Música por computadora

Ernesto Romero y Hernani Villaseñor
Centro Multimedia 2012

Sesión 1

1- Introducción a SuperCollider:
    Historia, relevancia, figuras destacadas, ejemplos de obras de distintos estilos y disciplinas.
2- Elementos estructurales de la aplicación:
    IDE, servidor y cliente.
3- Programación orientada a objetos:
    3.1- Sintáxis, objetos, métodos, argumentos(), rates, audio rate .ar, control rate .kr, *strings* y símbolos.

1 - Introdución a SuperCollider

SuperCollider (SC) es un ambiente y lenguaje de programación para síntesis en tiempo real y composición algorítmica. Está provisto de un lenguaje orientado a objetos que es interpretado y funciona como un cliente en red, con un servidor de síntesis en tiempo real de alto nivel de desempeño.

Historia

SuperCollider fue creado por James McCartney en 1996, la primer versión corría en Power Macintosh y costaba 250 dólares. Posteriormente fue liberada, para convertirse en una aplicación de uso libre y de código abierto; además, ahora es multiplataforma, es decir, que corre en Mac OSX, Windows y Linux.
Originalmente SC estaba hecho de dos programas separados: Synth-O-Matic, -escrito en 1990 por McCartney- y por un objeto de MAX llamado Pyrite, el cual contenía el lenguaje intérprete. A partir de la versión 2, SC se compone de los programas scsynth como servidor y sclang como lenguaje.

Relevancia

SC es una de las principales herramientas usadas en el ámbito de la composición de música electrónica académica, el arte sonoro y la multimedia. Es utilizado para la enseñanza en las aulas de muchas universidades y centros académicos alrededor del mundo. También tiene un papel importante en la investigación científica en ramas como la acústica y la psicoacústica.

Figuras destacadas

La comunidad que se ha formado en torno a esta herramienta es basta, algunas figuras destacadas son:

Nick Collins, Frederik Olofsson, Cylob, Andrea Valle, Sergio Luque, Roberto Morales, Juan Sebastian Lach, Dan Stowell.

Ejemplos de obras de distintos estilos y disciplinas

- sc140, códigos de 140 caractéres, 22 códigos en colaboración con la revista Wire, los cuales hacen referencia a el número de caractéres permitidos en un tweet.

- LiveCoding HackPact de Fredrick Olofsson.

- “Virtual electronic poem” de Andrea Valle, una recreación en SuperCollider de -“Poema electrónico” de Edgar Varèse.

2 - Elementos estructurales de la aplicación

IDE

Ambiente de desarrollo integrado o IDE (Integrated Development Environment), es una aplicación que nos permite ingresar comandos y pedir que estos se ejecuten, también nos muestra los resultados, errores y avisos del programa. En MacOS y en Windows, SC contiene su propia IDE. En Linux es necesario abrir una IDE independiente de SC y pedirle que entre en el modo de SuperCollider. Los IDE mas usados en Linux son: Gedit, Emacs y Vim.
Cada IDE tiene sus atajos o *shortcuts* para realizar las acciones esenciales de SuperCollider. A continuación presentamos una lista de estas acciones y sus atajos correspondientes en cada IDE.

Antes que nada prendemos y apagamos SC con el siguiente código:

```
// prende supercollider
s.boot;

// apaga supercollider
s.quit;
```

	MacOS	Windows	Gedit	Emacs
Evaluar selección	Enter (no Return)	Ctrl+Intro	Ctrl+E	Ctrl+C + Ctrl+C
Detener procesos	cmd+. (cmd+punto)	Alt+. (Alt+punto)	Esc	Ctrl+C + Ctrl+S
Abrir archivo de ayuda	cmd + D	F1	Ctrl+U	Ctrl+H

Normalmente SC muestra dos ventanas que representan los servidores, una Post Window que es una ventana donde se imprime le resultado del proceso de lo que realiza el programa, así como los errores y por último una ventana donde escribimos el código a manera de texto.

Servidor

Para producir sonido en SC es necesario prender un servidor. En SC existen dos servidores: servidor interno y servidor externo.

El servidor interno corre en el mismo proceso que la aplicación SC, llamada “sclang”, es interno al programa y por eso mismo tiene ciertas ventajas, producto de la mayor comunicación entre los dos.

El servidor local corre en la misma máquina que la aplicación SC, pero es un programa diferente: “scsynth”. La ventaja de usarlo es que en caso de que el servidor se caiga, la aplicación seguirá corriendo y viceversa.

Cliente

SC funciona bajo el modelo de cliente/servidor, los cuales trabajan dentro de una red, es decir el usuario escribe programas mediante los cuales el cliente solicita al servidor que haga algo.

3 Programación orientada a objetos

Sintaxis

La sintaxis es la manera en que debemos acomodar los elementos del lenguaje de SC para que este entienda nuestros códigos. La sintaxis es dada por el programa y tenemos que aprenderla para poder escribir cosas con sentido y comunicarnos con SC. Es difícil entender la sintaxis sin comenzar con ejemplos que involucren a la partes del lenguaje a ordenar. Aquí ponemos una lista de elementos de la sintaxis a modo de glosario. Durante el desarrollo de las sesiones se irá describiendo su uso ya dentro de un contexto específico.

//	las dos barras diagonales definen un comentario.
/*	una diagonal seguida de un asterisco abre una sección de comentario.
*/	el asterisco seguido de diagonal cierra una sección de comentario previamente abierta.
{ }	las llaves definen una función.
[ ]	los corchetes definen un arreglo.
( )	los parentesis definen un argumento o un bloque de código.
| |	los pipes definen argumentos.
Algo	Una palabra que inicia con mayúscula representa un objeto.
.otraCosa	Un punto seguido de una palabra que inicia con minúscula representa un mensaje o método.
;	el punto y coma indica una ruptura o *break* en el código.
,	la coma separa argumentos.
“lo que sea”	cualquier cosa escrita entre comillas es un *String.*
\algoMas	cualquier palabra escrita después de una diagonal es un símbolo.
~	la tilde indica una variable global.

Comentarios

Los caracteres `//` o `/* */` sirven para decirle a SC que vamos a escribir algo que no se interpretará como código. Esto se llama comentar. SC hará caso omiso de lo que comentemos. Hay dos maneras de comentar.

```
// Este es un comentario breve

/* Este es un comentario largo y tiene que cerrarse */
```

Objetos

Los elementos básicos de SC son llamados objetos. Casi todo en SC es un objeto. Los objetos son capaces de realizar ciertas tareas, y es la conjugación de varios objetos lo que nos permite realizar tareas complejas. Cuando escribimos un objetos siempre empieza con mayúscula.

Existe una clase especial de objeto: los `UGens` (*Unit Generators* o Unidades Generadoras). Estos son los objetos que en SC generan sonido y cuyas tareas son manejadas por el servidor. Ejemplo:

```
Pulse // UGen o unidad generadora de ondas de sonido cuadradas.
```

Métodos

Los métodos son también llamados mensajes y es a través de ellos que podemos decirle a los objetos lo qué queremos que hagan y cómo queremos que lo hagan. Al mismo objeto se le pueden enviar distintos métodos, pero un objeto no aceptará cualquier método, sino solo un

grupo de métodos que están asociados a el. Cuando escribimos un método siempre va después de un punto y empieza con minúscula.

Ejemplo: Los números enteros en SC son objetos, son instancias del objeto Integer. Enviemos al objeto `7` los mensajes `odd` y `plot`.

```
7.odd // odd pregunta si es el objeto es un número impar

7.plot /* plot hace una gráfica en 2D del objeto. Un número
          entero no puede arrojar una gráfica 2D por lo que al
          declarar esta línea obtenemos un error en la ventana
          Post. */
```

`.odd`, pero, por ejemplo si ponemos `.plot`, este nos arrojará un error, ya que es un mensaje que los números no aceptan.

También hay métodos que pueden escribirse en diferentes objetos.

Ejemplo:

```
// el mensaje .postln imprime información en la post window
"palabras".postln
```

Argumentos

Los argumentos son parte de un método y nos ayudan a decirle al objeto, específicamente, de que forma queremos que realice una tarea. Los argumentos se escriben entre paréntesis y van separados por comas en caso de ser más de uno. Ejemplo:

```
// nos redondea el número entero
10.0975.round (1)

// nos redondea a un decimal
10.0975.round (0.1)

/*
Los argumentos se escriben entre paréntesis y se separan por
comas si son mas de uno.
El mensaje linlin nos convierte un número dentro de un rango en
su equivalente dentro de otro rango. 98 es, dentro del rango de
0 a 100, igual a 0.98 dentro del rango de 0 a 1.0.
*/
98.linlin(0, 100, 0,1.0)
```

Rates

En SuperCollider la información de los `UGens` es tratada de dos formas: como audio o como control. Para esto están definidos dos mensajes que indican al objeto de que forma será tratado: `.ar` y `.kr`

Audio rate `.ar`

Los UGens que reciben el mensaje `.ar` corren a velocidad de audio: 44100 muestras por segundo. Hay que mandar el mensaje `.ar` a estos `UGens` cuando quieran ser tratados como señal (para que se escuchen).

Control rate `.kr`

Los `UGens` que reciben el mensaje `.kr` corren a velocidad de control: 689 muestras por segundo. Es por esto que los `UGens` de control son más baratos computacionalmente hablando que sus contrapartes a velocidad de *audio rate*. Notar que un `UGen` con el mensaje `.kr` no se escuha.

Los `UGens` de control los usamos como moduladores, esto es, como señales que le dan forma a una señal de audio. En otras palabras los `UGens` con `.kr` controlan los argumentos de los `UGens` con `.ar`

Strings

Los strings son cualquier cadena de caracteres (i.e. palabras, números), que se escriben entre comillas y no tienen ningún significado para SC. Ejemplo:

```
"SuperCollider no sabe que estamos hablando de él".postln
```

Símbolos

Los símbolos son nombres garantizados como únicos. No hay dos cosas nombradas con el mismo símbolo. Se escriben con una diagonal invertida antes de una cadena de caracteres.

Ejemplo:

```
\unSimbolo
```

Los símbolos son usados normalmente para nombrar estructuras de código predefinidas, como puede ser `SynthDefs` o `Tdefs`.

Notación

Las palabras con la tipografía Arial 11 se entienden como el texto del tutorial. Las palabras en `Courier 12` se entienden como código.

Referencias

McCartney, J. (1996). SuperCollider: a new real time synthesis langauge. ICMC.

Netri, E. y Romero, E. (2008). Curso de SuperCollider Principiantes. Centro Multimedia: México.

Polishook, M. (2004). Introductory Tutorial: For SuperCollider 3. Archivo de ayuda de SuperCollider.

Recursos

http://cmm.cenart.gob.mx/tallerdeaudio
http://supercollider.sourceforge.net

Música por computadora

Ernesto Romero y Hernani Villaseñor
Centro Multimedia 2012

Sesión 2

3.2 operadores y variables

Operadores

Los operadores son mensajes que realizan una operación matemática sobre un objeto. Los hay de dos tipos: simples (requieren solo un objeto para operar) y binarios (necesitan dos objetos para operar).

Ejemplos de operadores simples:

```
25.sqrt // la raíz cuadrada de 25
-6.abs  // el valor absoluto de -6
2.reciprocal // el recíproco de 2
```

Ejemplos de operadores binarios:

```
5+5  // la suma de 5 mas 5
10-3 // sustrae 3 a 10
3*8 // multiplca 3 por 8
24/8 // divide 24 entre 8. Representa también un número racional
2**3 // 2 elevado a la tercera potencia
```

Para una lista de operadores ir al archivo de ayuda de `Operators`

Variables

Las variables otorgan flexibilidad y comprensión de lectura a la programación, podemos definirlas como una especie de lugar reservado en la memoria de la computadora que representa un valor de datos y que son definidas por el usuario mientras programa. En SC las

variables se declaran al igualar una palabra, que empieza con mínuscula, a un número o una función. SC tiene dos tipos de variables: variables y variables globales.

La variables tienen que ser declaradas primero dentro de un fragmento de código, utilizando la abreviación `var`. Las variables solo afectan a un fragmento de código.

Ejemplo de variables

```
(
{var sinte, envolvente;

sinte = SinOsc.ar(400,0,0.1);
envolvente = EnvGen.kr(Env.perc(0.1,1),doneAction: 2);

sinte * envolvente}.play
)
```

Las variables globales a diferencia de las variables son cualquier letra del alfabeto en minúscula o cualquier palabra que empiece con una tilde ~. Las variables globales pueden ser declaradas fuera de un código cerrado para después ser usadas.

Ejemplos de variables globales

```
a = 10;
~algo = 10;
```

Referencias

Cottle, D. M. (2005). Computer music with examples in SuperCollider 3.

Dodge, Ch. y A. Jerse, T. (1997). Computer Music: Synthesis, composition and performance. Schirmer.

Netri, E. y Romero, E. (2008). Curso de SuperCollider Principiantes. Centro Multimedia: México.

Polishook, M. (2004). Introductory Tutorial: For SuperCollider 3. Archivo de ayuda de SuperCollider.

Música por computadora

Ernesto Romero y Hernani Villaseñor
Centro Multimedia 2012

Sesión 3

3.3 Funciones, Arreglos

Funciones

Las funciones en SC se encasillan entre llaves { }. Una función en SC representa una acción, por ejemplo hacer sonar un sonido o ejecutar una rutina. Las funciones por sí mismas no trabajan, necesitan de un mensaje para saber que hacer.

La siguiente función es una onda sinoidal, que mediante el mensaje .play sonará cuando evaluemos la línea de código. (recuerden encender el servidor con s.boot)

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

Las funciones también trabajan del mismo modo que en la forma tradicional matemática f(x). Por ejemplo si queremos hacer la función $f(x)=x^2$ podemos escribirla así:

```
f={|x| x**2}
```

Los caracteres | | indican que x es el argumento que recibirá el valor al que queramos aplicar la función.

Para dar un valor a x lo hacemos con el método o mensaje .value() . La siguiente línea da el valor 2 a x pidiendo así 2^2

```
f.value(2)
```

Al evaluar la función obtenemos, por supuesto, el número 4 en la post window.

Podemos usar mas de un argumento en la función. Por ejemplo para una suma de cuadrados:

```
f={|a, b| (a**2)+(b**2)}
f.value(3,4)
```

El resultado debe ser 25

O para la hipotenusa de un triángulo rectángulo. Recordemos el teorema de Pitágoras

$a^2 + b^2 = c^2$

```
f={|a, b| ((a**2)+(b**2)).sqrt}
f.value(3,4)    //  el resultado es 5
```

Se puede usar una función dentro de otra operación:

```
10*f.value(3,4)   //  el resultado es 50
```

Una función solo arroja el resultado de lo último que tenga escrito dentro de ella. En el siguiente ejemplo solo obtendremos el resultado de la suma a + b y no el de la multiplicación a * b

```
f={|a, b| a*b; a+b}
f.value(2,3)    // el resultado es 5
```

Una misma función puede ejecutarse cualquier cantidad de veces con el mensaje .do

```
5.do{"hola mundo".postln}
```

El resultado en la post es:

```
hola mundo
hola mundo
hola mundo
hola mundo
hola mundo
5
```

Para el siguiente ejemplo utilizaremos el mensaje .rrand
Si enviamos el mensaje .rrand a un número *a* y agregamos un número *b* como argumento podemos obtener un valor aleatorio dentro del rango entre *a* y *b*.

a.rrand(*b*)

Así si *a*=300 y *b*=500 obtendremos un número aleatorio entre 300 y 500

```
300.rrand(500)
```

Con este método .rrand podemos, por ejemplo, producir varias ondas senoidales.

```
10.do{{SinOsc.ar(300.rrand(500), 0, 0.1)}.play}
```

Notar que hay una función dentro de otra y por eso usamos llaves anidadas {{}}. También estamos escribiendo 0.1 en el argumento amp del SinOsc para bajar el volumen de cada onda, de esta forma al sumarse el volumen de cada una de las 10 ondas tenemos un volumen total de 1

SuperCollider tiene definida la función de iteración de la suma automáticamente. Una iteración es una operación que se realiza sobre el resultado de la misma operación las veces que se desee. Si partimos del 0, y la operación es sumar 1 al resultado, obtenemos una secuencia de números enteros:

0 + 1 = 1
1 + 1 = 2
2 + 1 = 3
3 + 1 = 4
.
.
.

Para ver esto simplemente hay que declarar la siguiente línea:

```
10.do{ | i |  i.postln }
```

En la post se imprime:

1
2
3
4
5
6
7
8
9
10

El 10 no fué un valor de i. El 10 se imprime por que la función se realizó 10 veces.

Lo que está sucediendo aqui es que el argumento i esta siendo iterado con la operación + 1

Con esta iteración, y el mensaje .do , podemos hacer funciones como la siguiente, donde al número 40 se le va sumando la iteración del argumento i dando como resultado los números del 40 al 49 (del 40 al 49 hay 10 números)

```
10.do{ | i | (40 + i).postln}
```

3

En la post se imprime:

41
42
43
44
45
46
47
48
49
10

El 10 no fué un valor de (40 + i). El 10 se imprime por que la función se realizó 10 veces.

Más aún, si usamos la suma 40 + i como argumento para una onda sinusoidal SinOsc podemos escuchar lo siguiente:

```
10.do{ | i | {Pulse.ar( 40 + i )}.play}
```

Arreglos

Los arreglos o *arrays* son parte del lenguaje de programación en general. Los arreglos son conjuntos ordenados. En SC los arreglos se encasillan entre corchetes [ ]. Sirven para contener objetos o valores de los cuales podemos hacer uso de diversas formas.

Ejemplo de un arreglo en SC con números
```
[1, 2, 3, 4, 5]
```

Ejemplo de un arreglo con números y *strings*
```
[1, 'nada', 3, 'algo']
```

Existen gran cantidad de mensajes que se pueden enviar a un arreglo. A continuación enlistamos algunos:

[0, 1, 2, 3, 4].reverse // pone el arreglo en reversa: [4, 3, 2, 1, 0]
[0, 1, 2, 3, 4].scramble // desordena aleatoriamente el arreglo: [2, 1, 4, 0, 3]
[0, 1, 2, 3, 4].plot // crea una gráfica en 2D
[0, 1, 2, 3, 4].scramble.plot // Se pueden aplicar mensajes consecutivos
[0, 1, 2, 3, 4].mirror // crea un espejo del arreglo: [0, 1, 2, 3, 4, 3, 2, 1, 0]
[0, 1, 2, 3, 4].sum // suma los elementos del arreglo : 0+1+2+3+4=10
[0, 1, 2, 3, 4].pyramid // crea una estructura piramidal con los elementos del arreglo: [0, 0, 1, 0, 1, 2, 0, 1, 2, 3, 0, 1, 2, 3, 4]

La forma piramidal se percibe más facilmente si la reescribimos así:

```
 [ 0,
   0, 1,
   0, 1, 2,
   0, 1, 2, 3,
   0, 1, 2, 3, 4 ]
```

También podemos aplicar operaciones a arreglos:

```
[0, 1, 2, 3, 4] + 10  //  Suma 10 a cada elemento:  [10, 11, 12, 13, 14]
[0, 1, 2, 3, 4] * 10   //  Multiplica por 10 cada elemento:  [0, 10, 20, 30, 40]
```

Se pueden hacer también operaciones entre arreglos. La correspondencia en estos casos es de uno a uno. Si se tienen dos arreglos *a* y *b* y se aplica una suma entre ellos el primer elemento de *a* se sumará al primer elemento de *b*, el segundo con el segundo y así sucesivamente.

```
[0, 1, 2, 3, 4] + [5, 6, 7, 8, 9]    //  el resultado es [5, 7, 9, 11, 13]
```

Si los arreglos tienen diferente cantidad de elementos la operación se aplicará uno a uno también, pero cuando el arreglo más corto se agote comenzará desde el principio de nuevo hasta que los elementos del arreglo más largo se terminen. En el siguiente ejemplo el arreglo *a* tiene 7 elementos mientras que el arreglo *b* tiene solo 5

```
a=[0, 1, 2, 3, 4, 5, 6]
b=[5, 6, 7, 8, 9]
a + b //  el resultado es [5, 7, 9, 11, 13, 10, 12]
```

Las sumas se realizan así:

```
[0+5, 1+6, 2+7, 3+8, 4+9, 5+5, 6+6]
```

Un atajo para crear un arreglo de números consecutivos es así:

```
(1..10)  //  crea el array [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Array es el objeto de SC para hacer arreglos con estructuras específicas sin tener que escribir cada uno de sus elementos. Existen una gran cantidad de mensajes para crear diferentes arreglos con el objeto Array. Daremos solamente tres ejemplos de mensajes, pero recomendamos estudiar todos los que vienen en el archivo de ayuda de Array y de los objetos de quien hereda.

```
Array.series(tamaño, comienzo, paso)
```

Crea una serie arímetica de valores, se definen tres argumentos que son: tamaño, comienzo y paso. Tamaño indica la cantidad de objetos o valores que contendrá el arreglo, comienzo nos dice a partir de que valor iniciar y paso es el la cantidad de valores intermedios entre cada objeto. (e.g. al valor se le suma el paso para obtener el siguiente valor)

Ejemplo de un Array usando el método .series

```
// se crea un Array de 5 valores que comienza en 2 y cuyo valor
subsecuente ira en aumento por valores de 4
Array.series(5, 2, 4)   //  el resultado es [2, 6, 10, 14, 18]
```

Array.geom(tamaño, comienzo, crecimiento)

Crea una serie geométrica de valores, se definen tres argumentos que son: tamaño, comienzo y crecimiento. Tamaño indica la cantidad de objetos o valores que contendrá el arreglo, comienzo nos dice a partir de que valor iniciar y crecimiento es el factor de crecimiento entre cada objeto. (e.g. el valor se multiplica por el factor de crecimiento para obtener el siguiente valor)

Ejemplo de un Array usando el método .geom

```
// se crea un Array de 5 valores que comienza en 2 y cuyo factor de
crecimiento es de 4
Array.geom(5, 2, 4)   //  el resultado es [2, 8, 32, 128, 512]
```

Array.rand(tamaño, mínimo, máximo)

Crea un arreglo de valores aleatorios, se definen tres argumentos que son: tamaño, mínimo y máximo. Tamaño indica la cantidad de objetos o valores que contendrá el arreglo, mínimo y máximo nos indican el límite inferior y el límite superior del rango de números entre los que se puede elegir. Se pueden repetir elecciones indefinidamente. (En caso de fraude) ← No leer

Ejemplo de un Array usando el método .rand

```
// se crea un Array de 5 valores aleatorios entre 0 y 10
Array.rand(5, 0, 10)   //  el resultado puede ser [0,5,8,6,8]
```

Notar que se eligió el número 8 dos veces

Los elementos de un arreglo tienen una posición que se indica por medio de un número llamado índice, El índice 0 corresponde al primer elemento del arreglo, el índice 1 corresponde al segundo elemento, el índice 2 al tercero y así sucesivamente.

Podemos pedir un elemento específico de un arreglo por medio del índice. Para hacer esto escribimos corchetes [] con el índice deseado después del arreglo.

```
x = [34, 51, 430, -2]     // definimos el arreglo x
x[0]  // pedimos el elemento en el índice 0 (e.g. el primer elemento = 34)
x[1]  // pedimos el elemento en el índice 1 (e.g. el segundo elemento = 51)
x[3]  // pedimos el elemento en el índice 0 (e.g. el cuarto elemento = -2)
```

Funciones mas arreglos

Podemos juntar funciones con arreglos para obtener ideas estructuradas.
Ejemplo:

Creamos el arreglo *a* con 4 elementos:

```
a=[261.6, 329.6, 392, 523.2]
```

Podemos pedir sus elementos por índice:

```
a[0]  //  el primer elemento de a es 261.6
a[1]  //  el segundo elemento de a es 329.6
a[2]  //  el tercer elemento de a es 392
a[3]  //  el cuarto elemento de a es 523.2
```

Creamos una función que itere 4 veces la suma +1 arrojando los valores 0, 1, 2, 3

```
4.do{ | i | i.postln }
```

En la post aparece esto:

```
0
1
2
3
4
```

El 4 no fué un valor de i. El 4 se imprime por que la función se realizó 4 veces.

Juntamos la función con el arreglo *a* usando i como índice

```
4.do{ | i | a[i].postln }  // nos imprime uno por uno los elementos de a en orden
```

```
261.6
329.6
392
```

523.2
4

El 4 no es parte del arreglo *a*. El 4 se imprime por que la función se realizó 4 veces.

Finalmente podemos hacer algo interesante: Usemos los elementos de *a* como argumentos de frecuencia para 4 SinOsc.

4.do{ | i | {SinOsc.ar(a[i], 0, 0.25)}.play }

La línea de código anterior produce lo mismo que las siguientes líneas, pero es más concisa y elegante.

```
(
{SinOsc.ar(261.6, 0, 0.25)}.play;
{SinOsc.ar(329.6, 0, 0.25)}.play;
{SinOsc.ar(392, 0, 0.25)}.play;
{SinOsc.ar(523.2, 0, 0.25)}.play;
)

// ================================
//  Pilón
// ================================

//  SuperCollider nos hace la tarea de crear la función
sinusoidal

{SinOsc.ar(100)}.plot

//  Pero nosotros tenemos ya las herramientas para hacerla

((2pi/100)*(0..100)).sin.plot
```

Música por computadora

Ernesto Romero y Hernani Villaseñor

Centro Multimedia 2012

Sesión 4

4 Herramientas propias de SuperCollider: ProxySpace, Demand

ProxySpace

Es parte de la librería de programación en tiempo real: JITLib o Just In Time Programming – Library.

Esta librería fue desarrollada por Julian Rohrhuber con la ayuda de Alberto de Campo y Fredrik Olofsson, la cual muestra los conceptos básicos de programación interactiva con SuperCollider y *ProxySpace.*

¿Programar en vivo?

Gracias a la posibilidad de declarar código sin tener que frenar las tareas que realiza el programa, la programación en vivo se hace posible. En el caso de SC la librería JITLib nos ayuda a realizar está práctica de manera muy efectiva, aunque no es la única técnica en la que podemos modificar código en tiempo real.

¿Qué es ProxySpace?

Un Proxy es un parámetro de sustitución, para algo que está sucediendo en el Servidor. Este parámetro puede ser usado para operar en algo que aún no existe. Cualquier objeto puede tener un comportamiento Proxy, especialmente las funciones.

Como iniciar ProxySpace

Para iniciar un ambiente Proxy o ProxySpace comenzamos declarando el siguiente código, que empuja la entrada a la librería JITLib, la cual nos permite modificar código en el vuelo para realizar prácticas como la de Live Coding.

```
p=ProxySpace.push(s.boot)

~out.play
```

Programamos a partir de funciones, las cuales contienen objetos que funcionan dentro de la lógica del Proxy, es decir que sus argumentos pueden ser modificados en el momento.

```
~out={SinOsc.ar(440,0,1)}
```

Argumentos y .set

Implementamos un argumento el cual podremos cambiar mediante el método .set, podemos asignar cuantos argumentos queramos y darles cualquier nombre siempre que empiecen con minúscula. Para usar el método .set se engloba entre paréntesis los argumentos que se van a modificar, se agrega una diagonal al nombre del argumento y a continuación se indica el nuevo valor, todo se separa por comas.

```
//creamos el argumento frec, lo igualamos a 440 y lo sustituimos
dentro de los argumentos de SinOsc

~out={|frec=440| SinOsc.ar(frec,0,1)}

//con el mensaje .set cambiamos la frecuencia
~out.set(\frec, 880)
```

Nota como cambia inmediatamente la frecuencia, sin necesidad de parar el código y volverlo a declarar, esta es la flexibilidad y poder de la programación en tin tiempo real. La sintáxis es importante, si te fijas el argumento que delclaramos como frec, al momento de implementarlo con el método .set se le añadie antes una diagonal, \frec.

Veamos un ejemplo de como implementar dos argumentos y modificarlos.

```
//creamos el argumento amp, lo igualamos a 1 y lo sustituimos
dentro de los argumentos de SinOsc

~out={|frec=440, amp0=1| SinOsc.ar(frec,0,amp)}

//con el mensaje .set cambiamos la frecuencia y la amplitud
~out.set(\frec, 880, \amp, 0.5)
```

.fadeTime y .xset

fadeTime es un método que implementa un desvanecimiento gradual de volumen cada ves que modificamos un argumento o que ponemos en pausa el Proxy. El método .xset funciona igual que .set, la diferencia es que .xset genera un cruce de volumenes (*crossfade*) entre el valor anterior del argumento y el nuevo valor. El tiempo que dura el cruce de volumenes está determinado por al valor que asignamos a fadeTime.

```
~out={|frec=440, amp0=1| SinOsc.ar(frec,0,amp)}

// usamos .fadeTime para asignar un tiempo de desvanecimiento en segundos
~out.fadeTime=5

//con el mensaje .xset cambiamos la frecuencia y veamos como el valor anterior y el nuevo se cruzan en un tiempo de 5 segundos
~out.xset(\frec, 880)
```

Canales

Al utilizar ProxySpace de esta manera la salida del sonido es por el primer canal de SuperCollider, o sea el lado izquierdo, para salir en ambos canales podemos usar el UGen Pan2.ar

```
//salida solo por el canal izquierdo
~out={Pulse.ar(2,0.5,0.1)}

// salida por ambos cnalaes, usando una iteración !2
~out={Pulse.ar([1,2],0.5,0.1)}

// salida por ambos canales usando Pan2.ar
~out={Pan2.ar(Pulse.ar(2,0.5,0.1),0,1)}
```

pause/resume

Con el mensaje .pause mandado a la variable ponemos en pausa las funciones o las operaciones asignadas a ella. Con resume regresamos de la pausa lo que esta contenido en la variable. El sintetizador sigue activo mientras está en pausa.

