

test1

September 12, 2011

music.lib/name	Music Library
music.lib/author	GRAME
music.lib/copyright	GRAME
music.lib/version	1.0
music.lib/license	LGPL
math.lib/name	Math Library
math.lib/author	GRAME
math.lib/copyright	GRAME
math.lib/version	1.0
math.lib/license	LGPL

This document provides a mathematical description of the Faust program text stored in the `test1.dsp` file. See the notice in Section 3 (page 3) for details.

1 Mathematical definition of process

The `test1` program evaluates the signal transformer denoted by `process`, which is mathematically defined as follows:

1. Output signals y_i for $i \in [1, 2]$ such that

$$y_1(t) = s_7(t)$$

$$y_2(t) = s_7(t)$$

2. Input signal (none)
3. User-interface input signal u_{s_1} such that

$$\text{"k"} \quad u_{s_1}(t) \in [-70, 4] \quad (\text{default value} = 0)$$

4. Intermediate signals p_1 , s_i for $i \in [1, 7]$, r_i for $i \in [1, 8]$ and v_1 such that

$$p_1(t) = 0.001 \cdot 10^{0.05 \cdot u_{s_1}(t)}$$

$$\begin{aligned}
s_1(t) &= k_2 + r_2(t-1) \\
s_2(t) &= k_3 + r_3(t-1) \\
s_3(t) &= k_4 + r_4(t-1) \\
s_4(t) &= k_5 + r_5(t-1) \\
s_5(t) &= k_6 + r_6(t-1) \\
s_6(t) &= k_7 + r_7(t-1) \\
s_7(t) &= r_8(t) \cdot (v_1[\text{int}(65536 \cdot r_7(t))] + v_1[\text{int}(65536 \cdot r_6(t))] \\
&\quad + v_1[\text{int}(65536 \cdot r_5(t))] + v_1[\text{int}(65536 \cdot r_4(t))] \\
&\quad + v_1[\text{int}(65536 \cdot r_3(t))] + v_1[\text{int}(65536 \cdot r_2(t))])
\end{aligned}$$

$$\begin{aligned}
r_1(t) &= 1 \oplus r_1(t-1) \\
r_2(t) &= s_1(t) - \lfloor s_1(t) \rfloor \\
r_3(t) &= s_2(t) - \lfloor s_2(t) \rfloor \\
r_4(t) &= s_3(t) - \lfloor s_3(t) \rfloor \\
r_5(t) &= s_4(t) - \lfloor s_4(t) \rfloor \\
r_6(t) &= s_5(t) - \lfloor s_5(t) \rfloor \\
r_7(t) &= s_6(t) - \lfloor s_6(t) \rfloor \\
r_8(t) &= p_1(t) + 0.999 \cdot r_8(t-1)
\end{aligned}$$

$$v_1[t] = \sin(9.58737992428526 \cdot 10^{-05} \cdot (r_1(t) \ominus 1)), \quad \text{when } t \in [0, 65535]$$

5. Constants k_i for $i \in [1, 7]$ such that

$$\begin{aligned}
k_1 &= f_S \\
k_2 &= \frac{440.329987}{k_1} \\
k_3 &= \frac{587}{k_1} \\
k_4 &= \frac{587.25}{k_1} \\
k_5 &= \frac{440}{k_1} \\
k_6 &= \frac{330.200012}{k_1} \\
k_7 &= \frac{330}{k_1}
\end{aligned}$$

2 Block diagram of process

The block diagram of `process` is shown on Figure 1 (page 3).

