido extraer la amplitud de una sinusoide por medios puramente matemáticos.

7.1.3. El principio de funcionamiento de la Transformada de Fourier

Una vez que se logró obtener la amplitud de una sinusoide, partiendo del valor de sus muestras, podemos hallar el modo de calcular la amplitud de las componentes sinusoidales que se encuentran en una señal compleja.

Para lograr este objetivo, la Transformada de Fourier multiplica las muestras de la señal a analizar por las muestras de sinusoides de distintas frecuencias. Si la sinusoide por la que multiplico se encuentra dentro de la señal compleja, los valores de las muestras se elevan al cuadrado, y al sumarlos dan un número distinto de cero.

Se trata de un procedimiento de prueba y error, en el cual voy buscando, a través de un repertorio de sinusoides, cuál de ellas puede estar contenida en la forma de onda a analizar.

Puede demostrarse que si multiplicamos valor por valor de dos sinusoides de distinta frecuencia, y de igual número de muestras, y las sumamos, el resultado da cero. Solo en los casos en que las sinusoides multiplicadas y sumadas tienen igual frecuencia, es que podemos extraer un valor de amplitud para esa frecuencia.

7.1.4. Frecuencia fundamental de análisis

Si el procedimiento se basa en probar sinusoides de distinta frecuencia cabe preguntarse por dónde comenzar, cuántas utilizar, a qué frecuencia finalizar.
De acuerdo con lo que Fourier expresa, una onda compleja infinitamente periódica, siempre puede ser descompuesta en sinusoides de frecuencias cuyos valores son múltiplos. Por ello, si analizamos un ciclo de una señal compleja de *N* muestras, solamente deberemos multiplicarla por sinusoides de frecuencias múltiplos, de las cuales, la menor es llamada "fundamental de análisis".
Para averiguar la frecuencia de la fundamental de análisis, simplemente debemos dividir la cantidad de muestras que entran en un segundo (*R* o frecuencia de muestreo), por la cantidad de muestras contenidas en un ciclo de la onda a analizar *(N)*. Como el resto de las sinusoides son múltiplos de ella, podemos expresar la frecuencia de cada componente del siguiente modo:

$$f_k \text{ para } k = \frac{kR}{N} \; 0, 1, 2, \ldots, N\text{-}1$$

donde f_k es la frecuencia del armónico k (k = 0, 1, 2, 3, etc.), R es la frecuencia de muestreo y N es la cantidad de muestras a analizar.
La transformada multiplica la señal a analizar por la frecuencia fundamental de análisis y continúa con sus armónicos, hasta llegar a la frecuencia de muestreo.
Si bien estamos acostumbrados a ver solamente la parte positiva del espectro, en rigor, la Transformada calcula la parte positiva hasta una frecuencia igual a la mitad de la frecuencia de muestreo y luego continúa en forma decreciente con la parte negativa, que es simétrica a la positiva y, por lo tanto, redundante a los efectos prácticos.
Así como la forma de onda está muestreada en el tiempo, el espectro también lo está, pero en frecuencia. La cantidad de muestras que posee el espectro es equivalente a la cantidad de muestras de la señal analizada, pero solo la mitad se encuentra en el eje positivo de frecuencias.

7.1.5. La sinusoide compleja

Según vimos, la Transformada multiplica la señal a analizar por distintas sinusoides. Pero, en realidad, ocurre que la señal se multiplica por una sinusoide y luego por una cosinusoide de igual frecuencia. A esa combinación se la denomina "sinusoide compleja", y el nombre se relaciona obviamente con los números complejos. Al resultado obtenido al multiplicar por la cosinusoide se lo denomina "parte real" *(a)*, y a la multiplicación por la sinusoide, "parte imaginaria" *(b)*.

7.1.6. Transformada Rápida de Fourier

La Transformada Rápida de Fourier (o FFT por *Fast Fourier Transform*) es un algoritmo computacional que implementa de forma veloz la Transformada Discreta de Fourier.
El único requerimiento que presenta la FFT es que el número de muestras a analizar *(N)* debe ser potencia de 2.
El objeto de PD que implementa la Transformada Rápida devuelve un par de valores de amplitud para cada frecuencia, los cuales resultan de multiplicar a la señal de entrada por la cosinusoide y la sinusoide.
A partir de esos valores podemos determinar la amplitud y la fase de cada componente. Dado que no vamos a realizar procesos que empleen la fase de las componentes, no nos vamos a referir al modo de extraerla. Pero en el caso de las amplitudes, dados *a* y *b* (parte real e imaginaria), debemos aplicar la siguiente fórmula, que proviene del teorema de Pitágoras:

$$A_k = a_k^2 + b_k^2$$

donde Ak es la amplitud del armónico k de la fundamental de análisis, ak es la parte real de la amplitud del armónico *k*, y b_k es la parte imaginaria.

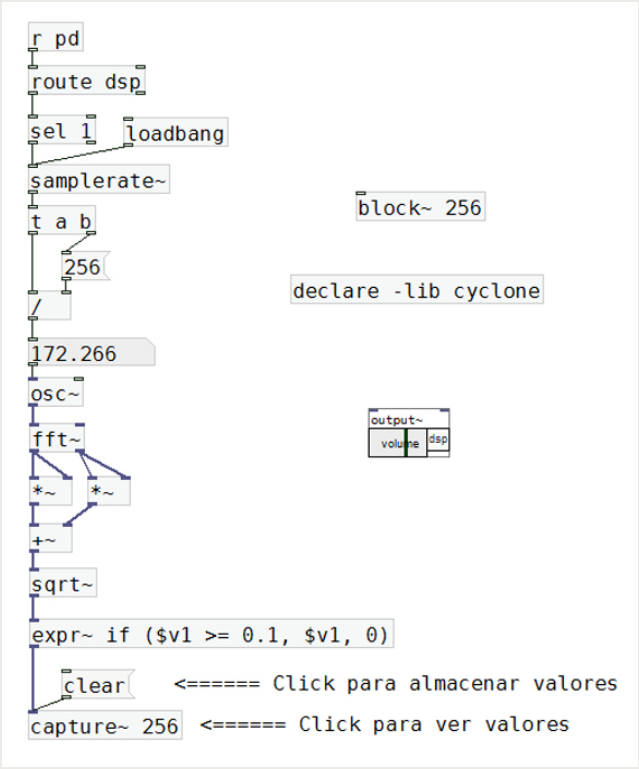
7.2. Comprobaciones teóricas

De acuerdo con lo visto hasta el momento, el objeto de PD que implementa la Transformada Rápida de Fourier, va a devolver *N* pares de valores (*a* y *b*), de los cuales la primera mitad corresponde a la parte positiva del espectro.
Los pares contienen la amplitud y la fase de las componentes.

Para extraer la amplitud debo aplicar el teorema de Pitágoras, y la amplitud que obtendré estará escalada por N/2; lo cual significa que si analizo una componente de amplitud 1, de una señal de 256 muestras, el algoritmo de la FFT indicará una amplitud 128.

Para conocer la frecuencia de las componentes debo dividir *R* y *N* y multiplicar por el número de par. Este último es un dato fácil de conocer, ya que los pares salen ordenados de 0 hasta *N* - 1, y puedo contarlos.
Todas las frecuencias de las componentes calculadas son múltiplos de la frecuencia menor, llamada "fundamental de análisis".
Entendido esto, estamos en condiciones de comprobar los fundamentos teóricos de la FFT, mediante el *patch* de G.7.4., que analizaremos a continuación.

G.7.4. Comprobación de la FFT



El *patch* "57-comprobación FFT.pd" contiene la programación de G.7.5.

El programa comienza recibiendo un mensaje que PD envía cada vez que se enciende o apaga el procesamiento de audio, y que es "dsp 1" o "dsp 0". El objeto *route*, al tener inscripto el argumento dsp, reconoce el término y devuelve lo que sigue en el mensaje, o sea, el 1 o el 0. De estos dos posible valores, el objeto *select* elige el 1 y envía un *bang*. Vale decir que lo realizado hasta el momento nos sirve para detectar cuándo el usuario del programa enciende el procesamiento de audio.

El mensaje *bang* que ocurre al encender el procesamiento ingresa a un objeto *samplerate~*, que reporta la frecuencia de muestreo *(R)*. Este procedimiento suele utilizarse cuando el proceso depende de la frecuencia de muestreo, pues, si el usuario cambia esa variable, los resultados pueden ser impredecibles.

Por otra parte, debemos informar a PD la cantidad de muestras que queremos analizar o, mejor dicho, el tamaño de la ventana de análisis; pues, una vez que termine con un bloque de datos seguirá analizando el resto de la señal que ingresa al objeto. Para determinar *N*, o sea el tamaño de la ventana o bloque, utilizamos el objeto *block~*.

Conocidos ahora *R* y *N*, podremos calcular la frecuencia fundamental de análisis, realizando *R/N*.

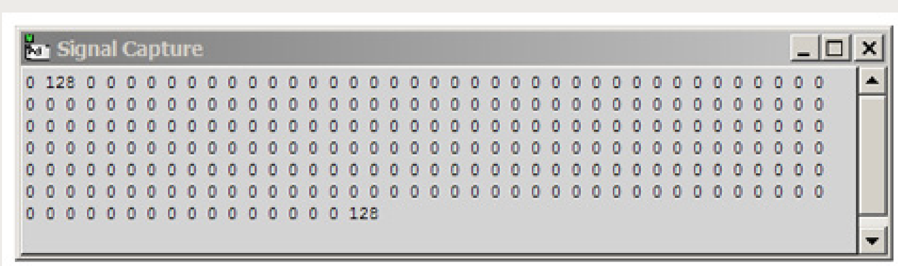
En la figura se observa que la frecuencia fundamental de análisis para una frecuencia de muestreo de 44.100 *Hz*, y una ventana de 256 muestras, es de 172.266 *Hz*. Esta frecuencia será la primera que intentará encontrar en la señal a analizar, siguiendo por sus armónicos (344.5, 689, 1378.1, etcétera).

Según se observa, con el oscilador ingresando al objeto *fft~* queremos comprobar si detecta la amplitud de una simple sinusoide cuya frecuencia es la fundamental de análisis.

A los pares de datos a y b que salen del objeto *fft~* les aplicamos el teorema de Pitágoras, es decir, los elevamos al cuadrado, los sumamos, y hallamos la raíz cuadrada con *sqrt~*.

Finalmente, guardamos los primeros 256 valores de amplitud en una tabla, mediante el objeto *capture~* de la librería Cyclone. Cada vez que enviamos el mensaje *clear* se rellena la tabla y podemos ver los resultados haciendo clic sobre el objeto.

G.7.5. Ventana de captura de datos



Observando la tabla de G.7.5. vemos que se inicia con un 0, que corresponde a la amplitud para 0 *Hz*, que en nuestro caso no nos interesa.

El segundo número es la amplitud de la frecuencia fundamental de análisis. Da el valor 128, que debe interpretarse como la amplitud multiplicada por *N* y dividida por 2, o sea 1.

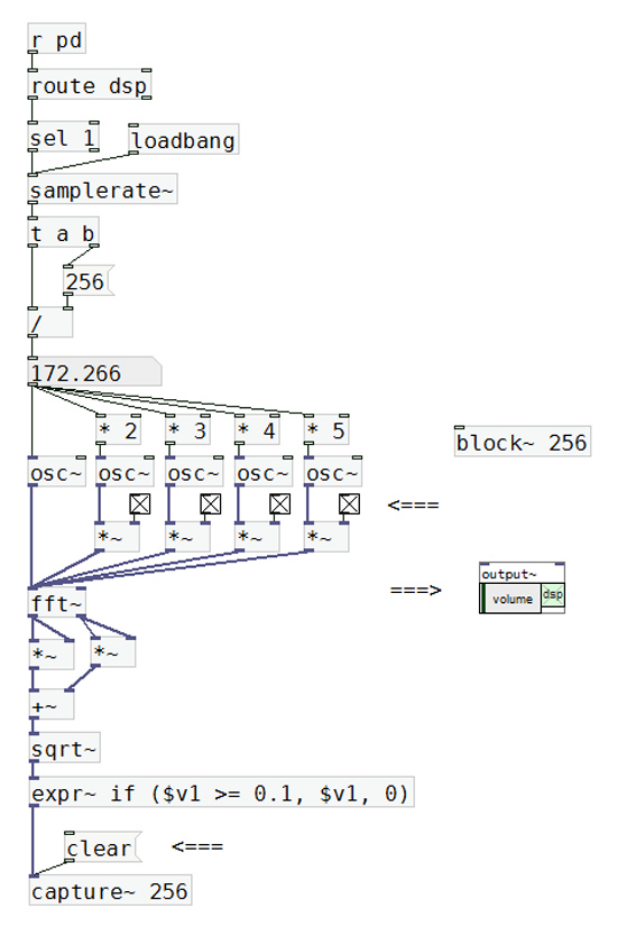
Los valores restantes dan 0, lo cual es correcto pues no hay otras componentes presentes en la señal, salvo el 128 que aparece al final de la tabla, que es la versión negativa (en espejo) de la frecuencia fundamental de análisis. Según dijimos antes, de los 256 números presentes nos interesan particularmente los 128 primeros, que reflejan la parte positiva del espectro.

Antes del objeto *capture~* observamos un objeto denominado *expr~*, que permite realizar cálculos sobre las muestras de una señal de audio. Mediante un condicional *if*, preguntamos si la muestra que ingresa, almacenada en la variable *$v1* es mayor o igual a 0,1. Si la condición se cumple, la enviamos a la salida, pero si no se cumple, enviamos un 0 en su lugar. La función de esta parte del programa es eliminar números extremadamente pequeños con muchos decimales que, a causa de cuestiones de precisión en el cálculo del algoritmo de la FFT, ocupan espacio innecesario en la tabla y complican su lectura.

7.2.1. Análisis de una señal compleja

Analizaremos ahora un *patch* similar en el que ingresaremos una señal compleja, sintetizada a partir de los cinco primeros armónicos. Puede verse la programación en G.7.6.
El objeto *fft~* recibe las muestras de una señal periódica, cuya frecuencia fundamental coincide con la frecuencia fundamental de análisis.
Todas las componentes tienen la misma amplitud y su salida no está atenuada, por lo cual, el valor es 1.

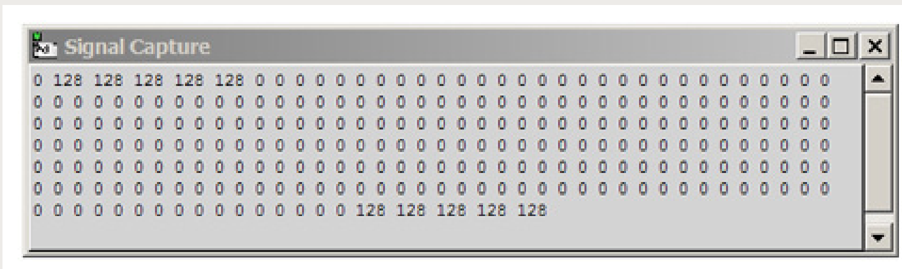
G.7.6. Análisis de una señal compleja



El *patch* "58-comprobación FFT con armónicos.pd" contiene la programación de G.7.6.

En la tabla generada por *capture~* podemos apreciar, luego de la amplitud de 0 *Hz*, los cinco valores iguales que corresponden a los cinco primeros armónicos de la fundamental de análisis, y la simetría de la parte negativa del espectro (G.7.7.).

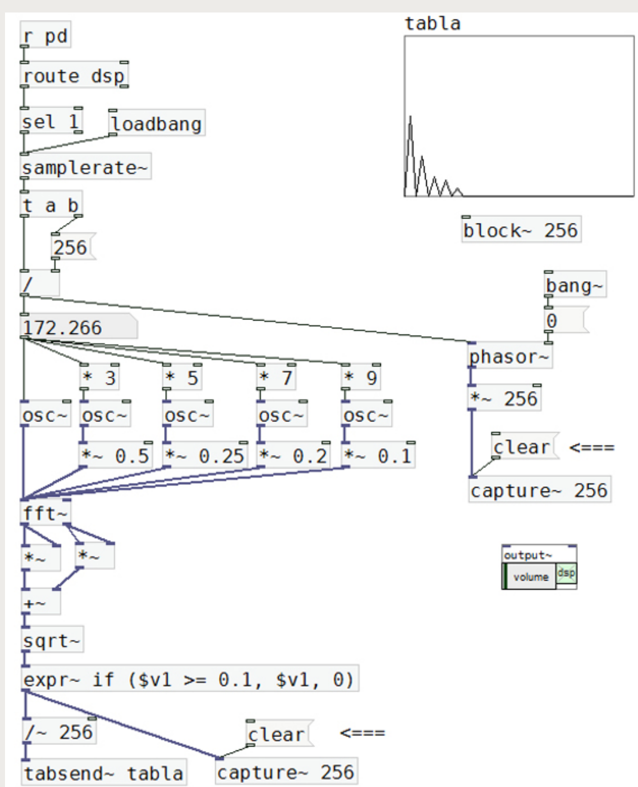
G.7.7. Ventana de captura de datos



7.2.2. Índices de frecuencia

Finalmente, para concluir con esta comprobación de resultados de la Transformada, vamos a obtener el número de armónico que corresponde a cada par de valores calculado por el objeto *fft~*.
G.7.8. muestra el *patch* de análisis, modificado para sintetizar los cinco primeros armónicos impares con amplitudes decrecientes, y una representación gráfica muy simple del espectro.

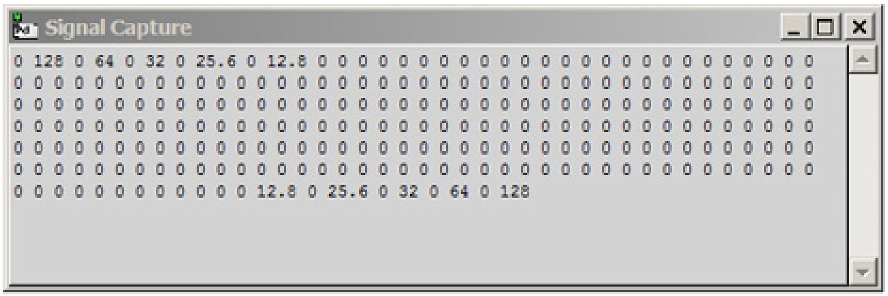
G.7.8. Número de armónico de las componentes analizadas



El *patch* “59-FFT y número de armónico.pd” contiene la programación de G.7.8.

En la tabla de captura de las amplitudes se observa que los resultados obtenidos son coherentes con los datos de la señal sintetizada.

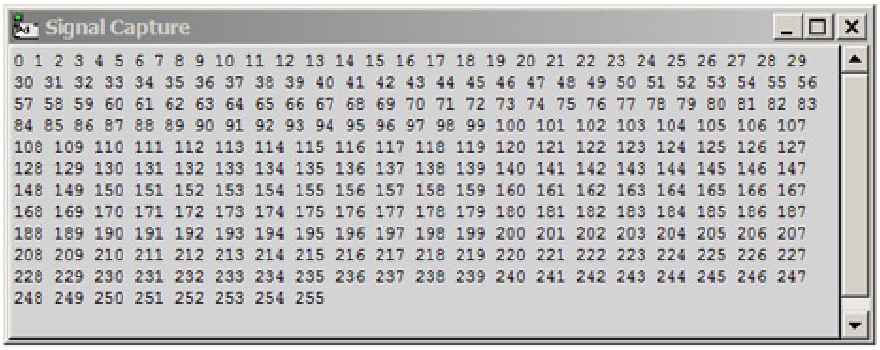
G.7.9. Ventana de captura de datos



Para generar los números de armónico utilizamos un objeto *phasor~*, a la frecuencia fundamental de análisis. Como sabemos, este objeto produce una rampa cíclica entre 0 y 1, pero si multiplicamos su salida por N, podremos obtener números enteros entre 0 y ese valor.

Estos números deben estar sincronizados con los pares de valores que salen de la FFT; por lo cual, utilizamos un objeto *bang~* que emite un mensaje *bang* cada vez que da comienzo un nuevo bloque de la Transformada. Este mensaje dispara, a su vez, un mensaje con un 0, que ingresa al inlet de fase del objeto *phasor~*, forzando el inicio de la rampa en alineación con el bloque (ver la tabla en G.7.10.).

G.7.10. Ventana de captura de índices de frecuencia



7.2.3. Interpretación de los resultados de la FFT

Analizados varios ejemplos, cabe preguntarse qué sucede si ingresamos a la FFT una sinusoide cuya frecuencia no es múltiplo de la fundamental de análisis.

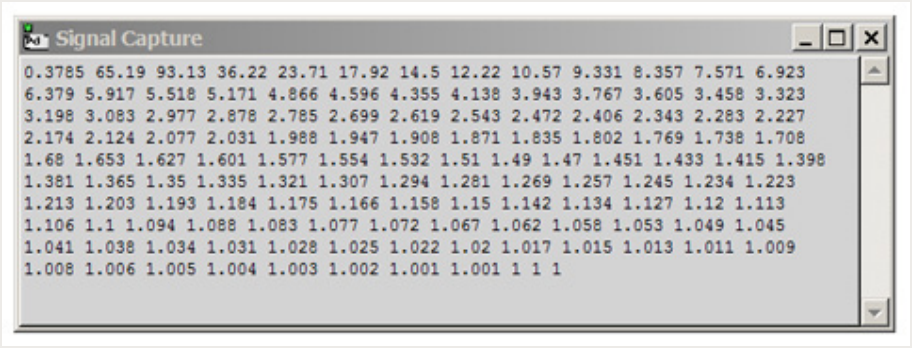
Supongamos que la señal de entrada es una sinusoide cuya frecuencia se encuentra justo entre la fundamental de análisis y su segundo armónico. Eso significa que en N muestras, la sinusoide no entró un número entero de veces, sino una vez y media.

