Parte 4: Síntesis aditiva y control MIDI

La síntesis aditiva es una técnica para crear timbres nuevos a partir de la ***adición*** (suma) de formas de onda más simples. En un sentido más estricto, esto puede relacionarse con el análisis espectral de Fourier y el concepto de que cualquier sonido periódico, puede descomponerse en una suma de sinusoides.

Los ***timbres*** están formados por cantidades variables de armónicos o ***parciales*** que cambian a lo largo del tiempo con respecto a una ***frecuencia fundamental***. Los parciales son las ondas que complementan a la fundamental para crear un timbre, y si sus frecuencias son ***múltiplos enteros*** de la frecuencia fundamental son denominados parciales ***armónicos***, y si son múltiplos reales son denominados no armónicos. Al realizar síntesis aditiva necesitaremos controlar la amplitud de cada uno de los parciales de formas específicas para lograr las distintas formas de onda tradicionales.

Para realizar síntesis aditiva se hace necesario disponer de un banco de osciladores para que generaren las diferentes ondas que complementan la onda fundamental, cada una con amplitudes y frecuencias diferentes además de su propia envolvente configurable de volumen. El caso más simple es partir de una suma de sinusoides, pero con frecuencia los sintetizadores utilizan 4 osciladores con formas de onda básicas: ***sinusoidal***, ***cuadrada***, ***diente de sierra*** y ***triangular***; y son capaces de generar timbres más complejos combinando estas formas de onda.

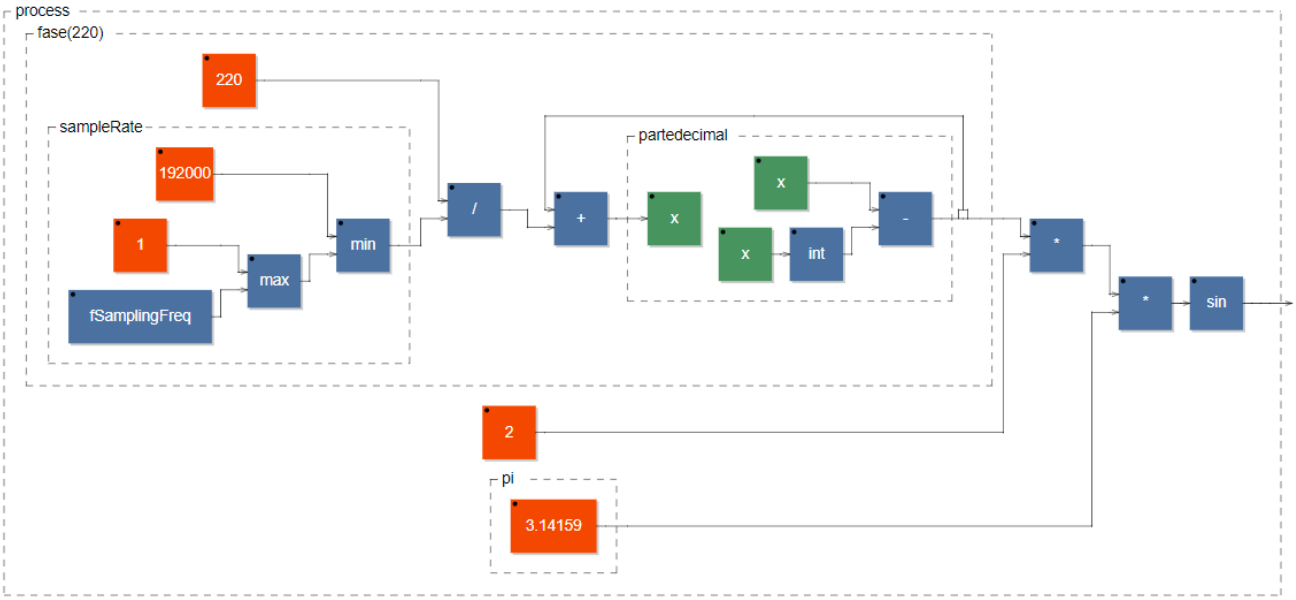
Contenidos | Parte 4

Ejemplo 1: Generador de onda senoidal

El generador de fase también es la base del generador de onda senoidal. Utilizando la función seno `sin(x)`, podremos generar una señal senoidal a través de esta simple fórmula:

```
senoidal = sin(fase * 2 * PI);
```