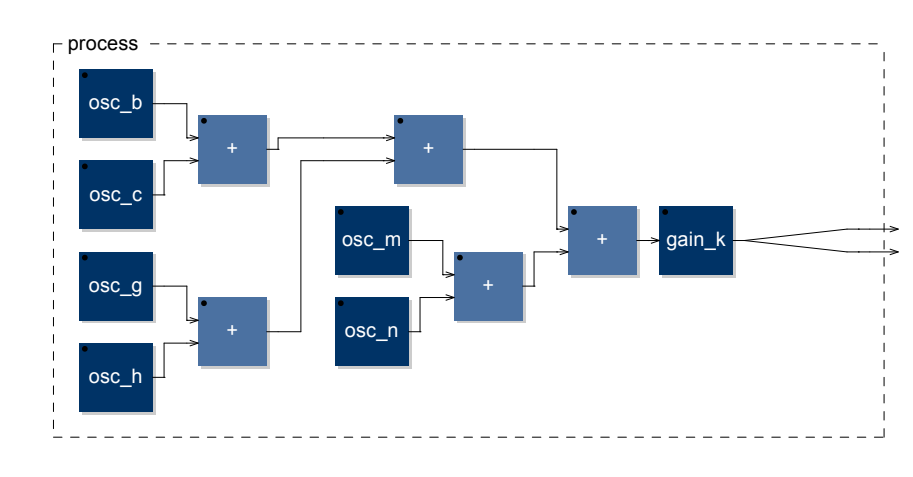


Figure 1: Block diagram of `process`

3 Notice

- This document was generated using Faust version 0.9.36 on September 12, 2011.
- The value of a Faust program is the result of applying the signal transformer denoted by the expression to which the `process` identifier is bound to input signals, running at the f_s sampling frequency.
- Faust (*Functional Audio Stream*) is a functional programming language designed for synchronous real-time signal processing and synthesis applications. A Faust program is a set of bindings of identifiers to expressions that denote signal transformers. A signal s in S is a function mapping¹ times $t \in \mathbb{Z}$ to values $s(t) \in \mathbb{R}$, while a signal transformer is a function from S^n to S^m , where $n, m \in \mathbb{N}$. See the Faust manual for additional information (<http://faust.grame.fr>).
- Every mathematical formula derived from a Faust expression is assumed, in this document, to having been normalized (in an implementation-dependent manner) by the Faust compiler.

¹Faust assumes that $\forall s \in S, \forall t \in \mathbb{Z}, s(t) = 0$ when $t < 0$.

- A block diagram is a graphical representation of the Faust binding of an identifier I to an expression E; each graph is put in a box labeled by I. Subexpressions of E are recursively displayed as long as the whole picture fits in one page.

- $\forall x \in \mathbb{R}$,

$$\text{int}(x) = \begin{cases} \lfloor x \rfloor & \text{if } x > 0 \\ \lceil x \rceil & \text{if } x < 0 \\ 0 & \text{if } x = 0 \end{cases} .$$

- This document uses the following integer operations:

<i>operation</i>	<i>name</i>	<i>semantics</i>
$i \oplus j$	integer addition	$\text{normalize}(i + j)$, in \mathbb{Z}
$i \ominus j$	integer subtraction	$\text{normalize}(i - j)$, in \mathbb{Z}

Integer operations in Faust are inspired by the semantics of operations on the n-bit two's complement representation of integer numbers; they are internal composition laws on the subset $[-2^{n-1}, 2^{n-1}-1]$ of \mathbb{Z} , with $n = 32$. For any integer binary operation \times on \mathbb{Z} , the \otimes operation is defined as: $i \otimes j = \text{normalize}(i \times j)$, with

$$\text{normalize}(i) = i - N \cdot \text{sign}(i) \cdot \left\lfloor \frac{|i| + N/2 + (\text{sign}(i) - 1)/2}{N} \right\rfloor,$$

where $N = 2^n$ and $\text{sign}(i) = 0$ if $i = 0$ and $i/|i|$ otherwise. Unary integer operations are defined likewise.

- The `test1-mdoc/` directory may also include the following subdirectories:
 - `cpp/` for Faust compiled code;
 - `pdf/` which contains this document;
 - `src/` for all Faust sources used (even libraries);
 - `svg/` for block diagrams, encoded using the Scalable Vector Graphics format (<http://www.w3.org/Graphics/SVG/>);
 - `tex/` for the L^AT_EX source of this document.

4 Faust code listings

This section provides the listings of the Faust code used to generate this document, including dependencies.

