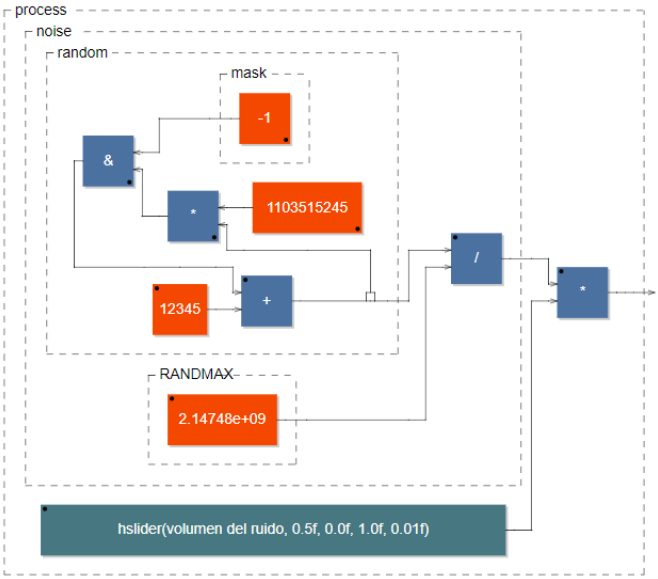
Parte 5: Filtros y síntesis sustractiva

La síntesis sustractiva es lo opuesto a la síntesis aditiva. Consiste en partir de un sonido rico, por ejemplo, ruido blanco o una diente de sierra, y esculpir su espectro mediante filtros y otros procesos. Existen muchos enfoques para llevarla a cabo, pero en esta sección vamos a aprender a generar algunos filtros y herramientas para crear un sintetizador simple mediante el algoritmo Karplus-Strong. Este algoritmo nos permitirá simular de una forma muy simple sonidos de tipo cuerda pulsada o frotada, e incluso percusivos.

Contenidos | Parte 5

Ejemplo 1: Generando ruido blanco

Antes de comenzar, veremos uno de los posibles generadores para el Karplus-Strong: el ruido blanco. Podríamos generar ruido blanco con un par de trucos matemáticos-digitales, básicamente generando números aleatorios para cada muestra. Pero Faust nos provee una función que ya resuelve este problema: `no.noise`. Esta función no requiere argumentos y nos generará ruido blanco con amplitud máxima entre 1.0 y -1.0.




```
import("stdfaust.lib");

process = no.noise * hslider("volumen del ruido", 0.5, 0, 1, 0.01);
```

Podemos observar en el diagrama el funcionamiento interno de `no.noise`, que consta principalmente de una suma recursiva de un número de base (12345) con una multiplicación por otro número mucho mayor, y un enmascarado digital. La recursión genera resultados que son pseudoaleatorios, es decir, ***parecen aleatorios*** porque no somos capaces de predecirlos o de ver fácilmente un patrón, pero se trata de una operación determinista: dadas las condiciones apropiadas, utilizando los mismos números de base, obtendremos exactamente la misma progresión de números como resultado.

Afortunadamente, no tendremos que preocuparnos por nada de esto gracias a `no.noise`, que nos dará un suficientemente convincente ruido blanco.

Las librerías de Faust nos ofrecen distintos tipos de ruido para distintas aplicaciones:

https://faust.grame.fr/doc/libraries/#noises.lib

Ejemplo 2: Filtro pasa bajos

Podemos implementar distintos tipos de filtros incluidos en las librerías de Faust. Por ejemplo, pasemos el ruido blanco por un filtro pasa bajos (Butterworth) con `fi.lowpass(orden, frecuenciaDeCorte)`. El orden (número entero) afectará la pendiente del corte, lo que nos servirá para dar distinto carácter al filtro.

```
import("stdfaust.lib");

process = no.noise * hslider("Ruido", 0.5, 0, 1, 0.01)
          : fi.lowpass(3, hslider("Agudos", 10000, 20, 22000, 1));
```

Ejemplo 3: Filtro pasa altos

Podemos implementar un pasa altos mediante `fi.hihpass(orden, frecuenciaDeCorte)`.

```
import("stdfaust.lib");

process = no.noise * hslider("Ruido", 0.5, 0, 1, 0.01)
          : fi.highpass(3, hslider("Graves", 10000, 20, 22000, 1));
```

Ejemplo 4: Filtro pasa banda (compuesto)

También podemos combinarlos para crear una suerte de pasabandas.

