test2

Grame

September 13, 2011

name	test2
version	1.0
author	Grame
license	BSD
copyright	(c)GRAME 2007
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 `test2.dsp` file. See the notice in Section 3 (page 2) for details.

1 Mathematical definition of `process`

The *test2* 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

$$y_1(t) = r_1(t)$$

$$y_2(t) = r_2(t)$$

2. Input signals x_i for $i \in [1, 2]$
3. User-interface input signals u_{si} for $i \in [1, 2]$ such that
   - test2/echo 1000/

"millisecond"	$u_{s1}(t) \in [\,0, 1000\,]$	(default value = 0)
"feedback"	$u_{s2}(t) \in [\,0, 100\,]$	(default value = 0)

4. Intermediate signals p_i for $i \in [1, 2]$ and r_i for $i \in [1, 2]$ such that

$$p_1(t) = \text{int}\,(1 \oplus \text{int}\,(\text{int}\,(\text{int}\,(k_1 \cdot u_{s1}(t)) \ominus 1) \wedge 65535))$$

$$p_2(t) = 0.01 \cdot u_{s2}(t)$$

$$r_1(t) = x_1(t) + p_2(t) \cdot r_1(t - p_1(t))$$

$$r_2(t) = x_2(t) + p_2(t) \cdot r_2(t - p_1(t))$$

5. Constant k_1 such that

$$k_1 = 0.001 \cdot \min\,(192000, \max\,(1, f_S))$$

2 Block diagram of `process`

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

3 Notice

- This document was generated using Faust version 0.9.44 on September 13, 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 mapping[1] 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.

