ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA

Trabajo fin de Grado

Grado en Ingeniería Informática - Ingeniería del Software

SuperCollider

Realizado por
Jose Luis Caro Bozzino

Dirigido por
Victor Jesús Díaz Madrigal

Departamento
Lenguajes y Sistemas Informáticos

Sevilla, junio de 2020

Contenido

1.Introducción

Este documento corresponde a la memoria del Trabajo de Fin de Grado para la titulación de Ingeniería Informática – Ingeniería de Software.

A lo largo de este trabajo estudiaremos el lenguaje de programación musical *SuperCollider*; desarrollado en 1996 por James McCartney. En 2002 este lenguaje fue lanzado como software gratuito bajo la licencia pública general de GNU, y actualmente es mantenido por los propios usuarios, tratándose así de un proyecto completamente *opensource*.

Para la realización de este trabajo, estaremos utilizando la versión 3.10.00.

La plataforma ofrece tres componentes principales, los cuales se estudiarán en profundidad más adelante:

- **Scsynth:** Un servidor para audio en tiempo real. Aunque se suele usar desde SuperCollider, se puede acceder a este acceder a él de forma independiente. Incluye una gran cantidad de *UGens* o generadores unitarios, además de poder importar nuevos *UGens* programados en *C++*, facilitando la creación de *plugins* potentes para el lenguaje.
- **Sclang:** Un lenguaje de programación interpretado. Está enfocado, pero no limitado, al sonido. Controla *scsynth* mediante *Open Sound Control*. Puede usarse para composición algorítmica y secuenciación, conectar a hardware externo como controladores *MIDI*, puedes crear aplicaciones visuales o interfaces gráficas para este lenguaje…

  Es similar a *Ruby* o a *Smalltalk*, y su sintaxis recuerda a *Javascript* o *C*.

  Las extensiones para *SuperCollider* programadas por los usuarios se denominan *Quarks*.

- **Scide:** Un editor para *sclang* con un sistema integrado de ayuda.

A diferencia de otros lenguajes de programación musical, en *SuperCollider* nos encontramos frente a un lenguaje de programación orientado a objetos, cuyo dinamismo

y expresividad permite que cada vez más músicos lo utilicen como instrumento principal en sus conciertos o shows, que en este caso se denominan “sesiones de live-coding”, junto con científicos que ha encontrado en él una herramienta para desarrollar y experimentar en el campo de la investigación acústica.

Mi motivación principal a la hora de realizar este trabajo ha sido el poder estudiar por primera vez un lenguaje de programación musical, concepto que dista mucho de las materias impartidas en la titulación y que me resulta de especial interés ya que todas mis aficiones giran en torno a la música en directo y la grabación y edición musical. Un lenguaje de programación de estas características supone una herramienta más, bastante útil, con el fin de desarrollar mis conocimientos y poder experimentar en el campo del audio digital.

Para realizar este trabajo hemos tomado como fuentes de información principal la propia documentación de *SuperCollider*, disponible en su página web, así como videotutoriales alojados en *Youtube*.

2. Nociones básicas

En este capítulo trataremos varios conceptos básicos sobre el sonido desde los puntos de vista físicos, musicales y digitales, a fin de sentar una base que nos ayude a estudiar el lenguaje de programación en cuestión en mayor profundidad.

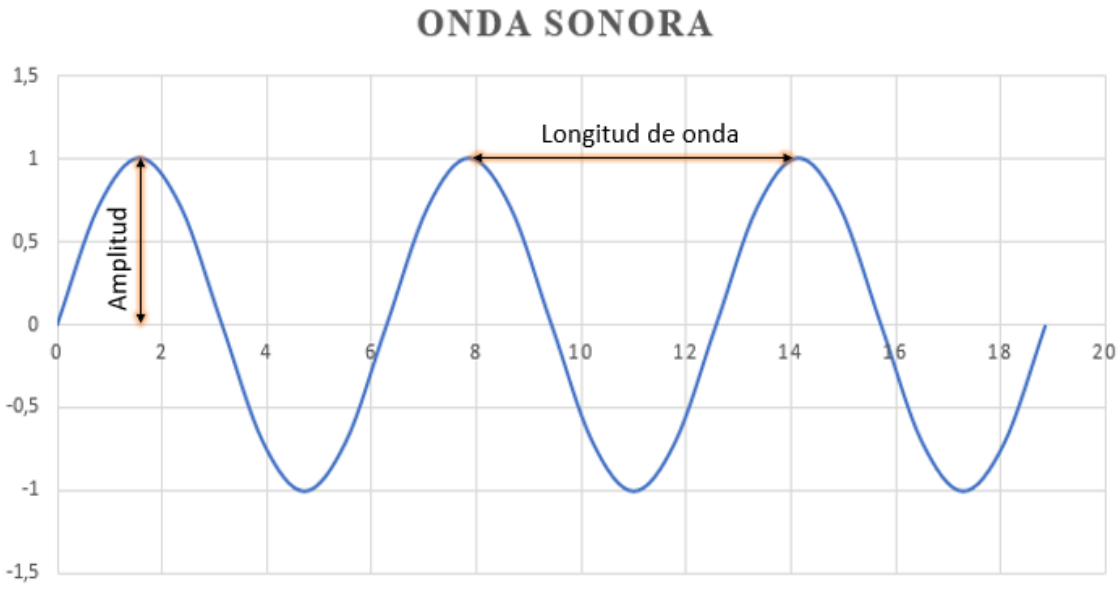
2.1 Física del sonido

Desde un punto de vista físico podemos definir el sonido como la propagación de ondas que se originan por la vibración de un cuerpo a través de un fluido o un medio elástico, generalmente el aire. Estas ondas comparten las características de las ondas mecánicas:

• **Frecuencia (f):** Medida en Hercios (Hz), describe la cantidad de ciclos o perturbaciones completadas por unidad de tiempo, normalmente medida en segundos. Representa la altura del sonido, ya que en base a su frecuencia distinguimos sonidos graves y agudos. Entre los 20 y los 20.000 Hz consideramos frecuencias audibles, puesto que por encima y por debajo de esa franja, los sonidos no son perceptibles por el oído humano.

• **Amplitud:** Es la distancia entre el punto más alto y el más bajo de una onda. Representa la intensidad del sonido, lo que llamaríamos comúnmente "volumen".

• **Longitud de onda:** Mide la distancia que recorre una onda en un periodo concreto de tiempo. Se aplica en el caso de ondas periódicas. También es conocida como periodo espacial, que es el inverso de la frecuencia y representa el tiempo que tarda una onda en completar un ciclo.

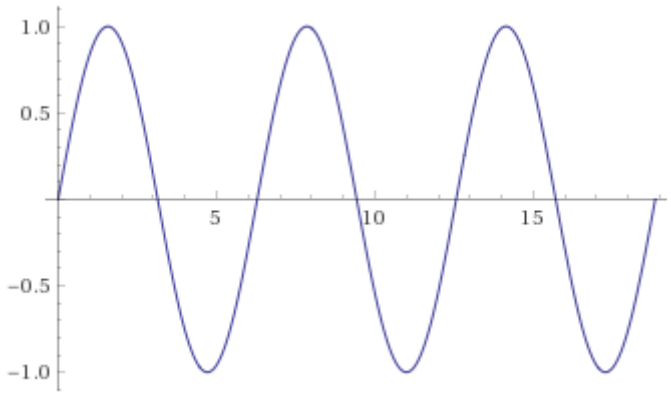


2.2 Propiedades acústicas de la música

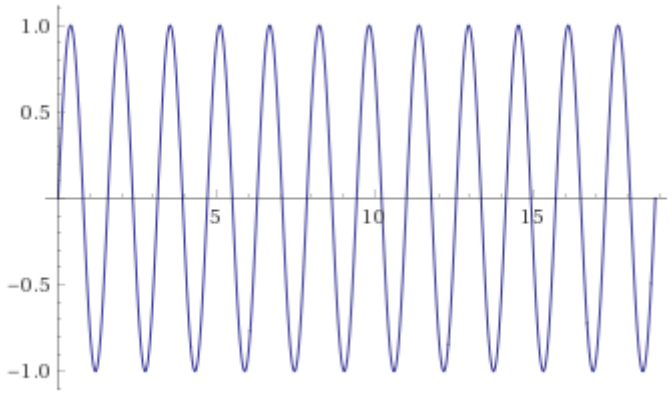
Dentro de un sonido podemos encontrar cuatro propiedades que percibimos de forma subjetiva y que vienen derivadas de las características físicas del sonido. Estas propiedades son: altura, duración, intensidad y timbre.

Gracias a estas propiedades podemos distinguir un sonido agradable, producido por una vibración armónica y regular, de un ruido.

• **Altura o tono:** Es una propiedad que percibimos de forma subjetiva y que deriva de la frecuencia. Mientras mayor frecuencia decimos que un sonido tiene un tono más “alto” y viceversa. Nos referimos a estos sonidos respectivamente como “agudos” y “graves”. A diferencia de la frecuencia, la altura es subjetiva y por tanto no es cuantificable, lo que hace que dependiendo del receptor y la situación se perciba de forma distinta.

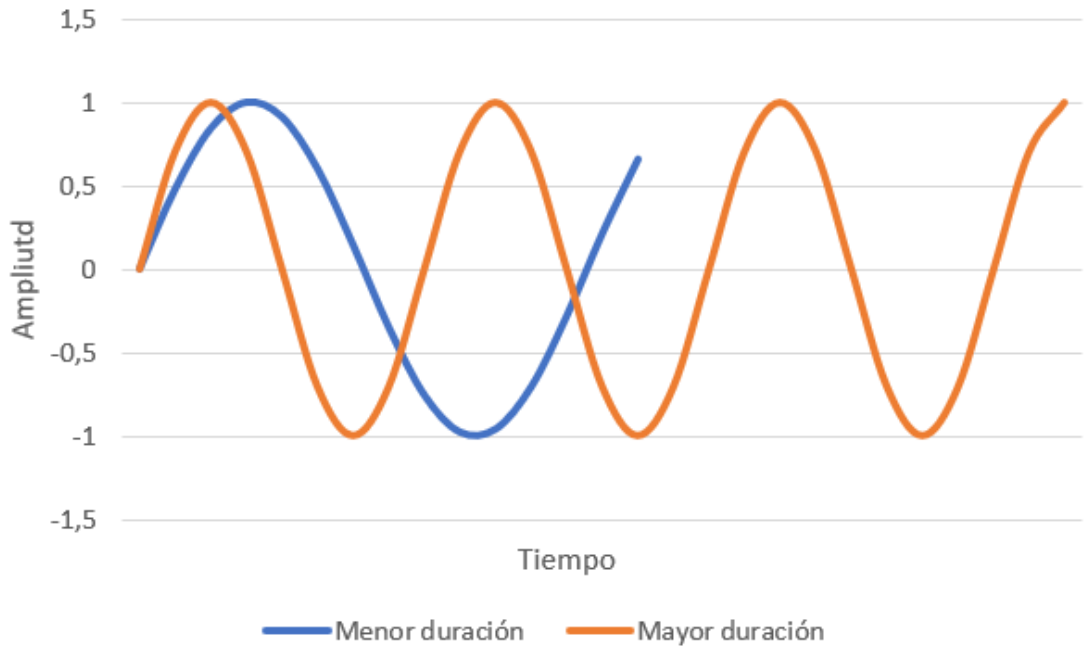


Sonido grave



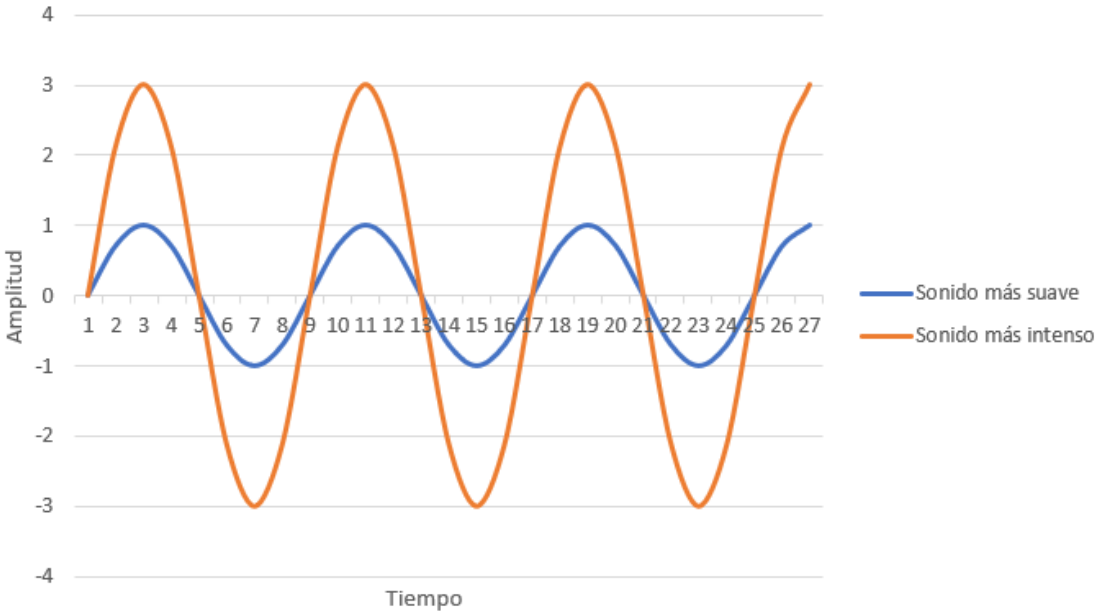
Sonido agudo

• **Duración:** Representa el tiempo que se extiende un sonido desde su inicio hasta su extinción. En función de la duración existen sonidos largos, medios, cortos, muy cortos… Estos sonidos de duración variable combinados originan ritmos.

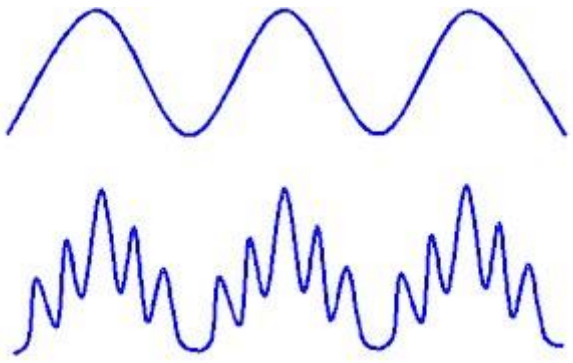


• **Intensidad:** Es la cantidad de energía contenida en un sonido; hace que podamos escucharlo desde una distancia mayor o menor. La intensidad deriva de la amplitud y la potencia acústica de un sonido, y se mide en decibelios (dB). Un sonido es audible a partir de los 0 dB y comienza a causar dolor y malestar al oído humano a partir de los 130 dB. En la propagación real, cambios físicos en el aire

como la humedad, la presión o la temperatura hacen que el sonido se amortigüe o se disperse.



• **Timbre:** Permite que distingamos una misma nota tocada por instrumentos musicales distintos. Representa la forma de onda, que nos ayuda a distinguir la fuente de sonidos con la misma frecuencia e intensidad. Esta propiedad agrupa las tres anteriores y solo se puede describir, no medir.



Ondas igual frecuencia (nota) pero distinto timbre

3. Audio digital

Con las características principales del sonido ya definidas, es el momento de plantearnos cómo lograr convertirlas en información procesable por un ordenador y poder manipularlas.

Las técnicas que se emplean se basan en imitar el funcionamiento del oído humano y convertir esas ondas mecánicas en impulsos eléctricos que podamos almacenar como tal o convertir en señales digitales.

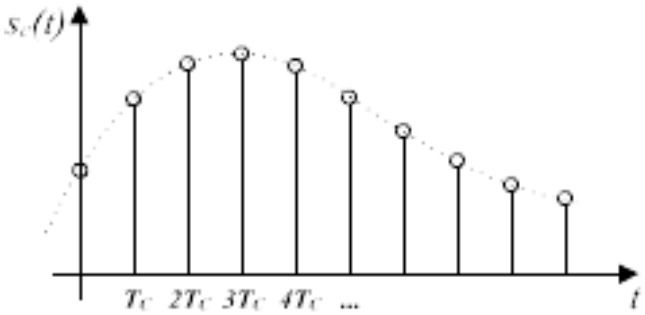
Dicha codificación consiste en una secuenciación de valores enteros que se obtienen mediante el muestreo y la cuantificación digital.

3.1 Muestreo y cuantificación

El **muestreo** es el proceso por el cual se captura la amplitud de la señal eléctrica en intervalos regulares de tiempo, denominados “tasa de muestreo”.