El número `PI` podríamos tipearlo a mano (3.1416…), pero para mayor precisión podemos obtenerlo del mismo modo que la frecuencia de muestreo, utilizando la función `ma.PI` incluida en `stdfaust.lib`. El número `fase` lo obtendremos justamente de nuestro generador de fase, al cual podremos especificarle la frecuencia deseada como hicimos con los ejemplos anteriores:




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

// les damos otro nombre para que sean mas faciles de leer
sampleRate = ma.SR;
pi = ma.PI;

// el generador de fase de siempre
partedecimal(x) = x - int(x);
fase(frec) = frec/sampleRate : (+ : partedecimal) ~ _;

// onda senoidal a partir de funcion seno
senoidal(frecuencia) = sin(fase(frecuencia) * 2 * pi);

process = senoidal(440) * 0.5;
```

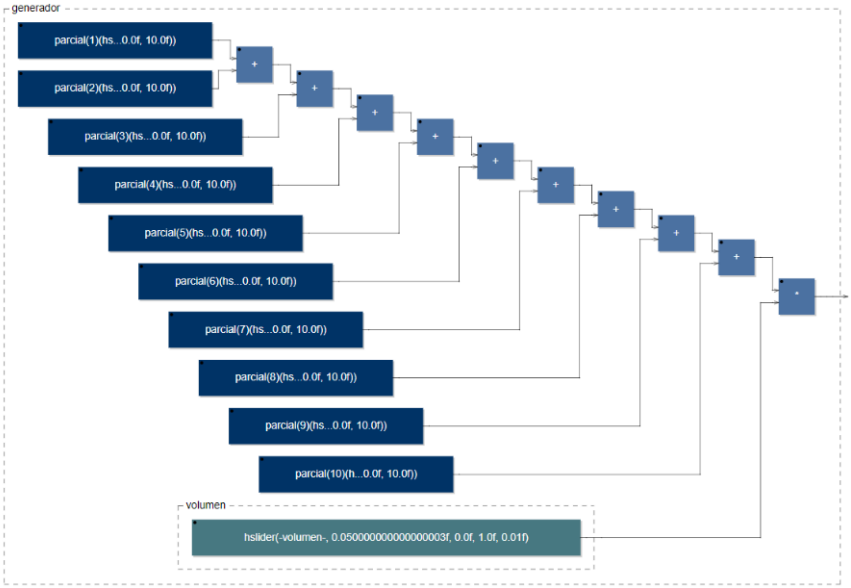
Ejemplo 2: Síntesis aditiva de diente de sierra

En el siguiente ejemplo generaremos una señal de diente de sierra utilizando síntesis aditiva, sumando sinusoides cuyas frecuencias sean múltiplos de una fundamental, y con amplitudes decrecientes. Para lograr esto introduciremos dos nuevos elementos a nuestro arsenal de Faust.

Hasta aquí hemos generado las formas de onda desde su matemática más fundamental, pero Faust nos provee una serie de osciladores pre-armados y fáciles de utilizar. Uno de ellos es `os.osc(frec)`, un generador de onda senoidal que solo requiere indicar la frecuencia, sin tener que preocuparnos por el generador de fase y demás cuestiones.

Por otro lado, sería muy ineficiente escribir a mano cada uno de los osciladores necesarios para producir una señal con, digamos, 20 o 30 parciales. Faust nos ofrece la función de iteración integrada `sum(indice, cantidad, bloque)`, que nos permitirá duplicar cualquier bloque tantas veces como necesitemos, pero pudiendo asignar a cada uno valores distintos gracias a su variable de índice. `sum` duplicará el ***bloque*** indicado tantas veces como diga el parámetro de ***cantidad***, pero en cada duplicación incrementará el valor de la variable de ***índice***, de manera que podamos aprovecharlo para cambiar las propiedades de cada nuevo bloque. Cabe aclarar que `sum`, como su nombre sugiere, ***sumará todas las señales*** producidas por las iteraciones que genere.

En el siguiente ejemplo duplicaremos un oscilador senoidal hasta generar un total de 10 parciales, y aprovecharemos el índice para indicarle a cada uno frecuencias múltiplos de la fundamental, e intensidades que disminuyen en proporción inversa a su número de parcial.



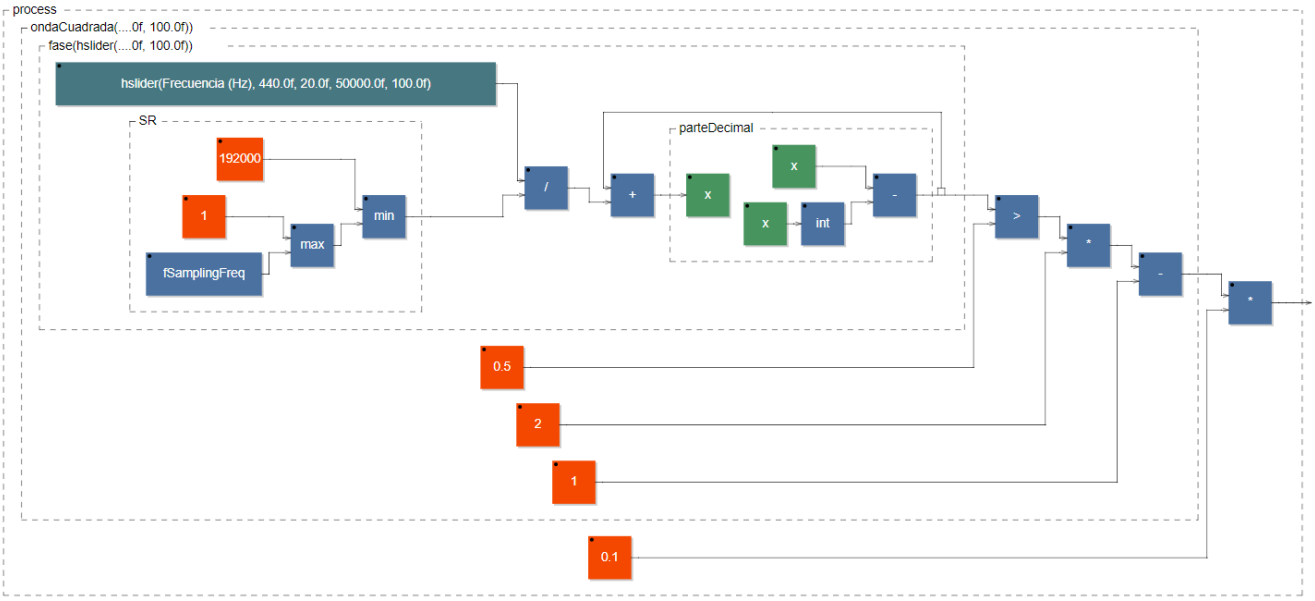