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

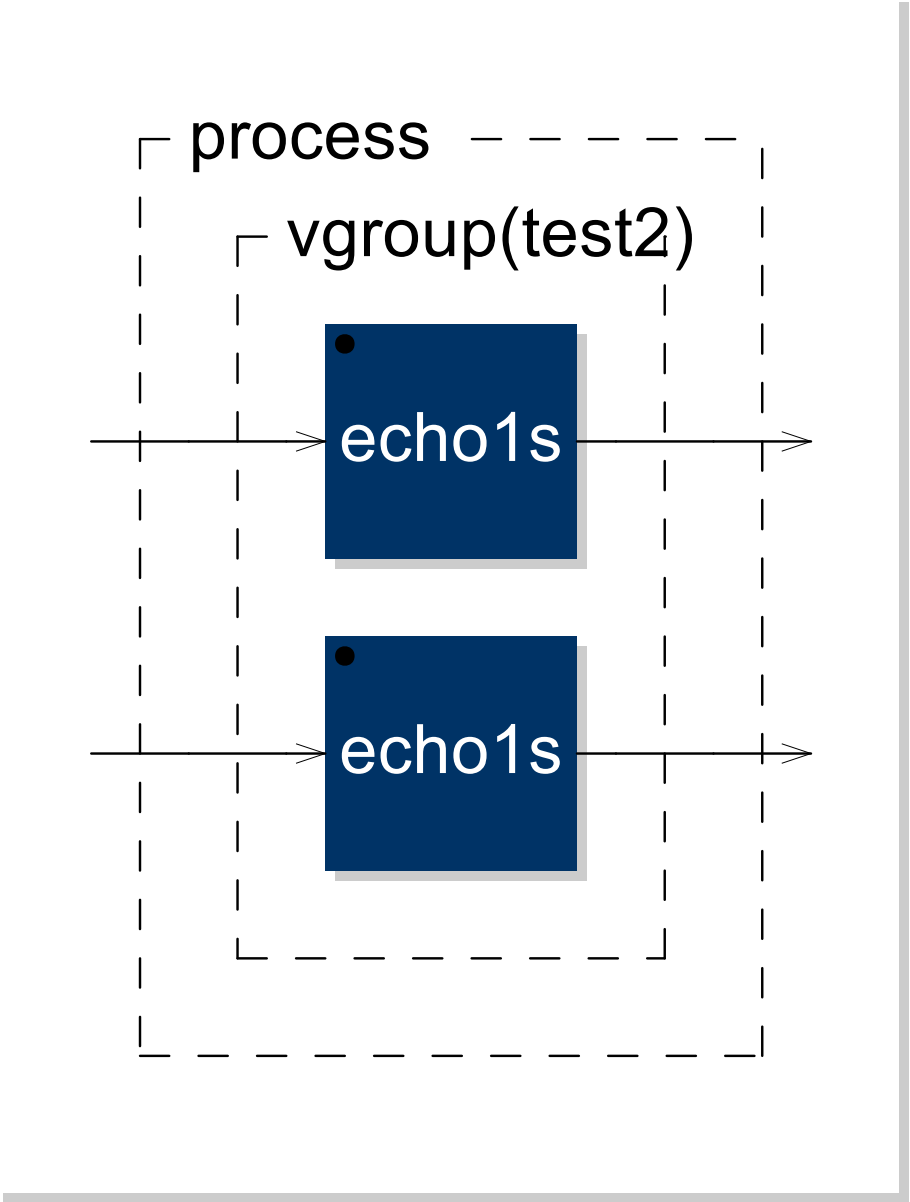

Figure 1: Block diagram of `process`

- 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.

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

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

- This document uses the following integer operations:

operation	*name*	*semantics*
$i \oplus j$	integer addition	normalize$(i+j)$, in $\mathbb{Z}$
$i \ominus j$	integer substraction	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: $i \otimes j = \mathrm{normalize}(i \times j)$, with

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

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

- The `test2-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 LaTeX 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: `test2.dsp`

```
declare name      "test2";
declare version   "1.0";
declare author    "Grame";
declare license   "BSD";
declare copyright  "(c)GRAME 2007";

//-----------------------------------------------
//            Test #2
```

```
//-----------------------------------------------

import("music.lib");

process = vgroup("test2", (echo1s, echo1s));
```

Listing 2: `music.lib`

```
/************************************************************************
 ************************************************************************
    FAUST library file
    Copyright (C) 2003-2011 GRAME, Centre National de Creation Musicale
    --------------------------------------------------------------------
    This program is free software; you can redistribute it and/or modify
    it under the terms of the GNU Lesser General Public License as
    published by the Free Software Foundation; either version 2.1 of the
    License, or (at your option) any later version.

    This program is distributed in the hope that it will be useful,
    but WITHOUT ANY WARRANTY; without even the implied warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
    GNU Lesser General Public License for more details.

    You should have received a copy of the GNU Lesser General Public
    License along with the GNU C Library; if not, write to the Free
    Software Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA
    02111-1307 USA.
 ************************************************************************
 ************************************************************************/

declare name "Music Library";
declare author "GRAME";
declare copyright "GRAME";
declare version "1.0";
declare license "LGPL";

import("math.lib");

//-----------------------------------------------
//                DELAY LINE
//-----------------------------------------------
frac(n)              = n-int(n);
index(n)             = &(n-1) ~ +(1);            // n = 2**i
//delay(n,d,x) = rwtable(n, 0.0, index(n), x, (index(n)-int(d)) & (n-1));
delay(n,d,x)  = x@(int(d)&(n-1));
fdelay(n,d,x) = delay(n,int(d),x)*(1 - frac(d)) + delay(n,int(d)+1,x)*frac(d);


delay1s(d)    = delay(65536,d);
delay2s(d)    = delay(131072,d);
delay5s(d)    = delay(262144,d);
delay10s(d)   = delay(524288,d);
delay21s(d)   = delay(1048576,d);
delay43s(d)   = delay(2097152,d);

fdelay1s(d)   = fdelay(65536,d);
fdelay2s(d)   = fdelay(131072,d);
fdelay5s(d)   = fdelay(262144,d);
fdelay10s(d)  = fdelay(524288,d);
fdelay21s(d)  = fdelay(1048576,d);
fdelay43s(d)  = fdelay(2097152,d);

millisec   = SR/1000.0;