```
~out.pause
~out.resume
```

send/release

Send y release son mensajes que parece que hacen la misma funcion que los anteriores pause/resume, la diferencia es que send enciende un nuevo sinte y release lo libera. El efecto auditivo es que entra y sale suavemente el sonido dependiendo del tiempo de desvaneciemiento.

```
~out.release
```

```
~out.send
```

rebuild y el factor aleatorio

Con los mensajaes rand y exprand podemos agregar comportamiento aleatorio a los UGens. El mensaje .rebuild nos reconstruye los valores *random* pudiendo así modificar los valores de argumentos en random con tan solo mandar el mensaje desde la variable.

```
// un sinte con valores aleatorios, en este caso rrand es aplicado a el argumento de frecuencia

~out={SinOsc.ar(rrand(100,1000),0,0.1)}

// declarando esta línea de código reconstruye los valores random

~out.rebuild
```

p.clear

Remueve todos los sintes, grupos y monitor además libera el bus, el número entre paréntesis indica el tiempo en segundos que tardará en realizarse esta acción.

```
p.clear(10)
```

p.pop

Este mensaje es usado para salir del ambiente Proxy.

```
p.pop
```

Demand

Demand.kr (trigger, reset, demand rate Ugens)

Extrae elementos de uno o varios arrays con cierta periodicidad y con cierto orden.

La clase Demand trabaja siempre junto con otra clase de tipo *demand rate*. Estos UGens se escriben siempre empezando con la letra D mayúscula. Por ejemplo Dbrown, Dgeom, Dseries, etc. Los *demand rate* UGens definen los elementos de un arreglo y la manera en que serán entregados cuando el Demand lo solicite. Esta entrega puede ser de varias formas: en secuencia, aleatoriamente, siguiendo un patrón aritmético o geométrico, etc. Demand puede trabajar con .kr o con .ar cuidando que el trigger si es un UGen sea del mismo rate (e.g. Demand.ar (Impulse.ar).

Los argumentos de Demand son:

Trigger: Puede ser una señal o un argumento que se modifica desde afuera. Cuando el trigger cambia de no positivo a positivo se extrae un elemento.

Reset: Reinicia los Demand Ugens de modo que el siguiente elemento que se extraiga sea el primero de cada UGen.

Demand UGens: Puede ser cualquiera de los *demand rate* UGens como Dseq o Drand. Estos Ugens contienen los arreglos de donde el Demand extrae los elementos. Cada UGen se comporta de manera distinta el entregar los elementos de los arreglos.

Veamos algunos *demand rate* UGens:

Dseq (array, length)

Entrega en orden los elementos de un array un determinado número de veces.

array: array con los elementos en orden que se quieren entregar al Demand.

length: número de repeticiones. Una repetición es la lectura de todo el array.

Ejemplo 1

Aqui creamos dos argumentos: t_trig para pedir un elemento nuevo del array y t_reset para reiniciar el Dseq. Es necesario usar este tipo de argumentos antecedidos por t_ para que funcione. En el tercer argumento del Demand ponemos un Dseq que nos entregará en secuencia los elementos de su array infinitas veces. El Demand está asignado a la variable ~freq que hace las veces de frecuencia para un SinOsc.

```
p=ProxySpace.push(s.boot)

~out.play

~freq={|t_trig,t_reset|
Demand.kr(t_trig,t_reset,Dseq([440,493.88,554.36,587.32,659.25,7
39.98,830.6,880],inf))};

~sig={SinOsc.ar(~freq.kr)};

~freq.set(\t_trig, 1)
~freq.set(\t_reset, 1)

~out=~sig
```

Ejemplo 2

Ahora usamos una señal como trigger. La señal es un Impulse que pedirá un elemento al Dseq 2 veces por segundo. Dseq entregará los elementos de su array una sola vez.

```
~freq={|t_reset|
Demand.kr(Impulse.kr(2),t_reset,Dseq([440,493.88,554.36,587.32,6
59.25,739.98,830.6,880],1))};
~sig={SinOsc.ar(~freq.kr)};

~freq.set(\t_reset, 1) // vuelve a empezar la secuencia desde el
primcipio

~out=~sig

~out=0
```

Drand (array, length)

Entrega en orden aleatorio los elementos de un array un determinado número de veces.

array: array con los elementos que se quieren entregar al Demand.

length: número de elementos del array que se entregarán. Diferente de Dseq donde se cuentan las veces que se entrega todos los elementos. En Drand se cuenta elemento por elemento.

Ejemplo 3

Igual que el Ejemplo 1 pero con Drand. Aqui el uso del t_reset es irrelevante ya que, al ser aleatorio el orden, no hay un primer elemento.

```
~freq={|t_trig|
Demand.kr(t_trig,0,Drand([440,493.88,554.36,587.32,659.25,739.98
,830.6,880],inf))};

~sig={SinOsc.ar(~freq.kr)};

~freq.set(\t_trig, 1)

~out=~sig
```

Ejemplo 4

Ahora usamos una señal como trigger. La señal es un Impulse que pedirá un elemento al Drand 2 veces por segundo. Dseq entregará solo 4 elementos de su array.

```
~freq={Demand.kr(Impulse.kr(2),0,Dseq([440,493.88,554.36,587.32,
659.25,739.98,830.6,880],4))};

~sig={SinOsc.ar(~freq.kr)};

~out=~sig

~freq.rebuild  //  para que vuelva a arrojar 4 elementos al azar

~out=0
```

Ejemplo 5

Se pueden poner otros demand rate Ugens como elementos del array de un demand rate Ugen

```
~freq={Demand.kr(Impulse.kr(8),0,Dseq([Dseq([211.8,200,211.8,224
.4,211.8,200,211.8],1), Drand
     ([399.9,336.3,158.7,118.9],1)],inf))};
 ~sig={Blip.ar(freq*200, 2)};
```

Ejemplo 6

```
~freq={Demand.kr(Impulse.kr(8),0,Dseq([Dseq([211.8,200,211.8,224
.4,211.8,200,211.8],1), Drand
     ([399.9,336.3,158.7,118.9],1)],inf))};
~armonicos={Demand.kr(Impulse.kr(10),0,Dseq([4,2,3,4,5,6,7,8,9,1
0,11,12,13].pyramid,inf))};
~sig={Blip.ar(~freq.kr, ~armonicos.kr)};

// ejemplo del help de Demand

(
{
     var trig, seq;
     trig = Impulse.kr(12);
     seq = Drand([
     Dseq([4,1,0,2,2,0,1,2]),
     Dseq([1,0,0,0,2,1]),
     Dseq([4,1,0,1,1]),
     Dseq([4,4,0,0])], inf);
     trig = Demand.kr(trig, 0, seq * 0.4) * trig;
     {LPF.ar(PinkNoise.ar, 12000)}.dup * Decay.kr(trig, 0.5);
}.play;
)
```

Dswitch (array, index)

Posee un array de donde puede escoger a voluntad un elemento y mandarlo al Demand.

array: array con los elementos que se pueden entregar al Demand. Pueden ser números, demand rate Ugens o alguna otra Clase.

index: Indice del elemento del array que queremos entregar al Demand.

Ejemplo 6

Usamos un Dswitch para escoger diferentes patrones numéricos y asignarlos a la frecuencia de un Pulse. Creamos el argumento switch dentro de la funcion para poder escoger el índice del array del Dswitch.

```
~freq={|switch| Demand.kr(Impulse.kr(8), 0,
Dswitch1([Dseq([300,336.7,377.9], inf), Dseq([300,336.7,356.7],
inf), 300, MouseY.kr(600,300)], switch))};
~sig={Pulse.ar(~freq.kr)!2};

~out=~sig

~freq.set(\switch, 0)  //  Dseq([300,336.7,377.9], inf)
~freq.set(\switch, 1)  //  Dseq([300,336.7,356.7], inf),
~freq.set(\switch, 2)  //  300
~freq.set(\switch, 3)  //  MouseY.kr(600,300)
```

Ejemplo 7

Se puede utilizar el Demand con audio rate para generar ondas. Recuerden que el trigger debe tener audio rate también

```
Array.series(100,0,0.1).sin.plot  //  los puntos que forman la
onda

~sig={|freq=7033|Demand.ar(Impulse.ar(freq), 0,
Dseq(Array.series(600,0,0.1).sin,inf))};

~out=~sig

~sig.set(\freq, rrand(200,7033))

s.scope

~sig={|freq=7033| Demand.ar(Impulse.ar(freq),0,Dseq([0,-
1,0.4,0.6,0.9,0.1,1,0.5],inf))}

~sig.set(\freq, rrand(200,7033))

~sig={|freq=7033|
Demand.ar(Impulse.ar(freq),0,Dseq(Array.geom(10,1,0.28).reverse.
mirror,inf))}
```

Música por computadora

Ernesto Romero y Hernani Villaseñor

Centro Multimedia 2012

Sesión 5

Canales - Out
SynthDef

Canales de salida

SuperCollider es un programa que nos permite manejar la salida del sonido por múltiples canales. Las tarjetas de sonido de las computadoras están configuradas en estéreo, es decir cuentan con dos canales: izquierdo y derecho, que en SC están asignados al número 0 para izquierdo y 1 para derecho.

En el caso de contar con una interfaz de audio tenemos la posibilidad de usar más de 2 canales. Algunas interfaces de audio cuentan hasta con 8 salidas simultáneas. Los 8 canales de una interfaz de este tipo enumera sus canales a partir del número 1, mientras que SC lo hace a partir del 0. Entonces, si mandamos una señal por el canal 0 de SC saldrá por el canal 1 de la interfaz.

Canales de la Tarjeta de sonido	1 2 3 4 5 6 7 8
Canales de salida de SuperCollider	0 1 2 3 4 5 6 7

En SC existen varias clases que nos ayudan a trabajar con los canales de audio asignando por donde saldrá el sonido, dos de las más comunes son: Out.ar y Pan2.ar.

Out.ar (canal, señal)

Asigna la salida del sonido a partir de un canal específico. Ese canal define un punto de partida u offset a partir del cual se va a distribuir el sonido.

canal: 0 izquierdo, 1 derecho, 3, 4, 5, ...multicanal
señal: cualquier UGen.

Ejemplos:

```
{Out.ar(0,Saw.ar(40)*EnvGen.kr(Env.perc(0.01,1),doneAction:2))}.
scope  // izquierda
```

```
{Out.ar(1,Saw.ar(40)*EnvGen.kr(Env.perc(0.01,1),doneAction:2))}.
scope  // derecha

{Out.ar(2,Saw.ar(40)*EnvGen.kr(Env.perc(0.01,1),doneAction:2))}.
scope  //  un tercer canal que solo sonara si tenemos una
interface de audio
```

Pan2.ar (señal, posición)

Distribuye el sonido entre dos canales consecutivos conservando su potencia. Es decir, que no suena más fuerte cuando está en los dos canales al mismo tiempo ni más quedito cuando está solo en uno o en otro. Si el Pan2 esta dentro de un Out los canales consecutivos en los que se distribuyen se cuentan a partir del offset del Out.

señal: cualquier oscilador o generador de sonido.
posición: -1 izquierda, 0 centro, 1 derecha y con todo el intervalo continuo entre -1 y 1 extrapolando el sonido entre los dos canales o bocinas.

```
{Pan2.ar(Pulse.ar(100,0.01, 0.2),-1)}.scope // izquierdo

{Pan2.ar(Pulse.ar(100,0.01, 0.2),0)}.scope // centro

{Pan2.ar(Pulse.ar(100,0.01, 0.2),1)}.scope // derecho
```

Dentro de un Out con canal offset en 1

```
{Out.ar(1,Pan2.ar(Pulse.ar(100,0.01, 0.2),-1))}.scope // derecho

{Out.ar(1,Pan2.ar(Pulse.ar(100,0.01, 0.2),0))}.scope // derecho
y tercer canal

{Out.ar(1,Pan2.ar(Pulse.ar(100,0.01, 0.2),1))}.scope // tercer
canal
```

SynthDef

Es una clase para crear sonidos complejos a partir de una plantilla de código. Su variedad y funcionalidad es bastante flexible y permite construir sonidos con alto grado de manipulación. Podríamos pensarlo como el diseño de los instrumentos de una orquesta que posteriormente sonarán cuando se ejecuten notas con ellos.

Para entender su funcionamiento analicemos el siguiente SynthDef:

```
(
SynthDef(\sinte1,{|frec=1000, amp=0.5, out=0|
     var sen, env;
     sen = SinOsc.ar(frec,0,amp);
     env = EnvGen.kr(Env.perc(0.1,1),doneAction:2);
     Out.ar(out, sen * env)
     }).send(s)
);

Synth(\sinte1)
```

Lo primero es declarar el SynthDef y ponerle un nombre, para lo que usamos una diagonal invertida y a continuación nombramos el sinte como queramos:

```
(
SynthDef(\sinte1,
```

Luego abrimos una llave y entre pipes | | declaramos cuantos argumentos queramos usar.

```
{|frec=1000, amp=0.5, out=0|
```

Enseguida declaramos nuestras variables, recordemos que tanto argumentos como variables pueden llamarse como queramos, siempre y cuando comiencen con una letra minúscula.

```
     var sen, env;
```

A continuación le damos una funcionalidad a nuestras variables, en este caso le asignamos el UGen de SinOsc a la variable sen y le asignamos a la variable env un envolvente con el objeto EnvGen.kr de tipo percusivo Env.perc.

```
     sen = SinOsc.ar(frec,0,amp);
     env = EnvGen.kr(Env.perc(0.1,1),doneAction:2);
```

Por último asignamos la salida de nuestro SynthDef a 0, con lo cual sonará a partir del canal izquierdo. Multiplicamos sen por env, lo que dará como resultado que el tono de 1000 Hz suene el tiempo que tiene programada la envolvente, en este caso 1.1 segundos lo que ocasiona que suene un pequeño “blink”.

```
     Out.ar(out, sen * env)
```

Y con el mensaje .send(s) le indicamos que se cargue al servidor.

```
      }).send(s)
)
```

Para hacer sonar nuestro SynthDef declaramos la siguiente línea de código, la cual indica el nombre del sinte que queremos hacer sonar.

```
Synth(\sinte1)
```

Nota como en el servidor se inserta un nuevo sinte y como desaparace en el momento que termina de sonar, esto es debido al tipo de envolvente usada y el método con el que decimos como ingrese al servidor.

Para modificar los valores de un SynthDef necesitamos igualarlo a una variable y posteriormente mandar los cambios mediante el mensaje .set

```
a=Synth(\sinte1)

// el argumento se antecede de una diagonal invertida y después
de la coma se ingresa el nuevo valor, en este caso 1200

a.set(\frec, 1200)
```

Envolventes

Los envolventes determinan el comportamiento de un sonido respecto al desarrollo de la amplitud en el tiempo, más adelante los veremos con detalle. Por el momento podemos decir que SC maneja envolventes de tipo percusivo, o sea que aparecen y desaparecen por si mismos y otros que requieren de una orden para dejar de sonar, o sea que funcionan mediante un *gate* o disparo.

En este caso el envolvente de tipo percusivo tarda 0.1 de segundo en aparecer y 1 segundo en desaparecer.

```
EnvGen.kr(Env.perc(0.1,1),doneAction:2)
```

Envolvente de tipo asr, tarda 0.1 segs en aparecer, hasta que alcanza la amplitud 1, nosotros le indicamos cuando deje de sonar mediante un *gate*, lo que en este caso le llevará 1 seg en desaparacer.

```
(
var gate=1;
EnvGen.kr(Env.asr(0.1,1,2),gate,doneAction:2);
)
```

Ejemplo de un SynthDef con envolvente percusivo y paneo

```
(
SynthDef(\sinte2,{|frec= 500|
     var sen, pan, env;
     sen=LFTri.ar(frec,0,1);
     pan=Pan2.ar(sen,0,1);
     env=EnvGen.kr(Env.perc(0.1,2),doneAction:2);
Out.ar(0, pan * env);
}).send(s)
)

b=Synth(\sinte2);
```

Ejemplo de un SynthDef con envolvente asr y paneo

```
(
SynthDef(\sinte3,{|frec= 500, gate=1, paneo=0|
     var sen, pan, env;
     sen=LFTri.ar(frec,0,1);
     pan=Pan2.ar(sen,paneo,1);
     env=EnvGen.kr(Env.asr(0.1,1,5),gate,doneAction:2);
Out.ar(0, pan * env);
}).send(s)
)

c=Synth(\sinte3);
c.set(\frec, 600);
c.set(\frec, 700, \paneo, -1);
c.set(\frec, 800, \paneo, 1);
c.set(\frec, 900, \paneo, 0);
c.set(\gate, 0)
```

Música por computadora
Ernesto Romero y Hernani Villaseñor
Centro Multimedia 2012

Sesión 6

Tdef, boleanos, if, switch.

Tdef

Veamos que es y como funciona la herramienta Tdef de SuperCollider. Tdef viene de Task definition, en español definición de tarea. Tdef es útil para realizar precisamente tareas que requieren varios pasos en su desarrollo. Es un tipo de rutina, donde rutina es el término general empleado en el ambito de los lenguajes de programación para designar un paquete de acciones que han de realizarse en cierto orden y en determinado tiempo. Para hacer un Tdef se necesita escribir primero su forma básica:

Ejemplo 1

```
Tdef(\baladi, { "no soy baladi".postln })
Tdef(\baladi).play
```

El Tdef necesita tener un nombre que se escribe antecedido de una diagonal invertida: \nombre. Esto es un símbolo. El Tdef anterior se llama \baladi. Después del nombre hay que agregar una función; habremos de usar las llaves { } para esto. Dentro de las llaves escribimos lo que queremos que haga el Tdef. El Tdef del ejemplo 1 tiene por tarea imprimir en la post window el mensaje "no soy baladi" una vez. Para que el Tdef realice su tarea hay que enviarle el mensaje .play. Podemos pedirle al Tdef que realice más de una acción en su tarea. Imprimir por ejemplo 2 mensajes. Observar que hay que poner punto y coma ; al final de cada acción.

Ejemplo 2

```
Tdef(\baladi, {
"no soy baladi".postln;
"o si?".postln;
})
```

Podemos pedirle al Tdef que espere cierta cantidad de tiempo entre la realización de las acciones.

Para esto le enviamos el mensaje .wait a un número que representa la cantidad de tiempo en segundos que queremos que espere.

Ejemplo 3

```
Tdef(\baladi, {
     "no soy baladi".postln;
     2.wait; // espera dos segundos
     "o si?".postln;
     1.wait; // espera un segundo
     "%&*^@#$*&*".postln;
     {SinOsc.ar(SinOsc.kr(10,0,100,400))*Line.kr(1,0,2)}.play
});

Tdef(\baladi).play
```

Ahora bien, si juntamos la estructura n.do{ } con el Tdef podemos repetir n veces las acciones que queramos.

Ejemplo 4

```
Tdef(\baladon, {
     8.do{
     {SinOsc.ar(SinOsc.kr(10,0,100,400))*Line.kr(1,0,0.1)}.play;
     0.25.wait
     };
     16.do{
     {SinOsc.ar(SinOsc.kr(15,0,100,700))*Line.kr(1,0,0.1)}.play;
     0.125.wait
     }
});

Tdef(\baladon).play
```

Podemos anidar funciones y hacer infinitas repeticiones de ellas escribiendo inf.do

Ejemplo 5

```
Tdef(\baladon, {
     inf.do{
     2.do{
     {SinOsc.ar(SinOsc.kr(10,0,100,400))*Line.kr(1,0,0.1)}.play;
     0.25.wait
     };
     2.do{
     {SinOsc.ar(SinOsc.kr(15,0,100,700))*Line.kr(1,0,0.1)}.play;
     0.25.wait
     };
```

```
      4.do{
      {SinOsc.ar(SinOsc.kr(10,0,100,400))*Line.kr(1,0,0.1)}.play;
      0.125.wait
      };
      1.do{
      {SinOsc.ar(SinOsc.kr(15,0,100,700))*Line.kr(1,0,0.1)}.play;
      0.5.wait
      }
      }
});

Tdef(\baladon).play

Tdef(\baladon).stop
```

Booleanos y condicionantes

Los conceptos de boleanos y condicionantes nos ayudarán a introducir estructuras más complejas en nuestros Tdefs.

Los booleanos emplean operadores relacionales (e.g. <, >, ==) entre datos para obtener un valor lógico de verdadero o falso.

Ejemplo 6

```
4 < 8 // ¿Es cuatro menor que ocho? true
4 > 8 // ¿Es cuatro mayor que ocho? false
2 == 3 // ¿Es dos igual a tres? false
2 != 3 // ¿Es dos diferente de tres? true
```

El simbolo doble == pregunta si los valores son iguales. Es diferente del simbolo sencillo = que nos sirve para asignar valores a variables.

```
a = 10 // a es igual a 10
a == 10 // ¿Es a igual a 10?
```

Conociendo esto podemos usar los booleanos como condición para la ejecución de una función en combinación con la noción if.

Para usar if necesitamos un booleano, una función a ejecutar en caso de que el booleano sea verdadero y otra función a ejecutar en caso de que el booleano sea falso.

```
if(true, {"neto"}, {"cabula"})
if(false, {"neto"}, {"cabula"})
if(4 < 8, {"neto"}, {"cabula"})
if(4 > 8, {"neto"}, {"cabula"})
if(2 == 3, {"neto"}, {"cabula"})
if(2! = 3, {"neto"}, {"cabula"})
```

Recordemos ahora la iteración que viene implementada en las funciones:

```
10.do{|i| i.postln}
```

Si evaluamos el argumento *i* con una condicionante podemos hacer cosas como:

```
10.do{|i| if(i<5,{"si es menor".postln}, {"nel, no es
menor".postln})}
```

Y metiendolo dentro de un Tdef:

```
Tdef(\menor, {
10.do{|i| i.postln;
if(i<5,{"si es menor que 5".postln}, {"nel, no es menor que
5".postln});
0.5.wait;
}
})

Tdef(\menor).play
```

Podemos pedir que solo cuando *i* sea igual a cierto valor se ejecute una acción.

```
Tdef(\si, {
inf.do{
4.do{|i| i.postln;
if(i==0,{{SinOsc.ar(SinOsc.kr(2,0,500,700))*Line.kr(1,0,0.1)}.play;},
{{SinOsc.ar(SinOsc.kr(15,0,100,500))*Line.kr(1,0,0.1)}.play;}
);
0.5.wait;
}
}
})

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

Modulo %

El concepto de modulo nos limita la cantidad de números que tenemos para contar. Por ejemplo, si decimos que estamos en modulo 5 solo tenemos 5 números para contar: 0, 1, 2, 3 y 4. Como ven se empieza a contar desde el cero. Si tuvieramos que contar hasta el 5 ya no podemos por que solo tenemos hasta el 4. En ese caso tenemos que regresar al cero.

Entonces una enumeración en modulo cinco quedaría así:

```
0%5=0
1%5=1
2%5=2
3%5=3
4%5=4
5%5=0
6%5=1
7%5=2
8%5=3
9%5=4
10%5=0
```

Si usamos el modulo % dentro de un Tdef podemos hacer ciclos como el siguiente:

```
Tdef(\mod, {
     inf.do{|i| (i%8).postln;
     0.5.wait;
     }
})

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

Switch

Switch es una especie de condicionante if pero que puede verificar la igualdad entre más de un valor.

```
Tdef(\si, {
     inf.do{|i|
switch((i%8).asInteger, 0,{{SinOsc.ar(345)*Line.kr(1,0,0.1)}.play;},
1,{{SinOsc.ar(1345)*Line.kr(1,0,0.1)}.play;},
2,{{Pulse.ar(35)*Line.kr(1,0,0.1)}.play;},
3,{{WhiteNoise.ar(0.7)*Line.kr(1,0,0.1)}.play;},
4,{{SinOsc.ar(345)*Line.kr(1,0,0.1)}.play;},
5,{{Pulse.ar(75)*Line.kr(1,0,0.1)}.play;},
6,{{Pulse.ar(20)*Line.kr(1,0,0.1)}.play;},
7,{{WhiteNoise.ar(0.7)*Line.kr(1,0,0.1)}.play;}
);
0.2.wait;
}
})

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

Música por computadora

Ernesto Romero y Hernani Villaseñor
Centro Multimedia 2012

Sesión 7

Patterns

Los Patters son patrones con los que podemos crear secuencias, los patrones o Patterns se pueden expresar de varias maneras y dependen a su vez de Patterns capaces de empaquetar el flujo de datos producido por distintos tipos de Patterns como es el caso de Pbind.

Pbind (llave, patrón1, llave2, patrón2...)

Combina una serie de flujos (*streams*) provenientes de varios valores, en un solo *stream* de datos. La estructura Pbind se crea de la siguiente manera:

```
// en este ejemplo usamos un Synth que viene en SC, y comenzamos
un stream con el valor de duración
Pbind(\dur, 0.25).play

// agregamos el argumento freq
Pbind(\dur, 0.25, \freq, 440).play
```

Lo que suena es un Synth establecido internamente en SC, los argumentos dur y freq ya están programados para crear flujos de datos concernientes a parámetros definidos, en este caso duración y frecuencia.

Hay varios argumentos que podemos utilizar y que están previamente programados:
\dur, \freq, \amp, \pan, \sustain, \note, \midinote

Los métodos que utiliza Pbind son entre otros: .play, .stop, .clock, .quant

Podemos usar patrones para generar secuencia por ejemplo variar la duración de cada nuevo evento del *stream* así como su frecuencia, para lograr estos patrones usamos: Pseq, Prand, Pcollect, etc. Veremos dos.

Pseq ([lista], repeticiones)

Pertenece a `ListPatterns` y deposita valores en una lista de manera secuencial. Esta formado por una lista de donde tomará los valores en el orden que esten colocados, después indicamos cuantas veces queremos que suceda la secuencia de valores.

```
//controlamos la duración mediante una lista de valores
Pbind(\dur, Pseq([0.125,0.25,0.5,1],inf).play
```

Prand ([lista], repeticiones)

Pertenece a `ListPatterns` y deposita valores en una lista de manera aleatoria. Es practicamente igual que `Pseq` pero con la variante de que los valores de la lista son leídos de manera aleatoria.

```
//controlamos la duración mediante una lista de valores
Pbind(\dur, Prand([0.125,0.25,0.5,1],inf).play
```

Pdef

`Pdef` es una rutina donde podemos anidar un `Pbind` y la cual podemos modificar y evaluar en tiempo real sin tener que detener SC, ya que es una clase que funciona mediante una referencia a un flujo de datos. Esta clase es parte de la librería JITLib.

```
// este es un modelo muy sencillo de un Pdef
Pdef (\secuencia,
     Pbind (\dur, Pseq([0.25,0.5],inf),
     \freq, Prand([200,250,300,350],inf)).play

// igualamos el Pdef a una variable
a=Pdef (\secuencia,
     Pbind (\dur, Pseq([0.25,0.5],inf),
     \freq, Prand([200,250,300,350],inf))

a.play
a.stop
```

Podemos modificar y evaluar en tiempo real el código interno del `Pdef` sin que tengamos que parar el servidor.

Silencios

Los silencios son aplicables a llaves con modelos de altura como frecuencia o nota, cualquier símbolo es interpretado como silencio.

```
// nota como los siguientes símbolos dan silencio: \, \silencio
a=Pdef (\secuencia,
     Pbind (\dur, Pseq([0.25,0.5,0.5],inf),
     \freq, Pseq([300,\,350,400,450,\silencio],inf)))

a.play
a.stop
```

Otra forma de generar silencios es usar una lista antecedida del caracter #, y colocar una r donde queremos el silencio.

```
// nota como en el argumento freq usamos la r para indicar
silencio
a=Pdef (\secuencia,
     Pbind (\dur, Pseq([0.25,0.5,0.5],inf),
     \freq, Prand(#[200,r,250,300,350],inf)))

a.play
a.stop
```

Patrones con nuestros instrumentos

Para usar sonidos diseñados mediante SynthDef, es necesario mandarlos al servidor con el mensaje .add por convención.

```
SynthDef(\triangulo,{|frec=400, amp=1, paneo=0|
     var sen, pan, env;
     sen=LFTri.ar(frec, 0, amp);
     pan=Pan2.ar(sen, paneo, amp);
     env=EnvGen.kr(Env.perc(0.1, 1), doneAction:2);
Out.ar(0,pan*env)
}).add
```

Ahora solo necesitamos usar el argumento `\instrument` y el nombre de nuestro sintetizador, en este caso `\triangulo`, para añadirlo a la cadena de argumentos y valores de la estructura `Pdef`.

```
// nota que usamos \instrument con el nombre del synth para
decirle a Pdef que reconozca nuestro sonido

(
c=Pdef (\secuencia,
     Pbind (\instrument, \triangulo,
     \dur, Pseq([0.25,0.5,0.5],inf),
     \freq, Prand([200,250,300,350],inf)))
)

c.play
c.stop
```

Música por computadora

Ernesto Romero y Hernani Villaseñor
Centro Multimedia 2012

Sesión 8

Live Coding

¿Qué es *Live Coding*?

Live coding es el ejercicio escénico de escribir o modificar código en el momento, con el fin de obtener un resultado estético, generalmente visual o sonoro. El código que se escribe normalmente se proyecta sobre una pantalla, de tal forma que puede ser visto por el público.