La Transformada de Fourier espera encontrar un solo ciclo de una señal infinitamente periódica, y en este caso va a devolver varios armónicos, pues la forma de un ciclo y medio no es la que corresponde a una sinusoide, sino a una onda compleja.

Aún sabiendo que la señal ingresada como dato es una sinusoide, el análisis arroja resultados distintos al esperado, registrando amplitud en las frecuencias vecinas. Por esta razón, notamos que los resultados de la Transformada de Fourier requieren de cierta interpretación.

Empleando el *patch* de G.7.4. generamos una señal sinusoidal de 258.4 *Hz*, o sea 1.5 veces mayor a la fundamental de análisis. Los resultados obtenidos se aprecian en G.7.11. Vemos amplitud en 0 *Hz*, en la fundamental de análisis (65.19), más en el segundo armónico (93.13) y también en los restantes. Si bien podemos intuir dónde se encuentra la componente, los datos no lo revelan con tanta claridad.

G.7.11. Frecuencia no múltiplo de la fundamental de análisis



A los pequeños valores de amplitud que aparecen en la mayoría de las componentes del ejemplo se los suele denominar "artefactos de análisis". Existe una forma de disminuir los artefactos, creando un pequeño *fade-in* al comienzo de la señal de *N* muestras a analizar, y un pequeño *fade-out* al finalizar. Y una manera efectiva de llevarlo a la práctica es multiplicando la señal por una función de *N* muestras, con forma de campana, similar a la que utilizamos para suavizar las transiciones en la síntesis granular.

Esas funciones poseen nombres propios, tales como Hamming, Hanning, Blackman, etc., y cada una presenta ciertas particularidades.

7.3. Aplicaciones de la FFT en el procesamiento de sonido

La Transformada Rápida de Fourier resulta de gran utilidad en el desarrollo de técnicas de síntesis y transformación del sonido. La operación sobre los datos espectrales nos permite modificar puntual o globalmente la amplitud o la fase de cada componente del análisis, y lograr así resultados muy interesantes.

7.3.1. *Crossover*

Un crossover es un tipo de circuito utilizado en los *baffles*, que divide la señal de audio en bandas, cada una de ellas dirigida a un parlante distinto. La banda grave se destina a un parlante desarrollado para reproducir graves, los medios a otro, y así.

Vamos a programar un *crossover* espectral que corte con total precisión en determinado armónico de la fundamental de análisis, separando los graves, para reproducirlos con el parlante izquierdo, y los agudos con el derecho.

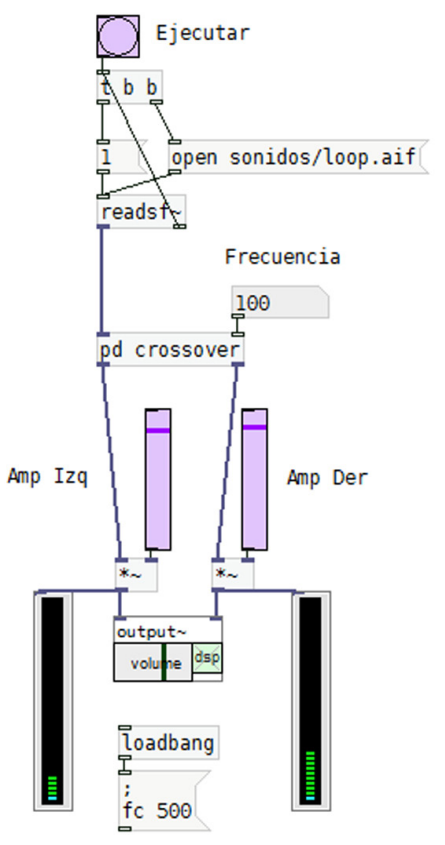
G.7.12. muestra la pantalla principal. Al *subpatch* de procesamiento ingresa la señal del archivo de audio a reproducir y el valor de frecuencia donde deseamos producir el corte. A las salidas tendremos las señales dirigidas a cada parlante. Agregamos *sliders* para controlar la amplitud de cada canal y poder escuchar a uno de ellos silenciando al otro.

La amplitud de la señal puede visualizarse mediante vúmetros, programados como abstracciones, con el nombre *vu~*. El *subpatch* de procesamiento realiza la FFT de la señal de entrada (ver G.7.13.) empleando una ventana Hanning que multiplica a cada bloque. Las funciones de ventana se encuentran en la librería Windowing.

Los resultados de la FFT se van a dividir en dos ramas y cada una de ellas, luego de un simple procesamiento, va convertirse nuevamente en forma de onda mediante la FFT inversa. Pero en este caso, para calcular la FFT no utilizaremos los objetos *fft~ e ifft~, sino rfft~* y *rifft~*. Estas versiones solo calculan la parte positiva del espectro y rellenan con ceros la parte negativa.

Por debajo de estos objetos, observamos una división por N, dado que las amplitudes están escaladas, y nuevamente la aplicación de la ventana Hanning suavizando las discontinuidades producidas por el proceso.

G.7.12. *Crossover* espectral



La parte superior derecha del programa selecciona qué componentes del espectro van a pasar al canal izquierdo y cuáles al derecho, poniendo en cero aquellas que se destinan al canal contrario.

A un objeto *expr~* ingresan por la izquierda los números de armónico de la fundamental de análisis, y por la derecha, el número de armónico donde deseamos cortar, calculado a partir de una frecuencia especificada por el usuario.

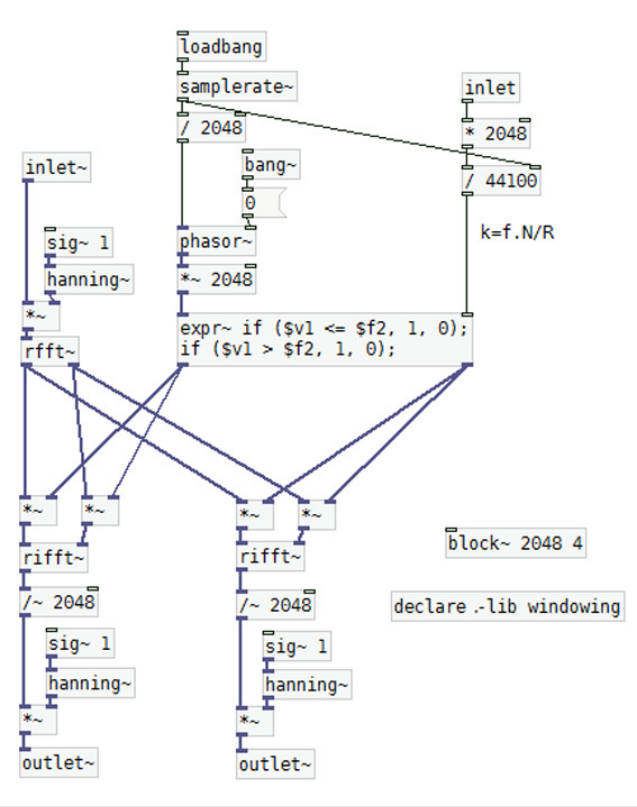
Para calcular el índice de frecuencia *k* dada la frecuencia *f*, utilizamos la siguiente fórmula, ya estudiada:

$$f_k = \frac{kR}{N}$$

Pero en este caso, vamos a despejar *k*, pues *f* es dato:

$$k = \frac{f_k \cdot N}{R}$$

G.7.13. *Subpatch* del *crossover* espectral



El objeto *expr~* incluye dos condicionales *if* separados por punto y coma. Este tipo de sintaxis produce dos *outlets*, uno para cada condición. El primero pregunta si el número de armónico actual es menor o igual que el especificado por el usuario. De ser cierto, envía un 1 por la izquierda, y si es falso un 0. El otro condicional hace lo contrario y envía el dato por la derecha. Multiplicar luego por 1 equivale a dejar pasar, y por 0, a anular la señal.

Una última cuestión a considerar es la forma de especificar el tamaño del bloque. Al objeto *block~* le agregamos otro argumento (el número 4) que indica un solapamiento de las ventanas de análisis, lo cual va arrojar resultados más precisos.

Si bien el tema del solapamiento excede los alcances de este curso, su utilización en el ejemplo es necesaria para mejorar la calidad del sonido.

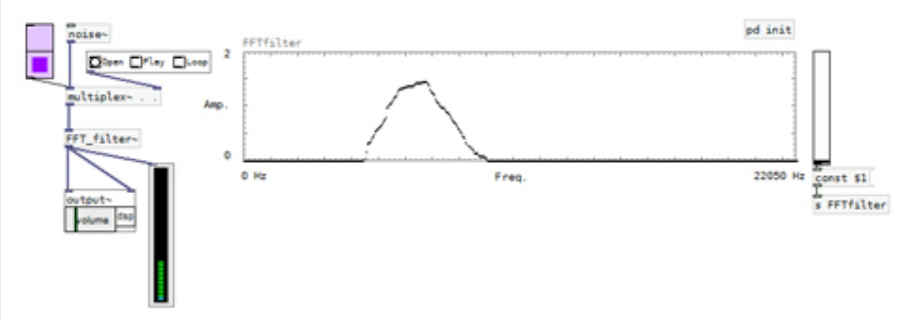
El *patch* "60-FFT-Crossover.pd" contiene la programación de G.7.12.

7.3.2. Ecualizador gráfico por FFT

La siguiente aplicación es un ecualizador gráfico espectral de 512 bandas.

Para especificar la amplitud de cada banda utilizamos una tabla de 512 posiciones, donde cada posición representa a un *slider* de ajuste de la amplitud de la banda.

G.7.14. Ecualizador gráfico por FFT

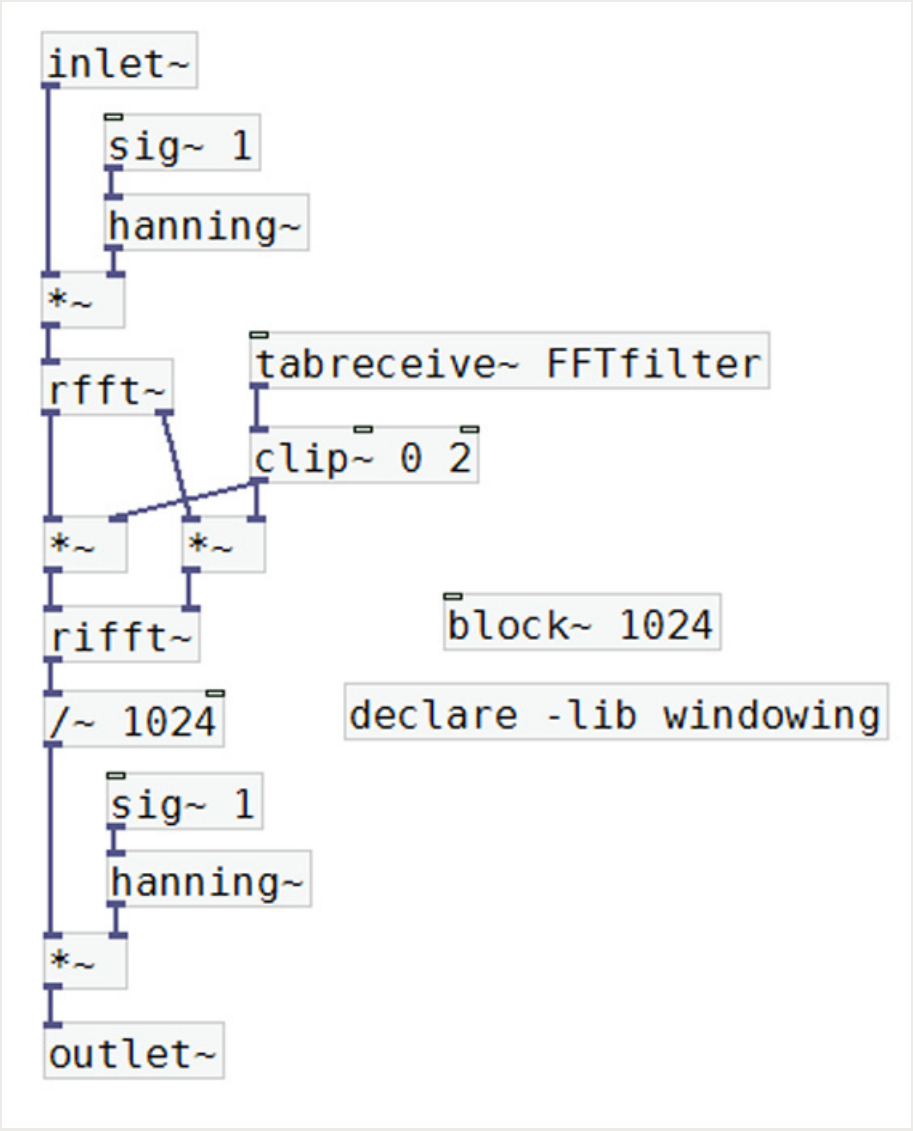


La abstracción donde se realiza el procesamiento está representada en G.7.15. La entrada de audio es multiplicada por una ventana Hanning, al igual que la salida.

Cada bloque es de 1024 muestras. El objeto *tabreceive~* lee la tabla a frecuencia de muestreo, de manera tal que las 512 posiciones son leídas dos veces por bloque. En ambos casos, los valores de amplitud de la tabla multiplican a los valores a y b de la FFT, que por ser real (objeto *rfft~*) tienen ceros en las amplitudes que corresponden a las frecuencias negativas del espectro, o sea en los últimos 512 pares de valores.

El objeto *clip~* mantiene el rango de los valores provenientes de la tabla entre 0 y 2. Esto resulta necesario dado que, por errores de PD, es posible que el usuario dibuje fuera del cuadro de la tabla, generando así valores mayores o menores a lo esperado, de forma involuntaria.

G.7.15. Abstracción del ecualizador gráfico por FFT



El *patch* "61-FFT-Ecualizador.pd" contiene la programación de G.7.14.

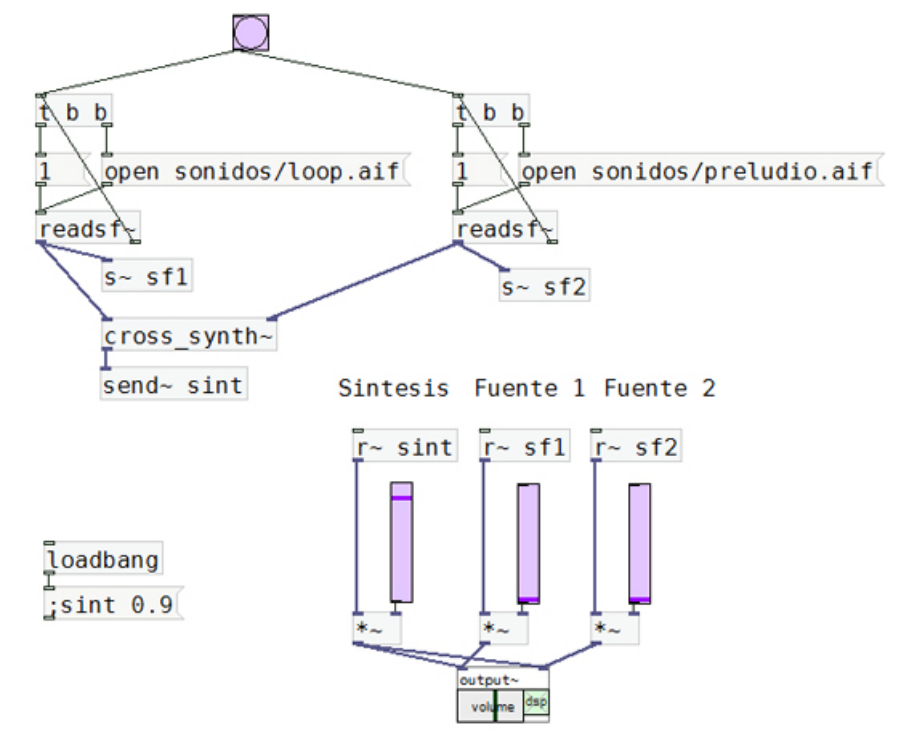
7.3.3. Síntesis cruzada

La síntesis cruzada resulta de la multiplicación de dos espectros. La multiplicación se realiza muestra a muestra, por lo cual, si la amplitud de una componente está en cero, anula a la misma componente del otro espectro.

El proceso en sí mismo se denomina "convolución simple de amplitud", pero como ha sido ampliamente utilizado en la producción de sonidos, conforma una técnica de síntesis llamada "síntesis cruzada".

G.7.16. muestra la pantalla principal del programa. Las señales de dos archivos de sonido ingresan en la abstracción, y sale el producto de la síntesis. Mediante *sliders* es posible escuchar el resultado, o bien cada una de las fuentes por separado, para compararlos:

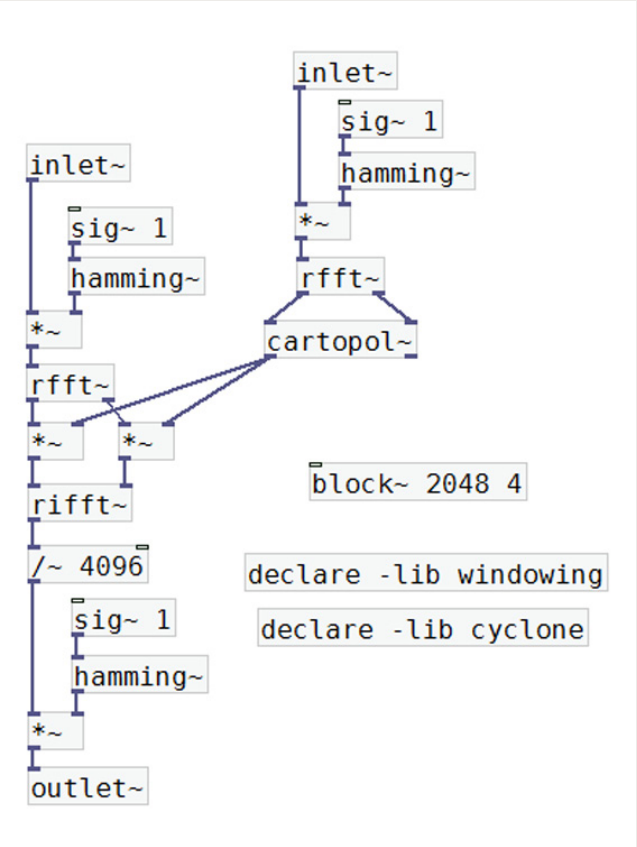
G.7.16. Síntesis cruzada



G.7.17. muestra la abstracción que realiza la síntesis.

Los valores *a* y *b* de un espectro son multiplicados por la amplitud calculada del segundo espectro. Si bien podríamos haber calculado las amplitudes de ambos espectros y luego multiplicarlas, esta forma es más eficiente y ahorra bastante proceso.

G.7.17. Abstracción de síntesis cruzada



Para calcular las amplitudes del segundo espectro no utilizamos el teorema de Pitágoras, sino un objeto que realiza el proceso directamente. Al objeto *cartopol~* ingresan los valores *a* y *b* de la FFT, y este devuelve la amplitud por la izquierda y la fase de la componente por la derecha.

El *patch* "62-FFT-Síntesis cruzada.pd" contiene la programación de G.7.16.

Di Liscia, P. "Análisis espectral", [EN LÍNEA]. En: *Apuntes de Computación Aplicada a la Música II*. Universidad Nacional de Quilmes. 2010.
Disponible en: <http://musica.unq.edu.ar/personales/odiliscia/papers/audig-re2.htm>
[Consulta: 25 de julio de 2013].

Actividad 11

a. Abra la abstracción *spectrogram~* de la librería externa extra, y analice su contenido.

b. Investigue por qué se usa la variable $0 y por qué emplea un filtro pasa altos con la frecuencia de corte en 3 *Hz*.

En el archivo "Respuesta Actividad 11.pd" encontrará la solución de este ejercicio.

Actividad 12

a. Programe un afinador de la nota LA de 440 *Hz* utilizando el objeto *fiddle~*. Este objeto recibe una señal de audio y por su tercer *outlet* devuelve el número de nota MIDI de la altura detectada.

b. Convierta a frecuencia esa nota y grafique en una tabla para visualizar la afinación.

En el archivo "Respuesta Actividad 12.pd" encontrará la solución de este ejercicio.

8. Localización espacial del sonido

Objetivos

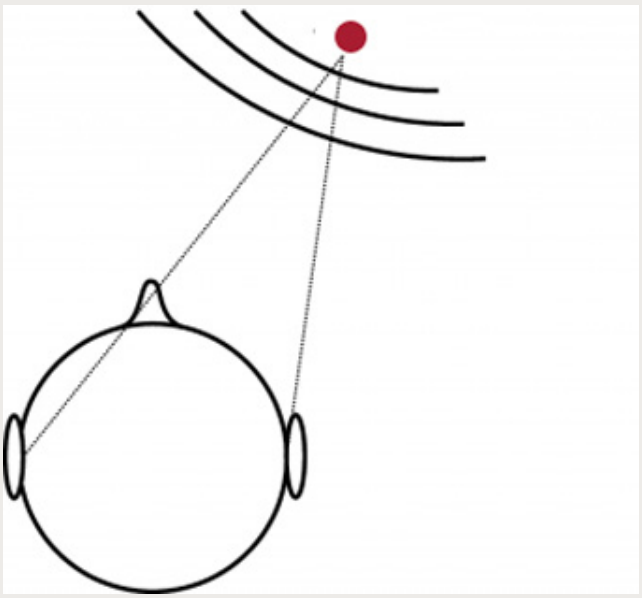
- Conocer los factores psicoacústicos que determinan la localización espacial del sonido.
- Realizar programas de simulación de fuentes virtuales en ambientes ilusorios.

8.1. Factores psicoacústicos que determinan la localización de las fuentes sonoras

Cuando percibimos un evento sonoro recurrimos a diversos indicios que nos ayudan a determinar la ubicación de la fuente que lo produce. Estos indicios dependen de la dirección y la distancia, y suelen dividirse en dos grupos: *binaurales* (con base en la comparación de las diferencias percibidas por cada oído) y *monoaurales* (aquellos que, percibidos por un único oído, resultan relevantes para la localización).