Listing 1: test1.dsp

```

1 //=====
2 //  Faust code for Max file :
3 //    test1.maxpat
4 //=====
5 import ("music.lib");
6
7 //=====
8 //  Messages resulting from translation from Max to Faust :
9 //    Message : This processor is connected in output to a digital to audio converter;
10 //    Message : This processor is connected in output to a digital to audio converter;
11 //<-----[ezdac~, , ID = obj-12]
12 //=====
13 // Comments in original Max patch :
14 // Expressions :
15 a = +; //<-----[newobj, +~]
16 osc_b = osc(330.000000); //<-----[newobj, cycle~ 330]
17 osc_c = osc(330.200012); //<-----[newobj, cycle~ 330.2]
18 e = +; //<-----[newobj, +~]
19 f = +; //<-----[newobj, +~]
20 osc_g = osc(440.000000); //<-----[newobj, cycle~ 440]
21 osc_h = osc(440.329987); //<-----[newobj, cycle~ 440.33]
22 i = _m2f_ezdac_bi_i with { _m2f_ezdac_bi_i(leftchannel, rightchannel) = leftchannel,
23 rightchannel;}; //<-----[ezdac~, ]
24 j = +; //<-----[newobj, +~]
25 gain_k = *(vslider("k", 0, -70, +4, 0.1) : db2linear : lisse(0.999)) with {lisse(c) =
26 *(1-c) : +*(c);}; //<-----[gain~, ]
27 l = +; //<-----[newobj, +~]
28 osc_m = osc(587.000000); //<-----[newobj, cycle~ 587]
29 osc_n = osc(587.250000); //<-----[newobj, cycle~ 587.25]
30
31 //Main Process :
32 process = (osc_b,osc_c:+),(osc_g,osc_h:+):(_,+), (osc_m,osc_n:+):_,_+:+:gain_k<:_,_+,_;

```

Listing 2: music.lib

```

1 /*****
2 *****/
3 FAUST library file
4 Copyright (C) 2003-2011 GRAME, Centre National de Creation Musicale
5 -----
6 This program is free software; you can redistribute it and/or modify
7 it under the terms of the GNU Lesser General Public License as
8 published by the Free Software Foundation; either version 2.1 of the
9 License, or (at your option) any later version.
10
11 This program is distributed in the hope that it will be useful,
12 but WITHOUT ANY WARRANTY; without even the implied warranty of
13 MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
14 GNU Lesser General Public License for more details.
15
16 You should have received a copy of the GNU Lesser General Public
17 License along with the GNU C Library; if not, write to the Free
18 Software Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA
19 02111-1307 USA.
20 *****/
21 *****/
22
23 declare name "Music Library";
24 declare author "GRAME";
25 declare copyright "GRAME";
26 declare version "1.0";

```

```

27 declare license "LGPL";
28
29 import("math.lib");
30
31 //-----
32 //          DELAY LINE
33 //-----
34 frac(n)          = n-int(n);
35 index(n)         = &(n-1) ~ +(1);          // n = 2**i
36 //delay(n,d,x) = rwtable(n, 0.0, index(n), x, (index(n)-int(d)) & (n-1));
37 delay(n,d,x)    = x@(int(d)&(n-1));
38 fdelay(n,d,x)  = delay(n,int(d),x)*(1 - frac(d)) + delay(n,int(d)+1,x)*frac(d);
39
40
41 delay1s(d)      = delay(65536,d);
42 delay2s(d)      = delay(131072,d);
43 delay5s(d)      = delay(262144,d);
44 delay10s(d)     = delay(524288,d);
45 delay21s(d)     = delay(1048576,d);
46 delay43s(d)     = delay(2097152,d);
47
48 fdelay1s(d)     = fdelay(65536,d);
49 fdelay2s(d)     = fdelay(131072,d);
50 fdelay5s(d)     = fdelay(262144,d);
51 fdelay10s(d)    = fdelay(524288,d);
52 fdelay21s(d)    = fdelay(1048576,d);
53 fdelay43s(d)    = fdelay(2097152,d);
54
55 millisec       = SR/1000.0;
56
57 time1s = hslider("time", 0, 0, 1000, 0.1)*millisec;
58 time2s = hslider("time", 0, 0, 2000, 0.1)*millisec;
59 time5s = hslider("time", 0, 0, 5000, 0.1)*millisec;
60 time10s = hslider("time", 0, 0, 10000, 0.1)*millisec;
61 time21s = hslider("time", 0, 0, 21000, 0.1)*millisec;
62 time43s = hslider("time", 0, 0, 43000, 0.1)*millisec;
63
64
65 echo1s = vgroup("echo 1000", +(delay(65536, int(hslider("millisecond", 0, 0, 1000, 0.10)*
66   millisec)-1) * (hslider("feedback", 0, 0, 100, 0.1)/100.0)));
67 echo2s = vgroup("echo 2000", +(delay(131072, int(hslider("millisecond", 0, 0, 2000, 0.25)*
68   millisec)-1) * (hslider("feedback", 0, 0, 100, 0.1)/100.0)));
69 echo5s = vgroup("echo 5000", +(delay(262144, int(hslider("millisecond", 0, 0, 5000, 0.50)*
70   millisec)-1) * (hslider("feedback", 0, 0, 100, 0.1)/100.0)));
71 echo10s = vgroup("echo 10000", +(delay(524288, int(hslider("millisecond", 0, 0, 10000,
72   1.00)*millisec)-1) * (hslider("feedback", 0, 0, 100, 0.1)/100.0)));
73 echo21s = vgroup("echo 21000", +(delay(1048576, int(hslider("millisecond", 0, 0, 21000,
74   1.00)*millisec)-1) * (hslider("feedback", 0, 0, 100, 0.1)/100.0)));
75 echo43s = vgroup("echo 43000", +(delay(2097152, int(hslider("millisecond", 0, 0, 43000,
76   1.00)*millisec)-1) * (hslider("feedback", 0, 0, 100, 0.1)/100.0)));
77
78
79 //-----sdelay(N,it,dt)-----
80 // s(smooth)delay : a mono delay that doesn't click and doesn't
81 // transpose when the delay time is changed. It takes 4 input signals
82 // and produces a delayed output signal
83 //
84 // USAGE : ... : sdelay(N,it,dt) : ...
85 //
86 // Where :
87 // <N> = maximal delay in samples (must be a constant power of 2, for example 65536)
88 // <it> = interpolation time (in samples) for example 1024
89 // <dt> = delay time (in samples)
90 // < > = input signal we want to delay
91 //-----
92
93 sdelay(N, it, dt) = ctrl(it,dt),_ : ddi(N)
94