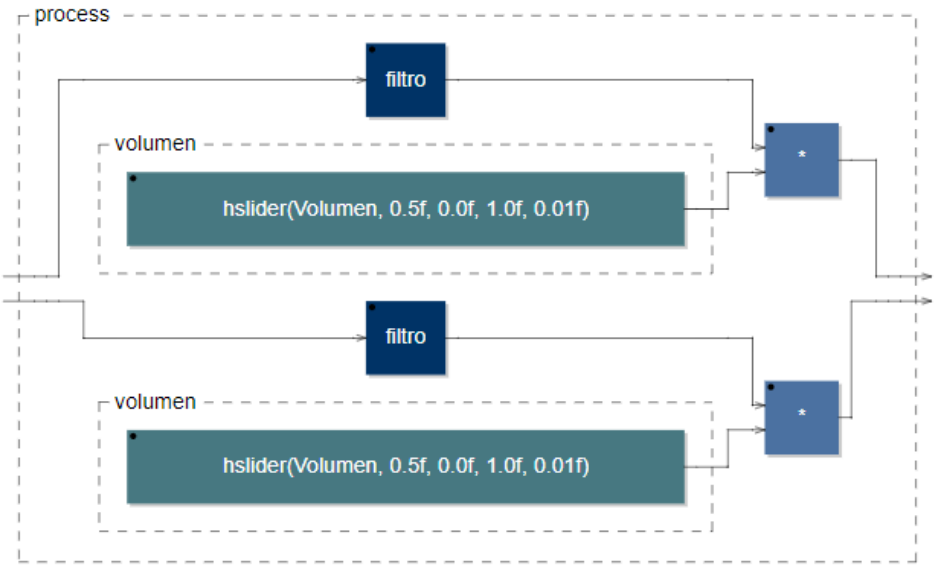
```
import("stdfaust.lib");
frecuencia = hslider("Frecuencia Central", 2000, 20, 20000, 1);
ancho = hslider("Ancho de banda Hz", 500, 1, 5000, 1);
ORDEN = 50;

// Utilizamos min y max para no exceder el rango 10 - 19000 Hz
// Podemos generar artefactos al cruzar el limite grave
process = no.noise * hslider("Ruido", 0.5, 0, 1, 0.01)
          : fi.highpass(ORDEN, max(frecuencia - ancho/2, 10))
          : fi.lowpass(ORDEN, min(frecuencia + ancho/2, 19000));
```

También podemos crear con este principio un filtro rechaza banda, pero esto requerirá revisar las conexiones: si nos limitamos a invertir las frecuencias de corte de ambos filtros, ¡quedaremos sin señal!, ya que uno eliminará un fragmento, y el otro eliminará el propio. Lo que precisamos en este caso es ***sumar*** los resultados de ambos filtros. Nuevamente, disponemos de una gran variedad de filtros en las librerías de Faust:

https://faust.grame.fr/doc/libraries/#filters.lib

Ejemplo 5: Filtro resonante

Un tipo interesante de filtros son los llamados ***resonantes***. Su disposición interna hace que exista una resonancia, por lo general en su frecuencia de corte, que podemos aprovechar para distintos efectos. Utilizaremos `fi.resonlp(frecuencia, Q, gain)` (pasa bajos resonante) para hacer resonar una determinada frecuencia sobre el audio que esté sonando. En la Parte 6 de este curso veremos nuevamente este tipo de filtros y haremos que estas resonancias modulen automáticamente, generando efectos aún más interesantes.