```

```
time1s  = hslider("time", 0, 0, 1000, 0.1)*millisec;
time2s  = hslider("time", 0, 0, 2000, 0.1)*millisec;
time5s  = hslider("time", 0, 0, 5000, 0.1)*millisec;
time10s = hslider("time", 0, 0, 10000, 0.1)*millisec;
time21s = hslider("time", 0, 0, 21000, 0.1)*millisec;
time43s = hslider("time", 0, 0, 43000, 0.1)*millisec;


echo1s = vgroup("echo 1000", +~(delay(65536, int(hslider("millisecond", 0, 0, 1000, 0.10)*
     millisec)-1) * (hslider("feedback", 0, 0, 100, 0.1)/100.0)));
echo2s = vgroup("echo 2000", +~(delay(131072, int(hslider("millisecond", 0, 0, 2000, 0.25)*
     millisec)-1) * (hslider("feedback", 0, 0, 100, 0.1)/100.0)));
echo5s = vgroup("echo 5000", +~(delay(262144, int(hslider("millisecond", 0, 0, 5000, 0.50)*
     millisec)-1) * (hslider("feedback", 0, 0, 100, 0.1)/100.0)));
echo10s = vgroup("echo 10000", +~(delay(524288, int(hslider("millisecond", 0, 0, 10000,
     1.00)*millisec)-1) * (hslider("feedback", 0, 0, 100, 0.1)/100.0)));
echo21s = vgroup("echo 21000", +~(delay(1048576, int(hslider("millisecond", 0, 0, 21000,
     1.00)*millisec)-1) * (hslider("feedback", 0, 0, 100, 0.1)/100.0)));
echo43s = vgroup("echo 43000", +~(delay(2097152, int(hslider("millisecond", 0, 0, 43000,
     1.00)*millisec)-1) * (hslider("feedback", 0, 0, 100, 0.1)/100.0)));


//-------------------------sdelay(N,it,dt)---------------------------
// s(mooth)delay : a mono delay that doesn't click and doesn't
// transpose when the delay time is changed. It takes 4 input signals
// and produces a delayed output signal
//
// USAGE :  ... : sdelay(N,it,dt) : ...
//
// Where :
//  <N>  = maximal delay in samples (must be a constant power of 2, for example 65536)
//  <it> = interpolation time (in samples) for example 1024
//  <dt> = delay time (in samples)
//  <  > = input signal we want to delay
//-----------------------------------------------------------------------

sdelay(N, it, dt) = ctrl(it,dt),_ : ddi(N)

   with {

       //---------------------------ddi(N,i,d0,d1)-------------------------------
       // DDI (Double Delay with Interpolation) : the input signal is sent to two
       // delay lines. The outputs of these delay lines are crossfaded with
       // an interpolation stage. By acting on this interpolation value one
       // can move smoothly from one delay to another. When <i> is 0 we can
       // freely change the delay time <d1> of line 1, when it is 1 we can freely change
       // the delay time <d0> of line 0.
       //
       // <N>  = maximal delay in samples (must be a power of 2, for example 65536)
       // <i>  = interpolation value between 0 and 1 used to crossfade the outputs of the
       //        two delay lines (0.0: first delay line, 1.0: second delay line)
       // <d0> = delay time of delay line 0 in samples between 0 and <N>-1
       // <d1> = delay time of delay line 1 in samples between 0 and <N>-1
       // <  > = the input signal we want to delay
       //-----------------------------------------------------------------------
       ddi(N, i, d0, d1) = _ <: delay(N,d0), delay(N,d1) : interpolate(i);


       //---------------------------ctrl(it,dt)-----------------------------------
       //  Control logic for a Double Delay with Interpolation according to two
       //
       // USAGE : ctrl(it,dt)
       // where :
       // <it> an interpolation time (in samples, for example 256)
       // <dt> a delay time (in samples)
       //
       // ctrl produces 3 outputs : an interpolation value <i> and two delay
```

```
        //  times <d0> and <d1>. These signals are used to control a ddi (Double Delay with
              Interpolation).
        //  The principle is to detect changes in the input delay time dt, then to
        //  change the delay time of the delay line currently unused and then by a
        //  smooth crossfade to remove the first delay line and activate the second one.
        //
        //  The control logic has an internal state controlled by 4 elements
        //  <v> : the interpolation variation (0, 1/it, -1/it)
        //  <i> : the interpolation value (between 0 and 1)
        //  <d0>: the delay time of line 0
