RAMP CODE

Un lenguaje de livecoding enfocado en la síntesis y la simpleza

QUÉ ES QUÉ

Livecoding es un tipo de performance donde se improvisan piezas programando algoritmos que son visibles por la audiencia.

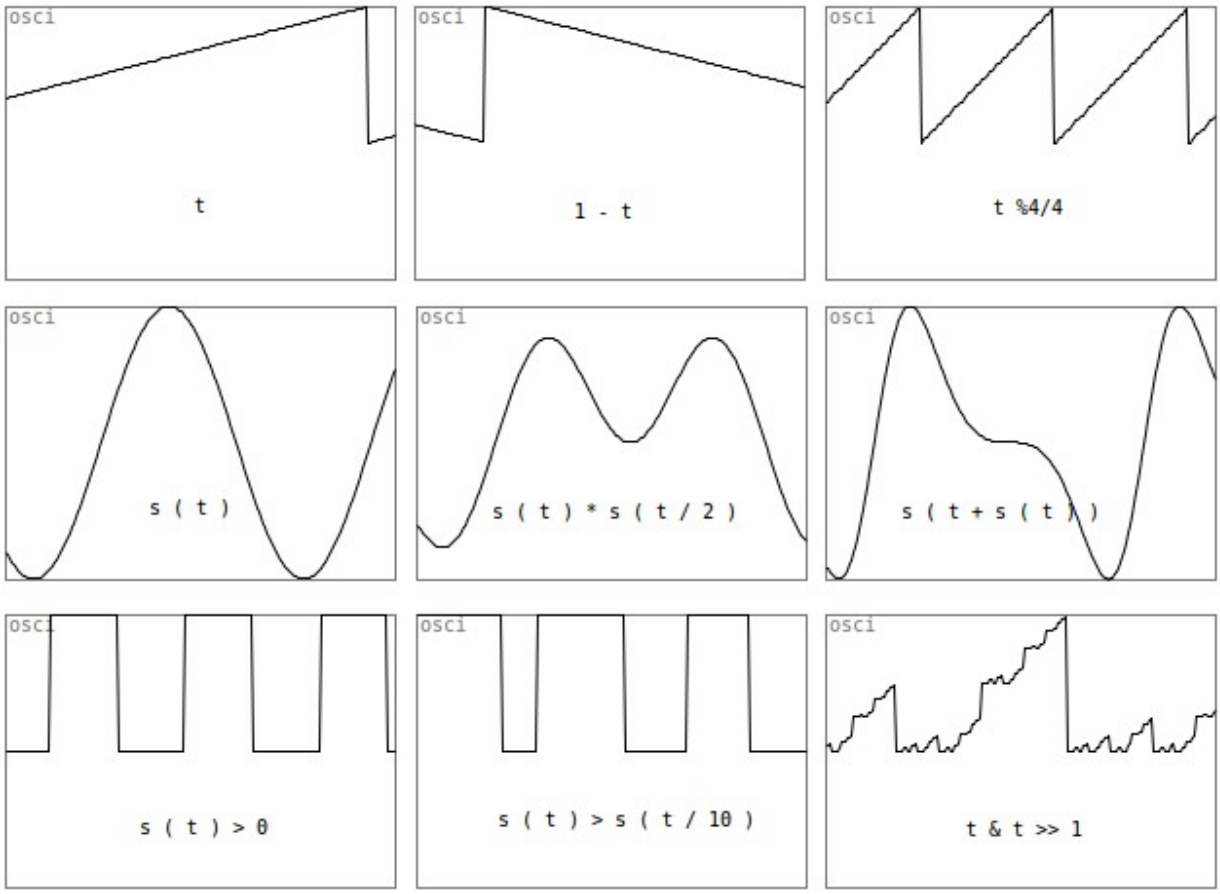
La mayoría de los lenguajes de livecoding -así concebidos- están enfocados en la articulación de sonidos pre seteados. La particularidad de *rampcode* es que se enfoca en la experimentación con síntesis.

Es software libre, está programado en Pure Data y la técnica que emplea está inspirada en el *bytebeat*. Esto significa que todo el resultado sonoro es producto de una función cuyo único input es el tiempo.