```

```

89   with {
90
91       //-----ddi(N,i,d0,d1)-----
92       // DDI (Double Delay with Interpolation) : the input signal is sent to two
93       // delay lines. The outputs of these delay lines are crossfaded with
94       // an interpolation stage. By acting on this interpolation value one
95       // can move smoothly from one delay to another. When <i> is 0 we can
96       // freely change the delay time <d1> of line 1, when it is 1 we can freely change
97       // the delay time <d0> of line 0.
98       //
99       // <N> = maximal delay in samples (must be a power of 2, for example 65536)
100      // <i> = interpolation value between 0 and 1 used to crossfade the outputs of the
101      //      two delay lines (0.0: first delay line, 1.0: second delay line)
102      // <d0> = delay time of delay line 0 in samples between 0 and <N>-1
103      // <d1> = delay time of delay line 1 in samples between 0 and <N>-1
104      // < > = the input signal we want to delay
105      //-----
106      ddi(N, i, d0, d1) = _ <: delay(N,d0), delay(N,d1) : interpolate(i);
107
108
109      //-----ctrl(it,dt)-----
110      // Control logic for a Double Delay with Interpolation according to two
111      //
112      // USAGE : ctrl(it,dt)
113      // where :
114      // <it> an interpolation time (in samples, for example 256)
115      // <dt> a delay time (in samples)
116      //
117      // ctrl produces 3 outputs : an interpolation value <i> and two delay
118      // times <d0> and <d1>. These signals are used to control a ddi (Double Delay with
119      // Interpolation).
120      // The principle is to detect changes in the input delay time dt, then to
121      // change the delay time of the delay line currently unused and then by a
122      // smooth crossfade to remove the first delay line and activate the second one.
123      //
124      // The control logic has an internal state controlled by 4 elements
125      // <v> : the interpolation variation (0, 1/it, -1/it)
126      // <i> : the interpolation value (between 0 and 1)
127      // <d0>: the delay time of line 0
128      // <d1>: the delay time of line 1
129      //
130      // Please note that the last stage (!,_,_,_) cut <v> because it is only
131      // used internally.
132      //-----
133      ctrl(it, dt) = \ (v,ip,d0,d1).( nv, nip, nd0, nd1)
134      with {
135
136          // interpolation variation
137          nv = if (v!=0.0, // if variation we are interpolating
138                  if( (ip>0.0) & (ip<1.0), v , 0), // should we continue or not ?
139                  if ((ip==0.0) & (dt!=d0), 1.0/it, // if true xfade from d10 to d11
140                    if ((ip==1.0) & (dt!=d1), -1.0/it, // if true xfade from d11 to d10
141                      0)); // nothing to change
142
143          // interpolation value
144          nip = ip+nv : min(1.0) : max(0.0);
145
146          // update delay time of line 0 if needed
147          nd0 = if ((ip >= 1.0) & (d1!=dt), dt, d0);
148
149          // update delay time of line 1 if needed
150          nd1 = if ((ip <= 0.0) & (d0!=dt), dt, d1);
151
152          } ~ (_,_,_,_) : (!,_,_,_);
153
154      };
155
156      //-----
157      //      Tempo, beats and pulses

