Fb1 – an interface for single sample feedback and feedforward in SuperCollider

Daniel Mayer
University of Music and Performing Arts Graz
Institute of Electronic Music and Acoustics (IEM)
mayer@iem.at

ABSTRACT

The class Fb1, contained in miSCellaneous_lib, an extension of SuperCollider (SC), enables single sample feedback and feedforward with arbitrary block sizes. This is made possible by an iterative application of the defining relation in the SynthDef (instrument) graph, a method suggested by Nathaniel Virgo. While alternative approaches exist, e.g. by setting SC server's block size to 1, Fb1 provides a clear interface: the relation, which describes the calculation of subsequent samples from previous samples, is passed via an SC Function (the capital indicates the SC object) with the two arguments 'in' (feedforward) and 'out' (feedback). Consequentially, difference equations of linear filters can almost directly be taken over in the syntax, in which they are described in standard DSP literature. Because any operators can be written in the Function, non-linear filters and totally irregular feedback / feedforward setups can also be explored. Further options include arbitrary look-back depths and multichannel sizes, which allow complicated setups with a few lines of code. In August 2019 the class Fb1_ODE for integrating resp. audifying ordinary (systems of) differential equations with initial values in realtime was added to miSCellaneous_lib v0.22. It is based on Fb1 and will be described in more detail in a separate paper.

1. INTRODUCTION

Like many other audio engines, SC [1, 2] server processes audio data in chunks of *n* samples. *n* is called the block size and defaults to 64. The block size determines the minimum delay time, which cannot be overcome with SC's standard unit generators (UGens) for managing feedback. It equals

$$\frac{block\ size}{sample\ rate} \tag{1}$$

and, depending on the system, usually defaults to

$$\frac{64}{44100}\,sec = 0.00145\,sec \tag{2}$$

Copyright: © 2020 Daniel Mayer. This is an open-access article distributed under the terms of the Creative Commons Attribution 3.0 Unported License, which permits unrestricted use, distribution, and reproduction in any médium, provided the original author and source are credited.

This so-called control duration corresponds to the control rate frequency of ca. 689 Hz, which can often occur as a background tone under certain circumstances. Setting the block size to smaller values – still it must always be a power of 2 – raises the control rate frequency and makes control rate operations more CPU-costly, though it enables lower feedback delay times. Regarding the audible results and the behaviour of changes, the differences between block size 64 and 1 can be huge. With block size 64, for example, slowly evolving changes can sometimes be perceived up to the point of a blowup, whereas similar setups with block size 1 often immediately derail, and modulations of delay time can have equally disparate consequences. To summarise: from the viewpoint of practical sound synthesis, single sample feedback is not only a special option of "default" feedback (if we regard the latter as feedback with a minimum delay time equaling default control duration), but a method that behaves quite differently and therefore opens new and interesting aesthetic possibilities.

Now, as single sample feedback can, in principle, be easily achieved by setting SC server's block size to 1, why is there a need for a dedicated single sample feedback class? When setting block size to 1, single sample feedback is only possible in its raw form, which means that operators can be iteratively applied from sample to sample. Performing tasks like operating on a certain previous – and not only the last – sample (essential for many filters), multichannel handling and setting initial values would require extra implementational efforts on a per-case base. This all is covered by Fb1, and from my experience so far, this is its main benefit. As an extra bonus, Fb1 can do single sample feedback with arbitrary block sizes, though it doesn't make a difference from the viewpoint of the interface. Instead it's a trade-off: higher block sizes lead to higher compile times and larger SynthDefs, whereas the efficiency option of control rate unit generators is preserved. Which block size is practical depends on the concrete example: for experimenting with new SynthDefs, including Fb1, I tend to choose block sizes of 8 or 16. Originally Fb1 was assumed to tackle single sample feedback at audio rate. A control rate variant has been added to miSCellaneous_lib in version 0.22 [3].