Para abarcar todo el espectro audible por el oído humano suele ser suficiente con una tasa de 40kHz, lo que implica 40000 muestras por segundo de audio capturado.

Por otro lado, la **cuantificación** se refiere al proceso de transformar las muestras fijadas durante el muestreo, que suele ser un valor de tensión, en un valor entero dentro de un rango determinado. Con una cuantificación lineal de 8 bits discriminamos 256 niveles equidistantes de señal.



Muestreo digital de una señal de audio

3.2 Formatos más importantes

Una vez definida la forma de transformar ondas acústicas analógicas en señales digitales que pueden ser procesadas, cabe destacar los formatos más importantes de estas.

• **Formatos PCM:** Denominados *PCM* por sus siglas (*Pulse Coded Modulation*), albergan en su totalidad la información que se obtuvo del convertidor analógico a digital sin omitir nada, otorgándole una calidad mejor al resto de formatos.

Dentro de los *PCM* tenemos los formatos *WAV, AIFF*, *SU*, *AU* y *RAW*, cuyo encabezado posee unos 1000 bytes al comienzo del archivo.

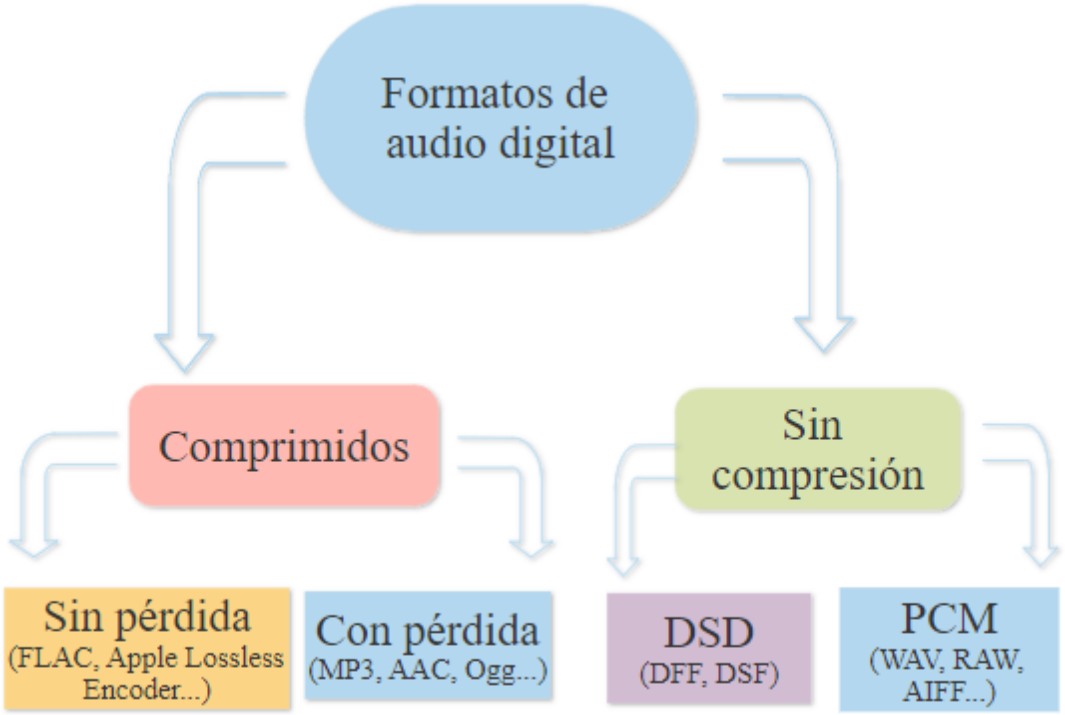
• **Formatos DSD:** Los formatos *Direct-Stream Digital* (marca registrada de Sony Corp. Y Philips) se basan en tecnologías de registro y reconstrucción de señales de audiofrecuencia, que se usaron por primera vez en el formato *Super Audio CD* y actualmente en los formatos *DSF* y *DFF*.

Esto se hace mediante el método de *Pulse Density Modulation* (*PDM*), el cual se diferencia del *PCM* en poseer una profundidad de bits bajísima, de tan solo 1 bit, en contraste con una gigantesca frecuencia de muestreo de 2,8224 Mhz.

• **Formatos comprimidos:** Para evitar usar tanta memoria como los formatos mencionados anteriormente, existen formatos que comprimen la información, como el archiconocido *MP3*, *AAC*, *Ogg*…

Estos formatos están basados en algoritmos que eliminan de las pistas de audio aquella información que no es perceptible por nuestro oído, llegando a reducir el espacio en memoria de un archivo hasta en más de una decena de veces en comparación con el mismo archivo en formato *PCM*.

Esta pérdida de información hace que a estos formatos se les considere formatos comprimidos "con perdida", aunque también existen formatos comprimidos sin pérdida o *"lossless"* como pueden ser *FLAC* o el *Apple Lossless Encoder*, cuyo tamaño ronda la mitad de un archivo PCM.



• **Formatos descriptivos:** Más conocido como archivos *MIDI*, no pertenecen técnicamente al audio digital, pero sí a la informática musical.

Un archivo *MIDI* no almacena sonido capturado por una grabadora de ningún tipo, sino que está compuesto por indicaciones para que cualquier dispositivo *MIDI* como podría ser un sintetizador, un *launchpad* o una guitarra *MIDI* interpreten una serie de notas y acciones; haciendo que sea un equivalente moderno a las partituras, con los nombres de los instrumentos, las notas, tiempos y más indicaciones.

4. MIDI

En este capítulo trataremos más a fondo qué es el estándar *MIDI* y cómo funciona.

El estándar *MIDI* (*Musical Instrument Digital Interface*) surge en 1983 como un convenio del que resultó la *MMA* (*MIDI Manufacturers Association*).

Dicho estándar tecnológico describe un protocolo, una interfaz digital y conectores con el objetivo de hacer que ciertos instrumentos musicales electrónicos y computadores puedan relacionarse y comunicarse entre ellos.

Una sola conexión de este tipo tiene la capacidad de transmitir hasta 16 canales de información que se pueden conectar a distintos equipos.

Los mensajes de evento descritos en el protocolo *MIDI* especifican notación musical, tono, velocidad, señales de control de parámetros musicales (dinámica, vibrato, tempo…).

A continuación, se muestra una tabla que describe los comandos admitidos en estos mensajes:

Mensaje	***Código***	***Parámetro 1***	***Parámetro***
Note Off	0x8	Número de nota	Velocidad
Note On	0x9	Número de nota	Velocidad
Note Aftertouch	0xA	Número de nota	Presión
Controller	0xB	Número de controlador	*Controller Value*
Program Change	0xC	Número de programa	-
Channel Pressure	0xD	Presión	-
Pitch Bend	0xE	*Pitch Value (LSB)*	*Pitch Value (MSB)*

Los mensajes *Note On* y *Note Off* están codificados en un rango que admite las 88 teclas de un piano más algunas notas extra que no existen en instrumentos analógicos:

Note Numbers												
Octava	**C**	**C#**	**D**	**D#**	**E**	**F**	**F#**	**G**	**G#**	**A**	**A#**	**B**
-1	0	1	2	3	4	5	6	7	8	9	10	11
0	12	13	14	15	16	17	18	19	20	21	22	23
1	24	25	26	27	28	29	30	31	32	33	34	35
2	36	37	38	39	40	41	42	43	44	45	46	47
3	48	49	50	51	52	53	54	55	56	57	58	59
4	60	61	62	63	64	65	66	67	68	69	70	71
5	72	73	74	75	76	77	78	79	80	81	82	83
6	84	85	86	87	88	89	90	91	92	93	94	95
7	96	97	98	99	100	101	102	103	104	105	106	107
8	108	109	110	111	112	113	114	115	116	117	118	119
9	120	121	122	123	124	125	126	127				

Estos mensajes se envían a través de un cable *MIDI* a los demás equipos conectados, pero también pueden ser grabados en secuenciadores, tanto software como hardware, con el fin de poder editar esta información a posteriori.

La mayor ventaja de este formato es poder codificar composiciones completas en un espacio de tan reducido como un par de kilobytes, así como poder manipular y editar las distintas pistas asignadas a cada instrumento.

Además, cabe recalcar que a raíz de este protocolo han ido apareciendo una serie de extensiones que permiten desde controlar el transporte de dispositivos hardware de grabación (*MIDI Transport Control*) hasta poder sincronizar máquinas, sonidos y pirotecnia (*MIDI Show Control*) para exhibiciones de museo, escenarios de rodaje…

5. Supercollider

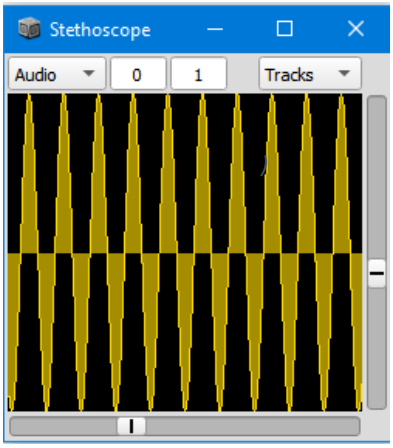
5.1 UGens

En *SuperCollider*, el concepto de *UGen* ó *Unit Generator* define a los objetos que producen señales; sus nombres siempre comienzan por mayúscula y a su conjunto nos referiremos como *patch.*

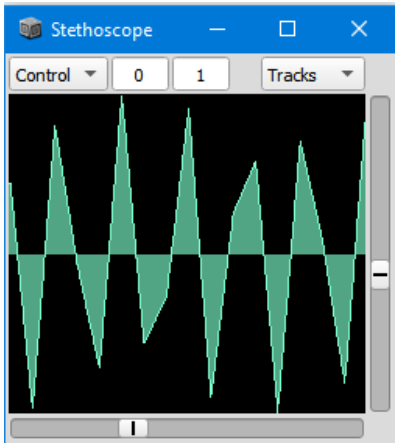
Dentro de los *UGens* encontramos dos tipos:

- **Audio Rate (.ar):** Cuando un *UGen* recibe un mensaje *ar*. lo ejecuta a una velocidad de 44k muestras por segundo. Si el *UGen* va a ser parte de una cadena de audio que vaya a ser escuchada se le debe enviar un mensaje de este tipo.
- **Control Rate (.kr):** En este caso, el *UGen* corre a velocidad de control. Producen una muestra por cada 64 producidas por el *UGen* a velocidad de audio. Se usan como moduladores para dar forma a la señal de audio, y son más baratos computacionalmente.

A continuación, tenemos una representación gráfica, producto de visualizar mediante la función *scope* un oscilador sinusoidal a 440 Hz.



SinOsc.ar(440)



SinOsc.kr(440)

5.1.1 Osciladores

Los osciladores en *SuperCollider* se rigen por una serie de argumentos que actúan como los parámetros que definen el comportamiento de estos.

Los más comunes son:

- **Frecuencia:** Su valor por defecto es de 440Hz, que representa la nota La índice 5.
- **Fase: Por defecto** vale 0, representando al inicio del ciclo. Su valor puede oscilar de 0 a 2pi.
- **Amplitud:** Su valor por defecto es 1, no alterando la señal.
- **Suma:** Es un número que se le suma a la señal. Se aplica justo después de multiplicar la señal por el valor del parámetro amplitud. Su valor por defecto es 0.

Aunque existen *UGens* que poseen argumentos propios, estos son cuatro son generales para casi todos.

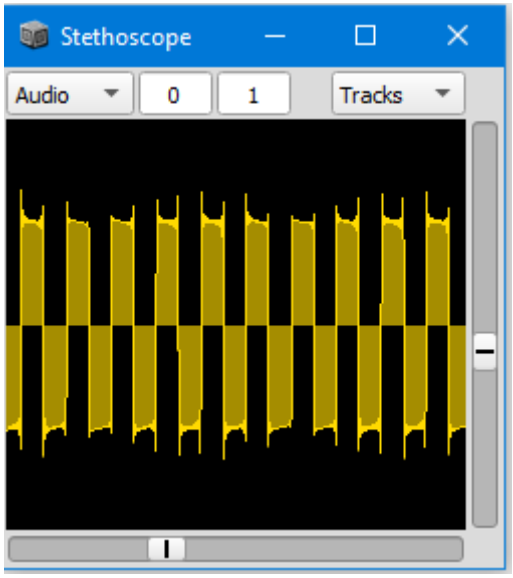
A continuación, diseccionaremos los osciladores más comunes que podemos encontrar en *SuperCollider*.

- **SinOsc:** Es un oscilador de onda senoidal. Funciona tanto con *.ar* como con *.kr* y recibe como atributos (en este orden): frecuencia, fase, amplitud, suma.

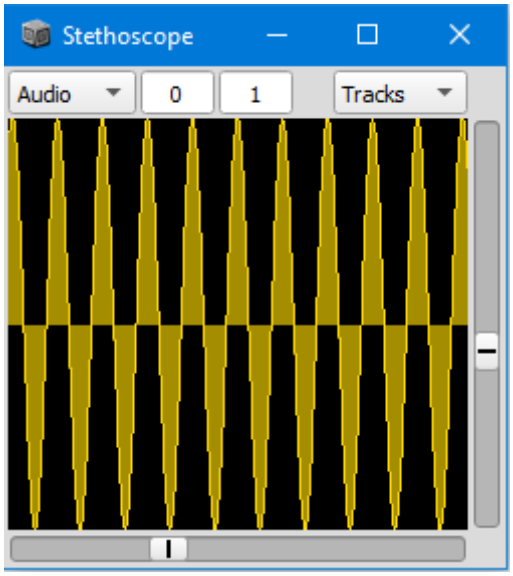
- **Pulse:** Oscilador de onda cuadrada. Solo funciona con *.ar*. Recibe como argumentos: frecuencia, ancho de banda, amplitud y suma.
  El ancho de banda está en el intervalo abierto (0, 1) y gestiona el timbre de la señal. Por defecto su valor es 0,5.

- **LFTri:** Oscilador de onda triangular. Funciona tanto con *.kr* como con *.ar*. Recibe como parámetros: frecuencia, fase, amplitud y suma

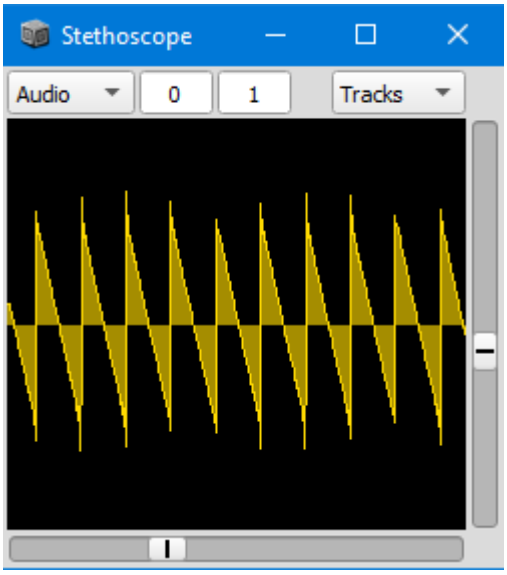
- **Saw:** Es un oscilador de dientes de sierra. Solo funciona con *.ar* y recibe como parámetros: frecuencia, amplitud y suma.



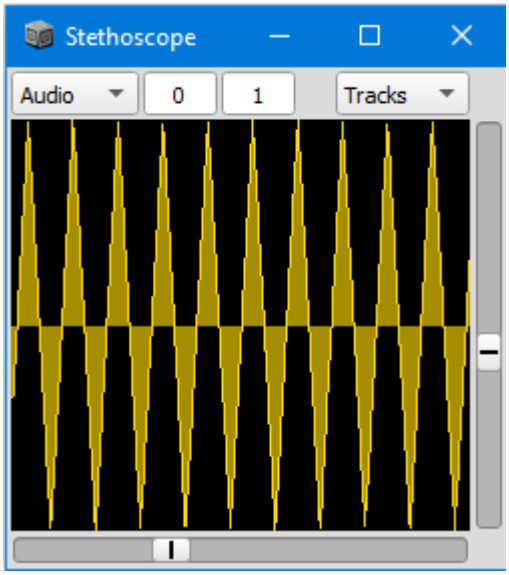
Pulse



SinOsc



Saw



LFTri

5.1.2 Generadores de ruido

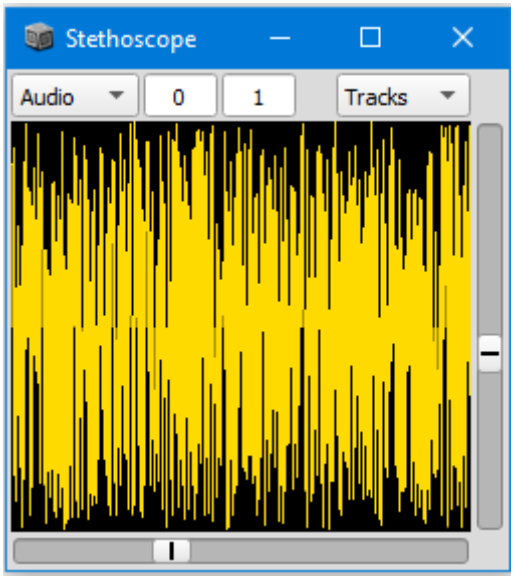
Por último. hablaremos de los generadores de ruido más utilizados dentro de *SuperCollider*.