```

```

156 //-----
157
158 tempo(t) = (60*SR)/t;      // tempo(t) -> samples
159
160 period(p) = %(int(p))^(1); // signal en dent de scie de periode p
161 pulse(t) = period(t)==0;   // pulse (10000...) de periode p
162 pulsen(n,t) = period(t)<n; // pulse (1110000...) de taille n et de periode p
163 beat(t) = pulse(tempo(t)); // pulse au tempo t
164
165 //-----
166 // conversions between db and linear values
167 //-----
168
169
170 db2linear(x) = pow(10, x/20.0);
171 linear2db(x) = 20*log10(x);
172
173 //=====
174 //          Random and Noise generators
175 //=====
176
177 //-----
178 //          noise : Noise generator
179 //-----
180
181
182
183 random = +(12345) ^ *(1103515245);
184 RANDMAX = 2147483647.0;
185
186 noise = random / RANDMAX;
187
188 //-----
189 // Generates multiple decorrelated random numbers
190 // in parallel. Expects n>0.
191 //-----
192
193
194 multirandom(n) = randomize(n) ~_
195 with {
196     randomize (1) = +(12345) : *(1103515245);
197     randomize (n) = randomize(1) <: randomize(n-1),_;
198 };
199
200 //-----
201 // Generates multiple decorrelated noises
202 // in parallel. Expects n>0.
203 //-----
204
205
206 multinoise(n) = multirandom(n) : par(i,n,/(RANDMAX))
207 with {
208     RANDMAX = 2147483647.0;
209 };
210
211 //-----
212 //          noises(N,i) = multinoise(N) : selector(i,N);
213
214
215 //-----
216 //          osc(freq) : Sinusoidal Oscillator
217 //-----
218
219
220
221 tablesize = 1 << 16;
222 samplingfreq = SR;
223

```

```

224 time = +(1^-_) - 1; // 0,1,2,3,...
225 sinwaveform = float(time)*(2.0*PI)/float(ablesize) : sin;
226
227 decimal(x) = x - floor(x);
228 phase(freq) = freq/float(samplingfreq) : (+ : decimal) ^ _ : *(float(ablesize));
229 osc(freq) = rdtable(ablesize, sinwaveform, int(phase(freq)) );
230 osci(freq) = s1 + d * (s2 - s1)
231 with {
232     i = int(phase(freq));
233     d = decimal(phase(freq));
234     s1 = rdtable(ablesize+1,sinwaveform,i);
235     s2 = rdtable(ablesize+1,sinwaveform,i+1);};
236
237
238 //-----
239 //          ADSR envelop
240 //-----
241
242 // a,d,s,r = attack (#samples), decay (sec), sustain (percentage), release (sec)
243 // t = trigger signal
244
245 adsr(a,d,s,r,t) = env ^ (_,_) : (!,_) // the 2 'state' signals are fed back
246 with {
247     env (p2,y) =
248         (t>0) & (p2|(y>=1)), // p2 = decay-sustain phase
249         (y + p1*u - (p2&(y>s))*v*y - p3*w*y) // y = envelop signal
250         *((p3=0)|(y>=eps)) // cut off tails to prevent denormals
251     with {
252         p1 = (p2==0) & (t>0) & (y<1); // p1 = attack phase
253         p3 = (t<=0) & (y>0); // p3 = release phase
254         // #samples in attack, decay, release, must be >0
255         na = SR*a+(a==0.0); nd = SR*d+(d==0.0); nr = SR*r+(r==0.0);
256         // correct zero sustain level
257         z = s+(s==0.0)*db2linear(-60);
258         // attack, decay and (-60dB) release rates
259         u = 1/na; v = 1-pow(z, 1/nd); w = 1-1/pow(z*db2linear(60), 1/nr);
260         // values below this threshold are considered zero in the release phase
261         eps = db2linear(-120);
262     };
263 };
264
265
266 //-----
267 //          Spatialisation
268 //-----
269
270 panner(c) = _ <: *(1-c), *(c);
271
272 bus2 = _,_;
273 bus3 = _,-,_;
274 bus4 = _,-,-,_;
275 bus5 = _,-,-,-,_;
276 bus6 = _,-,-,-,-,_;
277 bus7 = _,-,-,-,-,-,_;
278 bus8 = _,-,-,-,-,-,-,_;
279
280 gain2(g) = *(g),*(g);
281 gain3(g) = *(g),*(g),*(g);
282 gain4(g) = *(g),*(g),*(g),*(g);
283 gain5(g) = *(g),*(g),*(g),*(g),*(g);
284 gain6(g) = *(g),*(g),*(g),*(g),*(g),*(g);
285 gain7(g) = *(g),*(g),*(g),*(g),*(g),*(g),*(g);
286 gain8(g) = *(g),*(g),*(g),*(g),*(g),*(g),*(g),*(g);
287
288
289 //-----
290 //
291 //          GMEM SPAT

