Faust Tutorial 2

GRAME, 9 rue du Garet, 69001 Lyon

*b*y Tiziano Bole
August 22, 2008

Special thanks to ...
for the support, the reading and the corrections.

Contents

Part I

Examples

Chapter 1

Panners

Introduction

In this chapter are shown several panners written in FAUST. Some of them can not be really used in most cases, and are intended only as intermediate steps. This is the case of the first two panners, described in sections 1 and 2 ("The Simplest Panner" and "2nd Simplest Panner" respectively); in section 3 ("Angle-Interpolated Panner") is shown a really working panner and is explained how to import FAUST libraries; section 4 ("Computational Overview") reports some observations about the computational cost of these panners and of later presented ones, and explains some C++ code; in section 5 ("Output-Interpolated Panner"), is shown a new "quite correct" panner, less expensive than *Angle-Interpolated* but equivalent to it in most cases; in section 6 ("Stereo Panners") are shown two stereo-input panners: *Angle-Interpolated Stereo Panner* (subsection 6.1) and *Output-Interpolated Stereo Panner* (subsection 6.2); finally a short conclusion will introduce you to the next chapter...

1.1 The Simplest Panner

Let's take the simplest panner we could build in FAUST. It should be something like this:

```
1 //-----------------------------------------------
2 //          The Simplest Panner
3 //-----------------------------------------------
4
5 c = hslider("pan", 0.5, 0, 1, 0.01);
6 process = _ <: *(1-c), *(c);
```

In the `process` definition, we can see that a mono input is split into two outputs (`_ <:`) and then scaled with coefficients `1-c` and `c` respectively; `c` is the

pan control, defined in line 5, and varies between 0 and 1. The resulting .svg block-diagram is shown in Fig. 1.1.

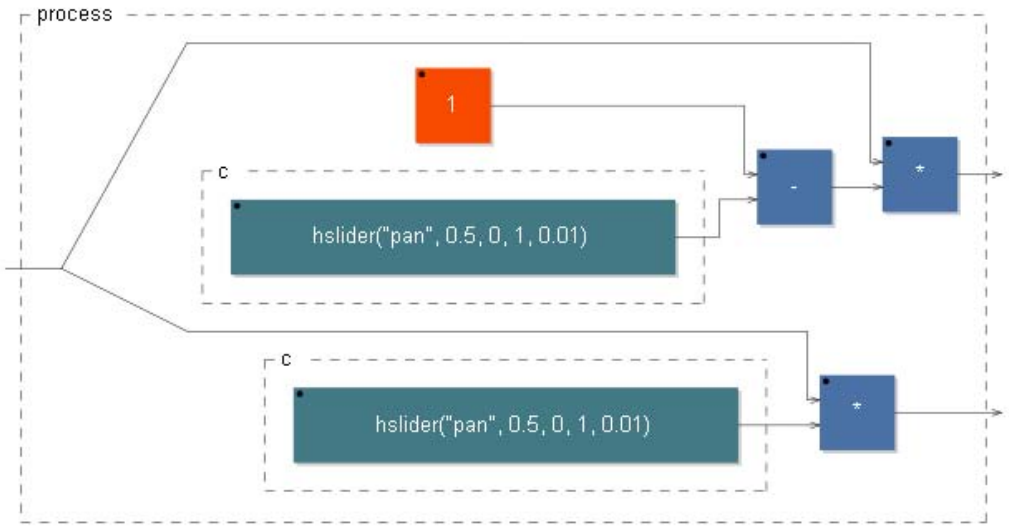

Figure 1.1: The "simplest panner" - block-diagram

So, when c is at its minimum value, 0, the first channel (that is the left one) will have the original amplitude (scale factor $1 - 0 = 1$) while the second one (that is of course the right one) will be silent (scale factor 0). When c is at its maximum value, 1, it will be just the opposite. With c at its central value, 0.5, the two outputs will have the same scale factor so we'll hear the sound at the center. Let's call these requested values **condition (1)**:

$$S(c) = \begin{cases} (1,0) & \text{if } c = 0 \text{ (pan to the left)} \\ (0,1) & \text{if } c = 1 \text{ (pan to the right)} \\ (x,x) & \text{if } c = 0.5 \text{ (centered pan)} \end{cases}$$

Unfortunately, **condition (1)** is not sufficient for a good panner, and this is basically for two reasons:

1. varying the pan control does affect the global intensity;

2. varying the pan control does produce clicks because of the non-continuous changing of the scale factors.

Let's see how to solve these problems.

1.2 Panner maintaining global intensity ("2nd Simplest Panner")

Imagine to use our panner on the channels of a mixer: we expect that changing the pan value of one channel doesn't affect its loudness, in order to let us changing the pans without affecting the mixing balance. In fact when we multiply a signal by a scaling factor (like the *S(c)* function in our panner) we are proportionally scaling its amplitude. But the loudness our ears perceive deals with its *intensity*, that is proportional to the square of the (average) amplitude[1]:

$$I \propto A^2$$

This means that if we want to maintain constant the global intensity in our panner, the sum of the *intensities* has to be constant, and not the sum of the *amplitudes*. Let's call this **condition (2)**:

$$I(c) = I_L(c) + I_R(c) \propto A_L^2(c) + A_R^2(c) = const.$$

So, the correct function to use for the output's scaling is not $S(c) = (1-c,\ c)$ but $S(c) = (\sqrt{1-c},\ \sqrt{c})$. In this way we will achieve both **condition (1)** and **condition (2)**. In fact:

$$A_L^2(c) + A_R^2(c) = A^2(\sqrt{1-c}^2 + \sqrt{c}^2) = A^2$$

where A is the original amplitude, and is invariant respect to c. So $S(c)$ meets **condition (2)**; it's easy to check that it meets also **condition (1)**.
We can now replace this $S(c)$ in our code:

```
1 //-----------------------------------------------
2 //       The Second Simplest Panner
3 //-----------------------------------------------
4
5 c = hslider("pan", 0.5, 0, 1, 0.01);
6 process = _ <: *((1-c) : sqrt), *((c) : sqrt);
```

I've simply "linked" the original outputs with a `sqrt` function in `process`. The name `sqrt` is very common in programming languages and in FAUST too stays for $\sqrt{}$. The link is made by the sequential[**?**] operator "`:`". The same `process` can be written without some parenthesis thanks to the operator's priorities (see [Orl07], pag. 4):

```
process = _ <: *(1 - c : sqrt), *(c : sqrt);
```

[1] see [Loy06], p. 123

or, if you prefer, in the more "usual" way:

```
process = _ <: *(sqrt(1 - c)), *(sqrt(c)) ;
```

You can see the resulting block-diagram in Fig. 1.2.

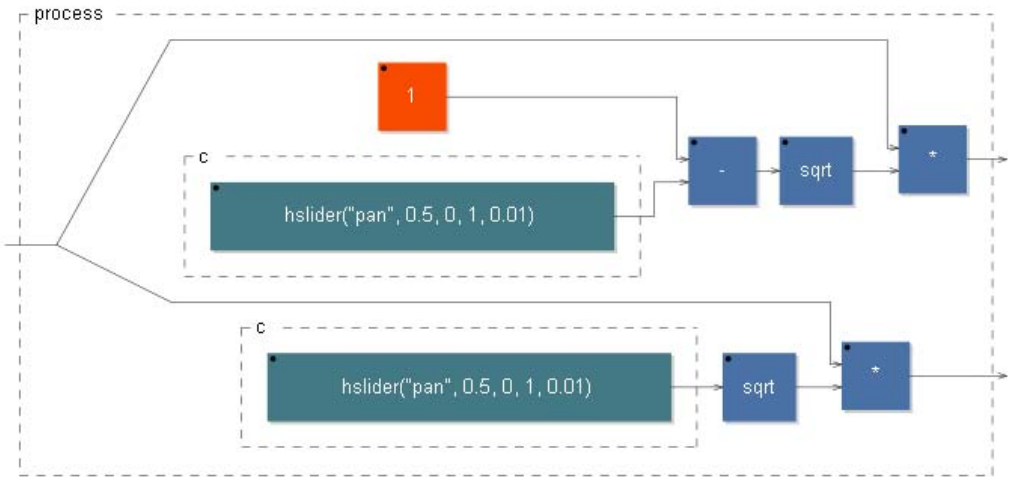

Figure 1.2: The "second simplest panner" - block-diagram

You can also see the graph of the new $S(c)$ function in Fig. 1.3; the two outputs in $c = 0.5$ have the same value, significantly greater than 0.5, that was the old $S(0.5)$ value. This means that the old function $S(c) = (1 - c,\ c)$ "sounded" too soft for the centered pan.

We have now solved the first problem, we still need to solve the second one: the clicks.

1.3 Panner with interpolation (Angle-Interpolated Panner)

The problem deals with the fact that c changes at *control rate* (minor than the *sampling rate*) because it's a slider's value, not an input signal's one, and FAUST upgrades it every n samples, depending on the host platform settings. So as you can smoothly move it, it will always be constant for n samples and then jump to a new value, and the output signal in that instant will change amplitude so quickly that probably you'll hear a click[**?**].[2]

The solution of this problem is interpolating in some way the consecutive

[2]It's like "cutting" at an arbitrary non-zero position the wave: the result is the typical appearance of a lot of components all over the spectrum that we call "click".

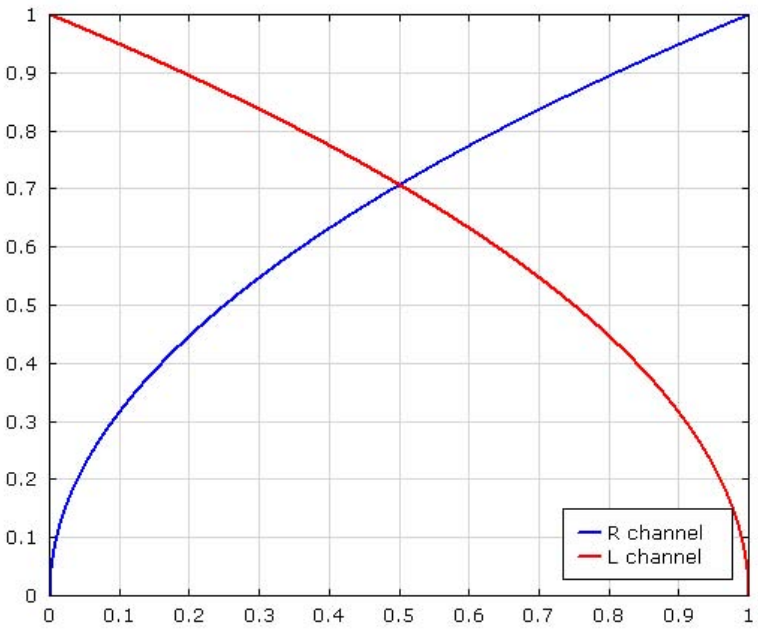

Figure 1.3: The $S(c) = (\sqrt{1-c},\ \sqrt{c})$ function graph

given values of c, distributing that step among the intermediate samples, so that then c varies at *sampling rate*. This operation is equivalent to filtering the c values, thought as a signal, with a low-pass filter. We can take, for example, the `smooth` function, defined in the FAUST library Filter.lib by Julius O. Smith. `smooth` is a low-pass filter that can take as argument an other function from the same library, `tau2pole`, that sets the `smooth`'s parameter so that `smooth(tau2pole(t))` becomes equivalent to an interpolator with t as *smoothing time* in seconds (i.e. in how much time we want the `smooth` function to slide between stepping values *quite completely*: see Fig. 1.4). We could choose an other low-pass function, but we had to set then its parameters manually, dealing not directly with a time quantity.

Let's insert this interpolation in `c` definition, sequentially after the slider reading, and also a new slider for the interpolative time value.

```
1 //-----------------------------------------------
2 //       Angle-Interpolated Panner
3 //-----------------------------------------------
4
5 import("filter.lib");
6 t = hslider("interpolation time", 0.001, 0, 0.01, 0.0001);
7 c = hslider("pan", 0.5, 0, 1, 0.01) : smooth(tau2pole(t));
8 process = _ <: *(1-c : sqrt), *(c : sqrt);
```

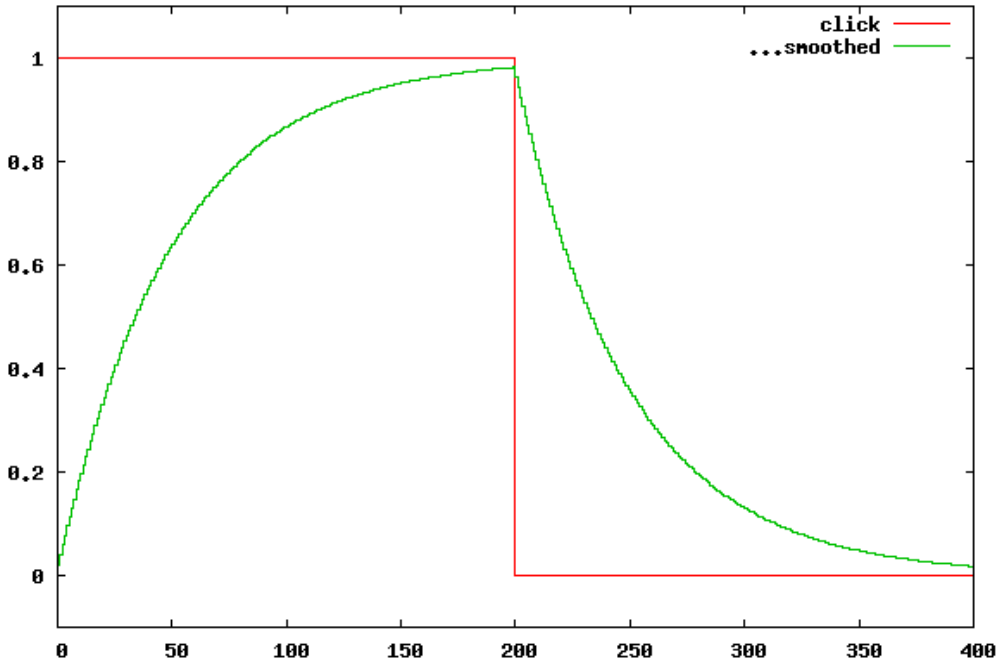

Figure 1.4: In green, the "smooth(tau2pole)" function applied to a signal with two steps (reported in red): 0-1 at sample 0 and 1-0 at sample 200. The "tau2pole" argument was set as the duration of 50 samples. You can see that the "smooth(tau2pole)" function **does not** reach the target value after 50 samples, because it follows the values only asymptotically, using the exponential function. In fact, coming from 0 value and rising to 1, after 50 samples it reaches only $1 - e^{-1} \approx 0.63$; it is very close to 1 only after 200 samples. A similar behavior can be seen on the "falling to 0" part, in which after 50 samples (at sample 250) the smoothed signal is at $e^{-1} \approx 0.37$ and is close to 0 only successively. This slowness of the "smooth(tau2pole)" function is not nasty in most cases, but if you need a more punctual interpolation, just set the tau2pole argument to a smaller value, for example a quarter of the desired time. See section xxx for an explanation about plotting signals.

We have to call of course the filter.lib library at the beginning of the .dsp code, so that FAUST will know the definition of the `smooth` and `tau2pole` functions. When you need some particular function, you can scan the .lib files you find in \architecture directory, opening them just like a .dsp code, with the same editor. When you import libraries, pay attention not to be using for your dsp code function names already used in the libraries you have imported, otherwise you'll get the error message:

```
ERROR: redefinition of symbols are not allowed :
BoxIdent[****] is already defined in file "****.lib" line **
```

An other way to use functions defined inside a FAUST library is, of course, copying their definition inside your .dsp code (and whatever other definition of function they could call).

Finally, you can see the .svg block-diagram in Fig. 1.5, with the insertion of the `smooth(tau2pole)` module. Note that two "c" blocks are shown in `process`: this doesn't mean that the "c" block is computed two times for each sample, this is only a graphic representation of a two-time use of the "c" block's value inside the `process` block.

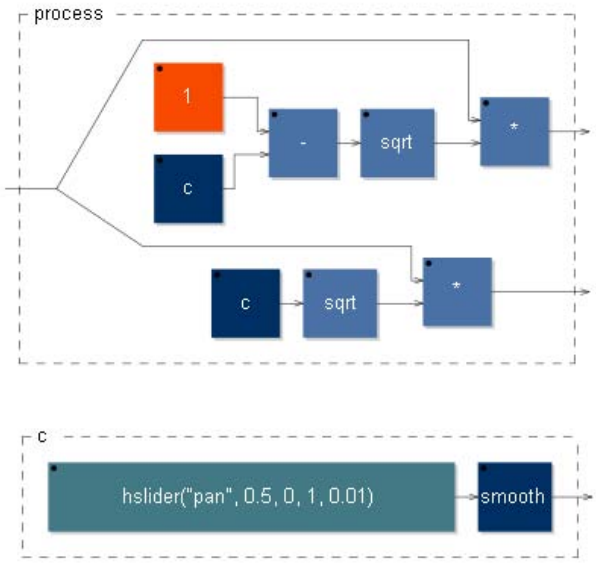

Figure 1.5: The Angle-Interpolated Panner - block-diagrams. The "c" module is explored in a second step. The "smooth" module is not explored because its definition has not been treated in this chapter and we will consider it as a ready-made function.

1.4 Computational overview

In the last section we got a "perfect working" panner, but at an important computational cost: in Fig 1.6 you can see the evaluation of the output bandwidth of some panners; until here, we have seen in the same order the first three panners: the other plot's panners will be discussed later in this chapter. The greater is the output bandwidth, the less expensive is the computation of that panner (see section ... for details about computational evaluation).

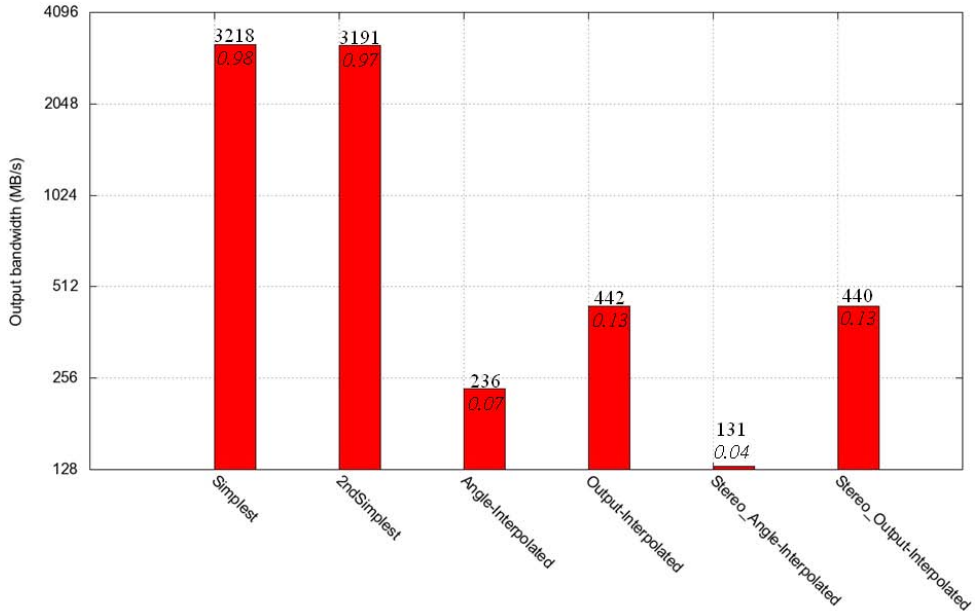

Figure 1.6: The output bandwidth of some panners (MB/s) tested on a Pentium 4, 2.66 GHz. The ratio with a reference value is shown in italics; the reference is a simply `.dsp` that splits an input into two outputs, and represents the minimal, without-calculations and common to all panners algorithmic unit; so it shows the maximum possible output bandwidth, limited only by reading and writing processes. Its value on the same computer is of about 3282 MB/s.

You can notice that *2ndSimplest* is computationally slightly more expensive than *Simplest*, of course because of the `sqrt` function adding. But you can see how the computational cost of *Angle-Interpolated*, the last panner we have discussed, is dramatically greater than the first two, more than 13 times greater! To understand this expensiveness, let's take a look to the C++ code generated by FAUST, in *Angle-Interpolated* and *2ndSimplest* cases (Fig 1.7). I remind you that you can get this code simply typing `Faust` and the name of the `.dsp` file, without further options.

I have evidenced the `sqrt` function position by a solid-line rectangle, and the

2nd simplest:

```
        [...]
        virtual void compute (int count, float** input, float** output) {
                float* input0 = input[0];
                float* output0 = output[0];
                float* output1 = output[1];
                float fSlow0 = fslider0;
                float fSlow1 = sqrtf((1 - fSlow0));
                float fSlow2 = sqrtf(fSlow0);
                for (int i=0; i<count; i++) {
                        float fTemp0 = input0[i];
                        output0[i] = (fSlow1 * fTemp0);
                        output1[i] = (fSlow2 * fTemp0);
                        // post processing
                }
        }
};
```

Angle-Interpolated:

```
       [...]
        virtual void compute (int count, float** input, float** output) {
                float* input0 = input[0];
                float* output0 = output[0];
                float* output1 = output[1];
                float fSlow0 = expf((0 - (fConst0 / fslider0)));
                float fSlow1 = (fslider1 * (1.000000f - fSlow0));
                for (int i=0; i<count; i++) {
                        fRec0[0] = (fSlow1 + (fSlow0 * fRec0[1]));
                        float fTemp0 = input0[i];
                        output0[i] = (fTemp0 * sqrtf((1 - fRec0[0])));
                        output1[i] = (fTemp0 * sqrtf(fRec0[0]));
                        // post processing
                        fRec0[1] = fRec0[0];
                }
        }
};
```

Figure 1.7: The C++ code generated by Faust in the 2ndSimplest and Angle-Interpolated cases. It is shown here only the end of the "minimal" code, without all the platform-depending part.

`for` cycle by a dashed-line one. The `sqrt` function is the one we used in our panners, while the `for` cycle is where the evaluation takes place at *sampling rate*; the code before it is evaluated at *control rate* instead. I remind you that the control rate is a submultiple of the sampling rate depending on the platform settings, so the `sqrt` function in the first case is evaluated once every n samples, while the same function in the second case is evaluated at *each* sample. This increases greatly the computational cost of the second case, because `sqrt` is a very expensive function (see chapter...), and so explains why *Angled-Interpolated* had a so greater computational cost than *2nd simplest*.

1.5 Output-Interpolated Panner

There are no less expensive ways to do the same work the *Angle-Interpolated* panner does, but we can try an approximation. We have to get the `sqrt` function's evaluation out of the `for` cycle, i.e. before it.

In the *Angle-Interpolated* panner, we had first the `smooth` function, just after the slider reading, and then the two `sqrt` functions. In fact, looking at Fig. 1.5, you can imagine the slider's data getting out of the `hslider` object and into the `smooth` one, all inside the "c" block; then in `process` this data flow gets out of that block and into the `sqrt`s, eventually passing through the `-` module. In other words, the `smooth` function's result becomes, with some mapping, the argument of the `sqrt` functions. In this way, we force the `sqrt` function to be evaluated at *sampling rate*, because its argument, requiring the evaluation of the function `smooth`, changes at that rate.

If we want the expensive `sqrt` functions to be evaluated at *control rate*, like in *2nd Simplest*, we have to do just the opposite, using the `sqrt` functions as arguments of two `smooth` functions, exchanging the order of the sequential link.

Here is the code of this new panner, its .`svg` block-diagram (Fig. 1.8) and the C++ generated code (Fig 1.9).

```
1 //-----------------------------------------------
2 //      Output-Interpolated Panner
3 //-----------------------------------------------
4
5 import("filter.lib");
6 t = hslider("interpolation time", 0.001, 0, 0.01, 0.0001);
7 c = hslider("pan", 0.5, 0, 1, 0.01);
8 process = _ <: *(1-c : sqrt) : smooth(tau2pole(t)),
9                (*(c : sqrt) : smooth(tau2pole(t)));
```

The computational cost of this panner is quite less than the *Angle-Interpolated* one, and it does *quite* the same work. The difference is that in *Angle-Interpolated* every value (also the ones in the interpolation phase) got into the $S(c)$ function

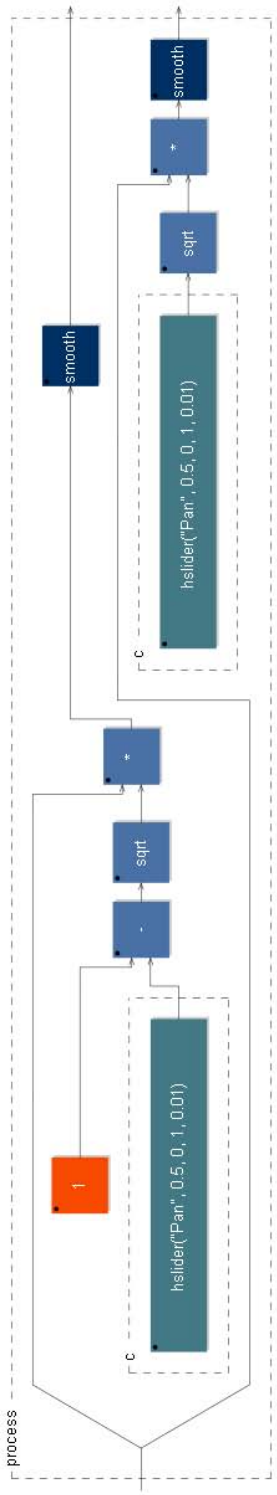

Figure 1.8: The Output-Interpolated Panner block-diagram.

```
virtual void compute (int count, float** input, float** output) {
        float* input0 = input[0];
        float* output0 = output[0];
        float* output1 = output[1];
        float fSlow0 = expf((0 - (fConst0 / fslider0)));
        float fSlow1 = (1.000000f - fSlow0);
        float fSlow2 = fslider1;
        float fSlow3 = (sqrtf((1 - fSlow2)) * fSlow1);
        float fSlow4 = (fSlow1 * sqrtf(fSlow2));
        for (int i=0; i<count; i++) {
                float fTemp0 = input0[i];
                fRec0[0] = ((fSlow0 * fRec0[1]) + (fSlow3 * fTemp0));
                output0[i] = fRec0[0];
                fRec1[0] = ((fSlow0 * fRec1[1]) + (fSlow4 * fTemp0));
                output1[i] = fRec1[0];
                // post processing
                fRec1[1] = fRec1[0];
                fRec0[1] = fRec0[0];
        }
}
};
```

Figure 1.9: The Output-Interpolated Panner C++ code (only "minimal" part).

(the one that needs the $\sqrt{}$: see section 2); so every value met what we called the **condition (2)**. In *Output-Interpolated* only the slider's values get into the $S(c)$ function, while the interpolation comes after: so **condition (2)** is met only by the slider's values (updated at control rate), and not by the intermediate interpolation ones. This difference should be appreciable only during slider's changing: in *Angle-Interpolated* you should listen always the same intensity, while in *Output-Interpolated* the intensity could change for some instants during a slider's step. So if the slider will change in a continuous way, you could hear only interpolation phases and the original intensity will be in some way corrupted. This is not a frequent case, but for example in acoustic experiments this could happen and you should use the *Angle-Interpolated*; for common musical cases feel free to use the *Output-Interpolated* instead. You can see a simulation of a typical pan movement using this panner in fig. 1.10.