Imaginemos una fuente sonora puntual que se desplaza horizontalmente siguiendo una trayectoria circular alrededor de la cabeza de un sujeto. G.8.1. muestra, para un instante determinado, la diferencia en las distancias que separan la fuente de cada uno de los oídos. Esa diferencia –que depende del ángulo de posicionamiento de la fuente– encuentra su mayor valor cuando el ángulo es de 90° y un valor nulo para 0° y 180°.

G.8.1. Lateralización de la fuente sonora



Según el dibujo, la onda sonora (representada por medio de rayos) alcanza al oído derecho antes que al izquierdo, generándose un pequeño intervalo en los tiempos de arribo y una diferencia en la intensidad percibida. A la diferencia de intensidad registrada se la denomina Diferencia Interaural de Intensidad (DII) y, al intervalo entre la llegada de la señal a cada oído, Diferencia Interaural de Tiempo (DIT).

A pesar de tratarse de valores muy pequeños, el cerebro es capaz de reconocerlos y utilizar esa información para determinar el ángulo de lateralización de la fuente. Cuando la fuente se encuentra a 90°, la diferencia temporal entre la llegada de la señal al oído más cercano (lateral) y el más alejado (contralateral) es de tan solo 630 µs.

La diferencia interaural de intensidad se debe principalmente a la acción separadora de la cabeza que actúa como una pantalla. Para frecuencias cuya longitud de onda es menor que el diámetro de la cabeza, no se produce difracción, registrándose una diferencia de nivel importante, que está en función del ángulo de incidencia de la onda.

Un microsegundo (µs) equivale a la millonésima parte de un segundo.

La difracción es la capacidad de ciertas ondas de contornear objetos y continuar su recorrido. La difracción se produce cuando la longitud de la onda es igual o mayor que el diámetro del obstáculo.

La diferencia interaural de tiempo también depende del ángulo de posicionamiento de la fuente, y el pequeño retardo entre las señales que arriban a cada uno de los oídos, según vimos, está directamente asociado con la diferencia de longitud entre los caminos recorridos.

Ambos indicios actúan de forma combinada, principalmente entre los 800 y los 1600 *Hz*. Con sonidos puros, la DIT es efectiva para las frecuencias inferiores a 800 *Hz* y disminuye hacia el agudo, hasta llegar a los 1600 *Hz*. La DII, en cambio, resulta efectiva para señales de más de 1600 *Hz* y disminuye hacia el grave, hasta llegar a los 800 *Hz*.

La distancia entre la fuente y el sujeto afecta a las diferencias interaurales. La diferencia interaural de intensidad depende en gran medida de la distancia, mientras que la diferencia interaural de tiempo se ve escasamente afectada.
Los indicios binaurales no dependen de manera crítica de las características de la fuente, o del conocimiento previo que el oyente posea de ella. En la audición practicada con un solo oído, en cambio, resulta importante conocer de antemano el sonido que esa fuente provoca, a fin de que podamos localizar su ubicación correctamente.
Un indicio monoaural de importancia es el Indicio Monoaural de Espectro (IME), mediante el cual se contemplan las modificaciones espectrales de un evento sonoro en relación con su posición. Estas transformaciones se deben principalmente a la acción de las orejas que actúan como filtros.

Aun las más pequeñas alteraciones de las señales que llegan a los canales auditivos pueden producir notables alteraciones en la imagen espacial. Acústicamente hablando, el pabellón auditivo se comporta como un filtro lineal que afecta fundamentalmente a las altas frecuencias. Produce una distorsión en la señal, en relación con el ángulo de incidencia y la distancia, codificándola con atributos temporales y espectrales. El efecto acústico del pabellón contribuye tanto a la correcta discriminación entre frente y atrás, como a la detección del grado de elevación de la fuente.
Por mucho tiempo se consideró que los pabellones auditivos no tenían un rol importante en la audición, considerándolos meros protectores del sistema auditivo. En la actualidad sabemos que cumplen una función determinante en la audición espacial, además de servir a la eliminación del ruido del viento.

Con respecto a la determinación de la distancia a la que se encuentra una fuente, la familiaridad con el tipo de sonido juega un rol importante en la eficacia de los resultados. La voz hablada, a sonoridad normal, permite una correspondencia bastante alta entre la posición del evento sonoro y su localización, por ejemplo.

Al igual que ocurre con la intensidad de la luz, o con el tamaño de las imágenes que vemos, la intensidad del sonido decrece si la fuente se aleja del sujeto receptor, y lo hace en proporción inversa con el cuadrado de la distancia. Pero esta no es la única transformación que sufre la onda sonora: el aire mismo actúa como un filtro pasa bajos, atenuando las frecuencias agudas. Su acción filtrante se aprecia claramente cuando escuchamos acercarse un avión, por ejemplo, y percibimos que no solo se escucha más fuerte, sino que el ruido que produce se torna más rico y brillante.

8.2. Audición en recintos cerrados

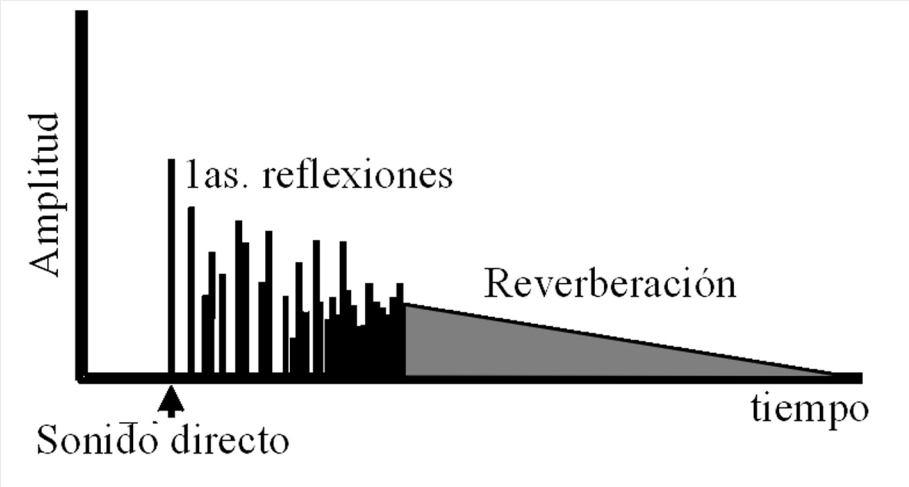
Cuando experimentamos la sensación sonora en un ambiente cerrado, llega primero a nuestros oídos el sonido directo, que es el que proviene directamente de la fuente. En segundo término las reflexiones de primer orden, que parten de la fuente sonora, chocan con un objeto (la pared, por ejemplo) y alcanzan nuestros oídos, luego las de segundo orden (con dos reflexiones), y así en orden creciente, hasta percibir una sensación difusa denominada reverberación. Las primeras reflexiones –en general seis, si consideramos un cuarto vacío con cuatro paredes, techo y piso– contribuyen a la determinación de la posición de la fuente, especialmente si el ataque del sonido es impulsivo. La reverberación, por otra parte, nos brinda información sobre las características materiales de la sala y sobre sus dimensiones. El tiempo de reverberación (t_{60}) se calcula considerando el intervalo transcurrido entre la finalización de la señal acústica y el momento en que el nivel de la reverberación cae 60 dB. El tiempo de reverberación, cuando el volumen *(V)* es medido en m^3 es, según Sabine:

$$t_{60} = 0.16 \frac{V}{A}$$

donde *A* representa la sumatoria de los productos entre los coeficientes de absorción y las superficies reflectoras. Los coeficientes de absorción dependen de la frecuencia de la señal acústica, por lo cual, suele utilizarse el denominado "coeficiente de reducción del ruido", que es la media aritmética entre 250, 500, 1000 y 2000 *Hz*.

El gráfico que sigue (G.8.2.) muestra la distribución de las reflexiones en el tiempo, producidas por un impulso dentro de una sala. La altura de las líneas representa su amplitud relativa.

G.8.2 Respuesta típica de una sala

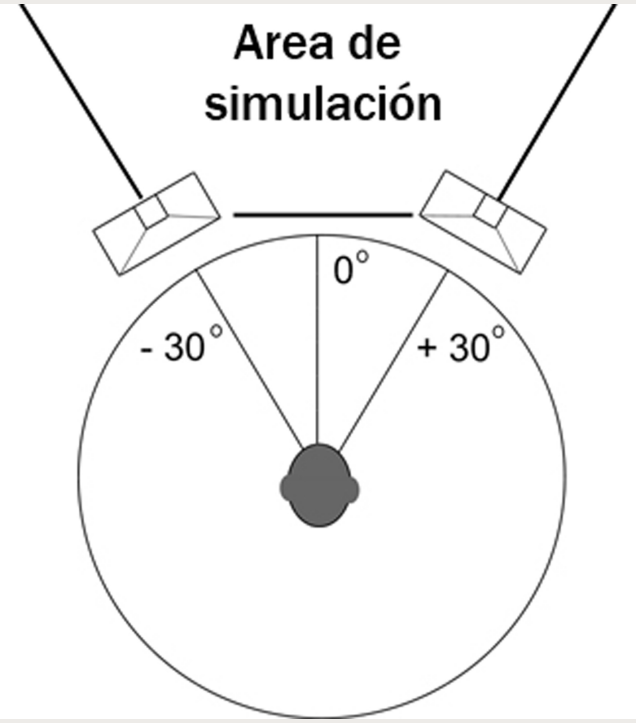


Si bien las reflexiones llegan desde diferentes direcciones, atribuimos una posición única y correcta a la fuente debido a que estas se funden con el sonido directo, agregando una sensación espacial más completa y una mayor sonoridad. Este fenómeno de supremacía de la dirección del directo frente a la de los ecos es conocido como "efecto de precedencia" y ha sido objeto de amplio estudio en psicoacústica.

8.3. Simulación de fuentes virtuales a partir de la estereofonía

El sistema estereofónico nos permite simular la ubicación de una fuente sonora virtual, mediante dos parlantes situados a ambos lados del oyente. Si aplicamos variaciones en la intensidad de las señales emitidas por cada parlante, la fuente parece desplazarse lateralmente de un lado a otro. Recurriendo a otras transformaciones, también resulta posible simular un alejamiento de la fuente, y crear un espacio ilusorio detrás de la posición de los parlantes (ver G.8.3.).

G.8.3 Espacio virtual generado en la simulación



La razón por la cual percibimos una fuente que se desplaza o se ubica en algún punto intermedio entre los parlantes obedece a que el sistema estereofónico genera artificialmente una diferencia interaural de intensidad en el oyente. Si el parlante derecho suena más fuerte que el izquierdo, nuestro oído derecho recibe mayor intensidad, y por lo tanto el sistema perceptual atribuye un grado de lateralización a la fuente, proporcional a la diferencia de intensidad percibida por ambos oídos.

Si bien en un sistema estéreo tradicional la separación abarca unos 60º, vamos a disponer los parlantes formando un ángulo de 90°. Esta licencia nos permitirá extender luego el sistema a uno cuadrafónico (cuatro parlantes rodeando al oyente) sin mayores inconvenientes. Además, a los efectos prácticos, vamos a considerar el ángulo de posicionamiento de la fuente igual a 0° cuando coincide con la posición del parlante izquierdo, y 90° cuando coincide con la del derecho. Trataremos, luego, de determinar la amplitud que debe tener cada parlante para que la fuente virtual se ubique en el ángulo deseado.

Para simular una fuente equidistante al oyente, que se mueva de izquierda a derecha, la suma de las intensidades producidas por ambos parlantes debe ser constante en todo momento.

$$I_i + I_d = k$$

Suponiendo que medimos la intensidad de las señales con valores entre 0 y 1, la suma debe ser siempre igual a 1 (*k* = 1). Si el sonido se encuentra en la posición del parlante izquierdo, las intensidades son 1 para el izquierdo y 0 para el derecho; si está en la posición del parlante derecho, son 0 y 1 respectivamente; y, si la fuente se ubica en el medio, siempre a la misma distancia del oyente, son 0,5 y 0,5.

Pero en los programas de audio digital no operamos en general sobre la intensidad, sino sobre la amplitud de las señales. La intensidad es proporcional a la amplitud al cuadrado y decimos que es proporcional, y no igual, porque ambas poseen unidades diferentes.

$$I \propto A^2$$

Conociendo esto último y reemplazando amplitud por intensidad, podemos escribir:

$$A_d^2 + A_i^2 = 1$$

Del mismo modo, si la fuente estuviera en la posición del parlante izquierdo las amplitudes serían 1 y 0 y, si estuviera en la posición del parlante derecho, serían 0 y 1. Pero si la fuente se ubicara en el centro, ¿serían 0,5 y 0,5? La respuesta es no, porque 0,5 al cuadrado más 0,5 al cuadrado no es 1, sino 0,5. Variando la amplitud linealmente, cuando la fuente se ubique en el medio ($A_i = A_d = 0{,}5$) la intensidad total caerá a la mitad. Perceptualmente, esto puede ser interpretado como un aumento en la distancia. A medida que la fuente se acerca al punto medio entre los parlantes, baja gradualmente la intensidad, y da la impresión de que la fuente se aleja de nosotros.

Para poder calcular las amplitudes correctas para cada posición de la fuente, entre 0° y 90° vamos a recurrir a una identidad trigonométrica:

$$sin^2\theta + cos^2\theta = 1.$$

$$A_d^2 + A_i^2 = sin^2\theta + cos^2\theta = 1, \text{ por lo cual}$$

$$A_d = sin\,\theta \text{ y } A_i = cos\,\theta$$

De este modo, variando el ángulo de posicionamiento de la fuente entre 0 y 90°, y calculando el seno y el coseno, obtendremos la amplitud que debe tener la señal del parlante derecho e izquierdo, respectivamente. Recordemos que el seno de 0° es 0 y el coseno es 1, y que el seno de 90° es 1, mientras el coseno es 0. Cuando el ángulo es de 45° (fuente en el centro) tanto el seno como el coseno, elevados al cuadrado, valen 0,5.

$$A_d^2 + A_i^2 = sin^2 45° + cos^2 45° = 0{,}5 + 0{,}5 = 1$$

Resumiendo, la ganancia que debe tener el parlante derecho para un ángulo θ es *sin* θ, y la ganancia del parlante derecho es *cos* θ . Si el ángulo es de 30°, por ejemplo, la amplitud del parlante derecho será *sin* 30° = 0,5, y la del izquierdo *cos* 30° = 0,866. Si elevamos esos números al cuadrado y los sumamos, el resultado es 1, lo cual indica que la intensidad se mantiene constante.

8.4. Implementación del sistema estereofónico en PD

Antes de comenzar nuestro trabajo de programación del sistema estereofónico, debemos tener en cuenta que los objetos que calculan el seno y el coseno requieren que el ángulo esté expresado en radianes, y no en grados sexagesimales. Recordemos que un radián es el ángulo que se forma cuando la longitud del arco de circunferencia barrido es igual al radio. Siguiendo la definición, 360° equivalen a 2π radianes (el radio entra 2π veces en la longitud de la circunferencia), 180° a π radianes, 90° a π /2 radianes, etcétera.

Para convertir grados a radianes podemos multiplicar el valor del ángulo por 2π y dividirlo por 360°. A los efectos de evitar repetir cálculos innecesarios en las aplicaciones que programemos, podemos resolver de antemano la división de estas dos constantes y multiplicar al ángulo directamente por el resultado de esta operación:

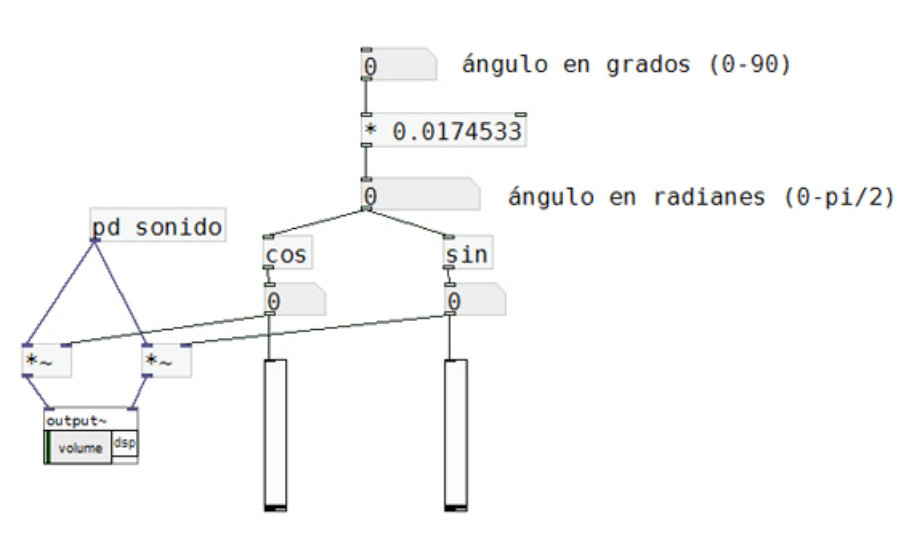
$$Ang_{rad} = Ang^{o} \,.\, 2\pi \,/\, 360^{o} = Ang^{o} \,.\, 0{,}0174533$$

Si multiplicamos un ángulo medido en grados por 0,0174533, lo convertimos a radianes.

El siguiente gráfico ilustra un *patch* que implementa la simulación de una fuente virtual en estereofonía. Para designar a la técnica de balance de la intensidad entre dos parlantes suele emplearse el anglicismo paneo. La denominación en inglés es *intensity panning*.

Se observa en el ejemplo que la señal de la fuente a espacializar (*subpatch* sonido) se divide en dos canales, y cada uno se multiplica por el coseno y el seno de un ángulo entre 0° a 90°, convertido a radianes. Cambiando el ángulo entre esos valores, el sonido parece desplazarse de izquierda a derecha en un arco de circunferencia cuyo radio equivale a la distancia entre el oyente y los parlantes.

G.8.4. Paneo de amplitud

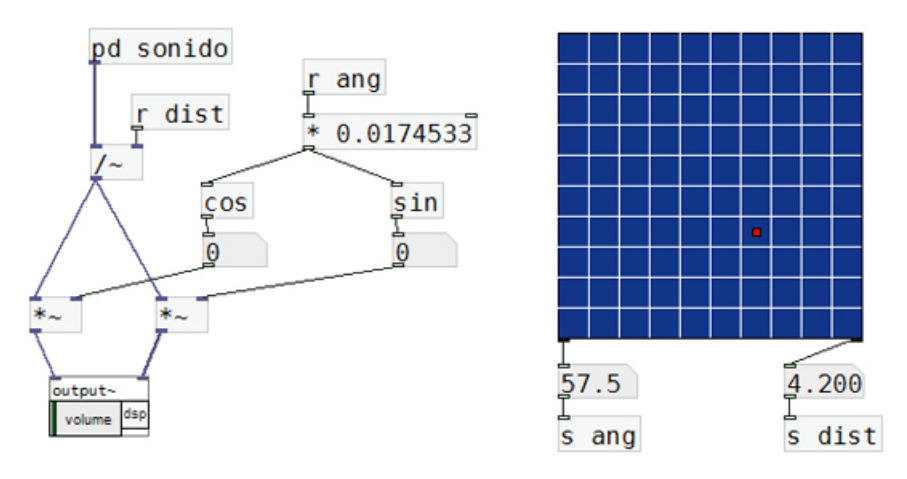


El *patch* "63-paneo.pd" contiene la programación de G.8.4.

Procederemos a implementar la simulación de la distancia. Según vimos, la intensidad decrece con el cuadrado de la distancia. Nuevamente, como operamos sobre las amplitudes y no sobre las intensidades, debemos considerar que la amplitud decrece inversamente con la distancia (1 / *d*). La amplitud que llega al sujeto receptor es la amplitud de la onda que emana la fuente, dividida la distancia que los separa.

A fin de controlar el ángulo de lateralización de la fuente y la distancia fuente-sujeto con un único dispositivo virtual, vamos a recurrir al objeto *grid*, de la librería *unauthorized*, que forma parte de la instalación de PD-*extended*. Este objeto consta de una matriz de dos dimensiones que registra el paso del puntero del mouse y devuelve las coordenadas correspondientes (ver G.8.5.). Podemos establecer, entonces, que los movimientos sobre el eje horizontal correspondan al ángulo de lateralización (0° a 90°), y que los movimientos sobre el eje vertical correspondan a la distancia fuente-sujeto. Para fijar esos rangos, presionamos el botón derecho del mouse sobre el objeto y accedemos al menú **Properties**. Tanto el ángulo como la distancia los enviamos remotamente con objetos *send* y *receive*.

G.8.5. Paneo con distancia

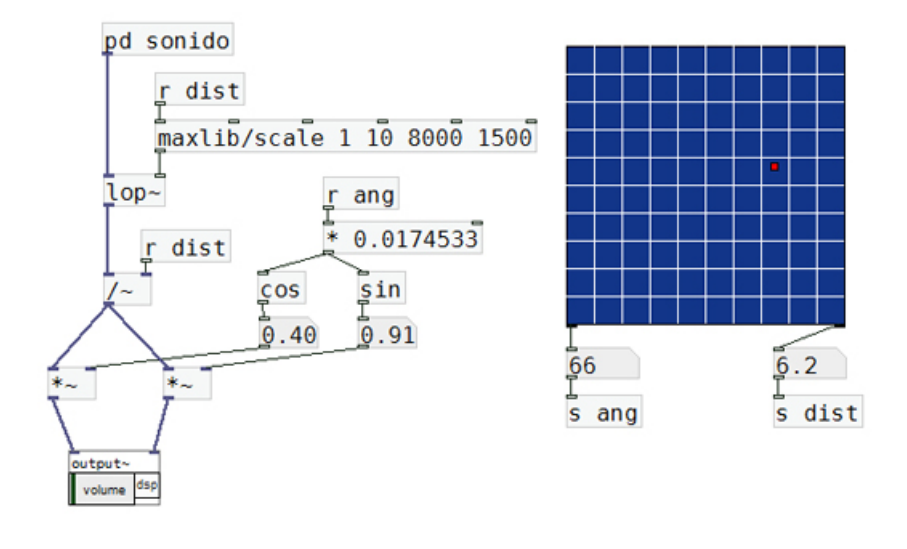


El *patch* "64-paneo con distancia.pd" contiene la programación de G.8.5.