Tiempo (t) aquí es una *rampa* o contador, un número que se incrementa regular y constantemente. Con la ayuda de operadores lógicos y matemáticos, esa rampa se puede moldear para generar todo tipo de síntesis y articulaciones.

A esto nos referimos al decir que rampcode está enfocado en la simpleza. No al hecho de que sea más o menos sencillo de comprender sino a lo que en términos estéticos o científicos se conoce como *elegancia*: obtener los resultados más precisos y complejos a partir de la mínima cantidad de elementos y operaciones.

CÓMO SE HACE



En las figuras de arriba se pueden observar los ejemplos más básicos de cómo es posible modelar la rampa.

1. la *rampa* o *tiempo* (t), una función lineal
2. invertida (útil para generar *envolventes*)
3. uso de módulo para subdividir los ciclos

4. seno
5. amplitud modulada (AM)
6. frecuencia modulada (FM)

7. onda cuadrada
8. modulación de ancho de pulso (PWM)
9. uso de operadores *bitwise* (bytebeat)

ZOON POLITIKON

La simpleza de *rampcode* no es caprichosa. Este lenguaje está diseñado para explotar eficientemente los recursos de procesamiento, siendo posible lograr resultados muy sofisticados con técnicas y tecnologías que el mercado estima obsoletas.

A su vez, la elección por el software libre tampoco es casual. La práctica del livecoding hereda interesantes implicancias políticas que tienen que ver con la interdisciplinariedad, la divulgación libre del conocimiento, la horizontalidad, la desalienación y el trabajo en equipo.

Por eso rampcode es un lenguaje versátil que se presta a la experimentación límite pero también incluye herramientas que están orientadas al desarrollo rítmico y melódico propio de las expresiones musicales populares.

Si te interesa conocer más acerca del *livecoding* y la escena local no dejes de visitar:

https://toplap.org/

https://algorave.com/

http://livecodear.github.io/

https://gabochi.github.io/gede/

A LAS COSAS MISMAS

Definir en una única función las secuencias, envolventes, timbres y absolutamente todo lo referido a una pieza sonora o musical puede resultar ciertamente muy abstracto, por eso la mejor manera de entender es ponerse a jugar y meter mano en ejemplos ya hechos. A continuación hay una pequeña guía con muchos posibles usos *rampcode* pero todo su potencial, como el de cualquier lenguaje, es prácticamente inabarcable. Mucha suerte!

1. RAMPA Y CANALES

`r` determina la velocidad de la rampa en hz

Todo aquí es una función de la rampa, así que modificar su velocidad tiene efectos en todo lo demás.

Intentá cambiar la velocidad de vez en cuando para cambiar la atmósfera por completo.

1 hz es la velocidad por defecto.

```
r @ 1;
```

Podés usar muchos canales pero vamos a trabajar solamente con uno (suele ser suficiente).

Usá `c` para enviar mensajes a cada canal, separá los argumentos con `@` y terminá cada mensaje con `;`

```
c0 @ <expresión>;
```

2. SENO, TIEMPO Y AMPLITUD

`s()` es la función seno

`t` es el valor actual de la rampa
(+8000 por ciclo)

`*` en este caso funciona como amplitud

`cl` limpia el canal

`[ctrl+e]` envía la línea actual a Pure Data

```
c0 @ s(t) * 0.1;
c1 @ 0;
```

3. ENVOLVENTES

Podés agregar envolventes en cualquier lado

e o **i** deben seguir de dos caracteres que indican:

```
e/i <subdivisión> <curva>
```

Estos caracteres son *hexadecimales*: 1-F
Subdivisión quiere decir ciclo/X

```
c0 @ s(t) * 0.1 * e41;
c0 @ s(t) * 0.1 * e4F;
c0 @ s(t) * 0.1 * e12;
```

```
c0 @ s(t) * 0.1 * i12;
```

Dividir o multiplicar t por números enteros da como resultado la serie armónica:

```
c0 @ s(t/4) * 0.1 * e44;
c0 @ s(t*2) * 0.1 * e44;
```

e o *i* son formas abreviadas de decir

```
(1-t%(ciclo/subdiv)/(ciclo/subdiv))^curva
```

```
(t%(ciclo/subdiv)/(ciclo/subdiv))^curva
```