```
import("stdfaust.lib");
volumen = hslider("-volumen-", 0.05, 0, 1, 0.01);
fundamental = hslider("-frecuencia-", 80, 20, 2500, 10);

// oscilador de cada parcial, será duplicado por sum()
parcial(indiceParcial, frecFundamental) = os.osc(frecFundamental * indiceParcial) *
    hslider("%indiceParcial Parcial", 1 / indiceParcial, 0, 1, 0.01);

// duplicamos el oscilador 10 veces para tener 10 parciales, probar que pasa con mas!
generador = sum(numParcial, 10, parcial(numParcial + 1, fundamental) ) * volumen;
process = generador, generador;
```

Ejemplo 3: Fenómeno de alias más allá de Nyquist (SR/2)

En el ejemplo anterior, podríamos haber notado una peculiaridad: al tener fundamentales y sus armónicos lo suficientemente agudos, comienzan a aparecer sonidos extraños e inesperados. Un problema bien conocido en el campo del sonido digital es el ***alias*** de frecuencia: cualquier frecuencia que se intente generar más allá de la ***mitad de la frecuencia de muestreo*** (es decir, la frecuencia de Nyquist) se ***plegará*** dentro del espectro audible. Si nuestra frecuencia de muestreo es de 48 kHz, la frecuencia máxima que podremos representar es 24 kHz. Si intentamos generar (o muestrear) un tono superior, digamos de 30 kHz, éste será representado (***alias***) como una nueva frecuencia de 18 kHz (48 – 30), totalmente falsa respecto de lo esperado. Podremos escuchar este efecto claramente con nuestro viejo ejemplo de la onda cuadrada, escogiendo fundamentales altas que permitan que sus armónicos superen la frecuencia de Nyquist:




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

// generador de onda cuadrada
parteDecimal(x) = x - int(x);
fase(f) = f/ma.SR : (+ : parteDecimal) ~ _;
ondaCuadrada(f) = ((fase(f) > 0.5) * 2) - 1;

// probar que ocurre con el sonido a medida que subimos la frecuencia
process = ondaCuadrada( hslider("Frecuencia (Hz)", 440, 20, 50000, 100) )  * 0.1;
```

Como puede observarse, aparecen sonidos cuasi caóticos (hasta cómicos a veces) que poco tienen que ver con la nota y armónicos que esperamos escuchar. Existen distintas estrategias para solucionar este problema, pero las más usadas son dos: ***filtrar la señal*** (con un filtro ***pasabajos***) para que no contenga información por encima de la frecuencia de Nyquist, o -de ser posible- generar directamente formas de onda ***sin parciales por encima*** de Nyquist.