Es importante considerar que al dividir por la distancia, esta nunca puede ser igual a 0, pues, en ese caso el resultado sería indeterminado. Por esa razón, fijamos para la distancia mínima un valor igual a uno, que corresponde a la distancia a la que se encuentran los parlantes. Si los parlantes se ubican a dos metros del oyente la distancia mínima a la que se encuentra la fuente ($d = 1$) también son dos metros, y para una distancia igual a dos ($d = 2$), la fuente se encuentra al doble de distancia que los parlantes, es decir, a cuatro metros. Por consiguiente, nuestra unidad de medida no es otra que la distancia entre el sujeto y los parlantes.

A fin de tornar más realista la sensación de distancia vamos a simular el efecto de absorción del aire. Si bien ese efecto sería despreciable para distancias relativamente pequeñas, sirve a nuestro propósito, pues engaña efectivamente a la percepción, reforzando la sensación de acercamiento o lejanía de la fuente sonora. Para ello recurrimos a un filtro pasa bajos (objeto *lop~*), cuya frecuencia de corte haremos disminuir a medida que aumente la distancia. El escalamiento de los valores de distancia lo realizamos con *maxlib/scale*, de la librería *maxlib*. A este objeto ingresan números entre 1 y 10 (distancia) y salen números que varían proporcionalmente entre 8000 a 1500 y representan a la frecuencia de corte del filtro (ver G.8.6.).

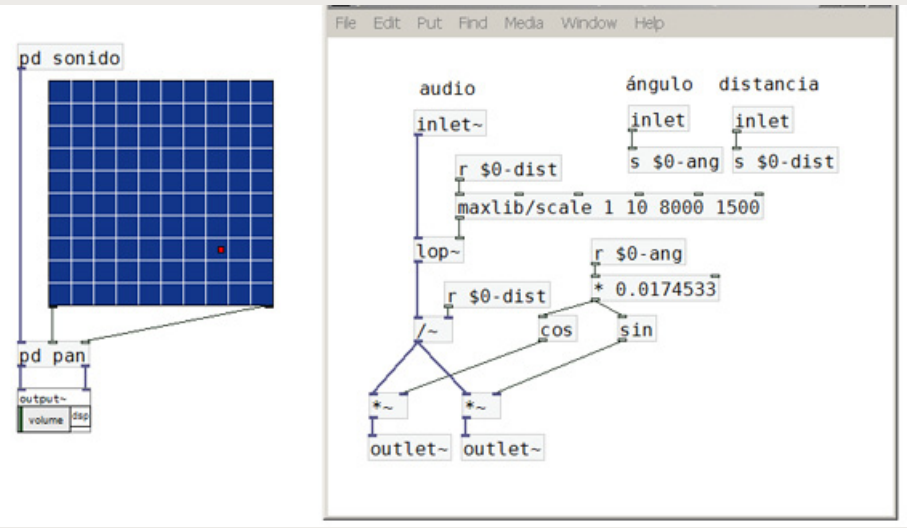
G.8.6. Paneo con distancia y filtro PB



El *patch* "65-paneo con dist y filtro.pd" contiene la programación de G.8.6.

Nuestro espacializador estéreo está listo para ser encapsulado. Una vez logrado esto, podremos generar copias y utilizar varios de ellos en una misma aplicación. G.8.7. muestra la abstracción, denominada *pan*, y su contenido.

G.8.7. Espacializador estéreo encapsulado

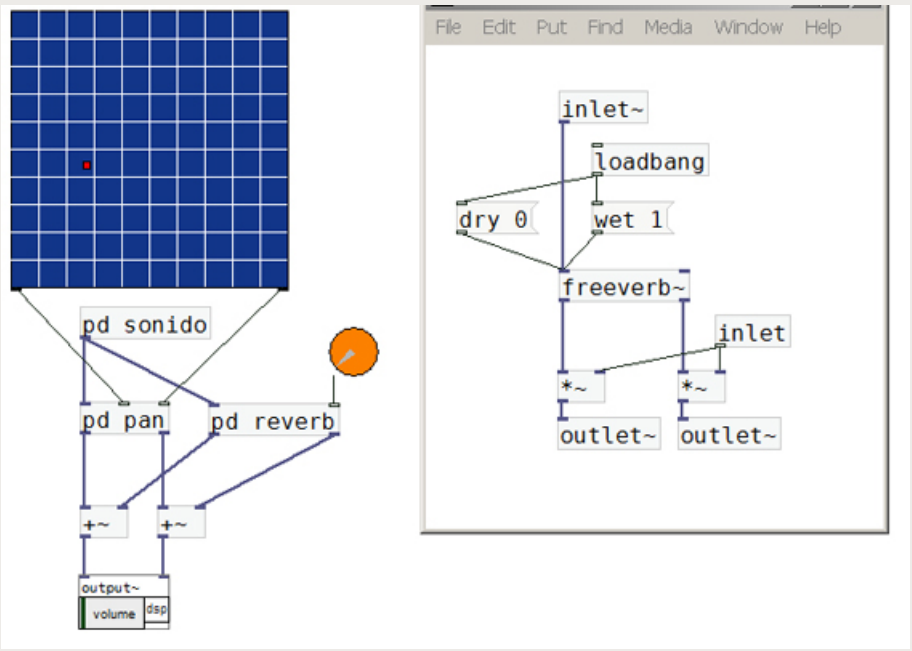


El *patch* "66-espacializador estéreo encapsulado.pd" contiene la programación de G.8.7.

Supongamos ahora que nos encontramos en un ambiente de dimensiones considerables, con un elevado tiempo de reverberación (una iglesia, un garaje, etc.). Si nos hablan desde una distancia cercana, percibimos un nivel alto de sonido directo y muy poco sonido reverberado. Pero, en cambio, si nos hablan desde lejos, percibimos un nivel de sonido reverberado mucho mayor, pues, el directo disminuye con la distancia, mientras que la reverberación se mantiene más o menos constante.

Tratemos ahora de recrear esta situación a través de la simulación de un recinto cerrado. Para ello, agregamos una cámara de reverberación (ver *subpatch* con el objeto *freeverb~*, en G.8.8). Controlamos el nivel de reverberación con una perilla (objeto *mknob* de la librería Moonlib) unida al *subpatch reverb*. Configurando un nivel bajo lograremos que, a medida que la fuente se aleja, se haga más intensa la reverberación, y a medida que se acerca, el sonido directo de la fuente la enmascare.

G.8.8. Espacializador con reverberación



El *patch* "67-espacializador estéreo con reverberación.pd" contiene la programación de G.8.8.

8.5. Simulación mediante la cuadrafonía. El modelo de Chowning

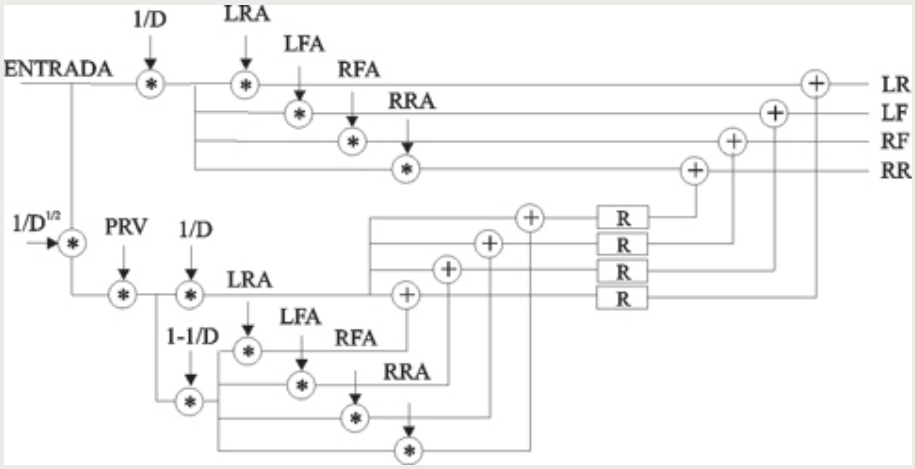
A fin de ampliar el espacio de simulación en un ángulo superior a 90°, o bien con el propósito de aumentar la definición de la localización, la técnica de balance de intensidad entre dos parlantes –utilizada en la estereofonía– puede extenderse a un número mayor de canales.

Disponiendo cuatro parlantes en los vértices de un cuadrado, por ejemplo, es posible ampliar el área de posicionamiento de la fuente virtual a 360° sobre el plano horizontal. Con seis parlantes dispuestos en forma de hexágono también logramos un campo de 360°, pero en este caso con mayor definición en la localización. Puede decirse, como regla general, que a mayor cantidad de canales, mayor resolución y calidad de la simulación. Sin embargo, cualquiera sea el sistema que se emplee, debe presentar una relación razonable entre costos de implementación y calidad de los resultados y, en este sentido, la cuadrafonía ha sido ampliamente aceptada al cumplir con esos requisitos.

El modelo de John Chowning –ilustrado en G.8.9.– se basa en un sistema cuadrafónico que reproduce el sonido directo de la fuente virtual, reverberación local (diferenciada para cada uno de los parlantes de acuerdo con la posición de la fuente) y reverberación global (común a los cuatro canales).

Compositor e investigador estadounidense, reconocido por ser el inventor de la técnica de síntesis del sonido por Frecuencia Modulada, entre otros aportes. Chowning empleó el sistema de localización espacial del sonido que aquí tratamos en la composición de su obra *Turenas* (1972).

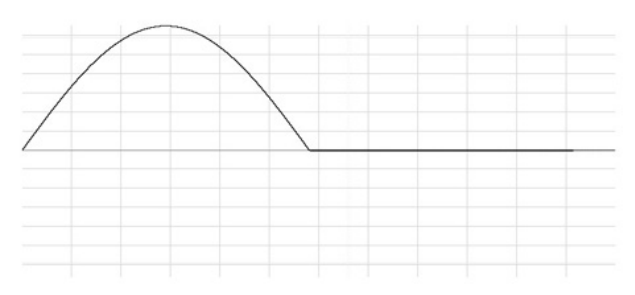
G.8.9. Modelo cuadrafónico de John Chowning



La amplitud de la señal de audio de entrada es atenuada en función de la distancia (1/D), y luego escalada por un coeficiente distinto para cada parlante (LRA, LFA, RFA, RRA) en relación con el ángulo de posicionamiento de la fuente (0º-360º).

Para evitar la realización de los cálculos de ganancia para los cuatro parlantes, se implementa la función que se muestra en G.8.10. Los valores de la función –cuya amplitud varía entre 0 y 1– determinan la amplitud relativa que debe poseer cada parlante para generar un movimiento de 360º. Los cuatro canales leen la misma tabla, pero con fases iniciales diferentes. Vemos que el primer cuarto de la función es una sinusoide entre 0 y 90º, el segundo cuarto una cosinusoide, también entre 0 y 90º, y el resto es 0. Imaginemos que el primer parlante toma los sucesivos valores del primer cuarto (seno de 0º a 90º); el segundo, los del segundo cuarto (coseno de 0º a 90º); el tercer parlante permanece en silencio y el cuarto también. El recorrido que realiza la fuente parte del segundo parlante (que desciende su amplitud con la parte coseno) y se dirige al primer parlante (que crece en amplitud con la parte seno), mientras los otros dos parlantes permanecen en silencio. La función posee 512 muestras y se aloja en una tabla.

G.8.10. Función que evita el cálculo de las ganancias

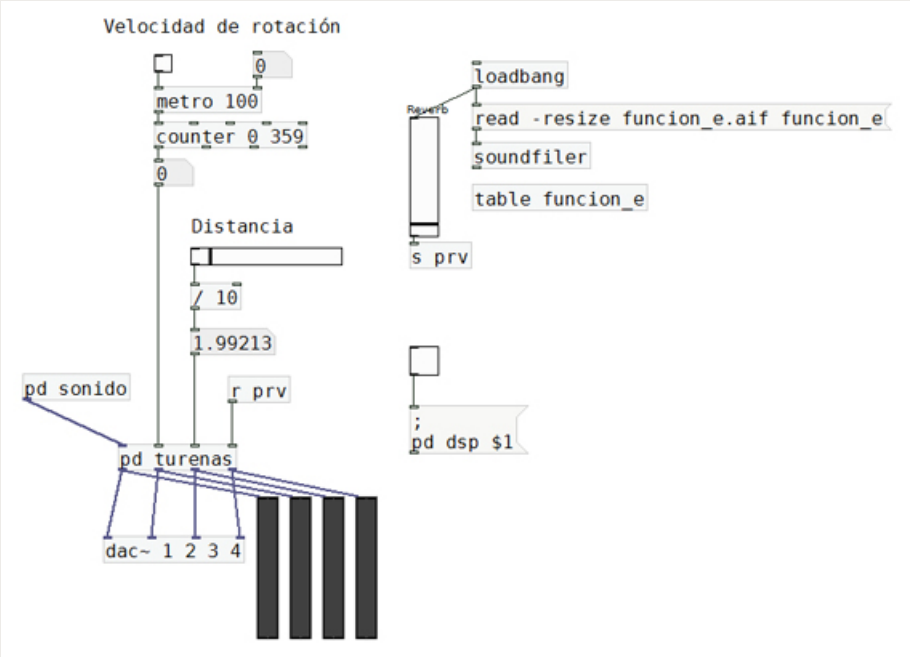


La parte inferior del esquema del modelo (G.8.9.) muestra la rama de tratamiento de la reverberación. La señal es también atenuada en función de la distancia, pero ahora por la raíz cuadrada de la inversa de la distancia, y posteriormente por un valor empírico (PRV) que representa al porcentaje de reverberación. La rama de reverberación es luego escalada por 1/D a nivel global, mientras la local es atenuada por 1-1/D y por los coeficientes en función del ángulo de posicionamiento de la fuente (LRA, LFA, RFA, RRA). El sistema utiliza cuatro unidades de reverberación diferenciadas, con el objeto de hacer más realista la simulación.

8.5.1. Implementación del sistema cuadrafónico en PD

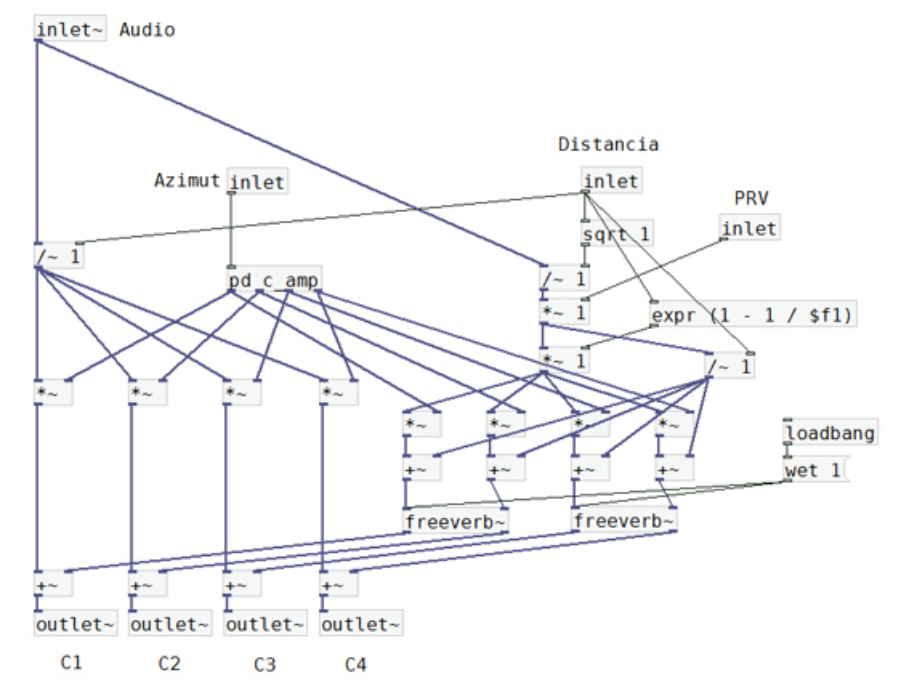
En G.8.11. se muestra la unidad denominada *Turenas* –por el título de la obra en la que Chowning utilizó este sistema de especialización–, donde se encapsuló el modelo. Los parámetros que recibe son –de izquierda a derecha– la señal de audio de entrada, el ángulo de posicionamiento (llamado "ángulo de azimut"), la distancia fuente-sujeto y el porcentaje de reverberación. La *tabla table funcion_e* (parte superior derecha del gráfico) almacena la función que aporta los coeficientes de amplitud en función del ángulo. Finalmente, las cuatro salidas son enviadas al conversor digital analógico.

G.8.11. Modelo de espacializador de Chowning



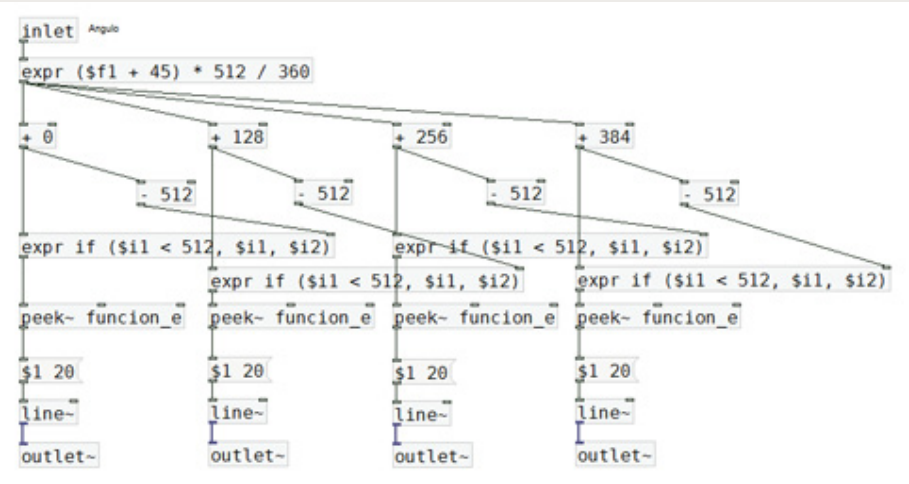
Observemos ahora, en G.8.12., la implementación del modelo propiamente dicho. Aquí encontramos parte del código nuevamente encapsulado: se trata de las operaciones destinadas a obtener cada coeficiente de amplitud en función del ángulo (ver luego en G.8.13.).

G.8.12. *Subpatch* del espacializador



Según dijimos, los cuatro canales leen la misma tabla (con la función *peek~*), pero con las diferencias de fase necesarias.

G.8.13. *Subpatch* de lectura de amplitudes



El *patch* "68-modelo de Chowning.pd" contiene la programación de G.8.11.

8.6. Espacialización con Ambisonics

Ambisonics es una técnica de grabación del sonido, inventada por Michael Gerzson a principios de la década de 1970, que emplea un micrófono especialmente desarrollado para los fines buscados. La característica principal de ese micrófono –denominado en inglés *soundfield microphone*– radica en que combina varias cápsulas, con el propósito de captar información direccional sobre las tres dimensiones del espacio (ejes *x, y, z*) e información omnidireccional. La grabación así obtenida, convenientemente decodificada, reproduce fielmente las cualidades espaciales existentes al momento de grabar.

Una ventaja importante de esta técnica es que la grabación puede ser decodificada para diferente cantidad de parlantes, dispuestos en diversas configuraciones. Un mismo registro puede ser reproducido en dos dimensiones, utilizando cuatro parlantes dispuestos en un cuadrado, o seis en un hexágono, u ocho en una circunferencia, entre otras combinaciones. O bien, en tres dimensiones, empleando ocho parlantes ubicados en los vértices de un cubo imaginario que tiene al oyente como centro. Durante la reproducción, todos los parlantes contribuyen a recrear alrededor del sujeto los movimientos del aire que rodeaban al micrófono durante la toma de sonido.

Resulta interesante, además, que aun cuando no contemos con el micrófono en cuestión, es posible simular su uso mediante un algoritmo de codificación. En la versión más básica de Ambisonics, denominada de primer orden, las señales que se obtienen se denominan *X, Y, Z* y *W*. Las tres primeras son las que contienen información direccional, mientras que la cuarta *(W)* es omnidireccional. El proceso de decodificación de estas señales, para un arreglo particular de parlantes, se realiza mediante combinaciones entre ellas. De este modo, codificando una señal monofónica, y luego decodificándola para una configuración de parlantes elegida, es posible simular una fuente virtual en un espacio bidimensional o tridimensional.

Las ecuaciones de codificación que nos permiten simular una fuente virtual en una posición determinada del espacio (*x, y, z*) son:

$$X = x.\ s(n) / (x^2 + y^2 + z^2)$$

$$Y = y.\ s(n) / (x^2 + y^2 + z^2)$$

$$Z = z.\ s(n) / (x^2 + y^2 + z^2)$$

$$W = 0{,}707\ s(n) / x^2 + y^2 + z^2$$

donde s(*n*) es la señal monofónica a especializar, y *x, y* y *z* son las coordenadas cartesianas donde queremos ubicar la fuente virtual. Debemos prestar especial atención al resultado de la suma de los cuadrados de *x, y* y *z*, pues, no debe ser inferior a 1; caso contrario se produciría un crecimiento exagerado de la amplitud de las señales resultantes.