        //  <d1>: the delay time of line 1
        //
        //  Please note that the last stage (!,_,_,_) cut <v> because it is only
        //  used internally.
        //--------------------------------------------------------------------------
        ctrl(it, dt) = \(v,ip,d0,d1).( (nv, nip, nd0, nd1)
            with {

                // interpolation variation
                nv = if (v!=0.0,                          // if variation we are interpolating
                       if( (ip>0.0) & (ip<1.0), v , 0), //     should we continue or not ?
                    if ((ip==0.0) & (dt!=d0), 1.0/it,  // if true xfade from dl0 to dl1
                    if ((ip==1.0) & (dt!=d1), -1.0/it,  // if true xfade from dl1 to dl0
                    0)));                                 // nothing to change
                // interpolation value
                nip = ip+nv : min(1.0) : max(0.0);

                // update delay time of line 0 if needed
                nd0 = if ((ip >= 1.0) & (d1!=dt), dt, d0);

                // update delay time of line 0 if needed
                nd1 = if ((ip <= 0.0) & (d0!=dt), dt, d1);

            } ) ~ (_,_,_,_) : (!,_,_,_);
    };


//-----------------------------------------------
//         Tempo, beats and pulses
//-----------------------------------------------

tempo(t)   = (60*SR)/t;          // tempo(t) -> samples

period(p)  = %(int(p))~+(1);     // signal en dent de scie de periode p
pulse(t)   = period(t)==0;       // pulse (10000...) de periode p
pulsen(n,t) = period(t)<n;       // pulse (1110000...) de taille n et de periode p
beat(t)    = pulse(tempo(t));    // pulse au tempo t


//-----------------------------------------------
//  conversions between db and linear values
//-----------------------------------------------

db2linear(x)   = pow(10, x/20.0);
linear2db(x)   = 20*log10(x);


//===============================================
//         Random and Noise generators
//===============================================


//-----------------------------------------------
//         noise : Noise generator
//-----------------------------------------------

random     = +(12345) ~ *(1103515245);
RANDMAX    = 2147483647.0;
```

```
185
186 noise      = random / RANDMAX;
187
188
189 //-----------------------------------------------
190 // Generates multiple decorrelated random numbers
191 // in parallel. Expects n>0.
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 //-----------------------------------------------
202 // Generates multiple decorrelated noises
203 // in parallel. Expects n>0.
204 //-----------------------------------------------
205
206 multinoise(n) = multirandom(n) : par(i,n,/(RANDMAX))
207 with {
208     RANDMAX = 2147483647.0;
209 };
210
211
212 //-----------------------------------------------
213
214 noises(N,i) = multinoise(N) : selector(i,N);
215
216
217 //-----------------------------------------------
218 //         osc(freq) : Sinusoidal Oscillator
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(tablesize) : sin;
226
227 decimal(x) = x - floor(x);
228 phase(freq)    = freq/float(samplingfreq) : (+ : decimal) ~ _ : *(float(tablesize));
229 osc(freq)  = rdtable(tablesize, 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(tablesize+1,sinwaveform,i);
235            s2 = rdtable(tablesize+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
```

```
    p3 = (t<=0) & (y>0);              // p3 = release phase
    // #samples in attack, decay, release, must be >0
    na = SR*a+(a==0.0); nd = SR*d+(d==0.0); nr = SR*r+(r==0.0);
    // correct zero sustain level
    z = s+(s==0.0)*db2linear(-60);
    // attack, decay and (-60dB) release rates
    u = 1/na; v = 1-pow(z, 1/nd); w = 1-1/pow(z*db2linear(60), 1/nr);
    // values below this threshold are considered zero in the release phase
    eps = db2linear(-120);
    };
};


//-----------------------------------------------
//         Spatialisation
//-----------------------------------------------

panner(c) = _ <: *(1-c), *(c);

bus2 = _,_;
bus3 = _,_,_;
bus4 = _,_,_,_;
bus5 = _,_,_,_,_;
bus6 = _,_,_,_,_,_;
bus7 = _,_,_,_,_,_,_;