Osciladores integrados y librerías de Faust

Hemos hecho menciones ocasionales acerca de las librerías de Faust, que habitualmente incluimos utilizando `import("stdfaust.lib");`. La documentación de estas librerías podemos encontrarla en:

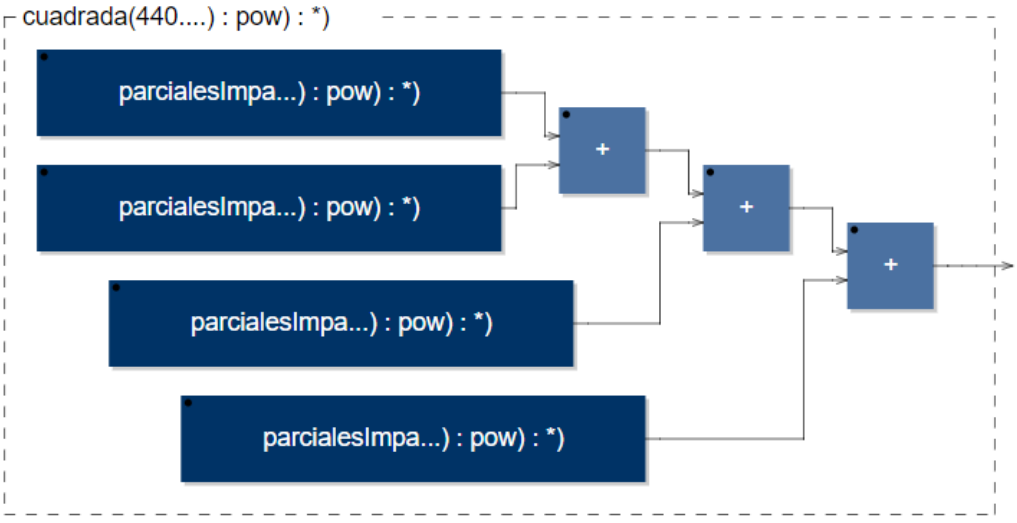
https://faust.grame.fr/doc/libraries/#standard-functions-1

Allí encontraremos funciones para todo tipo de necesidades que podamos tener, incluyendo muchos osciladores con banda limitada para evitar el mencionado ***aliasing***, filtros, analizadores, envolventes, efectos, generadores de ruido y muchas cosas más. Algunos elementos los iremos viendo en el resto de este curso, pero se recomienda echar un vistazo rápido a las opciones disponibles en la documentación, como para tener una idea de las distintas posibilidades. Si bien el contenido está en inglés, los nombres de los elementos serán fácilmente identificables para su utilización.

Ejemplo 4: Síntesis aditiva de onda cuadrada

Vimos anteriormente cómo producir una señal de onda cuadrada ***perfecta***, debido a que podía realizar una transición instantánea entre el máximo y el mínimo, lo cual es físicamente imposible. En teoría, esta señal cuadrada perfecta ***contiene un número infinito de armónicos impares*** que, debido al muestreo, se plegarán sobre el espectro audible, resultando en los efectos de alias que describimos. Utilizando una estrategia diferente, podemos aproximar la señal cuadrada mediante síntesis aditiva, agregando una serie limitada de ***armónicos impares***, que son los que componen esta forma de onda. En contraste, para generar una diente de sierra utilizábamos todos los armónicos enteros, pares e impares.