Ahora, si queremos decodificar estas señales para una configuración de cuatro parlantes dispuestos en los vértices de un cuadrado, las ecuaciones serían las que siguen. En ellas se descarta la componente *z*, debido a que con solo cuatro parlantes no es posible espacializar el sonido en tres dimensiones, sino solamente en dos (sobre el plano horizontal).

$$c1(n) = W - X - Y$$

$$c2(n) = W + X - Y$$

$$c3(n) = W + X + Y$$

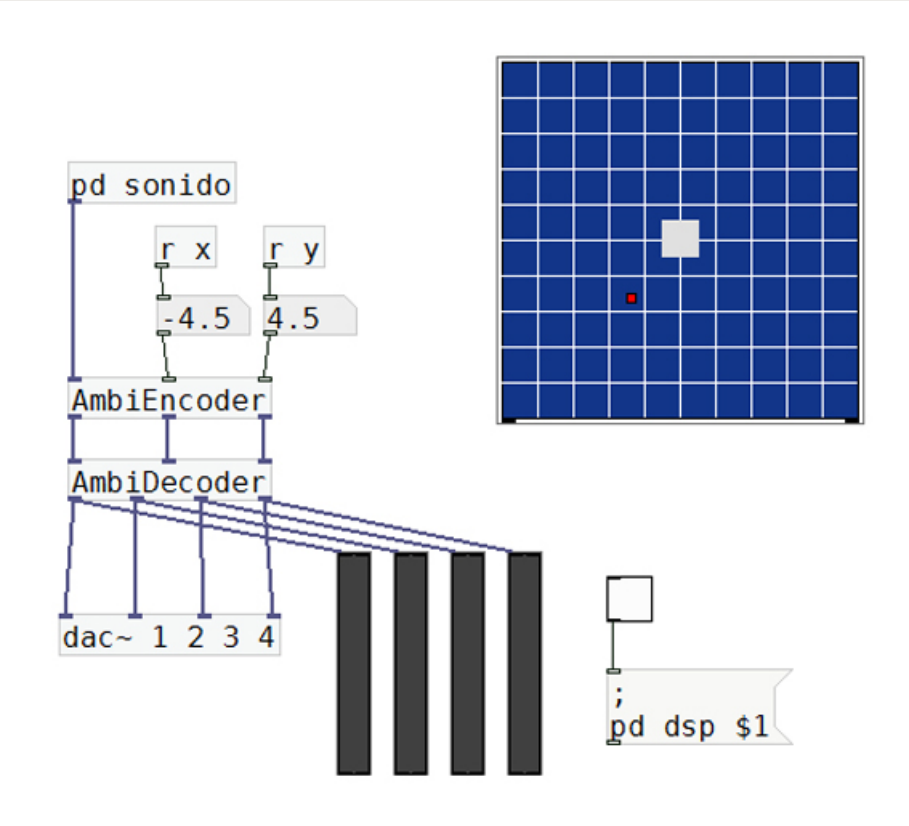
$$c4(n) = W - X + Y$$

donde c1*(n)*, c2*(n)*, c3*(n)*, c4*(n)* son las señales destinadas a los canales 1 al 4, ubicados en el siguiente orden: frente izquierda (1), frente derecha (2), atrás derecha (3) y atrás izquierda (4).

8.6.1. Implementación de Ambisonics en PD

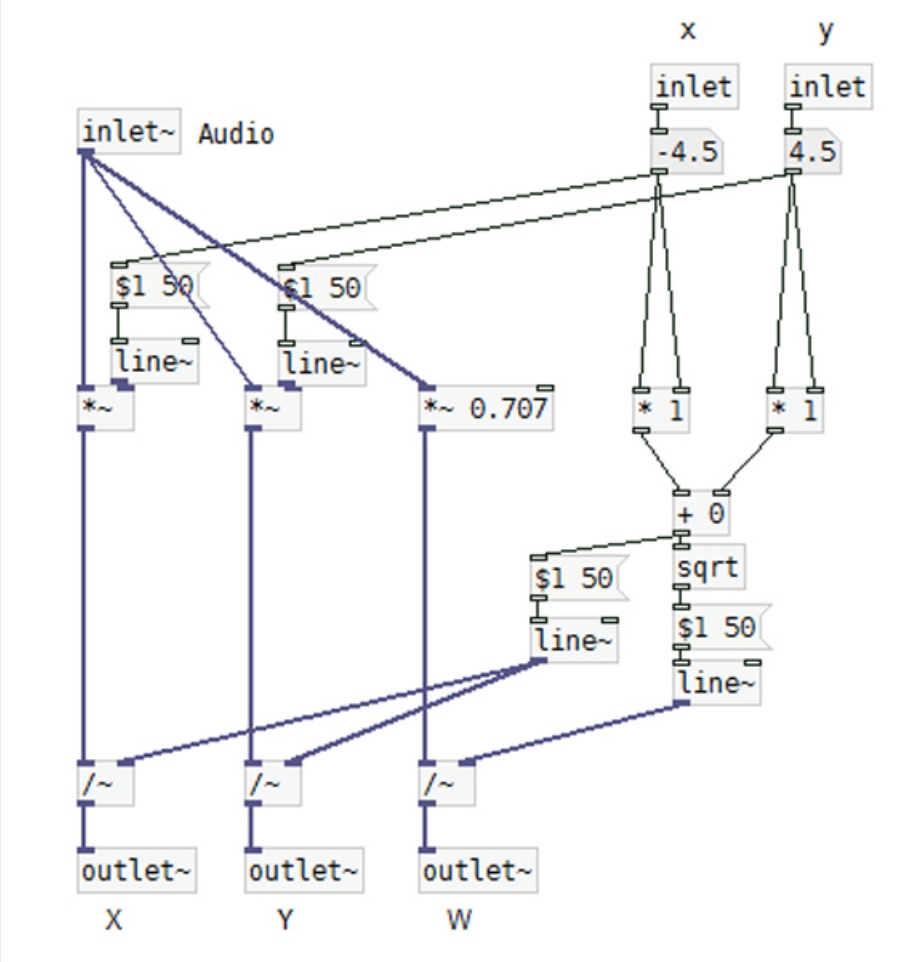
Observamos en G.8.14. la programación general del espacializador Ambisonics. Distinguimos la grilla para establecer las coordenadas *x* e *y* de posicionamiento de la fuente virtual y las abstracciones de codificación y decodificación de señales de audio. Se trata de una implementación cuadrafónica, con los parlantes dispuestos en los vértices de un cuadrado. El oyente –ubicado en el área central de la grilla– ocupa las coordenadas (0, 0).

G.8.14. Espacializador Ambisonics



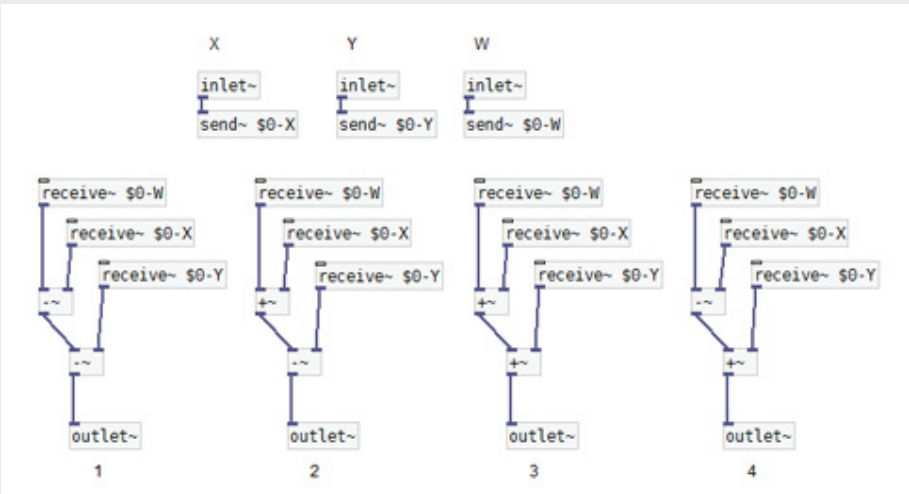
La etapa de codificación recurre a las ecuaciones vistas anteriormente, pero no considera el cálculo de la señal Z, dado que la decodificación se efectuará para un espacio bidimensional.

G.8.15. Abstracción de codificación



La abstracción de decodificación emplea envíos remotos. Las etiquetas de los objetos *send~* y *receive~* incorporan la variable $0 para generar un nombre único para cada instancia creada del objeto. Cada vez que duplicamos AmbiDecoder se generan etiquetas únicas que evitan el cruce de señales entre diferentes instancias del mismo objeto.

G.8.16. Abstracción de decodificación



El *patch* "69-modelo Ambisonics.pd" contiene la programación de G.8.14.

A partir de estos ejemplos, hemos visto tres maneras de lograr la espacialización de fuentes virtuales en ambientes ilusorios. Los sistemas analizados pueden ser ampliados, incorporando la simulación de las primeras reflexiones, la absorción del aire con la distancia o la direccionalidad de la emisión de las fuentes sonoras, con el propósito de lograr un mayor nivel de realismo. Su aplicación en instalaciones sonoras o multimediales brinda posibilidades expresivas sumamente interesantes, que merecen ser exploradas.

Cetta, P. (2004), "Modelos de localización espacial del sonido y su implementación en tiempo real" en: *Altura - Timbre – Espacio*. EDUCA, Buenos Aires, pp. 269-292.

Cetta, P. (2009), "Integración de la música al espacio virtual" en: *Música y espacio: ciencia, tecnología y estética*. Universidad Nacional de Quilmes, Bernal, 271-287.

Actividad 13

Responda las preguntas y realice la actividad indicada:

a. Indique cuáles son los principales indicios que determinan la localización espacial de fuentes sonoras.

b. Describa los fenómenos que ocurren en la audición de eventos sonoros en recintos cerrados.

c. Si consideramos un sistema estereofónico en el cual la posición que se halla justo al frente del oyente corresponde a 0º, y los parlantes se ubican a -45º y +45º respectivamente, ¿cuáles deben ser las amplitudes de los canales izquierdo y derecho para un posicionamiento de la fuente a 15º y una distancia equivalente a la que separa al oyente de cada parlante? Compruebe el resultado empleando la fórmula de la suma de los cuadrados de las amplitudes.

d. Realice un relevamiento, a través de Internet, de sistemas de localización espacial del sonido no estudiados en el curso y escriba un resumen sobre sus particularidades. Investigue si existen librerías de objetos de Pure Data que implementen esos modelos. Se recomienda emplear palabras clave en inglés: *sound localization, spatial location, pure data, vbap, binaural, wavefield synthesis*.

Actividad 14

Programe una aplicación que utilice tres espacializadores estereofónicos y determine las trayectorias automáticamente, con un criterio establecido o bien aleatoriamente. Grafique de algún modo esas trayectorias.

9. Transmisión y recepción de datos

Objetivos

- Conocer las posibilidades de comunicación entre Pure Data y otras aplicaciones, instrumentos musicales electrónicos e interfaces multimediales.
- Realizar programas de transmisión y recepción de información destinada a la síntesis del sonido.

9.1. Comunicación entre aplicaciones y dispositivos

En la actualidad se producen instalaciones multimediales de gran complejidad, en las que el tratamiento simultáneo del sonido y la imagen en tiempo real, y su control mediante sensores e interfaces, requieren una capacidad de procesamiento importante.

La comunicación entre programas y dispositivos posibilita la realización de tales proyectos. Mediante el intercambio de información entre computadoras conectadas en red se incrementa la potencia de cálculo, particularmente, si asignamos a cada una de ellas una tarea específica.

Por otro lado, las propuestas de interacción a través de Internet, o de teléfonos celulares, en las cuales participan público y artistas, separados a veces por grandes distancias, no podrían concretarse sin la posibilidad de transmitir o recibir datos.

En esta unidad vamos a analizar los medios disponibles para comunicar los programas desarrollados en Pure Data con otras aplicaciones o dispositivos multimediales.

Comenzaremos por estudiar el protocolo MIDI, para continuar luego con la transmisión y recepción de mensajes a través de redes de computadoras.

9.2. El protocolo MIDI

MIDI *(Musical Instruments Digital interface)* es un sistema de comunicación entre instrumentos musicales digitales, creado en 1983 a partir de una asociación formada por fabricantes *(MIDI Manufacturers Association)*. Entre los instrumentos MIDI más utilizados encontramos teclados (órganos, sintetizadores), guitarras, baterías, acordeones, instrumentos de viento y módulos de producción de sonido.

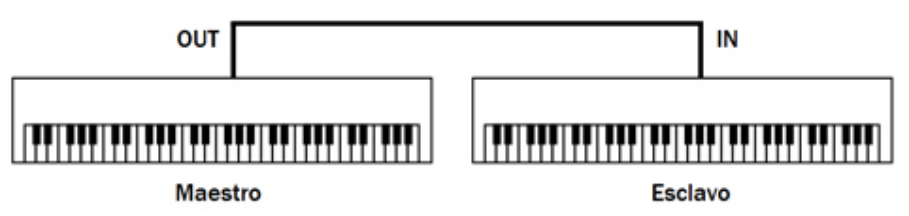
Para saber más sobre esta organización y consultar tutoriales y especificaciones MIDI puede visitar el sitio de MMA:
<http://www.midi.org>

Cada instrumento posee una interfaz incorporada, cuyas especificaciones están detalladas en la norma MIDI. Los fabricantes de instrumentos deben seguir esas especificaciones para que sus instrumentos puedan comunicarse con instrumentos de otras marcas.

Conectando la salida de un instrumento con la entrada de otro, podemos lograr que lo que se interpreta en el primer instrumento sea reproducido por el segundo. El instrumento en el que tocamos es, a veces, llamado *controlador,* pues, no es el que produce el sonido directamente, sino que envía datos a otros generadores del sonido.
La interfaz MIDI consta de tres puertos: IN, OUT y THRU:

- Por IN entran los datos provenientes de otros dispositivos MIDI.
- Por OUT salen los mensajes generados por el propio dispositivo.
- El THRU está conectado con el IN, internamente, y sirve para dar salida hacia otros dispositivos a los mensajes que arriban al IN.

G.9.1. Conexión MIDI



Un instrumento funciona como *maestro* y el otro como *esclavo*. El primero transmite mensajes relacionados con las operaciones que sobre él se practican, por ejemplo, bajar una tecla, cambiar el tipo de sonido (piano, guitarra, etc.), modificar el volumen. El esclavo recibe estos mensajes y realiza las mismas operaciones que el maestro (ver G.9.1.).

También es posible conectar una computadora a través de MIDI. Para esto es necesario colocarle una interfaz para computadoras y conectarla a los instrumentos empleando cables MIDI. G.9.2. muestra un tipo de configuración posible.

G.9.2. Conexión MIDI con computadora

La función de la computadora es capturar los mensajes que viajan por la línea, con la posibilidad de almacenarlos y, más tarde, poder editarlos y/o reproducirlos. Es importante aclarar que en este caso la computadora no graba el sonido mismo, sino tan solo una sucesión de órdenes que envía el instrumento mientras se lo ejecuta, concretamente qué notas se ejecutaron, con qué intensidad, en qué momento. Luego, la computadora puede reenviarlos al instrumento para que este interprete automáticamente.

En algún sentido, el sistema se parece a una pianola. Cuando el pianista toca sus teclas, en el rollo se inscriben mensajes que indican cuándo se tocó una nota, qué notas se tocaron, a qué intensidad, cuánto dura cada nota, cuál es el tempo de la música. Luego, la pianola puede leer la información del rollo y ejecutar la música sin la presencia del músico. Resulta obvio que en el papel quedaron registradas las acciones del pianista, pero no el sonido del piano, y esto nos lleva a pensar las diferencias entre MIDI y audio digital.

Un archivo MIDI contiene información sobre cómo interpretar una música, como si se tratara de una partitura. Un archivo de audio digital contiene una forma de onda, a la música grabada.

La ventaja principal de MIDI es que la cantidad de información necesaria para interpretar la música es muy pequeña (comparada con la misma música registrada digitalmente). Esto permite almacenarla y transmitirla con facilidad y, además, procesar los mensajes en tiempo real, aun con una bajísima capacidad de procesamiento de datos. Pero la principal desventaja es que se hacen necesarios los instrumentos musicales electrónicos que reciban los mensajes MIDI y los conviertan en sonido y música.

9.2.1. Canales MIDI

Existen 16 canales MIDI y en la mayoría de los mensajes es necesario especificar un número de canal. Esto permite que los dispositivos que reciben la información puedan configurarse para recibir parte de toda la información que se transmite a través de la conexión. Supongamos que un instrumento musical que funciona como esclavo está configurado para recibir solamente lo que se transmite por el canal 1, en este caso, todos los mensajes que no pertenezcan a ese canal serán ignorados por el instrumento.

La división de la información por canales resulta muy útil cuando se envían mensajes desde una computadora. Si un tema musical está conformado por una parte a cargo de piano, por ejemplo, y otra a cargo de una flauta, los mensajes a cada uno de estos instrumentos deberán ser enviados por canales MIDI diferentes. De este modo, el instrumento electrónico que recibe los datos podrá discriminar qué notas asignar al piano que hay en él y qué notas derivar a la flauta. Vale aclarar que un mismo instrumento electrónico (un órgano o sintetizador) puede imitar varios instrumentos al mismo tiempo. La cantidad de instrumentos a imitar simultáneamente es en general 16, cifra que coincide con la cantidad total de canales MIDI.

G.9.3. Panel de interfaz y cables MIDI

9.2.2. Los archivos .MID

Los archivos MIDI son generados y reproducidos con programas denominados secuenciadores *(Sonar, Cubase,* etc.*)*. Cada programa graba un formato de archivo propio, como ocurre con cualquier procesador de texto, por ejemplo. Con el objeto de unificar el formato, los creadores de la norma MIDI diseñaron un tipo de archivo intercambiable entre programas, denominado MIDI Standard File, cuya extensión es .mid.

9.2.3. *General* MIDI

Antes de que la norma MIDI incorporara el concepto *General* MIDI, los fabricantes de instrumentos electrónicos incluían diversos "timbres", como piano, guitarra o violín, que ordenaban de la manera que les parecía más conveniente. Una secuencia realizada para un instrumento de determinada marca, al ser reproducida en otro instrumento diferente producía resultados inesperados, ya que las notas eran las que correspondían, pero interpretadas por otros timbres. Para solucionar este problema, se agregó a la norma una lista ordenada de timbres denominada *General* MIDI. Cuando un instrumento electrónico admite la configuración *General* MIDI, garantiza la correcta ejecución de una secuencia creada con base en esa norma.

9.2.4. Algunos tipos de mensajes

Los tipos de mensajes más comunes del protocolo MIDI son los siguientes:

Note On
Enciende una nota en un instrumento.
Note Off
La apaga.
Program Change
Cambia de timbre en el instrumento MIDI.
Control Change
Envía mensajes a los controladores, que son las palancas o perillas del panel de control de un instrumento MIDI que alteran al sonido en alguna forma.

El mensaje *Note On*, por ejemplo, tiene una longitud de 3 *bytes*. Al primer *byte* se lo denomina *Status Byte*, porque especifica el tipo de mensaje, a los restantes se los denomina *Data Bytes*.

El término *bit* (acrónimo de *Binary Digit*) es un dígito del sistema de numeración binario y puede valer 0 o 1. Un *byte* es, en general, un número formado por 8 *bits* y puede contener valores entre 0 (00000000) y 255 (11111111), que resultan de la combinación de unos y ceros.
Si precisa ampliar información sobre *bits*, *bytes* y el sistema de numeración binario puede realizar consultas en Wikipedia <(http://es.wikipedia.org)>.

El primer *byte* indica, entonces, el tipo de mensaje (en este caso *Note On*) y además el canal MIDI por el cual se envía. Vemos que hay dos informaciones distintas codificadas en un mismo *byte*: los 4 *bits* menos significativos indican canal MIDI (4 *bits* = 16 posibilidades), y los otros 4 *bits* indican el tipo de mensaje.
El segundo *byte* indica qué número de nota hay que tocar (0 a 127).
El tercer *byte* indica a qué intensidad debe ejecutarse esa nota (*Key Velocity*, de 1 a 127).

Los *bytes* de datos siempre tienen un 0 en el *bit* más significativo, por lo tanto, la información que pueden llevar es solamente de 7 *bits* (números decimales comprendidos entre 0 y 127).

Veamos un mensaje *Note On* en sistema de numeración binario:

byte 1 : 1 0 0 0 x x x x

El "1" del *bit* más significativo señala que se trata de un *byte* de *Status*. "000" significa *Note On*. Las *x*, si se reemplazan con unos o ceros, dan 16 posibilidades y especifican el canal MIDI.

byte 2 : 0 x x x x x x x

El primer "0" indica que se trata de un *byte* de datos. Las x, siete en total, permiten enviar un número entre 0 y 127 que es la nota MIDI a tocar.

byte 3 : 0 x x x x x x x

Otra vez se trata de un *byte* de datos, pero ahora las x representan la intensidad de la nota a ejecutar. Las intensidades varían de 1 a 127, debido a que una intensidad cero equivale a apagar la nota, lo cual se denomina mensaje alternativo de *Note Off*.

9.2.5. MIDI y Pure Data

Los objetos MIDI de PD pueden dividirse en dos grupos. El primer grupo está formado por objetos que reciben mensajes MIDI, mientras que el segundo está integrado por objetos que los envían. Para recibir o enviar mensajes debemos contar con un instrumento MIDI (un teclado, por ejemplo) que los genere o los interprete, según sea el caso, conectado a la computadora a través de una interfaz.