2. IMPLEMENTATION

2.1 Iteration

A generalised feedback / feedforward relation can be written in the form *out[n] = F(out[n-1], out[n-2], ... , in[n], in[n-1], ...)* (3)

out[n] denotes the new output sample which is calculated on the one hand from feedback data, the previous output samples *out[n-1], ... , out[n-j]* and on the other hand from feedforward data, the current and previous input samples *in[n], in[n-1], ... , in[n-k]* by a function *F*. In the multichannel case with channel sizes *p* and *q* every *in[i]* and *out[i]* would denote a collection of *p* resp. *q* samples. *F* can be an arbitrary function, not restricted to the linear difference equations of the standard filter types. Implementing this relation with a block size $b > 1$ requires the *b*-fold usage of *F* for blockwise calculation of the samples *out[0], out[1], ... , out[b-1]* in the SynthDef graph and storing the results in a buffer.
It's important to note that *F* is implemented as a control rate operation, whose iterative *b*-fold application fills a buffer of *b* samples. After this filling (and this refers to the order in the SynthDef graph), an audio rate phasor reads out the data from the buffer. Fb1 is a so-called pseudo ugen, like a macro in other languages it establishes a compound structure, a subgraph of iterated UGens, in contrast to SC's basic Ugens, which are written in C++.

This conception of the procedure has been described in a thread on the SC mailing list by Nathaniel Virgo [4]. An open point at that time was the integration of audio rate input signals, the feedforward component. Fb1 implements it like this: data is stored in temporary buffers first and written with control rate to copied buffers thereafter. That way, the copied buffers can be used in the iteration process which is based on the order of the SynthDef graph. The temporary buffers cannot be taken for this purpose, as they are overridden immediately.

2.2 Look-back Depth

In the implementation of Fb1, the relation (3) is passed as a SC Function object, where the two arguments 'in' and 'out' denote the feedback and feedforward signals. Indexing within the Function refers to previous samples. E.g. SC's OnePole UGen with coefficient *c* is implemented by the formula

out[n] = (1 - abs(c)) in[n] + c out[n-1] (4)

This can directly be taken over for a re-implementation with Fb1, only the index offset -1 has to change the sign. An application to white noise with $c = 0.95$ would translate into the SC syntax

```
{
    Fb1.ar({ |in, out|
            (in[0] * 0.05) + (out[1] * 0.95)
        },
        WhiteNoise.ar(0.3)
    )
}.play
```

being equivalent to

```
{ OnePole.ar(WhiteNoise.ar(0.3), 0.95) }.play
```

Looking back to previous samples requires the setting of appropriate inDepth / outDepth arguments in Fb1, e.g. for a second order filter with feedforward coefficients *a0, a1, a2* and feedback coefficients *b1, b2* one would have to set inDepth and outDepth to the value 3:

```
{
    var a = [-0.6, 0.5, -0.7];
    var b = [0.5, -0.1];
    var src = Saw.ar(200, 0.1);
    // out[0] is passed formally
    // to allow taking over index convention
    Fb1.ar({ |in, out|
        (a[0] * in[0]) + (a[1] * in[1]) +
        (a[2] * in[2]) + (b[0] * out[1]) +
        (b[1] * out[2])
    }, src, inDepth: 3, outDepth: 3)
}.play
```

When referring not to adjacent collections of previous samples but to selected earlier samples with gaps in between, passing specified inDepth / outDepth indices is saving UGens. In the following example – supposed block size 64 – the expression `in[0]` still refers to *in[0]*, the current feedforward sample, but `in[1]` refers to *in[n-56]*. The double bracketing in the inDepth argument is necessary to distinguish from differentiation in a multichannel situation which would be done with single bracketing. The sound resulting from this setup and its following variants is reminiscent of a blend of voice and brass.

```
{
    var src = SinOsc.ar(500 * LFDNoise3.ar(5));
    Fb1.ar({ |in, out|
        (out[1] / max(0.01, (in[1] - in[0]))).tanh
        }, in: src, inDepth: [[0, 56]], outInit: 1
    ) * 0.1
}.play
```

2.3 Multichannel Handling

Feedback and feedforward signals can have an arbitrary number of channels, whereby the number of feedback channels must be explicitly passed by the key 'outSize'. Here a stereo source is passed to Fb1, and the recursion depth is differentiated per channel. Within the Function the expressions `in[0]`, `in[1]` and `out[1]` all refer to stereo signals.

```
{
    var src = SinOsc.ar(500 * LFDNoise3.ar(5!2));
    Fb1.ar({ |in, out|
        (out[1] / max(0.01, (in[1] - in[0]))).tanh
    },  outSize: 2, in: src,
        inDepth: [[0, 56], [0, 29]], outInit: 1
    ) * 0.1
}.play
```