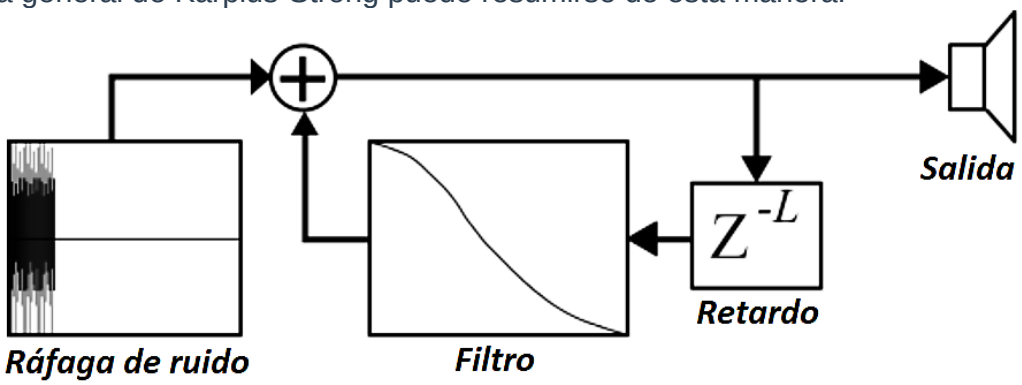

```
import("stdfaust.lib");

filtro = fi.resonlp(
                hslider("Frecuencia Resonante", 5000, 100, 10000, 1),
                hslider("Q", 25, 1, 100, 0.01),
                0.5);    // Gain fijo de 0.5

volumen = hslider("Volumen", 0.5, 0, 1, 0.01);

// Utilizar con un sonido en WAV o MP3, preferiblemente música.
// Probar mover en tiempo real la frecuencia con audio sonando!
process = filtro*volumen , filtro*volumen;    // Estereo sencillo
```

Ejemplo 6: Karplus Strong (parte 1)

Finalmente estamos en condiciones de experimentar con el algoritmo de Karplus-Strong, uno de los más utilizados tanto en la síntesis analógica (¡mucho!) como la digital. Se trata de un método de síntesis por ***modelado físico***, que repite una forma de onda ***muy breve*** a través de una línea de retardo, luego filtrada para simular el sonido de una cuerda percutida, pulsada, frotada o incluso algunos tipos de percusión de membranas (como un tambor).

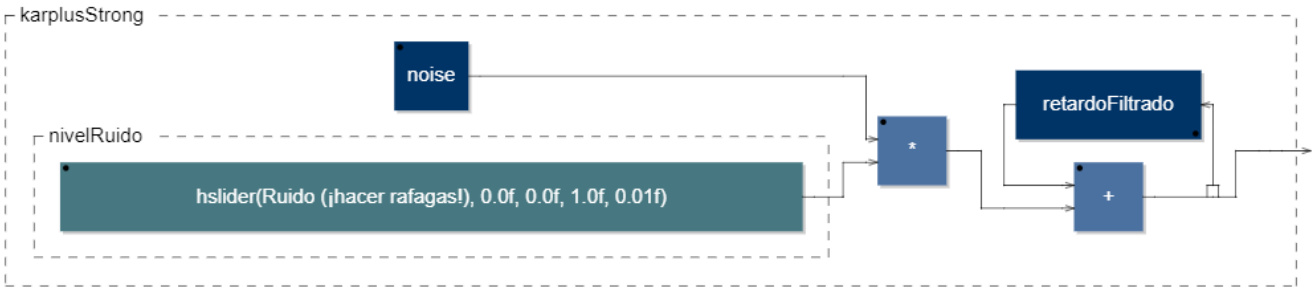
Aunque se la considera un caso típico de síntesis sustractiva, también puede verse a Karplus-Strong como un tipo muy elemental de síntesis por tabla de ondas (wavetable), ya que la línea de retardo almacena, en definitiva, un período de la señal. Es habitual en la síntesis de sonido que las distintas categorías se superpongan en ciertos aspectos.