Para esto usaremos la misma técnica que con la diente de sierra, pero haremos una pequeña cuenta para saltearnos los números pares: al número de índice (parciales) lo multiplicaremos por 2 y le sumaremos 1. Teniendo en cuenta que este número arranca a contar desde el 0 (debido al comportamiento de `sum`), obtendremos como resultado una serie de números impares a partir del 1. Por otra parte, haremos uso de la función `ba.midikey2hz(notaMIDI)` para convertir un número de nota MIDI a la frecuencia en Hz correspondiente. De esta manera podemos generar un control por slider que tenga un sentido más ***musical*** que el número crudo de la frecuencia.



```
import("stdfaust.lib");
nParciales = 10; // cambiar y comparar el timbre y forma de onda! (solo numeros enteros)

// obtenemos los numeros IMPARES desde los indices: 2*indice + 1
parcialesImpares(indice, frecuencia) = os.osc((2*indice + 1) * frecuencia)  /
      (2*indice + 1);
cuadrada(frecFundamental) = sum(indiceParcial, nParciales,
      parcialesImpares(indiceParcial, frecFundamental));

// convertimos un numero entero de nota MIDI a su equivalente en Hz
nota = ba.midikey2hz(hslider("Nota MIDI", 60, 32, 80, 1));

process = cuadrada(nota), cuadrada(nota);
```

Experimentar lo que ocurre cambiando el número de parciales definido al comienzo, y observar cómo cambia el timbre y la forma de onda en el osciloscopio en consecuencia. Este número debe incluir la fundamental, por lo que puede ser cualquier entero mayor a 1.

Ejemplo 5: ¡Al fin! Tocando notas vía MIDI

Faust tiene la capacidad de recibir datos externos como información de notas e intensidades. Si conectamos un controlador MIDI a la computadora, podremos elegirlo en el editor web como entrada MIDI. Alternativamente, el editor nos ofrece utilizar el teclado de la computadora para introducir notas, como ocurre en Ableton Live y otros softwares de audio. Se recomienda comenzar a probar ideas usando el teclado de la PC. Su digitación es la siguiente:

Debemos ver o entender a un sintetizador que hayamos escrito en Faust como una ***voz monofónica***, capaz de responder a ***una sola*** nota MIDI (independientemente de si el sintetizador genera notas musicales u otro tipo de sonidos). Esto no quiere decir que no podamos tener múltiples notas al mismo tiempo, pero una vez que tengamos nuestra ***voz***, Faust se ocupará automáticamente de duplicarla las veces que sea necesario y administrar las distintas voces.

Por defecto los programas en Faust son monofónicos, pero podemos modificar esto en el editor web, en ***Poly Voices*** del panel de la izquierda.

Veamos cómo lograr que Faust responda al ingreso de notas de un controlador o el teclado. Hasta ahora hemos escrito ***etiquetas*** de texto entre comillas como “Volumen” o “Nivel” en los sliders, botones y distintos controles gráficos, que nos servían para identificar cada función, pero no tenían relevancia para el funcionamiento del programa en sí. Sin embargo hay tres etiquetas específicas que Faust reconoce como funciones de control de notas:

- “freq” *es la frecuencia que corresponde a la* ***nota*** *ingresada*
- “gain” *es el equivalente al* ***Velocity****, pero con un rango de 0.0 a 1.0*
- “gate” *será 1 si se recibe un* ***noteOn****, y 0 si se recibe un* ***noteOff***

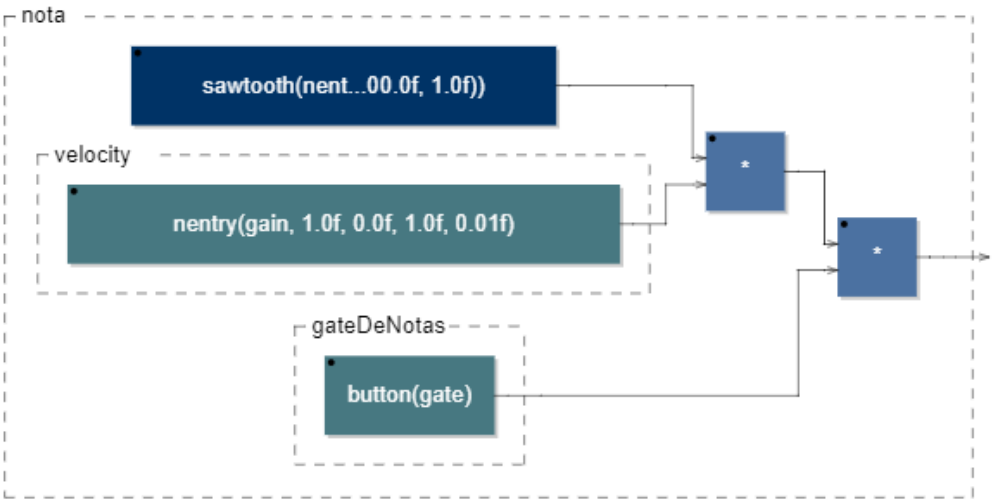
Esto quiere decir que si creamos, por ejemplo, un botón con la etiqueta “gate”, Faust automáticamente lo tomará como representación de un noteOn o noteOff, poniéndolo en 1 o 0 según la información que ingrese vía MIDI o por el teclado.

```
gateDeNotas = button("gate");
```