- **WhiteNoise:** Es un generador de señales aleatorias, que produce señales en todas las frecuencias y con la misma potencia.

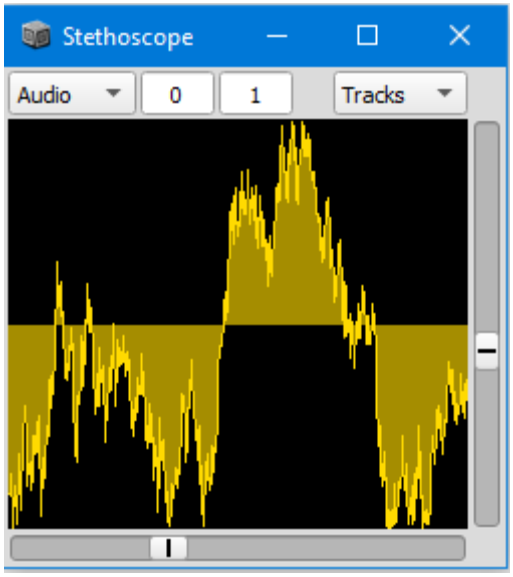
- **PinkNoise:** En este caso, las señales aleatorias generadas decrecen en un factor de 3dB por cada octava, misma proporción en la que aumenta el ancho de banda.

- **BrownNoise:** Es similar al *PinkNoise*, solo que en este caso decrece a un nivel de 6dB por cada octava.

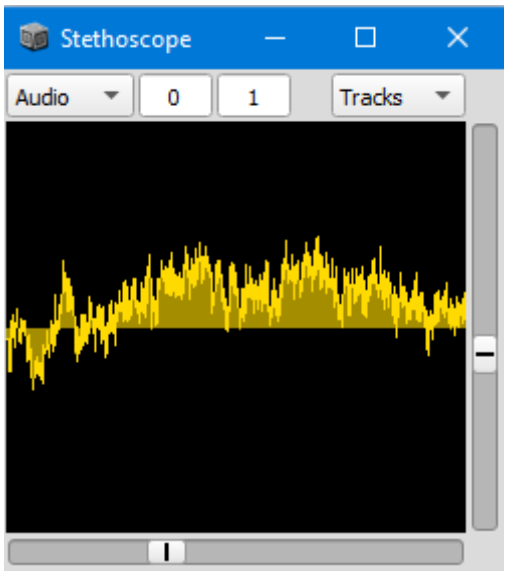
- **Dust:** Este *UGen* genera impulsos aleatorios.



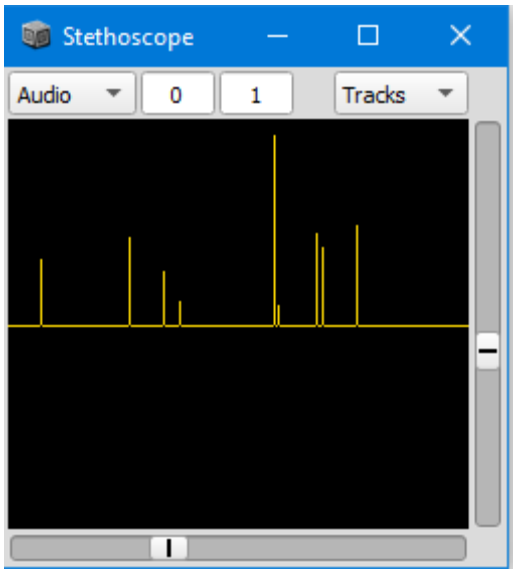
WhiteNoise



BrownNoise



PinkNoise

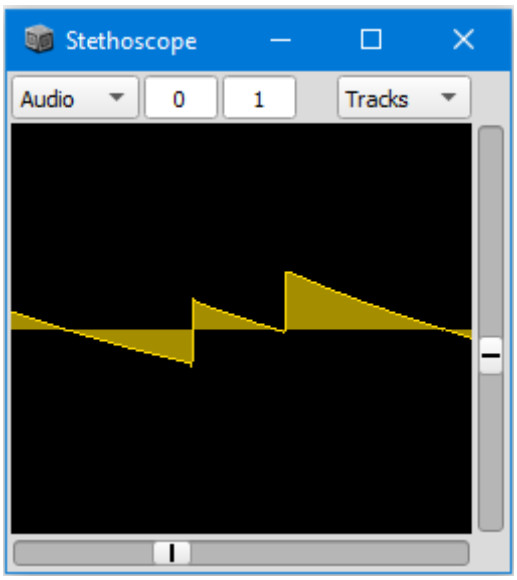


Dust

5.1.3 Operaciones con UGens

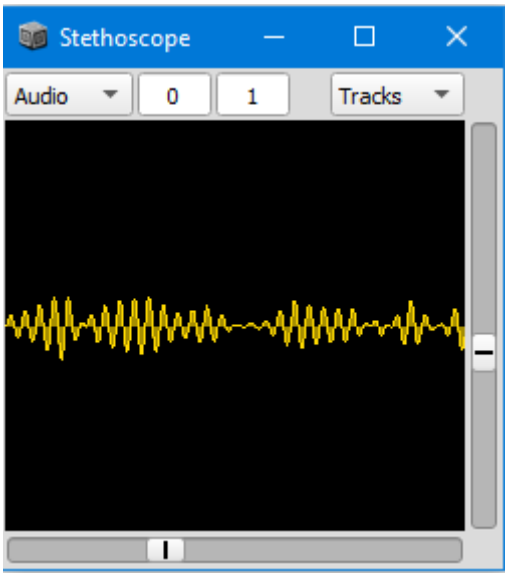
Una vez comprendido el concepto de *UGen* que maneja *SuperCollider* nos encontramos con que podemos crear nuevos timbres a partir de estos mediante sumas y multiplicaciones.

Las sumas dan como resultado un nuevo timbre mediante síntesis aditiva, en el que se siguen percibiendo los dos sonidos previos. Las amplitudes de los dos *UGens* se suman, no debiendo superar el valor 1.0.



{Saw.ar(30,0.3)+Saw.ar(70,0.3)}

Por otro lado, al multiplicar dos *UGens* obtenemos un timbre de mayor complejidad.



*{SinOsc.ar(1360,0,0.6)*WhiteNoise.kr(0.3)}*

5.2 Filtros

En electrónica, un filtro es un elemento que en función de uno parámetros discrimina una señal de entrada, realizando cambios en su salida.

Mientras nos refiramos a los filtros en *SuperCollider* trataremos como primer parámetro la señal a filtrar y tras esta la frecuencia de corte.

A continuación, estudiaremos los tres filtros principales que encontramos en este lenguaje de programación. Los tres funcionan tanto en *.ar* como en *.kr*, teniendo como único requisito que tanto la señal de entrada como el filtro a aplicar tenga el mismo *rate*.

- **HPF (High Pass Filter):** El filtro paso alto atenúa todas las frecuencias inferiores a la de corte, solo permitiendo pasar las que estén por encima de esta.
  Recibe como argumentos una señal de entrada, una frecuencia de corte, una amplitud y la suma.

- **LPF (Low Pass Filter):** Los filtros paso bajo funcionan al contrario que los filtros *HPF*; solo permiten pasar las frecuencias inferiores a la frecuencia de corte que recibe como argumento. Estos filtros también reciben los mismos argumentos que su contraparte.

- **BPS (Band Pass Filter):** Los filtros paso banda solo permiten el paso de frecuencias que se encuentren dentro de una banda concreta y limitada por una frecuencia suelo y una frecuencia techo. Entre ambas se encuentra la frecuencia de corte.

  Para calcular las cotas inferiores y superiores aplicamos las siguientes fórmulas:

$$Cota\ inferior = \frac{Frecuencia\ de\ corte - Ancho\ de\ banda}{2}$$

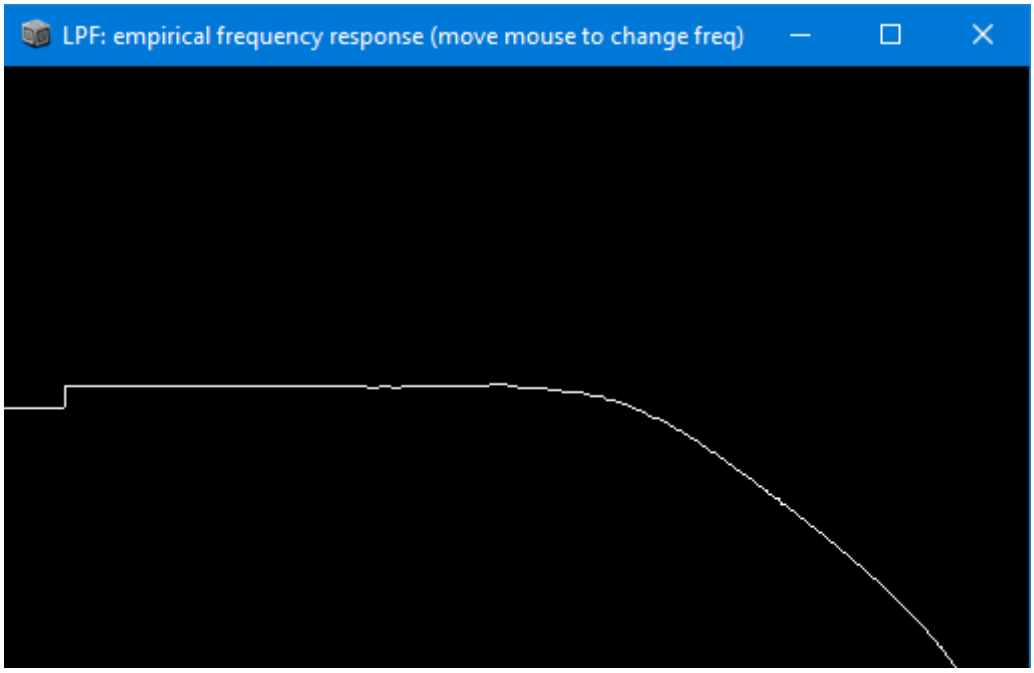
$$Cota\ superior = \frac{Frecuencia\ de\ corte + Ancho\ de\ banda}{2}$$

  De esta forma, si conocemos las cotas que buscamos es tan sencillo como aplicar las siguientes fórmulas para hallar la frecuencia de corte y el ancho de banda buscados:

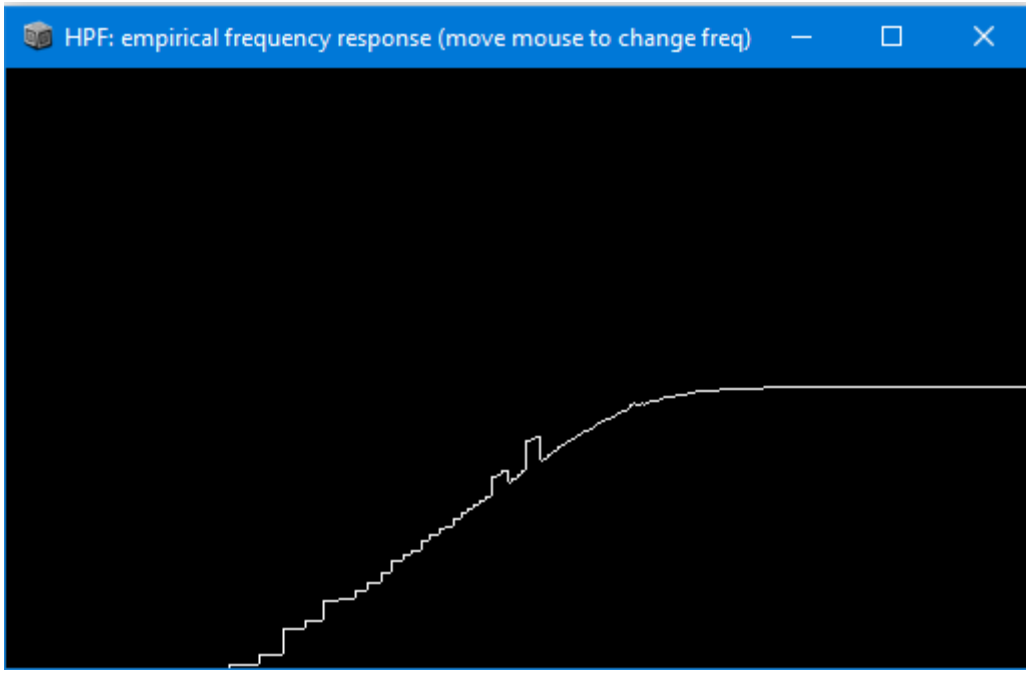
$$Ancho\ de\ banda = Cota\ superior - Cota\ inferior$$

$$Frecuencia\ de\ corte = \frac{Cota\ inferior + Ancho\ de\ banda}{2}$$

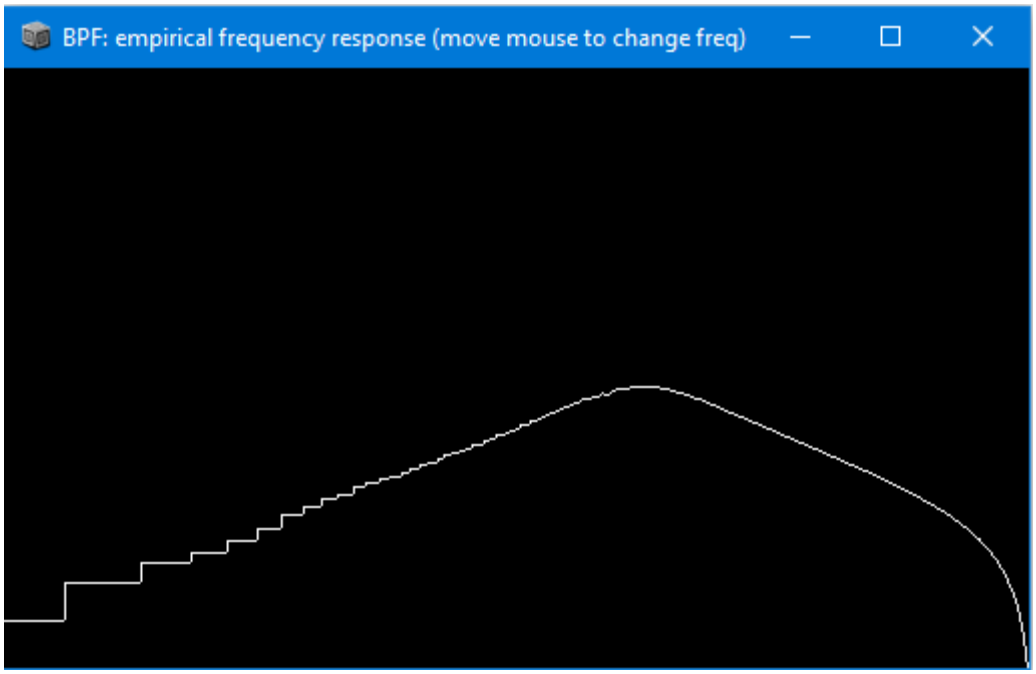
  Este filtro recibe como parámetros una señal de entrada, una frecuencia de corte, *qr* (que representa al ancho de banda partido de la frecuencia de corte), una amplitud y una suma.



Filtro paso baja



Filtro paso alto



Filtro paso banda

5.3 Envolvente

La envolvente representa la persistencia en el tiempo de un sonido frente a su amplitud, es decir, con qué intensidad se escucha el timbre a lo largo del ciclo de vida que dura desde que se produce dicho sonido hasta que se extingue.

Por poner un ejemplo más sencillo de comprender podríamos pensar en el sonido que produce un bombo de batería al golpearlo con una baqueta; este sonido nace siendo muy intenso, pero rápidamente se diluye hasta reducir su amplitud de forma casi total.