El *live coding* es una corriente que comenzó a tomar fuerza en Europa en los años 90 y actualmente es un ejercicio dentro del arte electrónico experimental. McLean (129), menciona que el término surgió alrededor del 2003, para describir una actividad con una aproximación a nuevas formas de hacer música por computadora y video animación. Asimismo sugiere que el término l*ive Coding* se usa más en el contexto de la improvisación.

Algunos festivales que integran las prácticas de *live coding* son Changing Grammars en Alemania, LOSS Livecode Festival y //FLOW en Inglaterra, Piksel en Noruega, Make Art en Francia, Lambda en Bélgica, /*vivo*/ en México y Live.Code.Festival en Alemania.

Desde el 2010 el Taller de Audio del CMM ha impulsado la práctica del *live coding* a partir de talleres, conferencias y conciertos mensuales y en 2012 con el Simposio Internacional de /*vivo*/.

Otras plataformas

ChucK
Pure Data
Impromptu
Fluxus
live-processing
Max/MSP

ProxySpace

ProxySpace es uno de los cuatro ambientes de la librería JITLib creada por Julian Rohrhurber, con la que modificamos código mientras este corre dentro de un ambiente especial llamado Proxy.

```
//prendemos el servidor y entramos al ambiente Proxy
p=ProxySpace.push(s.boot)

// declaramos un desvanecimiento del tiempo, 10 segundos
p.fadeTime=10

// creamos una variable global con el mensaje .play
~out.play

// la igualamos a una función con un UGen dentro
~out={SinOsc.ar(400,0,0.5)}

// agregamos un argumento a frecuencia y lo llamamos frec
~out={|frec=400| SinOsc.ar(frec,0,0.5)}

// modificamos el argumento frec mediante el método .set
~out.set(\frec, 600)
```

Ejemplo con Demand

```
~out2.play
~out2={SinOsc.ar (Demand.kr(Impulse.kr(6), 0,
Dseq([200,100,300],inf)))}

~out3.play
~out3={LFTri.ar (Demand.kr(Impulse.kr(2), 0,
Dseq([1000,500,250],inf)),0,0.2)}

~out4.play
~out4={LFTri.ar (Demand.kr(Impulse.kr(1), 0,
Drand([250,500,400,750],inf)),0,0.2)}
```

Ejemplo con Pbind

```
~out5= Pbind(\freq, Pseq([250,500],inf))
```

```
~out5.play
```

SynthDef dentro de Proxy

Para usar SynthDef dentro de ProxySpace, es necesario declarar el Synth con .add.

Ejemplo:

```
SynthDef(\sinte, {|frec=200,out=0|
Out.ar(out,SinOsc.ar(frec,0,0.1)
* EnvGen.kr(Env.perc(0.1,1),doneAction:2))}).add
```

De esta manera tenemos la posibilidad de usar un Synth en combinación con un Pdef. Otra forma de usar un SynthDef es de la siguiente manera.

```
// sint con demand y sin enviar al servidor

~out6=SynthDef(\hoy2,{
|frec=100|
Out.ar(0,SinOsc.ar(Demand.ar(Impulse.ar(4),0,Drand([100,200,400],
inf)),0,0.1)
* EnvGen.kr(Env.asr(0.1,0.4,2),1,doneAction:2))})

~out6.play
```

Ejemplo con Pdef

```
// usamos el SynthDef anterior llamado \sinte
~seq.play
~seq=Pdef(\uno,Pbind(\instrument,\sinte, \dur, Pseq([0.5],inf)))
~seq.release
~seq.send
```

Referencias

McLean, A. (2011). *Artist-Programmers and Programming Languages for the Arts.* (Tesis de Doctorado). University of London.

Música por computadora

Ernesto Romero y Hernani Villaseñor
Centro Multimedia 2012

Sesión 9

Teoría del sonido

Sonido, Frecuencia, Amplitud, Periodo, Velocidad, Longitud de onda, Fase, Armónicos/Timbre, Envolvente.

Sonido

El sonido es la perturbación de las moléculas del aire cuando un objeto o cuerpo entra en vibración. Esta vibración es cíclica y está formada por una frecuencia y una amplitud. El sonido viaja por el aire y al llegar a nuestros oídos hace vibrar al tímpano el cual esta conectado a una serie de huesitos que transmiten esta vibración al caracol, el cual contiene un líquido donde unas pequeñas vellosidades llamadas cilios convierten esta información en impulsos eléctricos, los cuales son interpretados por nuestro cerebro. El entorno afecta el comportamiento del sonido, es decir, su comportamiento es diferente cuando el sonido se produce al aire libre que cuando se produce dentro de un recinto.

Frecuencia

Son los ciclos por segundo que contiene la vibración del sonido, se mide en Hertz. A frecuencias más altas sonidos más agudos y viceversa. El oído humano puede escuchar de 20 Hz a 20,000 Hz.

```
{SinOsc.ar(Line.kr(20,20000,20),0,0.5)}.play
```

Amplitud

Es la fuerza con la que oscila el sonido, se mide en decibeles. Los límites a los que puede someterse el oído humano sin sufrir daño son 120 dB considerado como umbral de la sensación. Una conversación a un metro de distancia llega a tener una amplitud de 70 dB aproximadamente. En SuperCollider la amplitud esta normalizada por lo que ocurre en valores de 0 mínimo a 1 máximo.

El tercer argumento de SinOsc se llama multiplicador y tiene el efecto de incrementar el volumen o amplitud de nuestro sonido, es este caso valores que van de 0 a 1, valor de amplitud normalizada.

```
// amplitud máxima sin distorsión
{SinOsc.ar(440,0,1)}.play

// amplitud intermedia
{SinOsc.ar(440,0,0.5)}.play

// amplitud mínima
{SinOsc.ar(440,0,0)}.play
```

Velocidad

La velocidad del sonido es de 344 metros/segundo a una temperatura de 21 grados centigrados. Se suma 0.6 m/s por cada grado que sube la temperatura o se resta en caso de que baje.

V = 344 m/s @ 21 grados centigrados +/- 0.6

Ejemplo

Cálculo de la velocidad del sonido a 0 grados centigrados.

```
// primero multiplicamos 0.6 por cada grado centigrado que baja
la temperatura a partir de nuestra referencia de 21 grados
0.6 * 21
12.6

// el resultado se lo restamos a 344
344 - 12.6
331.4
```

331.4 metros/segundo es la velocidad del sonido a 0 grados centigrados. Intenta cualcular la velocidad del sonido a 30 grados centígrados.

Longitud de Onda

Es la longitud de un ciclo de sonido expresado en metros y se calcula con la siguiente fórmula:

L = V/F

L = longitud de onda
V = velocidad del sonido
F = frecuencia

Ejemplo

Calcular la longitud de de onda de 500 Hz a 21 grados C.

```
// usando la fórmula L=V/F, sustituimos los datos
344 / 500

// el resultado es 0.688 metros
0.688
```

Periodo

El periodo es el tiempo que dura un ciclo de una frecuencia en desenvolverse, se expresa por la letra T. Para calcular el periodo se usa la siguiente fórmula:

T=1/F

T= periodo en segundos
F= frecuencia en Hz

Fase

La fase nos indica en que momento comienza a desarrollarse la onda. Para entender el fenómeno podemos usar una onda sinuidal y medirla en grados, 0 es el inicio, 90 el punto más alto 180 la mitad y 270 el punto más bajo.

El fenómeno de la fase se puede oír cuando sumamos dos ondas sinuidales, si una onda comienza en 0 y otra en 180, se encuantran desfasadas 180 grados, esto anula la fuerza de ambas y si tienen la misma amplitud no sonará nada.

En algunos objetos de SuperCollider la fase se mide en radianes. Aquí una tabla de conversión entre grados y radianes:

grados	radianes
0	0
90	pi/2
180	pi
270	3pi/2
360	2pi

```
// la onda empieza en 0 grados. Es decir 0 radianes
{SinOsc.ar(100,0,1)}.plot

// la onda empieza en 90 grados. Es decir pi/2 radianes
{SinOsc.ar(100,pi/2,1)}.plot

// la onda empieza en 180 grados. Es decir pi radianes
{SinOsc.ar(100,pi,1)}.plot

// la onda empieza en 270 grados. Es decir 3pi/2 radianes
{SinOsc.ar(100,3pi/2,1)}.plot

// la onda empieza en 360 grados. Es decir 2pi radianes
{SinOsc.ar(100,2pi,1)}.plot

// si sumamos dos ondas con 180 grados de desfase se cancelan
{SinOsc.ar(100,0,1) + SinOsc.ar(100,pi,1)}.plot
```

Armónicos

Los armónicos son frecuencias múltiplos de una frecuencia fundamental. Podemos nombrarlos como fundamental, armónicos nones y armónicos pares. La fundamental es la primer frecuencia de la serie de armónicos y es la frecuencia que suena con mayor intensidad.

Tomemos como frecuencia fundamental 100 y agreguemos sus primeros 9 armónicos:

```
f * 1 = 100 // fundamental
f * 2 = 200 // segundo armónico
f * 3 = 300 // tercer armónico
f * 4 = 400 // cuarto armónico
f * 5 = 500 // quinto armónico
f * 6 = 600 // sexto armónico
f * 7 = 700 // séptimo armónico
f * 8 = 800 // octavo armónico
f * 9 = 900 // noveno armónico
```

Ejemplo

```
// serie de 9 armónicos
{var arm=9; Mix.fill(arm,{|i| SinOsc.ar((100 * (i+1)).postln, 0,
1/arm)})}.play
```

```
// armónicos pares de la misma serie
{var arm=4; Mix.fill(arm,{|i| SinOsc.ar((100 *
((i+1)*2)).postln, 0, 1/arm)}) + SinOsc.ar(100,0,0.1)}.play

 // armónicos nones de la misma serie
{var arm=9; Mix.fill(arm,{|i| SinOsc.ar((100 * (((i+1)*2)
-1)).postln, 0, 1/arm)})}.play

s.scope
```

Timbre

El timbre es el sonido característico de un instrumento musical y esta determinado por la relación de sus armónicos. Es una cualidad compleja del sonido que nos permite, por ejemplo, distinguir que la misma nota es ejecutada en instrumentos distintos.

Envolvente

El envolvente de un sonido es el desarrollo de la amplitud en el tiempo. El envolvente de uso más generalizado es la envolvente de tipo ADSR. Esta envolvente pasa por los puntos *Attack, Decay, Sustain, Release* (Ataque, Decaimiento, Sostenimiento y Relajamiento).

Cada sonido presenta un envolvente único y no está necesariamente conformado por estos cuatro pasos, por ejemplo un sonido de percusión contiene dos pasos definidos: ataque y decaimiento.

EnvGen.kr y Env

En SC tenemos el objeto `EnvGen.kr` para generar el envolvente que deseamos usar. Env es el objeto que define el envolvente. Los envolventes se pueden dividir en envolventes de duración fija, donde un *gate* funciona como disparo; y envolventes sostenidos donde el envolvente se queda abierto hasta que *gate* vale 0.

Envolventes de duración fija

Env.new ([niveles], [tiempos], curve)
4 niveles, 3 tiempos, etc.. Las curvas pueden ser ‘exponential’, ‘linear’,’sine’, ‘welch’, ‘step’, Estas curvas determinan el tipo de pendiente del envolvente.

```
Env.new([0,1,1,0],[1,1,1],'linear').test.plot
```

Env.linen(at, st, rt, nivel, curve)
Las curvas pueden ser como en el caso del Env.new

```
Env.linen(0.5,1,1,1,'welch').test.plot
```

Env.triangle(duración, nivel)
Es una envolvente en forma de triángulo.

```
Env.triangle(2,1).test.plot
```

Env.sine(duración, nivel)
Es una envolvente en forma circular o senoide.

```
Env.sine(1,1).test.plot
```

Env.perc(at, rt, nivel, curva)
Envolvente percusiva. En este caso la curva se refiere a la curvatura del envolvente, se expresa con números, prueba con varios en valores entre 4 y -4. Los valores positivos te darán una curvatura en reversa.

```
Env.perc(0.5,1,1,-1).test.plot
```

Envolventes sostenidos

Env.adsr (at, dt, sL, rt, pL, curva)
Este es el envolvente con el que normalmente se ejemplifica el fenómeno, notar que la curva puede expresarse en número o en los tags antes mencionados (‘linear’ etc..)

```
Env.adsr(0.2, 0.5, 0.5, 0.5, 1, -4).test.plot
```

Env.dadsr (delayT, at, dt, sl, rt, pL, curva)
Es igual que el anterior pero tiene un tiempo de delay al inicio, que normalmente es de 0.1

```
Env.dadsr(0.1,0.2, 0.5, 0.5, 0.5, 1, -4).test.plot
```

Env.asr (tiempo de ataque, nivel de sostenimiento, tiempo de relajamiento, curva)

```
Env.asr(0.5,1,1,-1).test.plot
```

Env.cutoff (tiempo de relajamiento, nivel, curva)

```
Env.cutoff(0.5,1,1).plot
```

Ejemplo de envolvente de duración abierta con adsr.

```
(
SynthDef(\envolvente, {|gate=1|
     var sig, env;
     sig=Pulse.ar(40);
     env=EnvGen.kr(Env.adsr(0.01,1,0.5,2),gate,doneAction:2);
     Out.ar(0,sig*env);
     }).send(s);
)

~envolvente=Synth(\envolvente)
~envolvente.set(\gate, 0)
```

Es necesario crear un argumento *gate* para poder abrir y cerrar el envolvente. Inicializamos gate = 1 para que el envolvente abra inmediatamente que disparemos el synth. Enviamos el valor 0 al *gate* para cerrar el envolvente.

Se pueden usar envolventes para otras cosas aparte de la amplitud, como las frecuencias:

```
(
SynthDef(\envolvente, {|gate=1, arrayFreq=#[136.91134813639,
202.81123520967, 65.394989722475, 132.95621819661,
62.525500970093, 127.88861670575, 65.360868138958,
70.935648492864, 50.681074552298, 75.824601465534,
98.656687077899, 157.24500577093, 175.45642644699,
105.33968122787, 40.347150207583, 81.445376707312,
32.552185955962, 43.582164642573, 38.873095084923,
116.28469356642 ], arrayTiempos=#[ 1.0105085394288,
1.9728531571695, 1.5922585011823, 1.1866095668553,
2.3625114103724, 2.4987886147012, 0.89903059583291,
0.99887201423248, 2.2075978419634, 1.166383247018,
1.4351805643996, 2.4908415530114, 1.0988777081828,
1.5792846388545, 2.9589556934781, 0.55861861034951,
1.0624474655186, 1.3681685701116, 1.3537558879009 ]|

var sig, env, freq;
freq=EnvGen.kr(Env.new(arrayFreq,arrayTiempos));
sig=Pulse.ar(freq);
env=EnvGen.kr(Env.asr(1,1,1),gate,doneAction:2);
```

```
            Out.ar(0,sig*env);
            }).send(s);
)

~freqs=Array.exprand(20,30,300) + 1000;

~temps=Array.exprand(19,0.5,3);

~envolvente1=Synth(\envolvente, [\arrayFreq,
~freqs,\arrayTiempos, ~temps]);

~envolvente2=Synth(\envolvente, [\arrayFreq,
~freqs.scramble,\arrayTiempos, ~temps.scramble]);

~envolvente3=Synth(\envolvente, [\arrayFreq,
~freqs.scramble,\arrayTiempos, ~temps.scramble]);

~envolvente1.set(\gate, 0);
~envolvente2.set(\gate, 0);
~envolvente3.set(\gate, 0);
```

Percepción del sonido

Este tema es parte de la psicoacústica, ciencia que estudia los fenómenos de percepción del sonido y las sensaciones psicológicas de la audición.

El sonido se puede estudiar en dos campos, uno físico o cuantificable y otro psicológico o cualitativo, el primero basado en medidas exactas del sonido en un medio físico y el segundo en experiencias subjetivas.

Lo primero que veremos es la altura y la sonoridad, dos cualidades subjetivas de percepción sonora que responden a un estímulo físico.

Altura tonal

Es la respuesta al estímulo físico donde percibimos un cambio de tono, determina que un sonido es más agudo o más grave, depende en primer lugar de la frecuencia del estímulo pero también de su intensidad.

Sonoridad

Es la respuesta al estímulo físico donde percibimos un cambio de intensidad, determina que un sonido es más fuerte o más quedo, depende en primer lugar de la intensidad del estímulo pero también de la frecuencia.

Es preciso mencionar que las experiencias subjetivas no tienen una relación directa con los fenómenos físicos sonoros. Mucho tiempo se creyó que la sonoridad era una respuesta equivalente a la amplitud, así como la altura tonal de la frecuencia.

Percepción del espacio

Uno de los atributos tonales es la ubicación percibida de un sonido, o sea de donde parece provenir. El hecho de tener dos oídos nos permite localizar la ubicación de una fuente sonora a partir de la diferencia de tiempo de arribo y la diferencia de intensidad. Además, la parte externa del oído llamada pinna sirve para codificar el sonido que llega de arriba y de abajo y ayuda a diferenciar sonidos que llegan de atrás y adelante.

La técnica de detectar la ubicación del sonido se llama localización binaural. La barrera de la cabeza acentúa la diferencia de intensidad entre los dos oídos.

Diferencia de tiempo de arribo: es un método que usamos al escuchar para determinar de dónde viene el sonido, haciendo una comparación entre el tiempo que tarda en llegar un sonido al oído derecho y al oído izquierdo. Es decir, si un sonido está más cerca del oído derecho, este llegará primero a él.

Diferencia de intensidad: Aunado a lo anterior el oído también es sensible a la diferencia de volumen que capta cada oído para determinar su ubicación, en este caso, si algo suena más cerca del oído derecho, lo escucharemos con mayor fuerza y con menor fuerza en el izquierdo.

Reflexión

El sonido viaja a través del aire y cuando encuentra un obstáculo en su camino se refleja, posteriormente sigue viajando hasta que encuentra otro obstáculo y vuelve a reflejarse, así sucesivamente hasta que pierde fuerza y desaparece.

El sonido se refleja de manera distinta según la textura y forma de cada superficie. Una superficie lisa refleja el sonido igual a su ángulo de incidencia, una superficie cóncava focaliza el sonido a un punto, mientras que una convexa lo esparce.

Absorción

Al momento que un sonido se refleja en una superficie, una proporción es absorbida, esto se manifiesta en una conversión de energía acústica en calor, así, el sondio cada vez que rebota en una superficie va perdiendo fuerza debido a la propiedad de absorción de los materiales.

Cada material tiene un coeficiente de absorción distinto, que es un número que indica la capacidad del material para absorber sonido, este coeficiente se mide en diferentes rangos de frecuencia, es decir, el sonido se absorbe de manera distinta a lo largo de su rango de frecuencia. Así, hay materiales que absorben rápidamente los agudos, mientras que los graves casi no, o hay meteriales que absorben poco sonido mientras que otros absorben mucho.

Reverberación

La reverberación no es otra cosa que múltiples reflexiones del sonido dentro de un espacio cerrado, ya que una de las propiedades del sonido es la de reflejarse en una superficie. Las reflexiones se comportarán de manera distinta dependiendo del material y medidas de cada lugar, el cerebro es capaz de entender esta información lo que nos ayuda a entender en que espacio estamos situados y donde se encuentra ubicada una fuente sonora.

La manera electrónica de simular estas reflexiones es mediante la repetición de fragmentos de sonido que después son sumados a la señal original, logrando una sensación de que el sonido está en un espacio específico. Esto también sucede de manera natural, cuando nosotros grabamos un sonido, por ejemplo, en un espacio grande con paredes muy duras y reflejantes como puede ser una iglesia, las reflexiones son captadas junto con el sonido original y esto es lo que hace que un instrumento o una voz suene diferente en un espacio grande o en uno pequeño; en uno abierto o en uno cerrado.

Las reflexiones del sonido también nos dan información de los materiales del lugar donde está sonando la fuente sonora, por ejemplo un cuarto de madera suena distinto a un cuarto de piedra.

El sonido que sale de una fuente sonora sin tener contacto con alguna superficie es el sonido directo, los primeros rebotes dentro de un espacio se llaman reflexiones tempranas, estas reflexiones se suman a otras hasta tener un sonido denso en reflexiones el cual desaparecerá en un tiempo determinado, a esta densidad de reflexiones sumadas se le llama reverberación y al tiempo que tarda en desaparecer se le llama tiempo de reverberación.

En SC existen tres UGens para simular Reverberación: FreeVerb, FreeVerb2, GVerb.

FreeVerb.ar (in, mix, room, damp, mul, add)

in: señal de entrada
mix: balance dry/wet, rango 0..1

room size: tamaño del cuarto, rango 0..1
damp: atenuación de las frecuencias agudas, rango 0..1

A continuación un ejemplo usando un Pulse, intenta modificar los ultimos tres argumentos (los últimos tres 1), en valores que estn ente 0 y 1 para que escuche la diferencia.

```
// muy reverberante
{FreeVerb.ar(Pulse.ar(500,0.5)*Line.kr(1,0,0.5), 1, 1, 1)}.play

// poco reverberante
{FreeVerb.ar(Pulse.ar(500,0.5)*Line.kr(1,0,0.5), 0.8, 0.5,
0.5)}.play
```

FreeVerb2.ar (in1, in2, mix, room, damp, mul, add)
Es un Reverb que admite dos señales, el funcionamiento es igual que FreeVerb.

```
// escucha la señal que sale por izaquierda y derecha
{FreeVerb2.ar(LFNoise0.ar(0.5),LFNoise0.ar(1), 0.9, 0.9,
0.8)}.play
```

Gverb (in, roomsize, revtime, damping, inputbw, spread, drylevel, earlyreflevel, taillevel, maxroomsize, mul, add)

Este reverb es más complejo, procesa la señal en estéreo. Esta basado en el efecto LADPSA GVerb.

in: señal de entrada
roomsize: tamaño del cuarto en metros cuadrados
revtime: tiempo de reverberación en segundos
damping: corte de frecuencias agudas, entre 0 y 1, 0 atenua los agudos por completo, 1 no tanto
inputbw: atenua la señal como **damping** pero actua en todo el rango de frecuencia de la señal
spread: controla el esparcimiento de la señal estéreo y la difusión del reverb
drylevel: cantidad de señal seca
earlyreflevel: cantidad de refelxiones tempranas
taillevel: cantidad del nivel de la cola
maxroomsize: pone el tamaño de las lineas de delay. Normalmente +1

Ejemplo dentro de un SynthDef

```
(
SynthDef(\gverb,{|roomsize=3, revtime=2, damping=0.5,
inputbw=0.1, spread=15, drylevel= -11, earlyreflevel= -6,
taillevel= -17|
var in, rev;

in=Dust.ar(2); rev=GVerb.ar(in,
roomsize,
revtime,
damping,
inputbw,
spread,
drylevel,
earlyreflevel,
taillevel,mul:0.4);

Out.ar(0,rev+in)
}).send(s)
)

a=Synth(\gverb)
a.set(\roomsize,2,\revtime,3.5,\damping,0.2,\inputbw,0.6,\spread
,12,\drylevel,-6,\earlyreflevel,-3,\taillevel,-17)
a.free
```

Biblografía

Coren, Stanley (1999). *Sensación y Percepción*. México D.F.: McGraw-Hill.

Cohen, Jozef (1969). *Sensación y percepción auditiva y de los sentidos menores*. México D.F. : Trillas

Everest, F. Alton (2001). *The Master Handbook of Acoustics*. New York: McGraw-Hill.

Música por computadora

Ernesto Romero y Hernani Villaseñor
Centro Multimedia 2012

Sesión 10

Conceptos musicales

Hasta ahora hemos revisado conceptos de la teoría del sonido en general. En esta sesión estudiaremos los principales conceptos de la teoría de la música. La sesión no pretende constituir un tratado de música. El propósito de esta sesión es que se obtenga un lenguaje a ravés del cual podamos expresar lo que estamos haciendo y ligarlo a los conceptos de teoría de la música por computadora que veremos más adelante.

Altura

La altura se refiere a la frecuencia de un sonido. En lenguaje musical nos referimos a las notas. Las notas corresponden a frecuencias específicas. Existen 12 notas en la música tradicional occidental:

Nombre de la nota	Sistema inglés	Frecuencia	MIDI
Do	C	261.625	60
Do sostenido	C#	277.182	61
Re	D	293.664	62
Mi bemol	Eb	311.126	63
Mi	E	329.627	64
Fa	F	349.228	65
Fa sostenido	F#	369.994	66
Sol	G	391.995	67
La bemol	Ab	415.304	68
La	A	440	69
Si bemol	Bb	466.163	70
Si	B	523.251	71

Entre cada nota consecutiva se dice que hay ½ tono. Cada una de estas notas se pueden tocar más graves (multiplicándolas por ½) o más agudas (multiplicándolas por 2).

Dentro de este conjunto de notas se pueden hacer diferentes escalas. Las escalas son un conjunto de notas específicas. Este conjunto de notas definen una tonalidad. A partir de ahora usaremos el sistema inglés para referirnos a las notas.

La escala mayor de C es:

C D E F G A B C

Si analizamos la distancia en ½ tonos que hay entre cada nota obtenemos la siguiente estructura

1 1 ½ 1 1 1 ½

Esto es: Entre C y D hay 1 tono, entre D y E un tono, entre E y F ½ tono, etc.

Esta escala tiene 7 notas. Si se agrega una nota más volvemos a tener C pero se dice que esta una octava arriba.

La tabla 1 tiene las notas dentro de la octava central del piano y se le conoce como la octava índice 5. Las mismas notas pero una octava arriba estarán en el indice 6. Los índices de las notas cambian a partir de cada C.

La escala menor armónica de C es:

C D Mb F G Ab Bb C

sus intervalos son

1 ½ 1 1 ½ 1 1

Si usamos la raíz doceava de dos podemos obtener un medio tono a partir de cualquier frecuencia. La raíz doceava de 2 en SuperCollider se escribe así:

```
2**(1/12)
```

C es 261.625. Si queremos obtener ½ tono arriba de este C, es decir C# lo hacemos así:

```
261.625 * 2**(1/12)
```

Y para obtener D lo multiplicamos otra vez:

```
261.625 * 2**(1/12) * 2**(1/12)
```

Es más práctico elevar al cuadrado 2**(1/12) que multiplicarlo dos veces:

261.625 * ((2**(1/12))**2)

La potencia a la que estamos elevando 2**(1/12) indica el número de medios tonos que obtendremos arriba de una nota.