1.6 Stereo Panners

The panners we've seen until now took always a mono input and split it with some amplitude corrections into two outputs. Let's call them *Mono Panners* because of the mono signal as input. How should we map now a stereo input into the two outputs? The idea I've followed is to mix the inputs into one output channel and leave the other one silent when the pan control is completely turned in one direction (left or right); to assign then the unchanged inputs to the respective outputs when the panner is at the center; let's pause and call these two sentences **condition (1b)**:

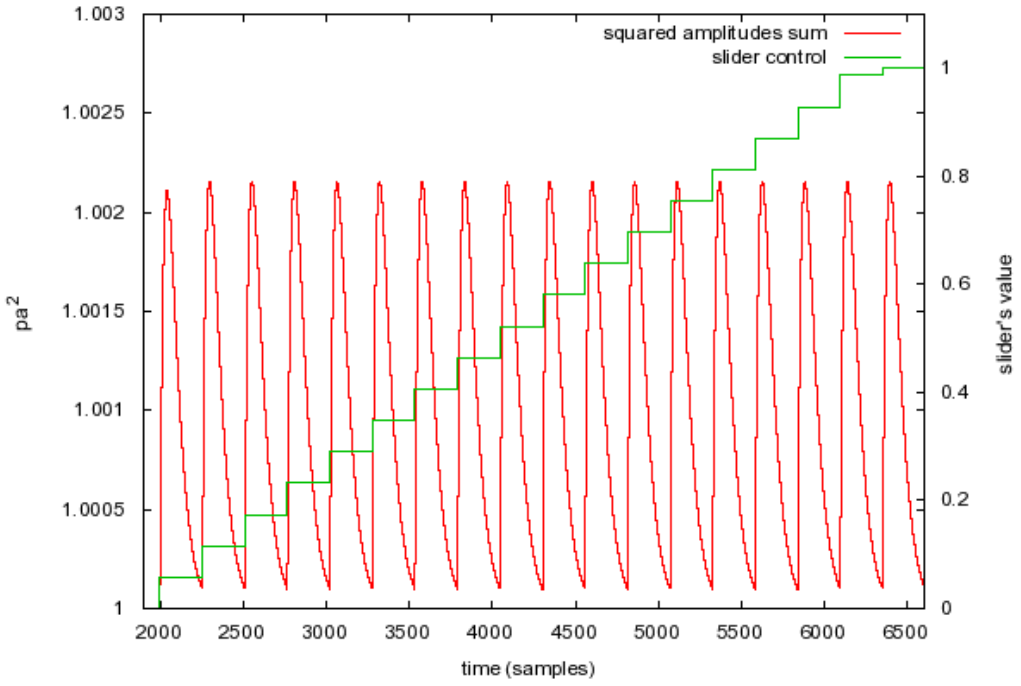

Figure 1.10: Simulation of a "pan movement" using the Output-Interpolated Panner. The green stepping line are the slider's values in time domine. The slider goes from 0 to 1 in 0.1 seconds, at a sampling rate of 44100 Hz and control blocks of 256 samples. So the slider's value is updated once every 256 samples, remaining constant during the rest of the control block. The slider's values are represented on the right scale (0..1). The red line represents the squared amplitudes sum of the two output channels. This value is proportional to the global intensity and should not change. With this panner it does change but in a very little range (left scale): from 1.002062 to 1.000100 pa^2, that's about 0.0085 dB, very minor than the just noticeable difference (JND). The interpolative time is set to 0.1 ms. So in a "normal" panner use, this panner works very well; however, significant differences could be if the pan control steps in a really short time between two distant values (like $(-1, 1)$).

$$(S'(c))(l,r) = \begin{cases} (l+r,\ 0) & \text{if } c = 0 \text{ (pan to the left)} \\ (0,\ l+r) & \text{if } c = 1 \text{ (pan to the right)} \\ (l,\ r) & \text{if } c = 0.5 \text{ (centered pan)} \end{cases}$$

$S'(c)$ is the new scaling function, c is the pan position value which range is $[0, 1]$, and (l, r) is the couple of input signal streams, that is the argument of $(S'(c))$ function.

Finally, we should use a square rooted amplitude function for the intermediate pan positions (of course, to respect **condition (2)**). The *amplitude vs. pan position* graph shown in Fig 1.11 will help you in understanding this idea. $S'(c)(l, r)$ is broken in four functions (see the figure's caption for details).

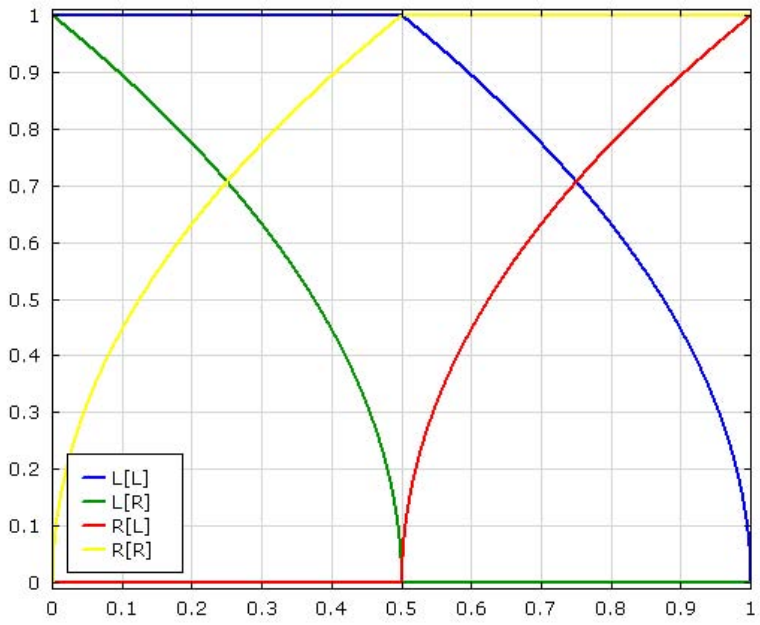

Figure 1.11: The "amplitude vs pan position" graph for a stereo input panner. $L[\bullet]$ and $R[\bullet]$ represent the amount of input sent respectively to left and right outputs; the meta-arguments are left $[L]$ and right $[R]$ inputs. For example, $L[R]$ is the amount of right input to be sent to left output, and is of course a function of c, so I'll write in this case $L[R](c)$; finally, I'll write $S'(c, l, r) = [...] = (Out_L, Out_R)$, where (l, r) are respectively the left and right input channels and (Out_L, Out_R) the output ones.

You can see by the graph that **condition (1b)** is met by these functions, since

when $c = 0$ (pan to the left), with (l, r) as inputs, the output is:

$$\begin{aligned} Out_L &= L[L](c{=}0) \cdot l + L[R](c{=}0) \cdot r = \\ &= 1 \cdot l + 1 \cdot r = \\ &= l + r \\ Out_R &= R[L](c{=}0) \cdot l + R[R](c{=}0) \cdot r = \\ &= 0 \cdot l + 0 \cdot r = \\ &= 0 \end{aligned}$$

So the output couple is $(Out_L,\ Out_R) = (l + r,\ 0)$.
In the same way you can see by the graph that:

$$\begin{aligned} &S'(c{=}1,\ l,\ r) = (0,\ l + r) \text{ and} \\ &S'(c{=}0.5,\ l,\ r) = (l,\ r) \end{aligned}$$

Let's see now the four partial functions in analytic form:

$$\begin{aligned} L[L](c) &\stackrel{def}{=} \sqrt{\min(1,\ 2 - 2c)} \\ L[R](c) &\stackrel{def}{=} \sqrt{\max(0,\ 1 - 2c)} \\ R[L](c) &\stackrel{def}{=} \sqrt{\max(0,\ 2c - 1)} \\ R[R](c) &\stackrel{def}{=} \sqrt{\min(1,\ 2c)} \end{aligned}$$

The core of these functions is the $\sqrt{c}$ function, eventually translated, stretched and inverted, by a coordinates exchange, in order to "place" the four functions into the plane by the needed way. The min and max functions are finally used to maintain the codomain into the range [0,1] and to avoid negative values under the $\sqrt{\ }$. You can verify that **condition (2)** is met by these functions – considering the sum of the four intensities given by all of them.

In a more synthetic way, we can write now the $S'(c)$ partial function as a 2×2 matrix, that we'll call A, and the application to the (l, r) inputs as an array multiplication.

$$A \stackrel{def}{=} \begin{pmatrix} L[L](c) & L[R](c) \\ R[L](c) & R[R](c) \end{pmatrix} =$$

$$= \begin{pmatrix} \sqrt{\min(1,\ 2 - 2c)} & \sqrt{\max(0,\ 1 - 2c)} \\ \sqrt{\max(0,\ 2c - 1)} & \sqrt{\min(1,\ 2c)} \end{pmatrix}$$

$$S'(c,l,r) \stackrel{def}{=} A \begin{pmatrix} l \\ r \end{pmatrix} =$$

$$= \begin{pmatrix} \sqrt{\min(1,\ 2-2c)} & \sqrt{\max(0,\ 1-2c)} \\ \sqrt{\max(0,\ 2c-1)} & \sqrt{\min(1,\ 2c)} \end{pmatrix} \begin{pmatrix} l \\ r \end{pmatrix} =$$

$$= \begin{pmatrix} \sqrt{\min(1,\ 2-2c)} \cdot l + \sqrt{\max(0,\ 1-2c)} \cdot r \\ \sqrt{\max(0,\ 2c-1)} \cdot l + \sqrt{\min(1,\ 2c)} \cdot r \end{pmatrix} =$$

$$= \begin{pmatrix} Out_L \\ Out_R \end{pmatrix}$$

Now we can use these (two) functions in a FAUST code, and make the two variants of an *Angle-Interpolated Stereo Panner* and an *Output-Interpolated* one.

1.6.1 Angle-Interpolated Stereo Panner

Let's see the .dsp code and the .svg block diagram (Fig 1.12).

```
1  //-----------------------------------------------
2  //    Angle-Interpolated Stereo Panner
3  //-----------------------------------------------
4
5  import("filter.lib");
6  t = hslider("interpolation time", 0.001, 0, 0.01, 0.0001);
7  c = hslider("pan", 0.5, 0, 1, 0.01) : smooth(tau2pole(t));
8
9  OutL(c,l,r) = sqrt(min(1,2-2*c))*l + sqrt(max(0,1-2*c))*r;
10 OutR(c,l,r) = sqrt(max(0,2*c-1))*l + sqrt(min(1,2*c))*r;
11 pan(c,l,r) = OutL(c,l,r)/2, OutR(c,l,r)/2;
12
13 process = pan(c);
```

The `OutL(c,l,r)` and `OutR(c,l,r)` functions are of course the two rows of the output array. The division /2 of the Out_L and Out_R functions is to avoid clipping when the two inputs are mixed together into one single channel (as in $c = 0$ and $c = 1$ cases).

1.6.2 Output-Interpolated Stereo Panner

And finally the *Output-Interpolated Stereo Panner*, in which the `smooth` function is commuted with the `sqrt`, like in the mono-input version, and also with the other arithmetic operations in this case.

```
1  //-----------------------------------------------
2  //    Output-Interpolated Stereo Panner
3  //-----------------------------------------------
4
5  import("filter.lib");
6  t = hslider("interpolation time", 0.001, 0, 0.01, 0.0001);
7  c = hslider("pan", 0.5, 0, 1, 0.01);
8
9  OutL(c,l,r) = sqrt(min(1,2-2*c))*l + sqrt(max(0,1-2*c))*r,;
10 OutR(c,l,r) = sqrt(max(0,2*c-1))*l + sqrt(min(1,2*c))*r;
11 pan(c,l,r) = OutL(c,l,r)/2, OutR(c,l,r)/2 :
12             : smooth(tau2pole(t)), smooth(tau2pole(t));
13
14 process = pan(c);
```

1.7 Conclusion

We have seen in this chapter how to build panners, i.e. objects that simulate a sound source's position by changing the level of left & right channels. For a better simulation, we should use also a delay control. In next chapter we'll see some utility objects first, then in the *Delay lines* following chapter we will see several delay line functions, eventually (but not only) used in space positioning objects.

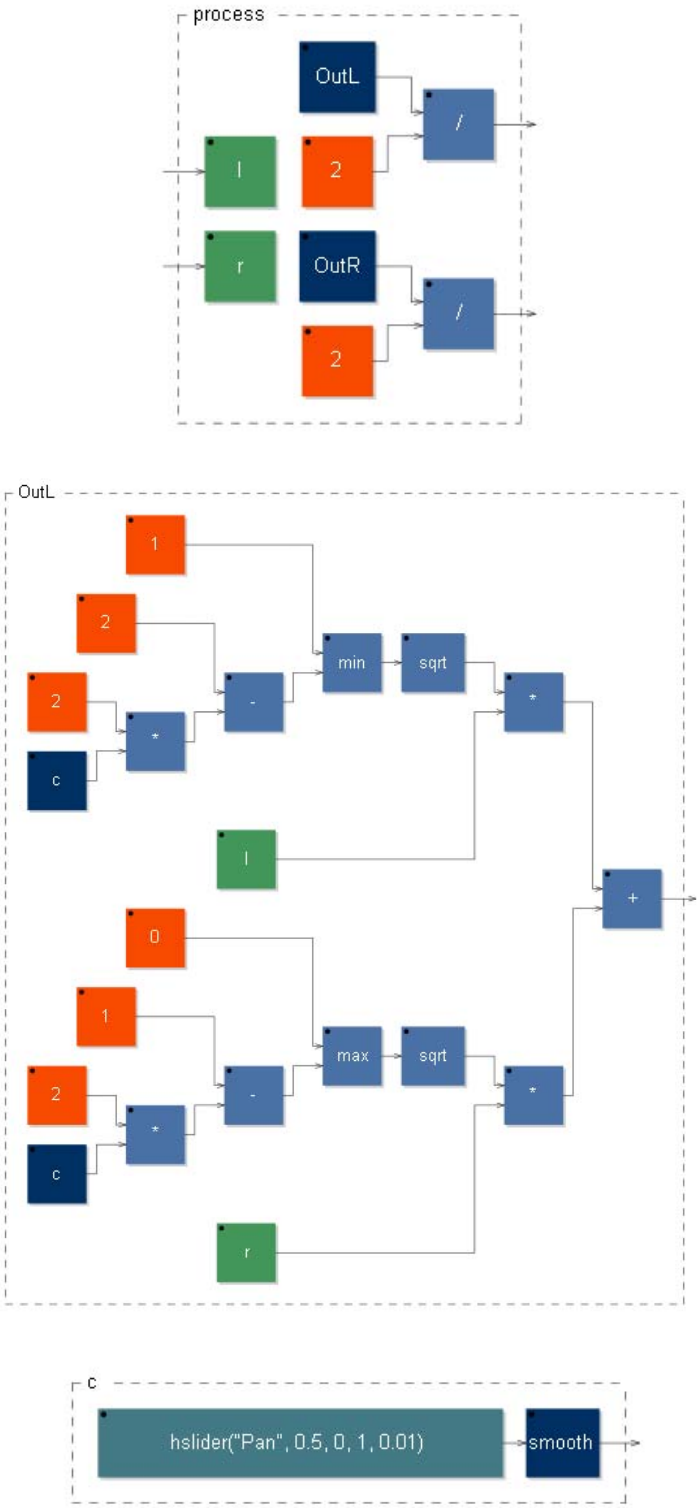

Figure 1.12: The "Angle-Interpolated Stereo Panner": block diagram (three steps). Here only left channel's processing is shown, the right's one is obviously similar to it.

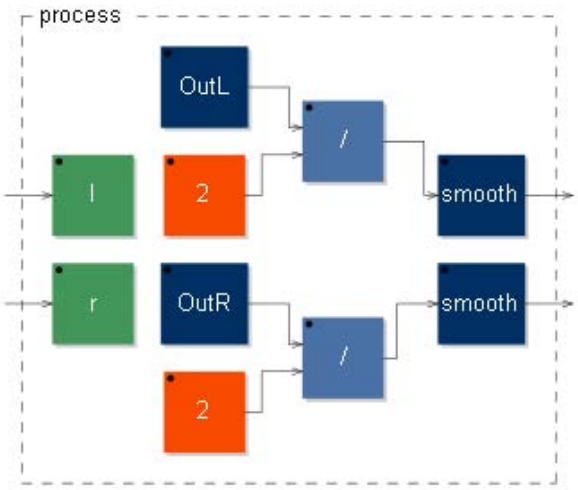

Figure 1.13: The "Output-Interpolated Stereo Panner": block diagram. "OutL" and "OutR" blocks, excepting for the "c" block, without "smooth" now, are the same of the "Angle-Interpolated Stereo Panner", so they are not shown here.

Chapter 2

Utility objects

Introduction

We need now to define some utility functions that will be necessary to build more complex objects. The first is a *Sample&Hold* (S&H), then a *Pitch tracker* using the signal's zero-crossing rate. This object will be developed in several steps: from a *Sinusoid Pitch Tracker* with a single-cycle analysis, to one with multiple-cycles analysis, ending with a *Universal Pitch Tracker*, working on an arbitrary signal. Finally you'll find and application example, a computational overview of the presented objects and a brief conclusion.

2.1 S&H

The *Sample&Hold* is a function that outputs the value of an incoming signal when it is triggered (*sample*), then it goes on to output that value (*hold*) as long as it doesn't receive a new triggering event. Then it starts to output the input signal's (new) value of that instant and so on. It will be essential in developing more complex objects. Let's see the FAUST code:

```
1 //-----------------------------------------------
2 //           S&H
3 //-----------------------------------------------
4
5 but        = button("Hold!");
6 SH(trig,x) = (*(1 - trig) + x * trig) ~ _;
7 process    = SH(but);
```

First there is a simple user interface: a button labelled "Hold!", which value is stored in the `but` variable. If triggered, the button will return the value 1, else the value 0. So these are the possible values of the `but` variable. Then there is the

`SH` function definition, in which the first argument, `trig`, is the triggering signal, while the second, `x`, the signal to be sampled. Finally, the `process` definition is the partial function `SH(but)`, in which the unique argument is the first one (the triggering signal, called `trig` in `SH` definition), while the second, unassigned, becomes an object inlet. So `but`, the button value, will trigger the inlet signal.

Let me explain you the `SH(trig,x)` definition. We want `SH` to be equal to `x` if `trig`= 1, and equal to its previous value else (t is the time instant):

$$SH_t(trig, x) = \begin{cases} x & \text{if } trig = 1 \\ SH_{t-1}(trig, x) & \text{if } trig = 0 \end{cases}$$

You will agree with me if I say that this function can be written in this single-line way:

$$SH_t(trig, x) = SH_{t-1}(trig, x) \cdot (1 - trig) + x \cdot trig$$

As long as $trig$ can have only the values $\{0, 1\}$, if it is 1 then the first addend is nulled and only the second one stays alive, if it is 0 it's just the opposite. Notice that if we allow also intermediate values for $trig$, then these two addends will represent a convex combination, i.e. a linear interpolation between $SH_{t-1}(trig, x)$ and x. Now, unfortunately in FAUST this writing has no sense, you can not define a function calling itself in this way. But you can use the `~` operator, that takes its right function's output and sends it back to its left function's input – of course, with a 1-sample delay. So, simply look at our single-line SH function, and just imagine to replace the call to SH_{t-1} with a cable outgoing from the same expression's ending. It becomes something like this:

$$SH_t(trig, x) = \hookrightarrow (\cdot(1 - trig) + x \cdot trig) \hookleftarrow$$

The arrows should represent this looping cable. This expression looks now very FAUST-like, we have only not to care about the instant t and to replace the symbols with the FAUST ones. Thus we obtain exactly the code line!

```
SH(trig,x) = (*(1 - trig) + x * trig) ~ _;
```

Let's see finally the `.svg` block-diagram in fig. 2.1. You can notice the "looping cable" outgoing from the output "cable" and turning back inside a `*` block.

2.2 Pitch tracker

The goal of a *pitch tracker* is to return the instant pitch of an input (periodic) signal. There are many ways to do so, like the *FFT method* or the *autocorrelation method* (see [Kuh90] for a discussion on these methods). Unfortunately, in FAUST these methods are not computably practicable, because actually you should apply them once per sample, even if it's not necessary, with a huge computational cost as result – in some future FAUST version this "limitation" could be removed. Instead,

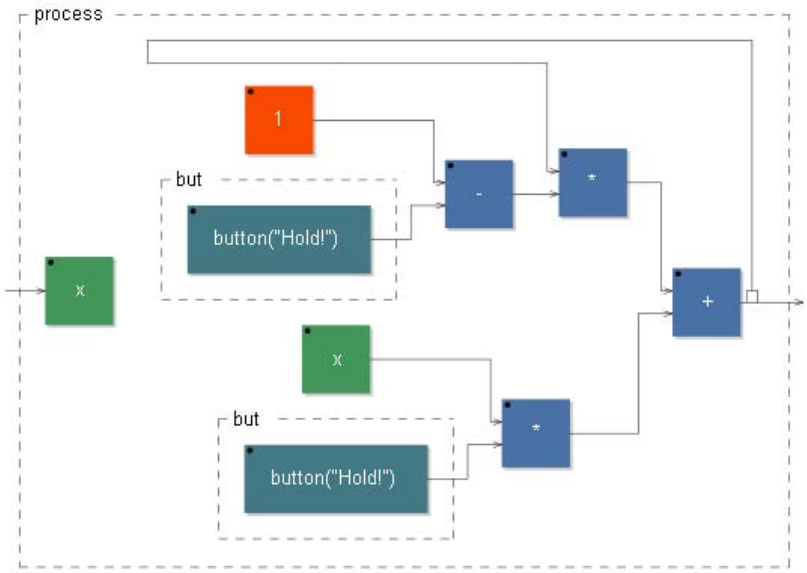

Figure 2.1: The "S&H" - block-diagram

we are going to implement a cheaper pitch tracker that returns the instant pitch analyzing the *zero-crossing rate* of the input signal. In fact, the wave period of a sinusoidal signal is exactly the time that elapses every three zeros (one at the beginning of the period, one at the passage from positive to negative, one at the ending of the period, that coincides with the next period's first zero); see fig. 2.2.

So, as long as we are dealing with sinusoids, we have simply to wait three zeros and to count the elapsed samples, then we can easily get the corresponding sinusoid's frequency. But what about if our input is a more complex waveform? We'll have to count more zeros because the wave could cross more times the zero value during each period. Or we can filter the signal with an adaptive lowpass filter, but we'll see this solution later. Let's develop a sinusoid pitch tracker first.

2.2.1 Sinusoid Pitch Tracker - one period measurement

Ok, the plain is that we build some simple functions, that one after the other gradually refine the signal period length information.

U function. The first of these function has to detect the zeros. To be more selective, we can ask only the zeros during which the signal is rising (not all the zeros then); for a sinusoidal signal, a period will be the elapsing time between only two of these kind of zeros. I've called this function $U(x_t)$, where x_t is the input

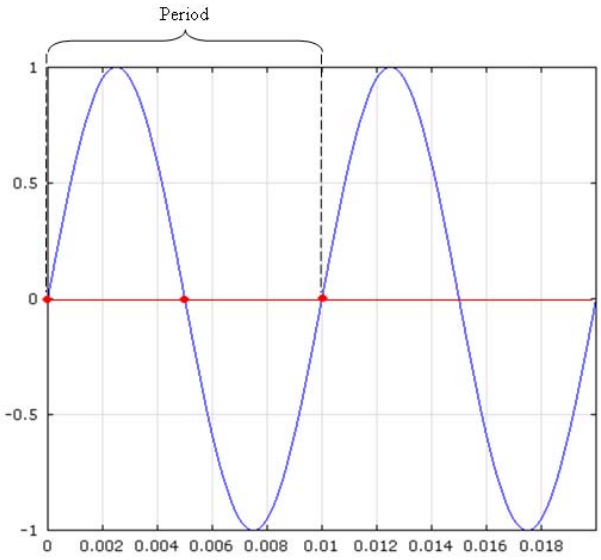

Figure 2.2: A 100 Hz sinusoid (time in seconds on the abscissa). The period takes 0.01 seconds, and you can see that it coincides with the time elapsed every three zeros.

signal at the instant t.

$$U(x_t) = \begin{cases} 1 & \text{if } x_{t-1} < 0 \text{ et } x_t \geq 0 \\ 0 & \text{else} \end{cases}$$

Notice that in the first condition I've wrote $x_t \geq 0$ and not $x_t = 0$, because in the discrete domine in which we are going to apply this function, we are not ensure that exactly the instant in which $x_t = 0$ will be sampled; thus we have to search two consecutive x_t of opposite sign, so that for the *Zeros theorem* we'll know that between them at least one zero had occured. We can express this U function in a FAUST code expression:

```
U(x) = (x' < 0) & (x >= 0);
```

where `'` is a single-sample delay line, so if `U(x)` is $U_t(x)$, then `x'` is x_{t-1}. The `&` operator is the logical AND, so it returns 1 (true) if the two conditions are true (1), else it returns 0 (false). You can see the $U(x_t)$ function graph in fig. 2.3 and the .svg block-diagram in fig. 2.4.

N function. We are interested in countering the time elapsed between two successive spikes of the $U(x_t)$ function. So we need a counter function – of course, we count the time in samples in this digital context. I've called this function $N(x_t)$, and it is 0 if $U(x_t) = 1$, else it rises each sample: so it counts the samples elapsed

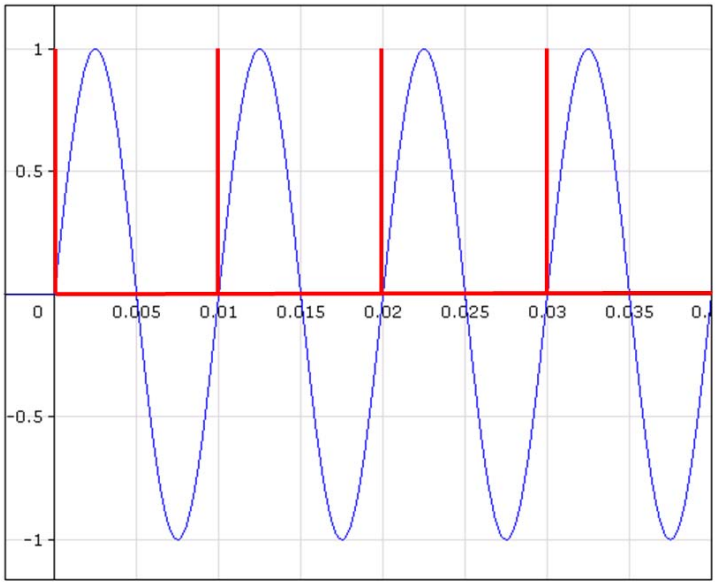

Figure 2.3: The $U(x_t)$ function (in red) with a 100 Hz sinusoid as argument (in blue). Time in seconds on the abscissa. A period is exactly the time elapsing between two $U(x_t)$ function spikes.

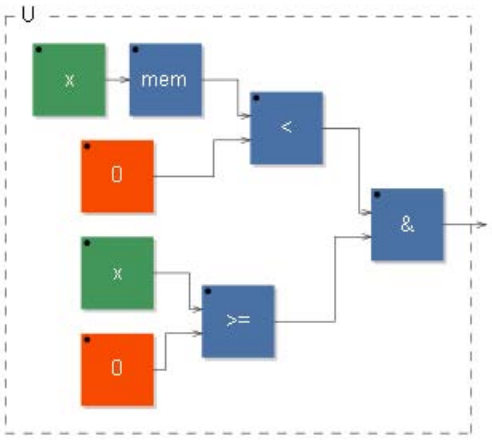

Figure 2.4: The U function block-diagram.

since the last $U(x_t)$ spike occurred (see fig. 2.5). The FAUST function definition is the usual counter (already presented in [GO03]), with the add of a multiplication that resets it to zero if $U(x_t) = 0$, and else leaves it untouched:

```
N(x) = (+(1) : *(1 - U(x))) ~ _;
```

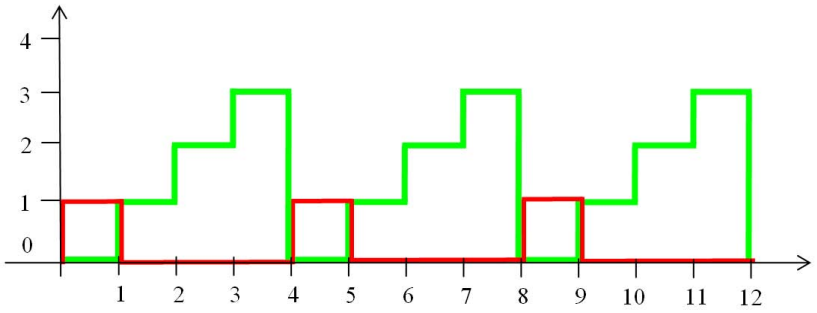

Figure 2.5: The $U(x_t)$ sampled function (in red) and the respective $N(x_t)$ function (in green). Time in samples on the abscissa, ordinate dimensionless. A period, that is also the time elapsing between two $U(x_t)$ function spikes, can be read from the last $N(x_t)$ value before a zero, plus 1.

You can see the N block diagram in fig. 2.6.