4. CONTADORES/SECUENCIAS

es un contador y sus argumentos son similares a **e** pero el segundo argumento indica la *longitud* de la cuenta:

```
# <subdivisión> <longitud>
```

```
c0 a s(t*#44) * 0.1 * e44;
c0 a s(t/#48) * 0.1 * e42;
c0 a s(t/#85/#43) * 0.1 * e83;
c0 a s(t/#65*#13) * 0.1 * e64;
```

```
c0 a s(t/#88*2) * 0.1 * e82 * (#88<5);
c0 a s(t/#88*2) * 0.1 * e82 * (#88>#85);
c0 a s(t/#88*2) * 0.1 * e82 * (#83==1 |
#84==1);
c0 a s(t/#88*2) * 0.1 * e82 * (#12==1);
```

```
c0 a (s(t/#88*2) * 0.1) * (i82 * (#88>5) +
e81 * (#88<6));
```

 # es una forma abreviada de decir

(t/(ciclo/subdiv)%(longitud))+1

5. ALEATORIEDAD Y CONCATENACIÓN

Podés obtener valores pseudo aleatorios con

`x<subdiv><semilla>` coma flotante entre 0 y 1
`X<subdiv><módulo>` números enteros

```
c0 @ s(t/X48) * 0.1 * e42;
```

```
c0 @ s(t/#44*X13) * 0.1 * e44;
```

```
c0 @ s(t*t) * e8F * 0.1 * (x20<0.5);
```

Usá la misma semilla para relacionar eventos aleatorios entre sí o una distinta para desvincularlos.

Escribir *x* es una forma abreviada de decir

```
a(s(t/(subdiv)%7919*s(t/(subdiv)
%7919+semilla)))
```

Podés concatenar todas las expresiones que quieras dentro de una misma expresión ya que básicamente estamos sumando señales.

Tené cuidado de no exceder la amplitud 1 y separá las operaciones de cada señal con ().

```
c0 @ s(t*t) * e8F * 0.1 * (x20<0.5) +
s(t/#84/2) * e82 * 0.1 * (x20>0.5);
```

```
c0 @ s(t*t) * e8F * 0.1 * (x21>0.5) +
s(t/#84/2) * e82 * 0.1 * (x2C>0.5);
```

6. TIMBRES Y FORMAS DE ONDA

Algunos ejemplos de distintas técnicas de síntesis para obtener timbres más interesantes:

AM, ring, vibrato...

```
c0 @ s(t/8) * s(t/1000) * 0.1;
c0 @ s(t/4) * s(t/500) * 0.1;
c0 @ s(t/8) * s(t/4) * 0.1;
c0 @ s(t/8) * s(t/50) * 0.1;
c0 @ s(t/4) * s(t/50) * 0.1;
```

Ondas cuadradas, PWM

```
c0 @ (s(t/#45)>0) * e41 * 0.1;
c0 @ (s(t/#45)>0.5) * e41 * 0.1;
c0 @ (s(t/#45)>s(t/500)) * e41 * 0.1;
c0 @ (s(t/#45)>s(t/8000)) * e41 * 0.1;
c0 @ (s(t/#85/2)>x80) * e82 * 0.1;
c0 @ (s(t/#85/XC2)>s(t/900*x80)) * e82 *
0.1;
```

FM

```
c0 @ s(t/8 + s(t/16)) * e21 * 0.1;
c0 @ s(t/8 + s(t/16) * 10) * e21 * 0.1;
c0 @ s(t/8 + s(t/16) * 10 * e28) * e22 *
0.1;
c0 @ s(t/32+ s(t/100) * s(t/930) * 500) *
e8F * 0.1;
c0 @ s(t*#45/3 + s(t/4) * s(t/900) * 6 *
e4A) * e42 * 0.1;
c0 @ s(t*#45/3 + s(t/#14) * s(t/200) * 6 *
e8A) * e84 * 0.1;
```

7. DISTORSIÓN

Hay muchísimas formas de distorsionar una onda.