Ejemplo:

```
261.625 * ((2**(1/12))**0) // cero medios tonos sobre C = C
261.625 * ((2**(1/12))**2) //  dos medios tonos sobre C = D
261.625 * ((2**(1/12))**4) //  cuatro medios tonos sobre C = E
261.625 * ((2**(1/12))**5) //  cinco medios tonos sobre C = F
261.625 * ((2**(1/12))**7) //  siete medios tonos sobre C = G
261.625 * ((2**(1/12))**9) //  nueve medios tonos sobre C = A
261.625 * ((2**(1/12))**11) //  once medios tonos sobre C = B
261.625 * ((2**(1/12))**12) //  doce tonos sobre C = C una octava arriba
```

Si escribimos en orden las potencias obtenemos esta serie:

0, 2, 4, 5, 7, 9, 11, 12

SuperCollider tiene el método .midiratio que simplifica el uso de (2**(1/12)

Así la tabla anterior se puede escribir de la siguiente forma:

```
261.625 * 0.midiratio  // cero medios tonos sobre C = C
261.625 * 2.midiratio  // dos medios tonos sobre C = D
261.625 * 4.midiratio  // cuatro medios tonos sobre C = E
261.625 * 5.midiratio  // cinco medios tonos sobre C = F
261.625 * 7.midiratio  // siete medios tonos sobre C = G
261.625 * 9.midiratio  // nueve medios tonos sobre C = A
261.625 * 11.midiratio  // once medios tonos sobre C = B
261.625 * 12.midiratio  // doce medios tonos sobre C = C una octava arriba
```

También existe el método midicps que convierte el código de notas midi a frecuencias. El código de notas midi con su correspondiente nota se encuentran en la tabla 1. El código se extiende por debajo del 60 y por arriba del 71 alcanzando las octavas inferiores y superiores respectivamente

60.midicps=261.625

Podemos hacer sonar una escala en SuperCollider así:

```
p=ProxySpace.push(s)
// mayor
~escala={Demand.kr(Impulse.kr(2),0,Dseq([0,2,4,5,7,9,11,12].midiratio,inf))}
// menorArmonica
~escala={Demand.kr(Impulse.kr(2),0,Dseq([0,2,3,5,7,8,10,12].midiratio,inf))}
// menor Melodica
~escala={Demand.kr(Impulse.kr(2),0,Dseq([0,2,3,5,7,8,11,12].midiratio,inf))}

~sonido={SinOsc.ar(~escala.kr*60.midicps)}
~sonido.play
~sonido.stop
```

Dinámica

La dinámica se refiere a la intesnsidad con la que se emite un sonido. Coloquialmente se utilizan palabras como “quedito”, “duro” o “fuerte” para hablar de esto. En la terminologa musical clasica los treminos que se usan no son muy diferentes, simplemente estan en italiano. Para hablar de un sonido suave se usa la palabra “piano” que en español literalmente significa “plano” y cuyo simbolo es la letra *p*. Para los sonidos fuertes se usa la palabra”forte” que significa lo mismo en españoly se usa la letra *f* para representarla. El instrumento de teclas negras y blancas que conocemos como piano originalmente se llamo *pianoforte* ya que podia tocar notas suaves y fuertes a diferencia del clavecin que solo podia tocar notas con una misma intensidad. Entre estas dos dinamicas, *p* y *f*, existe la subjetividad. Lo normal es que para un sonido muy suave se use *pp* (pianisimo), que es mas suave que *p*, y para sonidos muy fuertes se use *ff* (fortisimo). Asi mismo, para un sonido ni suave ni fuerte se usa *mf* (mezzoforte) que es medio fuerte. Existen compositores que incluso llegan a usar *ppp* y *fff.*

En SuperCollider la intensidad de los sonidos est normalizada. Es decir, 1 es el máximo y 0 el mínimo. El valor de la intensidad del sonido se le llama amplitud, ya que es la amplitud de una onda la que implica la furza que lleva, y mientras mayor la fuerza, mayor su intensidad. En los UGens el argumento mul es el que se encarga de asignarle valor a la amoplitud de las ondas y por default se le asigna el valor de 1: la maxima intensidad antes de que comience a distorsionarse. Si queremos hacer una correspondecia entre valores de amplitud y dinámicas podriamos hacerlo de la siguiente forma:

```
{SinOsc.ar(440,0,mul:1)}.play          // ff
{SinOsc.ar(440,0,mul:0.75)}.play       // f
{SinOsc.ar(440,0,mul:0.5)}.play        // mf
{SinOsc.ar(440,0,mul:0.25)}.play       // p
{SinOsc.ar(440,0,mul:0.125)}.play      // pp
```

El uso de un valor normalizado para la amplitud a veces no refleja la experiencia del oido humano. Para mejorar esta respuesta se usan los decibeles que son una medida del volumen, siendo este ya una caracterstica fisica del movimiento de particulas de aire en el espacio. 0

decibeles, o 0 *db* es el mximo de volumen. El oido humano percibe diferencias de volumen de 3 *db,* asi que si vamos bajando el volumen de un sonido obtenemos la siguiente regla:

0 *db*
-3 *db*
-6 *db*
-9 *db*
-12 *db*
-15 *db*
-18 *db*

El descenso de decibeles puede llegar al infinito, pero estando en -60 *db* practicamente ya no se escucha nada. En Supercollider se puede hacer uso de los decibeles con el mensaje dbamp que convierte decibeles a amplitud normalilzada

```
{SinOsc.ar(440,0,mul:0.dbamp)}.play          // ff
{SinOsc.ar(440,0,mul:-10.dbamp)}.play        // f
{SinOsc.ar(440,0,mul:-20.dbamp)}.play        // mf
{SinOsc.ar(440,0,mul:-35.dbamp)}.play        // p
{SinOsc.ar(440,0,mul:-50.dbamp)}.play        // pp
```

La dinamica tambien incluye cambios progresivos en la intensidad del sonido. Para un sonido cuya intensidad vaya creciendo se usa el termino *crescendo* y se usa la contraccion *cresc*. Para un sonido cuya intensidad vaya decreciendo se usa el termino *decrescendo o* diminuendo y se usan la contracciones de*cr* o *dim*. Tambien se usan los simbolos conocidos como reguladores < y > para el *cresc* y el *dim* respectivamente.

```
{WhiteNoise.ar(Line.kr(0,1,5,1,0,2)}.play      //  crescendo
{WhiteNoise.ar(Line.kr(1,0,5,1,0,2)}.play      //  diminuendo
```

Traten de escuchar si hay una diferencia entre los reguladores con amplitud normalizada y los que usan decibeles

```
{WhiteNoise.ar(Line.kr(-60.dbamp, 0.dbamp,5,1,0,2))}.play     //  crescendo
{WhiteNoise.ar(Line.kr(0.dbamp, -60.dbamp,5,1,0,2))}.plot     //  diminuendo
```

Densidad

La densidad se puede entender como la cantidad de eventos sonoros que suceden en un determinado tiempo.

Poca densidad:

```
{Dust.ar(1)}.play  //  poca densidad
```

{Dust.ar(20)}.play // mayor densidad

La densidad tambien se puede observar en el espectro de un sonido

{Dust.ar(20)}.play // poca densidad espectral
{Decay2.ar(Dust.ar(20),0.01,1,BrownNoise.ar)}.play // mayor densidad espectral

O en el espectro armonico

{Mix((Pulse.ar(4)*SinOsc.ar(300*[0,7].midiratio,0,0.1)))}.play // poca densidad armonica
{Mix((Pulse.ar(4)*SinOsc.ar(300*[0,4,7,11,14,17,21].midiratio,0,0.1)))}.play // mayor densidad armonica

Textura
La textura son las diferentes combinaciones que se pueden hacer de los elementos mencionados anteriormente. Al hablar de textura se suele referir a elementos fisicos, como cuando se habla de la textura de un muro liso o rasposo o con formas que emergen o con estrias... el elemento subjetivo aparece con mayor fuerza que en ningn otro de los parametros que hemos descrito anteriormente. La densidad es parte importante a la hora de describir una textura. Se habla de texturas densas o texturas poco densas. La duracion de los eventos sonoros pueden dar origen tambien a la percepcion de diferencias en la textura. El contraste asi mismo puede definir la textura de un gesto sonoro. En la musica instrumental academica la textura es a veces vista como la cantidad de instrumentos que tocan en una pieza.

Contraste

El contraste es uno de lo elementos sonoros mas importantes en cuanto a la claridad de las ideas se refiere. Hay que tener conciencia del contraste para poder haxcer una pieza que contenga cierto balance. Para obtener un minimo acercamiento a la idea musical o sonora que se desea es imprescindible tener siempre en cuante el contraste. El contraste se puede encontrar o provocar en las alturas, en las dinamicas, en las densidades, en las texturas, en las formas... en todo lo hay. Tambien es importante (igual de importante) tener claridad en cuanto a la no existencia del contraste, en caso de que sea deseado, o en caso de que sea un resultado secundario...

Espacialización (estereofonía y multicanal)

Como vimos, el cerebro humano es capaz de formar mapas de la localización a partir del tiempo y fuerza con que llega el sonido a ambos oídos.

El sistema estéreo

Consta de dos canales, uno izquierdo y uno derecho. Funciona con dos bocinas que emiten información diferente de la misma fuente sonora, si alguien se sienta en medio de dos bocinas

podrá percibir una imagen fantasma del sonido que sale justo en medio si el sonido es reproducido con la misma fuerza en ambas bocinas. Si se incrementa el volumen en una de ellas, por ejemplo la bocina derecha, percibimos que la imagen sonora se mueve a esa dirección. Esto sucede por la capacidad que tenemos de percibir la diferencia de intensidad.

Para colocar el sonido en un lugar determinado entre las dos bocinas usamos la el objeto Pan2.ar o panorama estéreo.

Pan2.ar(señal, panorama, amplitud)

```
//Sonido al centro
{Pan2.ar(Pulse.ar(4,0.5,0.5), 0, 1)}.play

//Sonido a la izquierda
{Pan2.ar(Pulse.ar(4,0.5,0.5), -1, 1)}.play

//Sonido a la derecha
{Pan2.ar(Pulse.ar(4,0.5,0.5), 1, 1)}.play

//Sonido moviendose de derecha a izaquierda con Line.kr
{Pan2.ar(Pulse.ar(4,0.5,0.5), Line.kr(1,-1,5,doneAction:2),
1)}.play
```

Sonido Multicanal

SC es capaz de gestionar varios canales. Con el objeto Out.ar es como direccionamos el sonido hacia el canal deseado.

A la práctica de colocar sonido en varias bocinas se le llama espacialización. La espacialización en SC la podemos gestionar a partir de varios objetos:

Pan2.ar // forma un panorama estéreo
Pan4.ar // forma una cuadrifonía
PanAZ.ar // realiza una distribución del sonido en un horizonte circula o azimhut
PanB2.ar // señal Ambisonics en formato B

Más acerca de gestión de múltiples bocinas:

BEASTmulch: software basado en SuperCollider que controla el sistema BEAST
Birmingham ElectroAcoustic Sound Theather

http://www.birmingham.ac.uk/facilities/BEAST/index.aspx

Música por computadora

Ernesto Romero y Hernani Villaseñor
Centro Multimedia 201

Sesión 11

Conceptos de Audio Digital

El proceso de conversión análogico-digital es aquel en el cual una señal eléctrica es convertida en números binarios digitales (bits) para ser procesada por un aparato digital.

Conversión ADC - DCA

Para convertir la señal analógica en digital se utiliza un convertidor llamado ADC, *Analog to Digital Converter*. Este convertidor trabaja de manera muy precisa tomando muestras del sonido cada segundo, por lo que utiliza un reloj que comanda cuantas veces por segundo trabaja el ADC.

Asimismo, existe el proceso inverso, es decir, la conversión de datos digitales en analógicos o reconstrucción de una onda que está representada en bits (0 y 1) en una señal eléctrica continua. El componente que revierte el proceso es un convertidor DAC, *Digital to Analog Converter.*

Teorema Nyquist

Nos dice que para poder tener una captura adecuada del rango de frecuencia de una señal, la frecuencia de muestra debe ser al menos el doble de la frecuencia más aguda que queremos digitalizar.

frecuencia mínima de muestra = 2 x frecuencia más alta a digitalizar

La frecuencia de muestra (*sampling frequency*) se expresa mediante la siguiente fórmula:

fs/2

Entonces si la frecuencia más aguda que escuchamos es 20,000 Hz, la frecuencia de muestra debe ser al menos de: 20,000 x 2, o sea, 40,000 Hz. Esto quiere decir que el reloj digital ADC debe realizar 40,000 muestras en un segundo, así nos aseguramos que al menos el pico positivo y el pico negativo de una onda son digitalizados.

Frecuencia de Muestra (*Sample Frequency*)

Es el número de muestras tomadas en 1 segundo a una señal análoga de sonido. Se mide en Hz y determina la velocidad con la que trabaja el convertidor análogo digital, normalmente es de 44.1 kHz para sonido en CD y de 48 kHz para video.

Aliasing

Es importante que señales mayores a la mitad de la frecuencia de muestra no entren en el proceso de conversión de la señal, si esto sucede, frecuencias erróneas conocidas como frecuencias alias, pueden irrumpir en la señal audible como falsas frecuencias escuchandose una distorsión armónica.

En SC se producen frecuencias Alias en los agudos al usar los UGens LF, como LFSaw. Podemos hacer la comparacaión gráfica como sugiere Collins:

```
(
{
[LFSaw.ar(1000),Saw.ar(1000)]
}.plot(0.01)
)
```

Resolución o cuantización (*Bit Rate*)

Representa el componente de amplitud del proceso de digitalización e indica cuántos pasos hay disponibles para codificarla. Es la traducción de los niveles de voltaje de una señal análoga continua en números binarios. Esta información es guardada en el dominio digital para posteriormente ser procesada o simplemente almacenada. Se mide en bits.

Normalmente la resolución de calidad CD es de 16 bits, lo cual quiere decir que tenemos 65,536 pasos para representar la forma de onda en su amplitud.

El resultado se expresa en una palabra de n bits, donde palabras digitales más largas se traducen en un incremento de resolución, ya que crece el número de pasos en que la señal puede ser codificada de manera digital.

8 bit = 256
16 bit = 65,536
20 bit = 1,048,576
24 bit = 16,777,216
32 bit = 4,294,967,296

Proceso de conversión de audio análogo/digital:

Entrada de audio - Dither - Filtro Anti Alias - Sample & Hold - A/D - Multiplexor - Corrección de errores - Modulación - Salida digital a medio de almacenamiento.

La señal de audio entra por un amplificador de línea en forma de voltaje, que en electricidad representa a la señal de audio y que es una señal continua.

Dither: el siguiente paso es agregar dither a la señal de audio, el cual es un ruido aleatorio usado para reducir la probabilidad de distorsión armónica en los bits menos significativos (LSB) que son cuantizados en 0.
Filtro Anti-Alias: es un filtro pasa bajos que corta las señales más agudas de la frecuencia de muestra y evita el fenómeno de Aliasing. Debido a la pendiente de este filtro la frecuencia de muestra se debe escoger un poco más alta que la frecuencia más aguda a digitalizar. Como estándar se ha escogido la frecuencia de 22,050 Hz que al multiplicarla por 2 resulta la frecuencia de muestra usada en los CD: 44,100 Hz o 44.1 kHz.

Muestra/retención (sample & hold): Este componente es el encargado de tomar las muestras de la señal análoga y lo hace basado en la frecuencia de muestra. Si es de 44.1 kHz, entonces este componente realiza 44,100 muestras en un segundo. Como trabaja es: toma la muestra de una instante de voltaje, la retiene y la envía al siguiente componente que es el ADC.

ADC: Este dispositivo es el encargado de convertir las muestras de voltaje tomadas por el s&h en valores digitales, o sea en números binarios. Si la cuantización es de 16 bit y la frecuencia de muestra es de 44.1 kHz, entonces el ADC producirá 44100 palabras de 16 bits en 1 segundo:

0101 0011 0110 1001
0001 0011 0110 1011
0100 0011 0110 1101
1101 0010 0110 1011
0111 0011 0111 1111
1000 1000 1000 1000
1100 1000 1110 0011
1110 1111 0001 1010
0110 1000 1110 0011
...

Este proceso aún no se tiene una precisión de los datos, por lo que son mandados a un multiplexor que manda la señal a un corrector de datos para posteriormente ser modulados en una onda PCM (tipo de onda cuadrada). Los datos en realidad no se almacenan como 0 y 1, se modulan en una onda cuadrada para poder ser almacenados en un disco duro, cd, usb, flash card etc.

Proceso de conversión de audio digital/análogo:

Es inverso al proceso de conversión análoga a digital.

Entrada digital del medio de almacenamiento - Modulación - Corrección de errores - Multiplexor - DAC - Sample & Hold - Filtro pasa bajos - Salida de audio

Formatos de audio

Aiff: *Audio Interchange File Format*, es el archivo de audio que soportan muchos programas diseñados para Mac.

Wav: *Windows Audio File*, es el archivo de audio que soportan muchos programas para windows y algunos de Mac.

Formatos de audio con compresión:

Se comprime la información para ocupar menor espacio y tener mayor velocidad de transmisión. Hay dos tipos de compresión, la destructiva y la no destructiva.

Mp3: formato de compresión de la MPEG (*Moving Pictures Expert Group*), es propietario.

Ogg: formato de compresión de uso libre.

Música por computadora

Ernesto Romero y Hernani Villaseñor
Centro Multimedia, México 2012

Sesión 12

Síntesis

Generadores

Los generadores son los encargados de producir sonido, en SC son llamados UGens o unidades generadoras de sonido. Podríamos dividirlos en dos grupos: Osciladores y Ruidos.

Osciladores

Los osciladores son objetos que generan una oscilación, es decir, la onda generada debe de hacer un recorrido pasando por los mismos puntos cada vez. También se les llama ondas periódicas. Un péndulo que oscila entre el punto *a* y el punto *b* es un ejemplo clásico de un oscilador.

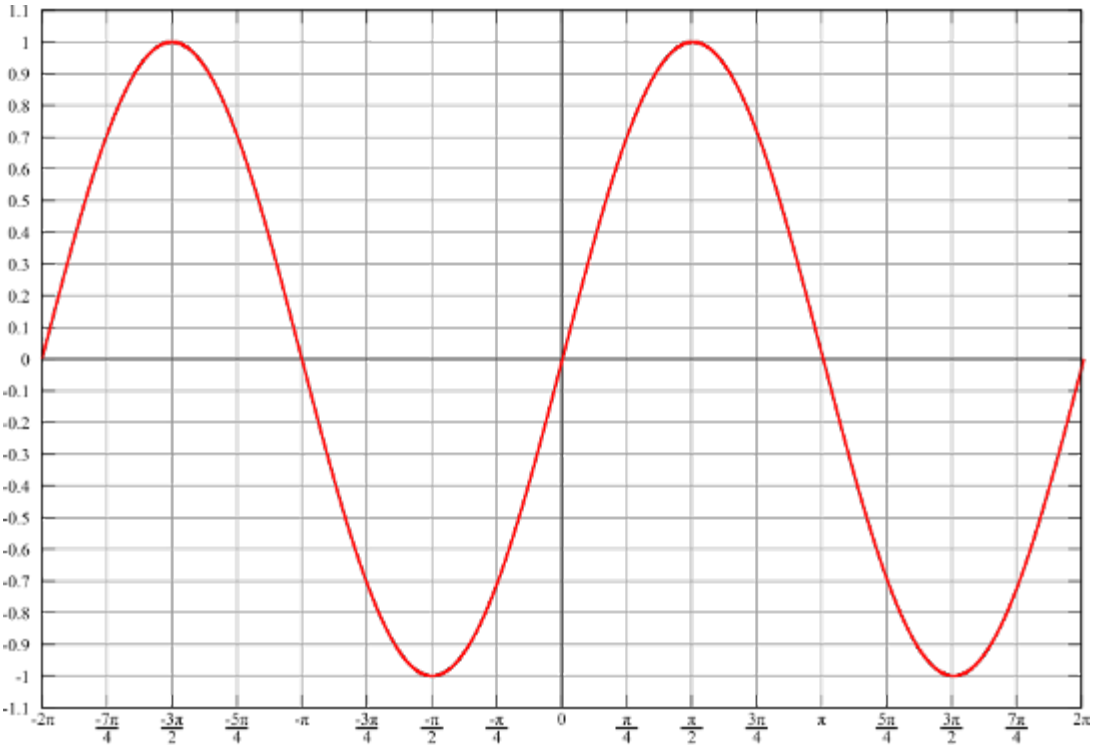
Formas de onda mediante voltaje: los primeros osciladores podían generar formas básicas como sinoidal, triangular o cuadrada, estos osciladores posteriormente fueron incorporados a los sintetizadores mediante un dispositivo llamado VCO (oscilador controlado por voltaje), con el tiempo dejaron de ser operados por voltaje y comenzaron a ser producidos de manera digital.

SC lo que hace es mandar al procesador una serie de funciónes matemáticas que le ordenan generar sonido. Así que parte de la programación de SC consiste en una serie de algoritmos diseñados para producir diferentes tipos de sonido electrónico mediante sus UGens.

Los UGens en SC se dividen en: fuentes: periódico y aperiódico, filtros, distorsión, paneo, reverbs, delays y ugens de buffer, síntesis granular, control: envolventes, triggers, contadores, compuertas, lapso, *decays*, espectrales.

Los osciladores en SC son considerados como fuentes periódicas de sonido.
Entre los osciladores más comunes están los de onda sinusoidal, cuadrada, triangular y de sierra.

Onda Sinusoidal
Aplica la función seno para generar una onda curva cuya amplitud oscila entre 1 y -1. Las fases de la onda se expresan en radianes y tienen un rango entre 0 y 2pi. La fase es un punto de la onda en particular.



Jean Baptiste Fourier demostró que cualquier onda periódica se puede formar a partir de la suma de una o más ondas sinusoidales[1].

La onda sinusoidal no produce armónicos. Es una onda con un timbre simple. Este tipo de ondas no se encuentran en la naturaleza ya que los sonidos naturales siempre vienen acompañados de armónicos. La carencia de armónicos de las ondas sinusoidales hacen que no podamos oírlas en frecuencias menores de 20 Hz (Aunque en la práctica la frecuencia más baja audible es de alrededor de 38 Hz).

En SC generamos una onda sinusoidal con el objeto SinOsc y sus argumentos son los siguientes:

SinOsc.ar(frecuencia, fase, multiplicación, adición)

frecuencia - Todos los osciladores tienen como primer argumento la frecuencia. De igual modo el SinOsc. Se expresa en hertz y determina los ciclos por segundo que realizará la onda. Rango audible entre 20 y 20,000 hz (teóricamente).

fase - Se expresa en radianes (0, 2pi) y determina el punto en el que iniciará la onda.

[1] Computer Music - Synthesis, Composition and Performance. Charles Dodge, Thomas A. Jerse. Pag. 47

multiplicación - Un número que multiplica a la amplitud de la sinoidal. Normalmente la amplitud va de 1 a -1. Si , por ejemplo ponemos como argumento del mul el número cinco nuestra amplitud resultante será de -5 a 5.

adición - Un número que se suma a la amplitud de la sinoidal. Normalmente la amplitud va de 1 a -1. Si , por ejemplo ponemos como argumento del add el número cinco nuestra amplitud resultante será de 4 a 6.

Todos los UGens tienen como últimos argumentos mul y add. Para saber manejar bien estos argumentos podemos recurrir a la siguiente técnica.

Ponemos entre braquets el rango inicial de la amplitud de nuestro UGen, lo multiplicamos por un número que será nuestro mul y le sumamos otro que será nuestro add. Declaramos la línea y el arreglo que obtenemos será la amplitud resultante.

Ejemplo

[-1, 1] * 5 + 5

La amplitud resultante es de 0 a 10. Podemos decir que mul estira la amplitud y add la traslada en el eje Y. Por eso al multiplicar [-1, 1] * 5 estamos estirando la amplitud a [-5, 5]. Al sumarle 5 estamos trasladando la amplitud hhacia arriba 5 unidades obteniendo [0, 10].

Onda Cuadrada

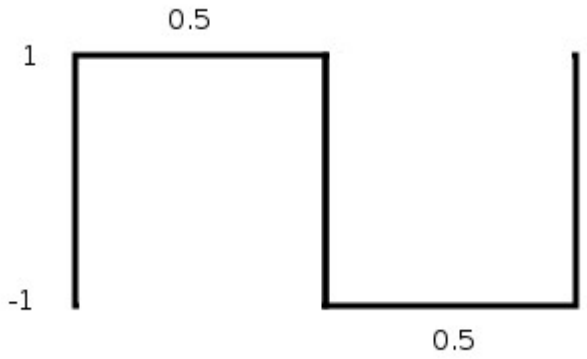
La onda cuadrada brinca de 1 a -1 en cada ciclo sin pasar por los números intermedios. La onda cuadrada arroja solo armónicos impares. Por ejemplo, si tenemos una onda cuadrada a 440 Hz estará compuesta por las siguientes frecuencias:

440 * 1 = 440
440 * 3 = 1320
440 * 5 = 2200
440 * 7 = 3080
440 * 9 = 3960
440 * 11 = 4840
.
.
.
etc

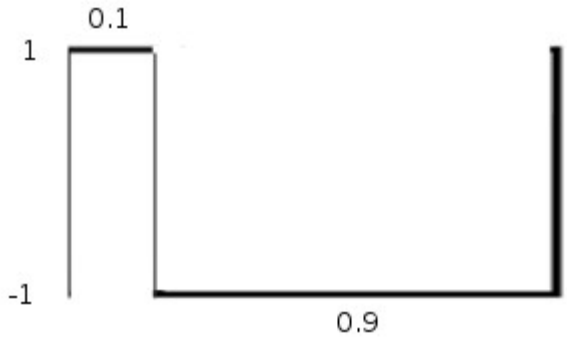
Mientras más definido sea el ángulo recto de la onda mas armónicos tendrá.

El hecho de que la onda cuadrada tenga armónicos nos permite que las escuchemos por debajo de los 20 hz. Atención: no estamos escuchando 20 Hz, sino sus armónicos.

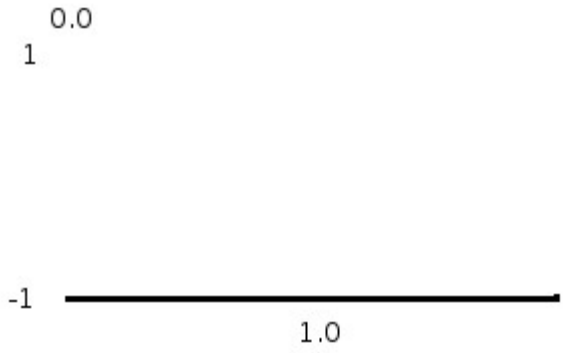
El tiempo que la onda está en 1 o en -1 en cada ciclo se le llama ancho del pulso y determina cambios en el timbre que percibimos. El ancho total de la onda es de 1 y representa la suma del tiempo que está en 1 más el que esta en -1. Por defecto SuperCollider establece el ancho de la onda en 0.5 se refiere a la parte de la onda que está en en -1. Esto hace que la parte de la onda que está en 1 quede en 0.5 también teniendo una proporción de 0.5/0.5.



Ancho de pulso = 0.5



Ancho de pulso = 0.9



Ancho de pulso = 1.0

El UGen que SuperCollider emplea para hacer ondas cuadradas es Pulse.ar y esta es su sintaxis.

Pulse.ar(frecuencia, ancho de pulso, multiplicación, adición)

frecuencia - Expresada en hertz. Podemos usar valores menores de 20hz.

ancho de pulso - rango entre 0 y 1. Si ponemos el ancho en 1 la parte positiva quedara en 0 por lo que no habrá oscilación y no sonara nada. Lo mismo ocurre si ponemos 0 en ancho de pulso. El oído aprecia igual los valores de ancho de forma simétrica a partir del 0.5, de esta forma oiremos igual un ancho de 0.25 que de 0.75.

multiplicación - Un numero que multiplica a la amplitud de la onda.

adición - Un numero que se suma a la amplitud de la onda y la traslada en el eje de las y.

Podemos escuchar las variaciones tímbricas que se obtienen al cambiar el ancho de banda de un Pulse insertando un MouseX en el argumento correspondiente.

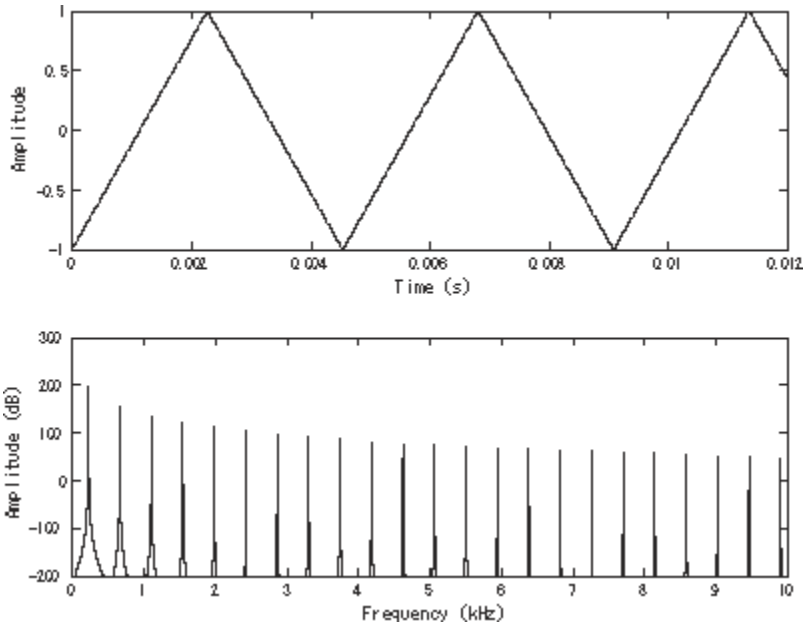
```
{Pulse.ar(440, MouseX.kr(0,1).poll)}.scope
```

Observen como el timbre es el mismo cuando estamos de un lado o del otro del 0.5.
El mensaje .poll nos ayuda a imprimir en la post window los valores de un UGen, en este caso el MouseX

Los Ugens en SC son aproximadamente 250, utiliza la ayuda para estudiar los UGens de tu interés.

Onda Triangular

La onda triangular presenta armónicos impares también pero decrece su amplitud con el inverso del cuadrado del numero armónico correspondiente. así la amplitud del armónico 5 será de 1/25. Es por esto que su espectro es mucho mas simple que el de la onda cuadrada.



En SUperCollider la onda triangular se genera con el objeto LFTri y esta es su sintaxis:

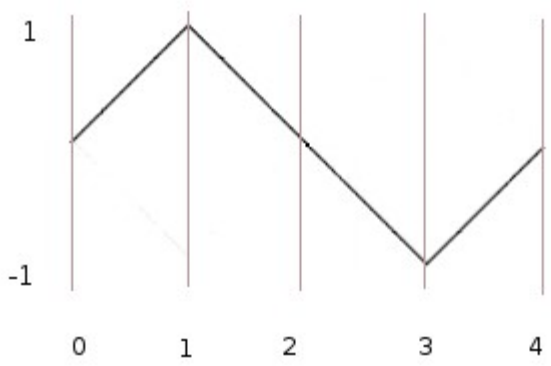
LFTri.ar(frecuencia, fase, multiplicación, adición)

frecuencia - Expresada en hertz. Podemos usar valores menores de 20hz.

fase - El punto en el que iniciara la onda. Los valores pueden ser entre 0 y 4.

multiplicación - Un numero que multiplica a la amplitud de la onda.

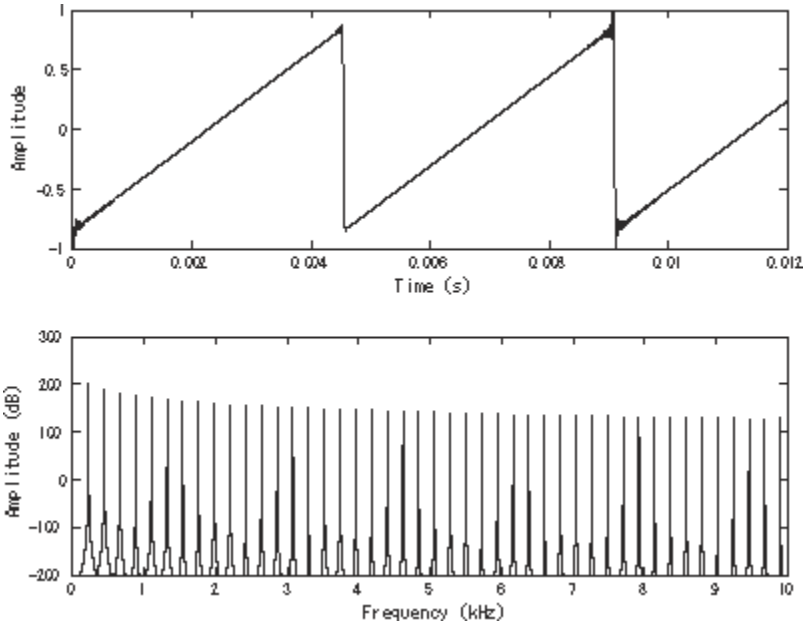
adición - Un numero que se suma a la amplitud de la onda y la traslada en el eje de las y.



fases de la onda triangular

Onda de Sierra

La onda de sierra contiene en su espectro a todos los armónicos. Su forma es parecida a los dientes de una sierra y por eso se le llama así.
Por tener esta riqueza tímbrica la onda de sierra es usada a veces para hacer sintesis sustractiva y obtener sonidos como el de las cuerdas.