bus8 = _,_,_,_,_,_,_,_;

gain2(g) = *(g),*(g);
gain3(g) = *(g),*(g),*(g);
gain4(g) = *(g),*(g),*(g),*(g);
gain5(g) = *(g),*(g),*(g),*(g),*(g);
gain6(g) = *(g),*(g),*(g),*(g),*(g),*(g);
gain7(g) = *(g),*(g),*(g),*(g),*(g),*(g),*(g);
gain8(g) = *(g),*(g),*(g),*(g),*(g),*(g),*(g),*(g);


//-----------------------------------------------------
//
//                   GMEM SPAT
//  n-outputs spatializer
//  implementation of L. Pottier
//
//-----------------------------------------------------
//
//  n = number of outputs
//  r = rotation (between 0 et 1)
//   d = distance of the source (between 0 et 1)
//
//-----------------------------------------------------
spat(n,a,d) = _ <: par(i, n, *( scaler(i, n, a, d) : smooth(0.9999) ))
    with {
        scaler(i,n,a,d) = (d/2.0+0.5)
                        * sqrt( max(0.0, 1.0 - abs(fmod(a+0.5+float(n-i)/n, 1.0) - 0.5) * n * d
                              ) );
        smooth(c) = *(1-c) : +~*(c);
    };


//--------------- Second Order Generic Transfert Function -------------------------
// TF2(b0,b1,b2,a1,a2)
//
//-------------------------------------------------------------------------------

TF2(b0,b1,b2,a1,a2) = sub ~ conv2(a1,a2) : conv3(b0,b1,b2)
    with {
        conv3(k0,k1,k2,x)  = k0*x + k1*x' + k2*x'';
        conv2(k0,k1,x)     = k0*x + k1*x';
        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
```

```
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
    GNU Lesser General Public License for more details.

    You should have received a copy of the GNU Lesser General Public
    License along with the GNU C Library; if not, write to the Free
    Software Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA
    02111-1307 USA.
 ************************************************************************
 ************************************************************************/

declare name "Math Library";
declare author "GRAME";
declare copyright "GRAME";
declare version "1.0";
declare license "LGPL";

//-------------------------------------------------------------------------------
//                    Mathematic library for Faust

// Implementation of the math.h file as Faust foreign functions
//
// History
// ----------
// 28/06/2005  [YO]    postfixed functions with 'f' to force float version
//                     instead of double
//             [YO]    removed 'modf' because it requires a pointer as argument
//-------------------------------------------------------------------------------------

// -- Utilities and constants

SR         = min(192000, max(1, fconstant(int fSamplingFreq, <math.h>)));
BS         = fvariable(int count, <math.h>);

PI         = 3.1415926535897932385;

// -- neg and inv functions

neg(x)     = -x;
inv(x)     = 1/x;

// -- Trigonometric Functions

//acos     = ffunction(float acosf (float), <math.h>, "");
//asin     = ffunction(float asinf (float), <math.h>, "");
//atan     = ffunction(float atanf (float), <math.h>, "");
//atan2    = ffunction(float atan2f (float, float), <math.h>, "");

//sin          = ffunction(float sinf (float), <math.h>, "");
//cos          = ffunction(float cosf (float), <math.h>, "");
//tan          = ffunction(float tanf (float), <math.h>,"");

// -- Exponential Functions

//exp      = ffunction(float expf (float), <math.h>,"");
//log      = ffunction(float logf (float), <math.h>,"");
//log10    = ffunction(float log10f (float), <math.h>,"");
//pow      = ffunction(float powf (float, float), <math.h>,"");
//sqrt     = ffunction(float sqrtf (float), <math.h>,"");
cbrt       = ffunction(float cbrtf (float), <math.h>,"");
hypot      = ffunction(float hypotf (float, float), <math.h>,"");
ldexp      = ffunction(float ldexpf (float, int), <math.h>,"");
scalb      = ffunction(float scalbf (float, float), <math.h>,"");
log1p      = ffunction(float log1pf (float), <math.h>,"");
logb       = ffunction(float logbf (float), <math.h>,"");
ilogb      = ffunction(int ilogbf (float), <math.h>,"");
expm1      = ffunction(float expm1f (float), <math.h>,"");

// -- Hyperbolic Functions
```

```
81
82  acosh      = ffunction(float acoshf (float), <math.h>, "");