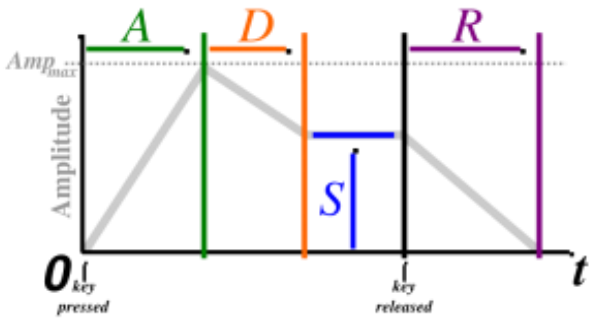
Una vez hemos definido el concepto de envolvente, procederemos a estudiar sus cuatro componentes más comunes, popularmente denominados *"ADSR"*:

- **Attack:** El ataque representa cuánto tarda el sonido en alcanzar su amplitud máxima una vez se ha producido. En los instrumentos mecánicos este punto es prácticamente instantáneo.

- **Decay:** El decaimiento es la cantidad de tiempo que tarda el sonido en estabilizarse una vez pasado su punto de amplitud más alto.

- **Sustain:** El sostenimiento mide cuánto tiempo dura esa estabilización que ocurre tras el decaimiento hasta que la nota se libera (deja de sonar).

- **Release:** La liberación representa cómo de rápido se desvanece una nota una vez termina (dejamos de pulsar una tecla de un piano, por ejemplo). Este tiempo puede ser más largo si utilizamos un sintetizador con algún efecto de *sustain*.



Representación gráfica de ADSR

Cabe destacar que no todas las envolventes necesitan de estas cuatro propiedades; puede haber envolventes que solo dependan de dos de ellas.

En *SuperCollider* manipulamos las envolventes mediante la clase *EnvGen,* la cual estudiaremos en modo *kr* (*control rate*) ya que la usaremos para modificar señales.

A esta clase se la llama mediante *EnvGen.kr*(*envolvente*, *gate*, *doneAction*). Estos tres parámetros funcionan de la siguiente forma:

- **Envolvente:** Podemos utilizar alguna de las muchas envolventes incluidas en SuperCollider. Por ejemplo; *Env.adsr(attack, decay, volumen, release)*, se usa para sonidos sostenidos y recibe como tercer argumento volumen en vez de tiempo. También tenemos *Env.perc(attack, release)* que usamos como envolvente para percusión y por último tenemos el ejemplo de *Env.asr(attack, volumen, release)* en el cual la nota se mantiene hasta que nosotros lo indiquemos y ya procede con su liberación.

- **Gate:** Activa la envolvente y la mantiene mientras su valor sea superior a 0. Si la envolvente tiene una duración fija, como es en el caso de *Env.perc*; donde su duración es igual a la suma del *attack* y el *release*, sirve simplemente como un activador. Por otro lado, si usamos una envolvente con *sustain*, esta se aplica hasta que el valor sea 0, en cuyo momento comienza la fase de *release*.
- **doneAction:** Es un entero que representa la acción que se realizará una vez se cierre la envolvente. Las acciones posibles van del 0 al 15 y se detallan en la siguiente tabla:

name	value	description
none	0	do nothing when the UGen is finished
pauseSelf	1	pause the enclosing synth, but do not free it
freeSelf	2	free the enclosing synth
freeSelfAndPrev	3	free both this synth and the preceding node
freeSelfAndNext	4	free both this synth and the following node
freeSelfAndFreeAllInPrev	5	free this synth; if the preceding node is a group then do g_freeAll on it, else free it
freeSelfAndFreeAllInNext	6	free this synth; if the following node is a group then do g_freeAll on it, else free it
freeSelfToHead	7	free this synth and all preceding nodes in this group
freeSelfToTail	8	free this synth and all following nodes in this group
freeSelfPausePrev	9	free this synth and pause the preceding node
freeSelfPauseNext	10	free this synth and pause the following node
freeSelfAndDeepFreePrev	11	free this synth and if the preceding node is a group then do g_deepFree on it, else free it
freeSelfAndDeepFreeNext	12	free this synth and if the following node is a group then do g_deepFree on it, else free it
freeAllInGroup	13	free this synth and all other nodes in this group (before and after)
freeGroup	14	free the enclosing group and all nodes within it (including this synth)
freeSelfResumeNext	15	free this synth and resume the following node

5.4 Canales

Tanto el sonido analógico como el digital pueden ser reproducidos en uno o más canales, siendo las formas más comunes el mono (un solo canal) y el estéreo (dos canales; uno izquierdo y uno derecho).

Las tarjetas de sonido nos dan la posibilidad de reproducir el sonido en más canales (2, 4, 8… incluso 9, como es el caso del audio 9.1, que se reproduce mediante tres canales frontales, dos laterales, tres traseros y uno en el techo).

En *SuperCollider* tenemos una serie de clases que nos ayudan a poder trabajar de forma multicanal:

- **Out.ar(canal, señal):** Reproduce el sonido a través de un canal concreto que sirve como *offset* para la distribución del sonido.

  Donde el canal es un número entero que representa el canal de salida del sonido; siendo 0 = canal izquierdo, 1 = canal derecho y 2,3,4,5… multicanal.

  La señal que recibe es cualquier oscilador que creemos, pudiendo este tener una envolvente aplicada.

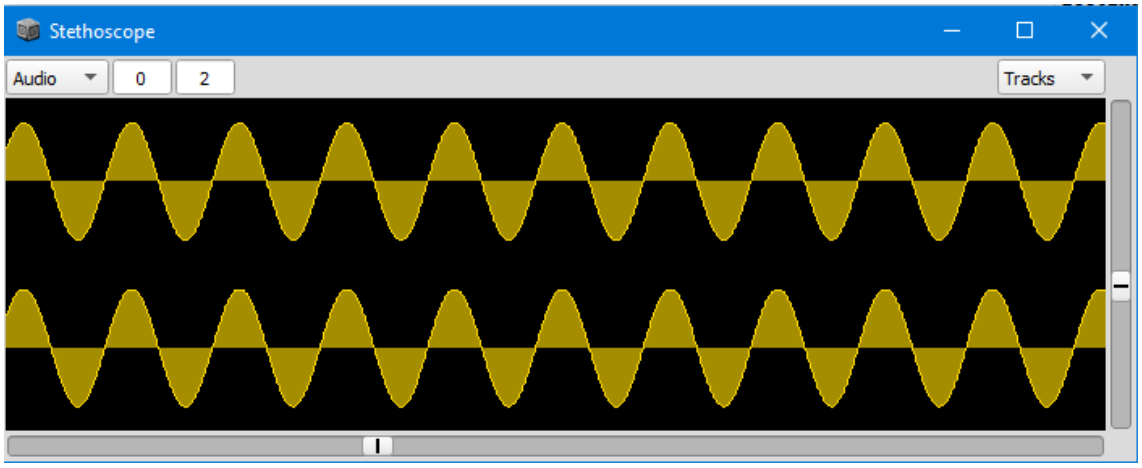
- **Pan2.ar(señal, posición):** En este caso el sonido es distribuido por los canales de salida, haciendo que el sonido no tenga mayor volumen si se reproduce en mono ni menor volumen al reproducirlo en multicanal.

  Al igual que *Out*, recibe una señal que puede ser cualquier oscilador, pero en vez de recibir un canal de salir recibe una posición, siendo -1 = izquierda, 1 = derecha y el resto del intervalo siendo una extrapolación del sonido entre los dos canales.

A continuación, se adjuntan algunos ejemplos interesantes del uso de estas clases:

En la siguiente imagen se muestra el resultado de ejecutar: ***{Out.ar(0,Pan2.ar(SinOsc.ar(440),0))}.scope***

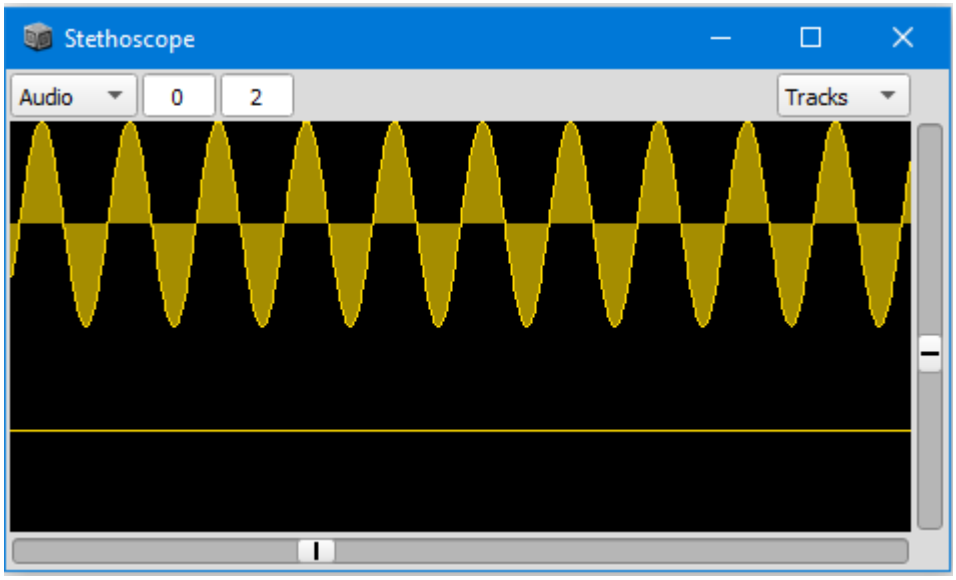
Esto genera un oscilador senoidal a 440Hz y lo reproduce por el canal izquierdo mediante la clase *Out*, a la vez que en vez de usar el oscilador como tal como argumento, lo utiliza dentro de la clase *Pan2* con posición 0, haciendo que se reproduzca por los dos canales con posición central y distribuyendo por ellos su sonido de forma uniforme.



{Out.ar(0,Pan2.ar(SinOsc.ar(440),0))}.scope

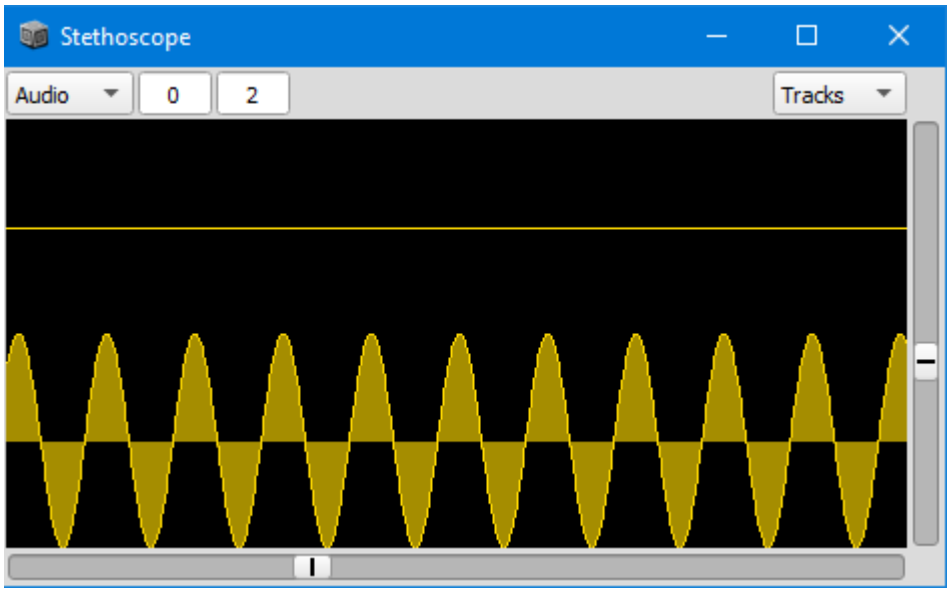
También podemos ejecutar el siguiente trozo de código:

{Pan2.ar(SinOsc.ar(440),-1)}.scope para reproducir el mismo oscilador senoidal solo por el canal izquierdo.



{Pan2.ar(SinOsc.ar(440),-1)}.scope

Y por último podemos ejecutar ***{Pan2.ar(SinOsc.ar(440),1)}.scope*** para reproducirlo por el canal derecho.



{Pan2.ar(SinOsc.ar(440),1)}.scope

5.5 Operaciones con MIDI

A la hora de trabajar con notas musicales en *SuperCollider* podemos hacerlo mediante interacciones con la representación numérica de estas en el protocolo *MIDI* de forma sencilla.

Para ello utilizaremos dos métodos:

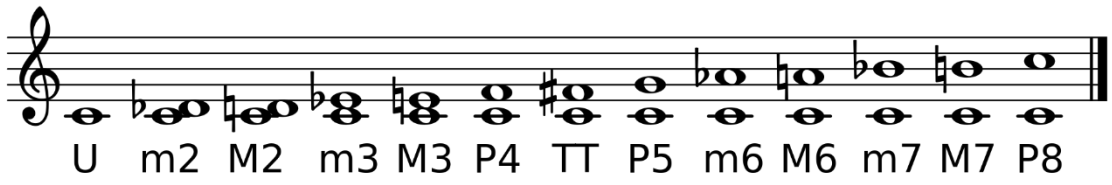
- **.midicps:** Este método convierte cualquier nota en notación *MIDI* a su equivalente en Hercios para su cómoda manipulación en *SuperCollider*.

  Por ejemplo; la nota Re sería representada en notación *MIDI* por el número 62, y para utilizarla en el entorno de SuperCollider la llamaríamos mediante *62.midicps*

- ***.cpsmidi:*** Por otra parte, este método es la contraparte de *.midicps*, y convierte una frecuencia dada en Hercios por su equivalente en notación *MIDI*.

  De esta forma, la nota La, cuya frecuencia es de 440 Hercios, podría ser convertida a notación *MIDI* mediante la llamada *440.cpsmidi*, que daría como resultado 69 en notación *MIDI*.

SuperCollider también nos ofrece la posibilidad de utilizar el método *.midiratio* para trabajar con intervalos expresados en semitonos.



Intervalos harmónicos a partir de la nota do

De esta forma podemos aplicar dicho método sobre un número entero y multiplicado por una nota con los siguientes resultados:

Número en *.midiratio*	**Nombre del intervalo**	**Distancia entre tonos y semitonos**
0	Unísono	Mismo sonido
1	Segunda menor	1 semitono
2	Segunda mayor	1 tono
3	Tercera menor	1 ½ tonos
4	Tercera mayor	2 tonos
5	Cuarta justa	2 ½ tonos
6	Cuarta aumentada (Tritono)	3 tonos
7	Quinta justa	3 ½ tonos
8	Sexta menor	4 tonos
9	Sexta mayor	4 ½ tonos
10	Séptima menor	5 tonos
11	Séptima mayor	5 ½ tonos
12	Octava justa	6 tonos

Los valores numéricos sobre los que usamos *.midiratio* también se pueden aplicar en forma de números negativos, haciendo de esta forma que en vez de aumentar un semitono por cada uno, se disminuya.

Así tendríamos que si la nota La (69 en notación *MIDI*) la multiplicamos por *8.midiratio* obtenemos su sexta menor:

69.midicps * 8.midiratio

Por otro lado, si la multiplicamos por *-5.midiratio* obtenemos su cuarta descendente:

69.midicps * -5.midiratio

5.6 Arrays

Al igual que en la mayoría de los lenguajes de programación modernos, *SuperCollider* nos permite trabajar con *arrays*, que representan un conjunto de elementos ordenados, los cuales se declaran dentro de corchetes y separados por comas.

Los *arrays* pueden ser utilizados para operar entre ellos, aunque con ciertas particularidades. Si operamos con dos *arrays* del mismo tamaño, la operación se aplica entre sus miembros uno a uno:

$$[\mathbf{1},\mathbf{2},\mathbf{3},\mathbf{4}] + [\mathbf{1},\mathbf{2},\mathbf{3},\mathbf{4}] == [\mathbf{1}+\mathbf{1},\mathbf{2}+\mathbf{2},\mathbf{3}+\mathbf{3},\mathbf{4}+\mathbf{4}]$$

Sin embargo, si los *arrays* poseen un distinto número de elementos, la operación se ejecuta miembro por miembro hasta que el de menor tamaño llega a su fin, momento en el que se continúa operando a partir del principio de este:

$$[\mathbf{1},\mathbf{2}] * [\mathbf{1},\mathbf{2},\mathbf{3},\mathbf{4}] == [\mathbf{1}*\mathbf{1},\mathbf{2}*\mathbf{2},\mathbf{1}*\mathbf{3},\mathbf{2}*\mathbf{4}]$$

A continuación, explicaremos algunos de los métodos más útiles que nos ofrece *SuperCollider* para trabajar con estos *arrays*, algunos de los cuales no facilitarán mucho el trabajo dependiendo de lo que necesitemos:

- **.size:** Devuelve como *output* el tamaño del *array*.
- **.scramble:** Altera de forma aleatoria el contenido del *array* sobre el que se ejecuta.
- **.mirror:** Genera un *array* que contiene un espejo del primero, es decir, coge el punto medio del array y a continuación añade los mismos elementos en sentido inverso.