SuperCollider utiliza la clase Saw para generar esta onda. Esta es su sintaxis:

Saw.ar(frecuencia, multiplicación, adición)

frecuencia - Expresada en hertz. Podemos usar valores menores de 20hz.

multiplicación - Un numero que multiplica a la amplitud de la onda.

adición - Un numero que se suma a la amplitud de la onda y la traslada en el eje de las y.

Notar que SuperCollider no posee un argumento para la fase de la onda de sierra

Ruidos

Los ruidos generan amplitudes de manera no oscilatoria. Son muy usados para hacer mediciones o como fuente primaria para cierto tipo de síntesis, en especial la síntesis sustractiva. Algunos ruidos se forman al generar todas las frecuencias al mismo tiempo. La diferencia de fuerzas entre las frecuencias nos da diferentes tipos de ruido. Uno de los ruidos más comunes es el ruido blanco o *White Noise* y no es otra cosa que todas la frecuencias sonando al mismo tiempo con la misma fuerza. El ruido rosa o *Pink Noise* disminuye la fuerza de sus frecuencias 3 decibeles por octava. El ruido café o *Brown Noise* lo hace 6 decibeles por octava.

Algunos de los UGens que tiene SC para generar estos ruidos son los siguientes. Observar que sus argumentos son solo de amplitud ya que no existen otros parámetros.

```
{WhiteNoise.ar(1)}.scope;
{PinkNoise.ar(1)}.scope;
{BrownNoise.ar(1)}.scope
```

Podemos observar el espectrograma de los UGens con la siguiente linea:

```
FreqScope.new
```

Ruidos aleatorios

Otros formas de hacer ruido es generando amplitudes de manera aleatoria. Algunas clases de SC que hacen esto son las de tipo LFNoise las cuales generan amplitudes aleatorias entre -1 y 1 con cierta frecuencia. Los cambios entre las amplitudes pueden darse de brinco, es decir de manera discreta, (LFNoise0), con una interpolación lineal (LFNoise1), o con interpolación cuadratica curva (LFNoise2).

```
{LFNoise0.ar}.plot // ruido a escalón, genera cambios discretos
{LFNoise1.ar}.plot// ruido en rampa, genera cambios continuos lineales
{LFNoise2ar}.plot// ruido cuadrático, genera cambios continuos curvos
```

La sintaxis de estos UGens es igual para todos, Veamos la de LFNoise0

LFNoise0.ar(frecuencia, multiplicación, adición)

frecuencia - La cantidad de amplitudes aleatorias por segundo entre 1 y -1. Expresada en hertz. Podemos usar valores menores de 20hz. No es una frecuencia de oscilación

multiplicación - Un numero que multiplica a la amplitud de la onda.

adición - Un numero que se suma a la amplitud de la onda y la traslada en el eje de las y.

Otro UGen que tiene SuperCollider para hacer ruido es el Dust. Genera pulsos aleatorios entre 1 y 0. A diferencia de los LFNoise, que generan amplitudes con una frecuencia definida, el Dust lo hace con una densidad aleatoria, obteniendo una frecuencia estadística. Es decir que si Dust tiene una densidad de 1 obtendremos aproximadamente 1 pulso por segundo. A veces mas, a veces menos.

Dust.ar(densidad, multiplicación, adición)

también es posible generar ruidos por medios distintos. Por ejemplo este código genera un ruido Browniano, de tal forma que se mueve una décima de amplitud cada vez hacia arriba o hhacia abajo aleatoriamente.

```
(
SynthDef(\browniano, {|amp=0|
                var sig;
                sig=SinOsc.ar(0,0,1,amp);
                Out.ar(0,sig);
                }).send(s);
)

~browniano=Synth(\browniano)

(
Tdef(\browniano, {var amp=1;
                inf.do{
                amp=((amp+([1,0, -1]*0.1).choose)%3) -1;
                ~browniano.set(\amp, amp.postln);
                0.001.wait;
                }
                }).quant_(0)
)
```

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

s.scope
```

también podemos generar ruido usando arreglos Demands:

aquí generamos amplitudes aleatorias de 1, 0 o -1 periódicamente usando un Impulse a 1000 hz

{Demand.ar(Impulse.ar(1000),0,Drand([1,0, -1],inf))}.scope

aquí generamos las mismas amplitudes pero en secuencia. Lo que hace que suene como ruido es el Dust que esta pidiendo estas amplitudes sin una periodicidad.

{Demand.ar(Dust.ar(1000),0,Dseq([1,0, -1],inf))}.scope

Síntesis Aditiva

La síntesis aditiva no es más que una técnica donde se suman tonos sinoidales. prácticamente cualquier sonido puede ser descompuesto en tonos puros (sinoidales). Así que, de manera inversa, agregando tonos sinoidales se puede construir un timbre.

Hay tres parámetros que afectan la relación de los sonidos que vamos a sumar y estos son: la frecuencia, la fase y la amplitud. Anteriormente vimos un ejemplo donde sumamos una serie de armónicos, este ejemplo es síntesis aditiva. También podemos sumar tonos que no tengan relación de múltiplos, como en el caso de las campanas cuyos armónicos no guardan relación matemática de múltiplos.

sine1 + sine2 + sine 3 + sine 4 + sine5 + sineN...

```
// suma de sinoidales con relación armónica
(
{var n=5; SinOsc.ar(220,0,1/n)
+ SinOsc.ar(440,0,1/n)
+ SinOsc.ar(660,0,1/n)
+ SinOsc.ar(880,0,1/n)
+ SinOsc.ar(1100,0,1/n)
}.play
)
```

```
// suma de sinoidales sin relación armónica, ni de fase
(
{var n=5; SinOsc.ar(220,0,1/n)
+ SinOsc.ar(475,0,1/n)
+ SinOsc.ar(527,pi,1/n)
+ SinOsc.ar(679,pi/2,1/n)
+ SinOsc.ar(735,3pi/2,1/n)
}.play
)
```

Hay objetos en SC que nos ayudan a sumar la señal como Mix.ar que mezcla un Array de UGens.

Mix.ar([sine1, sine2, sine3, sine4, sine5])

```
// suma de ondas sinoidales usando Mix, guardando una relación
armónica y de fase
(
{var n=5; Mix.ar([SinOsc.ar(220,0,1/n),
SinOsc.ar(440,0,1/n),
SinOsc.ar(660,0,1/n),
SinOsc.ar(880,0,1/n),
SinOsc.ar(1100,0,1/n)])
}.play
)

// suma de ondas sinoidales usando Mix, sin relación armónica ni
de fase
(
{var n=5; Mix.ar([SinOsc.ar(220,0,1/n),
SinOsc.ar(460,pi,1/n),
SinOsc.ar(665,0,1/n),
SinOsc.ar(840,0,1/n),
SinOsc.ar(1110,3pi/2,1/n)])
}.play
)
```

Podemos generar las ondas de los osciladores primarios usando exclusivamente la suma de ondas sinusoidales.

Onda cuadrada

Generamos sinoides con frecuencias de armónicos impares de una fundamental.

Si declaramos esta línea obtenemos un array del 1 al 40
(1..40)

Al multiplicar los elementos por dos obtenemos pares del 2 al 80
(1..40)*2

Y al restar uno convertimos los pares en impares del 1 al 79
(1..40)*2 -1

Usaremos este arreglo para los de armónicos de la frecuencia fundamental 440. Es necesario disminuir la amplitud de cada armónico para que no se sature la señal.

{var arm=(1..40)*2 -1; Mix(SinOsc.ar(440*arm,0,1/arm))}.scope

El resultado es muy similar a

{Pulse.ar}.scope

Onda Triangular

A partir de sinusoidales con frecuencias de los armónicos impares y cambiando cada (4n -1) armónicos la fase a pi, donde n es el número de armónico. Las amplitudes deben de ser el inverso del cuadrado del número de armónico, igual que con la onda cuadrada

```
(
~arm=(1..80)*2 -1;
~armFase=(4*~arm -1);
~arm.size.do{|i|
                if(~armFase.find([~arm[i]])==nil, {
            {SinOsc.ar(440*~arm[i],0,1/(~arm[i]**2))}.play;
                                    },
                                    {
            {SinOsc.ar(440*~arm[i],pi,1/(~arm[i]**2))}.play;
                                    }
                )}
)
```

Onda de Sierra

Se puede generar una onda de sierra sumando sinusoidales con las frecuencias de todos los armónicos de una frecuencia fundamental y disminuyendo la amplitud de cada una por el inverso del numero de armónico

{var arm=(1..100); Mix(SinOsc.ar(440*arm,0,0.5/arm))}.scope

Filtros

Un filtro es un sistema que, dependiendo de algunos parámetros, realiza un proceso de discriminación de una señal de entrada obteniendo variaciones en su salida.

Filtro Pasa Bajas

Es aquel que permite el paso de frecuencias bajas, desde la frecuencia 0 hasta una frecuencia determinada. Esta frecuencia determinada es la frecuencia de corte. Por ejemplo, si determinamos que sea 200 Hz la frecuencia de corte dejaremos pasar todas las frecuencias mas bajas que 200 Hz. Recordemos que no hay frecuencias menores a 0 Hz.

Utilicemos el UGen LPF - Low Pass Filter-- Filtro Pasa Bajas.

Sintaxis : LPF.ar(entrada, frecuencia de corte, multiplicación, adición)

entrada : La señal que queremos filtrar. Tiene que tener .ar.

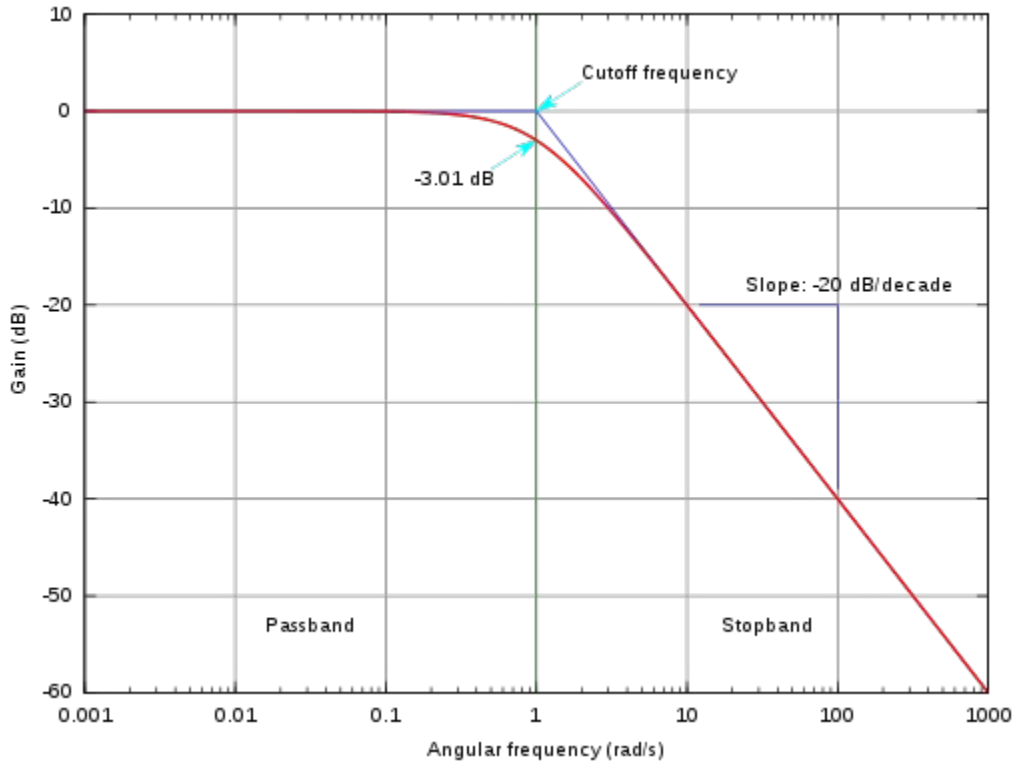
frecuencia de corte : Frecuencia en Hertz a partir de la cual se permitirá el paso de frecuencias mas bajas.
multiplicación : Número por el cual multiplicamos la señal del filtro. Generalmente se identifica con el volumen o amplitud del sonido siendo 0 el mínimo y 1 el máximo recomendado. El default es 1 dejando la señal sin alterar.
adición : Número que se le suma a la señal del filtro. Observar que a la señal se le aplica primero el mul y luego el add. El default es 0 dejando la señal sin alterar.

Ejemplos:

Pasan las frecuencias debajo de los 7030 Hz
{ LPF.ar(WhiteNoise.ar(0.7), 7030)}.scope //

Con los movimientos del mouse en y podemos variar la frecuencia de corte:
{ LPF.ar(WhiteNoise.ar(0.7), MouseY.kr(17000,200))}.scope

Un filtro ideal impediría completamente el paso de las frecuencias menores que la frecuencia de corte usando algo así como una recta vertical o un muro. Pero lamentablemente no vivimos en un mundo ideal y los filtros no pueden hacer esta función. Lo que hacen es un aproximado que resulta en una curva que va disminuyendo la amplitud de las frecuencias. Esto sucede con los filtros Pasa altas y Pasa bandas que veremos a continuacion



Filtro Pasa Altas

Es aquel que permite el paso de frecuencias desde una frecuencia determinada hhacia arriba, sin que exista un límite superior especificado. Esta frecuencia determinada es la frecuencia de corte. Por ejemplo, si determinamos que sea 700 Hz la frecuencia de corte dejaremos pasar todas las frecuencias mas altas que 700 Hz.

SuperCollider usa la clase HPF (High Pass Filter) para esto.

HPF.ar(entrada, frecuencia de corte, multiplicación y adición)

entrada : La señal que queremos filtrar. Tiene que tener .ar.

frecuencia de corte : Frecuencia en Hertz a partir de la cual se permitirá el paso de frecuencias mas altas.

multiplicación : Número por el cual multiplicamos la señal del filtro. Generalmente se identifica con el volumen o amplitud del sonido siendo 0 el mínimo y 1 el máximo recomendado. El default es 1 dejando la señal sin alterar.
adición : Número que se le suma a la señal del filtro. Observar que a la señal se le aplica primero el mul y luego el add. El default es 0 dejando la señal sin alterar.

Ejemplos

```
{ HPF.ar(WhiteNoise.ar(0.7), 7030)}.scope
{ LPF.ar(WhiteNoise.ar(0.7), MouseY.kr(17000,200))}.scope
```

Filtro Pasa Banda

Es aquel que permite el paso de frecuencias contenidas dentro de un determinado rango o banda, comprendido entre una frecuencia inferior y otra superior. La distancia entre estas frecuencias determina el ancho de banda. La frecuencia que está en el centro de esta distancia es la frecuencia de corte. Por ejemplo, si determinamos que sea 1000 Hz la frecuencia de corte y 200 Hz el ancho de banda podemos saber cual es el rango de frecuencias que dejaremos pasar usando la siguiente fórmula:

cota inferior = frecuencia de corte - ancho de banda/2

cota superior = frecuencia de corte + ancho de banda/2

La cota superior es la frecuencia límite superior y la cota inferior es la frecuencia límite inferior.

Entonces

cota inferior = 1000 - 200/2 = 900
cota superior = 1000 + 200/2 = 1100

Si sabemos cuáles son las cotas inferior y superior que queremos entonces podemos obtener el ancho de banda y la frecuencia de corte con la siguiente fórmula:

ancho de banda = cota superior - cota inferior

frecuencia de corte = cota inferior + ancho de banda/2

Entonces, siguiendo el ejemplo anterior

ancho de banda = 1100 - 900 = 200
frecuencia de corte = 900 + 200/2 = 1000

SuperCollider tiene el UGen BPF (Band Pass Filter)

BPF no es el único filtro que utiliza un ancho de banda. En este filtro y en todos los demás de este tipo el ancho de banda no se puede escribir directamente como un argumento. En vez de ancho de banda estos filtros tienen como argumento rq.

q = frecuencia de corte / ancho de banda.
Por lo tanto el recíproco de q = 1/q = ancho de banda / frecuencia de corte = rq

El máximo de rq es 2 y el mínimo es 0. El rq no puede ser mayor que 2 por que la cota inferior se iría por debajo de los 0hz.

Ejemplo

frecuencia de corte = 1000
rq = 2.5 = ancho de banda/1000

Despejando tenemos que

ancho de banda = 2.5 * 1000 = 2500

Entonces

cota inferior = 1000 - 2500/2 = -250

Sintaxis : BPF.ar(entrada, frecuencia de corte, rq, multiplicación, adición)

entrada : La señal que queremos filtrar.

frecuencia de corte : Frecuencia en Hertz que determina el centro de la banda de nuestro filtro.

rq : recíproco de q, es decir, ancho de banda / frecuencia de corte.

multiplicación : Número por el cual multiplicamos la señal del filtro. Generalmente se identifica con el volumen o amplitud del sonido siendo 0 el mínimo y 1 el máximo recomendado. El default es 1 dejando la señal sin alterar.

adición : Número que se le suma a la señal del filtro. Observar que a la señal se le aplica primero el mul y luego el add. El default es 0 dejando la señal sin alterar.

Algunos ejemplos.

```
// El tercer argumento del filtro pasa banda es el reciproco de
Q.
```

```
{ BPF.ar(WhiteNoise.ar(0.1), 7030, 700/7000)}.scope
```

En este ejemplo tenemos ancho de banda=700 y frecuencia de corte=7000. O sea, 700/7000=0.1 Por lo tanto rq=0.1

A veces es más rápido escribir el número décimal que el quebrado. Veamos entonces como queda sustituyendo del ejemplo anterior:

```
{ BPF.ar(WhiteNoise.ar(0.1), 7030, 0.1)}.scope

// aquí tenemos otro valor para el rq.
{ BPF.ar(WhiteNoise.ar(0.1), 7030, 1)}.scope
```

Si tenemos que 1 = ancho de banda / frecuencia de corte = rq, entonces sabemos que ancho de banda=frecuencia de corte = rq.
Y si frecuencia de corte = 7030,
entonces rq = 7030 / 7030.

Por lo tanto sabemos que la linea de código anterior se puede escribir también de la siguiente forma:

```
{BPF.ar(WhiteNoise.ar(0.1), 7030, 7030/7030)}.scope
Por último usemos el control del mouse en y para la frecuencia de
corte y en x para el rq

{ BPF.ar(WhiteNoise.ar(0.7), MouseY.kr(17000,200),
MouseX,kr(0,2))}.scope
```

Bibliografía

Hutchins, C. (2005). *SuperCollider Tutorial*.

Netri, E. y Romero, E. (2008). *Curso de SuperCollider: principiantes.* México DF: Centro Multimedia.

SuperCollider Help.

Wikipedia. Filter (signal processing). Recuperado de: http://en.wikipedia.org/w/index.php?title=Filter_%28signal_processing%29&oldid=493194971

Música por computadora

Ernesto Romero y Hernani Villaseñor
Centro Multimedia 2012

SESIÓN 13

Síntesis Sustractiva

La síntesis sustractiva funciona bajo el principio de restar o moldear sonido complejo mediante filtros. Generalmente se trabaja con ruido o sonidos complejos con un amplio espectro de frecuencias los cuales son filtrados hasta obtener un timbre deseado.

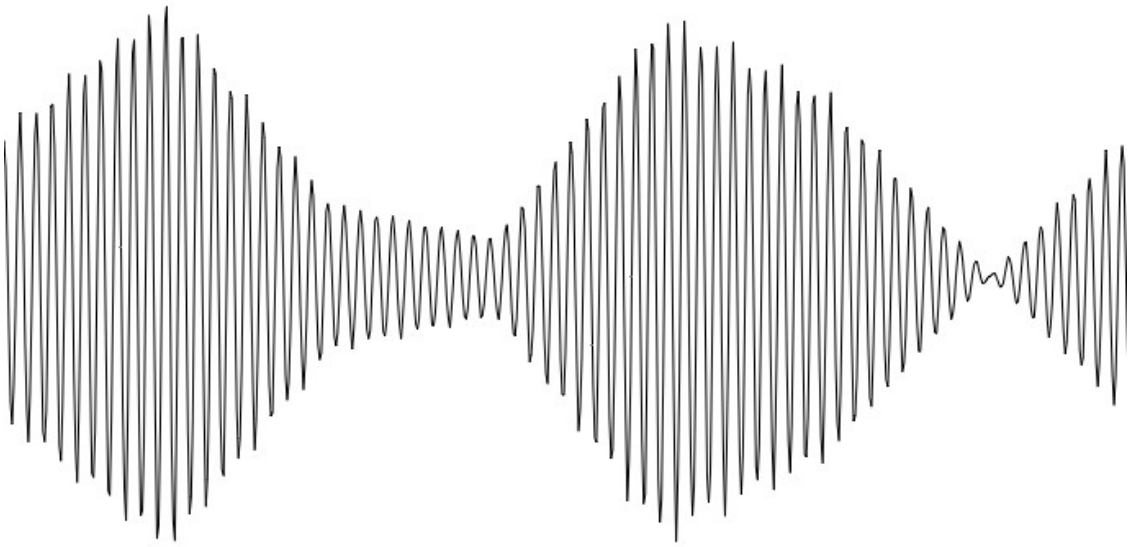
```
// ejemplo con un filtro sobre ruido rosa
{BPF.ar(PinkNoise.ar(0.5), MouseX.kr(40,10000),
MouseY.kr(0.01,2.0)}.scope

// síntesis sustractiva más compleja
(
SynthDef(\sustractiva,{|rq=0.01,c_frec=#[1,2,3,4,5,6,7,8,9,10],
gate=1|
     var sen, senfilt, mix, env;
     sen=WhiteNoise.ar(1);
     senfilt=Mix(BPF.ar(sen,c_frec,rq,1));
     env=EnvGen.kr(Env.asr(0.1,1,1),gate,doneAction:2);
     Out.ar(0,senfilt*env)
     }).send(s)
)

a=Array.exprand(10,1,1.5)
d=Synth(\sustractiva,[\c_frec,a*200])
d.set(\c_frec,a*exprand(200,1500))
d.set(\gate,0)
```

Modulación

Revisemos primero el concepto de Oscilador de Baja Frecuencia o LFO (Low Frequency Oscillator). El concepto de LFO es el de una frecuencia muy baja que sirve para modular una señal. Una oscilación puede ser vista como movimiento de vaivén, el cual puede ser usado para modular el parámetro de frecuencia o el de amplitud, en ambos casos escucharemos un sonido que se mueve en su altura o en su intensidad a la velocidad que oscila la baja frecuencia. Este comportamiento es el principio de la síntesis de Frecuencia Modulada (FM) y Amplitud Modulada (AM) respectivamente, las cuales veremos con detalle más adelante.



Gráfica que representa la modulación de amplitud.

LF-Unit Generators (Low Frequency UGens) Osciladores de baja frecuencia

SC tiene algunos UGens que están diseñados para este propósito. Estos generadores pueden ser usados para controlar o modular argumentos de la señal a partir de bajas frecuencias, aunque también pueden sonar, pero teniendo en cuenta que a altas frecuencias generan frecuencias alias.

LFPar, LFCub, LFTri, Impulse, LFSaw, LFPulse, VarSaw, SyncSaw

LFPar, LFCub, LFTri, LFSaw, Impulse
Argumentos: frecuencia, fase, multiplicador y suma

LFPulse, VarSaw
Argumentos: frequencia, fase, ancho de pulso, multiplicador y suma

SyncSaw
Argumentos: syncFrec, sawFrec, multiplicador y suma

Ejemplo de un SinOsc controlado en su frecuencia por un LFO (oscilador de baja frecuencia).

```
// oscilación lenta
{SinOsc.ar(LFPar.ar(0.2,0,200,800),0,0.5)}.play

// oscilación rápida
{SinOsc.ar(LFPar.ar(20,0,200,800),0,0.5)}.play
```

Tipos de síntesis por modulación

La modulación no es otra cosa que alterar la amplitud, frecuencia o fase de un oscilador por otra señal.

Frecuencia portadora: es la frecuencia que es modulada.

Frecuencia modulante: es la frecuencia que modula.

Bandas laterales: son frecuencias resultantes de la modulación.

Modulación AM

Dodge y Jerse (90), mencionan que hay tres tipos de amplitud modulada: la “clásica” amplitud modulada (AM), modulación por anillo (Ring Modulator) y modulación de banda lateral sencilla (single-sideband). De estas tres técnicas la AM y la modulación por anillo son las que revisaremos.

Para realizar síntesis AM se necesitan dos señales: una portadora y una modulante. Llamaremos amp a un valor de amplitud que será común para ambas frecuencias. En la señal modulante amp es multiplicada por un índice de modulación que normalmente va de 0 a 1. Si su valor es 0 no habrá modulación. Si el valor es 1 la modulación será máxima. En la señal portadora amp es multiplicado por la señal moduladora. El resultado final es una AM. En la técnica de AM escuchamos la frecuencia portadora y dos bandas laterales resultantes que son la suma y la diferencia de la frecuencia portadora y la frecuencia modulante.

banda lateral 1 = F portadora + F modulante
banda lateral 2 = F portadora - F modulante

Si tenemos una frecuencia portadora de 550 y una frecuencia modulante de 150, las bandas laterales serán de 300 y 700.

```
(
SynthDef(\am,{|frec_portadora=450, frec_modulante=2, ind_mod=1,
amp=0.5, gate=1|
     var portadora, modulante, envolvente;
     modulante=SinOsc.kr(frec_modulante,0,amp*ind_mod);
     portadora= SinOsc.ar(frec_portadora,0,amp*modulante);
     envolvente=EnvGen.kr(Env.asr(0.1,1,1),gate,doneAction:2);
     Out.ar(0,Pan2.ar(portadora,0,1)*envolvente)
}).send(s)
)

b=Synth(\am)
b.set(\gate,0)
```

La amplitud de las bandas laterales será de (ind_mod/2) * amp. Así, si la amplitud es 0.5, y el índice de modulación 1 tenemos que la amplitud de cada una de las bandas laterales es de (1/2) * 0.5 = 0.25. La amplitud total de la onda resultante es la suma de la amplitud de la frecuencia portadora más las amplitudes de las laterales.

amp portadora = 0.5
amp lateral 1 = 0.25
amp lateral 2 = 0.25

amp total = 1

Podemos comparar el resultado de la AM con portadora en 550 y modulante en 150 con tres sinusoidales con las frecuencias de la portadora y las bandas laterales y sus respectivas amplitudes.

```
~l1={SinOsc.ar(550 - 150, 0, 0.25)}.play;
~fp={SinOsc.ar(550, 0, 0.5)}.play;
~l2={SinOsc.ar(550 + 150, 0, 0.25)}.play;

~l1.free;
~fp.free;
~l2.free;
```

Modulación por Anillo

La modulación por anillo o *ring modulator* es un tipo de síntesis que toma su nombre de la forma que tenían los circuitos cuando eran diseñados de manera análoga, esta síntesis funciona al multiplicar la frecuencia portadora por la frecuencia modulante. Se utiliza solo una amplitud para la señal moduladora. Es importante recalcar que solo suenan la frecuencia de las bandas laterales.

F portadora * F modulante

```
(
SynthDef(\ringmodul,{|frec_portadora=550, frec_modulante=150,
amp=1, gate=1|
     var portadora, modulante, envolvente;
     modulante=SinOsc.ar(frec_modulante,0,amp);
     portadora= SinOsc.ar(frec_portadora,0,modulante);
     envolvente=EnvGen.kr(Env.asr(0.1,1,1),gate,doneAction:2);
     Out.ar(0,portadora*envolvente)
     }).send(s)
)

a=Synth(\ringmodul)
a.set(\gate,0)
```

Modulación FM

La modulación FM, según Hutchins (10), fue descubierta en Stanford por John Chowing en 1973. Posteriormente la licencia fue adquirida por Yamaha para implementarla en el sintetizador DX7.

En esta técnica la frecuencia moduladora hace su trabajo de modulación sobre la frecuencia portadora, causando un tipo de vibrato según la forma de onda moduladora.

```
(
SynthDef(\fm,{|gate=1 |
     var modulante, portadora, env;
     modulante=SinOsc.kr(4,0,100,300);
     portadora=SinOsc.ar(modulante,0,1);
     env=EnvGen.kr(Env.asr(0.1,1,1),gate, doneAction:2);
     Out.ar(0,portadora*env)
}).send(s))
```

```
c=Synth(\fm)
c.set(\gate,0)
```

Modulación de Fase

Es un tipo de modulación FM, donde la fase de la frecuencia portadora es modulada por una frecuencia modulante.