El esquema general de Karplus-Strong puede resumirse de esta manera:



Generaremos una breve ***ráfaga de ruido*** (¡aunque pueden ser otros sonidos de base!), utilizada como excitador del sistema. Esta ráfaga alimenta la salida y al mismo tiempo una ***línea de retardo***. Luego la salida del retardo es previamente intervenida por un ***filtro***, normalmente un pasa bajos. Finalmente esta señal con retardo y filtrada, es ***sumada a la señal original*** para dirigirse a la salida. Obtendremos distintos tipos y frecuencias de sonidos según cómo ajustemos los parámetros del filtro, del retardo, la forma de onda de la ráfaga, su duración, envolvente, etc.

El efecto de algunos parámetros puede predecirse, como por ejemplo la frecuencia fundamental, la duración de la nota, etc. Pero lo interesante del algoritmo es ***experimentar con él*** y ver qué ocurre con cada alteración. El código puede ser mucho más simple y sintético, pero lo haremos extenso y muy comentado para favorecer su comprensión. El diagrama del refleja lo visto en el esquema anterior.

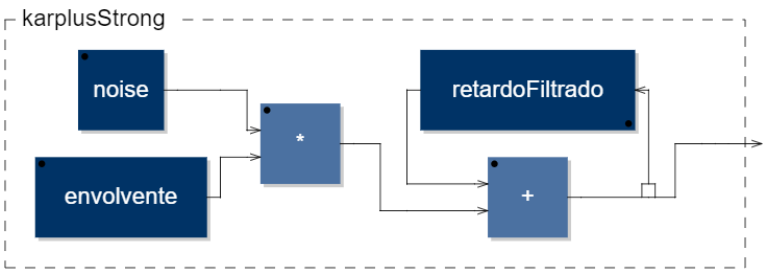


```
import("stdfaust.lib");

// Controles del ruido del Karplus-Strong
GAIN_MINIMO = 0.7;  // Cuanto menor sea, mas se parecera al ruido blanco
GAIN_MAXIMO = 0.999; // Cuidado no llegar a 1, ¡o el sonido no se detendra!
invertir = (checkbox("Invertir ruido") * 2 - 1) * -1;   // -1 a 1 ¡ver el efecto!
gainRealimentacion = hslider("Gain", GAIN_MAXIMO, GAIN_MINIMO, GAIN_MAXIMO, 0.0001);
nivelRuido = hslider("Ruido (¡hacer rafagas!)", 0, 0, 1, 0.01);

// Filtro. Probar cambiar sus parametros
filtroKarplus = fi.lowpass(2, 5000);

// Generamos el retardo con el filtro
retardoFiltrado = @ (hslider("Cantidad de muestras de retardo", 200, 1, 1000, 1))
                : filtroKarplus
                : *(gainRealimentacion * invertir);

// Componemos el algoritmo, con la recursion ~ para que se vaya sumando al original
karplusStrong = no.noise * nivelRuido : + ~ retardoFiltrado;

// Tambien podemos colorear el resultado final con otro filtro
filtroFinal = fi.lowpass(1, 10000);

// Karplus Strong! (mono duplicado en L y R mediante la bifurcacion <: )
process = karplusStrong : filtroFinal <: _, _;
```

Ejemplo 7: Karplus Strong (parte 2)

Ahora que conseguimos la base del algoritmo, vamos a darle una forma más aprovechable como sintetizador. En lugar de controlarlo con un slider, le añadiremos una envolvente dinámica de tipo AR (una versión simplificada del habitual ADSR). También definiremos la fundamental en términos de nota MIDI (en lugar de cantidad de muestras) y el nivel según el Velocity.