  Por ejemplo *[1,2,3].mirror* nos devuelve el *array* [1,2,3,2,1]

- **.choose:** Escoge un elemento al azar del *array*.
- **.wchoose:** Es similar a *.choose*, solo que recibe como parámetro un *array* de pesos probabilísticos con una entrada por cada elemento del *array*, en el cual se base para asignar la probabilidad de devolver cada uno de estos elementos. Debemos tener en cuenta de que la suma de los pesos no puede ser mayor que 1.

- **.sum:** Devuelve el valor resultante al sumar todos los elementos del *array*.

- **.find:** Recibe como argumento un objeto, el cual busca dentro del *array*. Si dicho objeto está contenido, devuelve su posición dentro de este; por otro lado, si no se encuentra el objeto en cuestión dentro del *array*, devuelve *nil*.

- **Acceder a datos de un array:** Si asignamos un *array* a una variable (por ejemplo, *arr = [1,2,3]*) podemos acceder al dato que se encuentre en su posición *i* mediante la llamada *arr[i]*.

  Para añadir un nuevo valor a un *array*, emplearemos la función *add*. Por ejemplo, *arr.add(3)* añadiría el valor 3 al final de *array*.

- **Métodos comunes:** Cabe destacar que cualquier método aplicable a u número es aplicable a un *array*. Por ejemplo, podemos ejecutar los antes mencionados *.midiratio, .midicps y .cpsmidi* sobre cualquier *array* numérico.

La clase array también posee constructores para crear *arrays* de datos con estructuras concretas. A continuación. veremos los más importantes:

- **Array.fill(i,"x"):** Crea un *array* que contiene el valor *"x"* repetido *i* veces.

- **Array.rand(i,x,z):** Genera un *array* que contiene *i* valores numéricos dentro, los cuales se generan aleatoriamente dentro del rango delimitado entre *x* y *z*.

- **Array.series(i,x,z):** Genera un *array* de *i* elementos. Este *array* comienza en *x* y se incrementa sumando este valor a *z* en cada iteración.

- **Array.geom(i,x,z):** Funciona de forma similiar a *array.series()*, a diferencia de que en este caso, en cada iteración el valor se multiplica por *z* en vez de sumarlo.

```
[60,62,63,65,67,68,70,72].midicps
```

Array con las notas de la escala menor natural

5.7 Variables

El concepto de variable es común a todos los lenguajes de programación. Representan espacios de memoria en los que se almacenan datos bajo un identificador, con el objetivo de poder acceder a ellos y manipularlos de forma cómoda.

Normalmente se usan para guardar información, asignar valores entre variables, representar información para imprimirla por pantalla…

Las variables no tienen un número máximo ni un nombre predefinido; podemos definir todas las que queramos o necesitemos y asignarles los nombres que nos resulten más claros a la hora de interactuar con ellas.

La forma de declararlas en SuperCollider es la siguiente:

1. Escribimos *var* seguido de un espacio y el nombre que deseemos asignar.
2. Las separamos entre sí mediante comas.
3. Una vez hayamos asignado un valor a cada variable la cerramos mediante punto y coma (*;*).
4. Si el nombre de nuestra variable solo consiste en una letra no es necesario declararla. Aunque debemos tener en cuenta que la variable s está reservada para referirse al servidor e interactuar con él.
5. El nombre de una variable siempre debe comenzar por minúsculas.

```
(
var test1, test2;
test1 = 1;
test2 = 2;
test1+test2;
)
```

Ejemplo de declaración de variables. El resultado mostrado por consola sería '3'

También podemos guardar osciladores concretos dentro de variables, y a su vez estas variables dentro de *arrays*.

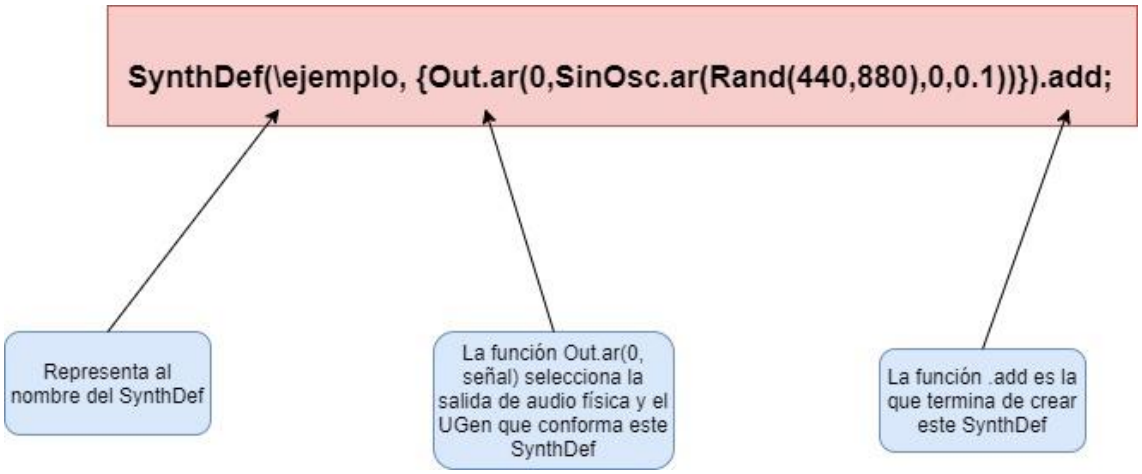
5.8 SynthDef

SuperCollider utiliza la definición de *SynthDef* como plantillas para hacer nuevos *synth*. Los *SynthDef* son la forma que utilizaremos en *SuperCollider* para crear sonidos.

Un *SynthDef* es la representación en el lado del cliente de un *synth*, el cual podremos reproducir, guardar en memoria…

Una vez definidos, estos *synth* se pueden manipular independientemente unos de otros, y son muy útiles para utilizarlos como "recetas" que podemos usar para diseñar y manipular varios *synth* a partir de un solo *SynthDef*.

La sintaxis de estos es la siguiente:



Para hacer sonar este *SynthDef* simplemente tendríamos que ejecutar la siguiente línea de código:

Synth.new(\ejemplo);

Una vez definido el *SynthDef* y visto como reproducirlo podríamos encapsular este *Synth* dentro de una variable, y sobre ella aplicar el método .*set* para alterar los argumentos propios del *SynthDef* del que parte, provocando alteraciones en el sonido en tiempo en real.

Cuando asignamos un *Synth* a una variable y lo reproducimos, recibimos por consola el nombre de este seguido de un número separado por dos puntos. Este número representa la identificación del nodo en el que se encuentra el *Synth* (*NodeID*).

Este identificador se adquiere al generar el *Synth*, que busca un identificador disponible y se lo autoasigna.

El servidor de síntesis maneja un árbol de nodos, que son objetos con los que se puede mantener una comunicación. Este árbol define el orden en el que los *synth* son ejecutados.

Los *SynthDef* son la estructura más compleja de manejar dentro de *SuperCollider*, pero al mismo tiempo son donde realmente reside la capacidad creativa de este lenguaje. Son la base fundamental para crear composiciones y poder reproducirlas, pero también son muy útiles para poder realizar interpretaciones en vivo, también conocidas como sesiones de *livecoding*, en las que se van alterando las propiedades de los *SynthDef* para conseguir patrones rítmicos y tonales variantes, que dan lugar a melodías complejas.

Sesión de live-coding en SuperCollider por Juan Romero y Patrick Borgeat

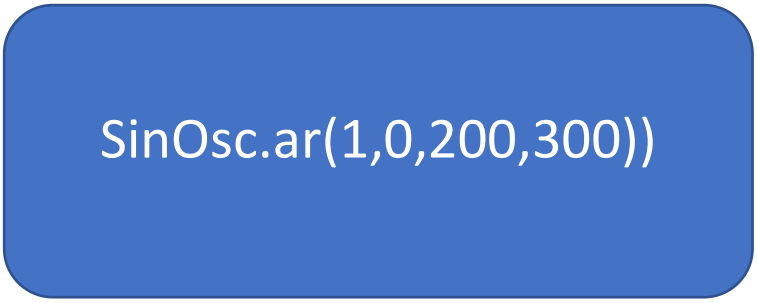
5.9 Uso de UGens para gestionar UGens

Hasta ahora hemos visto como trabajar con *UGens* usando argumentos en su creación creados a mano por nosotros; especificando su frecuencia en Hercios, su multiplicador…

También hemos visto en el capítulo anterior cómo crear *SynthDef* que se pueden manipular en tiempo real a partir de estos *UGens*.

Pero esta forma de creación es muy limitada y no permite trabajar con *UGens* realmente complejos, para lo que *SuperCollider* nos permite crear *UGens* que reciben en vez de parámetros numéricos, otros *UGens*, haciendo que sus propiedades varíen en el tiempo sin tener que recibir un *.set* manualmente.

Veamos algunos ejemplos para ilustrar esto:



En este caso tenemos un oscilador senoidal con frecuencia = 1, lo que hace que su amplitud se mueva en el rango [-1,1], siguiendo el patrón de valores 1 0 -1 0 1.

Su amplitud es de 200, lo que hace que los valores pasen a ser [-1,1] * 200.

Por último, tenemos el *'add'* ó sumador, el cual suma 300 al rango de nuestro oscilador senoidal, haciendo que su salida definitiva sea el rango [-1,1]*200+300. Por tanto, nuestro oscilador se mueve en el rango [100,500]

Este oscilador podría ser usado como argumento frecuencia a la hora de crear otro oscilador senoidal, haciendo que su frecuencia varíe en el tiempo siguiendo el patrón anteriormente descrito, entre 100Hz y 500Hz.

Esto es muy útil ya que el oscilador que hemos definido parte de una frecuencia de 1Hz, lo que la hace muy sencilla de manejar, pero no audible; es por esto que si lo utilizamos como argumento frecuencia en otro *SinOsc* podemos hacer variar su frecuencia en el tiempo, siguiendo los valores que arroja la ejecución del primer oscilador.

A este proceso se le denomina *FM* (Frecuencia Modulada), ya que estamos alterando la frecuencia de una señal de audio de forma periódica.

Cabe resaltar que ya que el oscilador que utilizamos para modular el que reproducimos es simplemente eso, un modulador, deberíamos usarlo en su forma .kr para ahorrar potencia computacional cuyo gasto es innecesario.

```
{SinOsc.ar(SinOsc.kr(1,0,200,300))}.play;
```

Oscilador senoidal con frecuencia modulada por otro oscilador senoidal

También podemos utilizar osciladores para trabajar con la amplitud de una señal, lo que se denomina *AM* (Amplitud Modulada).

Para trabajar con síntesis *AM* tenemos que recordar que el oscilador que utilizamos como argumento amplitud debe mantener por debajo de 1 la suma de su amplitud y su sumador (*add*).

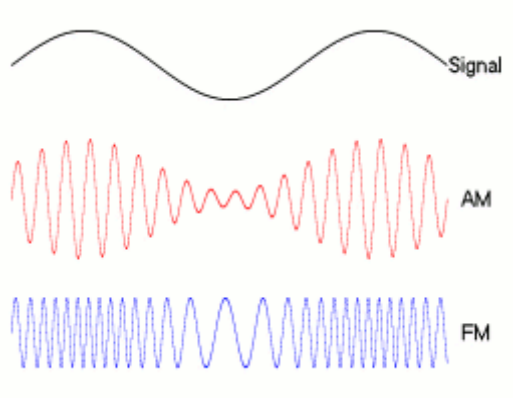
De esta forma tendríamos que si a un oscilador senoidal trabajando a 440Hz le pasamos como argumento amplitud otro oscilador senoidal que recibe como parámetros frecuencia = 1, fase = 0, amplitud = 0.5 y suma = 0.5, obtendremos como salida el rango [-1,1]*0’5+0.5, o lo que es lo mismo [0,1].

Así, el oscilador principal verá su frecuencia multiplicada por 0 y 1 periódicamente, haciendo que el sonido cambie de una frecuencia de 0Hz a su frecuencia original en un bucle periódico.

```
{SinOsc.ar(440,0,SinOsc.kr(1,0,0.5,0.5))}.play;
```

Oscilador senoidal con amplitud modulada por otro oscilador senoidal

Si combinamos la síntesis *FM* y *AM* con el concepto de *SynthDef* encontraremos una forma aún más interesante de generar sonidos en *SuperCollider*.



AM frente a FM

5.9 Ruido de baja frecuencia

Los *LFNoise* o *Low Frecuency Noise* son un tipo especial de *UGens* con los que se trabaja en SuperCollider.

Estos son generadores de ruido cuya amplitud oscila entre -1 y 1 de forma aleatoria, con una frecuencia máxima.

Dentro de estos, nos encontramos con tres tipos de generadores de ruido de baja frecuencia:

- **LFNoise0**: Este generador cambia de amplitud de forma brusca, mediante saltos imprevisibles, generando cambios discretos.

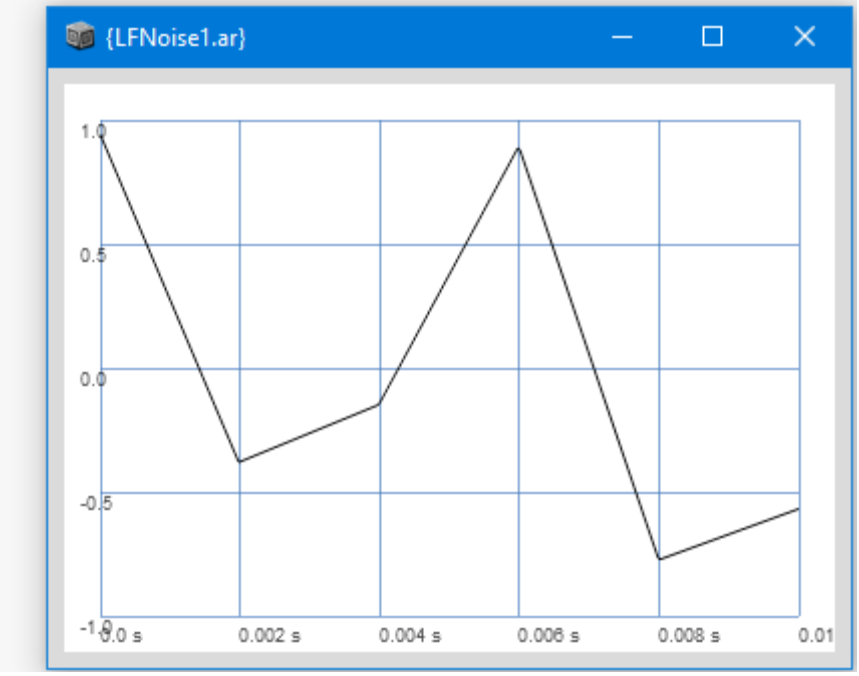
- **LFNoise1**: Por su parte, este generador funciona por interpolación lineal, recorriendo una línea que atraviesa los valores que separan dos puntos. Estos cambios son lineales.