```
(
SynthDef(\fase,{|gate=1,frec=300, frec_mod=2 |
     var modulante, portadora, env;
     modulante=SinOsc.kr(frec_mod,0,pi,pi);
     portadora=SinOsc.ar(frec,modulante,1);
     env=EnvGen.kr(Env.asr(0.1,1,1),gate, doneAction:2);
     Out.ar(0,portadora*env)
     }).send(s)
)

f=Synth(\fase)
f.set(\frec_mod,77)
f.set(\gate,0)
```

Bibliografía

Dodge, Ch. y A. Jerse, T. (1997). *Computer Music: Synthesis, composition and performance*. Schirmer.

Hutchins, C. (2005). *SuperCollider Tutorial: chapter 6.*

Valle, A. (2008). *The SuperCollider Italian Manual at CIRMA.* Torino: CIRMA.

Música por computadora

Ernesto Romero y Hernani Villaseñor

Centro Multimedia 2012

SESIÓN 14

Convolución

Es una forma matemática de combinar dos señales para obtener una tercera, para lo cual se usa una estrategia de descomposición de impulso. En la convolución convergen tres señales: señal de entrada, señal de salida y respuesta al impulso (Smith, 1999:109).

Waveshaping

Es una técnica de síntesis por distorsión que produce un espectro con una evolución dinámica de sus componentes, pero de banda limitada, es decir, se establece un número máximo de armónicos (Dodge & Jerse, 1997:139).

El espectro en esta síntesis es generado a partir de una distorsión controlada sobre la amplitud, esto mediante un índice que permite una variación dinámica.

Para realizar esta técnica se utiliza un *waveshaper* que es el encargado de alterar la señal que pasa por el, esto ocurre cuando a la señal de entrada se le incrementa la amplitud, lo que causa un cambio de su forma en la salida, de esta manera se incrementa el número e intensidad de los armónicos.

En SC existe el objeto Shaper que sirve para hacer este tipo de síntesis. Antes, es necesario definir una tabla de onda con un Buffer.alloc

La tabla de onda almacenada en nuestro Buffer es para definir una serie de amplitudes que moldearan nuestra onda. Entonces con el método .cheby definimos el Array de amplitudes que leerá el Shaper para modificar el timbre.

Shaper.ar/kr(bufnum, entrada, multiplicación, adición)

bufnum: el número de buffer vertido en un formato de tabla de onda (wavetable) que cumple la función de transfer
entrada: la señal de entrada

```
// primero hacemos un buffer
b = Buffer.alloc(s, 1024, 1)

// luego utilizamos el método .cheby con el que mandamos un array
de amplitudes. Puedes añadir más amplitudes al array
b.cheby([1,0.5,1,0.125]);

// usamos nuestro waveshaper con una onda sonoidal
(
SynthDef(\ws,{|frec=440, gate=1|
var sen, env;
sen=Shaper.ar(b, SinOsc.ar(440, 0, 0.5));
env=EnvGen.kr(Env.asr(0.1,1,1),gate,doneAction:2);
Out.ar(0,sen*env);
}).send(s)
)

c=Synth(\ws)
c.set(\gate,0)
```

Este fenómeno ocurre de manera similar en los instrumentos acústicos, por lo que esta técnica es muy eficiente para sintetizar instrumentos, sobre todo de aliento/metal.

Karplus-Strong

Este tipo de síntesis pertenece al grupo de síntesis por modelado físico, técnica basada en la forma física de los instrumentos musicales. Entre las técnicas de modelado físico, esta es relativamente sencilla. Lo que modela esta síntesis es una cuerda pulsada.

Según Collins, se requiere de un ruido dentro de una línea de delay basado en la altura de la nota que queremos obtener, después se filtra el delay de manera sucesiva hasta que el sonido decae.

SC tiene implementado el UGen Pluck.ar para generar síntesis Karplus-Strong.

Pluk.ar (señal de exitación, *trigger,* tiempo max delay, tiempo delay, tiempo decay, coef)

señal de exitación: una señal de audio compleja
trigger: una señal de impulso para ingresar la señal a la línea de delay
tiempo max delay: tiempo máximo de delay en segundos que inicializa el *buffer* interno
tiempo delay: el tiempo de delay en segundos

tiempo decay: tiempo de caída del eco, valores negativos enfatizan armónicos nones
coef: el coef de del filtro interno, rango -1 a +1.

```
{Pluck.ar(PinkNoise.ar(0.5),Impulse.kr(3),440.reciprocal,440.rec
iprocal,10,0.1)}.play
```

.reciprocal se usa para obtener el recíproco, en este caso 1/440, lo que afina nuestra Karplus-strong en La, también puede usarse con .midicps para acceder a las notas que queramos.

Se puede diseñar este tipo de síntesis usando ruido y una línea de delay como puede ser CombL. Cottle (2005:151) propone el siguiente ejemplo, el cual hemos adaptado:

```
(
{
var arranqueEnv, atk = 0, dec = 0.001;
var arranque, delayTiempo, delayDecay = 0.5;
var notaMidi = 69; // A 440
delayTiempo = notaMidi.midicps.reciprocal;
//RandSeed.kr(Impulse.kr(1/delayDecay), 111);
arranqueEnv = EnvGen.kr(Env.perc(atk, dec), gate:
Impulse.kr(1/delayDecay));
arranque = PinkNoise.ar(arranqueEnv);
CombL.ar(arranque, delayTiempo, delayTiempo,
delayDecay, add: arranque);
}.play
)
```

Lo que sucede en el ejemplo anterior es que estamos mandando un ruido rosa con un envolvente muy corto a través de la línea de delay CombL, para repetirlo varias veces mientras decae, así es posible percibir un tono.

Nota que delayTiempo es convertido a nota midi y después se obtiene su recíproco, así conseguimos afinar el resultado.

Si descomentas RandSeed.kr obtendrás a cada impulso el mismo arranque de ruido, por lo que cada pulsación sonará igual.

Bibliografía

Collins, N. SCCourse. Recuperado de: http://www.sussex.ac.uk/Users/nc81/courses/cm1/workshop.html

Cottle, D.M. (2005). *Computer Music with examples in SuperCollider 3.*

Dodge, Ch. y A. Jerse, T. (1997). *Computer Music: Synthesis, composition and performance*. Schirmer.

Smith, S.W. (1999). *The Scientist and Engineer's Guide to Digital Sound Processing.* San Diego: California Technical Publishing.

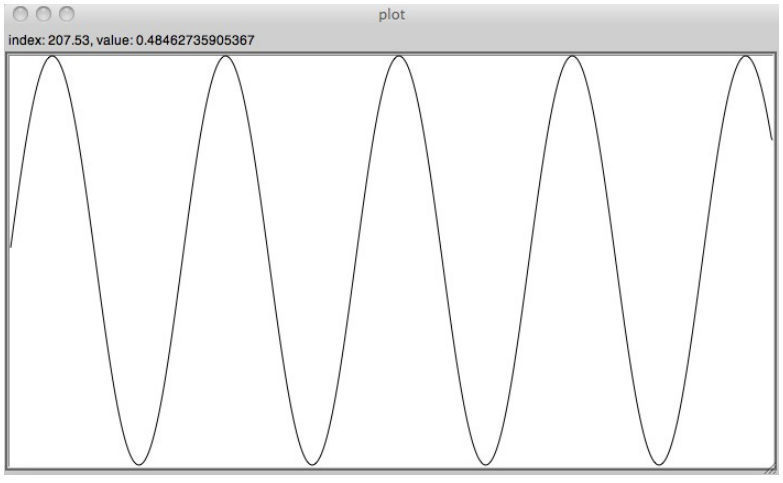
Música por computadora

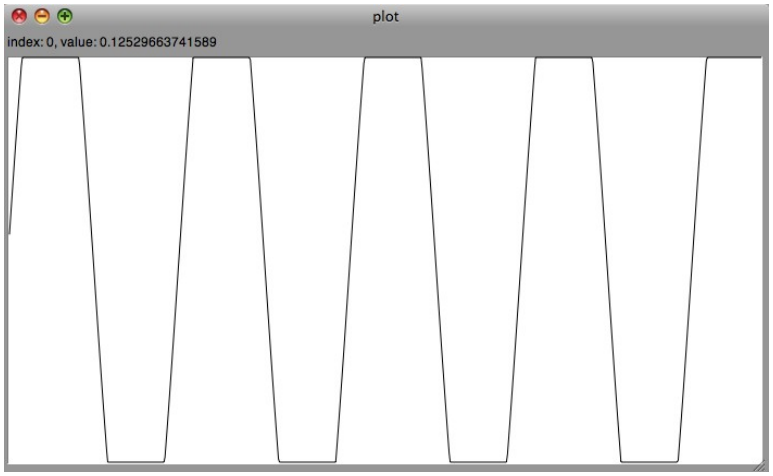
Ernesto Romero y Hernani Villaseñor
Centro Multimedia 2012

SESIÓN 15

Distorsión

La distorsión en la señal de audio se da cuando está sobrepasa la capacidad de un componente para reproducir adecuadamente el proceso al que se está sometiendo. La distorsión más común es la generada por el proceso de incrementar la amplitud. Cuando la señal sufre un recorte abrupto sobre la amplitud esta se cuadratiza, fenómeno que es conocido como *clipping.* Lo que sucede es que se la señal adquiere una forma de onda cuadrada en los picos lo que genera un gran contenido de armónicos y así se produce el sonido que conocemos como “distorsionado”. Este sonido entonces es una distorsión de la amplitud.





En las gráficas anteriores vemos en el primer caso un SinOSc con mul:1, en el segundo caso un SinOsc con mul:2, lo que sobrepasa su capacidad de rango dinámico y cuadratiza la forma redonda de la onda sinoidal.

En SC existen objetos y mensajes que tienen como finalidad generar distorsión de manera deliberada, veamos algunos.

Clip

Cuadratiza la señal a partir de un umbral determinado.

Clip.ar/kr(entrada, bajo, alto)

entrada: señal de entrada
bajo: umbral de cuadratización bajo
alto: umbral de cuadratización alto

```
// cuadratizando la onda sinoidal, prueba otros UGens
{Clip.ar(SinOsc.ar(440,0,1), -0.5, 0.5}.scope
```

Nota que ambos umbrales deben ser un valor menor al del argumento de mul de la señal, de otra manera no ocurrirá la cuadratización.

.clip2

Este método cuadratiza de igual manera la señal, solo que a diferencia del Objeto Clip en este método no se definen los umbrales. Es necesario que el argumento de este mensaje sea menor al valor de argumento mul de la señal al que se esta aplicando.

```
// para verlo
{SinOsc.ar(440,0,1).clip2(0.15)}.plot

// para oirlo
{SinOsc.ar(440,0,1).clip2(0.15)}.scope
```

Wrap

Enrolla una señal a partir de umbrales definidos.

Wrap.ar/kr(entrada,bajo,alto)

entrada: señal que será enrollada
bajo: umbral bajo del enrrollado, debe ser menor que el alto
alto: umbral alto del enrollado, debe ser mayor que el bajo

.wrap2

Este método enrrolla la señal de manera bilateral, es un operador binario, que actúa sobre la señal de tal manera que la dobla, a diferencia del objeto Wrap, este método no define el umbral alto y bajo.

El mensaje .wrap2 recibe el argumento de la cantidad de doblado que haremos, este valor debe ser menor que el argumento :mul de nuestra señal para que actue.

```
// para ver como actua
{SinOsc.ar(200,0,1).wrap2(0.2)}.plot

// para escucharlo
{SinOsc.ar(200,0,1).wrap2(0.2)}.scope
```

.distort

Es un método de distorsión no lineal perteneciente al grupo de operadores unitarios.

Manipulación de muestras de audio

s.boot;

b=Buffer.read(s, "/home/tito/share/SuperCollider/sounds/noPontoLoop.wav")
b.play
b.query

```
RecordBuf.ar(inputArray, bufnum, offset, recLevel, preLevel, run, loop,
trigger, doneAction)
(
SynthDef(\record, {|buf=0, dur=44100|
                    var sig, env;
                    sig=RecordBuf.ar(SoundIn.ar(0),buf,loop:0);
                    env=EnvGen.kr(Env([0,1,1,0],[0.01,dur -0.02,0.02]),doneAction:2);
}).send(s);
)
 s.meter

c=Buffer.alloc(s,44100*4,1)
Synth(\record, [\buf, c.bufnum, \dur, 44100*4])
c.play
c.write("/home/tito/share/SuperCollider/sounds/voz.aiff", "AIFF", "int16")

write(path, headerFormat, sampleFormat, numFrames, startFrame, leaveOpen,
completionMessage)
path:  string, entre comillas. Indica la ruta en donde se quiere escribir el archivo.
headerFormat: Formato de audio con el que se quiere escribir el archivo. Debe ir entre comillas
i con mayusculas(e.g."AIFF")
sampleFormat: Formato de sampleo con el que se quiere escribir el archivo (e.g. "int16")


d=Buffer.alloc(s,44100*4,1)
Synth(\record, [\buf, d.bufnum, \dur, 44100*4])
d.play
d.write("/home/tito/share/SuperCollider/sounds/garganta.aiff", "AIFF", "int16")

e=Buffer.alloc(s,44100*4,1)
Synth(\record, [\buf, e.bufnum, \dur, 44100*4])
e.play
e.write("/home/tito/share/SuperCollider/sounds/masca.aiff", "AIFF", "int16")

f=Buffer.alloc(s,44100*1,1)
Synth(\record, [\buf, f.bufnum, \dur, 44100*4])
f.play
f.write("/home/tito/share/SuperCollider/sounds/pop.aiff", "AIFF", "int16")


PlayBuf.ar(numChannels, bufnum, rate, trigger, startPos, loop, doneAction)

(
SynthDef(\play, {|bufnum=0, rate=1, trigFreq=0.01, pos=0, loop=1, dur=44100, amp=1|
```

```
		var trigger,sig, env;
		trigger=Impulse.kr(trigFreq);
		sig=PlayBuf.ar(1,bufnum,rate,trigger,pos,loop)*amp;
		env=EnvGen.kr(Env([0,1,1,0],[0.01,dur -0.02,0.02]),doneAction:2);
		Out.ar(0,sig*env);
}).send(s);
)


Synth(\play, [\bufnum, c.bufnum, \dur, (c.numFrames/44100), \rate, 1, \trigFreq, 0.01, \pos, 0,
\loop, 0])
Synth(\play, [\bufnum, c.bufnum, \dur, (c.numFrames/44100), \rate, 2, \trigFreq, 0.01, \pos, 0,
\loop, 0])
Synth(\play, [\bufnum, c.bufnum, \dur, (c.numFrames/44100)*4, \rate, 1, \trigFreq, 0.01, \pos,
0, \loop, 1])
Synth(\play, [\bufnum, c.bufnum, \dur, (c.numFrames/44100), \rate, 1, \trigFreq, 2, \pos, 0, \loop,
0])

Synth(\play, [\bufnum, c.bufnum, \dur, (c.numFrames/44100), \rate, -1, \trigFreq, 0.01, \pos,
(c.numFrames/44100)/4, \loop, 1])


(
Tdef(\bufnum, {var buf;
	inf.do{
		buf=[c,d,e,f].choose.postln;
		Synth(\play, [\bufnum, buf.bufnum,
			\dur, (buf.numFrames/44100),
			\rate, 1,
			\trigFreq, 0.01,
			\pos, 0,
			\loop, 1]);
		(buf.numFrames/44100).wait;
			}
	}).quant_(0);
)

Tdef(\bufnum).play
Tdef(\bufnum).stop



(
Tdef(\rate, {var rate;
```

```
	inf.do{
		rate=rrand(0.5,2)*[1, -1].choose;
		Synth(\play, [\bufnum, c.bufnum,
			\dur, (c.numFrames/44100),
			\rate, rate,
			\trigFreq, 0.01,
			\pos, (c.numFrames/44100),
			\loop, 1]);
		1.wait;
			}
	}).quant_(0);
)

Tdef(\rate).play
Tdef(\rate).stop


BufDur.kr(bufnum) // Duracion del buffer en segundos

u={PlayBuf.ar(1,e.bufnum,1,Impulse.kr(MouseY.kr(0.1,40)),MouseX.kr(0,BufDur.kr(e.bufnum)*4
4100).poll)}.play
u.free

(
Tdef(\trigPos, {var trigFreq, pos;
	inf.do{
		trigFreq=[8,4,2].choose;
		pos=[91815, 79289, 140242].choose;
		Synth(\play, [\bufnum, e.bufnum,
			\dur, 1/trigFreq,
			\trigFreq, trigFreq,
			\pos, pos,
			\amp, 0.5,]);
		(1/trigFreq).wait;
			}
	}).quant_(0);
)
(
i={RLPF.ar(Impulse.ar(2,[0,0.5]), [1000,2000],0.1)*8}.play;
Tdef(\trigPos).play
)
(
i.free;
Tdef(\trigPos).stop
```

```
)

(
SynthDef(\playMod, {|bufnum=0,dur=44100, amp=1, ringFreq=1000, ringAmp=0, ringAdd=1,
fmFreq=0,d=0,rate=1, loop=0|
                    var rateFM,sig, env;
                    rateFM=SinOsc.kr(fmFreq,0,d,rate);
                    sig=PlayBuf.ar(1,bufnum,rateFM,
loop:loop)*SinOsc.kr(ringFreq,0,ringAmp,ringAdd);
                    env=EnvGen.kr(Env([0,1,1,0],[0.01,dur -0.02,0.02]),doneAction:2);
                    Out.ar(0,sig*env*amp);
}).send(s);
)

// Sin modulación
Synth(\playMod, [\bufnum, c.bufnum, \dur, (c.numFrames/44100)])


// Modulación de Anillo
Synth(\playMod, [\bufnum, c.bufnum, \dur, (c.numFrames/44100), \ringFreq, 1000, \ringAmp,
1, \ringAdd, 0])


// Frecuencia modulada (rate)

Synth(\playMod, [\bufnum, c.bufnum, \dur, (c.numFrames/44100), \fmFreq, 7, \d, 0.05, \rate, 1])
Synth(\playMod, [\bufnum, c.bufnum, \dur, (c.numFrames/44100), \fmFreq, 100, \d, 10, \rate,
100, \loop, 1])


// Control grano
(
SynthDef(\playGrano, {|bufnum=0, rate=1, pos=0, dur=44100, amp=1|
                    var sig, env;
                    sig=PlayBuf.ar(1,bufnum,rate,1,pos)*amp;
                    env=EnvGen.kr(Env([0,1,1,0],[0.001,dur -0.002,0.001]),doneAction:2);
                    Out.ar(0,sig*env);
                    }).send(s);
)


Synth(\playGrano, [\bufnum, d.bufnum, \dur, 0.01, \rate, 1,\pos, 0])

~buf=d.bufnum;
```

```
~rate=rrand(0.5,2);
~dur=0.01;
~pos=rrand(0,d.numFrames);
(
Tdef(\grano, {var rate, dur, pos;
	inf.do{
		rate=rrand(0.5,2);
		dur=~dur;
		pos=~pos;
		Synth(\playGrano, [\bufnum, ~buf,
			\dur, dur,
			\rate, rate,
			\pos, ~pos ]);
		~dur.wait;
			}
	}).quant_(0);
)

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

Audacity es un programa de uso libre donde pueden editar sus sonidos antes de pasarlos a SuperCollider.

Música por computadora

Ernesto Romero y Hernani Villaseñor
Centro Multimedia 2012

Sesión 16

Síntesis Granular

La síntesis granular fue explorada de manera conceptual y aplicada en instrumentos y música electrónica a finales de los años 50 por Iannis Xenakis e implementada posteriormente en la computación por Road Curtis y Barry Truax en los años 70 (Collins, 5.2).

La síntesis granular parte del presupuesto que fragmentos microscópicos de sonido al ser escuchados juntos son percibidos como un sonido continuo (Valle, 245:2008).

Cuando combinamos muchos sonidos microscópicos para formar grandes nubes sonoras, podemos obtener paisajes sonoros macroscópicos (Collins, 5.2).

La duración de estos granos de sonido puede variar entre 1 y 100 ms, cada grano tiene un envolvente de amplitud definido lo que influye en el resultado sonoro.

En la síntesis granular podemos trabajar a partir de ondas sinoidales con un envolvente muy corto o pequeñas muestras de sonido.

Esta síntesis se basa en señal impulsiva, los granos pueden ser controlados sobre el tiempo para obtener masas sonoras, por ejemplo al alterar su densidad.

Cada grano tiene diferentes parámetros: afinación, duración del envolvente o ventana, posición en el campo estéreo o paneo.

La nube sonora cuyos parámetros afectan globalmente al sonido son: rango de frecuencias de los granos, densidad o cantidad de granos.

Materiales con los que podemos trabajar:

```
SinOsc.ar // con envolventes micro
FSinOsc.ar // con envolventes micro
PlayBuf.ar // Muestras de Sonido
```

Objetos para síntesis granular:

GrainBuf

GrainBuf.ar(numCanales, disparo, dur, sndbuf, velocidad, pos, interpolación, pan, envbufnum, maxGranos)

Es síntesis granular a partir de un sonido guardado en un buffer.

Necesitamos leer un sonido desde un buffer para lo que utilizamos Buff.read y lo igualamos a una variable.

```
b = Buffer.read(s, "sounds/a11wlk01.wav")
(
SynthDef(\GrainBuf,{|sndbuf,envbuf ,gate=1|
var sen, env, pan;
sen=GrainBuf.ar(1,
Impulse.ar(10),0.1,sndbuf,LFNoise1.kr.range(1,2),LFNoise2.kr.ran
ge(0,1),1,0,envbuf)*4;
env=EnvGen.kr(Env.asr(1,1),gate,doneAction:2);
Out.ar(0,sen*env)
}).send(s)
)

a=Synth(\GrainBuf)
```

TGrains

TGrains.ar (numCanales, disparo, bufnum, velocidad, posCentral, dur, pan, amp, interpolación)

Es un granulador de Buffer por lo que necesitamos crear un Buffer.alloc en el cual alojaremos sonido. Entonces el Buffer lo diseñamos para poder alojar sonido en segundos. Si sabemos que un segundo de audio en formato digital contiene 44100 muestras, ese número lo multiplicamos por la cantidad de segundos que queramos nuestro Buffer.

```
// buffer con dos segundos de duración en mono
b = Buffer.alloc(s, 44100 * 2, 1)

// luego hacemos un buffer para grabar sonido
```

```
(
SynthDef(\grabar,{|sen=0|
     var entrada;
     entrada=SoundIn.ar(sen);
     RecordBuf.ar(entrada, b.bufnum);
     }).send(s)
)

x = Synth(\grabar)
x.free // al declarar esta linea se queda un sonido en el buffer

// reproducimoslo grabado
(
SynthDef(\reproducir,{|salida=0|
     var senal;
     senal=PlayBuf.ar(1,b.bufnum,1);
     Out.ar(salida,senal!2)
     }).send(s)
)

y=Synth(\reproducir)

// granulamos el buffer con TGrains, usamos BufDur para saber la
duración del buffer

(
SynthDef(\Tgrains,{|salida=0, buffer=0|
     var senal;
     senal=TGrains.ar(2,Impulse.ar(26),b.bufnum,1,MouseX.kr(0,Bu
     fDur.kr(b.bufnum)),0.1,0,1,2);
     Out.ar(0,senal!2)
     }).send(s)
)

w=Synth(\Tgrains)
```

Warp1

Warp1.ar (numChannels, bufnum, pointer, freqScale, windowSize, envbufnum, overlaps, windowRandRatio, interp, mul, add)

Este UGen estira y comprime un buffer. Basado en un objeto de Csound hecho por Richard Krpens.

También existen esto objetos GrainIn, GrainSin y GrainFM

Método manual

Para hacer una sintesis granular con completo dominio de cada parametro se puede usar el siguiente algoritmo que consiste en tres partes:

a- Un SynthDef que genere los granos
b- Un Tdef que active la granulacion
c- Un grupo de Tdef que cambien los parametros en el tiempo

a- a- Un SynthDef que genere los granos

s.boot

Primero cargamos en un buffer la muestra de audio que queramos granular

```
g=Buffer.read(s,"/home/harpo/share/SuperCollider/sounds/soundsQ-System/feedback.wav")
g.play
```

Este es el SynthDef para los granos. Se crea con un PLayBuf y se crean argumentos para cada parametro de modo que podamos cambiar sus valores desde afuera. La envolvente debe ser pequeña para cada grano.

```
(
SynthDef(\granulador, {|buf=0,rate=1,pos=0, amp=1, att=1,dur=0.01,rel=1,pan=0|
                var sig, env;
                sig=PlayBuf.ar(1,buf,rate,1,pos)*amp;
                env=EnvGen.kr(Env([0,1,1,0],[dur*att,dur * (1 - (att+rel)),dur*rel]),doneAction:2);
                Out.ar(0,Pan2.ar(sig,pan)*env);
                }).send(s);
);
```

b- Un Tdef que active la granulacion

Creamos variables globales para cada parametro de nuestra granulacion

```
(
~rate={1};
```

```
~pos={0};
~amp={1};
~att={1/2};
~dur={0.01};
~rel={1/2};
~pan={0};
~dens={0.01};
);
```

Creamos un Tdef que envie estos valores a los argumentos del SynthDef. Usamos la variable ~dens para el wait. Notar que el wait y la duracion del grano pueden ser diferentes, dejando asi espacios entre los granos o bien superponiendolos. De esta forma podemos crear densidades diferentes independientes del tamaño del grano.

```
(
Tdef(\granulador, {
                inf.do{
                Synth(\granulador, [\buf, g.bufnum,
                                \rate, ~rate.value,
                                \pos, ~pos.value,
                                \amp, ~amp.value,
                                \att, ~att.value,
                                \dur, ~dur.value,
                                \rel, ~rel.value,
                                \pan, ~pan.value]);
                ~dens.value.wait;
                }
                }).quant_(0);
);


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

Podemos asignar rangos de aleatoriedad para los parametros

```
(
~rate={rrand(0.9,1.1)};
~pos={44100*rrand(0.9,1.1)};
~att={rrand(1/2,1/100)};
~amp={rrand(1,0.1)};
~dur={rrand(0.001,0.1)};
```

```
~rel={rrand(1/2,1/100)}};
~pan={rrand(-0.5,0.5)};
~dens={rrand(0.1,0.001)}};
);
```

c- Un grupo de Tdef que cambien los parametros en el tiempo

////////////// RATE

Aqui se declara el Tdef que mueve los minimos y maximos del rango aleatorio del parametro de rate. Tambien indicamos una cantidad de tiempo que queremos que se tarde el Tdef en hacer el recorrido.

```
(
~rateSupIni=1;
~rateInfIni=1;

~rate={rrand(~rateSupIni,~rateInfIni)};

~rateSupFin=2;
~rateInfFin=0.5;

~rateTiempo=5;

Tdef(\rate, {var min, sup, pasoInf, pasoSup;
	min=~rateInfIni;
	sup=~rateSupIni;
	pasoInf=(~rateInfFin - ~rateInfIni)/100;
	pasoSup=(~rateSupFin - ~rateSupIni)/100;
	100.do{
		min=min+pasoInf;
		sup=sup+pasoSup;
		[min,sup].postln;
		~rate={rrand(min,sup)};
		(~rateTiempo/100).wait;
		}
	}).quant_(0);
)
Tdef(\rate).play
Tdef(\rate).stop
```

Aqui podemos indicar los nuevos minimos y maximos a los que queremos ir y el tiempo que se tomara en hacerlo

```
(
~rateInfIni=~rateInfFin;
~rateSupIni=~rateSupFin;

~rateInfFin=0.1;
~rateSupFin=0.8;

~rateTiempo=10;

Tdef(\rate).play
)
```

Lo mismo pero para el parametro de la posición

```
//////////////  POS
(
~posSupIni=44100;
~posInfIni=44100;

~pos={rrand(~posSupIni,~posInfIni)};

~posSupFin=44100*1.1;
~posInfFin=44100*0.8;

~posTiempo=5;

Tdef(\pos, {var min, sup, pasoInf, pasoSup;
        min=~posInfIni;
        sup=~posSupIni;
        pasoInf=(~posInfFin - ~posInfIni)/100;
        pasoSup=(~posSupFin - ~posSupIni)/100;
        100.do{
                min=min+pasoInf;
                sup=sup+pasoSup;
                [min,sup].postln;
                ~pos={rrand(min,sup)};
                (~posTiempo/100).wait;
```

```
                }
        }).quant_(0);
)
Tdef(\pos).play
Tdef(\pos).stop

(
~posInfIni=~posInfFin;
~posSupIni=~posSupFin;

~posInfFin=44100*0.5;
~posSupFin=44100*0.76;

~posTiempo=1;

Tdef(\pos).play
)
```

Biblografía

Collins, N. (s/a). SCCourse. Recuperado de: http://www.sussex.ac.uk/Users/nc81/courses/cm1/workshop.html

Valle, A. (2008). *The SuperCollider Italian Manual at CIRMA.* Torino: CIRMA.

Ixi Audio (2008). *SuperCollider Basics.* Recuperado de: www.ixi-audio.net

Música por computadora

Ernesto Romero y Hernani Villaseñor
Centro Multimedia 2012

Sesión 17

Delay

El delay es un objeto que trabaja con parámetros de tiempo creando repeticiones a partir de un sonido origen, este efecto se ha usado mucho en diferentes tipos de música, por ejemplo el Dub Jamaiquino. Cuando suena el sonido origen, inmediatamente después comienza una serie de repeticiones del mismo sonido las cuales sucede en un tiempo específico el cual se llama tiempo de delay, demás se establece la cantidad de repeticiones lo cual se llama *feedback.*

En SC exsiten tres diferentes tipos de *delay,* los denominados Delay, Comb y Allpass.

CombL es un objeto que genera una línea de delay con una interpolación lineal, para generar este proceso es necesario que el objeto contenga un buffer interno, en este caso esta dado por el tiempo máximo de Delay que establece la duración de un buffer interno, el parámetro de tiempo de decay es equivalente a parámetro de *feedback* de los delay hardware, y establece el número de repeticiones que genera la línea de delay.

CombL.ar/kr(entrada, tiempoMaxDelay, tiempoDelay, tiempoDecay, multiplicación, adición)

entrada: es la señal a la que aplicaremos el delay
tiempoMaxDelay: en segundos, su usa para inicializar el tamaño del buffer del delay
tiempoDelay: en segundos, duración entre cada repetición
tiempoDecay: en segundos, es el tiempo en el que el delay se va desvaneciendo

```
// intenta sustituir el objeto CombL por CombC y CombN
(
SynthDef(\Combdelay,{|amp=1|
     var sen,del;
     sen=WhiteNoise.ar(amp)*EnvGen.kr(Env.perc(0.01,0.1),Impulse.
     kr(0.25),doneAction:0);
     del=CombL.ar(sen,0.3,0.2,8);
     Out.ar(0,del);
}).send(s)
)
```

```
Synth(\Combdelay)
```

Existen otros dos delays de este tipo, CombN, que no usa interpolación y CombC que usa una interpolación cuadrada, la cual requiere de mayor procesamiento pero es más exacta.

AllpassL es un delay con una interpolación lineal, tiene los mismos argumentos que CombL

AllpassN.ar/kr(entrada, tiempoMaxDelay, tiempoDelay, tiempoDecay, multiplicación, adición)

```
// intenta sustituir el objeto AllpassL por AllpassC y AllpassN
(
SynthDef(\allpassdelay,{|amp=1|
     var sen,del;
     sen=WhiteNoise.ar(amp)*EnvGen.kr(Env.perc(0.01,0.1),Impulse.
kr(0.25),doneAction:0);
     del=AllpassL.ar(sen,0.3,0.2,8);
     Out.ar(0,del);
}).send(s)
)

Synth(\allpassdelay)
```

Igual que los delays de tipo Comb, Allpass tiene tres tipos de interpolación lineal, cuadrada y sin interpolación: AllpassL, AllpassC y AllpassN. La diferencia con los delays de tipo Comb es que Allpass manda a la salida de manera inmediata lo que está entrando por lo que su amplitud tiene a ser más baja, pero utilizado como delay no tienen mucha diferencia.

Delay es un objeto que genera ecos sin una parámetro que controla el *feedback.* Al igual que los anteriores, cuenta con tres tipos de interpolación lineal, cuadrada y no lineal: DelayL, DelayC y DelayN.

DelayL.ar/kr(entrada,tiempoMaxDelay, tiempoDelay, multiplicación, adición)

Para estos ejemplos usaremos el objeto Decay, de esta manera simulamos un tipo de feedback, el cual no contiene los objetos Delay.ar.

Decay.ar(señal, tiempoDecay, multiplicación, adición)

Ahora usamos el Decay y DelayL dentro de un Synth, para obtener un eco. En esta caso el argumento de multiplicación de Decay sirve como señal de entrada, lo mismo que el argumento de adición de DelayL.