Para ello necesitaremos hacer unas pocas modificaciones a nuestro código anterior. Como funcionará con MIDI (o un teclado), no olvidar elegir una cantidad de voces en Poly Voices, y elegir la entrada apropiada en MIDI Input. Culminaremos el sintetizador añadiendo un filtro resonante variable, estilo efecto de ***Wah Wah***, que podremos controlar con la rueda de modulación MIDI o a mano. Esto lo haremos añadiendo `[midi:ctrl 1]` en la etiqueta para forzar a Faust a usar el CC 1 (modulación) con este slider, pero el número puede cambiarse por cualquier otro CC MIDI.



```
import("stdfaust.lib");

// frecuencia desde MIDI o teclado
frecuencia = nentry("freq", 120, 60, 20000, 1);

// cantidad de muestras de retardo segun la fundamental deseada
// puede haber desafinaciones porque los retardos son numeros enteros!
retardo = int(ma.SR / frecuencia);

// Controles del ruido del Karplus-Strong
GAIN_MINIMO = 0.7;  // Cuanto menor sea, mas se parecera al ruido blanco
GAIN_MAXIMO = 0.999; // Cuidado no llegar a 1, ¡o el sonido no se detendra!
invertir = (checkbox("Invertir ruido") * 2 - 1) * -1;   // -1 a 1 ¡ver el efecto!
gainRealimentacion = hslider("Gain", GAIN_MAXIMO * 0.99, GAIN_MINIMO, GAIN_MAXIMO, 0.
0001);
envolvente = en.ar(0.003, 0.01, button("gate")) ;  //  envolvente para el ruido, expe
rimentar!

// Filtro. Probar cambiar sus parametros
filtroKarplus = fi.lowpass(2, 5000);

// Generamos el retardo con el filtro
retardoFiltrado = @ (retardo)
                : filtroKarplus
                : *(gainRealimentacion * invertir);

// Componemos el algoritmo, con la recursion ~ para que se vaya sumando al original
karplusStrong = no.noise * envolvente : + ~ retardoFiltrado;

// si se dispone de un control de modulacion MIDI, jugar con este filtro
frecResonante = hslider("Resonante Wah (¡mover!)[midi:ctrl 1]", 1, 0, 1, 0.01) : si.s
moo;
filtroRes = fi.resonlp(frecResonante * 4000 + 300,
                hslider("Q del filtro Resonante", 10, 1, 100, 0.01),
                0.5);    // Gain fijo de 0.5

// incorporamos el velocity MIDI
velocity =  nentry("gain", 0, 0, 1, 0.0001);

// Karplus Strong! (mono duplicado en L y R mediante la bifurcacion <: )
process = karplusStrong * velocity : filtroRes <: _, _;
```