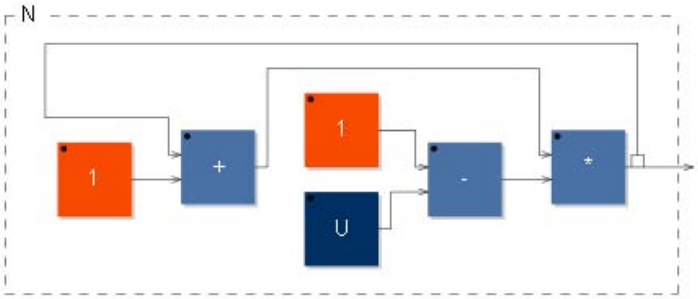

Figure 2.6: The N function block-diagram.

M function. So the original problem of determining the number of samples in one period, is equivalent to determining the last $N(x_t)$ value plus one, before a $N(x_t)$'s zero. This is an easy task for our *S&H* object (see 2.1). The trigger will be the condition $N(x_t) = 0$, while the sampled function will be $N(x_{t-1}) + 1$, the previous N value plus 1. So if $N(x_t) = 0$, the $N(x_{t-1})$ value is sampled and holden until the next N zero, i.e. for an input signal period time. I've called this new function $M(x_t)$:

```
M(x) = SH(N(x) == 0, N(x)' + 1) ;
```

I remind you that the first `SH` argument is the trigger, the second the signal to be sampled. Inside the trigger argument I've used the `==` operator, that returns 1 if $N(x_t) = 0$, 0 else.

Pitch function. So now we have the input signal period's length in samples inside the $M(x_t)$ function. If we want its frequency instead, we have just to divide the *sampling rate* by the $M(x_t)$ value:

$$Pitch(x_t) = \frac{SR}{M(x_t)}$$

`SR` is a function that returns the hosting *sampling rate*, and is defined in the math.lib library. To avoid divisions by zero, we can insert a max function inside the denominator, for example $max(M(x_t), 1)$, so that even if $M(x_t) = 0$, the max function will select the value 1 (because in this case $1 > M(x_t)$) and the fraction will still be defined. Since this case should occur only at the beginning of the computation, due to the initialization values, then maybe we'd expect the $Pitch(x_t)$ function to be null, rather than to value $\frac{SR}{1}$. So we have to subtract SR if $M(x_t) = 0$. The resulting FAUST code is then the following:

```
Pitch(x) = SR / max(M(x), 1) - SR * (M(x) == 0) ;
```

Whole code. Ok, we have finished our *Sinusoid Pitch Tracker*, let's write now the whole code.

```
1  //-----------------------------------------------
2  //      Sin Pitch Tracker (1 period measurement)
3  //-----------------------------------------------
4
5  import("math.lib") ; //for SR definition
6  SH(trig,x) = (*(1 - trig) + x * trig) ~ _ ; //SH definition
7
8  U(x) = (x' < 0) & (x >= 0) ;
9  N(x) = (+(1) : *(1 - U(x))) ~ _ ;
10 M(x) = SH(N(x) == 0, N(x)' + 1) ;
11 Pitch(x) = SR / max(M(x), 1) - SR * (M(x) == 0) ;
12 process = Pitch ;
```

The `process` calls the `Pitch` function without any argument: so the requested `x` argument becomes an object inlet: the input signal.

Ok let me write now the same code with a new syntax:

```
1  //-----------------------------------------------
2  //       Sin Pitch Tracker (1 period measurement)
3  //-----------------------------------------------
4
5  import("math.lib") ; //for SR definition
6  SH(trig,x) = (*(1 - trig) + x * trig) ~ _; //S&H definition
7
8  Pitch(x) = SR / max(M, 1) - SR * (M == 0)
9  with {
10       U = (x' < 0) & (x >= 0) ;
11       N = (+(1) : *(1 - U)) ~ _;
12       M = SH(N == 0, N' + 1) ;
13       } ;
14 process = Pitch ;
```

Have you noticed something? I've used the `with{...}` syntax. I've defined the main function (the one that returns the pitch), and without putting `;` at its end, I've written `with{...}` and only then I've putted the `;`. Inside the `with{...}` I've defined all the other functions I needed to define the main one, without specifying the arguments (I've written `U` and not `U(x)` as it normally should had been): in fact they are passed from the main function. This syntax has two advantages in respect to the normal function definitions: one is that the main function's arguments are passed inside the `with` and you have not to write them each time, as just seen; the other is that the functions inside the `with` are not defined outside it, so you can use the same function names in other places without multiple definitions errors.

2.2.2 Sinusoid Pitch Tracker - several periods measurement

There is a problem in the *sinusoid pitch tracker* we have just made. In fact, for the frequency calculation we have used the $M(x_t)$ function, that counts the time in samples, so there could be some significant error because of the time quantization. A solution is to count the duration of a set of several periods and then divide this result for the number of periods considered; fortunately, in a normal pitched sound the pitch won't change inside this small set of periods. So let's call a the number of periods we consider – this value will be assigned to a slider.

V function. Take the $N(x_t)$ function, that we have used as a counter reseted to zero by the event $U(x_t) = 1$ (i.e. a zero in the input signal in a "rising" section). Now we should consider a set of a of these events, so we want the $N(x_t)$ function to be reseted to zero only after a $U(x_t) = 0$ events. Of course, now we need a counter for the $U(x_t) = 0$ events. I've called this counter $V(x_t)$, let's see its FAUST definition:

```
V(a,x) = +(U(x)) ~ %(int(a)) ;
```

It should remind you a counter like:

```
count = +(1) ~ _;
```

only that instead of `+(1)` I've placed `+(U(x))`, that is equal to 1 during a (important) zero, else it's equal to 0. So instead of having a counter that increases by 1 at each sample (last code line), we have a counter that increases by 1 at each $U(x_t) = 0$ event. The *modulo* function (`%`) recursively reset the counter to 0 as soon as it reaches the `int(a)` value; the function `int` (that returns the integer value of a number) seems inutile, in fact a is already an integer, being the number of periods we are considering; however, the compiler doesn't know this background and needs the `int(a)` specification. So, riassuming, the $V(a, x_t)$ function increases by one at each $U(x_t) = 0$ event and as it reaches the a value, it is reseted to 0.

W function. We should change now the definition of the $N(x_t)$ function, so that it doesn't stop rising until a $U(x_t) = 1$ *and* $V(a, x_t) = a$ event occurs, i.e. until a signal (important) zero is detected and exactly a of these zeros have been passed since the last $V(a, x_t)$ reset (and so also the last $N(x_t)$ one). It's simpler if we define a new function, $W(a, x_t)$, that represents this double requested condition.

$$W(a, x_t) = \begin{cases} 1 & \text{if } U(x_t) = 1 \text{ et } V(a, x_t) = a \\ 0 & \text{else} \end{cases}$$

In FAUST this should be something like this:

```
W(a,x) = (U(x) == 1) & (V(a,x) == a);
```

As in FAUST the "true" value is represented by 1, and the "false" one by 0, we can simplify the first condition, and the code becomes:

```
W(a,x) = U(x) & (V(a,x) == a);
```

Final changes. Now, we have to update the $N(x_t)$ function, replacing the $U(x_t)$ function with the just defined $W(a, x_t)$ one:

```
N(a,x) = (+(1) : *(1 - W(a,x))) ~ _;
```

Finally, the main function `Pitch(a,x)` has to divide $M(a, x_t)$ by a, and to be corrected in order to be null if $M(a, x_t) = 0$:

```
Pitch(a,x) = a * SR / max(M(a,x),1) - a * SR * (M == 0);
```

The whole code becomes then the following:

```
1  //-----------------------------------------------
2  //     Sin Pitch Tracker (a periods measurement)
3  //-----------------------------------------------
4
5  import("math.lib");
6  SH(trig,x) = (*(1 - trig) + x * trig) ~ _;
7  a = hslider("n cycles", 1, 1, 100, 1);
8
9  Pitch(a,x) = a * SR / max(M,1) - a * SR * (M == 0)
10 with {
11       U = (x' < 0) & (x >= 0);
12       V = +(U) ~ %(int(a));
13       W = U & (V == a);
14       N = (+(1) : *(1 - W)) ~_;
15       M = SH(N == 0, N' + 1);
16       };
17 process = Pitch(a);
```

The higher is a, the more precise is this pitch tracker; but remember that if you increase a, then the analysis flow delay also increases!

2.2.3 How to set "a" inside a code

Be careful that, at least on the FAUST version I'm actually using (0.9.9.4j-par), if you set a constant a value inside the code, the *Pitch Tracker* won't work, it will return always the value 0. A look to the generated C++ code will explain the cause of this error. I've took for example the last *Pitch Tracker* and modify its `process` in the following way:

```
process = Pitch(8), Pitch(a);
```

This `process` computes in parallel two `Pitch` function, with the a parameter fixed to 8 the first one, with the slider's value as the same parameter the second one. Well, only the second function will work properly. In fig. 2.7 you can see the last part of the C++ generated code. You can recognize the `for` cycle, inside which are computed the functions at the sampling rate. The two lines of code showed inside are the two output definitions: `output0[i]` is the first output, so it represents the `Pitch(8)` result (can you find the 8 presence in the code?); `output1[i]` is the second output and represents the `Pitch(a)` result instead. The last one, as just said, is the only one that will work. The reason stands in that `1.0f` in the second output, that is instead replaced with the `8` in the first one. In fact, for C++, `1.0f` is a floating point number, so all the computing results will be floating point (that is correct); the first line, on the contrary, deals in some way only with integer values, and the outputed result is then wrong. Then, it is sufficient to multiply

```
for (int i=0; i<count; i++) {
        [...]
        output0[i] = (fSamplingFreq * ((8 / max(iRec0[0], 1)) - (8 * (iRec0[0] == 0))));
        [...]
        output1[i] = (fSlow2 * ((1.0f / max(iRec3[0], 1)) - (iRec3[0] == 0)));
        [...]
}
```

Figure 2.7: The C++ generated code of the "wrong" process definition. Only the second output will work properly.

our fixed value (8 in this example) by a floating point coefficient of 1.0, or to use the `float` FAUST function, or simply type `8.0`, and also the first output will be working properly.

```
process = Pitch(8 * 1.0), Pitch(a) ;
process = Pitch(8 : float), Pitch(a) ;
process = Pitch(8.0), Pitch(a) ;
```

All of these lines will produce the same – correct – C++ code, that will be working for both the outputs (see fig. 2.8).
The present warning will be of much utility in the *Adaptive FM synthesis (delay-*

```
for (int i=0; i<count; i++) {
        [...]
        output0[i] = (fSamplingFreq * ((8.0f / max(iRec0[0], 1)) - (8.0f * (iRec0[0] == 0))));
        [...]
        output1[i] = (fSlow2 * ((1.0f / max(iRec3[0], 1)) - (iRec3[0] == 0)));
        [...]
}
```

Figure 2.8: The C++ generated code of the "correct" process definition. Both the outputs will work properly; you can notice in fact that in the first output instead of the 8 there is a 8.0f (floating point number).

line based PM technique), in 3.5, where the pitch tracker will controll a delay line.

2.2.4 Complex sounds Pitch Tracker

Now that we have a *sinusoid pitch tracker*, let's see what would happen if we apply this object on a complex sound instead. Take a look at fig. 2.9. If we apply to this wave our sinusoidal pitch tracker, it will count 3 "important" zeros during

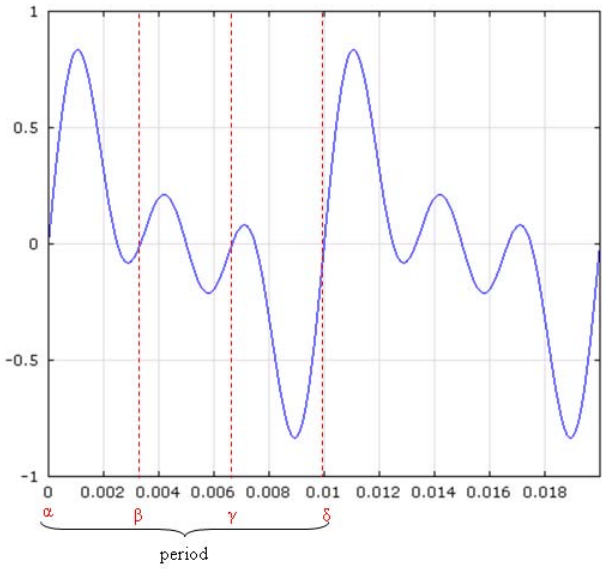

Figure 2.9: A 3-components harmonic wave at 100 Hz (two periods showed). Each period has 3 zeros during "rising" phases; these zeros have been labeled in the first period with Greek letters.

each period, so it will return a frequency of about 300 Hz instead of only 100. A solution could be an adaptive low-order lowpass filter, that filters the signal to be analyzed, and which cutoff frequency follows the detected pitch. For example, I'm going to use a *first-order Butterworth lowpass filter*. Let's apply this new algorithm to our complex waveform.

1st step. We have understood that the first signal pitch analysis says about 300 Hz. Thus the filter is applied with this as cutoff frequency. See fig. 2.10.

2nd step. This new version of the input signal is then sent to the sinusoid pitch tracker. Due to the filter's smoothing effect, a zero has been eliminated, and the new returned value, depending on the a amount, should be around 200 Hz (exactly 200 Hz if a is even). Then the lowpass filter is set with this new value as cutoff frequency (see fig. 2.11).

3rd step. Now the waveform has exactly one "important" zero per period, so the sinusoid pitch tracker will return the correct frequency. However, to reach a stable configuration we need a last step, in which the filter is set at the cutoff frequency of 100 Hz. This obviously won't affect the number of zeros anymore, and the pitch tracker will continue to return the correct frequency (see fig. 2.12).

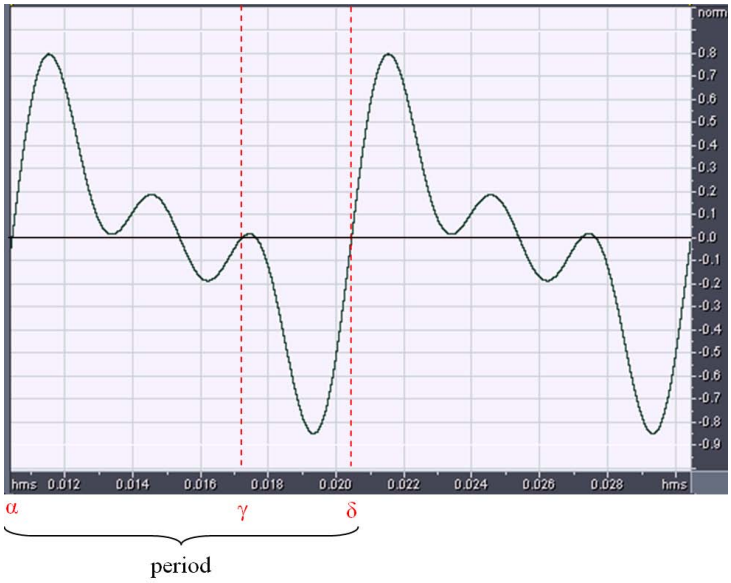

Figure 2.10: First step: the 3-components harmonic wave at 100 Hz is filtered by the lowpass filter at the cutoff frequency of 300 Hz. The "important" zeros per period are only 2 now, due to the filtering smoothing effect.

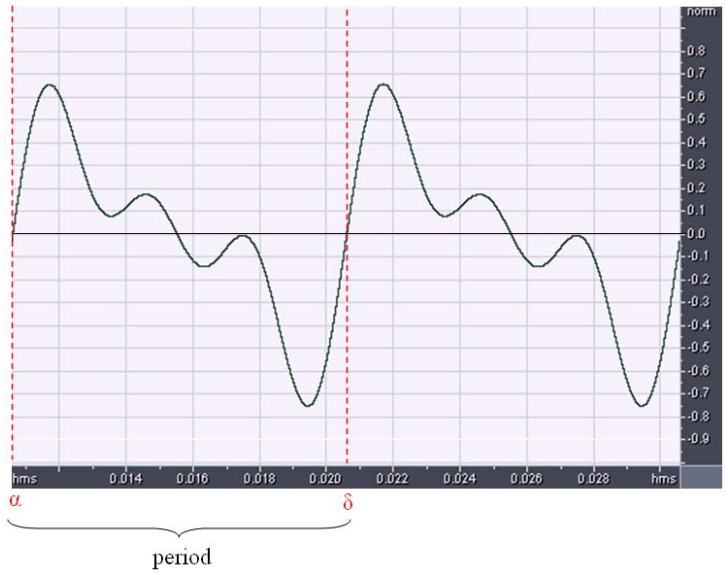

Figure 2.11: Second step: the 3-components harmonic wave at 100 Hz is filtered by the lowpass filter at the cutoff frequency of 200 Hz. There is only one "important" zero per period now.

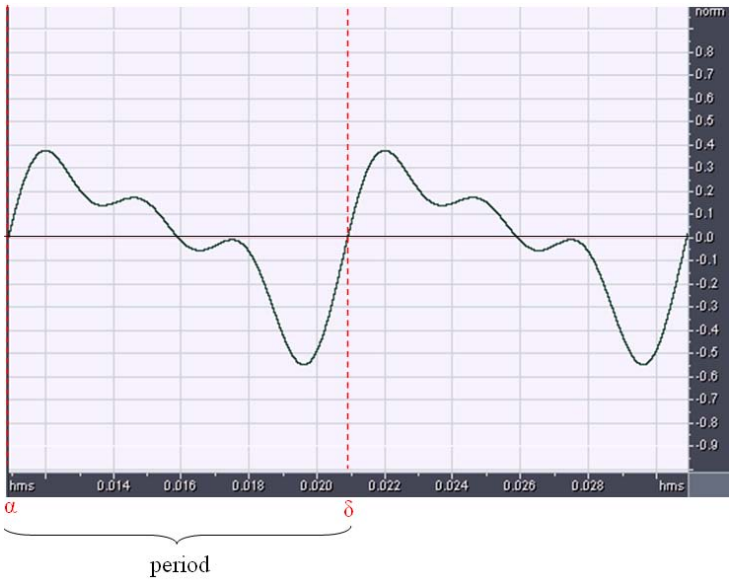

Figure 2.12: Third step: the 3-components harmonic wave at 100 Hz is filtered by the lowpass filter at the cutoff frequency of 100 Hz. The configuration is now stable (the next steps don't change anything as long as the signal remains the same).

Timing overview. Thus, already after two steps the pitch tracker begins to detect the correct pitch, and because its value is updated every a cycles, this detection takes in this case less than $0.019 \cdot a$ seconds. In fact, at the first step (no filtering) the detected period is 1/300 s, so it is returned after $a/300$ s; at the second step the detected period is about 1/200 s, so it is returned after $a/200$ s; since the third step the detected period is the correct one (1/100 s), and it is returned after $a/100$ s. If you sum all these timings, the total requested time will be the already said one. This is not a so quick process, in fact the a value could be at around 10 (see 2.2.5 for details about a value choice), so the requested detecting time would be in the considered case of about 0.2 seconds, that is the duration of a 16th note at 75 b.p.m., something very common in music. The true implemented code, as you will see, sets as initial filter's cutoff frequency the value of 100 Hz, before the *Sinusoid Pitch Tracker* starts its analysis. So in the real implementation, this considered case will be "solved" in only $0.01 \cdot a$ seconds, that's half of the previous timing.

Changing signal. Now, what happens if the signal pitch changes? Well, if it decreases, then also the detected pitch will decrease and in a few steps the filter will stably isolate, as already seen, the desired unique zero per period. If the pitch increases, you could think that the filter may "cut" all the waveform, being set with a cutoff frequency lower than the new signal fundamental. Instead, due to the

fact that the filter is not an ideal one, it lets the signal fundamental's zero-crossing rate information untouched. You can see an example of this behavior in fig. 2.13.

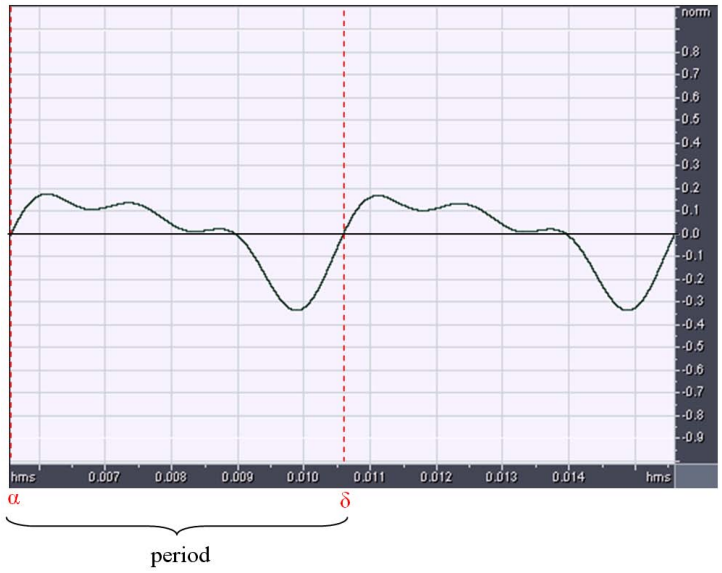

Figure 2.13: The input signal has increased its pitch: the 3-components harmonic wave has now 200 Hz and is filtered by the lowpass filter set at the "old" cutoff frequency of 100 Hz. There is no loss of information, and the sinusoid pitch tracker is able to detect the new fundamental frequency.

Faust code. The FAUST code for this new pitch tracker, compared with the *sinusoid* one, consists only in a modification of the `process` definition and in the add of a library.

```
1  //-----------------------------------------------
2  // Universal Pitch Tracker (a periods measurement)
3  //-----------------------------------------------
4
5  import("math.lib");
6  import("filter.lib");
7  SH(trig,x) = (*(1 - trig) + x * trig) ~ _;
8  a = hslider("n cycles", 1, 1, 100, 1);
9
10 Pitch(a,x) = a * SR / max(M,1) - a * SR * (M == 0)
11 with { [...]};
12 process = dcblockerat(80) : (lowpass1 : Pitch(a)) ~ max(100);
```

I've added the filter.lib library for the definitions of the filters I'm going to use. Then in the `process` definition I've put first `dcblockerat`, a high-pass filter set

with a cutoff frequency of 80 Hz. This is for removing any signal's DC offset and low-frequency noise that could affect the pitch detection. Then the adaptive filter, `lowpass1`, that implements a first-order Butterworth lowpass filter. It receives the signal from the `dcblockerat` and the cutoff frequency from the `Pitch` function, through the recursive operator `~`; on the "backing" path, the `Pitch` output passes through a `max` function that, selecting the maximum between the incoming value and 100, makes sure that the adaptive filter doesn't receive a cutoff frequency minor than 100 Hz. This is useful at the beginning of the running, because the filter's initialization value would be 0 Hz otherwise – it would be directly taken from the `pitch`'s output before the signal would reach it. You can see finally the `.svg` block-diagram in fig. 2.14.

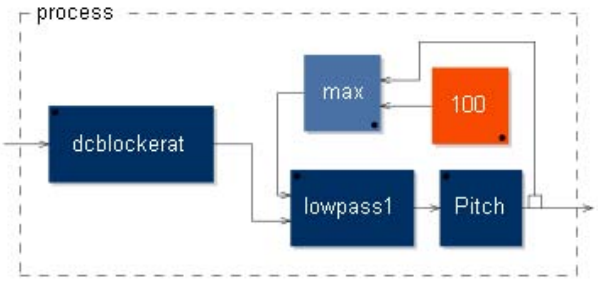

Figure 2.14: The "process" block-diagram of the "Universal Pitch Tracker".

2.2.5 Notes on the analysis cycles number

How to set, then, the number of cycles to be used in the analysis (the parameter we've called a)? It's easy to estimate the error between the returned frequency value and the true one. In fact, the true frequency f can be calculated in this way:

$$f = \frac{1}{T(f)}$$

where T is the period in seconds. A similar formula works for the returned frequency f_a, where a is the number of cycles used in the analysis:

$$f_a = \frac{1}{T_a(f)}$$

where $T_a(f)$ is the detected period in seconds. This period is calculated, as seen, with the formula:

$$T_a(f) = \frac{\lfloor M(a,f) \rfloor}{a \cdot SR}$$

where $\lfloor M(a,f) \rfloor$ is the function that returns the number of samples detected in a periods, and SR is the sampling rate. We know that $\lfloor M(a,f) \rfloor$ has to be an

integer, in fact I've used the *floor function* $\lfloor \ \rfloor$ inside its symbol now. We can think then the true signal period $T(f)$ as a function of the "true", non-integer, number of samples per a periods, that we can indicate with $M(a, f)$ instead. So we'll have, for the true period $T(f)$, the expression:

$$T([a,]f) = \frac{M(a,f)}{a \cdot SR} = \frac{a \cdot M(1,f)}{a \cdot SR} = \frac{M(1,f)}{SR} = T(f)$$

The true period $T(f)$ is in fact invariant in respect with a. Going on, it's obvious that:

$$M(a,f) = \frac{SR \cdot a}{f}$$

where $M(a, f)$ is the (true) number of samples per a periods. As usual, the relative error $E_a(f)$ should be defined in the following way:

$$E_a(f) \stackrel{def}{=} \frac{|f - f_a|}{f}$$

Starting from this formula, you can verify that with some substitutions for f_a, the following expression can be found:

$$E_a(f) = \left|1 - \frac{a \cdot SR/f}{\lfloor a \cdot SR/f \rfloor}\right| =$$

$$= \frac{frac(a \cdot SR/f)}{\lfloor a \cdot SR/f \rfloor} <$$

$$< \frac{1}{\lfloor a \cdot SR/f \rfloor} <$$

$$< \frac{1}{a \cdot SR/f - 1} =$$

$$= \frac{f}{a \cdot SR - f} \stackrel{def}{=} \bar{E}_a(f)$$

The last expression is an hyperbole and represents the limit superior of the $E_a(f)$ function. Because the frequencies we are going to use are minor than the Nyquist one (i.e. $\frac{SR}{2}$), we can approximate this function with the following one:

$$\bar{E}_a(f) \approx \frac{f}{a \cdot SR}$$

You can notice from this function that the relative maximum error is quite exactly proportional to $\frac{1}{a}$, to $\frac{1}{SR}$ and to f. In fig. 2.15 you can see a graphical

representation of $E_1(f)$ and $E_{10}(f)$ with the relative limit superior $\bar{E}_1(f)$ and $\bar{E}_{10}(f)$ functions. We can compare these results with the average JND (Just Noticeable Difference) in frequency (see [Roe73]), that represents *our ear's* relative error for pitch detection. It is usually greater than 0.5% (that's 0.005), so you can see by the graph that our *Pitch Tracker*, if set with $a = 1$, will be more precise than our perception only for frequencies minor than 200-300 Hz; on the other hand, if we set $a = 10$, it will be ok for frequencies up to 2000 Hz, that should be high enough for wathever foundamental tone.

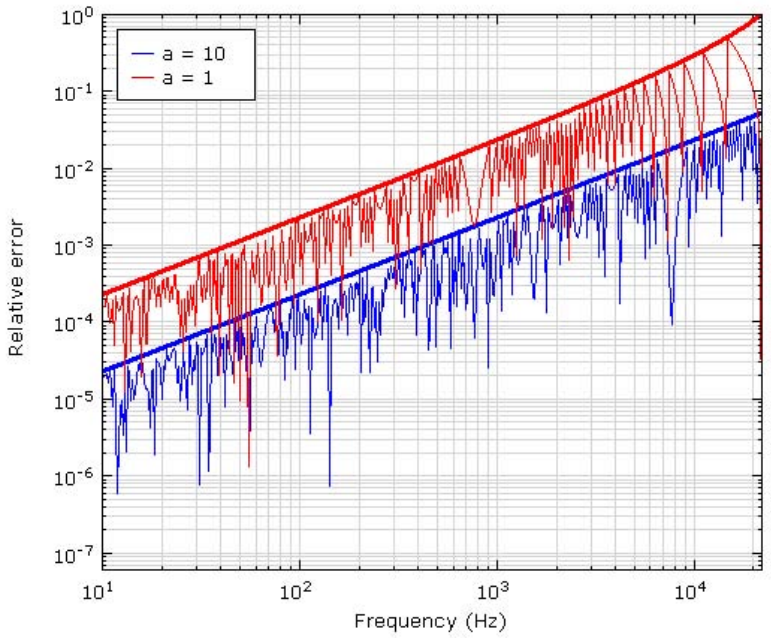

Figure 2.15: The relative errors of the "pitch trackers" in function of the signal's frequency, with a set at 1 (red) and at 10 (blue), at a sampling rate of 44100 Hz. It is clear the quite-linear behavior of the limit superior functions.