```
// intenta sustituir el objeto DelayL por DelayC y DelayN
(
SynthDef(\delay,{|amp=1|
var sen,del;
sen=Decay.ar(Impulse.ar(0.5),0.1,WhiteNoise.ar);
del=DelayL.ar(sen,0.5,0.1,1,sen);
Out.ar(0,del);
}).send(s)
)

Synth(\delay)
```

PitchShift

Es un objeto que cambia el tono y esta basado en el uso de granos con un envolvente triangular y una densidad de 4:1.

PitchShift.ar(entrada, tamañoVentana, tonoRango, tonoDispersion, tiempoDispersion, multiplicación, adición)

entrada: señal de entrada
tamañoVentana: tamaño de la ventana del grano en segundos
tonoRango: rango del pitchshift entre 0.0 y 4.0
tonoDispersion: la máxima desvición aleatoria del del tono establecida por el tonoRango
tiempoDispersion: no puede ser mayor a tamañoVentana, es un offset aleatorio que va de 0 a toempoDispersión y se agrega al delay de cada grano.

```
(
SynthDef(\pitchshift,{|amp=0.5|
     var sen,rangoTono, ps;
     rangoTono=MouseX.kr(0.0,4.0);
     sen=Blip.ar(800, 6, amp);
     ps=PitchShift.ar(sen,0.03,rangoTono,0,0.02);
     Out.ar(0,ps);
}).send(s)
)

Synth(\pitchshift)
```

SoundIn

Con este objeto leemos el sonido que entra a través de la tarjeta de sonido o el micrófono de nuestra computadora. Así que toda señal analógica o acústica que quieras conectar a SuperCollider lo puedes hacer a través de este objeto.

Está basado en el objeto In, y toma 0 como la primer entrada disponible independientemente de que haya múltiples señales de entrada. En el caso de contar con una tarjeta nativa de la computadora, 0 equivale a la entrada de micrófono o al canal de entrada izquierdo, 1 al derecho y así sucesivamente.

SoundIn.ar(entrada, multiplicación, adición)

entrada: el canal o array de canales que serán leídos.

```
// este código activa el micrófono o canal de entrada de tu
computadora
{SoundIn.ar(0)}.play

(
SynthDef(\soundin,{|amp=0.5, gate=1, en=0|
var entrada, env;
entrada=SoundIn.ar(en,0.1);
entrada=Pan2.ar(entrada,0,1);
env=EnvGen.kr(Env.asr(0,1,0),gate,doneAction:0);
Out.ar(0,entrada*env);
}).send(s)
)

p=Synth(\soundin)
p.set(\gate,0) // apaga
p.set(\gate,1) // prende
```

Nota que doneAction esta en 0, esto permite usar el argumento \gate para prender y apagara la entrada de la señal.

In

La clase In nos permite leer señales de audio desde un bus, el cual es un tipo de canal auxiliar. Esto es muy útil cuando queremos procesar una señal independientemente de su proceso, es decir queremos la señal limpia en un Synth y el proceso en otro Sytnh. Para lograr esto mandamos la señal del sonido a el Synth de la señal de proceso, para lo que usamos una combinación entre el Out del Synth donde está la señal y la clase In en el Synth que recibiremos la señal que queremos procesar.

In.ar/kr(bus, numeroCanales)

bus: el número de bus desde donde leer la señal
numeroCanales: número de canales desde donde leer la señal

En un SynthDef el argumento *out* saca la señal por diferentes canales o buses. Al mandar la señal por el *out* 0, el sonido va a la bocina izquierda, pero si lo mandamos por el *out* 4 no sonará, al menos que tengamos una tarjeta de sonido multicanal. Podemos ver lo que hay en el *out* 4 si abrimos el scope del servidor con el argumento 5 entre paréntesis: s.scope(5).

```
(
SynthDef(\ki,{|gate=1, out=0|
     var sen,env;
     sen=WhiteNoise.ar(0.1);
     env=EnvGen.kr(Env.asr(0,1,0),gate,doneAction:2);
     Out.ar(out,sen*env)
}).send(s)
)

~ki=Synth(\ki)
~ki.set(\out, 4);
s.scope(5);
~ki.set(\out, 0);
~ki.set(\gate, 0);
```

Ahora construimos un Synth con la clase In y lo llamamos \in. La clase In recibe la señal de audio del *out* 4 y la manda por un Out al canal 0. Algo importante de destacar es que cuando llamamos al Synth(\in) debemos hacerlo con el mensaje .after. Este mensaje determina la ubicación de nuestro synth en un árbol de nodos. Ahora podemos tomar la señal que está en el bus 4 y sacarlo de nuevo al canal 0.

```
(
SynthDef(\in,{|gate=1|
var sen,env;
sen=In.ar(4);
env=EnvGen.kr(Env.asr(0,1,0),gate,doneAction:2);
Out.ar(0,sen*env)
}).send(s)
)
```

```
t=Synth.after(s,\in)
t.set(\gate,0)
```

Hagamos un ejercicio para utilizar las clases In y Out para asignar efectos a una señal. Primero hacemos un SynthDef que lea una muestra de audio de un Buffer y mandamos su señal al bus 9.

```
a=Buffer.read(s,"/sounds/voz.aiff")
a.play

(
SynthDef(\sample,{|gate=1|
var sen,env;
sen=PlayBuf.ar(1,a.bufnum,loop:1);
env=EnvGen.kr(Env.asr(0.1,0.5,0.1),gate,doneAction:2);
Out.ar(9,sen*env)
}).send(s)
)

d=Synth(\sample)
d.set(\gate,0)
```

Después creamos un SynthDef(\clean) que lea la señal del bus 9 y la saque limpia por el bus 0.

```
(
SynthDef(\limpio,{|gate=1|
var sen,env;
sen=In.ar(9);
env=EnvGen.kr(Env.asr(1,0.4,2),gate,doneAction:2);
Out.ar(0,sen*env)
}).send(s)
)

     e=Synth.after(s,\limpio)
e.set(\gate,0)
```

Un SynthDef(\efx1) leerá la señal del bus 9 pero le agregará un efecto reverb y la sacará por el bus 0.

```
(
SynthDef(\efecto,{|gate=1|
```

```
var sen,env;
sen=In.ar(9);
15.do({ sen = AllpassN.ar(sen, 0.05, [0.05.rand, 0.05.rand],
2) });
sen;
env=EnvGen.kr(Env.asr(2,0.5,2),gate,doneAction:2);
Out.ar(0,sen*env)
}).send(s)
)

t=Synth.after(s,\efecto1)
t.set(\gate,0)
```

Así, alternando entre el Synth(\limpio) y el Synth(\efecto1) podemos sacar la señal limpia o con efecto.

Aquí hacemos lo mismo pero con una señal proveniente del micrófono de la computadora usando la clase SoundIn.

```
(
SynthDef(\micro,{|gate=1|
var sen,env;
sen=SoundIn.ar(0);
env=EnvGen.kr(Env.asr(0,1,0),gate,doneAction:2);
Out.ar(9,sen*env)
}).send(s)
)

q=Synth(\micro)
q.set(\gate,0)

(
SynthDef(\limpio,{|gate=1|
var sen,env;
sen=In.ar(9);
env=EnvGen.kr(Env.asr(1,0.4,1),gate,doneAction:2);
Out.ar(0,sen*env)
}).send(s)
)
```

```
     w=Synth.after(s,\limpio)
w.set(\gate,0)

(
SynthDef(\rev,{|gate=1|
var sen, efecto, env;
sen=In.ar(9);
efecto= 15.do({ sen = AllpassN.ar(sen, 0.05, [0.05.rand,
0.05.rand], 2) });
env=EnvGen.kr(Env.asr(1,0.4,1),gate,doneAction:2);
Out.ar(0,efecto*env)
}).send(s)
)

z=Synth.after(s,\rev)
z.set(\gate,0)
s.queryAllNodes
```

Bibliografía

Ixi Audio (2008). *SuperCollider Basics.* Recuperado de: www.ixi-audio.net

Netri, E. y Romero, E. (2008). *Curso de SuperCollider Intermedios.* Centro Multimedia: México.

Música por computadora

Ernesto Romero y Hernani Villaseñor

Centro Multimedia 2012

SESIÓN 18

Forma

Rutinas

Una rutina es una programa de lectura de datos, el cual funciona de manera cíclica en una estructura llamada *loop*. Mucha de la programación de *software* está basada en esta ejecución cíclica y jerárquica de código. Es decir, se programa de manera que el código se ejecuta por pasos y una vez terminada la lectura de la programación se repiten las mismas acciones. En la programación podemos insertar instrucciones para que en cada nueva lectura del código se realicen acciones diferentes, o sea damos instrucciones para que argumentos y variables cambien en el tiempo.

En una rutina podemos establecer cuantas veces se repite una acción, generalmente un programa mantiene un *loop* infinito, también determinamos el tiempo que debe esperar la computadora para volver a leer la serie de instrucciones de código.

En SC hay varias formas de hacer rutinas, una de ellas es Tdef y Routine.

Tdef: Task Definition

Crearemos un código con una rutina que controle unos SynthDefs y que desarrolle una forma musical de manera automática. El código estará escrito de tal manera que será muy fácil de declarar y de correr. Utilizaremos el mensaje .doWhenBooted. Este mensaje realiza una acción una vez que el servidor haya sido "booteado". En este ejemplo nos servira para cargar todos los SynthDefs y los Tdefs. Usando esto en combinación con secciones de codigo comentados podemos seleccionar todo y declararlo una única vez. Con esto evitamos tener que recordar que y en que orden debemos declarar un código para que corra bien.

```
//////////////////////////////////////////////////
//                                              //
//          TAKA-TAKA Style                     //
//                                              //
//  Selecciona todo y declara                   //
//                                              //
//////////////////////////////////////////////////
```

```
/*   CONTROL

Tdef(\general).play
Tdef(\general).stop

*/

s.boot.doWhenBooted({

//////////  SynthDefs
(
(
SynthDef(\sonido, {|freq=300, amp=0.25, dur=0.5, pan=0|
                    var sig, env;
                    sig=RLPF.ar(Saw.ar(freq,amp/2),freq*16, 0.2);
                    sig=sig+SinOsc.ar(freq,0,amp);
                    sig=Limiter.ar(sig,0.9);
                    sig=Pan2.ar(sig, pan);
                    env=EnvGen.ar(Env.perc(0.01,dur),doneAction:2);
                    Out.ar(0,sig*env*0.75)
                    }).send(s)
);
/*
Synth(\sonido)
*/
(
SynthDef(\bombo, {|freq=80, amp=1.75, dur=0.235, pan=0|
                    var sig, kick, env;
                    sig=SinOsc.ar(freq*[1,1.1,1.5],pi,amp/[1,1,2]).mean;
                    kick=EnvGen.ar(Env([1,0],[0.05]))*4;
                    sig=Limiter.ar(sig, 0.9);
                    sig=Pan2.ar(sig+kick, pan);
                    env=EnvGen.ar(Env.perc(0.01,dur),doneAction:2);
                    Out.ar(0,sig*env)
                    }).send(s)
);
/*
Synth(\bombo)
*/
(
SynthDef(\tarola, {|freq=220, amp=1, dur=0.25, pan=0|
```

```
                var sig, entorchado, kick, env;
                sig=SinOsc.ar(freq*[1,1.1,1.5],pi,amp/3);
                entorchado=WhiteNoise.ar(0.25)*Line.kr(1,0,dur/4);
                kick=EnvGen.ar(Env([1,0],[0.01]));
                sig=Pan2.ar(sig+kick+entorchado, pan);
                env=EnvGen.ar(Env.perc(0.01,dur),doneAction:2);
                Out.ar(0,sig*env)
                }).send(s)
);
/*
Synth(\tarola)
*/
(
SynthDef(\hi, {|freq=1220, amp=1, dur=0.25, pan=0|
                var sig, entorchado, kick, env;
                sig=SinOsc.ar(freq*Array.rand(15,4,8.0),pi,amp/3).mean;
                entorchado=WhiteNoise.ar(0.125)*Line.kr(1,0,dur/4);
                sig=Pan2.ar(sig+entorchado, pan);
                env=EnvGen.ar(Env.perc(0.01,dur),doneAction:2);
                Out.ar(0,sig*env)
                }).send(s)
);
/*
Synth(\hi)
*/
(
SynthDef(\melodia, {|freq=300, amp=0.25, dur=0.5, pan=0|
                var sig, env;
                sig=Formant.ar(freq, Line.kr(freq*2,freq,dur/2), freq);
                sig=Limiter.ar(sig,0.9);
                sig=Pan2.ar(sig, pan);
                env=EnvGen.ar(Env([0,1,1,0],[0.1,dur -0.2,0.1]),doneAction:2);
                Out.ar(0,sig*env*0.5)
                }).send(s)
);
/*
Synth(\melodia)
*/
(
SynthDef(\ruido, {|freq=300, amp=0.25, dur=10.5, pan=0|
                var freqM,sig, env;
                freqM=SinOsc.kr(7,0,0.1,1);
                sig=RLPF.ar(Pulse.ar(freq*freqM*Array.rand(19,1,4.0)), Line.kr(freq*8,freq,dur),
0.3).mean;
```

```
                sig=Limiter.ar(sig,0.9);
                sig=Pan2.ar(sig, pan);
                env=EnvGen.ar(Env([0,1,0],[0.1,dur -0.1]),doneAction:2);
                Out.ar(0,sig*env*0.75)
                }).send(s)
);
/*
Synth(\ruido)
*/
(
SynthDef(\ruido2, {|freq=300, amp=0.25, dur=10.5, pan=0|
                var freqM,sig, env;
                sig=RLPF.ar(Blip.ar(freq*[0,4,7,12,16,19,24].midiratio,6), Line.kr(freq,freq*8,dur),
0.3).mean;
                sig=Limiter.ar(sig*LFPulse.kr(8,0.1),0.9);
                sig=Pan2.ar(sig, pan);
                env=EnvGen.ar(Env([0,1,0],[0.1,dur -0.1]));
                sig=FreeVerb.ar(sig*env,0.5,0.979);
                Out.ar(0,sig)
                }).send(s)
);
/*
Synth(\ruido2)
*/
);

//////////  Tdefs

(
~bpm=120;
~tempo=60/~bpm;

(
Tdef(\rutina1, {
        inf.do{
        4.do{
                Synth(\sonido, [\freq, 300]);
                (~tempo/2).wait;
                };
        4.do{
                Synth(\sonido, [\freq, 300]);
                Synth(\sonido, [\freq, 300*4.midiratio]);
                (~tempo/2).wait;
                };
```

```
	4.do{
		Synth(\sonido, [\freq, 300* -10.midiratio/2, \amp, 1]);
		(~tempo/2).wait;
		};
	4.do{
		Synth(\sonido, [\freq, 300* -5.midiratio/2, \amp, 1]);
		Synth(\sonido, [\freq, 300* -1.midiratio/2, \amp, 1]);
		(~tempo/2).wait;
		};
		}
	}).quant_(0);
);
/*
Tdef(\rutina1).play;
Tdef(\rutina1).stop;
*/
(
Tdef(\rutina2, {
	inf.do{
	2.do{
		Synth(\sonido, [\freq, 300 * -8.midiratio/2, \dur, 1, \amp, 1]);
		(~tempo).wait;

		Synth(\sonido, [\freq, 300 * -10.midiratio/2, \amp, 1]);
		(~tempo/2).wait;
		Synth(\sonido, [\freq, 300 * -12.midiratio/2, \amp, 1]);
		(~tempo/2).wait;
		};

		Synth(\sonido, [\freq, 300* -5.midiratio]);
		(~tempo/2).wait;
		Synth(\sonido, [\freq, 300* -1.midiratio]);
		(~tempo/2).wait;
		Synth(\sonido, [\freq, 300* 7.midiratio]);
		(~tempo/2).wait;
		Synth(\sonido, [\freq, 300* 11.midiratio]);
		(~tempo/2).wait;

		Synth(\sonido, [\freq, 300* 2.midiratio]);
		(~tempo/2).wait;
		Synth(\sonido, [\freq, 300* 5.midiratio]);
		(~tempo/2).wait;
		Synth(\sonido, [\freq, 300* 11.midiratio]);
		(~tempo/2).wait;
```

```
                Synth(\sonido, [\freq, 300* 14.midiratio]);
                (~tempo/2).wait;
                }
        }).quant_(0);
);
/*
Tdef(\rutina2).play;
Tdef(\rutina2).stop;
*/
(
Tdef(\bataca, {
        inf.do{
        2.do{Synth(\bombo);
                Synth(\hi);
                (~tempo/2).wait;
                };
        Synth(\tarola);
        Synth(\hi);
        (~tempo/2).wait;
        Synth(\hi);
        (~tempo/2).wait;
                }
        }).quant_(0);
);
/*
Tdef(\bataca).play;
Tdef(\bataca).stop;
*/
(
Tdef(\rutina3, {
        inf.do{
        2.do{
        4.do{
                Synth(\sonido, [\freq, 300 * -1.midiratio]);
                (~tempo/4).wait;
                Synth(\sonido, [\freq, 300 * -2.midiratio]);
                (~tempo/4).wait;
                };

        4.do{
                Synth(\sonido, [\freq, 300 * 2.midiratio]);
                (~tempo/4).wait;
                Synth(\sonido, [\freq, 300 * 1.midiratio]);
                (~tempo/4).wait;
```

```
                };
                };

                2.do{
        4.do{
                Synth(\sonido, [\freq, 300 * (-1+2).midiratio]);
                (~tempo/4).wait;
                Synth(\sonido, [\freq, 300 * (-2+2).midiratio]);
                (~tempo/4).wait;
                };

        4.do{
                Synth(\sonido, [\freq, 300 * (2+2).midiratio]);
                (~tempo/4).wait;
                Synth(\sonido, [\freq, 300 * (1+2).midiratio]);
                (~tempo/4).wait;
                };
                };

                }
        }).quant_(0);
);
/*
Tdef(\rutina3).play;
Tdef(\rutina3).stop;
*/
(
Tdef(\rutina4, {
        inf.do{
        2.do{
        4.do{
                Synth(\sonido, [\freq, 300 * -4.midiratio/2]);
                (~tempo/4).wait;
                Synth(\sonido, [\freq, 300 * -5.midiratio/2]);
                (~tempo/4).wait;
                };

        4.do{
                Synth(\sonido, [\freq, 300 * -7.midiratio]);
                (~tempo/4).wait;
                Synth(\sonido, [\freq, 300 * -8.midiratio]);
                (~tempo/4).wait;
                };
                };
```

```
		2.do{
	4.do{
		Synth(\sonido, [\freq, 300 * (-4+2).midiratio/2]);
		(~tempo/4).wait;
		Synth(\sonido, [\freq, 300 * (-5+2).midiratio/2]);
		(~tempo/4).wait;
		};

	4.do{
		Synth(\sonido, [\freq, 300 * (-7+2).midiratio]);
		(~tempo/4).wait;
		Synth(\sonido, [\freq, 300 * (-8+2).midiratio]);
		(~tempo/4).wait;
		};
		}

		}
	}).quant_(0);
);
/*
Tdef(\rutina4).play;
Tdef(\rutina4).stop;
*/
(
Tdef(\bataca2, {
	inf.do{
		Synth(\bombo);
		Synth(\hi);
		(~tempo/2).wait;

		Synth(\tarola);
		Synth(\hi);
		(~tempo/2).wait;

		Synth(\bombo);
		Synth(\hi);
		(~tempo/4).wait;

		Synth(\bombo);
		(~tempo/4).wait;

		Synth(\tarola);
		Synth(\hi);
```

```
                (~tempo/2).wait;


                }
        }).quant_(0);
);
/*
Tdef(\bataca2).play;
Tdef(\bataca2).stop;
*/
(
Tdef(\rutinaM, {var wait;
        (~tempo*4*2).wait;
        inf.do{
                wait=(~tempo/[0.25,0.5,1,2].choose);
                Synth(\melodia, [\freq, 300 * [0,2,4,5,7,9,11,12].choose.midiratio*1, \dur, wait]);
                wait.wait;


                }
        }).quant_(0);
);
/*
Tdef(\rutinaM).play;
Tdef(\rutinaM).stop;
*/

/*
Tdef(\rutina4).play;
Tdef(\rutina4).stop;

Tdef(\bataca).play;
Tdef(\bataca).stop;


(
Tdef(\rutina1).play;
Tdef(\rutina2).play;
)
(
Tdef(\rutina1).stop;
Tdef(\rutina2).stop;
)
```

```
(
Tdef(\rutina3).play;
Tdef(\rutina4).play;
)
(
Tdef(\rutina3).stop;
Tdef(\rutina4).stop;
)

(
Tdef(\rutinaM).play;
Tdef(\rutina1).play;
Tdef(\rutina2).play;
Tdef(\rutina3).stop;
Tdef(\rutina4).stop;
Tdef(\bataca).play;
Tdef(\bataca2).stop;
)
(
Tdef(\rutinaM).stop;
Tdef(\rutina3).play;
Tdef(\rutina4).play;
Tdef(\rutina1).stop;
Tdef(\rutina2).stop;
Tdef(\bataca2).play;
Tdef(\bataca).stop;
)


(
Tdef(\rutina1).stop;
Tdef(\rutina2).stop;
Tdef(\rutina3).stop;
Tdef(\rutina4).stop;
Tdef(\bataca).stop;
Tdef(\bataca2).stop;
Tdef(\rutinaM).stop;
)
*/

(
Tdef(\general, {
```

```
(
Tdef(\rutinaM).play;
Tdef(\rutina1).play;
Tdef(\rutina2).play;
Tdef(\rutina3).stop;
Tdef(\rutina4).stop;
Tdef(\bataca).play;
Tdef(\bataca2).stop;
);

(~tempo*(8*6)).wait;

(
Synth(\ruido);
Tdef(\rutinaM).stop;
Tdef(\rutina3).play;
Tdef(\rutina4).play;
Tdef(\rutina1).stop;
Tdef(\rutina2).stop;
Tdef(\bataca2).play;
Tdef(\bataca).stop;
);

(~tempo*(4*8)).wait;

(
Tdef(\rutinaM).play;
Tdef(\rutina1).play;
Tdef(\rutina2).play;
Tdef(\rutina3).stop;
Tdef(\rutina4).stop;
Tdef(\bataca).play;
Tdef(\bataca2).stop;
);

(~tempo*(4*8)).wait;

(
Synth(\ruido2);
Tdef(\rutina1).stop;
Tdef(\rutina2).stop;
Tdef(\rutina3).stop;
Tdef(\rutina4).stop;
Tdef(\bataca).stop;
```

```
Tdef(\bataca2).stop;
Tdef(\rutinaM).stop;
);
}).quant_(0);
);
/*
Tdef(\general).play
Tdef(\general).stop
*/
)

// fin del doWhenBooted
})
```

Aleatoriedad y Estocástica

Aleatorio es todo aquello que tiene que ver con el azar. Lo aleatorio abarca lo probabilístico y lo no probabilístico.
Estocástico o probabilístico se refiere a aquellos eventos que tienen una probabilidad determinada, especialmente interesantes son las combinaciones de eventos cuyas probabilidades son distintas.

Aquí tenemos un ejemplo de aleatoriedad. Los eventos tienen la misma probabilidad de que ocurran porque estamos usando el mensaje choose. Tenemos tres frecuencias, la probabilidad de cada una es de ⅓.

```
~freq=[10,500,1500];

(
Tdef(\prob, {
          inf.do{
                    Synth(\ruido, [\freq, ~freq.choose.postln]);
                    1.wait;
                    }
          }).quant_(0);
);
/*
Tdef(\prob).play;
Tdef(\prob).stop;
*/
```

Podemos asignar diferentes probabilidades a las tres distintas frecuencias con el mensaje wchoose. La w significa weight. Esto es el peso probabilístico. A cada objeto de un array le asignamos un peso. El total de los pesos deben de sumar uno.

```
[10,500,1500].wchoose([0.7,0.2,0.1])
```

En la línea anterior el 10 tiene un peso de 0.7. Esto es equivalente a decir que hay un 70% de probabilidad de que se escoja. El 500 tiene un peso de 0.2 y el 1500 un peso de 0.1.

```
(
Tdef(\prob, {
          inf.do{
                    Synth(\ruido, [\freq,
~freq.wchoose([0.7,0.2,0.1]).postln]);
                    1.wait;
                    }
          }).quant_(0);
);
/*
Tdef(\prob).play;
Tdef(\prob).stop;
*/
```

Con el mensaje normalizaSum podemos hacer que los elementos de un arreglo sumen uno y de esta forma podemos usarlos como pesos en un wchoose.
~pesos=[1000,100,2].normalizeSum

Podemos cambiar en tiempo real los valores de los pesos

```
~pesos=[100,100,2000].normalizeSum
~pesos=[1000,1000000,2].normalizeSum

(
Tdef(\prob, {
          inf.do{
```

```
                    Synth(\ruido, [\freq,
[20,~f500.value,1500].wchoose(~pesos).postln]);
                    2.wait;
                    }
          }).quant_(0);
);
```

También podemos usar el mensaje coin para determinar una probabilidad de que occurra un evento. Funciona como una moneda que se lanza al aire pero que puede tener diferentes probabilidades de que caiga cada lado. También funciona con números del 0 al 1. El mensaje coin solo nos arroja true o false como los lados de la moneda. El número al que le enviemos el mensaje coin determinará la probabilidad de que caiga en true.

```
1.coin  //  Siempre nos arrojará true
0.5.coin // 50% de probabilidad de que salga true
0.1.coin // 10% de probabilidad de que salga true
```

Podemos usar coin para determinar una condición

```
if(0.2.coin, {Synth(\ruido)}, {"no salio".postln})
```

Podemos ver un ejemplo del uso de coin con la batería del Tdef(\bataca). Este Tdef es muy repetitivo, pero podemos agregarle un caractér improvisatorio si dejamos que la tarola suene a veces. El 10% de las veces, por ejemplo

```
(
Tdef(\bataca, {
               inf.do{
               2.do{Synth(\bombo);
                    Synth(\hi);
                    (~tempo/2).wait;
                    };
               Synth(\tarola);
               Synth(\hi);
               (~tempo/2).wait;
               Synth(\hi);
               (~tempo/2).wait;
                    }
               }).quant_(0);
);
```

```
(
Tdef(\tarolaCoin, {
                inf.do{
                if(0.1.coin,{Synth(\tarola);});
                (~tempo/4).wait;
                     }
                }).quant_(0);
);




/*
(
Tdef(\tarolaCoin).play;
Tdef(\bataca).play;
)
(
Tdef(\tarolaCoin).stop;
Tdef(\bataca).stop;
)
*/
```

Música por computadora

Ernesto Romero y Hernani Villaseñor
Centro Multimedia 2012

Sesión 19

Secuenciadores

Un secuenciador es un dispositivo en el que acomodamos eventos sonoros en dos ejes: altura y tiempo. El secuenciador viene de la tecnología de los primero sintetizadores, y toma como modelo la estructura musical, entonces la altura está basada en tonos y el tiempo en ritmo musical.

Originalmente los secuenciadores funcionaban a partir de pasos, o sea escaños donde acomodar en evento, la distancia entre cada escaño era simétrica de tal forma que con 16 pasos se formaba un compás de 4 cuartos subdividido en 4 dieciseisavos pr cuarto, o sea 16 pasos.

Posteriormente con el protocolo MIDI, esta resolución de 16 pasos se hizo más compleja, y la posibilidad de acomodar alturas quedó determinada por los 8 bits del protocolo, lo que dió 128 lugares para acomodar notas, más de las que tiene un piano convencional, así las notas se asignaron a un número del 0 al 127 donde 60 es el do central y cada número hacia arriba o hacia abajo representa un semitono.