Ejemplo 8: Kisana

Agregamos este ejemplo solo para divertirnos un poco con su funcionamiento. Fue creado por Yann Orlarey, miembro del equipo desarrollador de Faust. Kisana consta de varias “cuerdas” que componen tres “arpas”, creadas a partir del algoritmo Karplus-Strong, y una serie de condiciones y parámetros para automatizar su ejecución a lo largo del tiempo. Es algo complejo, pero no buscaremos entenderlo completamente ahora… ¡sólo disfrutarlo!. Experimentar con los tres sliders para generar distintas texturas. (Deben copiarse todas las páginas).

```
declare name    "myKisana";
declare author  "Yann Orlarey";

//Modifications GRAME July 2015

/* ========= DESCRITPION =============

- Kisana : 3-loops string instrument (based on Karplus-Strong)
- Head = Silence
- Tilt = High frequencies
- Front = High + Medium frequencies
- Bottom = High + Medium + Low frequencies
- Left = Minimum brightness
- Right = Maximum birghtness
- Front = Long notes
- Back = Short notes

*/

import("stdfaust.lib");

KEY = 48;   // basic midi key
NCY = 8;   // note cycle length
CCY = 8;   // control cycle length
BPS = 360;  // general tempo (ba.beat per sec)

process = kisana : volumen;
volumen = sp.stereoize( *(hslider("Volumen", 4, 0, 6, 0.01)));
//-------------------------------kisana-----------------------------------
// USAGE:  kisana : _,_;
//      3-loops string instrument
//------------------------------------------------------------------------

kisana = vgroup("MyKisana", harpe(C,11,48), harpe(C,11,60), (harpe(C,11,72) : *(1.5),
 *(1.5))
    :> *(1), *(1))
    with {
```

```
        l = -20 : ba.db2linear;//hslider("[1]Volume",-20, -
60, 0, 0.01) : ba.db2linear;
        C = hslider("[2]Brightness[acc:0 1 -
10 0 10]", 0.5, 0, 1, 0.01) : ba.automat(BPS, CCY, 0.0);
    };

//----------------------------------Harpe-----------------------------------
// USAGE:  harpe(C,10,60) : _,_;
//      C is the filter coefficient 0..1
//      Build a N (10) strings harpe using a pentatonic scale
//      based on midi key b (60)
//      Each string is triggered by a specific
//      position of the "hand"
//--------------------------------------------------------------------------
harpe(C,N,b) =  hand(b) <: par(i, N, position(i+1)
                            : string(C,Penta(b).degree2Hz(i), att, lvl)
                            : pan((i+0.5)/N) )
                    :> _,_
    with {
        att  = hslider("[3]Resonance[acc:2 1 -10 0 12]", 10, 0.1, 10, 0.01);
        hand(48) = vslider("h:[1]Instrument Hands/1 (Note %b)[unit:pk]", 1, 0, N, 1)
: int : ba.automat(120, CCY, 0.0);
        hand(60) = vslider("h:[1]Instrument Hands/2 (Note %b)[unit:pk]", 3, 0, N, 1)
: int : ba.automat(240, CCY, 0.0);
        hand(72) = vslider("h:[1]Instrument Hands/3 (Note %b)[unit:pk]", 5, 0, N, 1)
: int : ba.automat(480, CCY, 0.0);
        //lvl  = vslider("h:loop/level", 0, 0, 6, 1) : int : ba.automat(BPS, CCY, 0.0
) : -(6) : ba.db2linear;
        lvl = 1;
        pan(p) = _ <: *(sqrt(1-p)), *(sqrt(p));
        position(a,x) = abs(x - a) < 0.5;
    };

//----------------------------------Penta-----------------------------------
// Pentatonic scale with degree to midi and degree to Hz conversion
// USAGE: Penta(60).degree2midi(3) ==> 67 midikey
//        Penta(60).degree2Hz(4)   ==> 440 Hz
//--------------------------------------------------------------------------

Penta(key) = environment {

    A4Hz = 440;

    degree2midi(0) = key+0;
    degree2midi(1) = key+2;
    degree2midi(2) = key+4;
    degree2midi(3) = key+7;
```

```
    degree2midi(4) = key+9;
    degree2midi(d) = degree2midi(d-5)+12;

    degree2Hz(d) = A4Hz*semiton(degree2midi(d)-
69) with { semiton(n) = 2.0^(n/12.0); };

};

//-----------------------------------String-------------------------------
// A karplus-strong string.
//
// USAGE: string(440Hz, 4s, 1.0, button("play"))
// or     button("play") : string(440Hz, 4s, 1.0)
//------------------------------------------------------------------------

string(coef, freq, t60, level, trig) = no.noise*level
                            : *(trig : trigger(freq2samples(freq)))
                            : resonator(freq2samples(freq), att)
    with {
        resonator(d,a)  = (+ : @(d-1)) ~ (average : *(a));
        average(x)      = (x*(1+coef)+x'*(1-coef))/2;
        trigger(n)      = upfront : + ~ decay(n) : >(0.0);
        upfront(x)      = (x-x') > 0.0;
        decay(n,x)      = x - (x>0.0)/n;
        freq2samples(f) = 44100.0/f;
        att             = pow(0.001,1.0/(freq*t60)); // attenuation coefficient
        random          = +(12345)~*(1103515245);
        noise           = random/2147483647.0;
    };
```

¡Uf! Hemos visto como trabajar con generadores de ruido, filtros y finalmente generar un sintetizador a partir del algoritmo Karplus-Strong. En la próxima parte veremos cómo trabajar con los fundamentos de la síntesis FM, y aprenderemos a trabajar con envolventes dinámicas moduladas para generar nuevos timbres y efectos.