Ya que el resultado de la función seno está entre -1 y 1, podemos aplicarle un absoluto : `a()`

```
c0 @ a(s(t*#45/6 + s(t/#14/2) * s(t/200) * 6 * e8A)) * e82 * 0.15;
```

Módulo `%` es uno de los operadores más útiles en rampcode. Acá podemos usarlo como distorsión *sample&hold:*

```
c0 @ s(t/#25) * 100 % 10 / 10 * 0.1 * e22;
```

```
c0 @ s(t/#25) * 100 % 2 / 2 * 0.1 * e22;
```

```
c0 @ s(t/#25) * 1000 % 900 / 900 * 0.1 * e22;
```

```
c0 @ s(t/#25) * 1001 % 1000 / 1000 * 0.1 * e22;
```

```
c0 @ s(t/#25) * 1010 % 1000 / 1000 * 0.1 * e22;
```

8. UN POCO MÁS DE RUIDO

Ya que estamos haciendo ruido, probá:

```
c0 @ s(t*t) * 0.1 * eFF;

c0 @ s(t*s(t)) * 0.1 * e4F;

c0 @ s(t%64) * 0.025;

c0 @ s(t*(t%64)) * 0.025;

c0 @ s(t*(t%8)) * 0.025;

c0 @ s(t*(t%X4F)) * 0.025;
```

9. EJEMPLO COPADO DE DUBSETP

Apliquemos todo lo aprendido hasta ahora en un algoritmo de dubstep:

```
c0 @ s(t/8 + s(t/16) * 6 * e86) * e82 *
(#8F==1 ¦ x82>0.99) * 0.3 + s(t*s(t)) *
s(t/700) * 0.1 * e85 * (x41<0.7) + s(t*s(t
%32)) * e81 * (#8F==9) * 0.15 + a( s(t/32 +
s(t/32) * (x1E*10) * s(t/ (x2F*1000) )) ) *
0.25 * (X43!=1);
```

Y, sólo por diversión, agreguemos:

```
d0 @ (x26<0.5) * 0.7 @ x87 * 25; h0 @
(x19<0.25) @ X18*1000; p0 @ x8D*0.33+0.33;
```

La diferencia es notable. Acabamos de agregar algunos efectos...

10. CADENA DE EFECTOS

La cadena de efectos por canal va así:
`filtro → delay → s&h → paneo`

Limpiemos el canal y vamos de a poco.

```
cl@0;
```

FILTRO VCF

```
f<canal> @ <q> @ <frecuencia> ;
```

Ojo! `q` *no puede ser una expresión.*

```
f0 @ 6 @ 150;
```

Unos sintes retro tipo Vangelis (?)para probar el filtro que acabamos de preparar:

```
c0 @ (s(t/8/2)>s(t/9000)*0.5) *0.3 +
(s(t/6)>s(t/1000)*0.5) *0.3 +
(s(t/5*2)>s(t/450)*0.5) *0.4;
```

```
f0 @ 20 @ x40*9999;
```

```
cl@0;
```

DELAY DE TIEMPO VARIABLE

```
d <canal>@ <feedback> @ <delay en ms> ;
```

Usá `1000/<rampa>/<subdiv>` para ajustar el *tempo*

```
c0 @ s(t/(#45+1)) * e44 * 0.1;

d0 @ 0.5 @ 1000/1/8;

d0 @ x81 @ 1000/1/8;

d0 @ (x81>0.5)*0.5 @ 1000;

d0 @ 0.25 @ t%1000;

d0 @ 0.25 @ t%100;

d0 @ 0.5 @ a(s(t/8000))*100;

d0 @ 0.5 @ a(s(t/50))*5;

d0 @ x41*0.9 @ x82*50;
```

SAMPLE AND HOLD

```
h <canal> @ <wet/dry> @ <frecuencia> ;

c1@0;

c0 @ (s(t/(#17¦#14))+s(t/4/#52)) * 0.1 *e54;

h0 @ 1 @ 2000;

h0 @ X52-1 @ 4000;

h0 @ 1 @ X5F*500;
```

PANEO ESTÉREO

```
p <canal> @ <0-1>;
```

0.5 es el centro (por defecto)

```
p0@ x50;
```

```
p0@ x50*0.5+0.25;
```

```
p0@ x50*0.33+0.33;
```

```
cl@0;
```

11. MANIPULACIÓN DE ARCHIVOS .WAV

Los archivos .wav están en la carpeta /samples y tienen que estar nombrados como números (1.wav 2.wav, etc).