To be more exact, we can ask the condition:

$$\bar{E}_a(f) \leq JND = 0.005$$

You can check that this is equivalent to:

$$a \geq \left\lceil \frac{0.995 \cdot f}{0.005 \cdot SR} \right\rceil \stackrel{def}{=} \underline{a}(f, SR)$$

So this shows the "recommended" $\underline{a}$ values in function of the signal frequency f and the sampling rate SR. The $\lceil\ \rceil$ symbols represent the *ceiling* function, that approximates a real value into the smallest integer not less than it. You can notice

that the higher is the signal frequency f, the higher has to be a, and the higher is the sampling rate SR, the lower can be a. It is better not to set a to a too high value because this proportionally delays the analysis data flow and decreases the time pitch tracking precision. This is why I've called the threshold value $\underline{a}$ "recommended". In fig. 2.16 you can see the representation of the function $\underline{a}$(f,SR) for some common SR values.

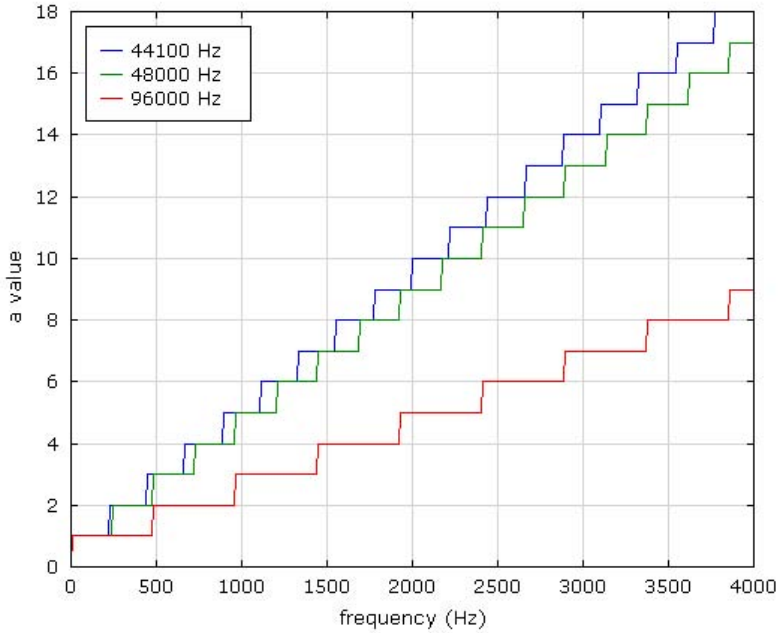

Figure 2.16: The $\underline{a}(f, SR)$ function representation, for three typical sampling rates (showed in blue, green and red), in function of the frequency f.

2.2.6 Max-MSP patch example

Typically pitch trackers are used to drive some sound control. In fig. 2.17 it's shown a MAX-MSP patch example in which the *Universal Pitch Tracker* is used to drive the center frequency of some notch filters. In this way the sound's first harmonics can be removed in real time, achieving an interesting noisy effect for example on the flute sound.

2.3 Computational overview

In fig. 2.18 you can see the output bandwidth of the objects presented in this chapter. I remind you that the smaller is the output bandwidth, the more expensive is the object. You can notice that the output bandwidths are sensibly

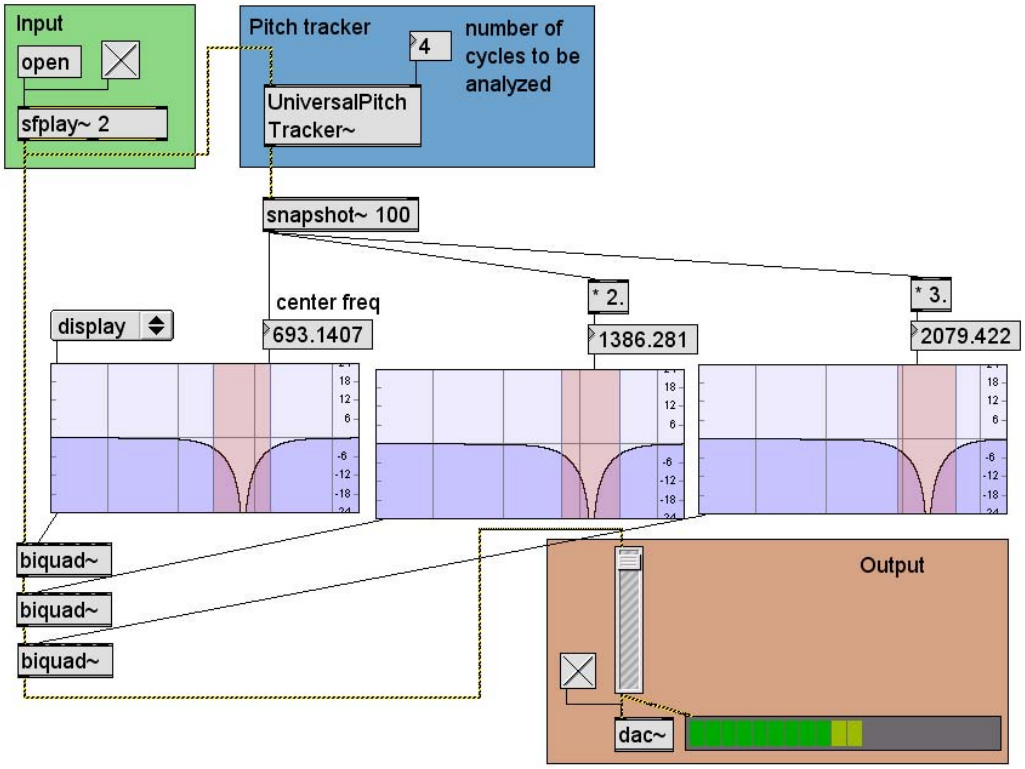

Figure 2.17: The Max-MSP pitch tracker example patch.

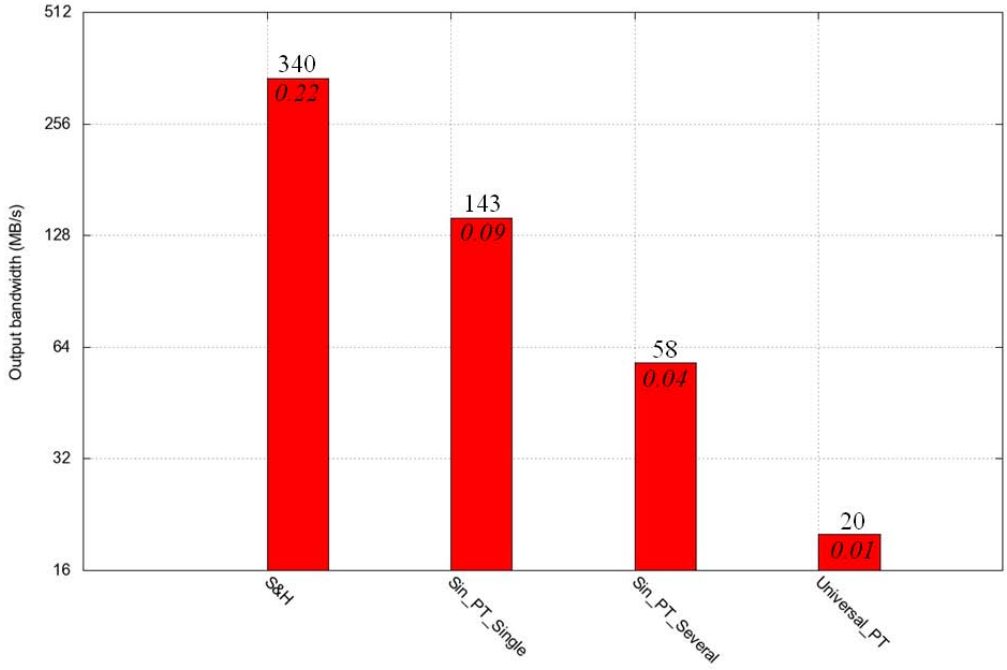

Figure 2.18: The chapter's objects output bandwidth graph: in order from left to right, are shown the *S&H*, the *Sinusoid Pitch Tracker (one period measurement)*, the *Sinusoid Pitch Tracker (several period measurement)* and the *Universal Pitch Tracker*. As already seen in 1.4 chapter, this graph shows for each object the exact bandwidth value (outside the bars) and the ratio with a test object (inside the bars); the test object in this case is a simply copy from a input to a output, and represents the maximum computational bandwidth for this kind of object (1576 MB/s). The data are been collected on a Pentium 4, 3.66 GHz laptop.

smaller than the one we've seen in the *Panners* chapter. This is because in the panners the output is double, while here the output is single: thus, intuitively, a panner will produce more output than a delay line. However, also the ratios with the respective test objects are in general lower in this case, so we can conclude that these object are also more expensive.

2.4 Conclusion

This chapter's object can be used for several purposes. You have already seen one in 2.2.6 with the MAX-MSP example patch, and I'll show you only some of the other possible uses, like for the adaptive-FM synthesis techniques (see 3.5 and **??**), or for a granulator (see **??**).

Chapter 3

Delay lines

Introduction

Delay lines can have many uses, as they deal with a central dimension of the sound: time. The most obvious use could be for a echo unit, in which the delayed signal is mixed with the original one. An other common use of delay lines is for phase-varying effects, like in phasers, flangers or choruses. We are not going to see any of these cases, principally because other people have already implemented them in FAUST (see [Gra06] for guitar-effects delay lines using). Instead we'll see some "countering", "spacial" and adaptive-synthesis uses. The first section, *Delay line types*, is an overview on the possible delay lines defined in FAUST; in the first two subsections are shown respectively the two basic ready-made elements available for delay lines: `@` and `rwtable`; in the third subsection are shown the external delay-line functions, as they are defined in the .lib libraries, and are discussed some other basic functions like the use of the `&` logical operator or the `<<` binary shift. This first section ends with a computational overview of the delay line functions (3.1.7). In the successive sections then are shown some example objects using delay-lines: *RMS* detector (3.2), *Interaural Time Difference* panner (3.3), *Wave Field Synthesis* algorithm (3.4), and *Adaptive FM Synthesis* using the delay-line based PM technique (3.5); finally there will be a computational overview of these examples (3.6) and a brief conclusion.

3.1 Delay line types

Almost all delay lines are not ready-made objects in FAUST, you have to define them first. Several delay line functions are defined in the music.lib and filters.lib libraries, and we are going to see these definitions first. So when we will use one of these delay lines we'll have simply to import the respective library in our .dsp file.

There are two possible basic objects in FAUST that can be used to realize a

delay line: the `@` and the `rwtable`. Their syntax and their use are explained in the "first" *Faust Tutorial* [GO03] and also in the *Quick Reference* [Orl07], but let me briefly remind them here.

3.1.1 The @

The `@` is a binary operator that delays a signal by a precise number of samples, so if you write `y = x@10`, `y` will be a 10-samples-delayed copy of `x`. Unluckily, the delay amount's range has to be known by the compiler during the compilation, so you can use this object only if that delay amount is constant or varies in a known finite range; for example, it could be a *slider*, because slider's values have the range you set in slider's definition. But it can not be a counter, for example! A solution to this problem could be explicitly limiting the counter's range by using a `min` and a `max` functions, as shown in the following code:

```
1 //-----------------------------------------------
2 //          The @ example
3 //-----------------------------------------------
4
5 a       = +(1)~_;
6 y(a,x)  = x @ (a : min(100) : max(1));
7 process = y(a);
```

This .dsp works, because the `a` counter is limited into the range $[1, 100]$. Here's some explanations of the code. The first line defines a counter called `a`, that will increment its value by 1 every sample. For an explanation of its definition, please see [GO03], section 6.2. The `min` and `max` are binary functions, so they have two inputs. You can in fact write `min(100,a)` and you'll get the minimum between these two values. If you write only one argument (`min(100)`), you'll have a function with one input – the other argument. This is called *partial function*, and writing `a : min(100)`, in other terms "sending" `a` into that partial function, is the same as writing `min(100,a)`. Thus, writing `a : min(100) : max(1)`, like in the exampling code, is like sending the result of `min(100,a)` into the second partial function, and is equivalent to `max(min(100,a))`. In a similar way, writing `process = y(a)` when `y` is defined with two argument, will create an input to the object, that will be of course the second `y`'s value, i.e. `x`.

The whole process is described by the block-diagram in Fig. 3.1.

Anyway, if you forget to limit the delay range with `@` operator, FAUST will give you the following error message:

```
ERROR : can't compute the min and max values of : int(proj0(W0)@0)
        used in delay expression : IN0@int(proj0(W0)@0)
        (probably a recursive signal)
```

In fact, the problem is that the delay amount is not limited.

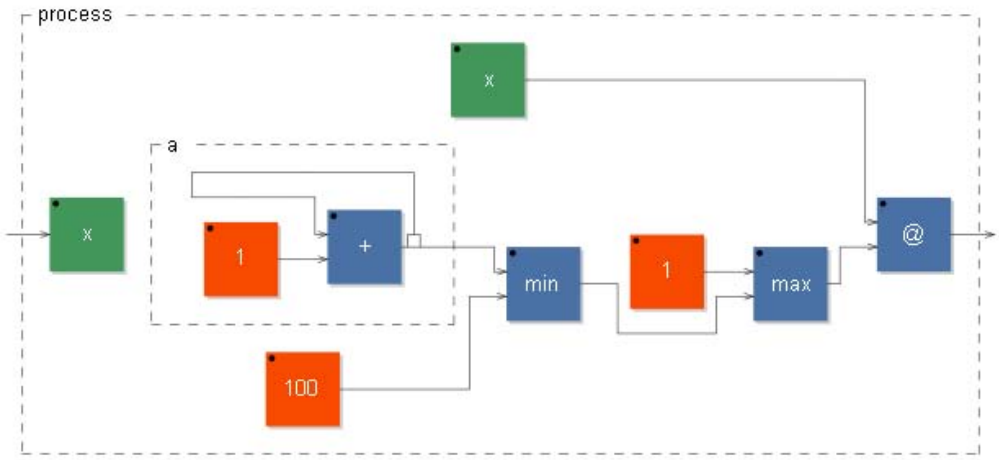

Figure 3.1: The "@ example" - block-diagram

3.1.2 The rwtable(n,i,w,x,r)

The other possible object that can be used for a delay line is a read-write-table, called in FAUST `rwtable`. It's a collection ("table") of values that you can read by an index, and write by an other index. The idea is that we continually write inside the table the input signal using an appropriate incrementing write-index, and continually read values from it using a incrementing read-index that is a kind of delayed version of the first one.

The `rwtable` has five arguments, respectively:

- **n**: the size of the table, that has to be an integer value, constant and known by the compiler at the time of the compilation. It can't be, for example, a function of the *sampling rate*. Sometimes in fact you'll have the big temptation of using the `SR` function inside this argument, but this won't work! On the other hand, you can write a so big number you'll be sure not to overflow it!

- **i**: the initializing signal, i.e. the values to be assigned to the table's cells in the initial instant. For a delay line you should need simply silence, in this case you have to write 0.0 for this argument; this is not simply a number but a signal, an unlimited sequence of 0s, so it's enough to fill the whole table. Remember to specify ".0": only in this way the cells will be ready for floating point numbers, while if you write simply "0" they will keep only the integer parts – and in place of a delayed signal you'll hear only silence in most cases!...

- **w**: the write-index, that should not exceed the size of the table (but in that case there won't be any error message!).

- **x**: the signal to be written: at every sample the actual value of this signal is written into the cell pointed by the write-index. Usually this is an input audio signal, and the write-index increments by one at each sample. So in that case successive values of the audio signal are written in successive cells.
- **r**: the read-index, that should not exceed the size of the table, like the write-index.

So, for example, one could write the following .dsp code:

```
1 //-----------------------------------------------
2 //          The rwtable(n,i,w,x,r) example
3 //-----------------------------------------------
4
5 a       = +(1)~_;
6 y(a,x)  = rwtable(100, 0.0, a % 100, x, (a-10) % 100);
7 process = y(a);
```

It realizes a constant delay line of 10 samples.

In the first line there is the same counter as the previous example; in the second line a `rwtable` is called. It has 100 cells, is initialized with silence (0.0), the write-index is the `a` counter *modulo* 100 expression, it writes a signal `x`, and the read-index is the $(a - 10)$ *modulo* 100 expression. The *modulo* function is represented in FAUST by a `%`, like in many other programming languages, and it is used here to make che counter restart from 0 when it has reached $99 + 1$ (it will never reach 100 in this way, and this is ok because cell's numbers go from 0 to 99). Finally, in the last line, the `process` is the partial function `y(a)`, so the ungiven y's second argument becomes an input of our object: it corresponds to the `x` that is the writing signal inside the `rwtable`. Thus this object takes an input signal, writes it cyclically in a table and reads this table with an other cyclical index that takes 10 samples to reach the current write index; so the input signal is outputed with a delay of 10 samples. Ok, we made a delay line with `rwtable`!

You can see the .svg block-diagram of this example in fig. 3.2.

3.1.3 delay(n,d,x)

Let's see now the external delay line functions, starting from the music.lib library. You'll see that all of them are based on the `rwtable` object: this is because in this way the delay amount has not to have known limits at the compilation time.

```
1 //-----------------------------------------------
2 //          The delay(n,d,x) function
3 //-----------------------------------------------
4
5 index(n)      = &(n-1) ~ +(1);
```

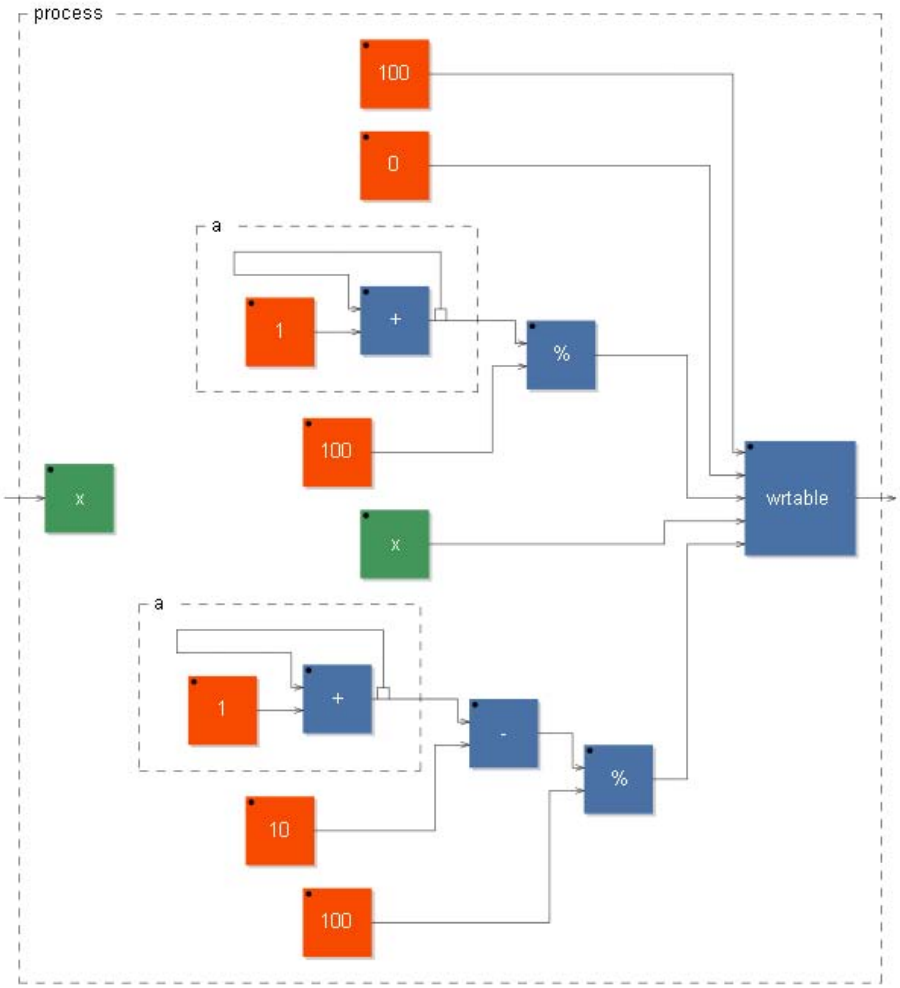

Figure 3.2: The “rwtable(n,i,w,x,r) example” - block-diagram

```
6 delay(n,d,x) = rwtable(n,0.0,index(n),x,(index(n)-int(d))
7                 & (n-1)) ;
```

It's very similar to the previous example. The differences are the followings:

- the size of the table, `n`, and the delay amount, `d`, are let as `delay` function's arguments; so remember that they have to be positive!
- in the read-index expression there is a `int` function, that returns the integer part of a number. Of course, we didn't need it as long as we used constant integer values: in the previous example the actual `d` was in fact the fixed delay amount of 10 samples. If we want to generalize this number then we have to use the `int` function, otherwise FAUST will give us an error message.
- There are not the *modulo* functions. At their place, there is the `&` (AND) binary operator; in fact these two expressions are equivalent if n is a power of 2 and a is a positive integer:

$$a \ \% \ n = a \ \& \ (n-1) \quad \text{if } n = 2^k$$

  If a is a negative number, then, the result of % in FAUST will be also negative, while the result of & will remain positive. This makes the & operator more adapt inside an index expression, that has to be of course positive. An other quality of it is that it's very cheaper than % in terms of computation. Ok, you'll have to remember to use only powers of 2 as n, as long as you don't want extremely unprevedible - and noisely - delays!
- This "new" *modulo* function "jumped" from the `rwtable` write-index argument (where it stayed in the previous example into the `a` expression) into the index definition. This makes also `index` a function of `n`, and does not change the result, neither in the `rwtable`'s read-index, where the *modulo* is now calculated two times. In fact we can "translate" the read-index expression in the following way (assuming d to be integer):

$$\begin{aligned} \text{read-index} &= (\text{index}(n) - d) \ \& \ (n-1) = \\ &= ((a \ \& \ (n-1)) - d) \ \& \ (n-1) = \\ &= ((a \ \% \ n) - d) \ \% \ n = \\ &= (a - d) \ \% \ n \end{aligned}$$

  where a is the `a` counter of the previous example. The last expression is then the read-index of the previous example, once you generalize the fixed values we used, so you can see that the two read-index expressions are equivalent. The last passage can be simply demonstrated with the definition of *modulo*.

There is no `process` because this is only a function definition. To make it a true object, one should specify some kind of `process`. For example, the following code calls this function, uses a slider as delay amount (in samples) and takes as input the signal to be delayed using the partial function `delay(n,d)`.

```
1 //-----------------------------------------------
2 //          The delay(n,d,x) example
3 //-----------------------------------------------
4
5 import("music.lib") ;
6 d       = hslider("delay", 100, 0, 10000, 1) ;
7 process = delay(1<<14,d) ;
```

Instead of a "normal" number as the first argument of `delay`, I've putted the expression `1<<14`. The `<<` symbol stays for the *left binary shift*, so I've said to take a binary 1 and to shift it to the left by 14 positions, leaving null the other bits. This binary number is then 2^{14}: remember in fact that this argument has to be a power of 2. Writing in this way is better for a human understanding: in fact you immediately recognize that that number is a power of 2, rather than writing directly the result (16384).

You can finally see the .svg block-diagram of this example in fig. 3.3; you can notice that the `1<<14` expression is replaced automatically by its decimal representation.

3.1.4 fdelay(n,d,x)

The `fdelay` ("fractional delay") object stays also in the music.lib library, and it's a delay line similar to the previous one, except that it can take as delay amount a non-integer value. It then utilizes a linear interpolation between two `delay` functions, evaluated respectively in two successive integer values of `d`. Let's see the code:

```
1 //-----------------------------------------------
2 //          The fdelay(n,d,x) function
3 //-----------------------------------------------
4
5 frac(n)       = n-int(n) ;
6 index(n)      = [...] ;
7 delay(n,d,x)  = [...] ;
8 fdelay(n,d,x) = delay(n,int(d),x)*(1 - frac(d))
9                 + delay(n,int(d)+1,x)*frac(d) ;
```

I've written here only the "new" definitions. There is a `frac` function that simply returns the fractional value of a number (by subtracting its integer part

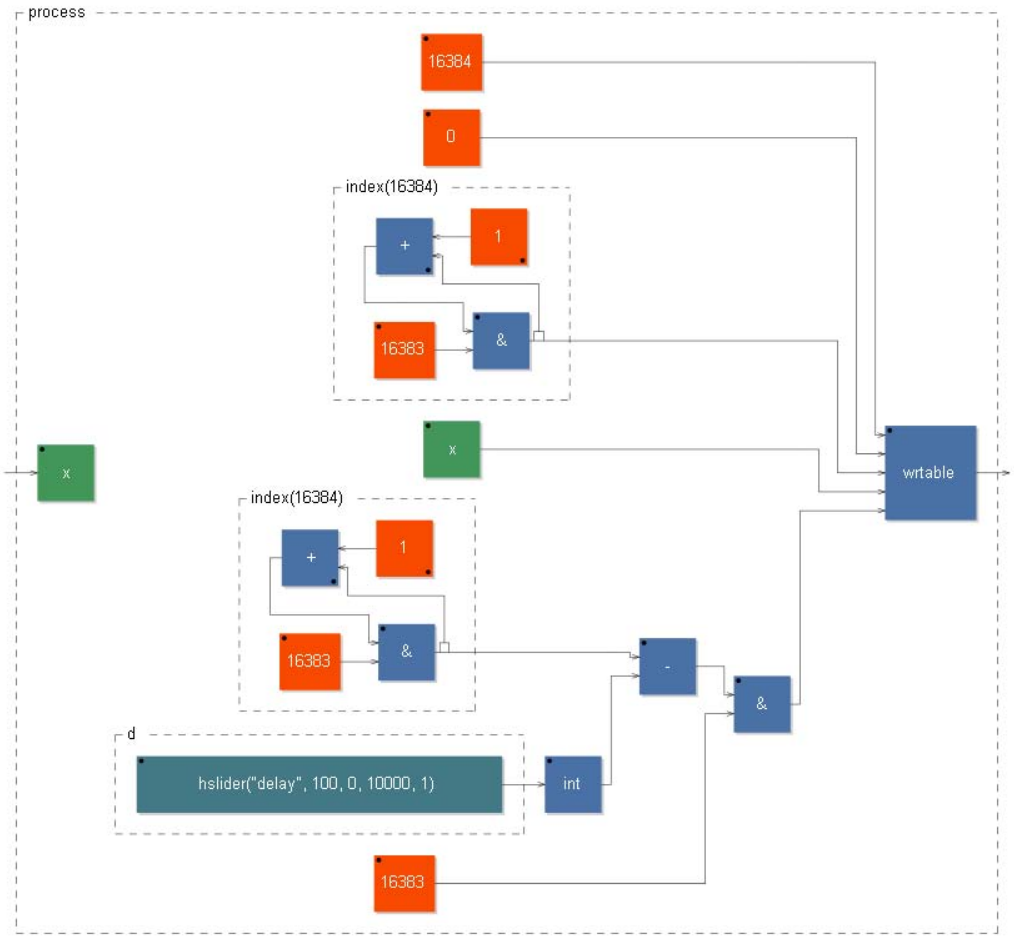

Figure 3.3: The “delay(n,d,x) example” - block-diagram

from it). The `fdelay` then is a convex combination (like a pondered mean i.e. a linear interpolation) of two `delay` functions with two successive `d` values. So, if `d` is an integer, then `frac(d)` is null and `fdelay` returns the value of the only `delay(n,d,x)` addend. If `d` is not an integer, `fdelay` returns an intermediate value between `delay(n,int(d),x)` and `delay(n,int(d)+1,x)`, so that the more close `d` is to one of that two points, the more important is the respective `delay` addend, in a *linear* way. This linear interpolation is not the "best" interpolation, and corresponds to a Lagrange interpolation of the first degree. The higher degrees are implemented in the filter.lib library by Julius O. Smith (see 3.1.6). I'm not going to show an example for this delay, because the previous one can be used, by replacing the `delay` with a `fdelay` and of course by letting the slider have also fractional values.