Entre los objetos más usados que reciben información MIDI encontramos los siguientes:

Objeto	**Descripción**	**Tipos de dato**
notein	Reporta el ingreso de un mensaje *Note In*.	Número de nota (0-127), *key velocity* o intensidad de la nota (1-127) y canal MIDI utilizado (1 a 16).
ctlin	Reporta datos de un controlador. El pedal de sostenimiento del sonido, por ejemplo, es el controlador 64.	Valor del controlador, número de controlador utilizado y canal MIDI.
pgmin	Reporta un cambio de "timbre" o instrumento.	Número de instrumento elegido y canal MIDI.
bendin	Reporta datos de la rueda de transposición de altura, denominada pitch wheel.	Valor de la posición de la rueda (0-127) y canal MIDI.

Los objetos más comunes que envían información son:

Objeto	**Descripción**	**Tipos de dato**
noteout	Transmite un mensaje *Note In* a un dispositivo MIDI para que ejecute una nota.	Número de nota, *key velocity* y canal MIDI de destino.
ctlout	Transmite datos a un controlador.	Valor del controlador, número de controlador utilizado y canal MIDI.
pgmout	Envía una solicitud de cambio de "timbre" o instrumento.	Número de instrumento elegido y canal MIDI.
bendout	Transmite datos a la rueda de transposición de altura, denominada *pitch wheel*.	Valor de la posición de la rueda y canal MIDI.

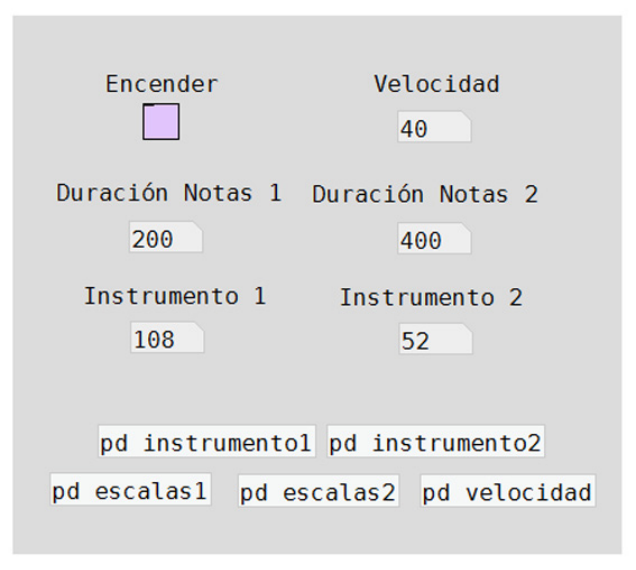
Si dispone de un teclado MIDI y una interfaz para conectarlo a la computadora, podrá comprobar la transmisión y recepción de datos, y el efecto que cada mensaje produce. De no poseer estos dispositivos, puede probar en una computadora que disponga de un sintetizador de sonido interno (varias de las computadoras que emplean una placa de sonido *on-board* poseen uno). Otro recurso posible es instalar un *sampler* o un sintetizador virtual, y comunicarlo con PD a través de puertos MIDI virtuales.

G.9.4. muestra un *patch* que envía mensajes *Note in* y mensajes de cambio de instrumento a un dispositivo MIDI externo, a un sintetizador interno, o bien a un dispositivo MIDI virtual (*sampler* o sintetizador) conectado a través de un puerto MIDI virtual. Para configurar alguna de estas opciones puede ir al menú Media-Preferencias MIDI.

Un *sampler* virtual es un programa que recibe mensajes MIDI y ejecuta muestras de sonidos instrumentales grabadas en archivos de audio. Por otra parte, un sintetizador virtual es una aplicación que genera el sonido en la misma computadora. En ambos casos, para enviarles mensajes MIDI a través de PD es necesario instalar un puerto MIDI virtual. Si ingresa en un buscador "virtual MIDI port" encontrará algunos gratuitos.

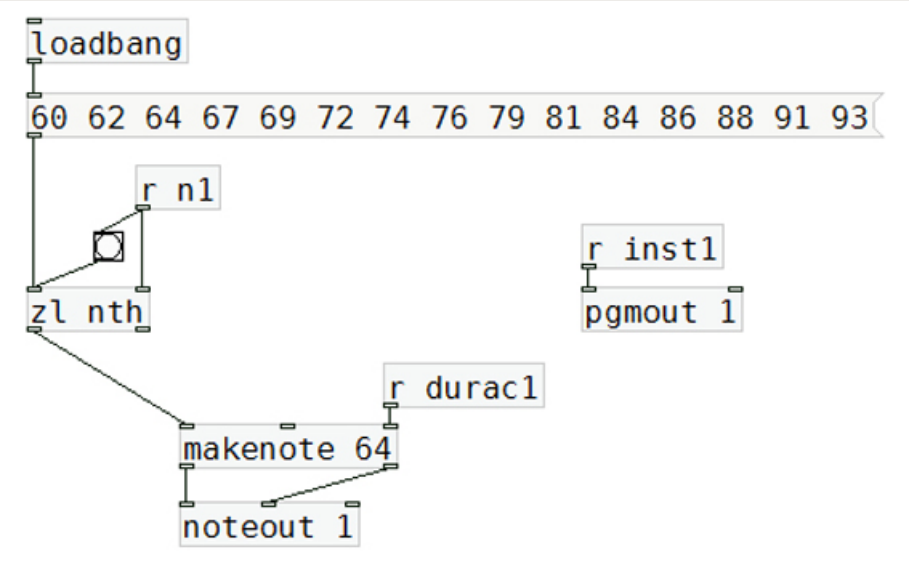
El *patch* genera sucesiones de notas de una escala pentatónica (escala de 5 sonidos, sin semitonos), ascendentes o descendentes, cambiando la direccionalidad de forma aleatoria. En la ejecución de los giros escalares, que abarcan 3 octavas, la velocidad aumenta o disminuye, simulando una interpretación expresiva. Para dar un grado mayor de interés al resultado, se agrega una segunda voz, más grave, cuyas notas atacan al doble de tiempo que las de la voz superior. Desde el panel, es posible modificar la duración de las notas (cada nota más corta o más larga), el instrumento (números que corresponden al catálogo General MIDI) y la velocidad promedio de la ejecución.

G.9.4. *Patch* de ejecución MIDI



El gráfico G.9.5. pertenece al *subpatch* PD instrumento1, que es el que interpreta la voz aguda. Observamos allí un nuevo objeto, denominado makenote, que recibe por sus *inlets*, de izquierda a derecha, un número de nota MIDI a ejecutar, su *key velocity* (intensidad) y la duración en milisegundos que deseamos darle al sonido. El objeto deja pasar el número de nota y el *key velocity* al objeto *noteout*, y transcurrido el tiempo de duración, envía nuevamente el número de nota y un valor de *key velocity* igual a cero, para apagar la nota encendida.

G.9.5. *Subpatch* instrumento1

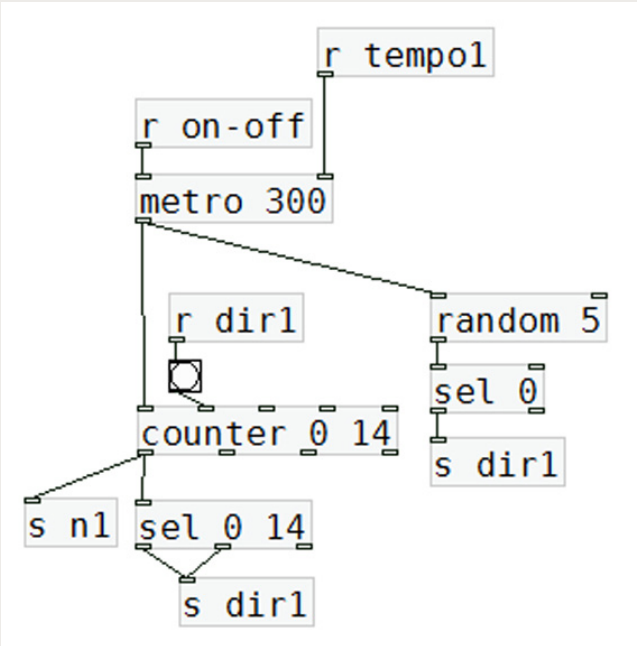


El objeto *zl*, de la librería Cyclone, lleva inscripto el argumento *nth*. Así configurado, el objeto devuelve el enésimo elemento de la lista que ingresa por su *inlet* izquierdo. El número de elemento a obtener será calculado en otro *subpatch*, y enviado a este remotamente mediante la variable n1.

El objeto *makenote* recibe por la izquierda la nota y por la derecha la duración, mientras que el valor de *key velocity* está especificado como argumento (64 equivale a *mezzo forte*).

La figura siguiente (G.9.6.) muestra el *subpatch* donde se genera el número de elemento a interpretar de la escala.

G.9.6. *Subpatch* escalas1



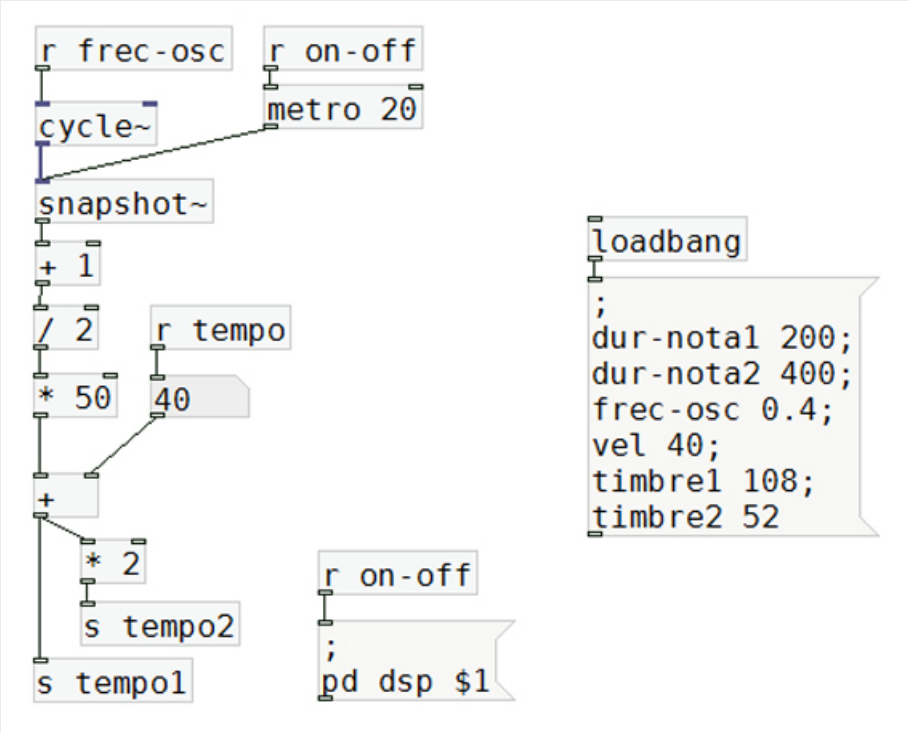
El objeto *counter* (librería Cyclone) cuenta entre 0 y 14, pero aleatoriamente recibe un *bang* por el segundo *inlet*, que lo obliga a cambiar de dirección en la cuenta (si cuenta de forma ascendente pasa a descendente y viceversa).

Por último, el *subpatch* velocidad (G.9.7.) produce una desviación en la velocidad establecida para generar las notas, sumando un valor de desvío que varía sinusoidalmente, a muy baja frecuencia.

De la salida de audio del oscilador sinusoidal se extrae una muestra cada 20 milisegundos, con el objeto *snapshot~*. Cada vez que este objeto recibe un *bang* desde metro, devuelve el valor de amplitud de la señal de audio. Este método se emplea para convertir señales de audio, a frecuencia de muestreo, a señales de control de pocas muestras por segundo.

Los valores de amplitud, convenientemente escalados, se suman a la velocidad establecida por el usuario del programa.

G.9.7. *Subpatch* velocidad



La programación del *patch* de G.9.4. se encuentra en el archivo "70-protocolo MIDI.pd". Para ejecutarlo debe disponer de una placa de sonido con sintetizador incorporado, un dispositivo físico MIDI de generación de sonidos, o bien un instrumento MIDI virtual (sampler o sintetizador).

9.3. Redes

El envío de datos en una red de computadoras se establece a partir de un método denominado "conmutación de paquetes". Y una técnica para encaminar los paquetes –utilizada por la red Internet– es la de los datagramas.

Mediante los datagramas, cada paquete lleva una parte de la información a transmitir y la dirección de destino. Por otra parte, cada paquete puede viajar por rutas diferentes, e incluso llegar a destino en un orden distinto en el que fue transmitido.

Dos de los protocolos de transporte de datos que se emplean son TCP *(Transmission Control Protocol)* y UDP *(User Data Protocol)*. El primero garantiza que todos los datos lleguen sin errores y que la información se reconstruya en el orden correcto. UDP, en cambio, no ofrece garantías de orden ni de recepción.

Por otra parte, un "puerto", en términos de redes de computación, es la interfaz a través de la cual se envían y se reciben los datos, y se identifica con un número entre 1 y 65535.

Algunos servicios de las computadoras utilizan un puerto específico. Por ejemplo, el puerto 80 es reservado para HTTP, y el puerto 25 para el envío de correo electrónico.

HTTP (*Hypertext Transfer Protocol*, en castellano Protocolo de Transferencia de Hipertexto) es un método de intercambio de información mediante el cual se transfieren las páginas web a una computadora.

Los dispositivos conectados a una red se identifican con una etiqueta numérica, denominada "dirección IP". La dirección IP puede ser fija (el dispositivo siempre se identifica del mismo modo, como es el caso de un servidor que muestra una página web) o dinámica, en cuyo caso las direcciones se asignan automáticamente de acuerdo con el momento y la cantidad de computadoras o dispositivos que se conectan a una red.

A su vez, existen las IP públicas y las privadas. La dirección IP pública es la que tiene asignada cualquier equipo que se conecta directamente a Internet, como los servidores que alojan páginas web y los *routers* o *modems* conectados a proveedores de Internet. La dirección IP privada es la que identifica a los equipos de una red doméstica o privada.

Para intercambiar información entre dispositivos deberemos, entonces, elegir un protocolo de transporte de datos (TCP o UDP), un puerto y conocer el nombre de red o la dirección IP del dispositivo con el cual nos vamos a conectar. En relación con el protocolo, si se trata de dos computadoras conectadas a través de Internet, lo más aconsejable es elegir TCP. La opción restante resulta útil en una red segura, la cual se establece uniendo a las máquinas a través de cables, o mediante un *router*, como los que se emplean en redes domiciliarias u oficinas para conectar varias computadoras a Internet.

Para la elección del puerto de comunicación es importante tener en cuenta que no se encuentre actualmente en uso. Si utilizamos el sistema operativo Windows sabremos qué puertos ya están siendo ocupados leyendo el archivo "services", ubicado en el directorio "c:/windows/system32/drivers/etc/". En los sistemas basados en Unix, podemos escribir en la terminal el comando *ifconfig*, y en MacOS podemos averiguarlo en **Preferencias del sistema-Compartir-Firewall**.

Como regla general, podemos establecer que el número de puerto deberá ser mayor a 1024, cualquiera sea el sistema operativo instalado en nuestra computadora.

Para conocer nuestra IP pública podemos ingresar al sitio en español <http://www.vermiip.es/> o al sitio en inglés <http://www.ip-adress.com/>. Para conocer la IP privada, en cambio, podemos realizar lo mismo que para conocer los puertos en uso, para los sistemas Windows y Unix, y en MacOS en **Preferencias del sistema-Red-Configurar-TCP/IP**.

A fin de ampliar la información sobre redes de computadoras realice consultas en *Wikipedia* sobre los siguientes temas: "Redes de computadoras", "router", "TCP", "UDP" y "dirección IP" <(http://es.wikipedia.org)>. Si tiene dudas sobre el modo de acceder a su dirección IP, utilice el navegador para buscar información acerca de su sistema operativo.

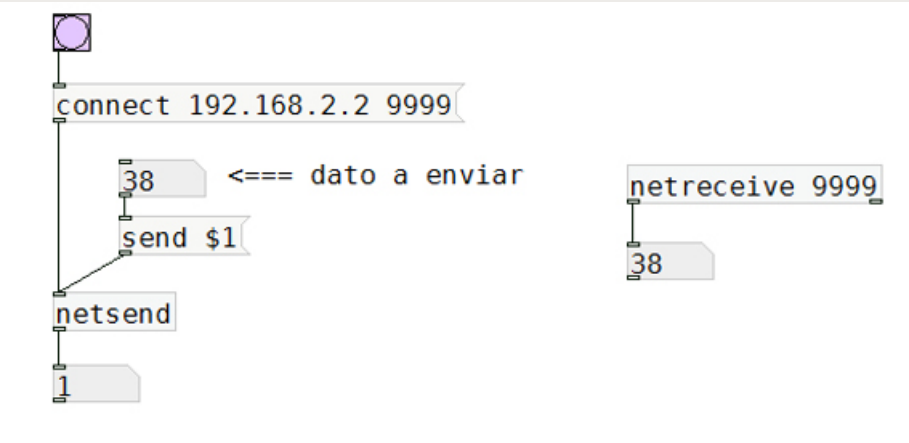
9.4. Objetos *netsend* y *netreceive*

Estos objetos están diseñados para enviar y recibir mensajes a través de una red. El objeto *netsend* se comunica utilizando los protocolos TCP o UDP, si bien el primero es el que emplea por defecto (para recurrir a UDP es necesario utilizar el número 1 como argumento).

A fin de establecer la conexión debemos introducir en *netsend* el mensaje *connect*, seguido de la dirección IP de la computadora de destino y un número de puerto. Por su *outlet* saldrá un 1 si la conexión fue establecida o un 0 en caso contrario. Para enviar datos, simplemente ingresamos al objeto un mensaje *send* seguido de los datos a transmitir.
Por otra parte, en el objeto *netreceive*, ubicado en la computadora que recibe los datos, escribimos solo el número de puerto como argumento.

G.9.8. muestra las dos partes de una conexión (transmisión y recepción). Para este ejemplo, utilizamos el protocolo de transporte TCP (por defecto), la dirección IP interna (192.168.2.2) de la computadora que recibe los datos y el puerto 9999.

G.9.8. Transmisión de datos mediante *netsend* y *netreceive*



No solo podemos transmitir números, sino listas de datos que pueden tener contenido numérico y símbolos. Si bien la idea es enviar información de una máquina a otra, el ejemplo así planteado igualmente sirve para comunicar a la computadora que ejecuta el *patch* con sí misma.

La programación del *patch* de G.9.8. se encuentra en el archivo "71-netsend y netreceive.pd". Para poder utilizarlo, si cuenta con una red doméstica, no olvide reemplazar la dirección IP del ejemplo por la de su computadora.

9.5. *Open Sound Control*

Open Sound Control (OSC) es un protocolo de comunicación entre dispositivos electrónicos multimediales conectados en red, desarrollado para la transmisión de datos entre aplicaciones.

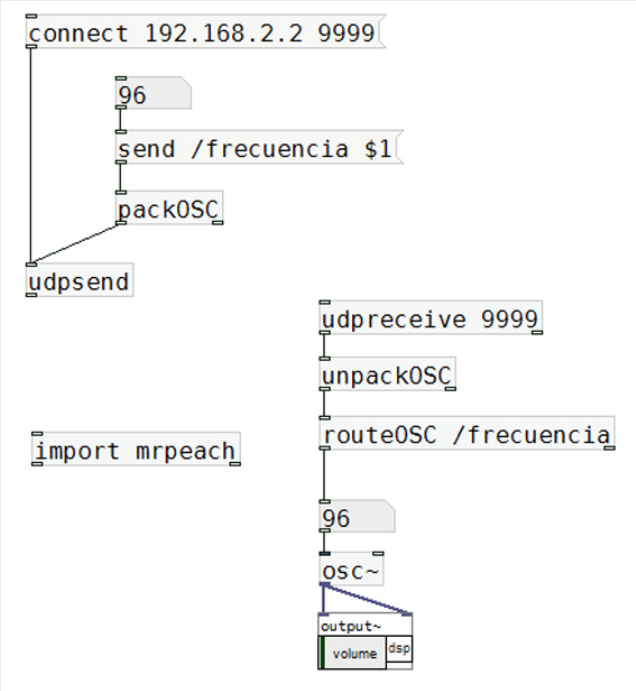
Originalmente, el proyecto fue pensado para el control de instrumentos musicales, como reemplazo de la norma MIDI, pero luego cobró mayores dimensiones debido a su alto grado de aceptación como medio de transmisión de información entre programas en red.

OSC fue creado en el CNMAT (Center for New Music and Audio Technology) de la Universidad de California en Berkeley. Para obtener más información sobre este protocolo puede consultar <http://opensoundcontrol.org>

Las áreas de aplicación de OSC son muy variadas. Entre ellas. la comunicación entre sensores conectados en red con sintetizadores virtuales, conversión de datos no musicales para la generación de sonido, control de procesos sonoros o musicales a través de múltiples usuarios, interfaces web, ejecuciones musicales en red, etcétera.

G.9.9. muestra el envío y la recepción de información mediante el protocolo OSC, empleando a UDP como protocolo de transporte. Los objetos *udpsend* y *udpreceive*, de la librería *mrpeach*, son los encargados de realizar la conexión y, según se observa, utilizan los parámetros de forma similar a *netsend* y *netreceive*. El objeto *packOSC* empaqueta la información a enviar, y los objetos *routeOSC* y *unpackOSC* clasifican los datos recibidos y los desempaquetan.

G.9.9. OSC a través del protocolo UDP



La programación del *patch* de G.9.9. se encuentra en el archivo "72-OSC básico.pd". Para poder utilizarlo, si cuenta con una red, no olvide reemplazar la dirección IP del ejemplo por la de su computadora.