```

```

292 // n-outputs spatializer
293 // implementation of L. Pottier
294 //
295 //-----
296 //
297 // n = number of outputs
298 // r = rotation (between 0 et 1)
299 // d = distance of the source (between 0 et 1)
300 //
301 //-----
302 spat(n,a,d) = _ <: par(i, n, *( scaler(i, n, a, d) : smooth(0.9999) ))
303   with {
304     scaler(i,n,a,d) = (d/2.0+0.5)
305       * sqrt( max(0.0, 1.0 - abs(fmod(a+0.5+float(n-i)/n, 1.0) - 0.5) * n * d
306         ) );
307     smooth(c) = *(1-c) : +~*(c);
308   };
309
310 //----- Second Order Generic Transfert Function -----
311 // TF2(b0,b1,b2,a1,a2)
312 //
313 //-----
314
315 TF2(b0,b1,b2,a1,a2) = sub ~ conv2(a1,a2) : conv3(b0,b1,b2)
316   with {
317     conv3(k0,k1,k2,x) = k0*x + k1*x' + k2*x'';
318     conv2(k0,k1,x)    = k0*x + k1*x';
319     sub(x,y)         = y-x;
320   };
321
322
323 /***** Break Point Functions *****/
324
325 bpf is an environment (a group of related definitions) tha can be used to
326 create break-point functions. It contains three functions :
327 - start(x,y) to start a break-point function
328 - end(x,y) to end a break-point function
329 - point(x,y) to add intermediate points to a break-point function
330
331 A minimal break-point function must contain at least a start and an end point :
332
333   f = bpf.start(x0,y0) : bpf.end(x1,y1);
334
335 A more involved break-point function can contains any number of intermediate
336 points
337
338   f = bpf.start(x0,y0) : bpf.point(x1,y1) : bpf.point(x2,y2) : bpf.end(x3,y3);
339
340 In any case the x_{i} must be in increasing order (for all i, x_{i} < x_{i+1})
341
342 For example the following definition :
343
344   f = bpf.start(x0,y0) : ... : bpf.point(xi,yi) : ... : bpf.end(xn,yn);
345
346 implements a break-point function f such that :
347
348   f(x) = y_{0} when x < x_{0}
349   f(x) = y_{n} when x > x_{n}
350   f(x) = y_{i} + (y_{i+1}-y_{i})*(x-x_{i})/(x_{i+1}-x_{i}) when x_{i} <= x and x < x_{i+1}
351
352 *****/
353
354 bpf = environment
355 {
356   // Start a break-point function
357   start(x0,y0) = \x).(x0,y0,x,y0);
358