3.1.5 Fixed lenght delay lines

The `n` length of the `rwtable` used by these delay lines, that we said has to be a power of 2, has been fixed for some common delay ranges. For example, after the definition of `fdelay` function in the music.lib library, there are definitions like these:

```
delay1s(d) = delay(65536,d);
[...]
fdelay1s(d) = fdelay(65536,d);
[...]
```

These are delay lines with one second as maximum delay. In fact the first argument of the `delay` or `fdelay` function, that is the length of the `rwtable` used, is fixed here to 65536 (that is 2^{16}, the same as writing `1<<16`). With a common sampling rate of 44100 Hz, you'll be sure that with such a table length your read-index won't "surpass" the write one after the *modulo* operation, as long as your delay amount is less than 1 second (and even more), that in this case corresponds to 44100 samples. If your sampling rate is greater than the `rwtable`'s length -1, for example 96000 Hz, then with a delay amount set to 1 second, your read-index will have the value:

$$r = (w - d) \ \% \ n = (w - 96000) \ \% \ 65536 = (w - 30464) \ \% \ 65536$$

that at this sampling rate corresponds to a delay amount of only 0.32 seconds. So there will be an *aliasing* effect due to the *modulo* function. To prevent this, if you are going to use high sampling rates, you should choose a bigger `n` amount inside the delay lines (and you shouldn't use the ready set delay functions here described). The other preset delay lines have the following max delay amounts: 2, 5, 10, 21, 43 seconds. So you could call for example the function `fdelay43s(d)` for having a fractional delay line with up to 43 seconds as delay amount `d`.

3.1.6 Lagrange and Thiran allpass interpolative delay lines

Finally, in the filter.lib library by Julius O. Smith, you can find some other fractional delay lines, based on *Lagrange* and *Thiran allpass* interpolation. I'm not going to explain here their definitions, but you can find in the library's references the theoretic explications (see [Smi07]). I'm going to show here only the way these functions have to be used. There is a certain number of applications in which you have a fractional delay amount and you need a very accurate interpolation, so the linear `fdelay` interpolation is not enough "smooth" (its first derivative is in fact discontinue). An example for this need is shown later in this chapter in a *Phase Modulation* application (see 3.5). Thus you can use these interpolative delay lines that can simulate better the "unknown" values of a signal between two successive samples.

The Lagrange interpolative delay line functions have the following names: `fdelay1`, `fdelay2`, `fdelay3`, `fdelay4` and `fdelay5` and the arguments are the same as the `fdelay` function (n,d,x). The number inside their name represents the *Lagrange interpolation order*, and usually the higher it is, the more accurate and well sounding is the interpolation. There are some differences between even and odd orders important in some situations, so you should check in these cases what is the requested order. You can read in the library that each of these functions has a minimum delay value allowed, for example in `fdelay4 d` should be at least 1.5 (in samples). Remember then that it's always needed a power of 2 as `n` argument - because all of these functions call the old `delay`.

Also the *Thiran allpass* interpolative delay line functions have several orders and their name are: `fdelay1a`, `fdelay2a`, `fdelay3a` and `fdelay4a`. They have exactly the same syntax of the Lagrange interpolative delay line functions, and so the same as `fdelay`. Check in the library the minimum delay value allowed, like in the Lagrange case. As already said, see later in this chapter for an example of Lagrange interpolative delay lines.

3.1.7 Computational overview

To conclude this overview on the delay lines available in FAUST, let's take a look to their computational cost. In fig. 3.4 you can see the output bandwidth of these objects. It's easy to see from the graph that the higher is the interpolation order, the more expensive is the delay line, and that in general the Lagrange interpolation is less expensive than the Thiran one.

3.2 RMS

The RMS is an object that returns the *Root Mean Square* (or *quadratic mean*) of a signal. So it has to sum the squares of the successive values of the signal, divide that sum for the number of values and calculate the square root of that

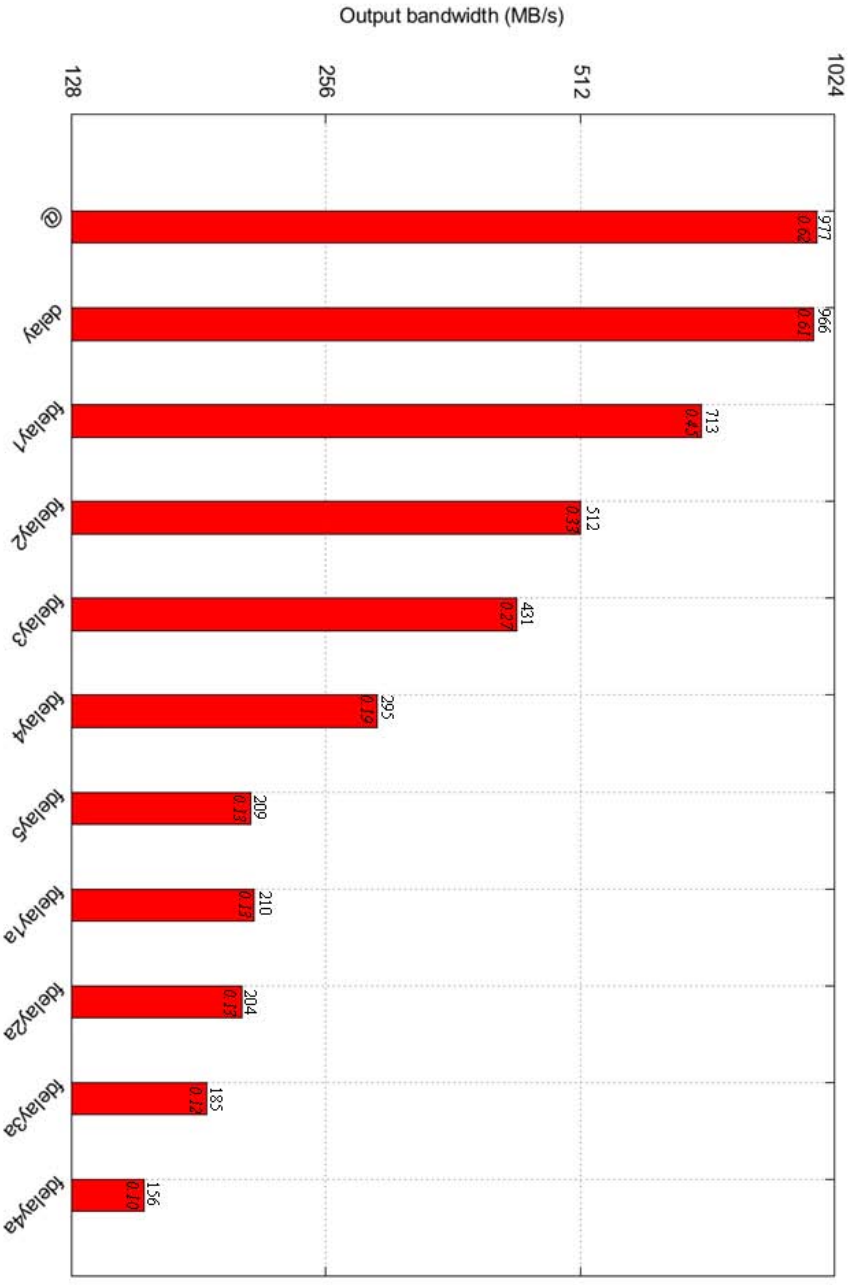

Figure 3.4: The delay line types - output bandwidth graph. The test object is a simply copy from an input to an output, as seen in 2.3.

value:

$$RMS(x_k) = \sqrt{\frac{\sum_{i=k-n}^{k} x_i^2}{n}}$$

3.2.1 RMS with fixed n

A code that realizes the RMS algorithm with a fixed n value is the following:

```
1 //-----------------------------------------------
2 //      The RMS example - fixed n
3 //-----------------------------------------------
4
5 S(n,x)  = +(x - x @ n) ~ _;
6 Quad(x) = x * x;
7 RMS(n)  = Quad : S(n) : /(n) : sqrt;
8 process = RMS(1000);
```

I've used the `@` operator in `S(n,x)` to subtract from the sum the value that the `x` signal had `n` samples before. In fact the `S(n,x)` function takes its precedent value (see the recursive composition operator `~`), adds to it the current `x` value and subtracts the "old" `x@n` value. For the first n samples the `x@n` expression will be simply null, so the function will be adding to itself the `x` value sample after sample. Then it will subtract the `x@n` value, so since that instant it will always keep the sum of the last n `x`'s samples. The `Quad(x)` function is a simply squaring function: I could write `pow(x,2)` instead of `x*x`, but the `*` function is very much cheaper than the `pow` one. Then, the `RMS(n)` function is a sequenced composition of the functions `Quad`, `S(n)`, `/(n)` and `sqrt`. Note that all of them have one input and one output: `Quad` has no argument so it takes an inlet as it, `S(n)` has not the second argument so it takes an input too and so on. So, being a sequenced composition of functions with one input and one output, also `RMS(n)` has one input (the `x` of the `Quad` function) and one output (the `sqrt`'s one). In `process` then I've called that function with `n`= 1000, so this code will calculate the RMS of 1000 samples, updating its value every sample. You can see the `.svg` block diagram in fig. 3.5.

3.2.2 RMS with changing n

If we want the user to choose the `n` value, we have to do several changes to our code. We can't simply assign a slider's value to `n`, because if it decrements while the sum `S(n,x)` is running, then some addends will remain "forever" inside – the `x@n` won't reach them anymore. We have to insert some conditional statements, so that *if* the `n` value changes, *then* the `S(n,x)` sum has to restart from zero. To do so, we have to multiply somewhere by the logical condition `n'==n`: the operator `'`

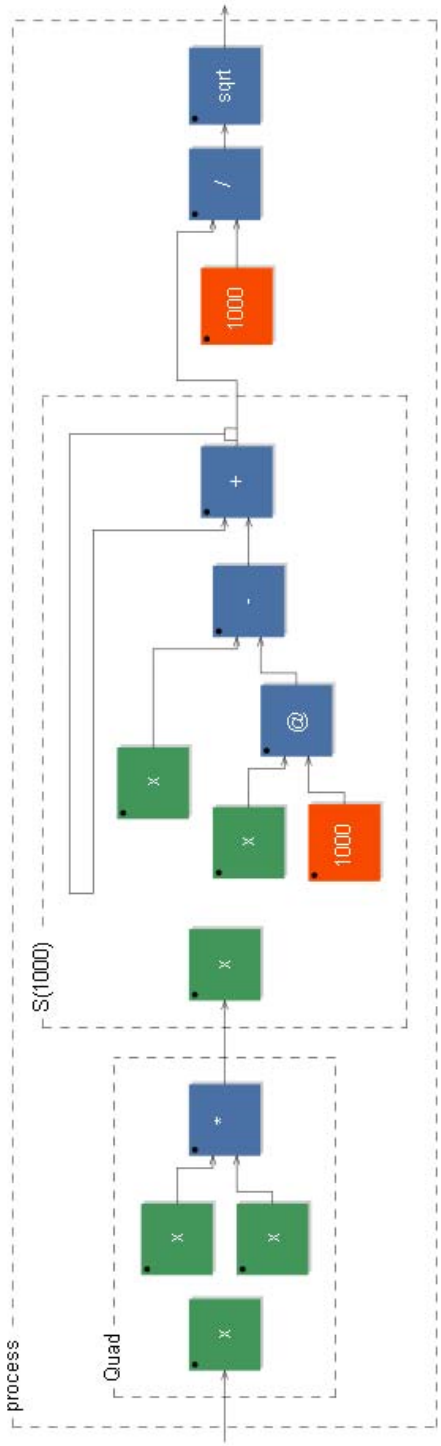

Figure 3.5: The “RMS (fixed n) example” - block-diagram

is a delay line of one sample (the same as `mem` or `@(1)`), so that expression will be true (value 1) if `n` is not changed during the last sample, 0 else. This is the kind of information we need, so let's see what the code has became:

```
1 //-----------------------------------------------
2 //     The RMS example - changing n
3 //-----------------------------------------------
4
5 n = hslider("number of samples", 100, 20, 44100, 10);
6 Count(n) = min(+(1), n+1) ~ *(n'==n);
7 S(n,x) = +(x - (x@n : *(Count(n)>n))) ~ *(n'==n);
8 Quad(x) = x * x;
9 process = Quad : S(n) : /(n) : sqrt;
```

Let's focus on the `S(n,x)` function first: I've inserted the `*(n'==n)` expression into the recursive pattern; thus, when `n` changes, this expression multiplies by 0 the `S(n,x)` value, else it leaves it untouched (multiplication by 1). But this is not enough, because if `n` changes, it's not sufficient for `S(n,x)` to restart from zero: we have also to forbid the subtraction by the `x@n` expression, until exactly `n` samples have passed. To do so, we need an other conditional statement that multiplies the `x@n` expression by the condition that a counter (`Count(n)`) has overflew the `n` value: `*Count(n)>n`.

These were the changes to the `S(n,x)` function; let's see now the counter definition. It recursively sums 1 to itself as far as `n` remains constant (see the `*(n'==n)` also here), if `n` changes it restarts from 0. But when it reaches the $n+1$ value, it remains constant (if `n` doesn't change) because a `min` function selects the minimum between the `+(1)` incrementing and the `n+1` value. This because a counter should not grow unlimitedly, else it would crash the host application in a certain time. In our case, the value of $n+1$ is sufficient to trigger the first conditional statement inside the `S(n,x)` definition, and there is no reason for `Count(n)` to grow further.

The rest of the code is identical to the previous version, except that there is a slider assignment. You can see some `.svg` block-diagrams in fig. 3.6 and fig. 3.7.

3.3 ITD panner

Let's see now a "panning use" of a delay line. In fact, when a sound source is not at the center of the auditive scene, there is a slight delay between the two ears's received signals, due to the different distances between each ear and the sound source. This phenomenon is called *Interaural Time Difference* (ITD) and has an important role in defining the spacial position of a sound source. A good panner should introduce also this kind of information in addiction to the classic *Interaural Level Difference* (ILD), which was the central goal of the *Panners* chapter's object.

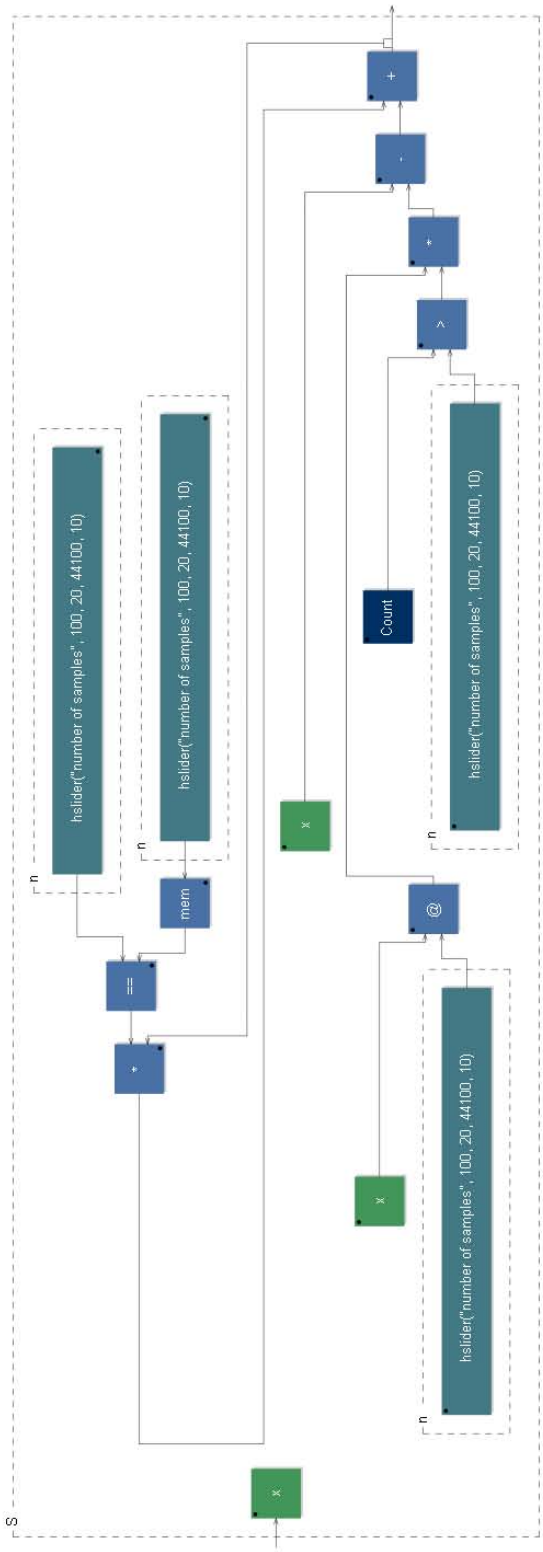

Figure 3.6: The “RMS (changing n) example” - block-diagram (“S” block)

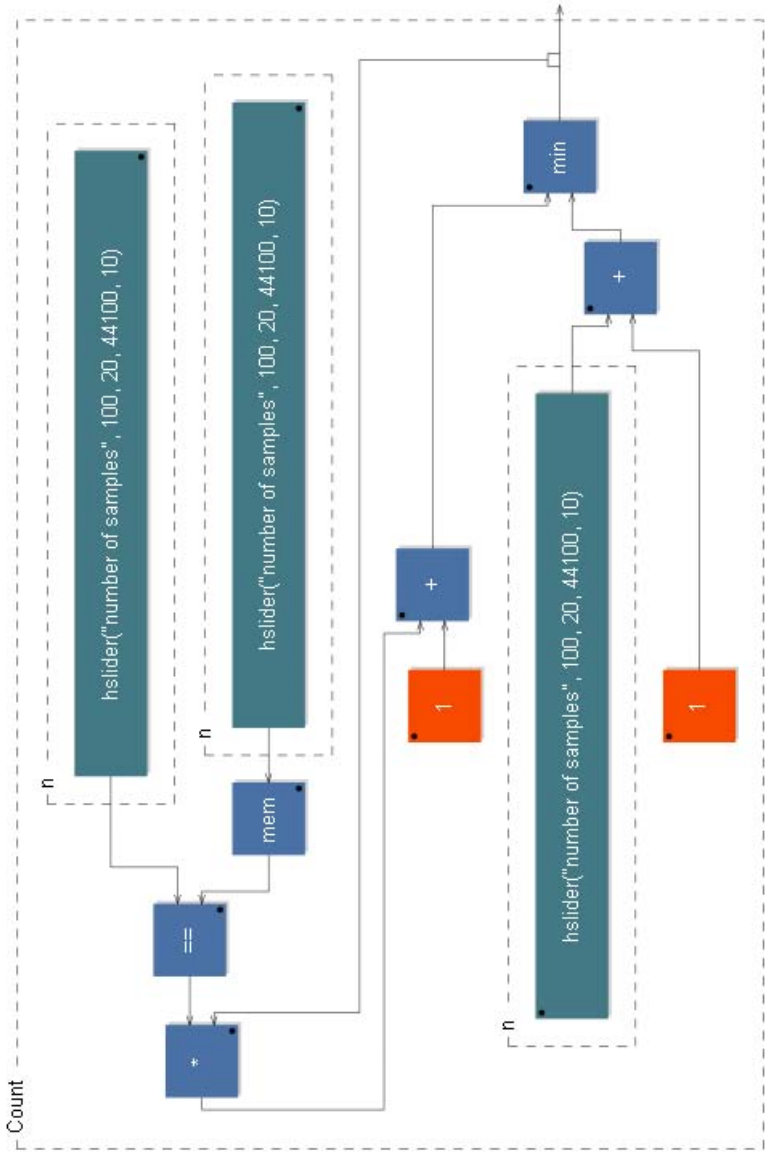

Figure 3.7: The "RMS (changing n) example" - block-diagram ("Count" block)

Let's see a scheme representing the typical geometrical configuration of a sound source and a listener's ears (fig. 3.8).

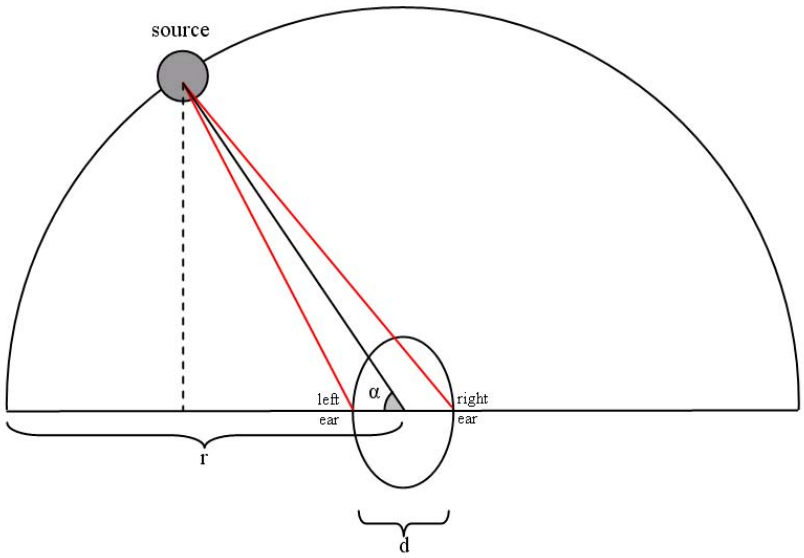

Figure 3.8: The geometrical configuration of a sound source and a listener's ears. The angle α is set like the usual panner control: minimum if the source is completely to the left and maximum in the opposite case, but moving in a range of $[0, \pi]$ instead of $[0, 1]$. The source distance from the center of the listener's head is constant (r) and the ears distance is d. The two read lines represent the two distances between the source and each of the two ears, and are a function of r, d and α.

It's easy to calculate the distances between the source and each of the two ears, as a function of the angle α, the source distance r and the ears distance d:

$$d_L(r, d, \alpha) = \sqrt{(r \cdot sin(\alpha))^2 + \left(r \cdot cos(\alpha) - \frac{d}{2}\right)^2}$$

$$d_R(r, d, \alpha) = \sqrt{(r \cdot sin(\alpha))^2 + \left(r \cdot cos(\alpha) + \frac{d}{2}\right)^2}$$

Then, we are interested in the difference between these two distances, and after some simplifications this is:

$$\begin{aligned} delta(r, d, \alpha) &= d_L(r, d, \alpha) - d_R(r, d, \alpha) = \\ &= \sqrt{r^2 - d \cdot r \cdot cos(\alpha) + \frac{d^2}{4}} - \sqrt{r^2 + d \cdot r \cdot cos(\alpha) + \frac{d^2}{4}} \end{aligned}$$

This difference is then converted into time delay by dividing it by 343 m/s, that is the sound speed in air. This time delay has to be given then to a delay line as a number in samples, thus we have to multiply it by the sample rate. The sample rate, as already seen, is returned by the function `SR`, defined in the library math.lib .

So, the whole code is the following:

```
1  //-----------------------------------------------
2  //      ITD panner
3  //-----------------------------------------------
4
5  import ("math.lib") ;
6  import ("filter.lib") ;
7  d = hslider("Ears distance (cm)",17,15,20,0.1) / 100 ;
8  r = hslider("Object distance (cm)",100,15,5000,1) / 100 ;
9  alpha = hslider("Angle (degrees)",90,0,180,0.1) * PI / 180 ;
10
11 quad(x) = x * x ;
12 delta(r,d,alpha)=sqrt(quad(r) - d*r*cos(alpha) + quad(d) / 4)
13 - sqrt(quad(r) + d * r * cos(alpha) + quad(d) / 4) ;
14
15 del(r,d,alpha) = delta(r,d,alpha) / 343 * SR ;
16 process = _ <: fdelay3(64, max(del(r,d,alpha), 0) + 1), fdelay3(64,
17 max(-del(r,d,alpha), 0) + 1) ;
```

Let's take a look to the `process` definition: an input signal is split into two copies with different delay amounts. Notice that we use here the `fdelay3` function (3rd-order Lagrange interpolation), that has to have a delay of at least 1 sample (see the comments in its definitions in filter.lib); so a sample is added inside the delay argument of `fdelay3`. The `max` function inside the same argument is used to assign positive delays to the left channel, leaving the other one untouched (that's the case $d_L(r,d,\alpha) \geq d_R(r,d,\alpha)$ i.e. $90° < \alpha \leq 180°$) and symmetrically negative delays to the right channel with inverted sign (i.e. $0° \leq \alpha \leq 90°$). This because in the first case the signal has to arrive to the right channel first, so we have to delay the signal assigned to the left channel, and in the second case just the opposite. The maximum delay length then is set as to be greater than the intuitive maximum value of the `del(r,d,alpha)` function, i.e. when the sound source is completely at one side and the difference $delta(r,d,\alpha)$ of the distances is equal obviously to the ear's distance d. Its maximum slider's value is then 20 cm (0.2 m) that are covered in 0.58 ms by the sound; this time is equivalent to about 56 samples at a sampling rate of 96 kHz, so we can choose 64 as maximum delay length because it's the successive power of 2 available.

Notice finally the `quad(x)` function that realizes the square power of a number. Its definition is here not so relevant – we could write a multiplication every time

we needed it instead of calling this function. But it will really help us when we'll have more complex bases (like already in the precedent RMS example).

In fig. 3.9 you can see the `process` representation, but the internal block diagrams are too complex to be shown without occupying a dozen of pages! And for the understanding of the FAUST code the block diagrams are not so necessary in this case.

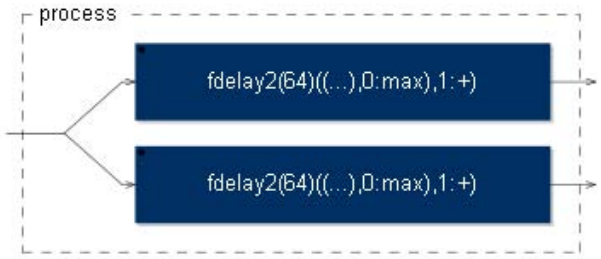

Figure 3.9: The ITD panner - block-diagram (process only)

3.4 WFS

Let's see an other "spacial" use of the delay lines. The *Wave Field Synthesis* (WFS) technique uses a line of speakers to simulate the wave field of a virtual source located behind this line. The idea is to use the *Huygens principle* and to think each speaker like a window in a wall: the sound coming through these windows will have the amplitude and the phase that a listener would hear in that points. At a certain distance from this wall, then, all these windows can approximate an infinite number of point-shaped sound sources; according to the Huygens principle, this configuration generates a wave field equivalent to the one generated by an unique source placed behind the wall. So a listener will hear a virtual sound source placed in some region of the space and even if he moves this region of space won't change (it won't follow the listener's movements) like a real sound source located in the space. The more speakers are used, and the less reverberating is the room, the more appreciable will be this effect. All we have to do is to calculate for each speaker what has to be the scaling amplitude factor and the correct delay amount, and possibly to generalize this results in therms of the speaker being considered. The scheme in fig.3.10 illustrates the geometrical configuration of this problem.

First of all, we have to choose a coordinate system. I've used the one illustrated in fig. 3.10: the speakers are aligned along the x axis, the origin is the center of the first speaker from left, the y axis is oriented in the opposite direction with respect to the listener and the unit is 1 m. So sound source's virtual positions will have positive ordinates, and listener's positions negative ones.