`l` carga un archivo en el canal y `w` lo lee:

```
l <canal> @ <número de archivo>;
w <canal> @ <amplitud> @ <cómo leer> ;
```

`W` es el total de samples en el archivo.
Cuidado, `<número de archivo>` no puede ser una expresion!

```
l0@1;
```

```
w0@0.25 @ t*5;
w0@0.25 @ t*2.5;
w0@0.25 @ t*10;
```

```
w0@0.25 @ t*5%(8000*5);
```

```
w0@0.25 @ i11*(W/3.3);
```

```
w0@0.25 @ i41*(W/16)+W*x10;
```

```
w0@0.25 @ t/2+t*5%1000;
```

```
w0@0.25 * e41 @t/2+t*5%1000;
```

Cuidado con enviar números que no sean enteros a `%`, puede crashear en pd-vanilla.

```
w0@0.25@t+t*5%(X4F*100);
```

12. OTRO EJEMPLO

```
r @ 0.9; w0 @ 0.25 * (x43<0.8) @ t + t*5 %
(X48*100); c1 @ s(t/8 + s(t/16) * X8F * 20 *
e8A) * (#88>#86) * e81 * 0.08 + s(t*s(t)) *
s(t/700) * e86 * 0.25 * (#88<=#86) +
(s(t*#13/#F5) > s(t/(x14*1000))) * 0.075 *
eF1 * (x43>0.8) + s(t*s(t%42)) * 0.15 * e1F;
```

Si ya te quemaste el cerebro podés limpiar todos los canales con `c1@8;`

13. BYTEBEAT

Bytebeat es la técnica en la que está inspirado rampcode pero -en rigor- el término byte hace alusión al uso de operadores lógicos *bitwise*. Es decir, el bytebeat propiamente dicho también toma como única input una rampa pero las operaciones que realiza son *comparaciones de bit por bit.*

1100	1100	1100
& 0110	| 0110	^ 0110
0100	1110	1010

El contador de rampcode incrementa 8000 *por ciclo* y -por defecto- llega a 128 ciclos antes de volver a iniciar. Por lo tanto, para obtener los resultados típicos de una operación de bytebeat hay que escalar la señal con `%256/256 (1111 1111)` o algo menor, según lo que se busque. Algunos ejemplos:

```
c0 @ ( t & t>>8 ) %64/64 *0.2;
```

```
c0 @ ( t*2 | t>>4 ) %256/256 *0.2;
c0 @ ( t ^ t%257 ) %256/256 *0.2;
c0 @ ( t*3 ^ t%250 | t>>4 ) %256/256 *0.2;
```

```
c0 @ ( t & t>>8 & t/16 ) %64/64 *0.2;
```

1100	0110
>>1	<<1
0110	1100

REFERENCIA RÁPIDA

CANALES Y EFECTOS

cA	canal A
@	separar argumentos
;	terminar mensaje
fA@B@C;	filtro VCF B=q C=frec.
dA@B@C;	delay B=feedback C=ms
hA@B@C;	s&h B=wet/dry C=frec.
pA @B;	paneo stereo B=<0-1>