- **LFNoise2**: Por último, este generador de ruido de baja frecuencia funciona por interpolación de curvas cuadráticas.

```
1 s.boot;
2 {LFNoise0.ar}.plot;
```

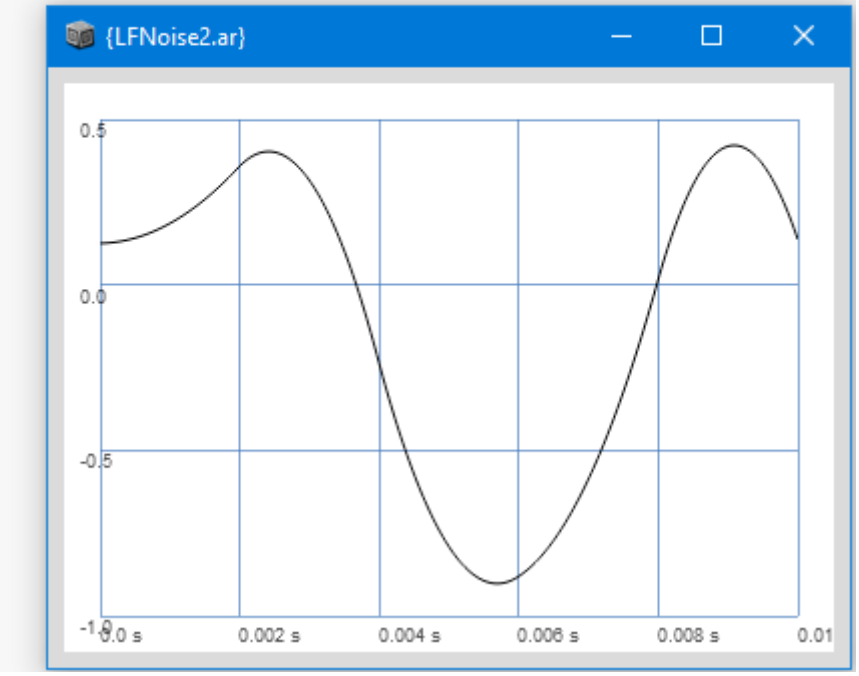
Representación gráfica de LFNoise0

```
1 s.boot;
2 {LFNoise1.ar}.plot;
```



Representación gráfica de LFNoise1

```
1 s.boot;
2 {LFNoise2.ar}.plot;
```



Representación gráfica de LFNoise2

Estos generadores de ruido son muy útiles para modular frecuencias de señales sonoras. Para usarlos dentro de un rango concreto de Hercios podemos aplicar las siguientes fórmulas:

$$amplitud = \frac{(rango\ máximo - rango\ mínimo)}{2}$$

$$sumador = amplitud + rango\ mínimo$$

De esta forma tenemos que si queremos trabajar en el rango [440Hz,2500Hz] deberemos usar:

$$amplitud = \frac{2500 - 440}{2} = 1030$$

$$sumador = 1030 + 2500 = 3530$$

Por tanto, para cambiar la frecuencia de un oscilador senoidal entre 440Hz y 2500Hz mediante ruidos de baja frecuencia podríamos hacer:

```
s.boot;
{SinOsc.ar(LFNoise0.kr(5,1030,3530),0,0.5)}.play;
{SinOsc.ar(LFNoise1.kr(5,1030,3530),0,0.5)}.play;
{SinOsc.ar(LFNoise2.kr(5,1030,3530),0,0.5)}.play;
```

De esta forma tendríamos tres osciladores senoidales en los que la frecuencia se mueve entre 440Hz y 2500Hz a razón de 5 veces por segundo.

Si reproducimos los tres, notaremos como en el primero los cambios de tono son muy bruscos, llegando a sonar como pitidos inconexos, en el segundo notaremos la frecuencia moverse en líneas rectas, mientras que en el segundo se nota una mayor suavidad entre los cambios de frecuencia.

5.10 Tdef

Las *Tdef* ó *Task Reference Definition* son objetos en *SuperCollider* que tienen como función definir tareas que se ejecutarán en un orden y tiempo dados por nosotros.

De esta forma se pueden crear de forma cómoda secuencias de acciones ordenadas para agilizar procesos.

Estas tareas tienen una sintaxis parecida a la de los *SynthDef* y pueden guardar dentro de ellas cualquier línea de código que quisiéramos ejecutar; osciladores, generadores de ruido, *synth*, impresiones por pantalla, operaciones matemáticas…

Ilustremos mejor este concepto con un ejemplo en código:

```
s.boot;
(
Tdef(\test, {
"Esto es una demo".postln;
5.wait;
"Ahora vamos a usar un oscilador".postln;
2.wait;
    a = {SinOsc.ar(LFNoise0.kr(5,1030,3530),0,0.5)};
    a.play;
});
)

Tdef(\test).play

```

En este fragmento de código declaramos una *tdef* llamada *"\test"*. Dentro de esta tarea mostramos el texto *"Esto es una demo"* por consola, tras lo cual mediante el método *wait* le decimos a la tarea que espere 5 segundos. Tras ese tiempo, se imprime por consola el texto *"Ahora vamos a usar un oscilador"*, y dos segundos después se declara un oscilador senoidal, el cual se reproduce y la tarea finaliza.

Para llamar a esta tarea, se ejecuta el comando ***Tdef(\test).play***.

Como acabamos de comprobar, las tareas otorgan un nuevo nivel de complejidad a nuestras composiciones en *SuperCollider*, ya que podemos programar *synth*, los cuales pasado una cantidad determinada de segundos varían en sus parámetros, haciendo que podamos establecer un sistema de *playback* que se ejecute con un comando simple, ahorrando trabajo respecto a tener que manipular un *synth* en tiempo real.

Existen algunas funciones que nos pueden ser muy útiles a la hora de trabajar con *Tdef*; por ahora definiremos las más importantes:

- **do:** La función do nos permite replicar un número determinado de veces una acción concreta, ahorrándonos el escribirla repetidas veces o ejecutarla manualmente más de una vez.

  Podemos pensar en esta función como si se tratase de un bucle *for* que se repite durante n ejecuciones, y que funciona siguiendo la sintaxis:

  n.do{instrucción}

  Donde *n* representa el número de repeticiones a ejecutar e instrucción es cualquier función que diseñemos en *SuperCollider*.

  Merece la pena destacar que, si en vez de un número *n* lo llamamos sobre *inf*, el bucle se ejecutará perpetuamente hasta que decidamos pararlo manualmente.

  A continuación, lo explicaremos mejor con un pequeño ejemplo:

```
1  s.boot;
2
3  Tdef(\prueba, {
4      10.do{
5          "Postear este texto cada 3 segundos durante 10 segundos".postln;
6      3.wait;
7  }
8  });
9
10 Tdef(\prueba).play
11 Tdef(\prueba).stop
```

  En este ejemplo vemos como definimos una *Tdef* "prueba", dentro de la cual se ejecuta 10 veces un ciclo en el que se muestra por consola la cadena de texto *"Postear este texto cada 3 segundos durante 10 segundos"*.

  Tras mostrar esa línea por consola, la tarea espera durante 3 segundos, tras lo cual repite el ciclo. Una vez ejecutado este ciclo 10 veces, la tarea finaliza.

- **play:** Al ejecutar la función *play* sobre una llamada a una tarea, esta se ejecuta en su totalidad.

- **stop:** Esta función detiene el proceso de ejecución de la tarea llamada en cualquier momento y punto en el que esta se encuentre al ser llamada.

```
s.boot;

Tdef(\prueba, {
    inf.do{
        "Funciono como un metrónomo a 60bpm (Tic, tac)".postln;
        {SinOsc.ar(440)*Line.kr(1,0,0.1,doneAction:2)}.play;
    1.wait;
}
});

Tdef(\prueba).play
Tdef(\prueba).stop
```

Ejemplo de Tdef con un bucle infinito y un oscilador senoidal funcionando en línea

5.11 Pbind

Otra forma de trabajar con repeticiones dentro de *SuperCollider* son los llamados objetos *Pbind*.

En este caso funcionan para crear patrones rítmicos dentro de nuestro código, recibiendo como parámetros:

- **Frecuencia (\freq)**
- **Duración (\dur)**
- **Amplitud (\amp)**
- **Posición de paneo (\pan)**
- **Sostenimiento (\sustain)**
- **Nota (\note)**
- **Nota MIDI (\midinote)**
- **Instrumento (\instrument)**

La estructura a la hora de definir un *Pbind* es la siguiente:

```
Pbind(
 \dur,1,
 \midinote, 69,
 \amp, 0.6).play;
```

En la imagen anterior se crea un objeto *Pbind* que reproduce la nota 69 en notación *MIDI* con un volumen de 0.6 cada segundo de forma infinita.

Vamos a ver ahora algunas funciones que nos resultarán muy útiles a la hora de crear *Pbinds* con un sonido más complejo:

- **Pseq:** Representa una secuencia de valores concretos que se recorre un número determinado de veces.

  Obedece a la siguiente sintaxis:

  Pseq([array de valores],número de repeticiones)

- **Prand:** Funciona de forma muy similar a *Pseq*, con la diferencia de que en este caso los valores del array que recibe como parámetro se recorren de forma aleatoria, no de forma ordenada como en su contraparte.

```
s.boot;

(
SynthDef.new(\sine, {
 arg freq=440, atk=0.005, rel=0.3, amp=1, pan=0;
 var sig, env;
 env = EnvGen.kr(Env.new([0,1,0], [atk,rel], [1, -1]),
doneAction:2);
 sig = Pan2.ar(SinOsc.ar(freq), pan, amp) * env;
 Out.ar(0, sig);
 }).add;
)


(
p = Pbind(
 \instrument, \sine,
 \midinote, Pseq([69,70,71,72,73,74,75],inf),
 \dur, Prand([0.2,0.4,0.6,0.8,1],inf)
).play;
)
```

En la imagen anterior tenemos un ejemplo de *Pbind* en el que primero creamos como instrumento un *SynthDef* con un oscilador senoidal multiplicado por su envolvente.

A continuación, declaramos el objeto *Pbind* que utiliza ese *SynthDef* como instrumento, un *Pseq* con un array notas *MIDI*, recorrido linealmente para reproducirlas, y por último utiliza como tempo un *Prand* de tiempos en fracciones de segundo que se recorre de forma aleatoria.

Como hemos visto, los *Pbind* son objetos sencillos de manipular y que pueden ser muy útiles a la hora de componer, ya que con una construcción de una complejidad relativamente baja nos permite crear fácilmente ritmos y melodías que se ejecuten de forma infinita en el tiempo.

5.12 Bloques condicionales

Como en prácticamente todos los lenguajes de programación modernos; SuperCollider nos permite utilizar bloques condicionales para poder ejecutar funciones dependiendo de la veracidad de un predicado lógico.

Esto podemos hacerlo mediante dos tiempos de estructuras que se detallarán a continuación.

5.12.1 Bloques if

Los bloques *if* son los bloques condicionales más sencillos y comunes en la programación informática.

Este tipo de bloques reciben como argumento una expresión lógica que debe de poderse evaluar de forma que devuelva como salida un *booleano True* o *False*.

Tras esto, el bloque recibe una o más funciones, que se ejecutarán dependiendo de si el predicado lógico que hemos pasado como atributo resulta ser cierto o falso.

En *SuperCollider*, si tras evaluar dicho predicado no hemos seleccionado una acción a ejecutar para la salida lógica obtenida, recibiremos como salida un valor *nil*, el cual corresponde a lo que en la mayoría de los lenguajes de programación se denomina ‘*null’* o valor nulo, y que representa la ausencia de salida.

La sintaxis para dichos bloques en *Supercollider* es la siguiente:

If(Expresión lógica,
{Función a ejecutar si es cierta},
{Función a ejecutar si es falsa};
)

A continuación, tenemos un ejemplo práctico de como utilizar estos bloques condicionales:

```
s.boot;

c =[1,2,3,4,5,6,7,8,9,10];

var num;
num = c.choose;

if(num%2==0,
    {(num.asString + "es un número par").postln},
    {(num.asString + "es un número impar").postln};
)
```

En este fragmento de código inicializamos un array con los números enteros comprendidos desde 1 hasta 10.

Tras esto creamos la variable '*num'*, dentro de la cual metemos un número de dicho array escogido al azar.

En un bloque *if*, dividimos el número escogido dentro de la variable *'num'* y lo dividimos entre 2; si su resto es 0 se muestra por consola el número seguido de la cadena de texto "es par", mientras que, si el resto es distinto de 0, se hace lo mismo, pero mostrando por consola que se trata de un número impar.

Como mostraremos a continuación, dentro de estos bloques *if* podemos anidar otros bloques *if*, haciendo que, si el predicado lógico es cierto, se puede realizar otra evaluación de otro predicado, y lo mismo si el predicado original es falso.

A continuación, mostramos un ejemplo de esto, además de mostrar como crear funciones que dependan de un argumento que nosotros tengamos que pasarle:

```
s.boot;


a ={ |n| if(n%2==0,
    {(n.asString + "es un número par").postln},
    {
        if(n%3==0,
            {(n.asString + "es un multiplo de 3").postln},
            {(n.asString + "es un número impar y no es múltiplo de 3 ").postln};
    )};
)}

```

En este caso el bloque es similar al anterior, pero con dos diferencias principales:

1. Dentro del bloque encontramos que en caso del número que pasemos no ser impar se ejecuta un segundo bloque *if*, en el que se comprueba si el número es divisible entre 3 para mostrar por pantalla el número seguido de la cadena "es múltiplo de 3", mientras que, si no se cumple, se muestra por pantalla el número seguido de la cadena de texto "es un número impar y no es múltiplo de 3"

2. Por otra parte, tenemos que encapsulamos este bloque *if* dentro de una función llamada 'a', la cual recibe un parámetro '*n*' en el cual base la comprobación del predicado lógico.

Para ejecutar esta función no tendríamos más que hacer una llamada de la siguiente forma:

```
a.value(3)
a.value(2)
a.value(5)
```

Llamadas las cuales tras su ejecución imprimirán por consola los siguientes resultados respectivamente:

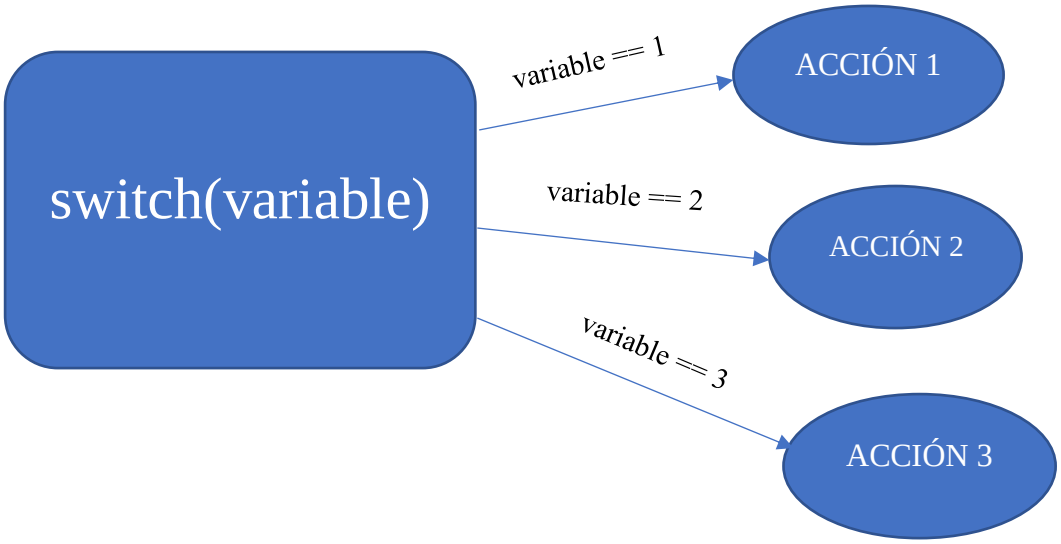
```
-> a Function
3 es un multiplo de 3
-> 3 es un multiplo de 3
2 es un número par
-> 2 es un número par
5 es un número impar y no es múltiplo de 3
-> 5 es un número impar y no es múltiplo de 3
```

5.12.2 Bloques switch

Los bloques *switch* también son comunes a muchos de los lenguajes de programación actuales.

Estos bloques se caracterizan por recibir una variable como argumento, y en función de su valor, ejecuta una orden u otra.

Veamos a continuación un esquema básico de este tipo de bloques:



Estos bloques son muy útiles a la hora de realizar múltiples acciones que dependan de una variable, ya que si tuviéramos que hacer esto con bloques *if* la extensión y

legibilidad de nuestro código se reduciría enormemente al tener que anidar muchos bloques unos dentro de otros.

Vamos a ver como funcionan estos bloques a nivel práctico:

```
s.boot;

a = {|n| switch(n, "sine",{{SinOsc.ar()}.scope},
    "saw",{{Saw.ar()}.scope},
    "pulse",{{Pulse.ar()}.scope},
    "triangle",{{LFTri.ar()}.scope};
);
}

```

En este bloque declaramos una función '*a*' que recibe como parámetro una cadena de texto '*n*'.

Esta cadena de texto entra en un bloque *switch*, en el que en función de su contenido se activa un oscilador, el cual se muestra mediante la interfaz de estetoscopio de *SuperCollider*.

Así tenemos, que si llamáramos la función ejecutando *a.value("pulse")* recibiríamos en respuesta la vista de estetoscopio y el sonido de un oscilador de pulsos.

5.13 Buffers

Los objetos buffer en *SuperCollider* tienen un interés especial, representan un espacio en el servidor en el que podemos cargar archivos de audio; ya sean canciones, efectos de sonido, *samples*…

En estos objetos podemos cargar archivos en formato *WAV* de la siguiente forma:

```
s.boot;

~b0 = Buffer.read(s,"C:/Users/joseb/Music/Royal Blood/08-
Careless.wav");
~b0.play;
```

En el ejemplo anterior cargamos una canción en formato *WAV* a un buffer en el servidor '*s*' y en la siguiente línea lo reproducimos.