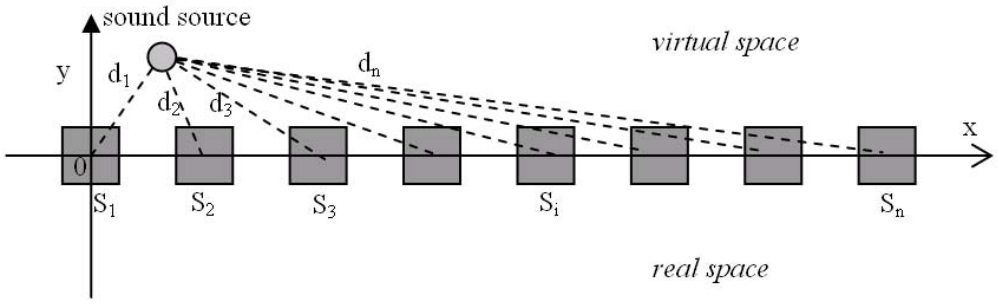

Figure 3.10: The WFS geometrical problem

We can now express each speaker's position as a function of its index, going from 1 to n (where of course n is the number of the speakers we are using) and of the speaker-to-speaker constant distance d in meters:

$$S_i(d): \quad (d \cdot (i-1), 0) \quad i = 1..n$$

The length of the speakers's line is of course equivalent to the S_n abscissa, so it's $d \cdot (n-1)$. Then, let's say (x, y) are the virtual source's coordinates. For each speaker we can now write its distance from the sound source position:

$$D_i(d, x, y) = \sqrt{[x - d \cdot (i-1)]^2 + y^2}$$

Now, we know that the sound pressure decays as $\frac{1}{r}$ where r is the distance from the sound source (its intensity decays in fact as $\frac{1}{r^2}$, and as said in 1.2, $I \propto A^2$, where A is the average amplitude). So we have to use for each speaker S_i a level scaling factor L_i proportional to $\frac{1}{D_i}$, for example:

$$L_i(d, x, y) = \frac{1}{D_i(d, x, y)}$$

In this case a D_i value minor than 1 will amplify the signal, so to avoid clipping we have to ask y – the virtual source's ordinate – not to be minor than 1. Instead the delay amount Del_i has to be proportional to D_i, as seen in the ITD section (see 3.3):

$$Del_i(d, x, y) = \frac{D_i(d, x, y)}{343 \text{ m/s}}$$

Now let's see the FAUST code that realizes the WFS with a generic number n of speakers.

```
1 //-----------------------------------------------
2 //      WFS
```

```
3  //-----------------------------------------------
4
5  import("math.lib") ;
6  import("music.lib") ;
7  import("filter.lib") ;
8
9  d = hslider("Speaker-to-speaker dist.(m)",0.5,0.1,10,0.1) ;
10 x = hslider("x (m)",0,0,10,0.1) : smooth(tau2pole(0.001)) ;
11 y = hslider("y (m)",1,1,100,0.1) : smooth(tau2pole(0.001)) ;
12 nSpeakers = 8 ;
13
14 Quad(x) = x * x ;
15 D(d,i,x,y) = Quad(x - (i - 1) * d) + Quad(y) : sqrt ;
16
17 // Amplitudes assignments:
18 Amp(d,i,x,y,sig) = sig / D(d,i,x,y) ;
19 OutA(d,1,x,y,sig) = Amp(d,1,x,y,sig) ;
20 OutA(d,i,x,y,sig) = OutA(d,i-1,x,y,sig), Amp(d,i,x,y,sig) ;
21
22 // Delay amounts assignments:
23 R(d,i,x,y) = fdelay1s(D(d,i,x,y) * SR / 343) ;
24 OutR(d,1,x,y) = R(d,1,x,y) ;
25 OutR(d,i,x,y) = OutR(d,i-1,x,y), R(d,i,x,y) ;
26
27 // sequence composition:
28 Out(d,n,x,y,sig) = OutA(d,n,x,y,sig) : OutR(d,n,x,y) ;
29 process = Out(d,nSpeakers,x,y) ;
```

So, first some libraries are imported: math.lib for the `SR` definition, music.lib for the `fdelay1s` definition and filter.lib for the `smooth` and `tau2pole` definitions. Then the user interface is generated, with an interpolation on the x and y sound source's coordinates. In the `nSpeakers` line you can specify the number of speakers you're going to use, but this choice has to be made before the computation, so you can't assign a slider for this quantity: this because of the *pattern-matching* (see later in this section). The distances D_i are then computed, using the already seen function `Quad` (used in *RMS* - see 3.2.1 - and in *ITD panner* - see 3.3). Then there are the *amplitudes assignments*. The `Amp` function is a copy of the incoming signal (`sig`) scaled with the L_i coefficient.

Pattern-matching For `OutA` is then used a *pattern-matching* function definition. Let me explain this concept with a substitution: if we call $foo(i)$ the partial function $OutA(d, \bullet, x, y, sig)$, then the pattern matching says that:

1. $foo(1) =$ (something);

2. $foo(i) = somefunction(foo(i-1))$.

You can see two definitions, the first one is used when the argument is 1, the second one is used for a generic i value as argument (so when $i \neq 1$). So $foo(i)$ is defined in a recursive way, and is computed by FAUST during the compilation. Returning to the `OutA` function, its definition is represented in fig. 3.11.

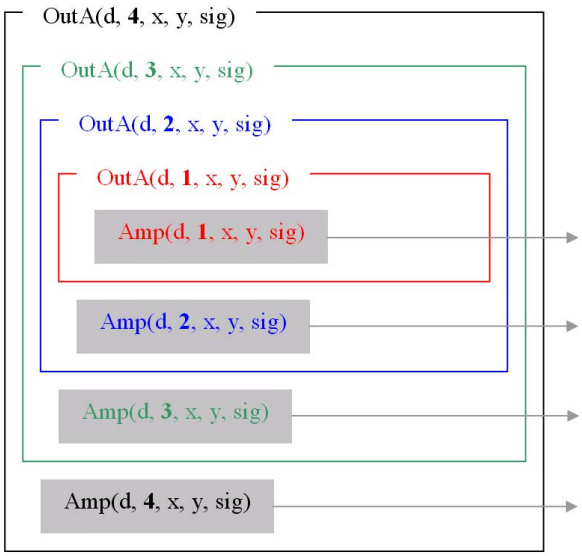

Figure 3.11: The pattern-matching representation of the function $OutA(d, i{=}4, x, y, sig)$. Each color represents a function call. At each step the index i (in bold style) decrements because of the *OutA*'s "generic-i-definition", until it reaches the value 1. At that point it's used the "i-equal-to-1-definition" of the same function instead, and the recursive calls do terminate. The resulting pattern in this case is a parallel pattern of 4 *Amp* functions with increasing i index; for a generic $OutA(d, \mathbf{n}, x, y, sig)$ the resulting pattern will have n *Amp* functions.

Thus, the pattern-matching generalizes a parallel composition of n `Amp` functions with an increasing index as an argument; thus `OutA` generates a collection of n `sig` signal versions scaled accordingly to each speaker's distance from the virtual sound source.

In the successive *delay amounts assignments* there is a similar pattern-matching defined function, `OutR`, which in the same way as `OutA` generalizes a parallel composition of n `R` functions. These functions are simply delay lines, like already seen in the *ITD Panner* (see 3.3), but this time using a simpler `fdelay1s` that realizes only a 1st-order Lagrange interpolation. Finally, the function `Out` realizes a sequence composition of `OutA` and `OutR`, with the second argument, n, passing the number of the pattern-matching steps to both of the functions. For example, calling the `Out` function with 8 as second argument will generate 8 parallel `Amp`

functions in sequence with 8 `R` functions. Thus, this is the processing chain for each speaker: an amplitude attenuation and a delay line. In fig. 3.12 a "block-diagram" equation shows the passages from the `Out` function call to the speaker's processing chains.

In fact, the `process` .`svg` block-diagram is shown in fig. 3.13 and shows the 8 processing chains in parallel – in fact 8 is the set value for the `nSpeakers` constant at the beginning of the code, of course you can change it. It's not shown here, but obviously the parameters from chain to chain are different, depending to the chain's index.

You can finally see some mathematical representations of the sound field generated by a virtual sound source positioned in $(1, 1)$, with 8 speakers at an equal intra-distance of 0.53 meters. In fig. 3.14 the coordinate system is the one already described for this object in fig. 3.10, with lines of equal intensity showed only in the *real space* (negative ordinate values), and the speakers aligned on the x axis. You can notice that if you are far enough from the speakers (let's say at least 2 meters in this case) the equal intensity lines have approximately the same shape of the circular ones produced by a single source placed in the virtual source position. In fig. 3.15 then, only the coordinate system is changed: while the x axis remains the same, the y one shows the distance from the virtual source position instead that from the speakers line. In this way, changing the x value and leaving the y constant, would correspond in the other coordinate system to moving on a circular trajectory around the virtual source, at a constant distance y. Moving in this way should not determine a change in the perceived sound intensity, so the equal intensity lines in this new coordinate system should be exactly horizontal. You can notice instead that if you are too close to the virtual source position, especially under 4 meters, the speaker placed at the opposite side with respect to the virtual source will be too loud. In the new coordinate system in fact, the virtual source stays in $(1, 0)$, and the S_8 speaker – the last one on the right in this example – is placed in $(8 \cdot 0.53,\ \sqrt{(8 \cdot 0.53 - 1)^2 + 1}) \approx (4.24, 3.39)$. Thus, as you can see, if you are closer than 4 meters from the virtual source, to hear the same intensity, you have to go farer in front of the S_8 speaker than in front of the other ones. This means that that speaker, the one opposite to the virtual sound source, is too loud at that distance; at a greater distance, in fact, this effect is very smaller and in a central x range we can consider the equal intensity lines horizontal – as we expect them to be in a perfect simulation. However, in this representation is not taken into account any interference effect.

3.5 Adaptive FM synthesis (delay-line based PM technique)

This object is the FAUST translation of one of the two Adaptive FM synthesis algorithms described in [LTL08]. The one we are interested in, and you can easily

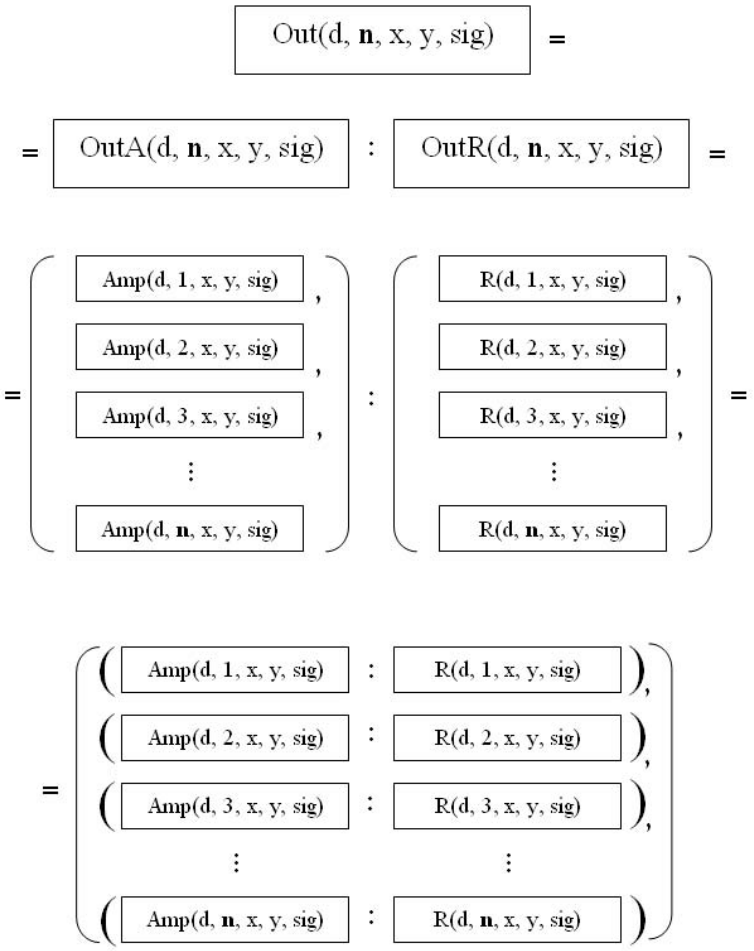

Figure 3.12: A "block-diagram" equation showing the passages from the "Out" function call to the speaker's processing chains. The "Out" second argument, labelled "n", controls the number of the final processing chains, so it should be equal to the number of speakers used. The first passage uses only the "Out" function definition, the second one expands the pattern-matching used in "OutA" and "OutR" functions definitions, and the last passage is a particular property of sequence and parallel operators that occurs when the number of outputs of the first system of parallel objects is the same than the number of inputs of the second system, and can be demonstrated using the respective graphs equivalence.

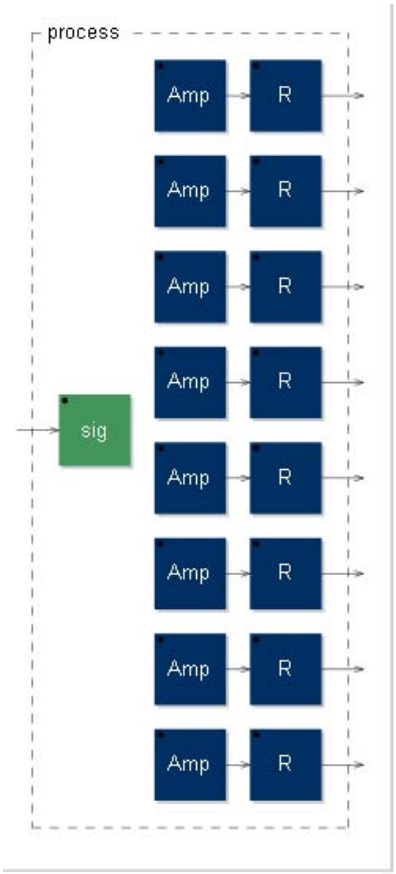

Figure 3.13: WFS “process” block-diagram.

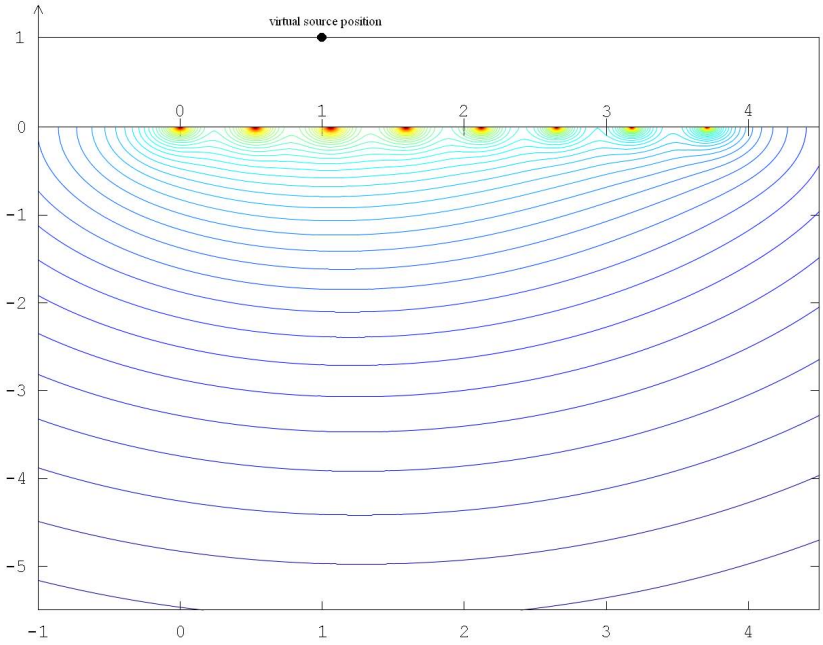

Figure 3.14: WFS mathematically obtained sound field for a virtual source placed in $(1, 1)$. Coordinate system as usual.

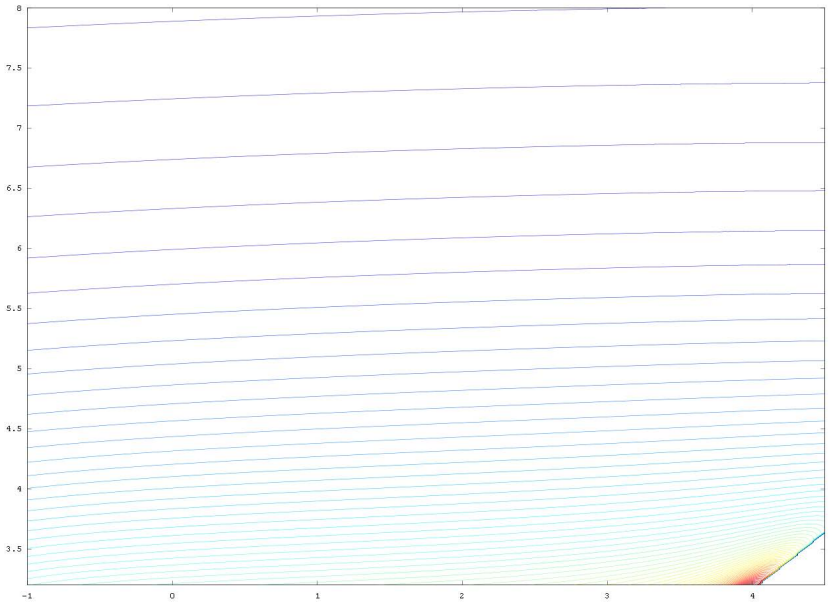

Figure 3.15: WFS mathematically obtained sound field for a virtual source placed in $(1, 0)$. Coordinate system with x as usual and y showing the distance from the virtual sound source.

understand why, uses the so called *Delay-Line Based Phase Modulation* technique. The FAUST code for the other technique, called *Phase Modulation Through Heterodyning*, will be presented in....

The *Adaptive FM synthesis* can be used to change the timbre of an arbitrary input signal, to make it synthesizer-sounding and to easily generate transitions between the original (natural) sound and the processed one. Both the techniques require a pitch detection on the input signal; the one we are going to use is the already seen FAUST object *Universal Pitch Tracker*, explained in section 2.2, that returns the signal's pitch analyzing its zero-crossing rate. I remind you that the main function was `Pitch(a,x)`, where `a` represents the number of cycles to be used in the analysis, and `x` the input signal. The algorithm consists in a $d(t)$ adaptive delay line on the input signal:

$$d(t) = \frac{I}{\pi f_c}\left[0.5\cos(2\pi f_m t) + 0.5\right]$$

where I is the *index of modulation*, f_c is the career signal frequency (the input signal's one) and f_m is the modulator's frequency. The f_m can be characterized through the expression:

$$f_m = \frac{f_c}{r_{c:m}}$$

where $r_{c:m}$ is the ratio between f_c and f_m, and will be specified by the user. The cos function inside the $d(t)$ expression can be replaced with a sin because the modulator's phase is not important in this application, and between the two function there is simply a phase difference of $\frac{\pi}{2}$. Thus, as the authors suggest, we are going to use a Lagrange interpolating delay line of order 3, and an interpolating oscillator, defined in music.lib and called `osci`. This object is simply a convex combination of two consecutive values of a sinusoid cycle stored in a table. The FAUST code for the whole object is then the following:

```
1  //-----------------------------------------------
2  //      aFM/PM (delay line based)
3  //-----------------------------------------------
4
5  import("math.lib");
6  import("music.lib");
7  import("filter.lib");
8
9  // S&H:
10 SH(trig,x) = (*(1 - trig) + x * trig) ~ _;
11
12 // pitch tracker:
13 Pitch(a,x) = a * SR / max(M, 1) - a * SR * (M == 0)
14 with {[...]};
15 PtchPr(a) = dcblockerat(80) : (lowpass1 : Pitch(a)) ~ max(100);
```

```
16
17 I = hslider("Index of modulation",0,0,5,0.001) ;
18 r = hslider("c:m",1,0.1,5,0.001) ;
19
20 // modulator:
21 del(r,I,x) = x : fdelay3(1 << 17, dt + 1)
22 with {
23       k = 8.0 ; // pitch-tracking analyzing cycles number
24       fc = PtchPr(k,x) ;
25       dt = (0.5 * osci(fc / r) + 0.5) * I / (PI * fc) *SR ;
26       } ;
27 process = del(r,I) ;
```

At the beginning of the code, some libraries are imported: math.lib for the `SR` and `PI` (π) functions, music.lib for `osci` and filter.lib for the filters used in the pitch detection and for the `fdelay3` definition. Then you can find the definitions of `SH` (see 2.1) and `Pitch` (see 2.2). Instead of the original `process` of the last one, I've defined the new function `PtchPr` – of course, `process` is reserved for the modulator in this object. Then the user interface is defined, with the two sliders for the index of modulation (variable `I`) and the c:m ratio (variable `r`). The modulator is then defined, with the function `del(r,I,x)`, in which `x` is the input signal. This function consists in a delay line with Lagrange interpolation of order 3 (`fdelay3`, see 3.1.4), applied to the signal `x`, with a maximum delay of 2^{17} samples (`1<<17`, see 3.1.3) and a delay amount of `dt + 1` samples, where `dt` is defined in the `with{ }` and the 1-sample adding is because this order of Lagrange interpolation needs a minimum delay of 1 sample (see the comments in `fdelay3` definition inside filter.lib). In the `with{ }` you can find the constant `k`, that will be used to set the analyzing cycles number in the following `PtchPr` function call; remember that you have to give this constant as a floating point number (argument discussed in 2.2.3). Then is defined `fc` (f_c), that will be the detected pitch of the input signal `x` with `k` as analyzing cycles number, as just say, instead of the original slider value a we have used in the pitch tracker stand-alone object (2.2.2). Then `dt` ($d(t)$) is defined with the expression shown at the beginning of this section, and its value (for that expression in seconds) is converted in samples through the final multiplication `*SR`: don't forget in fact that the delay lines we are using want always a delay amount expressed in samples, and not in seconds. You can notice that we are using `osci(fc/r)`, that returns a sinusoid of frequency $\frac{f_c}{r} = f_m$ instead of the cosine wave of the same frequency asked in the original $d(t)$ expression. But as we have already said, in this object the modulator's phase is not important and a sinusoid works as well as a cosine wave. Finally, the `process` calls the `del(r,I,x)` function without the last argument (the signal `x`), so it will become an object inlet. That's all!

I've tried this object with a flute sound as input signal, and it changes its sound preserving the original pitch and amplitude. With an integer "c:m" ratio value you

will hear an harmonic sound, with a non-integer value you will hear inharmonic or semi-harmonic sounds; in both cases, the difference with the original sound will be proportional to the I value, and the original sound will be heard if you set $I = 0$.

3.6 Computational overview

In fig. 3.16 you can see the output bandwidth of the last objects we have seen in this chapter. In this case the number of output channels is not the same for all the objects, so different test objects are been used to calculate the ratios. As we have already noticed, objects with two output channels have in general a greater output bandwidth than objects with only one output (see 2.3). But when the number of output channels is greater, then the things change and the time requested to write the outputs can limit the output bandwidth. In fact, the test used for the WFS object (with 8 outputs) had an output bandwidth minor than the 1-output test object's one (707 vs. 1576 MB/s). Finally, part of the "expensiveness" of the PM_aFM object is derived from the pitch tracker's one (see 2.3), obviously because it is used inside the PM_aFM code.)

3.7 Conclusion

We have seen in this chapter only a minimal part of the possibile uses of delay lines. We will see an other one later, inside the *granulator* object, but first we need some "randoming" functions. In fact this will be the content of the next chapter...

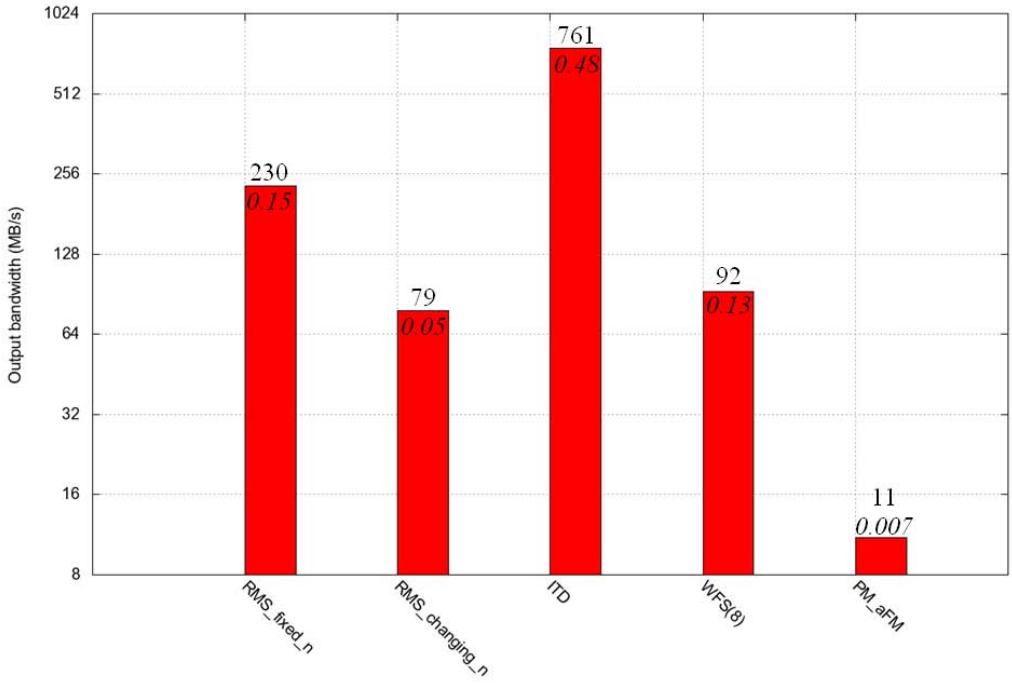

Figure 3.16: The chapter's example objects - output bandwidth graph. The test object depends from the number of output channels of each object. The objects are respectively: RMS with fixed number of analyzing samples, RMS with changing number of them, ITD panner, WFS set with 8 output channels, adaptive FM synthesis (delay-line based PM technique).

Chapter 4

Noisers

Introduction

Noisers are very used objects in electronic music, as sound or as parameter generators. They output random values, usually in a certain range, and with a certain distribution. A basic noiser has been already described in [GO03], but we are going now to modify that object and obtain different ones: starting from the reminding of that basic noiser, we'll implement a multichannel one with some *pattern-matching*, three *normal-distribution* ones and a *Bernoulli-distribution* one. Finally we'll use the basic noise to get a *dithering* adder objects. As usual, a computational overview and a brief conclusion will end the chapter.

4.1 Uniformly distributed mono noiser

In [GO03] a basic noiser has been presented, and since we're going to modify it to obtain all the successive noising objects, I'm going to remind you that noiser's code:

```
1 //-----------------------------------------------
2 //     Uniform mono noiser
3 //-----------------------------------------------
4
5 RANDMAX = 2147483647;
6 random = +(12345) : *(1103515245) ~ _;
7 noise = random * (1.0 / RANDMAX);
8 process = noise;
```

This object produces uniformly distributed random numbers, through the "unpredictable" wraping of the big variable values stored in `random` from positive to negative, and the successive normalization in `noise`, that fits that variable values

into the range [-1,1]. The method at the base of this algorithm is called of the *Linear Congruential Generator*, and is widely used among compilers. In general, this kind of generators have the following recurrence relation:

$$X_{n+1} = (aX_n + c) \mod m$$

In our case, we have the following parameters:

$$\begin{aligned} a &= 1103515245 \\ c &= 12345 \\ m &= 2^{32} \end{aligned}$$

Where m is not set, is "automatic" because of the 32 bits integer format. The choice of the other parameters is made so that all the m possible values are returned before the first value is repeated[1] – and then all the output pattern will be repeated periodically. Thus, due to the integer format, this generator will return numbers between -2^{31} and $2^{31} - 1$, and by dividing them by `RANDMAX` (that is $2^{31} - 1$), we normalize this range to [-1,1]. While the maximum pre-normalization random value is in fact equal to `RANDMAX`, the minimum one as you can see is slightly lower, but due to the loss of precision of the *floating point* format, the result of the division of that extreme by `RANDMAX` will be rounded to -1. But this means that -1 will be a bit more probable than the other values.

However, the output of this object is a stream of pseudo-random values, changing at sampling rate, and so perceived as a white noise. If we needed a random controller instead, we could use the S&H object with some kind of triggering signal (see 2.1 for S&H description and **??** for a coupled use of it with a noising object).

A particular property of this object is that it's fully-deterministic, so the first returned number will be always -0.695191, the second always -0.344851, the third always 0.106763 and so on. It's obvious, if you try it tomorrow nothing will be changed in the functions and the returned values will be always the same. For the same reason, if I put 10 copies of that object simply in parallel, then my outputs will be ten -0.695191 for the first sample, ten -0.344851 for the second one, and so on: the output channels will have an identical values stream. So, if you need for example a stereo noiser, you can't simply put two of these noisers in parallel: the result would be a mono stream split into two channels, that is an other thing than a true stereo noiser – you need in fact two different output streams. But don't worry: a multichannel noiser development is possible and is the topic of the next section.

[1] see [Knu97]

4.2 Uniformly distributed multichannel noiser

The idea is that we assign n successive *Uniform mono noiser*'s values to n output channels, so that we obtain n channels's 1-sample values from 1 channel's n-samples values. So we have to build a chain of n `random` functions in sequence, with the recursion only between the last chain's `random` and the first one. This sequence composition should also output the partial results, not only the last chain's element value. So it has to split each `random` output into two "wires", one linked to the successive element, the other one outgoing from the chain to return each of the n random numbers. We can use the *pattern-matching* technique to define this kind of special sequence composition:

```
S(1,F) = F ;
S(i,F) = F <: S(i-1, F), _ ;
```

The two arguments of this `S` function are the number of functions to be used in the chain (`i`) and the function to be used (`F`). Notice the "general" `S(i,F)` definition: from `F` (that in our case will have a mono output) there is a split (`<:`), to the successive parallel composition; this can be seen as a unique block with two inputs, the one that goes into the `S(i-1, F)` function (one because the `F` we are going to use has one input) and the one that goes to the Identity function (`_`) and so outside our chain. The result of this pattern-matching is showed in fig. 4.1.