```

```

359 // Add a break-point
360 point(x1,y1) = \(\x0,y0,x,y).(x1, y1, x, if (x < x0, y, if (x < x1, y0 + (x-x0)*(y1-y0)/(
      x1-x0), y1));
361
362 // End a break-point function
363 end (x1,y1) = \(\x0,y0,x,y).(if (x < x0, y, if (x < x1, y0 + (x-x0)*(y1-y0)/(x1-x0), y1));
364
365 // definition of if
366 if (c,t,e) = select2(c,e,t);
367 };

```

Listing 3: math.lib

```

1  /*****
2  *****/
3  FAUST library file
4  Copyright (C) 2003-2011 GRAME, Centre National de Creation Musicale
5  -----
6  This program is free software; you can redistribute it and/or modify
7  it under the terms of the GNU Lesser General Public License as
8  published by the Free Software Foundation; either version 2.1 of the
9  License, or (at your option) any later version.
10
11  This program is distributed in the hope that it will be useful,
12  but WITHOUT ANY WARRANTY; without even the implied warranty of
13  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
14  GNU Lesser General Public License for more details.
15
16  You should have received a copy of the GNU Lesser General Public
17  License along with the GNU C Library; if not, write to the Free
18  Software Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA
19  02111-1307 USA.
20  *****/
21  *****/
22
23  declare name "Math Library";
24  declare author "GRAME";
25  declare copyright "GRAME";
26  declare version "1.0";
27  declare license "LGPL";
28
29  //-----
30  //                               Mathematic library for Faust
31
32  // Implementation of the math.h file as Faust foreign functions
33  //
34  // History
35  // -----
36  // 28/06/2005 [YO]   postfix functions with 'f' to force float version
37  //                  instead of double
38  //                  [YO]   removed 'modf' because it requires a pointer as argument
39  //-----
40
41  // -- Utilities and constants
42
43  //SR           = min(192000, max(1, fconstant(int fSamplingFreq, <math.h>)));
44  SR           = fconstant(int fSamplingFreq, <math.h>);
45  BS           = fvariable(int count, <math.h>);
46
47  PI           = 3.1415926535897932385;
48
49  // -- neg and inv functions
50
51  neg(x)      = -x;

```

```

52 inv(x)      = 1/x;
53
54 // -- Trigonometric Functions
55
56 //acos      = ffunction(float acosf (float), <math.h>, "");
57 //asin      = ffunction(float asinf (float), <math.h>, "");
58 //atan      = ffunction(float atanf (float), <math.h>, "");
59 //atan2     = ffunction(float atan2f (float, float), <math.h>, "");
60
61 //sin       = ffunction(float sinf (float), <math.h>, "");
62 //cos       = ffunction(float cosf (float), <math.h>, "");
63 //tan       = ffunction(float tanf (float), <math.h>, "");
64
65 // -- Exponential Functions
66
67 //exp       = ffunction(float expf (float), <math.h>, "");
68 //log       = ffunction(float logf (float), <math.h>, "");
69 //log10     = ffunction(float log10f (float), <math.h>, "");
70 //pow       = ffunction(float powf (float, float), <math.h>, "");
71 //sqrt      = ffunction(float sqrtf (float), <math.h>, "");
72 cbrt       = ffunction(float cbrtf (float), <math.h>, "");
73 hypot      = ffunction(float hypotf (float, float), <math.h>, "");
74 ldexp      = ffunction(float ldexpf (float, int), <math.h>, "");
75 scalb      = ffunction(float scalbf (float, float), <math.h>, "");
76 log1p     = ffunction(float log1pf (float), <math.h>, "");
77 logb      = ffunction(float logbf (float), <math.h>, "");
78 ilogb     = ffunction(int ilogbf (float), <math.h>, "");
79 expm1     = ffunction(float expm1f (float), <math.h>, "");
80
81 // -- Hyperbolic Functions
82
83 acosh      = ffunction(float acoshf (float), <math.h>, "");
84 asinh      = ffunction(float asinhf (float), <math.h>, "");
85 atanh      = ffunction(float atanhf (float), <math.h>, "");
86
87 sinh       = ffunction(float sinhf (float), <math.h>, "");
88 cosh       = ffunction(float coshf (float), <math.h>, "");
89 tanh       = ffunction(float tanhf (float), <math.h>, "");
90
91 // -- Remainder Functions
92
93 //fmod      = ffunction(float fmodf (float, float), <math.h>, "");
94 //remainder = ffunction(float remainderf (float, float), <math.h>, "");
95
96 // -- Nearest Integer Functions
97
98 //floor     = ffunction(float floorf (float), <math.h>, "");
99 //ceil      = ffunction(float ceilf (float), <math.h>, "");
100 //rint     = ffunction(float rintf (float), <math.h>, "");
101
102 // -- Special Functions
103
104 erf        = ffunction(float erff(float), <math.h>, "");
105 erfc       = ffunction(float erfcf(float), <math.h>, "");
106 gamma     = ffunction(float gammaf(float), <math.h>, "");
107 J0         = ffunction(float j0f(float), <math.h>, "");
108 J1         = ffunction(float j1f(float), <math.h>, "");
109 Jn         = ffunction(float jnf(int, float), <math.h>, "");
110 lgamma    = ffunction(float lgammaf(float), <math.h>, "");
111 Y0         = ffunction(float y0f(float), <math.h>, "");
112 Y1         = ffunction(float y1f(float), <math.h>, "");
113 Yn         = ffunction(float ynf(int, float), <math.h>, "");
114
115
116 // -- Miscellaneous Functions
117
118 //fabs     = ffunction(float fabsf (float), <math.h>, "");
119 //fmax     = ffunction(float max (float, float), <math.h>, "");