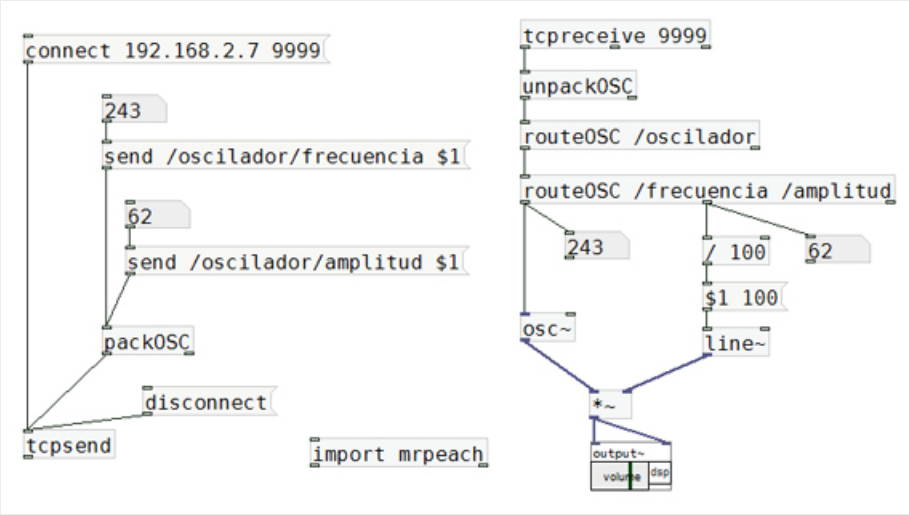
Las versiones para conexiones TCP de los objetos de envío y recepción son *tcpsend* y *tcpreceive*.

El protocolo OSC no posee mensajes predefinidos, lo cual lo vuelve muy interesante. Cada usuario puede definir sus propios mensajes de la manera que considere más apropiada.

Los mensajes pueden comenzar con una ruta que permita clasificarlos. Esta ruta luce como un sendero de subdirectorios, por ejemplo "/frecuencia".

Si deseamos enviar a un oscilador de un *patch* información sobre la frecuencia y la amplitud con las que debe operar, podemos organizar los datos bajo una etiqueta común "/oscilador", y luego etiquetas individuales seguidas de los datos. G.9.10. ilustra este procedimiento, y el modo en que se utilizan los objetos *routeOSC* para extraer la información. Conectados en serie permiten acceder a los sucesivos niveles de una ruta, pero si se declaramos dos o más argumentos, el objeto incrementa automáticamente su número de *outlets* y dos o más parámetros de un mismo nivel pueden ser extraídos.

G.9.10. Organización de los mensajes



La programación del *patch* de G.9.10. se encuentra en el archivo "73-OSC con rutas.pd".

A través de los objetos *tcpsend* o *udpsend* también podemos enviar listas de datos y desempaquetarlas al recibirlas con el objeto *unpack*.

Mediante los protocolos vistos podremos no solo conectar distintas computadoras ejecutando a PD, sino también a este con otras aplicaciones destinadas a desarrollo de software, como *Processing*.

Processing es un lenguaje de programación y entorno de desarrollo de aplicaciones multimediales. Para saber más sobre este programa consulte <http://processing.org/>

Del mismo modo, recibir datos de sensores o controladores, empleando programas especializados en obtener información de estos dispositivos y comunicarla a otros programas a través de OSC. Ejemplos de este tipo de aplicaciones son *OSCulator*, que permite comunicar a una computadora *Mac* con *Wiimotes* de *Nintendo*, *Iphones*, etc., o *Synapse*, para comunicar el sensor *Kinect* de *Microsoft* con otras aplicaciones.

Para obtener información sobre los recursos mencionados puede consultar los siguientes sitios web: <http://www.osculator.net/> (OSCulator), <http://latam.wii.com/> (controlador Wii), <http://www.xbox.com/es-ES/Kinect> (Kinect) y <http://synapsekinect.tumblr.com/> (Synapse para Kinect).

Causa, E. (2011), "Diseño de Interface para el desarrollo de una pantalla sensible al tacto con aplicación musical" en: *Revista de Investigación Multimedia Nro. 3*. IUNA, Buenos Aires, pp. 45-53.

Causa, E. (2011), "Desarrollo de un sistema óptico para interfaces tangibles (mesa con pantalla reactiva)" en: *Revista de Investigación Multimedia Nro. 3*. IUNA, Buenos Aires, pp. 54-67.

Actividad 15

Partiendo una red de dos computadoras, o bien comunicando una computadora consigo misma, utilice el *patch* de síntesis que desarrolló en la Actividad 6, ubicando los parámetros de control del lado del transmisor, y el resto,como receptor.

Actividad 16

Opción a) El objeto *hid*, de la librería que lleva el mismo nombre, permite leer las acciones realizadas sobre un controlador de video juegos *(joystick, gamepad)* conectado al puerto USB de la computadora. Si posee un controlador de ese tipo, programe un *patch* que detecte todas las acciones que puedan realizarse sobre el dispositivo.

Opción b) Desarrolle una aplicación que modifique sus parámetros a través del mouse y del teclado de la computadora. Los objetos clave para esta tarea son *key* y *cursor* (de la librería *hcs*).

En ambos casos puede consultar el siguiente texto:
Floss Manuals-Pure Data
<http://en.flossmanuals.net/pure-data/sensors/game-controllers/>
[Consulta: 3 de Agosto de 2013]. Controladores para Pure Data.

10. Procesos compositivos

Objetivos

- Formalizar procedimientos compositivos y analizar métodos de composición algorítmica.
- Programar aplicaciones de composición algorítmica y evaluar los resultados en términos musicales.

10.1. Composición asistida y composición algorítmica

Existen dos campos de aplicación de las computadoras en el proceso compositivo. Uno de ellos es el de la composición asistida, en el cual la computadora es considerada una herramienta capaz de ayudar a generar los materiales de la obra, a partir de la formalización de procedimientos que realiza el compositor. En la composición asistida se emplean, en general, programas especialmente diseñados o lenguajes de programación como Open Music y PWGL.

El lenguaje Open Music fue desarrollado en el IRCAM. Se basa en una interfaz gráfica similar a la de PD y está programado en lenguaje LISP. Para más información consulte <http://repmus.ircam.fr/openmusic/home>

PWGL fue creado en la SIbelius Academy de Finlandia y es similar al lenguaje anterior. <http://www2.siba.fi/PWGL/index.html>

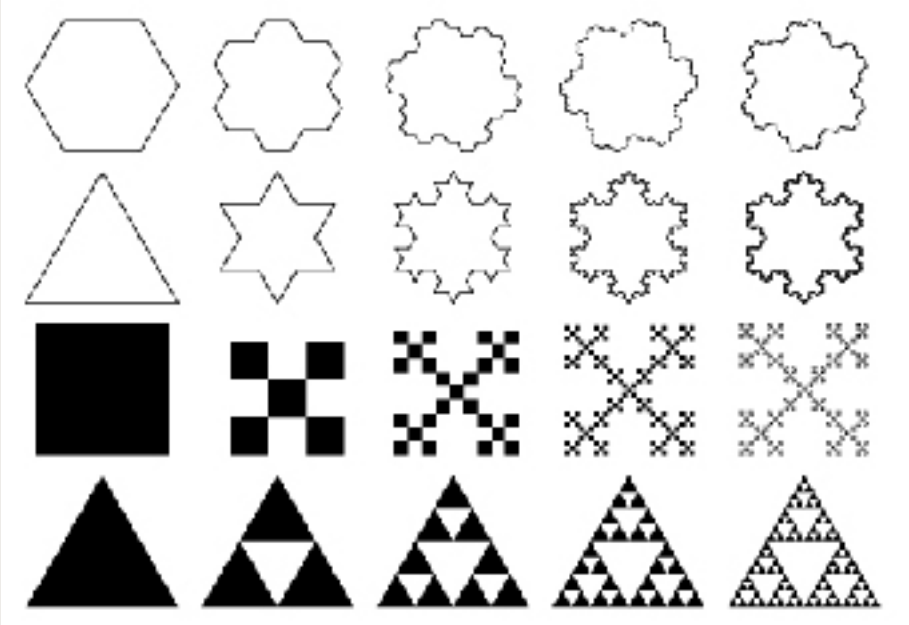
El segundo campo de aplicación es en la composición algorítmica. Si bien la composición realizada a través de algoritmos –entendidos estos como conjuntos de reglas– ha existido en la música desde siempre, nos referimos a composición algorítmica cuando en general interviene el azar en determinadas elecciones y los procesos se realizan con la participación de computadoras. La máquina "crea" la obra, o parte de ella, en virtud de que el algoritmo empleado es capaz de tomar decisiones durante el proceso creativo y, en la mayoría de los casos, también la interpreta.

Un modelo de composición algorítmica muy difundido en la actualidad se basa en algoritmos genéticos. A través de este modelo, la composición se desarrolla en términos evolutivos. La computación evolutiva deriva de la Inteligencia artificial y recurre a conceptos biológicos tales como selección natural, mutación, cruzamientos, y al estudio de poblaciones autoorganizadas y comportamientos colectivos.

Las técnicas actuales empleadas en la composición algorítmica son muy diversas y se basan también en modelos matemáticos, sistemas computacionales capaces de aprender y gramática formal, entre otras.

La composición algorítmica vinculada con modelos matemáticos, por su parte, ha utilizado con frecuencia la técnica de los fractales, objetos geométricos cuya estructura básica se multiplica a diferentes escalas (ver G.10.1.).

G.10.1. Imágenes de fractales



Otros modelos matemáticos se sustentan en la probabilidad y la estadística, o en la teoría del caos.

La primera obra creada por una computadora fue la Suite *Illiac* (1956), a partir de la programación realizada por Lejarin Hiller y Leonard Isaacson de la Universidad de Illinois. La pieza está escrita en cuatro movimientos e instrumentada para ser interpretada por un cuarteto de cuerdas.

Otro pionero en el uso de la computación en la composición algorítmica fue Iannis Xenakis, quien desarrolló obras musicales basadas en procesos estocásticos. Mediante estas técnicas, descriptas en su libro *Formalized Music* (1963), realizó las primeras piezas en este campo: *Atrées* (para 10 instrumentos) y *Morsima-Amorsima* (para piano, violín, violoncelo y contrabajo), ambas escritas en 1962.

Véanse artículos relacionados en Wikipedia.

En la teoría de la probabilidad y de la estadística, se define como estocástico a un proceso en el cual una o más variables aleatorias (el resultado de arrojar dos dados, por ejemplo) va cambiando en el tiempo. Si bien no se puede predecir con exactitud el resultado, es posible conocer la probabilidad de obtener un valor determinado, en relación con el conjunto de valores posibles.

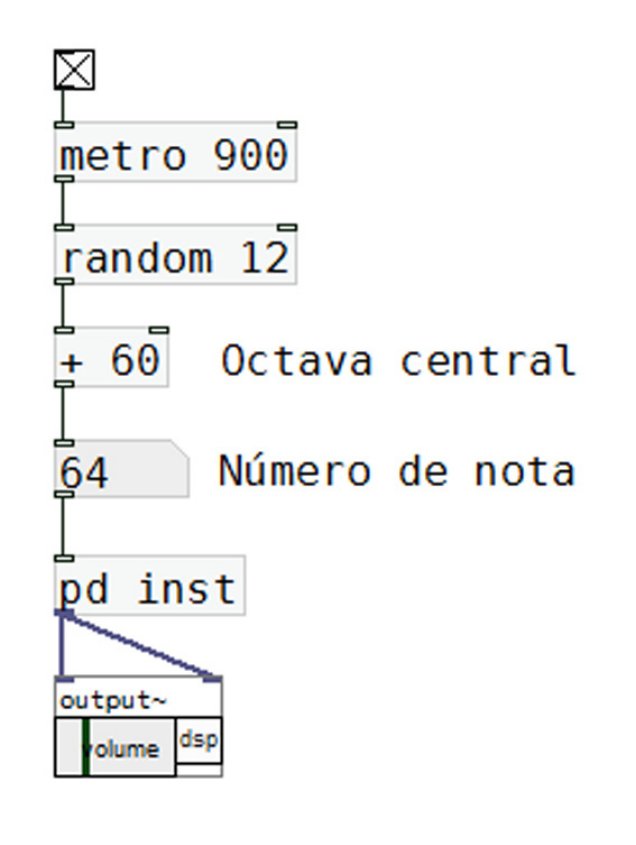
10.2. Variables aleatorias

Una técnica muy empleada en composición algorítmica consiste en determinar valores aleatorios para determinados parámetros musicales (la altura, por ejemplo) y someterlos a una serie de reglas. Si algún candidato no cumple con alguna de las reglas establecidas es rechazado.

Esta técnica puede ser utilizada en la composición de melodías, por ejemplo, donde su construcción está establecida por normas que determinan el registro, el ámbito (distancia entre la nota más grave y la más aguda de la melodía), los intervalos a emplear (distancias entre dos notas sucesivas), nota de inicio y de final, las duraciones, etcétera.

G.10.2. muestra un *patch* que genera notas al azar (números entre 0 y 11, que representan a las doce notas comprendidas en una octava). A los números de nota aleatorios les sumamos 60 para centrarlos en el registro (octava central del piano).

G.10.2. Generación de notas al azar

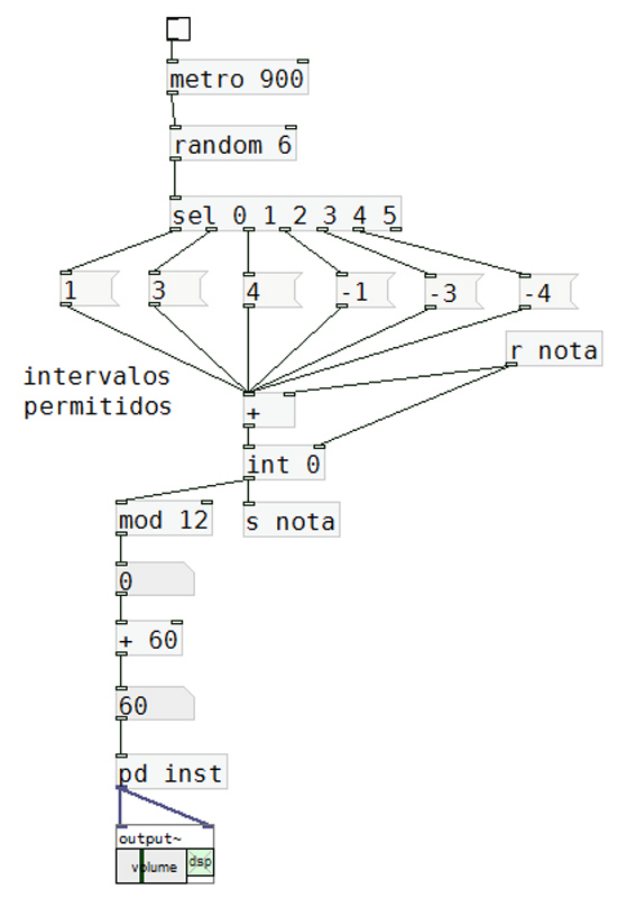


Vamos a modificar el *patch* anterior de manera tal que la sucesión de notas a generar respete alguna condición. La regla que vamos a establecer es que la distancia entre las notas solo puede ser de uno, de tres o de cuatro semitonos (intervalos o distancia que en música se denominan segunda menor, tercera menor y tercera mayor, respectivamente).

El semitono es la menor distancia entre dos notas (dos teclas consecutivas de un piano, por ejemplo). Es el intervalo más pequeño que se utiliza en la música tradicional occidental.

La figura siguiente (G.10.3.) muestra el *patch* que realiza esta tarea. Un objeto *random* decide si vamos a sumar o restar 1, 3 o 4 semitonos. Considerando que a partir de la suma podríamos superar el valor 11, y que a partir de la resta podríamos generar un valor menor que cero, aplicamos módulo 12. Lo cual equivale a obtener el valor absoluto (valor positivo) del resto de la división por 12. Por ejemplo, si de la suma obtenemos un 15, el resto de dividir por 12 es 3, que equivale a la misma nota de la octava superior, pero centrada entre 0 y 11. De este modo, recurrimos al azar, pero fijamos una regla relacionada con los intervalos permitidos en la sucesión de notas.

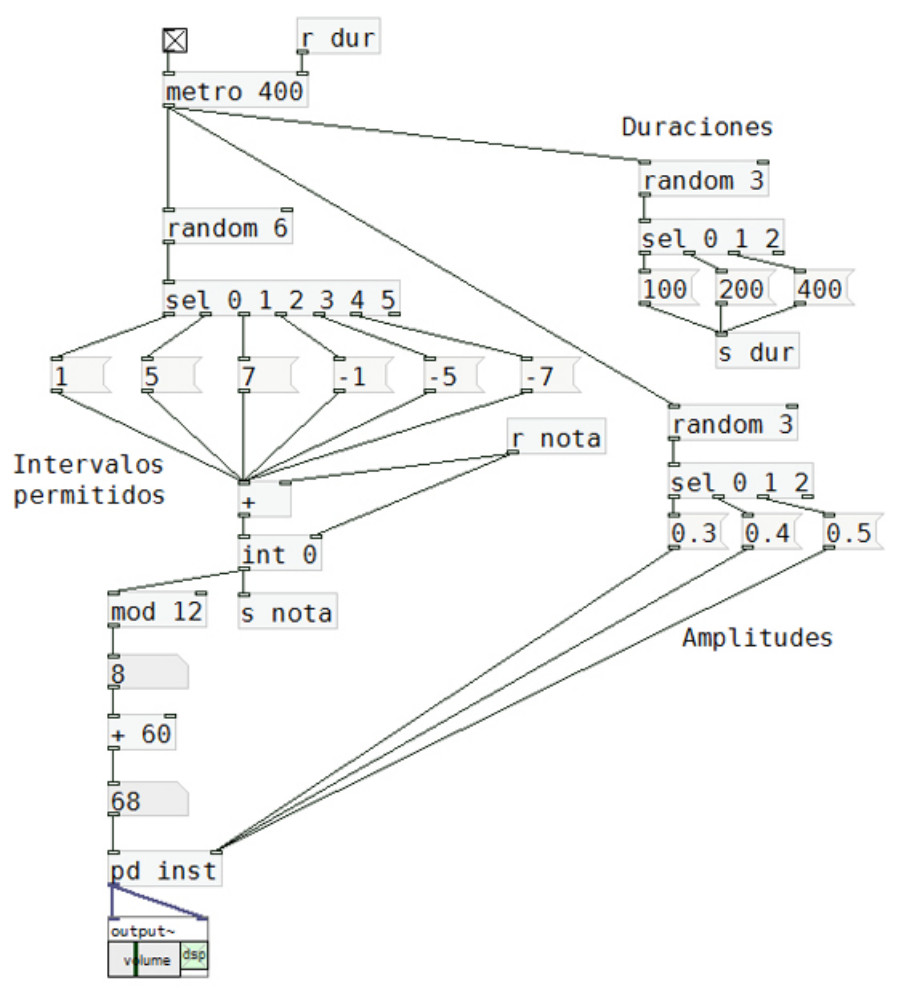
G.10.3. Azar limitado por reglas



La programación del *patch* de G.10.3. se encuentra en el archivo "74-regla interválica.pd".

Respecto a los tiempos de ataque de las notas, podríamos decidir que se produzcan de forma aleatoria, pero dentro de un repertorio preestablecido. Vamos a fijar para nuestro programa la duración menor, y luego dos y cuatro veces esa duración (por ejemplo 100, 200 y 400 milisegundos, de forma aleatoria). Para las amplitudes también fijamos 3 valores aleatorios y modificamos el *subpatch* del instrumento, para que pueda reproducir esos cambios. G.10.4. muestra el *patch* anterior transformado, donde también cambiamos los intervalos permitidos.

G.10.4. Azar limitado por reglas, aplicado a otros parámetros



La programación del *patch* de G.10.4. se encuentra en el archivo "75-reglas sobre otros parámetros.pd".

10.3. Cadenas de Markov

Las cadenas de Markov son el resultado de un proceso aleatorio que depende de un análisis estadístico, realizado previamente sobre el conjunto de valores posibles.

A fin de ejemplificar este proceso, vamos a utilizar un fragmento musical muy conocido: la melodía de la "Oda a la Alegría" de Ludwig van Beethoven, que forma parte de su *Novena Sinfonía*.