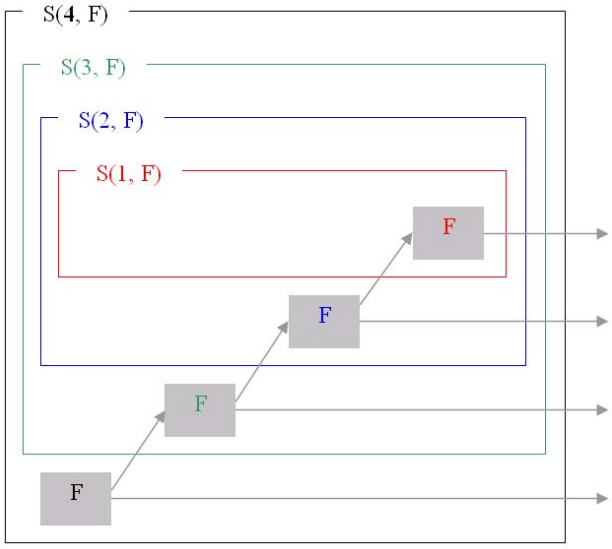

Figure 4.1: The $S(4, F)$ pattern matching scheme. The resulting object has 4 outputs, because at each step the F function output is split: a wire goes to the S(i-1, F) block and the other one goes outside the object (trough the Identity function).

This special sequence operator will be used to build our `randoms` chain. Now we have to redefine the `random` function, because we've said the recursion has to be only between the last chain's `random` and the first one. So we'll write simply:

```
random = +(12345) : *(1103515245);
```

Thus, our chain can be simply written in the following way:

```
chain(n) = S(n,random) ~ _;
```

where `n` is the number of output channel we need.
The last thing we need to do is to normalize the random values (to fit them into the range [-1,1]), by dividing them for `RANDMAX`. As we have n outputs now, we can't simply put a `/(k)` function in sequence with the `chain(n)` function – we would divide only one output in that way, because the `/(k)` function has only one input. We need a dividing function `Divide(n,k)` that takes n inputs, divide each of them by k, and outputs the n results separately: you can verify this is done by a parallel composition of n `/(k)` functions. Ok, this can be made by a single-string definition in FAUST!

```
Divide(n,k) = par(i, n, /(k));
```

The `par(a,b,F)` construction makes a parallel composition of b 'F_a' functions, with a running from 0 to $b-1$. It has been presented in [Orl07], section 5.6. Since in our case the dividing functions are identical, we don't use the information of their running index. So, with the string just showed, we build a parallel composition of n '`/(k)`' functions.
Finally we have to sequencially compose the `chain(n)` function with the `Divide(n,k)` one, replacing `k` with `RANDMAX` that was the original normalizing number. I'll call this function `NoiseN(n)` (whose name stays for 'noise-on-n-channels'):

```
NoiseN(n) = chain(n) : Divide(n,RANDMAX);
```

Then, in the `process` definition, the current number of output channels needed n will be set. Let's summarize the whole code now:

```
1 //-----------------------------------------------
2 //     Uniform multichannel noiser
3 //-----------------------------------------------
4
5 // Special sequence function:
6 S(1,F) = F;
7 S(i,F) = F <: S(i-1,F), _;
8
9 // Multichannel division function:
```

```
10 Divide(n,k) = par(i, n, /(k));
11
12 random = +(12345) : *(1103515245);
13 RANDMAX = 2147483647.0;
14 chain(n) = S(n,random) ~ _;
15 NoiseN(n) = chain(n) : Divide(n,RANDMAX);
16 process = NoiseN(3);
```

This is a 3-channel noiser then. You can finally find the relative .svg block-diagram in fig.4.2.

4.3 Normally distributed mono noiser

If you are going to use our noiser as a random generator, then for many applications you'll find it terrible! If you are going to assign, for example, its output to a oscillator frequency (with some kind of mapping I hope!), then the result won't be very natural. In fact you might expect the randomized frequency to be set with different probability to central values rather than to extreme ones. The *uniformly distributed noise* then is not what you need, because each value has the same occurrence probability. The *normally distributed* (or *Gaussian distributed*) one is a better choice, since the outputed values follow a Gaussian-distributed probability, centered in our case in 0, and with the typical bell shape. This Gaussian distribution can be obtained in several ways from the uniform one, but many of these ways (for example the *Polar* or the *Ziggurat* algorithms) use the *rejection sampling* technique, in which some generated values have to be rejected, and for this reason the technique is not usable in a *synchronous dataflow language* like FAUST. We are going to use two "direct" techniques instead, the first applying the *Central Limit Theorem*, the second using the *Box-Muller transform*.

4.3.1 Central Limit Theorem technique

The *Central Limit Theorem* states that the sum of a large number of independent and identically-distributed random variables will be approximately normally distributed if the random variables have a finite variance. Is this our case? Since we are going to use uniformly distributed variables U_i, we know that their variance is:

$$Var(U_i) = \frac{(b-a)^2}{12}$$

where a and b are the range extremes of the U_i random variables. Since we will use for the U_is the values returned by the code seen in 4.2, we know that this range is [-1,1]. Thus in our case we'll have:

$$Var(U_i) = \frac{1}{3} < \infty$$

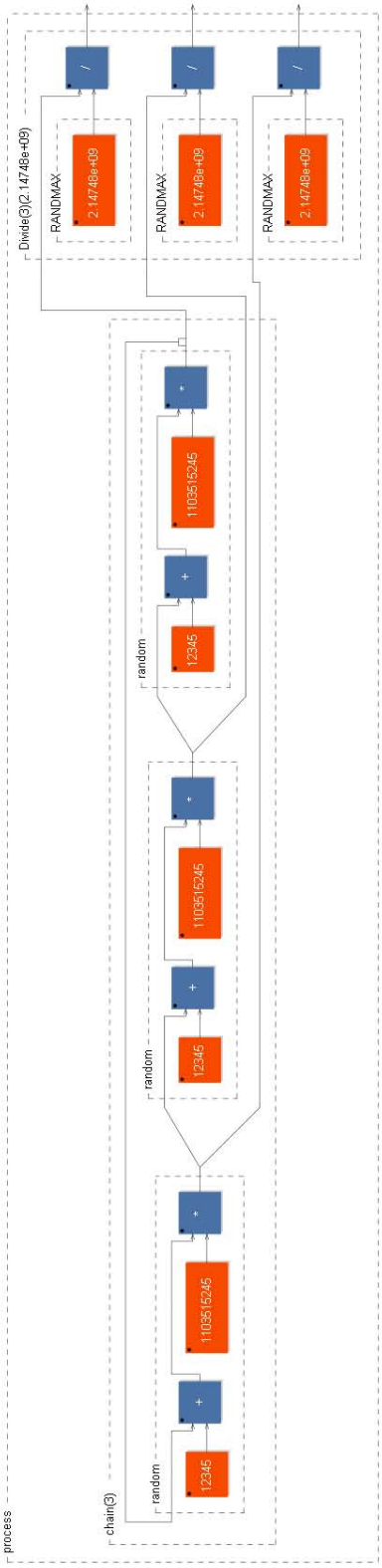

Figure 4.2: The "Uniform Multichannel Noiser" block-diagram.

So we can apply the *Central Limit Theorem*. We have just to sum a "large number n of independent and identically-distributed (in our case uniformly distributed U_i) random variables". Accordingly to the Central Limit Theorem, if we define a new variable Z_n in this way:

$$Z_n = \frac{S_n - n\mu}{\sigma\sqrt{n}} \quad \text{with } S_n = \sum_{i=1}^{n} U_i$$

where μ and σ^2 are respectively the mean and the variance of the U_i, then Z_n converges in distribution towards the *standard normal distribution* (in which the mean is 0 and the variance is 1). Since we are using uniformly distributed variables in the range [-1,1], their mean μ is 0 and their variance σ^2, as just said, is $\frac{1}{3}$. So in our case Z_n becomes:

$$Z_n = \frac{S_n\sqrt{3}}{\sqrt{n}}$$

This Z_n function is the one we are going to implement in FAUST, and the greater we choose n, the more close to a normal distribution Z_n will be. A very happy choice of n would be 3, because the Z_n expression would be very simple then (only the sum S_n of the U_i variables); but 3 is a too low number to well approximate the normal distribution, and its probability density function shape would seem more like a triangle. A more typical choice is 12, in fact:

$$Z_{12} = \frac{S_n}{2}$$

is a simple expression too. This will be our case. So, let's see the FAUST code.

```
1  //-----------------------------------------------
2  //   Gaussian noiser (CL theorem)
3  //-----------------------------------------------
4
5  // Uniform multichannel noiser:
6  S(1,F) = F;
7  S(i,F) = F <: S(i-1,F), _;
8  Divide(n,k) = par(i,n,/(k));
9  random = +(12345) : *(1103515245);
10 RANDMAX = 2147483647.0;
11 chain(n) = S(n,random) ~ _;
12 NoiseN(n) = chain(n) : Divide(n,RANDMAX);
13
14 // Gaussian noise:
15 Normal = NoiseN(12) : float :> _ : /(2);
16 process = Normal;
```

I've simply copied the *Uniform multichannel noiser* code in the first part. Then you can see the `Normal` variable definition, that first calls our *Uniform*

multichannel noiser with 12 outputs (`NoiseN(12)`, then only the first channel goes through a `float` function (I will explain this later), then a `:>_` operator merges the 12 channels values summing them into one channel, and finally the result is passed to the `/(2)` function. This string calculates what we've called $Z_1 2$. The `process` definition then simply calls this function.

I have now to explain to you the meaning of that "`float`" function inside the `Normal` definition. Without it, the factorization FAUST algorithm would simplify the calculation unifying the addictions and the multiplications-divisions, trying to optimize the number of operations to do. But in this case we don't want FAUST to change the operations inside the random generator (in particular the `chain` function), because its well working depends on the execution of exactly that operations in exactly that order. We can say that since we are using a *modular arithmetic* for the random generation (in fact positive numbers wrap into negative ones), the operations `+` and `*` we are using inside it are not the same we use outside it; for example adding 1 inside the random generator could turn a positive number into a negative one, while outside it, where numbers are less big, this behavior is not expected (or at least, is not wanted): so in a certain way that "+" used into the modular arithmetic is expected to work in a different way than the usual "+", so it's a different operation, and the same goes for the multiplication. These operations are different as long as the numbers involved are big enough, i.e. before the normalization trough the division by `RANDMAX` in the `NoiseN` block; so results could change if operations migrate from outside to inside that block or viceversa, but this is exactly the kind of stuff that happens during FAUST's optimizing factorization. If we put that "`float`" function then, FAUST won't try to factorize things before it with things after it, because at this time it doesn't know how to do that, and this solves the problem. Looking at the results, I've noticed it is sufficient to put this function only on the first channel, even if FAUST could factorize still among the other channels, but actually this idea seems not to reach it. I'm working with the `0.9.94j-par` version, but in future versions of FAUST maybe you'll have to put the "`float`" function on each channel, by replacing it in the code with the already seen `par` construction (used in 4.2):

```
par(i, 12, float) ;
```

You can see the `.svg` block diagram in fig. 4.3 and the resulting output statistics of both the wrong (without use of `float`) and the correct version of this object in fig. 4.4. You'll recognize the typical bell-shape in the correct version's output histogram, and just something wrong in the other one's.

4.3.2 Box-Muller transform technique

The *Box-Muller transform technique* makes use of only two uniformly distributed variables, to build a normally distributed one; on the other hand, it uses some very expensive functions like square roots or logarithms.

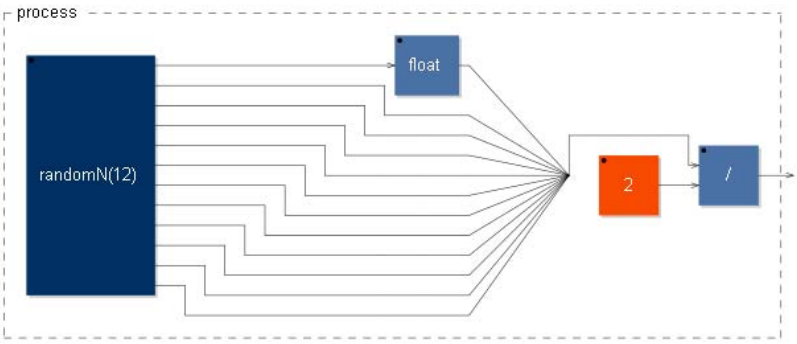

Figure 4.3: The "Central Limit Theorem technique Gaussian noiser" block-diagram.

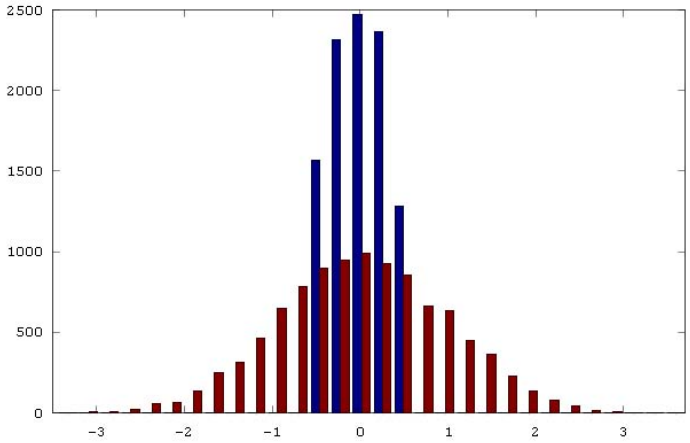

Figure 4.4: The "Central Limit Theorem technique Gaussian noiser" output statistics. Are shown here both the wrong object's results (blue) and the correct object's ones (red). There is no doubt that something is wrong in the blue histogram. The analysis has been made on 10000 samples in the way shown in **??**.

If U and V are our two independent random variables uniformly distributed in the interval (0,1], then the two following functions are independent random variables both with a normal distribution of standard deviation 1:

$$Z_0 = \sqrt{-2\ln U}\cos(2\pi V)$$

$$Z_1 = \sqrt{-2\ln U}\sin(2\pi V)$$

As in the previous case, we'll use our `NoiseN` function. Since the range of its output channels is [-1,1] and we need now a range of (0,1], we'll have to map that interval into the new one. The mapping function $M(x)$ has to preserve the uniform distribution, so it may be linear. The most obvious is this:

$$M : [-1, 1] \longmapsto (0, 1]$$

$$M(x) = \min(\frac{x+1}{2} + \epsilon,\ 1)$$

where ϵ is the smallest *normal* number in the *32 bits floating point* format (that is the one actually used by FAUST). Its value is:

$$\epsilon = 2^{-126}$$

Of course, $M(x)$ is not so linear because of the min function; but its presence changes the linearity of the mapping only if the independent variable value is 1, and the probability of this event is very low: it is $\frac{1}{m}$ where m is the number of the possibile returned values in the interval [-1,1], so because of the random generator we are using it's $m = 2^{32}$ (see m in 4.1). Thus the non-linearity introduced by the min function occurs exactly once every m samples, that is every 27 hours of noise at a sampling rate of 44100 Hz! We can consider then the $M(x)$ mapping linear enough. Inside the FAUST code then, the `min(1)` function is not necessary because the number $1 + \epsilon$ is already rounded to 1 because of the precision loss of the floating point format.

Thus, the FAUST code will call the `NoiseN` function with 2 outputs, and assign them to U and V variables by the mapping function $M(x)$:

```
U = NoiseN(2) : _,! : +(1) : /(2) : +(epsilon);
V = NoiseN(2) : !,_ : +(1) : /(2) : +(epsilon);
```

To select one of the two `NoiseN` output channels, I've used the object `wire,!` that is like the parallel composition of a cable and a pair of scissors (function `!`). In the first line, I keep only the first `NoiseN` output channel, in the second line the second one, changing the order of the said functions.

For the `epsilon` constant (ϵ), I'll use the `pow(a,b)` function, which returns the number a^b. I'll take it a bit greater by multiplying by 1.1, to be sure that it will not be rounded to a denormal number because of its decimal representation inside the C++ generated code. So `epsilon` is:

```
epsilon = pow(2, -126) * 1.1;
```

Usually, where it's possible, you should try not to use this function, because it is computationally very expensive (the most expensive of all!). But in this case the FAUST compiler will recognize that its result is a constant and it will replace it with the explicit value inside the generated C++ code. If you don't believe me, simply check this line inside the `for` cycle in the generated C++ code:

```
output0[i] = (sqrtf((0 - (2 * logf(1.17549e-38f + [...]
```

The small number `1.17549e-38f`, that like on the calculators stays for $1.17549 \cdot 10^{-38}$, is our `epsilon`. The rest of the code is really trivial. Apart from the random generator strings, that here are suppressed but that you should know quite well at this point, the whole code is the following one:

```
1  //-----------------------------------------------
2  //   Gaussian noiser (BM transform)
3  //-----------------------------------------------
4
5  import("math.lib"); //for PI definition
6
7  // Uniform multichannel noiser:
8  [...]
9
10 // Variables assignment:
11 epsilon = pow(2, -126) * 1.1;
12 U = NoiseN(2) : _,! : +(1) : /(2) : +(epsilon);
13 V = NoiseN(2) : !,_ : +(1) : /(2) : +(epsilon);
14
15 Z = -2 * log(U) : sqrt : *(cos(2 * PI * V));
16 process = Z;
```

Using table look-up Ok this is the typical case in which some expensive operations can be replaced by look-up tables. This kind of technique has already been shown in [GO03] in the *Harmonic Oscillator* example, for the *sin* function. The idea is that we calculate the expensive operations for a certain number of values in the range we need, store these results in a `rdtable` at the initialization time, and then simply consult this table during the running process. The look-up task has a certain time cost, so its use has to be preferred only for computationally expensive operations.

Let's change so the previous code: we have to store the U and V operations chains inside two separate `rdtable`s. To do so, we have to get these results, computed for the needed interval (0,1], in the form of two signals. So we first need for each operation chain a counter normalized into the interval we said. This can

be made with two different counters, the first incrementing by one at each sample (I've called this `time`), and the other one (that I've called `i`) dividing its value by an opportune number. Since we need the range (0,1], the first counter will start from 1, and the second one will divide it by the table size. In this way, the 0 will be excluded, and the maximum value will be 1:

```
time = +(1) ~ _;
i(time,size) = time / size;
```

where of course `size` represents the table size. Now, we have to define the signal where to store the operations chain result. I'll call the respective U and V chains resulting signals `Utable` and `Vtable`:

```
Utable(time,size) = -2 * log(i(time,size)) : sqrt;
Vtable(time,size) = cos(2 * PI * i);
```

Maybe it's better if we write these two functions with the "with" synthax:

```
Utable(size) = -2 * log(i) : sqrt
with {
     time = +(1) ~ _; // i = 1,2,...
     i = time / size;
     };

Vtable(size) = cos(2 * PI * i)
with {
     time = +(1) ~ _; // i = 1,2,...
     i = time / size;
     };
```

Now we have to build the two `rdtables`, that have the following syntax:

`rdtable`(size, values, read-index)

where "size" is the size (in samples of course), "values" the signal whose values will be stored sample-per-sample into the table (in our case it is `Utable(size)` or `Vtable(size)`), and "read-index" the index that says during the running process which cell's value we want to be read. The read-index will be the value of our U or V variables, normalized into the table size, so multiplied by `size`, and forced to be integer. So, the resulting Z variable will have the following expression:

```
Z(size) = U * size : int : rdtable(size,Utable(size)) :
    *(V * size : int : rdtable(size,Vtable(size)));
```

(notice that this is a single-line code because of the ";" presence only in the second line). The last thing to do is to limit the bounds of U and V variables, because

the compiler has to recognize the read-indexes range: we have to put then the `min(1)` and `max(0)` expressions into their definitions.

We have finished, the whole new code is the following one:

```
1  //-----------------------------------------------
2  // Gaussian noiser (BM transform) + rdtables
3  //-----------------------------------------------
4
5  import("math.lib");
6
7  // Uniform multichannel noiser:
8  [...]
9
10 // Variables assignment:
11 epsilon = pow(2, -126) * 1.1;
12 U = NoiseN(2) : _,! : +(1) : /(2) : +(epsilon) : min(1) : max(0);
13 V = NoiseN(2) : !,_ : +(1) : /(2) : +(epsilon) : min(1) : max(0);
14
15 Utable(size) = -2 * log(i) : sqrt
16 with {
17      time = +(1) ~ _; // i = 1,2,...
18      i = time / size;
19      };
20
21 Vtable(size) = cos(2 * PI * i)
22 with {
23      time = +(1) ~ _; // i = 1,2,...
24      i = time / size;
25      };
26
27 Z(size) = U * size : int : rdtable(size,Utable(size)) :
28         *(V * size : int : rdtable(size,Vtable(size)));
29
30 process = Z(1<<16);
```

In `process`, inside `Z`'s argument, you have to set the size of the two `rdtables`. The bigger is that number, the more accurate is the collection of the `rdtable` data, but the more memory is required. A value of 1000 is good enough to generate a distribution quite identical to the original version, without `rdtable`s, one. You can see in fact the confront between the distributions of the two object's versions in fig. 4.5, where `rdtable`'s size has been set to 1000. However, this number limits the possibile returned values variety: in fact if you set it to n, then each of the two U and V variables could read exactly n different values inside the respective table, so $Z(n)$, being a combination of the two look-ups, will return n^2 different

values. To have the same variability used for the *uniformly distributed noiser*, so 2^{32} values, n should be set to 2^{16}. This is why I've put `Z(1<<16)` inside the `process` definition, because it's equivalent to writing $Z(2^{16})$.

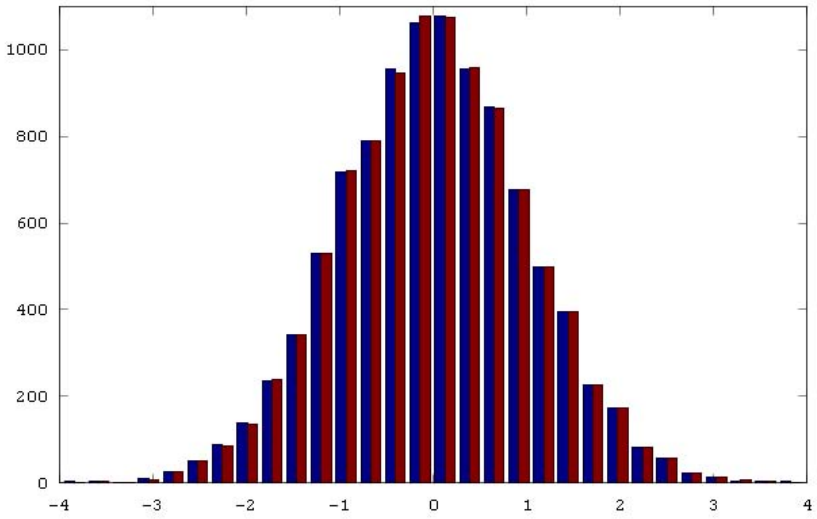

Figure 4.5: The statistics of the output of the two seen versions of the Box-Muller transform noiser. The "original" object's output statistics are the blue histogram and the "look-up tables" object's output statistics are the red one. Their shapes hare identical enough for musical purposes. The analysis has been made on 10000 samples.

4.3.3 Comparison between the three techniques

We have seen three ways of producing a normally distributed random variable. Which of the three objects should we choose? We have to evaluate which is the most correct and which is the lest expensive. About the former evaluation, see fig. 4.6, in which are shown the first two functions output statistics – the third one, the look-up tables version for the *Box-Muller transform* object, has already been shown in fig. 4.5, and we've found its shape identical to the without-look-up-tables version. I would say that the two functions return the same shape, apart from normal statistical fluctuations. In fig. 4.7 then you can see these object's output bandwidth instead. The *Box-Muller transform technique* with look-up tables is quite 2 times faster than the *Central Limit Theorem technique* on my computer, and more than 4 times faster than the *Box-Muller transform technique* without look-up tables. Different results, however, could occur with a vectorializing compiler (see **??** for a discussion on different compilers.), because heavy arithmetical functions like square roots and logarithms have a very better behavior on that kind of compilers. On my machine and with my Gcc compiler,

however, the choice is not difficult and the "Box-Muller transform technique" with look-up tables is very preferable.

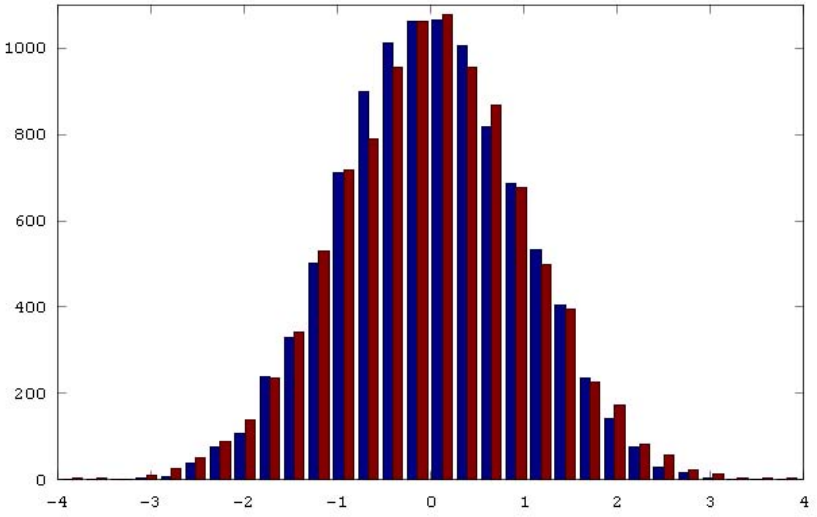

Figure 4.6: The statistics of the "Central Limit Theorem technique" output (in blue) and the "Box-Muller transform technique" without look-up tables (in red). Apart from normal statistical fluctuations, their shapes look identical. The analysis has been made on 10000 samples.

4.4 Noiser with Bernoulli distribution

A random variable with *Bernoulli distribution* is very useful for triggering events, since its output can be only 1 (with probability p) or 0 (with probability $1-p$). It is very easy to simulate it through a uniformly distributed random variable U: in fact if its interval is [0,1] then we can define a new variable B in the following way:

$$B(U,p) = \begin{cases} 1 & \text{if } U < p \\ 0 & \text{else} \end{cases}$$

So $B(U,p)$ will be 1 if U will be less than p, and since U is uniformly distributed, this event has exactly the probability p. The opposite case is complementary, so it has the probability $1-p$. Thus, by definition $B(U,p)$ has a Bernoulli distribution. Inside the FAUST code, all we have to do is to map the *uniformly distributed* noiser values, whose range is [-1,1], into the interval [0,1]: this can be done for example taking the absolute values. The code becomes then the following:

```
1 //-----------------------------------------------
2 //   Bernoulli distributed noise
3 //-----------------------------------------------
```

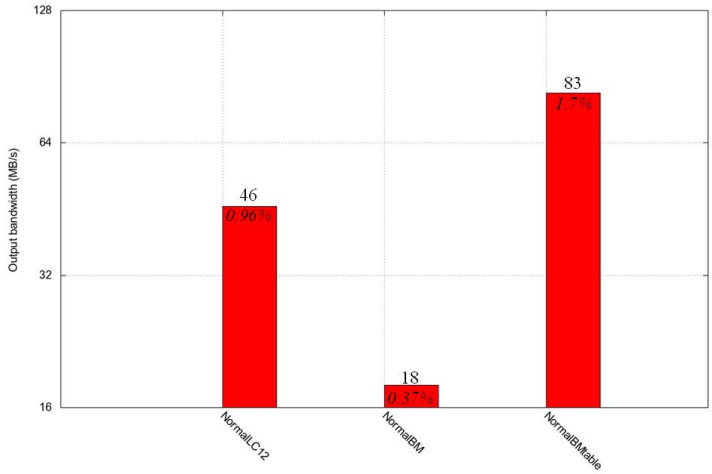

Figure 4.7: The output bandwidth respectively of the "Central Limit Theorem technique", of the "Box-Muller transform technique" without look-up tables, and of the "Box-Muller transform technique" with look-up tables Gaussian noisers. As usual, I've used a test value to confront these objects results with the other ones, in terms of pure computational cost. So I've used an object without inputs and with one output, like the three noisers, but without calculations. On my machine, this object had a huge output bandwidth of 4.8 Gb/s, because of the lack of any input reading time cost.

```
4
5 import("music.lib"); // for 'noise' definition
6 p = hslider("Probability parameter",0.5,0,1,0.0001);
7 Bernoulli(p) = abs(noise) < p;
8 process = Bernoulli(p);
```

I remind you that the `<` operator returns 1 if the condition is true, 0 else, while the function `abs` returns the absolute value of its argument. So `abs(noise)` is our U variable in the correct interval [0,1].