Merece la pena mencionar que si la ruta al archivo dentro del *buffer* la dejamos en blanco y arrastramos el archivo que deseamos a ese espacio, *SuperCollider* se encargará de rellenar la ruta automáticamente, lo cual es muy cómodo.

Antes de estudiar los argumentos que pueden recibir esta clase de objetos vamos a mencionar tres métodos que nos serán de gran utilidad al trabajar con *buffers*:

- **.zero:** Vacía el contenido del *buffer*, haciendo que a pesar de no liberarlo, este pase a ser vacío.

- **.free:** Libera un buffer, eliminándolo de los *synth*.

- **Buffer.freeAll:** Hace lo mismo que *.free* pero para todos los *buffers* declarados.

Los argumentos que utilizaremos a la hora de leer un *buffer* son los siguientes, aunque normalmente solo utilizaremos los dos primeros:

- **Server:** Define el servidor en el que se ubicará el *buffer*. Normalmente usaremos '*s*' para representar este servidor.

- **Path:** La ruta absoluta del archivo que queremos cargar en el *buffer*.

- **startFrame:** Indica el primer *frame* a reproducir del archivo que vamos a leer. Por defecto su valor es 0, que representa el inicio del archivo de audio.

  Es importante recordar que un *frame* es una muestra de audio, y que, por ejemplo, en un CD de audio, se encuentran 44100 muestras por segundo.

- **numFrames:** Cuántos *frames* se van a reproducir del archivo de sonido.

  Su valor por defecto es -1, que hace que se reproduzca el archivo completo.

  Por poner un ejemplo, si en este campo usamos el valor 88200 se reproducen dos segundos de audio.

- **Action:** Nos permite indicar una función que se ejecutará una vez la reproducción finalice.

Merece la pena mencionar también *PlayBuf*, que es un *UGen* que nos permite reproducir archivos de audio en un objeto *buffer* y modificar algunas de sus características, así como poder utilizarlos en la definición de *synths*.

Este *UGen* recibe los siguientes parámetros:

- **numChannels:** Es un número entero que representa el número de canales que tendrá el *buffer*.

  Este valor no se puede cambiar una vez se haya compilado el *SynthDef*.

- **bufnum:** Es el índice del *buffer* que se utiliza.

- **rate:** Es el tono al que se reproducirá el audio. 1.0 es la tonalidad estándar, 2.0 aumenta una octava, 0.5 disminuye una octava, -1.0 reproduce el audio a tono normal, pero hacia atrás…

- **startPos:** Elige la muestra a partir de la que comienza la reproducción del audio.

- **loop:** Es modulable, indica si se reproducirá el sonido en bucle. 1 lo marca a true y 0 lo marca a false.

- **doneAction:** Acción a ejecutar una vez finalice la reproducción del *buffer*.

Con estas opciones, obtenemos un *UGen* que podemos usar de forma sencilla en la definición de *synth* y que ofrece muchas posibilidades a la hora de hacer *loops* a partir de archivos de audio, ya sean grabados por nosotros o descargados de otras fuentes.

Podemos utilizarlo para agilizar el proceso de creación de *beats* y su edición en directo, dándonos una forma sencilla y cómoda de introducirnos en el mundo de la música en directo programada en *SuperCollider*, al mismo tiempo que nos facilita el grabar y guardar estos bucles de audio para poder utilizarlos en algún programa de edición y producción musical.

Veamos mejor esto con un ejemplo simple:

```
s.boot;

~b0 = Buffer.read(s,"C:/Users/joseb/Music/Royal Blood/08-Careless.wav",
0,44100*5+22050);


(
SynthDef.new(\playbuf_test, {
 arg amp=1, out=0, buf, rate=1, da=2;
 var sig;
 sig = PlayBuf.ar(1, buf, rate,loop:1, doneAction:da);
 sig = sig * amp;
 Out.ar(out, sig);
 }).add;
)


Synth.new(\playbuf_test, [\buf, ~b0.bufnum, \rate, 2]);

```

En la primera línea cogemos una pista de audio en formato *WAV*, en este caso será la canción *Careless* de la banda británica *Royal Blood*, recortada a sus primeros 5 segundos y medio.

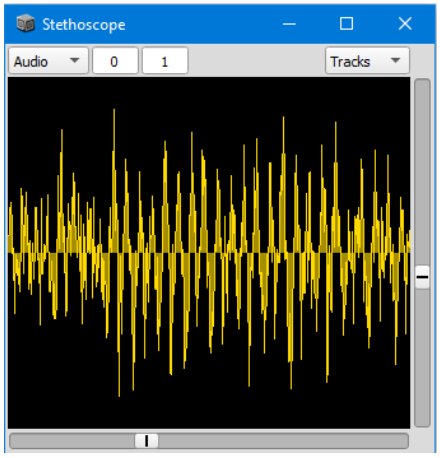
A continuación, definimos un *Synth* al que llamamos *\playbuf_test*, el cual se compone de un *PlayBuf* con un solo canal, un buffer cuyo número escogeremos a la hora de crear el *Synth*, un *rate* que escogeremos también al crear el *Synth*, el argumento *loop* a 1, haciendo que se reproduzca en bucle, y la *doneAction* seleccionada es la 2, que como vimos anteriormente significa liberar el *synth* en cuestión.

Por último, multiplicamos esta señal por una amplitud 1 y la reproducimos mediante un Out.ar por el canal 0.

Una vez definido este *SynthDef*, creamos un *Synth* a partir de este, seleccionamos el buffer que encapsula la pista de audio y aumentamos el *rate* 2, haciendo que se aumente en una octava.

Esto *synth*, al ser reproducido nos da un bucle de batería y bajo, más agudo que en la canción original, el cual podríamos usar como base para una composición.

El *UGen PlayBuf* nos permite también visualizarlo a través de estetoscopio de *SuperCollider* mediante la función *.scope*:



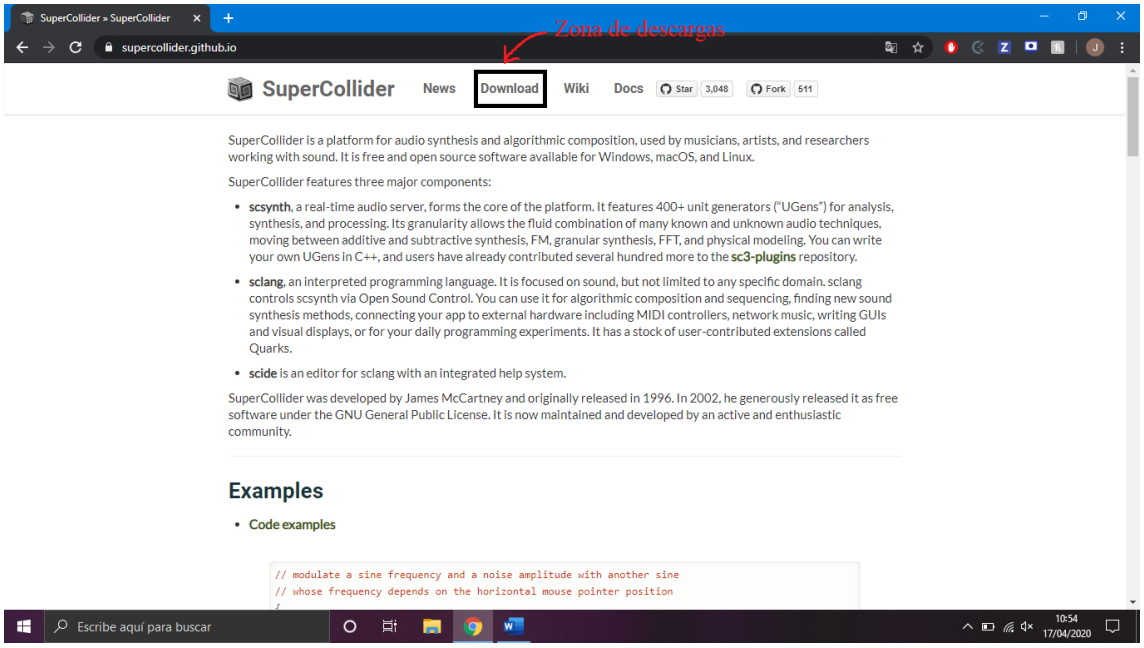
6. SuperCollider GUI

A estas alturas ya conocemos bastante sobre el funcionamiento del lenguaje de programación *SuperCollider*.

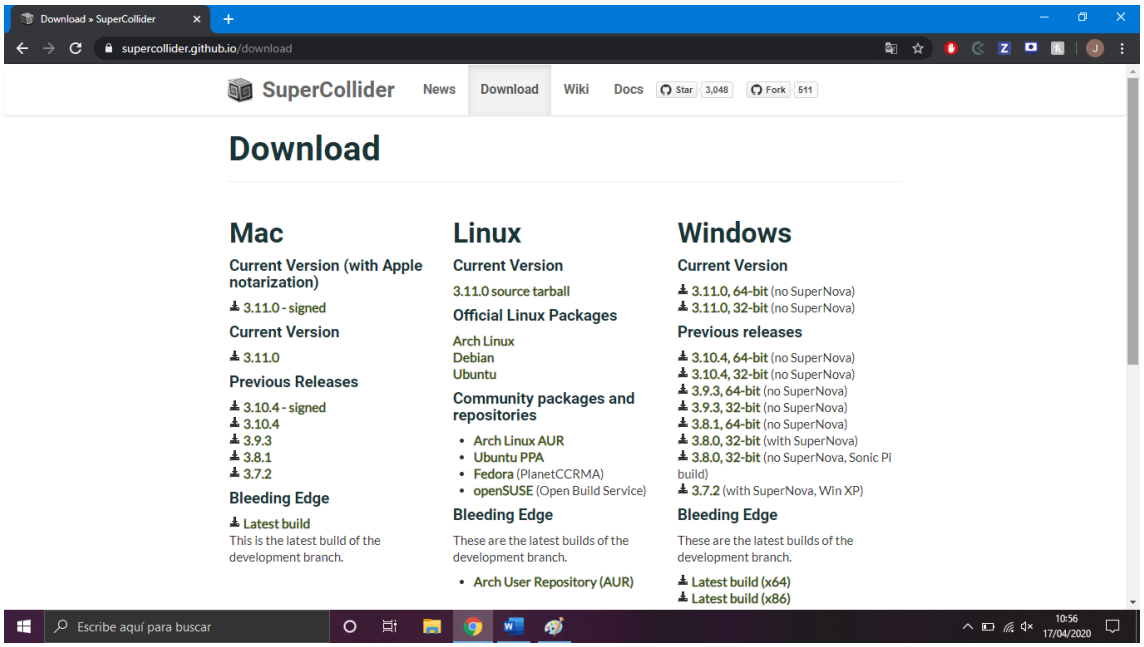
Por tanto, ahora deberíamos hablar de como llevar este conocimiento a la práctica y la experimentación; y haremos ambas cosas mediante el entorno de desarrollo e interfaz gráfica nativa de *SuperCollider*.

6.1 Instalación

Para instalar *SuperCollider* tendremos que ir a su repositorio oficial dentro de *GitHub* (*https://supercollider.github.io/*), donde nos encontraremos con la siguiente página de bienvenida con información varia, y un menú superior con distintas secciones, donde nos dirigiremos a la zona de descargas.



Una vez estemos en dicha zona, nos encontraremos con un listado de versiones para distintos sistemas operativos, donde deberemos escoger la opción que más se adapte a nuestro equipo.

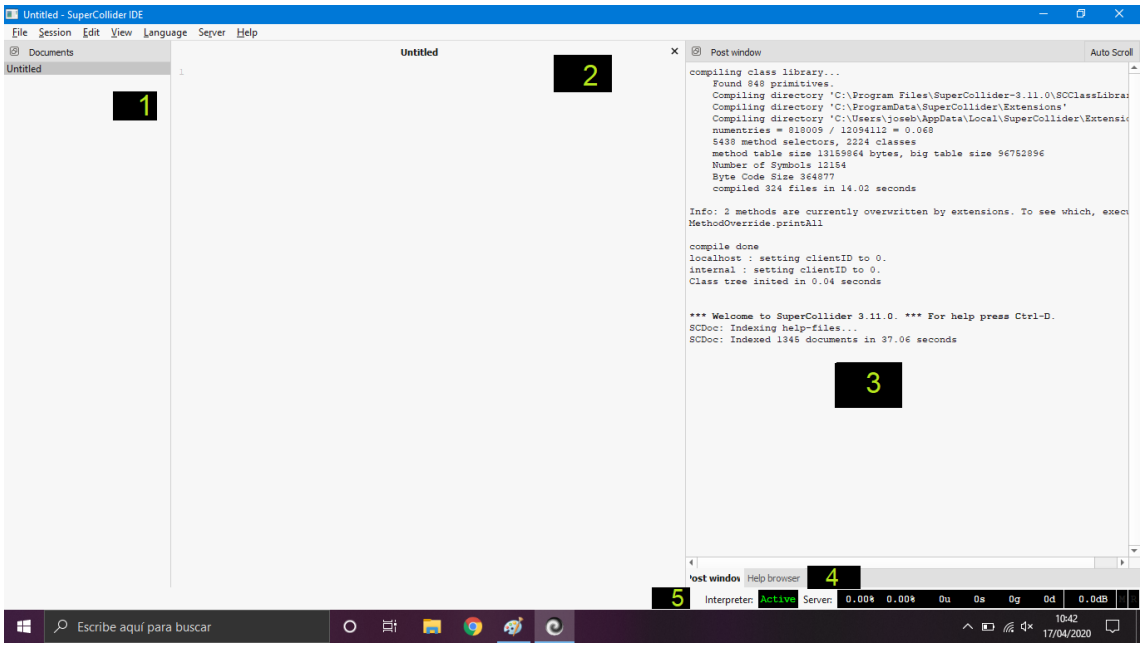


En el caso de la elaboración de este trabajo, esta ha sido la versión 3.11.0 para *Windows* de 64 bits, sin *Supernova*.

Una vez completada la descarga, procederemos con el instalador y ya tendremos *SuperCollider* listo para trabajar.

6.2 Interfaz

Cuando abrimos por primera vez *SuperCollider* nos encontramos con una vista general de la interfaz tal y como se ve en la siguiente imagen:



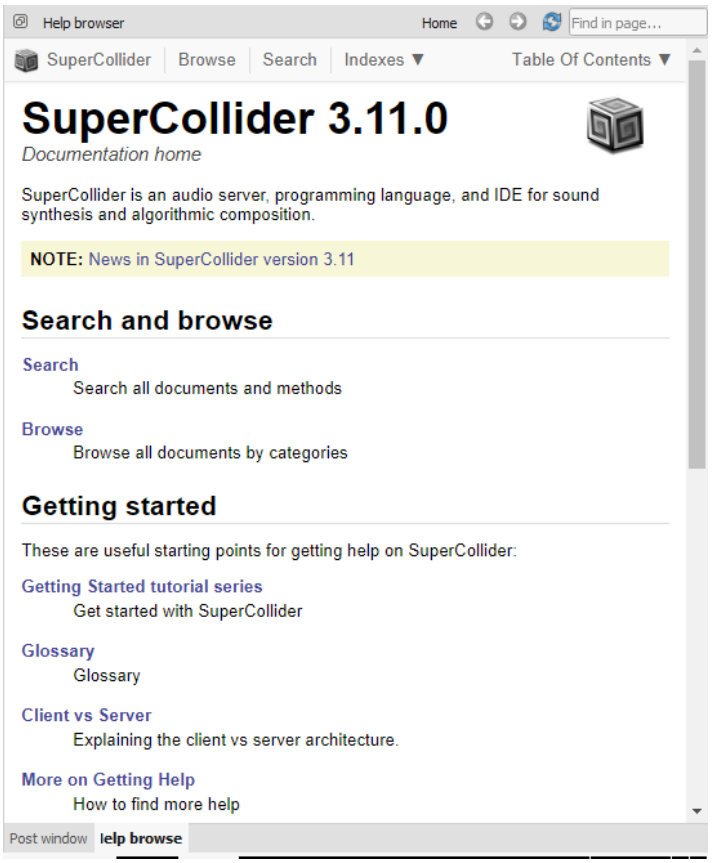
Hemos numerado del 1 al 5 las zonas principales de este entorno de desarrollo:

1. **Documents:** En esta sección encontraremos una lista de los archivos *.scd* que tenemos abiertos en ese momento.