Cabe la aclaración: la definición puede llevar cualquier nombre (***gateDeNotas*** aquí), lo importante es que la ***etiqueta*** del elemento sea exactamente ***“gate”***. Para que esto funcione, es necesario indicar en el editor web de Faust que queremos un número de voces determinado (que bien puede ser 1 sola), cambiando ***Poly Voices*** a cualquier valor diferente a Mono. Aunque no es obligatorio, lo más recomendable es utilizar “freq” y “gain” con controles de tipo ***nentry*** (entrada numérica simple, similar al slider pero sin control deslizante), y “gate” con un ***button***, ya que al utilizar un controlador externo para elegir las notas, ya no precisamos de un slider para elegir la frecuencia.

```
frecuencia = nentry("freq", 440, 20, 20000, 1);
velocity = nentry("gain", 1, 0, 1, 0.01);
gateDeNotas = button("gate");
```

La lógica básica es la siguiente: debemos utilizar el número recibido por “freq” para definir la frecuencia fundamental de nuestro sintetizador, y multiplicarlo por “gain” para darle un nivel según el velocity MIDI de la nota pulsada. El uso de “gate” puede depender del contexto: al cambiar de 0 a 1 cuando se active una nota, podemos usarlo para multiplicar todo lo anterior, de forma que solo haya sonido (multiplicando por 1) si se ingresa una nota, y se “silencie” (multiplicando por 0) si se detiene la misma. Veamos el siguiente ejemplo (no olvidar seleccionar una cantidad de voces en Poly Voices, y elegir la entrada MIDI).



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

  frecuencia = nentry("freq", 440, 20, 20000, 1);
  velocity = nentry("gain", 1, 0, 1, 0.01);
  gateDeNotas = button("gate");

nota = os.osc(frecuencia) * velocity * gateDeNotas;

process = nota, nota;
```

¡Bien! Ya tenemos nuestro primer sintetizador que responde al ingreso de notas. Habremos notado que si no controlamos el nivel general, es posible que saturemos la salida de audio al ingresar muchas notas. Por lo general de ello se encargará el sistema sobre el que funcione nuestro DSP, pero es aconsejable añadir algún tipo de control de volumen.

Ahora démosle un toque final a nuestro sintetizador utilizando una envolvente dinámica. Podemos hacer esto con la función `en.adsr(ataque, decay, sustain, release, gateDeNotas)`, que nos generará una envolvente dinámica según los tiempos (en segundos) que escojamos en ***ataque***, ***decay*** y ***release***. El valor de ***sustain*** no es tiempo, sino el **nivel** al que se sostendrá la nota mientras dure activada, antes de pasar a la fase de release. Finalmente el último parámetro, ***gateDeNotas***, nos servirá para definir **cuándo debe arrancar** la envolvente (al recibir un 1) y **cuándo debe finalizar** (al recibir un 0). Para esto nos sirve nuevamente el número recibido por la etiqueta ***"gate"*** que mencionábamos anteriormente. Veamos este ejemplo en el que utilizamos otro de los osciladores integrados en Faust para generar una diente de sierra sin aliasing.

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

  frecuencia = nentry("freq", 440, 20, 20000, 1);
  velocity = nentry("gain", 1, 0, 1, 0.01);
  gateDeNotas = button("gate");

ataque = hslider("-ataque (ms)", 1, 0.1, 1000, 1) / 1000;
decay = hslider("-decaimiento (ms)", 100, 0, 1000, 1) / 1000;
sustain = hslider("-sustain (nivel)", 0.5, 0, 1, 0.01);
release = hslider("-release (ms)", 50, 0, 1000, 1) / 1000;

nota = os.sawtooth(frecuencia) * en.adsr(ataque, decay, sustain, release, gateDeNotas) * velocity;

process = nota, nota;
```

Hemos visto cómo trabajar con síntesis aditiva y ¡finalmente hemos logrado hacer sonar un sintetizador en base a información recibida vía MIDI!. Además agregamos una envolvente dinámica para darle otro nivel de expresión a los sonidos producidos. En la próxima parte veremos síntesis sustractiva, que nos servirá para introducir distintos tipos de filtros y nuevas técnicas de síntesis.