Las notas de la melodía, en el orden en que aparecen, son las siguientes:

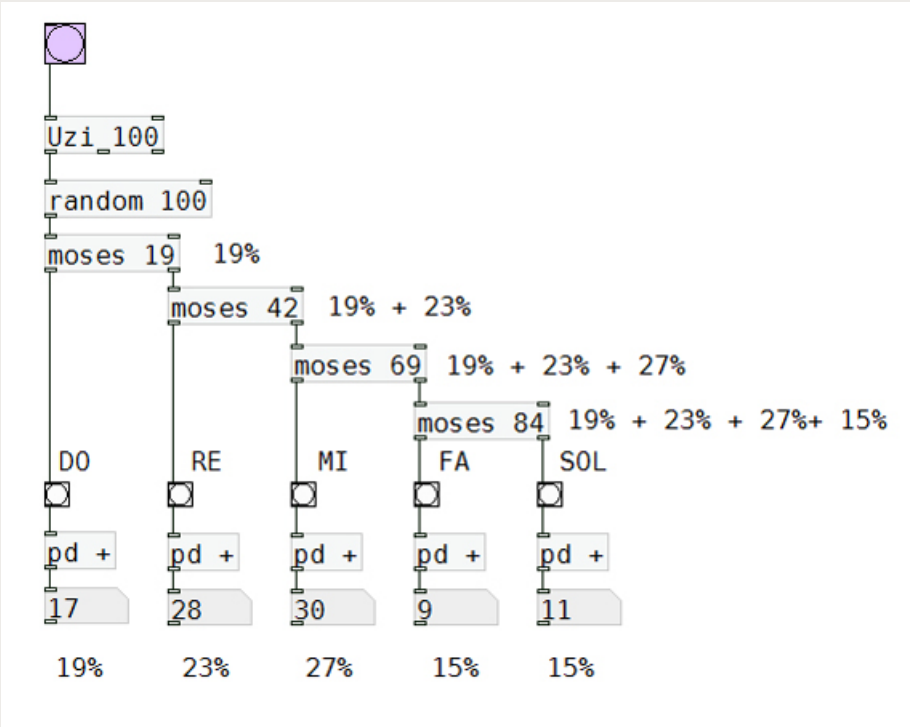
MI-FA-SOL-SOL-FA-MI-RE-DO-DO-RE-MI-MI-RE
MI-FA-SOL-SOL-FA-MI-RE-DO-DO-RE-MI-RE-DO

Podríamos decir, en términos estadísticos, que la nota Mi aparece 7 veces en un total de 26 notas. Significa que el porcentaje de recurrencia de esa nota sobre el total es de aproximadamente un 27 % (7 x 100 / 26). La nota SOL, por su parte, aparece 4 veces, y el porcentaje de aparición es del 15,3 %. La tabla siguiente ilustra los porcentajes para cada caso:

NOTA	APARICIONES	%
DO	5	19,23
RE	6	23,07
MI	7	26,92
FA	4	15,38
SOL	4	15,38
TOTAL	26	100,00

El *patch* de G.10.5. muestra cómo generar eventos aleatorios de acuerdo con una probabilidad. Mediante el objeto Uzi producimos 100 *bangs*, uno a continuación de otro y generamos 100 números al azar entre 0 y 99. Más abajo, observamos a varios objetos *moses*, que van a clasificar a los números generados aleatoriamente. A *moses* ingresa un valor, si ese valor es menor que el argumento, sale por la izquierda, y si es mayor o igual, sale por la derecha. Con el primer *moses* estamos separando números entre 0 y 18, y entre 19 y 99, o sea que lo que sale por la izquierda, representa el 19 % del total. El segundo *moses* separa el 23 % del resto, vale decir, ingresan números entre 19 y 99, y por la izquierda salen los números entre 19 y 41 (23 posibilidades), y así siguiendo. Los *subpatches* de la parte inferior (PD +) suman la cantidad de eventos ocurridos para cada nota. Y así es como asignamos a cada nota una probabilidad de aparición diferente.

G.10.5. Porcentaje de probabilidad



La programación del *patch* de G.10.5. se encuentra en el archivo "76-porcentaje de probabilidad.pd".

El *patch* también nos sirve para comprobar si la probabilidad se cumple o no. Ejecutando el programa repetidas veces podemos observar que, en un número alto de casos, los porcentajes coinciden con la cantidad de eventos registrados. Podríamos continuar esta idea y generar 26 notas al azar, pero dando a cada nota una probabilidad diferente. Y a este proceso de considerar probabilidades particulares para cada elemento del conjunto se lo denomina cadena de Markov de orden cero.

También podría resultar interesante considerar, no solo la cantidad de apariciones de cada nota en la melodía, sino analizar para cada una de ellas cuáles son sus sucesoras. Podemos escribir en la primera fila de una tabla las notas que suceden a DO, y qué cantidad de veces. En la segunda fila lo mismo para RE, y así siguiendo, como se observa a continuación:

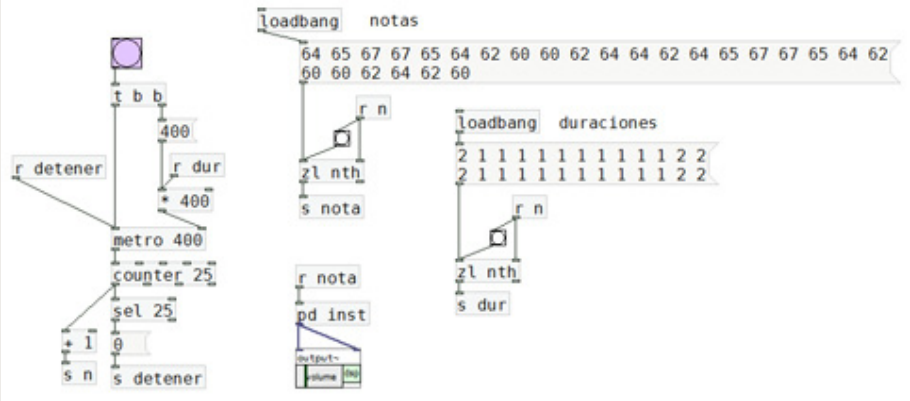
	DO	RE	MI	FA	SOL
DO	2	2			
RE	3		3		
MI		4	1	2	
FA			2		2
SOL				2	2

La nota DO es seguida dos veces por la misma nota y otras dos veces por RE, por ejemplo. Por lo cual, la posibilidad que los sucesores de DO sean DO o RE, es del 50 % para cada caso. La nota RE es seguida 3 veces por DO y tres veces por MI, con iguales porcentajes.

Este análisis nos podría servir para determinar las posibilidades que posee cada nota de ser sucesora de la nota anterior. En algunos casos vemos que la tabla está vacía, como en el caso de SOL, que nunca es seguida por DO, ni por RE, ni por MI, por lo cual, la posibilidad de que algo de esto ocurra es nula. Y, en otros casos, la posibilidad de que a una nota la siga otra podría ser del 100 %, si bien eso no sucede con la "Oda a la Alegría".

El *patch* siguiente (G.10.6.) corresponde a un secuenciador muy básico, capaz de almacenar notas de una melodía y sus duraciones, y reproducirlas en cualquier momento. Los mensajes que contienen a las notas y las duraciones son leídos secuencialmente por objetos *zl* con el argumento *nth*. Este secuenciador nos permite escuchar la "Oda a la Alegría" y nos va a servir para analizar auditivamente las variantes que obtengamos al utilizar las cadenas de Markov.

G.10.6. Secuenciador básico



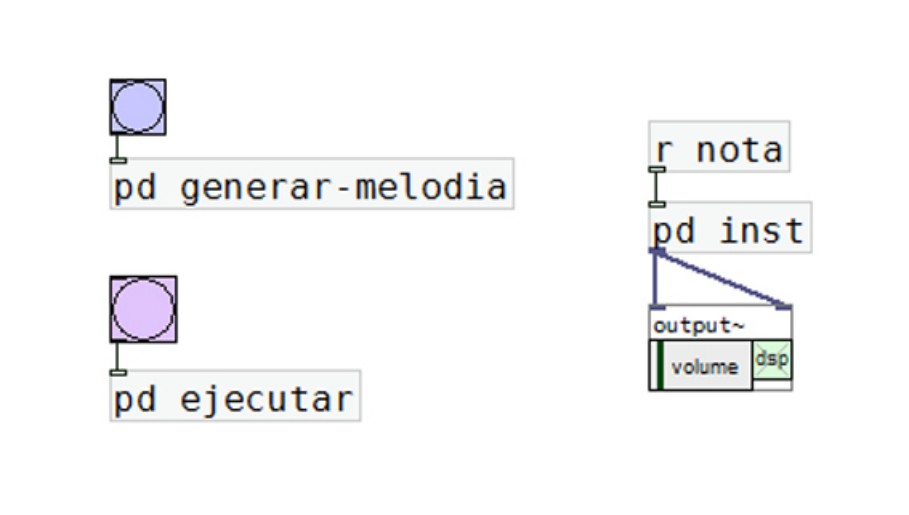
La programación del *patch* de G.10.6. se encuentra en el archivo "77-secuenciador.pd".

A partir del análisis estadístico que realizamos de la melodía de la "Oda a la Alegría" intentaremos obtener nuevas versiones, considerando las probabilidades de que a cada nota la sucedan las mismas que ocurren en el modelo. Al proceso que vamos a realizar lo denominamos "proceso de Markov de primer orden". De lo explicado se desprende que si considerásemos un número mayor de sucesores, por ejemplo dos, el proceso sería de segundo orden, y así siguiendo. Volviendo a la tabla anterior, vamos ahora a expresar la probabilidad de cada sucesor en porcentajes.

	DO	RE	MI	FA	SOL
DO	50 %	50 %			
RE	50 %		50 %		
MI		57 %	14 %	29 %	
FA			50 %		50 %
SOL				50 %	50 %

El *patch* de G.10.7. implementa las cadenas de Markov con el propósito de lograr variaciones de la melodía tomada como modelo.

G.10.7. Variación melódica usando cadenas de Markov

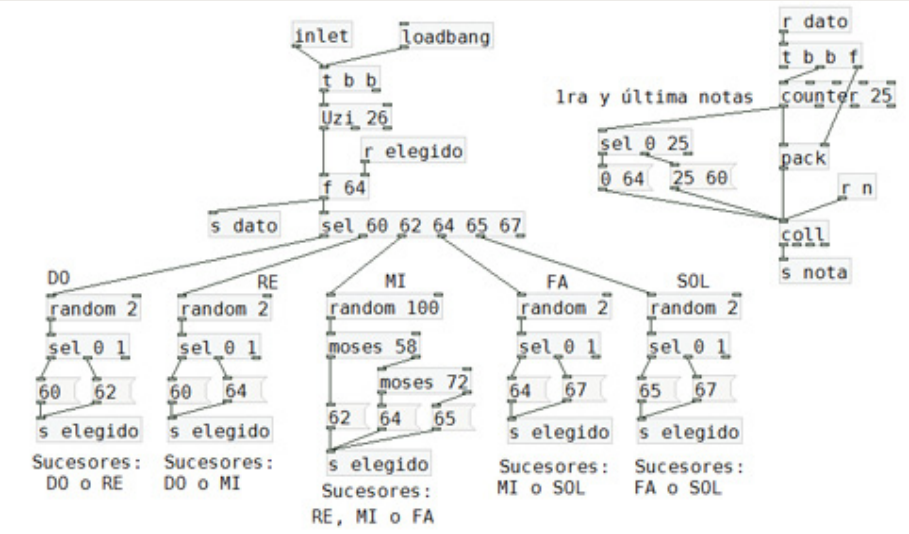


Para comenzar generamos la melodía, y luego la interpretamos, utilizando el secuenciador antes visto.

G.10.8. muestra el *subpatch* de generación melódica. Se trata de un algoritmo recursivo, en el cual se elige un sucesor para la primera nota, se almacena, y esa nueva nota reingresa en el circuito para buscar su propio sucesor. Las notas se guardan en un objeto *coll*, que recibe un mensaje con un número de orden (en nuestro caso 0 a 25) seguido del número de la nota a guardar. Para recuperar el dato solo se necesita enviar a este objeto el número de orden y devuelve la nota.

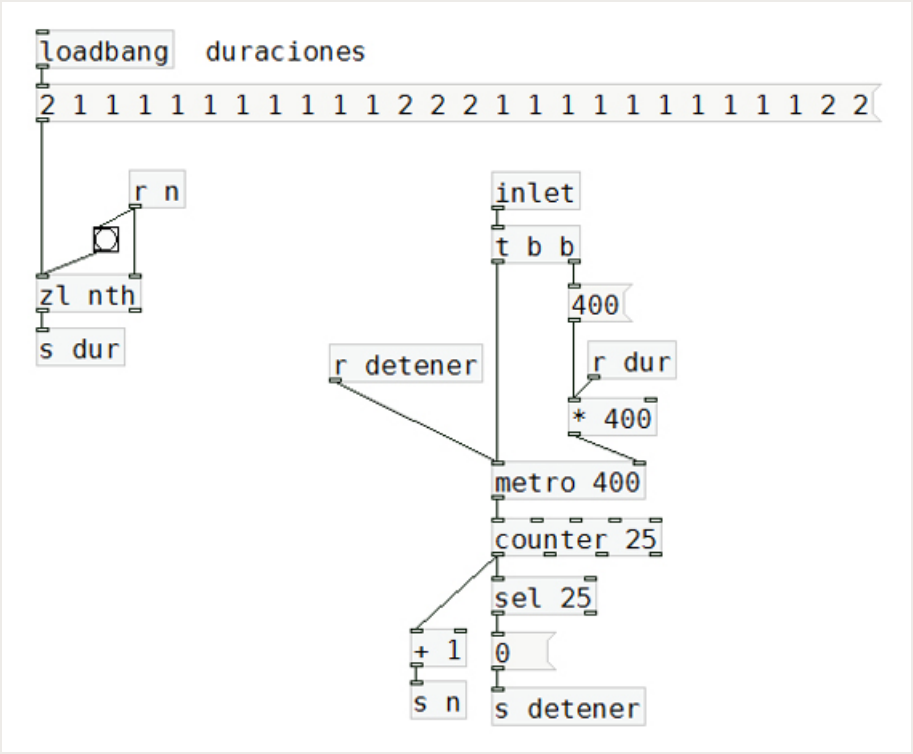
Los sucesores son elegidos teniendo en cuenta el análisis estadístico del modelo que resumimos en la tabla anterior. Finalmente, la primera y última notas se modifican para que sean un MI y un DO, respectivamente, pues dan mayor sensación de inicio y final.

G.10.8. *Subpatch* de generación melódica



El *subpatch* de secuenciación es similar al que programamos anteriormente. La diferencia reside en que no extraemos las notas de la melodía de una lista, sino del objeto *coll* donde fueron almacenadas.

G.10.9. *Subpatch* de secuenciación



La programación del *patch* de G.10.7. se encuentra en el archivo "78-cadenas de Markov.pd".

Romero Costas, M. (2008), "Algoritmos evolutivos y artes genético" en: Revista de Investigación Multimedia Nro. 2. IUNA, Buenos Aires, pp. 41-49.

Causa, E. y Sosa, A. (2008), "La computación afectiva y el arte interactivo" en: Revista de Investigación Multimedia Nro. 2. IUNA, Buenos Aires, pp. 51-60.

Actividad 17

a. Proyecte y desarrolle un trabajo final de cierta complejidad. El programa, realizado en PD, puede estar destinado al procesamiento de audio en tiempo real, o bien a la síntesis del sonido.

b. Puede, además, orientar su proyecto a la realización de una instalación sonora, que capture sonidos circundantes y los procese, o bien, que las transformaciones sean realizadas por los mismos espectadores, utilizando sensores de algún tipo.

Referencias bibliográficas

Unidad 1

Kreidler, J. *Programming Electronic Music in PD*, [en línea]. Berlín. Wolke Verlagsges. Mbh . 2009.
Disponible en: <http://pd-tutorial.com/>
[Consulta: 17 abril de 2013].
Traducción al castellano de Lucas Cordiviola.
Disponible en: <http://lucarda.com.ar/pd-tutorial/index.html>

Puckette, M. The Theory and Technique of Electronic Music, [en línea]. Singapur. World Scientific. 2007.
Disponible en: <http://crca.ucsd.edu/~msp/techniques/latest/book-html/>
[Consulta: 17 de abril de 2013].

Pure Data. <http://en.flossmanuals.net/pure-data/>
[Consulta: 17 de abril de 2013].
Descripción: Tutorial de PD.

Unidad 2

Pure Data. <http://en.flossmanuals.net/pure-data/>
[Consulta: 17 de Abril de 2013]. Descripción: Tutorial de PD.

Jordà, S. Manual de Introducción a PD, [en línea].
Disponible en: <http://www.tecn.upf.es/~sjorda/PD/IntroduccionPD3.pdf>
[Consulta: 17 de Abril de 2013].

Unidad 3

Cetta, P. (2004), "Procesamiento en tiempo real en la obra de Luigi Nono" en: *Altura-Timbre-Espacio. Cuaderno de Estudio* Nº 5. IIMCV-FACM-UCA, Buenos Aires, pp. 235-257.

–––––– (2005), "Procesamiento en tiempo real de sonido e imagen con PD-GEM" en: *Revista de Investigación Multimedia* Nº 1. ATAM-IUNA, Buenos Aires, pp. 28-34.

Jorda Puig, S. (1997), A*udio Digital y MIDI*. Anaya Multimeda, Madrid.

Moore, F. (1990), *Elements of computer music*. Prentice Hall, New Jersey.

Kreidler, J. (2009), "Programming Electronic Music in PD", [en línea]. Wolke Verlagsges.
Disponible en: <http://pd-tutorial.com/>
[Consulta: 17 abril de 2013].
Traducción al castellano de Lucas Cordiviola,
disponible en: <http://lucarda.com.ar/pd-tutorial/index.html>

Puckette, M. "The Theory and Technique of Electronic Music", [en línea]. Singapur. World Scientific. 2007.
Disponible en: <http://crca.ucsd.edu/~msp/techniques/latest/book-html/>
[Consulta: 17 de abril de 2013].

Unidad 4

Chowning, J. (1973), "The Synthesis of Complex Audio Spectra by Means of Frequency Modulation", en: *Journal of the Audio Engineering Society* 21 (7). 526-534.

Gabor, D. (1947), "Acoustical Quanta and the Theory of Hearing", en: *Journal Nature* Vol 159 Nro. 4044.
Nature Publishing Group, Londres, 591-594.

Gómez Gutiérrez, E. (2009). "Síntesis y procesado por granulación", en: Síntesis y Procesamiento del Sonido I. Barcelona.
ESMuC.
Disponible en: <http://www.dtic.upf.edu/~egomez/teaching/sintesi/sintesi1.html>
[Consulta: 17 de Mayo de 2013].

Jaffe, D. y Smith, J. (1983). "Extensions of the Karplus-Strong Plucked String Algorithm", en:
*Computer Music Journa*l. MIT Press. Massachussets, 7 (2), 56–69.

Karplus, K. y Strong, A. (1983), "Digital Synthesis of Plucked String and Drum Timbres", en:
omputer Music Journal 7(2). MIT Press, Massachussets, 43–55.

Kreidler, J. (2009), "Programming Electronic Music in Pd" ,en: Wolke Verlagsges.
Disponible en: <http://pd-tutorial.com/>
[Consulta: 17 abril de 2013].
Traducción al castellano de Lucas Cordiviola, disponible en:
<http://lucarda.com.ar/pd-tutorial/index.html>

Lebrun, M. (1979). "Digital waveshaping synthesis" en: *Journal of the Audio Engineering Society*, 27(4). 250-266.

Moore, F. (1990), *Elements of Computer Music*. Prentice-Hall, New Jersey.

Rocha Iturbide, M. (1999). *Les Techniques Granulaires dans la Synthèse Sonore*, en: París. Inédito.
Disponible en: <http://www.artesonoro.net/tesisgran/indicegran.html>
[Consulta: 17 de Mayo de 2013].

Unidad 5

Moore, F. (1990), *Elements of Computer Music*. Prentice-Hall, New Jersey.

Puckette, M. "The Theory and Technique of Electronic Music", [en línea]. Singapur. World Scientific. 2007.
Disponible en: <http://crca.ucsd.edu/~msp/techniques/latest/book-html/>
[Consulta: 17 de abril de 2013].

Smith, J. "Introduction to Digital Filters with Audio Applications", [en línea]. Universidad de Stanford. CCRMA. 2007.
Disponible en: <http://ccrma.stanford.edu/~jos/filters/>
[Consulta: 22 de abril de 2013].

Unidad 6

Moore, F. (1990), *Elements of Computer Music*. Prentice-Hall, New Jersey.

Puckette, M. "The Theory and Technique of Electronic Music", [en línea]. Singapur. World Scientific. 2007.
Disponible en: <http://crca.ucsd.edu/~msp/techniques/latest/book-html/>
[Consulta: 17 de Abril de 2013].

Smith, J. "Introduction to Digital Filters with Audio Applications", [en línea]. Universidad de Stanford. CCRMA. 2007.
Disponible en: <http://ccrma.stanford.edu/~jos/filters/>
[Consulta: 22 de Abril de 2013].

Unidad 7

Moore, F. (1990), *Elements of Computer Music*. Prentice-Hall, New Jersey.

Moore, F. (1978), "An Introduction to the Mathematics of Digital Signal Processing. Part II", en:
Computer Music Journal 2(2). MIT Press, Massachussets, 38-60.

Puckette, M. "The Theory and Technique of Electronic Music", [en línea]. Singapur. World Scientific. 2007.
Disponible en: <http://crca.ucsd.edu/~msp/techniques/latest/book-html/>
[Consulta: 17 de Abril de 2013].

Smith, J. "Mathematics of the Discrete Fourier Transform with Audio Applications", [en línea]. Universidad de Stanford. CCRMA. 2007.
Disponible en: <http://ccrma.stanford.edu/~jos/mdft/>
[Consulta: 22 de Abril de 2013].

Unidad 8

Blauert, J. (1997), *Spatial Hearing. The psychophysics of human sound localization*. The MIT Press, Massachussets.

Chowning, J. (1990), "Music from machines: Perceptual fusion and auditory perspective" en: *Report STAN* M-64. CCRMA, Stanford University, Palo Alto, 1-22.

_____(1971), "The simulation of moving sound sources", en: J.A.E.S. 19, 2-6.

Malham, J. (1995), "3D sound spatialization using Ambisonic techniques" en: *Computer Music Journal* 19(4), The MIT Press, Massachussets, 58-70.

Moore, F. (1990), *Elements of Computer Music*, Prentice-Hall, New Jersey.

Unidad 9

DE FURIA, S. (1989), *MIDI Programmers Handbook*. M&T Books, New York.

PENFOLD, R. (1992), *MIDI Avanzado*. RA-MA, Madrid.

Floss Manuals-Pure Data <http://en.flossmanuals.net/pure-data/sensors/game-controllers/. [
Consulta: 3 de Agosto de 2013]. Controladores para Pure Data.

Open Sound Control. <http://opensoundcontrol.org/>. [Consulta: 3 de Agosto de 2013]. Especificación OSC 1.0.

Unidad 10

Moore, F. (1990), *Elements of Computer Music*. Prentice-Hall, New Jersey.

Miranda, E. y Biles, J. (Eds.) (2007), *Evolutionary Computer Music*. Springer, Londres.

Nierhaus, G. (2008), *Algorithmic Composition-Paradigms of Automated Music Composition*. Springer, Londres.

Xenakis, I. (1963), *Formalized Music: Thought and Mathematics in Composition*. Indiana University Press, Bloomington IN.