```
s.boot
(
SynthDef(\gacho, {|freq=100, dur=0.25, amp=0.5, add=1, fmFreq=0, d=0, out=60, pan=0|
		var sig, env;

sig=Mix(RLPF.ar(Saw.ar(freq*[1,1.005]*LFNoise1.kr([0.01,0.011],0.01,1)*SinOsc.kr(fmFreq,0,d,a
dd),amp),LFNoise2.kr(0.013,5000,12000),0.01));
		sig=LPF.ar(sig,1200,amp*2)+SinOsc.ar(freq*[1,16],0,amp*[0.75,0.025]);
		env=EnvGen.kr(Env.perc(0.02,dur),doneAction:2);
		Out.ar(out,Pan2.ar(sig, pan)*env);
		}).send(s);
);
```

```
/*
Synth(\gacho, [\dur, 10])
*/
(
SynthDef(\gachoRev, {|mix=0.5, room=0.7, gate=1|
		var sig, env;
		sig=FreeVerb.ar(In.ar(60,2),mix,room);
		env=EnvGen.kr(Env.asr(0.01,1,4),gate,doneAction:2);
		Out.ar(0,sig*env);
		}).send(s);
);
/*
~gachoRev=Synth(\gachoRev, [\mix, 0.8815, \room, 0.865419], addAction: \addAfter)
~gachoRev.set(\mix, 0.8815, \room, 0.865419)
*/



~notas=[0,2,3,2,5,3,7,12,7,3,5,5,3,2]
~ritmo=[0,2,5,7,8]
```

Se puede hacer un secuenciador también con el método switch dentro de un Tdef. Usamos el argumento i del inf.do para crear un contador y lo limitamos a 8 pasos con el modulo %8. Escogemos despues en que numero de la cuenta habrá un evento y lo encerremos en una función.

```
(
Tdef(\secSwitch, {
		inf.do{|i|
		switch(i.asInteger%8, 0, {Synth(\gacho, [\dur, 0.1, \freq, 1])},
				2, {Synth(\gacho, [\dur, 0.1, \freq, 55])},
				3, {Synth(\gacho, [\dur, 0.1])},
				5, {Synth(\gacho, [\dur, 0.1, \freq, 100* -5.midiratio])}
			);
		0.2.wait;
		}
		}).quant_(0)
)

(
Tdef(\secSwitch).play;
Tdef(\secSwitch).stop
)
```

Podemos meter varios switch dentro del mismo Tdef usando diferentes módulos.

```
(
Tdef(\secSwitch, {
		inf.do{|i|
		switch(i.asInteger%8, 0, {Synth(\gacho, [\dur, 0.1, \freq, 1])},
				2, {Synth(\gacho, [\dur, 0.1, \freq, 55])},
				3, {Synth(\gacho, [\dur, 0.1])},
				5, {Synth(\gacho, [\dur, 0.1, \freq, 100* -5.midiratio])}
			);

		switch(i.asInteger%16, 0, {Synth(\gacho, [\dur, 0.1, \freq, 100* 12.midiratio])},
				4, {Synth(\gacho, [\dur, 0.1, \freq, 100* 11.midiratio])},
				7, {Synth(\gacho, [\dur, 0.1, \frec, 100* 3.midiratio])},
				11, {Synth(\gacho, [\dur, 0.1, \freq, 100* 2.midiratio])},
				12, {Synth(\gacho, [\dur, 0.1, \freq, 100* 1.midiratio])},
				14, {Synth(\gacho, [\dur, 0.1, \freq, 2])},
				15, {Synth(\gacho, [\dur, 0.1, \freq, 100* 13.midiratio])}
			);

		0.2.wait;
		}
		}).quant_(0)
)

(
Tdef(\secSwitch).play;
Tdef(\secSwitch).stop
)
```

Otro ejemplo con desplazamientos rítmicos usando módulo 7 y 8.

```
(
Tdef(\secSwitch, {
		inf.do{|i|
		switch(i.asInteger%8, 0, {Synth(\gacho, [\dur, 0.1, \freq, 100* -5.midiratio])},
				2, {Synth(\gacho, [\dur, 0.1, \freq, 100* -4.midiratio])},
				3, {Synth(\gacho, [\dur, 0.1, \freq, 100* -3.midiratio])},
				5, {Synth(\gacho, [\dur, 0.1, \freq, 100* 0.midiratio])}
			);

		switch(i.asInteger%7, 0, {Synth(\gacho, [\dur, 0.1, \freq, 100* 12.midiratio])},
				1, {Synth(\gacho, [\dur, 0.1, \freq, 100* 11.midiratio])},
				4, {Synth(\gacho, [\dur, 0.1, \frec, 100* 3.midiratio])},
```

```
                        6, {Synth(\gacho, [\dur, 0.1, \freq, 100* 2.midiratio])}
                    );

                0.2.wait;
                }
                }).quant_(0)
)

(
Tdef(\secSwitch).play;
Tdef(\secSwitch).stop
)
```

Otra técnica, mas poderosa es hacer arrays con valores específicos para cada evento. En el siguiente ejemplo tenemos un array cuyos elementos son arrays con pares de valores para altura y duración.

```
~sec=[[0,1/16],[0,1/16],[0,1/16],[0,1/16], [12,1/4],[0,1/8],[0,1/8], [12,1/4]];
~fund=60.midicps;

(
Tdef(\secuenciador, {var altura, dur;
                inf.do{|i|
                        altura=~sec[i%~sec.size][0].midiratio;
                        dur=~sec[i%~sec.size][1]*2;

                        Synth(\gacho, [\freq, altura*~fund, \dur, dur]);

                        dur.wait;
                        }
                }).quant_(0);
);
/*
Tdef(\secuenciador).play;
Tdef(\secuenciador).stop;
*/

~sec2=[[8,1/2],[8,1/4],[7,1/4],[5,1], [7,2],[8,1/2],[8,1/4],[7,1/4],[5,1], [0,2]];
~fund=60.midicps;

(
Tdef(\secuenciador2, {var altura, dur;
```

```
		inf.do{|i|
			altura=~sec2[i%~sec2.size][0].midiratio;
			dur=~sec2[i%~sec2.size][1]*2;

			Synth(\gacho, [\freq, altura*~fund/4, \dur, dur*2]);

			dur.wait;
			}
		}).quant_(0);
);
/*
Tdef(\secuenciador2).play;
Tdef(\secuenciador2).stop;
*/

/*
(
Tdef(\secuenciador).play;
Tdef(\secuenciador2).play;
)
(
Tdef(\secuenciador).stop;
Tdef(\secuenciador2).stop;
)
*/
```

Se pueden agregar también valores especiales para eventos en particular. El siguiente array incluye el string "f" en algunos de sus elementos. El Tdef usa una condicionante if para detectar si existe una modulación.

```
~sec=[[-10,1/16],[0,1/16],[-9,1/16],[0,1/16], [12,1/4, "f"],[1,1/8],[0,1/8], [24,1/4, "f"]];
~fund=60.midicps;

(
Tdef(\secuenciador, {var altura, dur;
		inf.do{|i|
			altura=~sec[i%~sec.size][0].midiratio*~fund;
			dur=~sec[i%~sec.size][1]*4;
			if(~sec[i%~sec.size][2]==nil,{
			Synth(\gacho, [\freq, altura, \dur, dur*0.1, \amp, 0.4, \pan, 0.5]);
					},
					{
			Synth(\gacho, [\freq, altura, \dur, dur, \fmFreq, 8, \d, 0.5, \pan, 0.5]);
					}
```

```
                        );
                        dur.wait;
                        }
                }).quant_(0);
);
/*
Tdef(\secuenciador).play;
Tdef(\secuenciador).stop;
*/
```

Ahora veamos como crear arrays independientes de alturas y duraciones para después unirlos en un array que los agrupe por pares en arrays más pequeños de altura y duración

```
~notas=Array.rand(12,0,12);
~dur=Array.rand(12,0,1.0).normalizeSum*4;
~sec=[~notas,~dur].flop;

/*
Tdef(\secuenciador).play;
Tdef(\secuenciador).stop;
*/
```

Agreguemos la posibilidad de hacer cambios en el tempo mediante una cifra metronómica bmp (bits por minuto).

```
~bpm=60

(
Tdef(\secuenciador, {var altura, dur;
                inf.do{|i|
                        altura=~sec[i%~sec.size][0].midiratio*~fund;
                        dur=~sec[i%~sec.size][1]*((60*4)/~bpm);
                        Synth(\gacho, [\freq, altura, \dur, dur*2, \amp, 0.65, \pan, -0.5]);
                        dur.wait;
                        }
                }).quant_(0);
);
/*
Tdef(\secuenciador).play;
Tdef(\secuenciador).stop;
*/
```

Y por último agreguemos otra voz con otro secuenciador.

```
~notas2=Array.rand(12,0,12) -12;
~dur2=Array.rand(12,0,1.0).normalizeSum*16;
~sec2=[~notas2,~dur2].flop;
(
Tdef(\secuenciador2, {var altura, dur;
                inf.do{|i|
                        altura=~sec2[i%~sec2.size][0].midiratio*~fund;
                        dur=~sec2[i%~sec2.size][1]*((60*4)/~bpm);
                        Synth(\gacho, [\freq, altura/2, \dur, dur*2, \amp, 0.65, \pan, -0.5]);
                        dur.wait;
                        }
                }).quant_(0);
);
/*
Tdef(\secuenciador2).play;
Tdef(\secuenciador2).stop;
*/

/*
(
Tdef(\secuenciador).play;
Tdef(\secuenciador2).play;
)
(
Tdef(\secuenciador).stop;
Tdef(\secuenciador2).stop;
)
(
~dur=[1/16!8, 1/8!4, 1/4!2,1/2].flat.scramble;
~notas=Array.rand(~dur.size,0,12);
~sec=[~notas,~dur].flop;

~dur2=[1/16!8, 1/8!4, 1/4!2,1/2].flat.scramble;
~notas2=Array.rand(~dur2.size,0,12) -12;
~sec2=[~notas2,~dur2].flop;
)
*/
```

Máquina de Estados Finitos

Es un modelo de comportamiento de un sistema donde la señal de salida depende de una señal de entrada más sus estados anteriores.

La máquina de estados finitos Se compone de una serie de estados. En cada estado se tienen diferentes opciones para moverse a otros estados. Por ejemplo, imaginemos que tenemos 4 estados en los que podemos estar. Partimos de un estado inicial, sea el 0. Del 0 podríamos ir al estado 1 o al 2. Si vamos al estado 1 podemos ir al estado 2. Si vamos al estado 2 encontramos que de ahi podriamos ir de regreso al estado 0 o ir al estado 3. Si estamos en el estado 3 ya acabamos.

Empiezas en 0
Si estas en 0 puedes ir a 1 o 2
Si estas en 1 puedes ir a 2
Si estas en 2 puedes ir a 0 o 3
Si estas en 3 ya acabaste
Hacer el recorrido una vez

Este es solo un caso hipotético. La cantidad de estados y las opciones a donde se puede dirigir en cada estado son determinadas por quien diseña la máquina.

En SC existe el patrón llamado Psfm *Pattern Finite State Machine*, para llevar acabo este tipo de secuencias.

Pfsm(lista, repeticiones)

Cada estado consiste de un artículo y un Array de índices enteros de posibles estados próximos. El artículo inicial se escoge de manera aleatoria del Array que representa los estados de entrada, ese estado que acaba de ser escogido se regresa y el siguiente estado es escogido de los posibles estados próximos. Cuando el último estado es escogido, ahí termina el flujo de estados.

```
Pfsm([
        #[estados iniciales posibles],
        estado_1, #[estados a los que se puede ir],
        estado_2, #[estados a los que se puede ir],
        estado_3, #[estados a los que se puede ir],
        .
        .
        .
        ultimo_estado, nil
        ],
```

```
	cantidad_de_repeticiones
	)
```

Hagamos un ejemplo lineal para empezar. En este caso solo podemos empezar en el estado 0. Del estado 0 solo podemos ir al 1, del 1 solo al 2 y del 2 solo al 3.

```
~mef=Pfsm(
	[
	#[0],
	0, #[1],
	1, #[2],
	2, #[3],
	3, nil
	],1).asStream
```

Necesitamos mandar el mensaje .asStream al Pfsm para poder pedir sus resultados. Para pedir los resultados usamos el mensaje .next

```
~mef.next
```

Si declaramos varias veces ~mef.next nos arrojará los números 0, 1, 2 y 3. Después de esto nos marcará error por que el Pfsm ha terminado de hacer su recorrido y solo le pedimos que lo hiciera una vez.

Ahora hagamos el caso hipotético que planteamos al inicio.

Esto:

Empiezas en 0
Si estas en 0 puedes ir a 1 o 2
Si estas en 1 puedes ir a 2
Si estas en 2 puedes ir a 0 o 3
Si estas en 3 ya acabaste
Hacer el recorrido una vez

Se convierte en esto:

```
~mef=Pfsm(
	[
	#[0],
	0, #[1, 2],
	1, #[2],
```

```
2, #[0, 3],
3, nil
],1).asStream
```

La máquina de estados finitos es perfecta para establecer recorridos armónicos obedeciendo las reglas cadenciales de la armonía clásica. Estas reglas en palabras son:

Tenemos los siguientes acordes:

i = primer grado
ii = segundo grado
iv = cuarto grado
v = quinto grado
vi = sexto grado

Las reglas de conducción cadencial son las siguientes:

Empiezas en i
Si estas en i puedes ir a ii, iv o a v
Si estas en ii puedes ir a v
Si estas en iv puedes ir a i o a v
Si estas en v puedes ir a i, vi
Si estas en vi puedes ir a iv

Implementado en un Pfsm queda así:

```
~mef=Pfsm(
[
#[0],
1, #[1, 2, 3], // 0
2, #[3],// 1
4, #[0, 3], // 2
5, #[0, 4], // 3
6, #[2], // 4
],inf).asStream

(
SynthDef(\cadencias, {|freq=#[200,300,400,450], gate=1|
var sig, env;
sig=SinOsc.ar(freq,0,1/3).mean;
env=EnvGen.kr(Env.asr(0.1,1,2),gate, doneAction:2);
Out.ar(0,sig*env);
}).send(s);
)
```

```
(
SynthDef(\cadencias2, {|freq=200, gate=1|
var sig, env;
sig=SinOsc.ar(freq,0,1/3);
env=EnvGen.kr(Env.asr(0.1,1,2),gate, doneAction:2);
 Out.ar(0,sig*env);
}).send(s);
)

~synth=Synth(\cadencias, [\freq, [0,4,7,12].midiratio*200])
```

Con un Tdef evaluamos el ~mef.next y definimos diferentes funciones para cada estado. en estas funciones estamos estableciendo las notas de los diferentes acordes.

```
~wait=1;

(
Tdef(\cadencias, {
inf.do{
switch(~mef.next.postln, 1, {~acorde=[0 -12,4,7,12]},
2, {~acorde=[2 -12,5,9,14]},
 4, {~acorde=[-7 ,5,9,12]},
 5, {~acorde=[-5 -12,5,7,11]},
 6, {~acorde=[-3 -12,4,9,12]}
);
~synth.set(\freq, ~acorde.midiratio*200);

~wait.wait;
}}).quant_(0);
)

Tdef(\cadencias).play
Tdef(\cadencias).stop

~synth.set(\gate, 0)

~synth2=Synth(\cadencias2, [\freq, [0,2,4,5,7,9,11,12].choose.midiratio*200])
~synth2.set(\freq, [0,2,4,5,7,9,11,12].choose.midiratio*200)
~synth2.set(\gate, 0)

(
Tdef(\melodia, {
inf.do{
```

```
~synth2.set(\freq, [0,2,4,5,7,9,11,12].choose.midiratio*200);

(~wait*[1/4,1/2,1].choose).wait;
}}).quant_(0);
)

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

Música por computadora

Ernesto Romero y Hernani Villaseñor
Centro Multimedia 2012

Sesión 20

Breve contexto de SC en la interdisciplina con nuevos medios

La interdisciplina podríamos entenderla como el entrecruce de diferentes disciplinas del arte que enriquecen las diferentes prácticas artísticas, así, el código ha sido un detonador en este cruce constante entre disciplinas artísticas y podríamos decir que también con la ciencia y tecnología. Podemos mencionar el desarrollo de proyectos audiovisuales, el uso de sensores como extenciones y nodos entre artes escénicas, música y artes visuales.

Análisis de datos

Existen innumerables maneras de analizar un grupo de datos. Lo interesante es determinar el algoritmo que nos realiza el análisis que deseamos. Aqui veremos un ejemplo de un algoritmo que realiza a un grupo de 100 números enteros el análisis siguiente:

1 - Determina que números son menores que 30 y los agrupa en un subconjunto A.
2 - Determina que números son mayores que 30 y menores que 60 y los agrupa en un subconjunto B.
3 - Determina que números son mayores que 60 y los agrupa en un subconjunto C.
4 - Determina cuantas veces se repite cada elemento de cada subconjunto.

```
// Primero creamos un conjunto ~u con 100 números enteros
aleatorios entre 0 y 100

~u=Array.rand(100,0,100);

// Creamos los subconjuntos vacíos ~a, ~b y ~c.

~a=[];
~b=[];
~c=[];
```

Con el siguiente algoritmo agrupamos en ~a los números menores que 30, en ~b los mayores que 30 y menores que 60, y en ~c los mayores que 60.

```
(

1
```

```
~u.size.do{|i|
          if(~u[i]<=30,{~a=~a.add(~u[i])});
          if((~u[i]>30)&&(~u[i]<=60),{~b=~b.add(~u[i])});
          if(~u[i]>60,{~c=~c.add(~u[i])});
          }
)

// Podemos ver cómo quedan agrupados los números en subconjuntos
después del primer análisis.

~a;
~b;
~c;

// También podemos visualizar una gráfica de cada subconjunto
para más padre.

~a.plot:
~b.plot;
~c.plot;
```

A continuación tenemos un algoritmo que determina cuántas veces está presente cada número. Pero primero observemos los mensajes .sort y .indicesOfEqual que se usan en el algoritmo:

```
// .sort nos pone en orden ascendente los números de un array.
Si declaramos:

[2,1,3,1,0,3].sort

// Obtenemos

[0,1,1,2,3,3]
```

.indicesOfEqual nos dice en qué indices está presente un elemento dentro de un array

Si queremos saber en qué indices se encuentra presente el número 1 dentro del array [0,1,1,2,3,3] declaramos:
[0,1,1,2,3,3].indicesOfEqual(1);

```
// Y obtenemos:
```

```
[1,2]

// Para saber la cantidad de veces que el número 1 se encuentra
en el array [0,1,1,2,3,3] hacemos esto:

[0,1,1,2,3,3].indicesOfEqual(1).size;
```

Ahora sí, el siguiente algoritmo determina cuántas veces está cada número en al subconjunto ~a y nos genera 2 arreglos: el arreglo ~elementosA con los elementos de ~a en orden y sin repetir y otro arreglo ~incidenciasA con la cantidad de veces que esta cada elemento en ~a.

```
(
var array=~a, elemento_anterior=nil;
~elementosA=[];
~incidenciasA=[];
array.size.do{|i|
          if(array.sort[i]!=elemento_anterior,
               {
               ~elementosA=~elementosA.add(array[i]);


~incidenciasA=~incidenciasA.add(array.indicesOfEqual(array[i]).s
ize);
               });
          elemento_anterior=array[i];
          };

);
(
~elementosA.postln;
~incidenciasA
);

// Lo mismo para ~b


(
var array=~b, elemento_anterior=nil;
~elementosB=[];
```

```
~incidenciasB=[];
array.size.do{|i|
          if(array.sort[i]!=elemento_anterior,
               {
               ~elementosB=~elementosB.add(array[i]);


~incidenciasB=~incidenciasB.add(array.indicesOfEqual(array[i]).s
ize);
               });
          elemento_anterior=array[i];
          };

)

(
~elementosB.postln;
~incidenciasB
)

// Y para ~c

(
var array=~c, elemento_anterior=nil;
~elementosC=[];
~incidenciasC=[];
array.size.do{|i|
          if(array.sort[i]!=elemento_anterior,
               {
               ~elementosC=~elementosC.add(array[i]);

~incidenciasC=~incidenciasC.add(array.indicesOfEqual(array[i]).s
ize);
               });
          elemento_anterior=array[i];
          };
)
(
~elementosC.postln;
~incidenciasC
```

```
)
```

Mapeo

Por mapeo entendemos la traducción de información de parámetros distintos, que por su cualidad informática están representados en números, pero que en su salida representan diferentes campos de acción. Es decir una frecuencia es representada en un número que puede ser traducido a un canal de color RGB, ambos son representados en números pero sus rangos son distintos. Aplicando una técnica de mapeo podemos transponer el rango de frecuencia del oído humano -20 a 20,000 Hz- a un canal de color del sistema RGB -0 a 255-. Esto es sencillo dado que dentro de la computadora estos números tienen el mismo comportamiento y son fácilmente escalables, toman sentido de ser frecuencia o color solo en su salida.

MouseX.kr / MouseY.kr(valor min, valor max, envolvimiento, intervalo)
Ugen del cursor, en plano vertical y horizontal de la pantalla.

valMin, valMax: rango de valores entre izquierda y derecha de la pantalla.
intervalo: factor del intervalo.
envolvimiento: curva del mapeo. 0 es lineal, 1 es exponencial.
intervalo: factor de intervalo, sirve para .. el movimiento del cursor.

MouseButton.kr(valMin, ValMax, intervalo)
Ugen del botón del ratón.
valMin: valor cuando el botón no está presionado.
valMax: valor cuando el botón está presionado.
intervalo: factor del intervalo.

```
(
SynthDef(\mapeo,{|out=0|
     var sen, env, mouseX, mouseY, mouseB;
     mouseX=MouseX.kr(300,3000,1);
     mouseY=MouseY.kr(1.0,0.001,1);
     mouseB=MouseButton.kr(0,1,0.1);
     sen=FSinOsc.ar(mouseX,0,mouseY);
     env=EnvGen.kr(Env.perc(0.1,3),mouseB,doneAction:0);
     Out.ar(out,sen*env)
     }).send(s)
)
a=Synth(\mapeo)
a.free
```

Otros recursos son los mensajes .linlin y .linexp que interpolan datos de manera lineal y exponencial.

.linlin(entradaMin, entradaMax, salidaMin, salidaMax, clip)
Envuelve los datos de tal manera que un rango lineal de entrada es mapeado a un rango lineal de salida.

clip: puede ser uno de estos 4: nil, no hay clip, \minmax hay clip en el mínomo y máximo de salida, \min hay clip en el mínimo de salida, \max hay clip en el máximo de salida.

.linexp(entradaMin, entradaMax, salidaMin, salidaMax, clip)
Envuelve los datos de tal manera que un rango lineal de entrada es mapeado a un rango exponencial de salida. Hay que tener cuidado que el mínimo y máximo de salida no sean menores de cero y que sean del mismo signo. El clip funciona igual en en .linlin.

OSC

Open Sound Control es un protocolo de comunicación en red que utilizan tanto computadoras como sintetizadores y otros aparatos multimedia. Asimismo este protocolo comunica diferentes programas. SC tiene implementado OSC para su comunicación interna y de manera externa para comunicarse con otros programas como pueden ser Processing, OpenFrameworks o PureData. Este protocolo también es útil para comunicar un *smartphone* o para hacer colaboraciones en red a través de Internet o de manera local mediante TCP/IP a través de los puertos UDP.

La comunicación mediante OSC podemos pensarla en de dos maneras, una es el envío de datos y otra es la recepción de datos.

Envío de datos

NetAddr(IP, puerto)
Dirección de red

IP: es una cadena de números: “127.0.0.1” representa comunicación interna.
puerto: es un número como este 57120, es el número de puerto exclusivo de SC.

```
// envío de datos de SC a SC dentro de la misma computadora

NetAddr("127.0.0.1", 57120)
```

Recepción de datos

OSCresponder(dirección, nombreComando, acción)

Cliente que responde a la red, registra una función para ser llamada con un comando específico de una dirección OSC específica.

dirección: la dirección desde donde recibe el *responder,* puede ser nil en cuyo caso responde a cualquier dirección
nombre de comando: un comando OSC del tipo: /algo
acción: una función que será evaluada cuando un comando de ese nombre es recibido por la dirección

```
// recepción de datos
OSCresponder (n, "/buenas/tardes", {arg tiempo, mensaje,
respuesta; [mensaje, tiempo]}).add
```

Comenzamos enviando datos internamente en SuperCollider.

```
// declaro el código para enviar datos a SC dentro de mi pc
n=NetAddr("127.0.0.1", 57120)

// delcaro el código para recibir datos de SC dentro de mi pc
r=OSCresponder(nil,"/recibo",{|tiempo,respuesta,mensaje,comando|
[tiempo,respuesta,mensaje,comando].postln}).add

// mando un mensaje con esta línea, en la post aparece el
resultado
n.sendMsg("/recibo",1)
```

La post me imprime una serie de valores encasillados dentro de un Array, de los cuales me interesa el tercero que es el mensaje, el cual a su vez es un Array que contiene el comando OSC y un valor útil. Para filtrar ese valor usaremos un OSCresponder que indica que información quiero leer.

Aplicaciones del mapeo y OSC

Sonificación de datos

La sonificacion aborda el manejo de un conjunto con una gran cantidad de elementos o de un flujo constante de datos variables para convertirlos en sonido.
Ahora vamos a sonificar el conjunto ~u después de haber sido analizado. Creamos primero un SynthDef que acepte cambios en argumentos de frecuencia y duración.

```
(
SynthDef(\sonido, {|freq=200, amp=0.5, dur=0.15|
```

```
     var sig, env;
     sig=Saw.ar(freq, amp);
     env=EnvGen.kr(Env.sine(dur,1),doneAction:2);
     Out.ar(0,sig*env);
     }).send(s);
);

Synth(\sonido)

// Reestructuramos los subconjuntos ~a, ~b y ~c.

(
~a=[];
~b=[];
~c=[];
~u.size.do{|i|
     if(~u[i]<=30,{~a=~a.add(~u[i])});
     if((~u[i]>30)&&(~u[i]<=60),{~b=~b.add(~u[i])});
     if(~u[i]>60,{~c=~c.add(~u[i])});
     }
)

// Declaramos una frecuancia fundamental.

~fund=30;
```

El siguiente algoritmo de sonificación le manda el mensaje .midiratio a los números enteros del subconjunto ~a para leerlos como semitonos sobre la fundamental. La duración es el inverso de la cantidad de veces que se repite el número entero. Además repite la nota que representa ese número entero la cantidad de veces que se repite dentro de ~a.

```
(
(
Tdef(\sonificacionA, {var dur;
               ~a.size.do{|i|
               dur=1/
(~incidenciasA[~elementosA.find([~a[i]])]).postln;
     ~incidenciasA[~elementosA.find([~a[i]])].do{
     Synth(\sonido, [\freq, ~a[i].midiratio*~fund, \dur, dur]);
     dur.wait;
```

```
     }
     }
     }).quant_(0);
);
/*
Tdef(\sonificacionA).play;
Tdef(\sonificacionA).stop;
*/

// Lo mismo para ~b.

(
Tdef(\sonificacionB, {var dur;
                    ~b.size.do{|i|
                              dur=1/
(~incidenciasB[~elementosB.find([~b[i]])]).postln;

     ~incidenciasB[~elementosB.find([~b[i]])].do{
                              Synth(\sonido, [\freq,
~b[i].midiratio*~fund, \dur, dur]);
                              dur.wait;
                              }
                              }
                    }).quant_(0);
);
/*
Tdef(\sonificacionB).play;
Tdef(\sonificacionB).stop;
*/

// Y para ~c

(
Tdef(\sonificacionC, {var dur;
                    ~c.size.do{|i|
                              dur=1/
(~incidenciasC[~elementosC.find([~c[i]])]).postln;

     ~incidenciasC[~elementosC.find([~c[i]])].do{
```

```
                                Synth(\sonido, [\freq,
~c[i].midiratio*~fund, \dur, dur]);
                                dur.wait;
                                }
                                }
                    }).quant_(0);
);
/*
Tdef(\sonificacionC).play;
Tdef(\sonificacionC).stop;
*/
)

// Aquí hechamos a andar los tresubconjuntos sonificados.

/*
(
Tdef(\sonificacionA).play;
Tdef(\sonificacionB).play;
Tdef(\sonificacionC).play;
);
(
Tdef(\sonificacionA).stop;
Tdef(\sonificacionB).stop;
Tdef(\sonificacionC).stop;
)
*/
```

Bibliografía

Netri, E. y Romero, E. (2008). *Curso de SuperCollider Intermedios.* Centro Multimedia: México.