Single components of a multichannel feedback / feedforward signal can also be addressed explicitly, which allows for easy establishing of complicated cross relations. Note that in the following example the expression `in[1][0]` means the sample *in[n-56]* of the first feedforward signal, and analogously `in[1][1]` refers to the sample *in[n-29]* of the second.

```
  {
    var src = SinOsc.ar(500 * LFDNoise3.ar(5!2));
    Fb1.ar({ |in, out| [
       out[1][0] / max(0.01, (in[1][0] * in[0][1])),
       out[1][1] / max(0.01, (in[1][1] * in[0][0]))
       ].tanh
    }, outSize: 2, in: src,
       inDepth: [[0, 56], [0, 29]], outInit: 1
    ) * 0.1
}.play
```

2.4 Further Fb1 Options

2.4.1 Initialisation

As arbitrary look-back depths can be passed to Fb1, there needs to be an option for setting initialisation values resp. sequences thereof for feedback and feedforward signals. This can be done with the Fb1 arguments ‘inInit’ and ‘outInit’. Sequences can be differentiated per channel in a multichannel case. See Ex. 3b of Fb1’s help file for the details of the convention.

2.4.2 Block Size

It's the user's responsibility to pass the correct number to Fb1 if the server’s block size differs from default 64. Other than that, irregular values of ‘blockSize’, not equal to a power of 2, can be used to play with artefacts.

Look-back depths exceeding the server’s block size can be defined as well – more likely to be desired with lower block sizes – by setting the argument ‘blockFactor’ to a sufficiently high value, ensuring that the maximum look-back depth is smaller than the product of ‘blockFactor’ and ‘blockSize’.

2.4.3 Differentiating the Function per index

Besides ‘in’ and ’out’ – as formal arrays referring to feedback and feedforward data – an index is passed as third argument to the Fb1 Function. That way the Function can be differentiated depending on the index within the block, which adds another possibility for conditional feedback.

3. APPLICATIONS

3.1 Filters

Simple examples of linear filters have been shown in chapter 2.2, in addition coefficients could be made dynamic, e.g. by passing audio rate signals via the 'in' argument. As there is already a source signal for the filter, the 'in' signal would have to be appended, which is no problem as the 'in' argument can take a multichannel signal of arbitrary size.

Re-implementing standard filters already contained in SC is of course not more interesting than just a proof of concept. However, there are many linear filters not implemented in SC, and writing them with the help of Fb1 can be useful, also as a test with the aim to write them as UGens in C++ later on.

More exciting from an aesthetic perspective might be the field of non-linear filters [5, 6]. An example of this kind is the Dobson-Ffitch filter [7], which contains a quadratic term:

$$out[n] = a\ out[n-1] + b\ out[n-2] + d\ out[n-L]^2 + in[n] - c \quad (5)$$

This example with GUI (also supposed block size 64) applies the filter to a sawtooth wave and allows changes to *a, b, c, d* and *L*. As the system tends to be unstable, a limitation by a sigmoid function within the feedback path is applied. It can be chosen from the methods `tanh`, `softclip` and `distort` with `limitType`.

```
SynthDef(\df, { |out, freq = 50, a = -0.4, b = 0.6,
    c = 0.2, d = 0.75, l = 32, limitType = 1,
    amp = 0.2|
    var src, sig;
    src = Saw.ar(freq);
    sig = Fb1.ar({ |in, out|
        var x = (a * out[1]) + (b * out[2]) +
            (d * Select.kr(l, out).squared) +
            in[0] - c;
        Select.kr(limitType, [
            x.tanh, x.softclip, x.distort
        ])
    }, in: src, outDepth: 50) * amp;
    Out.ar(0, LPF.ar(sig, 15000) ! 2)
}, metadata: (specs: (
    freq: [20, 7000, \exp, 0, 50],
    a: [-1, 1, \lin, 0, -0.4],
    b: [-1, 1, \lin, 0, 0.6],
    c: [0, 1, \lin, 0, 0.2],
    d: [-1, 1, \lin, 0, 0.75],
    l: [3, 50, \lin, 1, 32],
    limitType: [0, 2, \lin, 1, 1],
    amp: [0, 0.5, \lin, 0, 0.2]
))).add;

SynthDescLib.global[\df].makeGui;
```

3.2 Arbitrary Non-linear Operations

From the viewpoint of explorative synthesis, unary and binary operators deserve special attention. Usually they are applied to time-varying signals, but in the context of feedback / feedforward, the iteration on a single-sample base is itself the essential part of producing variations in time. Good candidates are already simple operators and their combinations, also with '+' and '-', e.g. '*', '/', '**' (power), '%' (modulo), trigonometric operators etc. Another interesting option is conditional feedback / feedforward, which means to insert branching into the relation. All of these options taken together open huge vistas for research and experimentation. Several examples are included in chapter 2 of Fb1's help file.