83  asinh      = ffunction(float asinhf (float), <math.h>, "");
84  atanh      = ffunction(float atanhf (float), <math.h>, "");
85
86  sinh       = ffunction(float sinhf (float), <math.h>, "");
87  cosh       = ffunction(float coshf (float), <math.h>, "");
88  tanh       = ffunction(float tanhf (float), <math.h>,"");
89
90  // -- Remainder Functions
91
92  //fmod     = ffunction(float fmodf (float, float),<math.h>,"");
93  //remainder    = ffunction(float remainderf (float, float),<math.h>,"");
94
95  // -- Nearest Integer Functions
96
97  //floor        = ffunction(float floorf (float), <math.h>,"");
98  //ceil     = ffunction(float ceilf (float), <math.h>,"");
99  //rint     = ffunction(float rintf (float), <math.h>,"");
100
101 // -- Special Functions
102
103 erf        = ffunction(float erff(float), <math.h>,"");
104 erfc       = ffunction(float erfcf(float), <math.h>,"");
105 gamma      = ffunction(float gammaf(float), <math.h>,"");
106 J0         = ffunction(float j0f(float), <math.h>,"");
107 J1         = ffunction(float j1f(float), <math.h>,"");
108 Jn         = ffunction(float jnf(int, float), <math.h>,"");
109 lgamma     = ffunction(float lgammaf(float), <math.h>,"");
110 Y0         = ffunction(float y0f(float), <math.h>,"");
111 Y1         = ffunction(float y1f(float), <math.h>,"");
112 Yn         = ffunction(float ynf(int, float), <math.h>,"");
113
114
115 // -- Miscellaneous Functions
116
117 //fabs     = ffunction(float fabsf (float), <math.h>,"");
118 //fmax     = ffunction(float max (float, float),<math.h>,"");
119 //fmin     = ffunction(float min (float, float),<math.h>,"");
120
121 fabs = abs;
122 fmax = max;
123 fmin = min;
124
125 isnan      = ffunction(int isnan (float),<math.h>,"");
126 nextafter  = ffunction(float nextafter(float, float),<math.h>,"");
127
128 // Pattern matching functions to count and access the elements of a list
129 // USAGE :  count ((10,20,30,40))  -> 4
130 //          take  (3,(10,20,30,40)) -> 30
131 //
132
133 count ((xs, xxs)) = 1 + count(xxs);
134 count (xx) = 1;
135
136 take (1, (xs, xxs))   = xs;
137 take (1, xs)          = xs;
138 take (nn, (xs, xxs))  = take (nn-1, xxs);
139
140 // linear interpolation between two signals
141 interpolate(i) = *(1.0-i),*(i) : +;
142
143 // if-then-else implemented with a select2.
144 if(cond,thn,els) = select2(cond,els,thn);
145
146
147 //-----------------------------------------------------------------
148 // countdown(count,trig)
```

```
// start counting down from count, count-1,...,0 when trig > 0
//-------------------------------------------------------------
countdown(count, trig) = \(c).(if(trig>0, count, max(0, c-1))) ~_;

//-------------------------------------------------------------
// countup(count,trig)
// start counting down from 0, 1, ... count-1, count when trig > 0
//-------------------------------------------------------------
countup(count, trig)  = \(c).(if(trig>0, 0, min(count, c+1))) ~_;

/*************************************************************
 *  Hadamard matrix function
 *  Implementation contributed by Remy Muller
 *************************************************************/

// bus(n) : n parallel cables
bus(2) = _,_; // avoids a lot of "bus(1)" labels in block diagrams
bus(n) = par(i, n, _);

// selector(i,n) : select ith cable among n
selector(i,n) = par(j, n, S(i, j)) with { S(i,i) = _; S(i,j) = !; };

// interleave(m,n) : interleave m*n cables : x(0), x(m), x(2m), ..., x(1),x(1+m), x(1+2m)...
//interleave(m,n) = bus(m*n) <: par(i, m, par(j, n, selector(i+j*m,m*n)));

// interleave(row,col) : interleave row*col cables from column order to row order.
// input : x(0), x(1), x(2) ..., x(row*col-1)
// 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), ...
interleave(row,col) = bus(row*col) <: par(r, row, par(c, col, selector(r+c*row,row*col)));

// butterfly(n) : addition then substraction of interleaved signals :
butterfly(n) = bus(n) <: interleave(n/2,2), interleave(n/2,2) : par(i, n/2, +), par(i, n/2,
     -);

// hadamard(n) : hadamard matrix function of size n = 2^k
hadamard(2) = butterfly(2);
hadamard(n) = butterfly(n) : (hadamard(n/2) , hadamard(n/2));
```