FUNCIONES

s()	seno
a()	absoluto
I()	entero

ENVOLVENTES

eAB / iAB	envolventes (ciclo/A)^B
#AB	contador (ciclo/A)%B+1

RANDOM

xAB	random (ciclo/A) seed B
XAB	random entero %B

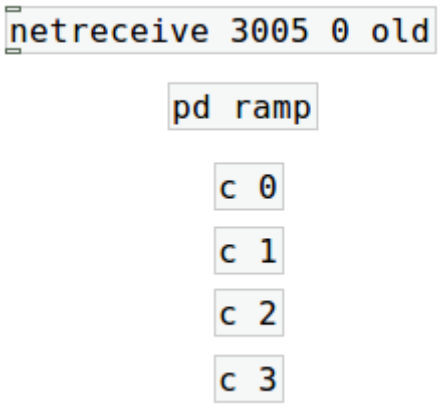
SAMPLES

lA @B;	cargar B en canal A
wA @B;	leer samples A según B

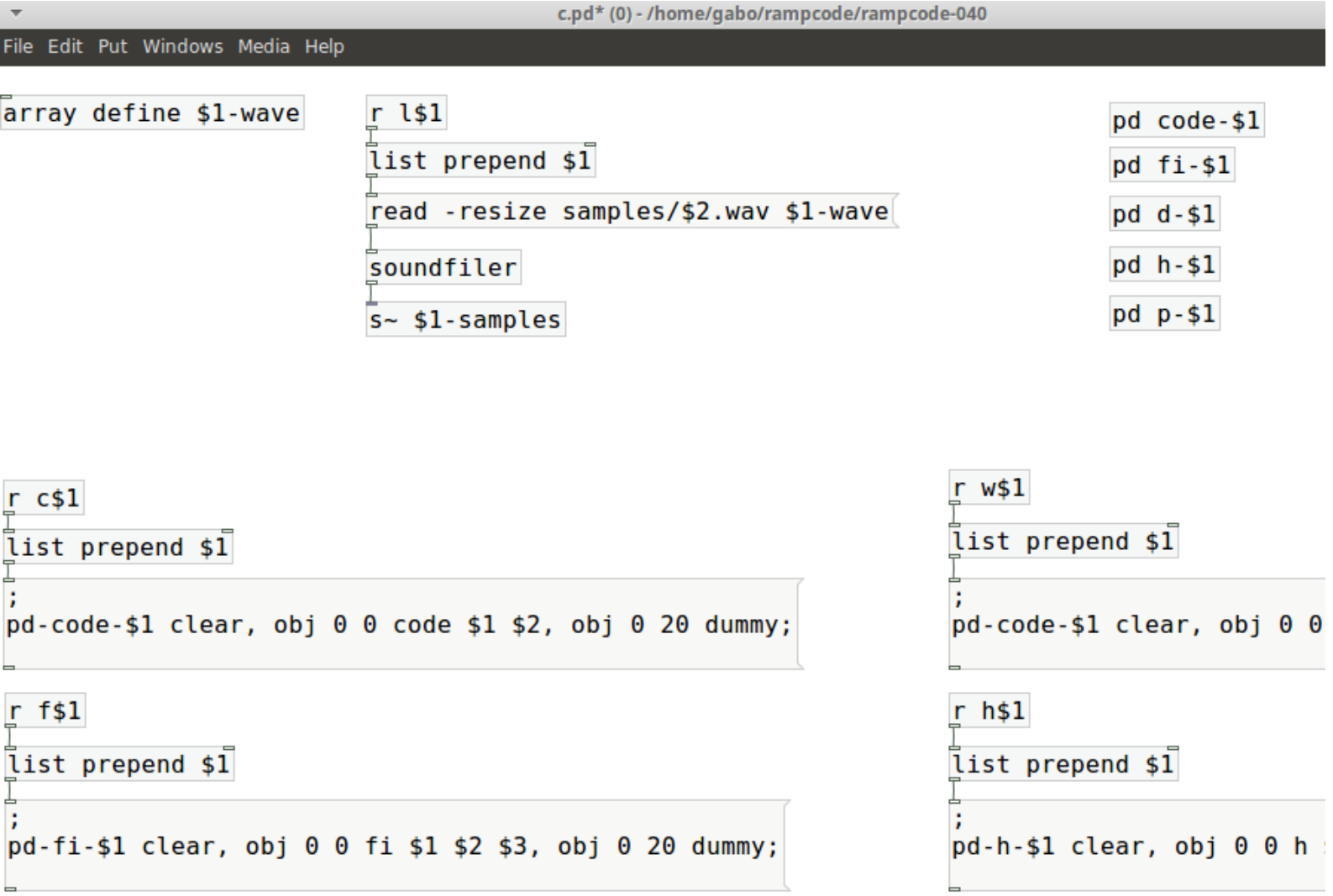
APÉNDICE

Rampcode está programado en *Pure Data* y fue testeado en Vanilla y L2ork. Básicamente es un parche que crea dinámicamente subparches con objetos *expr~* que reciben el valor de la rampa *phasor~* o el resultado de una operación previa de *expr~*.