2. **Vista de archivo:** En este panel se encuentra el archivo que estemos editando. Será aquí donde programemos y ejecutemos nuestro código.

3. **Post Window:** Actúa como la consola del lenguaje. Aquí veremos el resultado de la ejecución de las líneas de código, información sobre el sistema y servidores a la hora de arrancar un servidor o eliminarlo…

4. En esta zona podemos escoger entre visualizar la *Post Window* o el *Help Browser*, que es un panel conectado al repositorio en el que podremos consultar tutoriales, glosario y documentación sobre *SuperCollider*.



5. **Status bar:** Muestra información en tiempo real sobre el servidor de sonido y el intérprete del lenguaje. Además de notificaciones relacionadas con el *IDE*.

6.3 Atajos de teclado

A continuación, tenemos una tabla que recoge los atajos de teclado principales que existen en *SuperCollider*, en su versión para *SuperCollider* en *OSX* y *Windows*, *gedit* y *Emacs*.

Functions	OSX SC App	Win PsyCollider	sced gedit	scel Emacs
Language-Specific Commands:				
Interpret Selection	enter (fn+return)	ctl-enter	ctl-e	C-c C-x / C-c C-c
Interpret current line	enter (fn+return)	ctl-enter	ctl-e	C-c C-c
stop	cmd-.	alt-.	escape	C-c C-s
Run Main-run	cmd-r	alt-r		C-c C-r
recompile library	cmd-k	alt-k		C-c C-l
clear post window	cmd-sh-C	alt-p		C-c <
Open Help File	cmd-d	F1		C-c C-h
Open Help Browser	cmd-sh-D			C-M-h
Open Class Definition	cmd-j	alt-j		C-c :
Implementations of	cmd-y	alt-y		C-c ;
References to	cmd-sh-Y	alt-sh-Y		C-c ;
Files:				
Open text document	cmd-o	ctl-o	ctl-o	C-x C-f
New text document	cmd-n	ctl-n	ctl-n	(open non-existent file w. new name)
Close text document	cmd-w	ctl-w	ctl-w	C-x k
Save text document	cmd-s	ctl-s	ctl-s	C-x C-s
Save text document as	cmd-sh-S	ctl-sh-S	ctl-sh-S	C-x C-w
HTML doc window -> code win			Ctl-t	M-x sclang-minor-mode / M-x sclang-mode
Text Editing:				
Undo	cmd-z	ctl-z	ctl-z	C-x u
Redo	cmd-sh-Z	ctl-y	ctl-sh-Z	
Copy	cmd-c	ctl-c	ctl-c	M-w
Paste	cmd-v	ctl-v	ctl-v	C-y
Cut	cmd-x	ctl-x	ctl-x	C-w
select all	cmd-a	ctl-a	ctl-a	C-x a
select block		ctl-b		
goto line ...	cmd-,		ctl-l	M-g g
Find ...	cmd-f		ctl-f	C-s
Find next	cmd-g		ctl-g	C-s
Find previous	cmd-sh-G		ctl-sh-G	C-r
replace and find next	cmd-l			M-%
replace	cmd-=			M-,
copy text style only	cmd-alt-c			
paste text style only	cmd-alt-v			
Formatting:				
Syntax Colorize	cmd-'	(auto)		(auto)
Balance	cmd-sh-B			
Comment (add // in front)	cmd-/			ctl-/
Uncomment (remove //s)	cmd-sh-/			ctl-sh-/
Indent / Shift left (move selected text by one tab)	cmd-[			ctl-t
Unindent / Shift right (by one tab)	cmd-]			ctl-sh-T
Insert (): enclose selected text with (txt)	cmd-(			
Insert []: enclose selected text with [txt]	cmd-alt-[			
Insert {}: enclose selected text with {txt}	cmd-{			
Insert /* */: enclose selected text with /*txt*/	cmd-*			

Fuente: https://www.bartetzki.de/docs/sc_common/shortcuts.pdf

7.Experimentos

En esta sección realizaremos dos experimentos en los que se pondrán en práctica las nociones sobre *SuperCollider*.

7.1 Metrónomo

Un metrónomo es un aparato, ya sea mecánico o digital, que sirve para indicar los tiempos de las composiciones musicales, y son muy útiles de cara a la interpretación ya que facilitan mantener un pulso constante.

Estos aparatos emiten señales periódicas, ya sean visuales o sonoras, que representan tempos, por ejemplo, en *bpm* (*beats per minute*).

Metrónomo de cuerda

Metrónomo digital

Un metrónomo funcionando a un número determinado de *bpm* significa que este produce una señal sonora un número determinado de veces por cada minuto.

Podemos obtener el cambio de *bpm* a cada cuantos segundos se produce esta señal mediante la siguiente fórmula:

$$Cada\ cuantos\ segundo\ se\ emite\ la\ señal = \frac{60\ segundos}{x\ bpm\ (beats\ per\ minute)}$$

Para realizar esto en *SuperCollider* se ha definido la siguiente función:

metronomo.scd

```
s.boot;


a = {arg bpm; Tdef(\Metronomo,{
inf.do{
{LFTri.ar(100)*Line.kr(1,0,0.1,doneAction:2)}.play;
        "tic".postln;
        (60/bpm).wait;

    }

});
}


a.value(180).play;

```

metronomo.scd

Esta función recibe un argumento *'bpm'* que debemos pasarle a la hora de llamarla, el cual se usa en una *Tdef*, que de forma infinita ejecuta un oscilador triangular *LFTri* a 100Hz, cuya señal de salida se multiplica por un oscilador lineal en modo *.kr*, el cual sirve control *rate*.

Este sonido se ejecuta durante dos décimas de segundo a la vez que imprime la cadena de texto *'tic'* por consola tras cada ejecución.

Dicha acción se ejecuta de forma infinita siguiendo un pulso constante según el número de *bpms* que le pasemos como argumento en la llamada.

Así tenemos, que por ejemplo si quisiéramos mantener un tempo andantino (80 *bpm*), deberíamos llamarlo de la siguiente forma:

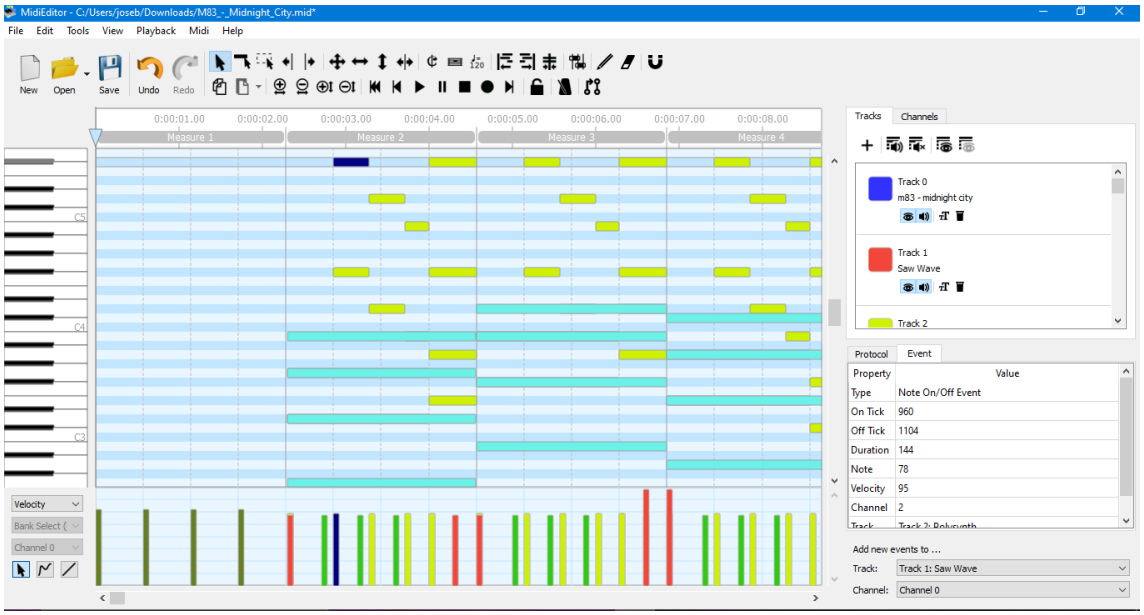
```
17 a.value(80).play;
```

7.2 Midnight City

En el siguiente experimento hemos tratado de replicar el riff de sintetizador principal de la canción *Midnight City*, perteneciente al álbum *"Hurry Up, We're Dreaming"* (*2011*, *Naïve/Hostess*) de la banda francesa *M83*.

Para ello se ha descargado de internet la versión *MIDI* de la canción, la cual se ha abierto con el programa *MIDIEditor*.

Gracias a este programa, se puede visualizar dicho archivo *MIDI* de la siguiente forma:



Gracias a esta interfaz podemos ubicar rápidamente cada nota *MIDI* y su equivalente en piano, haciendo que obtener los acordes que forman el *riff* principal sea bastante sencillo.

Una vez extraídos estos acordes, se ha procedido a crear un *SynthDef* en *SuperCollider* con dichas notas *MIDI*, para las cuales se ha tenido que obtener un tempo aproximado mediante un método de prueba y error.

Ya con los tiempos de duración de las notas y los silencios entre ellos, se ha definido una envolvente para intentar asemejar el sonido lo máximo posible al original, mediante un ataque de 0'05 y un *release* de un segundo para cada nota, que crea ese efecto de desvanecimiento propio de un sintetizador.

El oscilador escogido en este caso es un *LFTri* u oscilador triangular.

A continuación, se adjunta captura de pantalla del código:

midnight-city-riff.scd

```
s.boot;

(
SynthDef.new(\midnightcitykeyboard,{
    arg freq=440,atk=0.05,rel=1,amp=1.2,pan=0;
    var sig,env;
    env= EnvGen.kr(Env.new([0,1,0],[atk,rel],[-1,1]),doneAction:2);
    sig= Pan2.ar(LFTri.ar(freq),pan,amp)*env;
    Out.ar(0,sig);
}).add;
)

(
p= Pbind(
    \instrument,\midnightcitykeyboard,
    \midinote,Pseq([
[78,66],[74,62],71,[78,66,57,52],0,
[78,66],74,71,[78,66,57],0,
[78,66],[74,62],[71,59],[78,66,54,49],0,
[78,66],[74,62],71,[78,66,54,49],[78,66,54,49],[78,66,54,49],[78,66,54,49],[55,50],0

    ],inf),
    \dur,Pseq([
        0.56,0.56,0.4,0.82,0.8,
        0.56,0.56,0.4,0.82,0.8,
        0.56,0.56,0.4,0.82,0.8,
        0.56,0.56,0.4,0.17,0.17,0.17,0.17,0.17,0.5,

    ],inf)
).play;
)
```

midnight-city-riff.scd

8.Conclusiones

Si bien *SuperCollider* es un lenguaje de programación musical muy potente, en su estructura podemos observar que, a la hora de llevarlo a la práctica con el fin de componer piezas musicales, tiene una curva de aprendizaje muy elevada.

Se trata de un lenguaje que puede ser muy útil para personas especializadas en ingeniería acústica que quieran realizar experimentos en este ámbito, ya que además de los *UGens* ya existentes en este, cuenta con una comunidad que crea y comparte aquellos que ellos mismos programan, acentuando aún más el espíritu de software libre de esta plataforma.

Esta empinada curva de aprendizaje no impide que muchos artistas creen e interpreten en directo sus composiciones, llegando a realizar obras complejas que en muchos casos varían en tempo y sonido en función de ecuaciones matemáticas, lo que convierte a estos artistas más en científicos que en músicos como tal.

Durante la elaboración de esta guía hemos tocado los aspectos principales y más básicos de este lenguaje, una sección muy pequeña de lo que supone la documentación completa, dando de esta forma una idea general que, si bien es limitada, ayuda a

comprender la magnitud de esta plataforma y de las posibilidades que conlleva tanto en el aspecto puramente musical como en el aspecto científico de investigación acústica.

Y ya no solo acústica, puesto que en su estructura como lenguaje orientado a objetos en el que se pueden crear incluso interfaces gráficas nos encontramos con que podría llegar a tener uso como lenguaje de programación matemática inclusive, o como acompañante de producción y generación de *samples* que luego podrían ser utilizados en un *DAW* (*Digital Audio Workstation*) a la hora de grabar, producir y masterizar piezas de audio con un uso comercial.

Personalmente, tengo la sensación de que es un lenguaje relativamente poco conocido fuera de la comunidad musical y científica *underground*, y que podría verse muy mejorado mediante *UGens* comunitarios si se conociera más sobre él, tanto como se conocen plataformas similares como *PureData*, *Nyquist*, *CommonMusic* o *Alda*.

Por mi parte, una vez descubierto y aprendido sobre sus aspecto más básicos, seguiré navegando por su documentación, la cual está muy bien redactada y es de gran utilidad, con el fin de poder crear composiciones más complejas y usarlo como herramienta en mi vida diaria para ayudarme a componer e interpretar piezas musicales.

9.Referencias comentadas

En este capítulo se incluyen todas las referencias utilizadas a la hora de desarrollar este proyecto y un enlace para facilitar su consulta.

Algunas de estas referencias han sido útiles para poder explicar algún término concreto, mientras que otras han sido necesarias para desarrollar casi todo el proyecto, estando de esta forma ordenadas de las más importantes a las más puntuales.

Las referencias 1,2 y 3 han sido esenciales para desarrollar el proyecto, puesto que tratan *SuperCollider* y su composición en profundidad. Desde sus métodos, a sus variables y estructuras; todo lo explicado en los capítulos 5 y 6 está inspirado por estos recursos.

Las referencias 4,5,6 y 7 han servido de apoyo para el desarrollo teórico de los conceptos expuestos en los primeros capítulos de la memoria.

El resto de las referencias han servido para matizar algunos conceptos teóricos desarrollados a la hora de hablar de ciertos aspectos de *SuperCollider*; a excepción de la número 12, que es meramente ilustrativa del concepto de *live-coding*, y la número 14, que es la fuente de la tabla de atajos de teclado mostrada anteriormente.

Por último, la referencia número 15 es el repositorio de GitHub utilizado para la realización del proyecto, en el cual se encuentra esta misma memoria en formato *.doc*, *.pdf*, y los archivos necesarios para ejecutar en *SuperCollider* los dos experimentos realizados.

1. Web oficial de *SuperCollider* con toda la documentación: https://www.supercollider.github.io
2. Curso de *SuperCollider* para principiantes (*Ernesto Romero y Ezequiel Metri*), compuesto por capítulos que siguen el orden de aprendizaje más natural posible: http://cmm.cenart.gob.mx/tallerdeaudio/cursos/cursocollider/textos/curso%20de%20supercollider%20principiantes.pdf
3. Blog sobre *SuperCollider* que incluye algunas aclaraciones y demostraciones muy interesantes sobre algunos aspectos de *SuperCollider*: http://supercolliderspanish.blogspot.com/
4. Propiedades del sonido, artículo de *Andrew Rossi*: https://www.vix.com/es/btg/curiosidades/2010/09/22/propiedades-del-sonido
5. Concepto de sonido: https://concepto.de/sonido/
6. Conceptos básicos del sonido, parte del temario de la asignatura *Ingeniería de las Ondas*, del grado universitario en *Ingeniería de Telecomunicaciones de la Universidad de Valladolid* : https://www.lpi.tel.uva.es/~nacho/docencia/ing_ond_1/trabajos_05_06/io2/public_html/sonido.html
7. Audio digital: https://es.wikipedia.org/wiki/Audio_digital
8. Información sobre el protocolo *MIDI* aportada por la *Universidad Politécnica de Valencia* : http://www.disca.upv.es/adomenec/IASPA/tema5/Midi.html
9. Envolvente ADSR: https://www.wikiaudio.org/adsr-envelope/
10. Intervalos: https://es.wikipedia.org/wiki/Intervalo_(m%C3%BAsica)
11. Definición de variables en programación: https://lenguajesdeprogramacion.net/diccionario/que-es-una-variable-en-programacion/
12. Sesión de *livecoding*, descubierta durante la investigación sobre este género musical y utilizada como imagen ilustrativa en este trabajo (página 35): https://www.youtube.com/watch?v=Ix2b_qFYfAA
13. AM/FM: https://es.wikipedia.org/wiki/Frecuencia_modulada
14. Tabla de atajos de teclado, obtenida en la web de *André Bartetzki* y adjuntada en la página 56 : https://www.bartetzki.de/docs/sc_common/shortcuts.pdf
15. Repositorio de este proyecto: https://github.com/joscarboz/SuperColliderTFG