```

```

120 //fmin      = ffunction(float min (float, float),<math.h>,"");
121
122 fabs = abs;
123 fmax = max;
124 fmin = min;
125
126 isnan      = ffunction(int isnan (float),<math.h>,"");
127 nextafter  = ffunction(float nextafter(float, float),<math.h>,"");
128
129 // Pattern matching functions to count and access the elements of a list
130 // USAGE : count ((10,20,30,40)) -> 4
131 //        take  (3,(10,20,30,40)) -> 30
132 //
133
134 count ((xs, xxs)) = 1 + count(xxs);
135 count (xx) = 1;
136
137 take (1, (xs, xxs)) = xs;
138 take (1, xs) = xs;
139 take (nn, (xs, xxs)) = take (nn-1, xxs);
140
141 // linear interpolation between two signals
142 interpolate(i) = *(1.0-i),*(i) : +;
143
144 // if-then-else implemented with a select2.
145 if(cond,thn,els) = select2(cond,els,thn);
146
147
148 //-----
149 // countdown(count,trig)
150 // start counting down from count, count-1,...,0 when trig > 0
151 //-----
152 countdown(count, trig) = \c).(if(trig>0, count, max(0, c-1))) ~_;
153
154 //-----
155 // countup(count,trig)
156 // start counting down from 0, 1, ... count-1, count when trig > 0
157 //-----
158 countup(count, trig) = \c).(if(trig>0, 0, min(count, c+1))) ~_;
159
160 /*****
161 * Hadamard matrix function
162 * Implementation contributed by Remy Muller
163 *****/
164
165 // bus(n) : n parallel cables
166 bus(2) = _,_ ; // avoids a lot of "bus(1)" labels in block diagrams
167 bus(n) = par(i, n, _);
168
169 // selector(i,n) : select ith cable among n
170 selector(i,n) = par(j, n, S(i, j)) with { S(i,i) = _; S(i,j) = !; };
171
172 // interleave(m,n) : interleave m*n cables : x(0), x(m), x(2m), ..., x(1),x(1+m), x(1+2m)...
173 //interleave(m,n) = bus(m*n) <: par(i, m, par(j, n, selector(i+j*m,m*n)));
174
175 // interleave(row,col) : interleave row*col cables from column order to row order.
176 // input : x(0), x(1), x(2) ..., x(row*col-1)
177 // output: x(0+0*row), x(0+1*row), x(0+2*row), ..., x(1+0*row), x(1+1*row), x(1+2*row), ...
178 interleave(row,col) = bus(row*col) <: par(r, row, par(c, col, selector(r+c*row,row*col)));
179
180 // butterfly(n) : addition then subtraction of interleaved signals :
181 butterfly(n) = bus(n) <: interleave(n/2,2), interleave(n/2,2) : par(i, n/2, +), par(i, n/2,
182 -);
183
184 // hadamard(n) : hadamard matrix function of size n = 2^k
185 hadamard(2) = butterfly(2);
186 hadamard(n) = butterfly(n) : (hadamard(n/2) , hadamard(n/2));

```