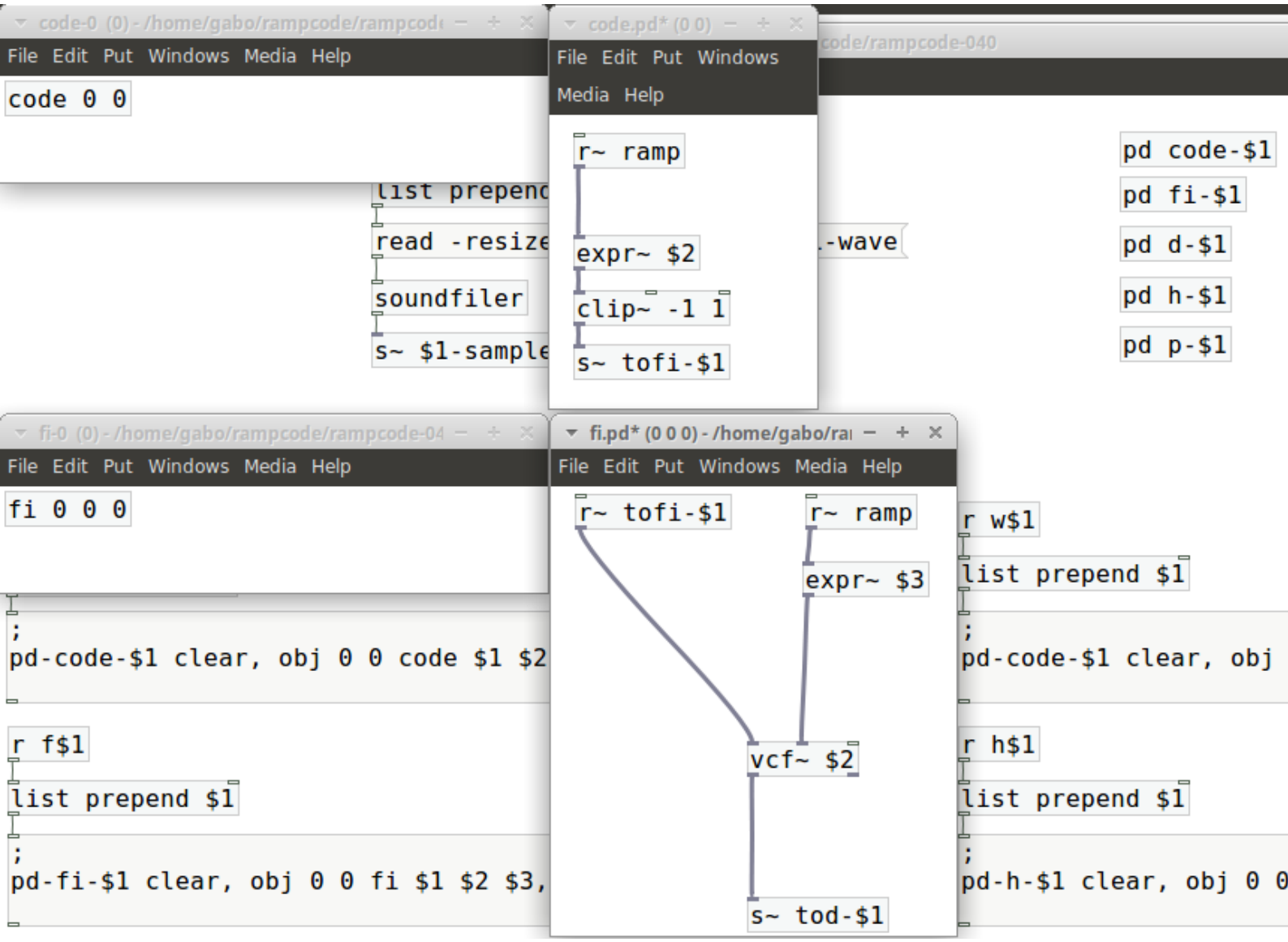
Los canales son abstracciones y pueden agregarse o quitarse para aumentar o disminuir la cantidad de canales que deseamos.



Dentro de cada canal encontramos un subparche que contiene los objetos creados dinámicamente. *code* es la expresión principal del canal y *fi*, *d*, *h* y *p* son la cadena de efectos.



El resultado parcial de cada efecto es enviado al siguiente con *send~* y *receive~*. Al recibir los mensajes correspondientes, el contenido de las abstracciones y subparches se actualiza.



Las abreviaturas se reemplazan con el script *sendline.sh* según las expresiones regulares definidas en *repl.sed*.

```
#!/bin/sh
text=`echo -n "$1" | sed -f repl.sed`
echo $text | pdsend 3005
```