4.5 Anti-denormal dithering

Sometimes could happen that, especially because of recursive filters, some non-zero values fall in absolute value under the balanced range supported by the *floating-point* format. These numbers are called denormal or subnormal, and could crash the computing. To avoid this, it might be necessary to add in that cases a dithering noise with absolute values greater than the smallest normal number. This can be made with the following code:

```
1 //-----------------------------------------------
2 //   Anti-denormal dithering
3 //-----------------------------------------------
4
5 import("music.lib"),;
6 epsilon = pow(2, -126) * 1.1;
7 dith = ((noise > 0) * 2 - 1) * epsilon;
8 process = +(dith);
```

I've imported the music.lib library first, for the *Uniform mono noiser* definition (see 4.1). Then I've defined the `epsilon` constant (ϵ), that represents the smallest normal number in absolute value for the 32 bit *floating point* format (already used in 4.3.2). Then the `dith` function takes the noise output (range [-1,1]) and first converts it in a binary output through the `>(0)` function; so at this point we have random values in the set {0,1}. Then the successive operations map these outputs in the set $\{-\epsilon, \epsilon\}$. So the `dith` function outputs randomly a ϵ or a $-\epsilon$. The `process` then adds these random values to the input signal.

This object should remove denormal numbers and solve crashing problems related to them. For example, in fig. 4.8 is shown the output bandwidth of a "denormal noiser" compared with the same object in sequence with the *Anti-denormal dithering*.

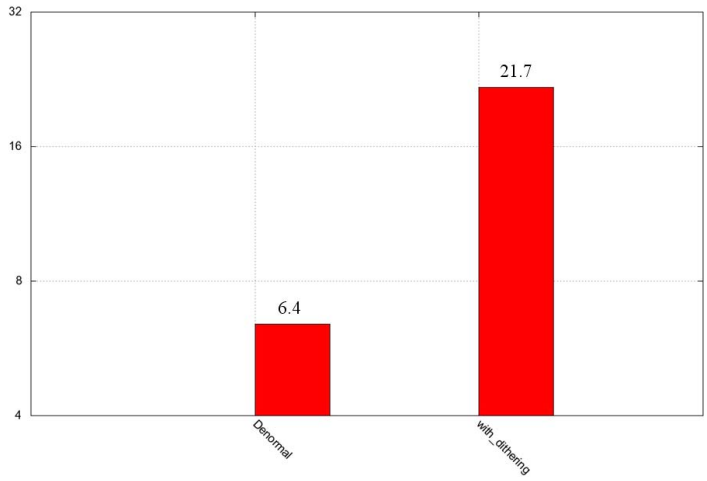

Figure 4.8: The comparison between a denormal numbers generator and the same object in association with the "anti-denormal dithering".
Even if the second code requires more operations, it's more than 3 times cheaper because of the absence of denormal numbers.

4.6 Computational overview

In fig. 4.9 are shown the output bandwidths of the objects we've seen in this chapter (except the *Box-Muller transform* without look-up tables and the *Central Limit Theorem* Gaussian noisers, because we had discarded them in 4.3.3).

4.7 Conclusion

The noisers seen in this chapter are very useful as random numbers generators, especially in association with the *Sampling&Hold* object. You will see an example of this use in the *Granulator* (see **??**). Before this last object, in next chapter we will see some filter uses...

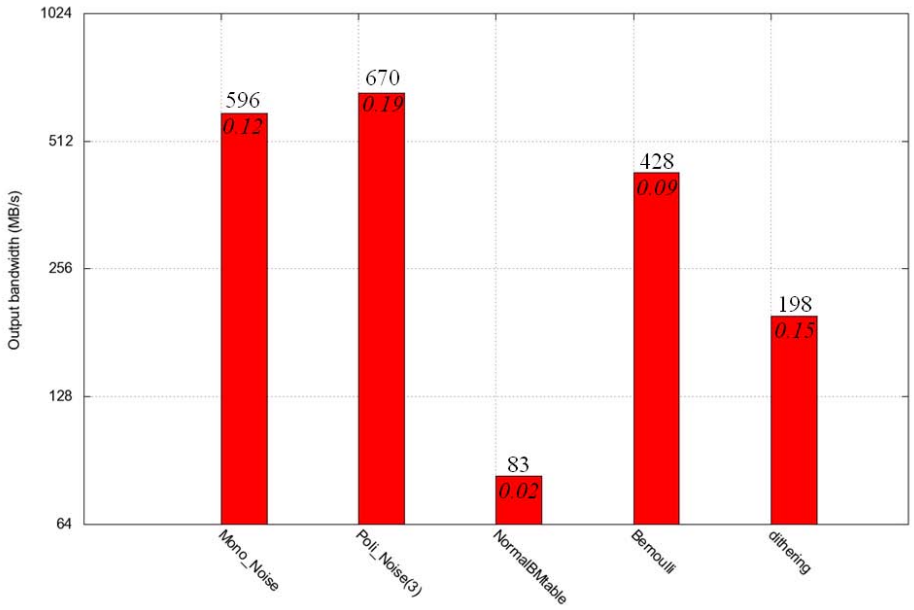

Figure 4.9: The output bandwidth of the chapters objects. You can see respectively the "Uniformly distributed mono noiser", the "Uniformly distributed multichannel noiser" (set with 3 outputs here), the "Gaussian noiser (with the Box-Muller trasform method with look-up tables)", the "Noiser with Bernoulli distribution" and the "Anti-denormal dithering". I've used different test object following the number of inputs and outputs of each object.

Chapter 5

Filters

Introduction

Filtering is a main tool in DSP applications, and plays a central role in physical modeling and sound manipulation. We have already seen some filters like the `smooth` for control signal smoothing in many objects, and highpass and lowpass filters for analysis tasks in the *Universal Pitch Tracker* (2.2.4); we'll see some other sound manipulation uses first, like the *Auto-Wha*, where will be also solved a precision loss problem with some numbers (5.1), or the more indirect use in the *Signal Sideband Modulation* (5.2). We'll see a heterodyning filter example in an alternative approach to the *Adaptive FM Synthesis* (5.3) and finally a physical modeling use in a *circular spazialisator* (5.4). As usual, a computational overview (5.5) and a brief conclusion (5.6) will end the chapter.

5.1 Auto-Wha

The "wha" effect is a very popular electric guitar pedal, that performs a spectrum emphasis around a running frequency, driven by the pedal's angle. The "auto-wha" is a variant of this effect, in which the running emphasizing frequency is driven by the amplitude of the input signal itself.

The *wha* in his classic version has been implemented by Julius Smith in his effect.lib FAUST library, in two different functions: `wha4`, using a 4th order Moog "Voltage Controlled Filter" (VCF) emulation, and a `crybaby`, that emulates directly this popular kind of wha pedals. We are going to use the last one for our *Auto-Wha*.

First, let's see the `crybaby` syntax. It simply requires the pedal's angle, normalized into the range [0,1], where 0 sets the minimum filter's frequency allowed and 1 the maximum one. So we have to analyze the input signal's level and map it into this range. The resultant scheme is shown in fig. 5.1.

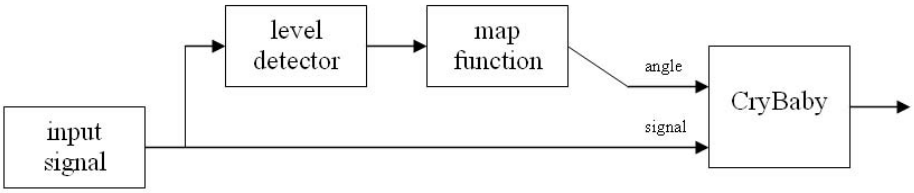

Figure 5.1: The "Auto-Wha" scheme

Level detector To analyze the level, we could use the *RMS* object seen in 3.2. But in this case an absolute values average, without squares and the expensive square root, is sufficient. First, we have to define the *summing function* `Sum(n,x)`, that is identical to the `S(n,x)` function used inside the *RMS with fixed n* object:

```
Sum(n,x) = +(x - (x @ n)) ~ _;
```

I remind you that inside this function is stored the sum of the last n samples value. Now, as I said, we'll simply use the average of the absolute values – we have to take the absolute values because else negative ones could make nulled the simple average for non-zero signals. A logic definition should be the following one:

```
Average(n,x) = x : abs : Sum(n) : /(n);
```

This should return the following expression:

$$Average_t(n,x) = \frac{\sum_{i=0}^{n-1} |x_{t-i}|}{n}$$

The problem is that if we defined the `Average` function as seen, its results would be too much distorted by the *floating point* format loss of precision. In fact, the floating point format has an higher precision around 0 than in greater numbers. So, when the `Average(n,x)` value increases enough, its precision becomes minor than the single samples values one, and it could happen that when the signal goes to zero, the subtraction of the old signal values brings the `Average` to negative values. Then we have to convert the calculations inside the `Average` function to a non-floating point format. This can be done by multiplying by a big integer number and converting the result to the integer format at the beginning of the incriminated expression, and by converting then to floating point format and dividing for the same big integer number at the end of it. The following expression defines the `Average` function with this kind of conversion inside:

```
Average(n,x) = x *(1<<22):int : abs:Sum(n) : float:/(1<<22) : /(n);
```

The choice of the multiplication and division by `(1<<22)`, that is 2^{22}, is not casual. In fact I'm going to set n, the number of analyzed samples, to 1000. Thus, the maximum `Sum(1000)` value will be 1000, being [-1,1] the signal normal range.

This number requires 10 bits to be stored (considering also the sign), because the biggest unsigned number you can express with n bits is $2^n - 1$. Then there are other 22 bits free, in the 32 bit integer format, and I can multiply by 2^{22} knowing the result won't go in overflow. In this way, being the precision the same for all magnitudes, the problem of the precedent `Average` version is not more present.

Mapping function Now that the level detector is ready, let's see the successive mapping function. The input of this function will be the output of the level detector, and this can't exceed the range [0,1]. Also the output of the mapping function has to be in the same range, because it will drive the normalized *wha* pedal angle. So, if the input signal reaches the maximum possible average level (it has to be a square wave of amplitude 1) the identity function is already a well working map. Else, in most of cases, the mapping function should "enlarge" the level detector's output by multiplying it by some value greater than 1. So our function should be something like this:

$$Map(x) = a \cdot x \qquad \text{with } a \geq 1$$

The a parameter will be set by the user accordingly to the input signal level and wave shape. To be sure the `Map` output won't exceed the allowed range, I have limited to [0,1] it through the `max` and `min` functions – who knows, maybe some roundings could make it negative... `max` and `min` functions are very cheep and it's better not to risk! The resulting code for the mapping function is the following:

```
Map(x) = x * a : max(0) : min(1);
```

CryBaby Now we have our driving signal, and we have to link it with the `crybaby` function. Its syntax, that you can verify in its definition in effect.lib , is:

```
crybaby(angle, signal)
```

so as "angle" argument we shall give the `Map` output, as "signal" the input signal. We can write this in this way:

```
process(x) = x : crybaby(x : Average : Map);
```

This is the first time we write an argument inside the `process`: this will become the object's inlet, and in this way we can give a name to its value, for example `x`. The `crybaby` function is called with one argument, that is so the first one ("angle"), and it's made by the chain we've spoken about. The `crybaby` with only this argument becomes a partial function, and we can "plug" into it the other needed argument, the signal, that will be our object inlet, `x`.

Whole code Finally, the whole code is the following.

```
1  //-----------------------------------------------
2  //     Auto-Wha
3  //-----------------------------------------------
4
5  import("effect.lib"); //for crybaby definition
6  a = hslider("Mapping",1,1,10,0.1) ;
7
8  Sum(n,x) = +(x - (x @ n)) ~ _;
9  Average(n,x) = x * (1<<22) : int : abs : Sum(n) : float : /(1<<22)
10                : /(n);
11 Map(x) = x * a : max(0) : min(1) ;
12 process(x) = x : crybaby(x : Average(1000) : Map) ;
```

5.2 SSM

The *Signal Sideband Modulation* (SSM) is a technique that returns only one of the two sidebands a classical *ring modulation* produces. Thus, if we keep only the upper sideband, it performs a spectrum shift, that differs from a classical pitch shift because here all the partials of a sound are shifted by the same amount. This turns an harmonic sound into an inharmonic one, because after the shift the ratios between the partials are no longer the original ones. If we keep only the lower sideband instead, it inverts the spectrum and shift it by a whatever amount. The mathematical instrument used is the *Hilbert transform* , with which negative frequencies are phase-advanced by 90 degrees and positive ones are phase-delayed by the same amount. This can be done with the use of some all-pass filters, which coefficient can be calculated with the *Parks-McClellan FIR filter design technique*, but we are not going to discuss here too technical issues. Then we'll multiply the filters output signals by a sine and a cosine wave, which frequency will determine the spectrum shifting amount like in a standard *ring modulation*. Finally, we'll have to subtract or to add the two ring-modulated results in order to obtain respectively the upper or the lower signal side band. The scheme of the main steps is shown in fig. 5.2. fig. 5.2.

5.2.1 Filters shaping

Fortunately, a well-working implementation of the filters we need is showed in the PURE DATA tutorial, inside the example called H07.ssb.modulation.pd. We are going simply to translate in FAUST language the filters used in that work. In the original patch, were used four *biquad~* objects, which in PURE DATA are set using the biquadratic filter coefficients of the difference equations in the so called *Direct*

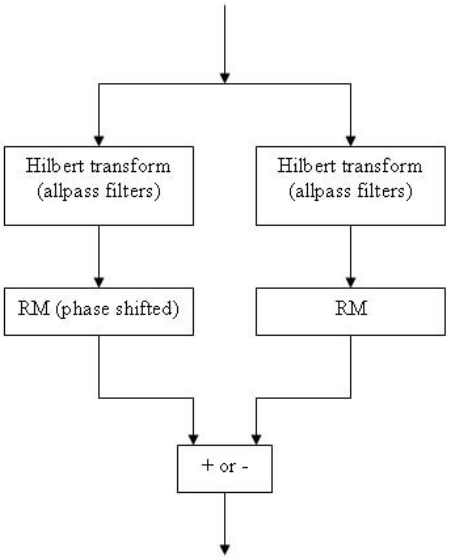

Figure 5.2: The "SSM" scheme

Form 2. This form's difference equations are the following:

$$y_n = b_0 w_n + b_1 w_{n-1} + b_2 w_{n-2}$$

where

$$w_n = x_n + a_1 w_{n-1} + a_2 w_{n-2}$$

The input signal at sample n is x_n, while the output, filtered signal at sample n is y_n. The coefficients b_0, b_1, b_2, a_1 and a_2 are the ones we'll copy from the PURE DATA patch. We could use the already existing `tf2` FAUST function, defined in filter.lib library, that implements a biquadratic filter. In that case we should know that its coefficients are used in a different difference equation, in which w_n is defined as follows:

$$w_n = x_n - a_1 w_{n-1} - a_2 w_{n-2} \quad \text{for tf2 function}$$

thus to have the same filter some signs in the coefficients should be changed. We are going to write instead from the beginning a `biquad` function with the same difference equations PURE DATA uses. I'll show you two possible way of writing it.

Filter code first version A first version code tries to transpose step-by-step the math different equation. The first line is obvious:

```
biquad(a1,a2,b0,b1,b2,x) = b0*w + b1*w' + b2*w'';
```

in fact it's exactly the transposition of $y(n)$ mathematical expression. I've simply used the "'" as one-sample delay line and "''" as two-samples one. But what about the $w(n)$ expression? Translating it in FAUST seems to be an easy task but it's not so. It's better if we draw the block-diagram first. It should look like the one in fig. 5.3. In fact, i take the x value, sum it with the $w(x)$ 2-samples delayed

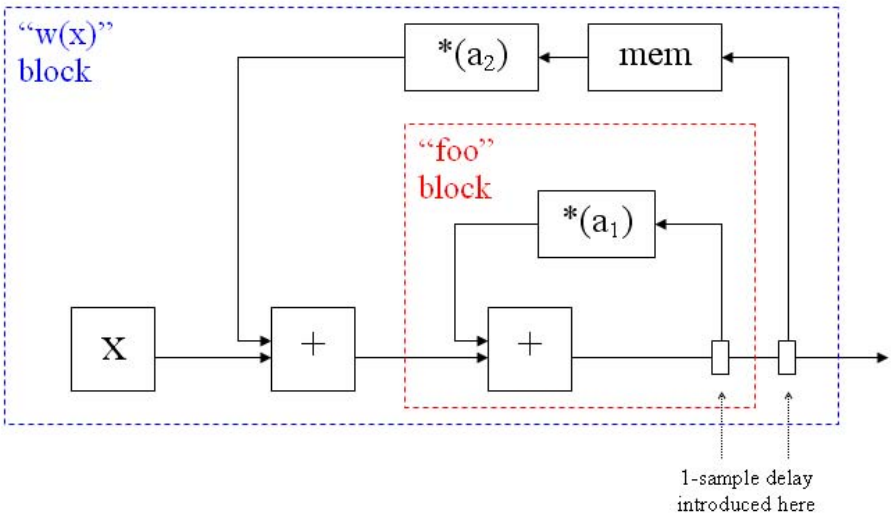

Figure 5.3: The "w(x)" block-diagram

value multiplied by a_2, and add finally the $w(x)$ 1-sample delayed value multiplied by a_1. Thus it is exactly the description of the $w(x)$ mathematical expression. You have only to be careful with the embeddings of the two recursive blocks: you can choose as you prefer the respective positions (which one has to stay inside and which one outside), because of the commutative propriety of the `+` operation; but it's essential that one stays inside the other, and not for example in sequence, because this is the only configuration in which all the informations reach both the memory levels (i.e. the 1-sample and 2-samples delayed loops). If you put them in sequence instead, the first one wouldn't receive the data from the second one. As the scheme shows, then, a 1-sample delay is introduced at the beginning of each loop, so the internal loop has already a 1-sample delay, while you have to add a `mem` function to the external loop to reach the desired 2-samples delay amount.

Now that we have the correct block-diagram, we have to translate it into a FAUST string. This can be done with the *Divide et impera* paradigm in three steps: in the first one (*divide*) we'll replace the internal loop description with a function name ("foo" is a perfect name in this cases!), and try to describe the remaining external structure. Then we'll translate the "foo" block with its specific code; finally (*impera*) we'll replace the "foo" block inside the first step's expression with the second step's code.

- First step. Ok, we have only a loop and some unary functions, describing

this in FAUST is not too difficult:

```
w(x) = x : (+ : foo) ~ (mem : *(a2)) ;
```

- Second step. The "foo" block has also a very simple expression:

```
foo = + ~ *(a1) ;
```

- Third step. Now simply merge the two expression:

```
w(x) = x : (+ : + ~ *(a1)) ~ (mem : *(a2)) ;
```

So, the whole `biquad` function definition is:

```
biquad(a1,a2,b0,b1,b2) = b0*w + b1*w' + b2*w''
with {
      w = x : (+ : + ~ *(a1)) ~ (mem : *(a2)) ;
} ;
```

Filter code second version A very elegant way of doing the same filter is the following code:

```
biquad(a1,a2,b0,b1,b2) = + ~ conv1(a1,a2) : conv2(b0,b1,b2)
with {
      conv2(k0,k1,k2,x)  = k0*x + k1*x' + k2*x'' ;
      conv1(k0,k1,x)     = k0*x + k1*x' ;
} ;
```

It could seem a bit hermetic, but it's exactly equivalent to the first version. Getting it requires a small trick. The trick consists in the assumption that the "w" block can be written in the form:

```
w = + ~ F ;
```

for a certain `F` function. This structure is represented in fig. 5.4, where is also labelled each signal flow. The first labels you should put are the input and output flows ones, `x` and `w` (that is $w(x)$ if you want to see it like a function as in the previous version, but now we don't need so). Then comes `w'` as a consequence, because that dataflow is a 1-sampled delayed version of the output one. Then comes `F(w')`, and we don't know how is this `F` now, so its output is for us whatever a function of its input `w'`. Finally, the `+` output is of course `x+F(w')`. Now we have just to find that `F` function, and it's not so difficult if we compare the two

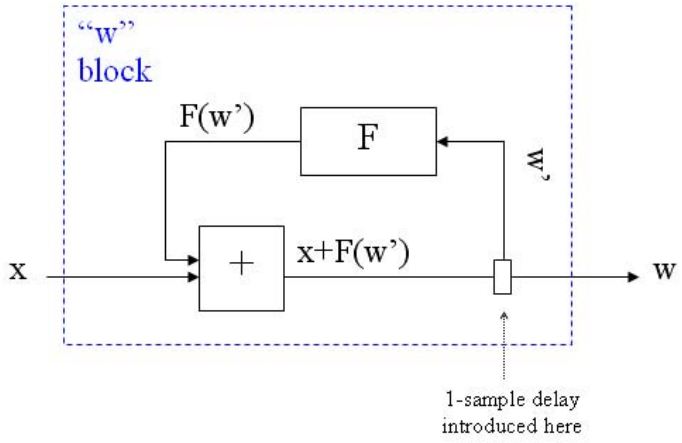

Figure 5.4: The "w" structure (filter second version)

different expressions we coherently used to label the same output dataflow. Let me translate them to a mathematical form:

$$x_n + F(w_{n-1}) = w_n$$

in which, as usual, the pedex indicates the time in samples. Now just replace the w_n expression with the required one, replace the w_{n-1} variable with k_n, and you'll obtain the following $F(k_n)$ expression:

$$F(k_n) = a_1 k_n + a_2 k_{n-1}$$

that in FAUST, after taking each parameter as variable, looks like the following code:

```
conv1(a1,a2,k) = a1 * k + a2 * k' ;
```

I've changed the function name in `conv1` because of the convolution in time domain this function represents. Then the `w` code is the following one:

```
w(a1,a2) = + ~ conv1(a1,a2) ;
```

and the `conv1` function has to be partial because it has to present an inlet (as said about the "w" structure assumption shown in fig. 5.4).

So we have just obtained the code for the w_n expression. The other equation, the y_n one, can be seen as a function that takes the `w` signal as input inlet. So it will be an other convolution, say `conv2`, with the more obvious code:

```
conv2(b0,b1,b2,w) = b0*w + b1*w' + b2*w'' ;
```

and it has to be partial and linked to the previous `w` code through a sequence operator, so that the previous dataflow becomes the `w` variable:

```
y(a1,a2,b0,b1,b2) = + ~ conv1(a1,a2) : conv2(b0,b1,b2);
```

Brackets are not necessary, because the recursive operator has a bigger priority than the sequence one. Ok, the work is done, all the strings we needed are been found; by changing some variable names, and using the "with" syntax, you'll obtain the `biquad` code shown at the beginning of the paragraph.

5.2.2 Interpolating oscillators

The filter is not enough for our object. We'll also need to multiply the filters output signals by a sine and a cosine wave. I've found the sound result much better, if the sine and cosine oscillators use an interpolated table looking-up. The sine oscillator with interpolation has already been used in 3.5, is called `osci` and is defined in music.lib . We need also a cosine oscillator, that is of course simply a phase-shifted version of an `osci`. So we could obtain that kind of oscillator by using an adaptive delay line on `osci`. Or we could more simply build a `cosci` function, similar to `osci` but with a cosine period stored in the memory table instead of a sine one. To do so, let's take a look to the original `osci` definition, inside music.lib:

```
osci(freq) = s1 + d * (s2 - s1)
 with {
 i = int(phase(freq));
 d = decimal(phase(freq));
 s1 = rdtable(tablesize+1, sinwaveform, i);
 s2 = rdtable(tablesize+1, sinwaveform, i+1);};
```

All the functions used here and not defined in the `with` syntax, are defined as stand-alone functions in the same file. You can see the table looking-up in `s1` and `s2` strings, where there are two `rdtable`s initialized with a signal called `sinwaveform`. Its definition is stand-alone:

```
sinwaveform = float(time)*(2*PI)/float(tablesize) :  sin;
```

This string has already been explained in [GO03] in the harmonic oscillator example, but I just want to remind you that all its first part is a normalized incrementing index that goes into the `sin` function of the second part. This is how the `sinwaveform` function generates a sine-wave-form. If we want a cosine-wave-form then, we have just to change the string second part; then `cosci` will be exactly like `osci` but its tables will be initialized with this other signal:

```
cosci(freq) = s1 + d * (s2 - s1)
 with {
```

```
coswaveform = time*(2*PI)/tablesize : cos;
i = int(phase(freq));
d = decimal(phase(freq));
s1 = rdtable(tablesize+1, coswaveform, i);
s2 = rdtable(tablesize+1, coswaveform, i+1);};
```

5.2.3 Whole code

The rest of the code is self-explaining. Notice only that `f1` and `f2` are the functions that realize the *Hilbert transform*, and each of them is made of two biquad filters in cascade. In `process`, finally, I've put the two possible sidebands in parallel, so that the left outlet will output the upper sideband, while the right outlet the lower one.

```
1  //-----------------------------------------------
2  //      SSM
3  //-----------------------------------------------
4
5  import("math.lib"); // for PI definition
6  import("music.lib"); // for osci definition
7
8  biquad(a1,a2,b0,b1,b2) = + ~ conv2(a1,a2) : conv3(b0,b1,b2)
9  with {
10       conv3(k0,k1,k2,x) = k0*x + k1*x' + k2*x'';
11       conv2(k0,k1,x)    = k0*x + k1*x';
12 };
13 cosci(freq) = s1 + d * (s2 - s1)
14 with {
15       cosinwaveform = time*(2*PI)/tablesize : cos;
16       i = int(phase(freq));
17       d = decimal(phase(freq));
18       s1 = rdtable(tablesize+1,cosinwaveform,i);
19       s2 = rdtable(tablesize+1,cosinwaveform,i+1);};
20
21 f1 = biquad(-0.02569, 0.260502, -0.260502, 0.02569, 1) :
22      biquad(1.8685, -0.870686, 0.870686, -1.8685, 1);
23 f2 = biquad(1.94632, -0.94657, 0.94657, -1.94632, 1) :
24      biquad(0.83774, -0.06338, 0.06338, -0.83774, 1);
25
26 a = hslider("Freq shift",0,-10000,10000,1);
27 process(sig) = f1(sig)*cosci(a) - f2(sig)*osci(a),
28                f1(sig)*cosci(a) + f2(sig)*osci(a);
```

5.3 Adaptive FM Synthesis (heterodyning technique)

5.4 Circular spazialisator

5.5 Computational overview

5.6 Conclusion

Bibliography

[GO03] E. Gaudrain and Y. Orlarey, *A Faust Tutorial*. Grame, Centre National de Creation Musicale, Lyon, 2003. Downloadable from http://faust.grame.fr/pubs.php.

[Gra06] A. Graef, "Q - equational programming language: 'examples' (website)," October 2006. 11/08/2008.

[Knu97] D. E. Knuth, *The Art of Computer Programming*, vol. 2: Seminumerical Algorithms. Reading, Massachusetts: Addison-Wesley, 3rd ed., 1997. pp. 17-19.

[Kuh90] W. B. Kuhm, "A real-time pitch recognition algorithm for music applications," *Computer Music Journal*, vol. 14, no. 3, pp. 60–71, 1990.

[Loy06] G. Loy, *Musimathics (The Mathematical Foundations of Music)*, vol. 1. The MIT Press, Cambridge, Massachusetts, 2006.

[LTL08] V. Lazzarini, J. Timoney, and T. Lysaght, "The generation of natural-synthetic spectra by means of adaptive frequency modulation," *Computer Music Journal*, vol. 32, no. 2, pp. 9–22, 2008.

[Orl07] Y. Orlarey, *Faust Quick Reference*. Grame, Centre National de Creation Musicale, Lyon, 2007. Downloadable from http://faust.grame.fr/pubs.php.

[Roe73] J. Roederer, *Introduction to the Physics and Psychophysics of Music*. The English Universities Press, London, 1973.

[Smi07] J. O. Smith, "Physical audio signal processing: 'lagrange interpolation' (website)," August 2007. 22/08/2008.

Index