3.3 Integration of Ordinary Differential Equations

Fb1_ODE, added to miSCellaneous_lib version 0.22, is a pseudo ugen for integrating / audifying ordinary (systems of) differential equations (ODEs) with initial values in realtime, which is based on the Fb1 single sample feedback class. It comes along with containers for ODEs and numerical solution methods (Fb1_ODEdef, Fb1_ODEintdef). Both are basically Dictionaries of Functions and provide an interface for adding new ODE systems or integration methods interactively. An Fb1_ODE – or one of its wrappers – then merges the functional composition of numeric procedure and ODE into the SynthDef graph.

This opens the possibility for immediate audio experiments with the sheer variety of models from physics, electrical engineering, population dynamics etc., preferably those with oscillatory or quasi-oscillatory solutions and/or chaotic features. Designing new ODEs from scratch can also be very interesting. Wrappers are included for well-known systems like Van der Pol, Duffing, Hopf, Mass-Spring-Damper and Lorenz, and others can be added interactively on the fly with Fb1_ODEdef. Examples with driven pendulum, two body problem, Lotka-Volterra, Hastings-Powell and extensions of exponential decay and harmonic oscillator equations are included in the Fb1_ODE and Fb1_ODEdef help files. The framework could also be used to test ODEs before considering how to write integrators as UGens in C++. The system will be described in more detail in a future paper.

4. CONCLUSIONS

Fb1 is a class for defining single sample feedback / feedforward relations in a syntax similar to definitions provided in DSP literature and independent from the chosen block size. That way linear and non-linear filters, which are not contained as UGens in SC, can easily be tested and used. Arbitrary non-linear mathematical operators, in combination with multichannel options and dynamic look-back depths open a wide vistas for audio exploration as well as the integration resp. audification of ordinary differential equations, which can be regarded as a special case of non-linearity with feedback, implemented by the class Fb1_ODE, based on Fb1.

Possible limitations and drawbacks: the convenience of direct definition of the feedback / feedforward relation comes with the price of a large number of UGens involved. Experimentation with different block sizes is recommended to find a suitable trade-off: higher block sizes cause higher SynthDef compile times and more UGens, though preserve the benefit of cheaper control rate UGens. On the other hand even thousands of UGens do not necessarily cause a high CPU load at runtime. It might therefore be useful to choose lower block sizes for experimentation (shorter SynthDef compile time) and higher block sizes for running already tested SynthDefs (less CPU usage). Chapter 4 of Fb1's help file is dedicated to various strategies for reducing CPU load. Note that SC running with a blockSize value unequal 64 requires above examples to be updated (with blockSize and, if necessary, also blockFactor arguments).

Acknowledgments

Big credit to Nathaniel Virgo for his initial idea that triggered my implementation of Fb1 [4], see also his Feedback quark with classes Fb and FbNode and the useful summary of practical feedback tips [8]. Thanks also to James Harkins for his remarks on graph order [9]. I'm very grateful to my colleagues here at IEM Graz, where audio feedback has often been the focus of discussion groups which stimulated valuable intellectual feedback: special thanks to Daniele Pozzi for mentioning the Dobson-Ffitch filter and to David Pirrò for pointing me to the symplectic integration methods [10], which are essential for numerically stable audifying of ODEs, without his help Fb1_ODE wouldn't have come into being.

5. REFERENCES

[1] S. Wilson, D. Cottle and N. Collins (Eds.). *The SuperCollider Book*. The MIT Press, Cambridge, 2008.

[2] SuperCollider main page, https://supercollider.github.io/ (accessed 2020-02-20)

[3] D. Mayer, miSCellaneous – a library of SuperCollider extensions. https://github.com/dkmayer/miscellaneous_lib (accessed 2020-02-20)

[4] Post by N. Virgo on the sc-users mailing list https://www.listarc.bham.ac.uk/lists/sc-users-2011/msg01337.html (accessed 2020-02-20)

[5] R. Holopainen, "Nonlinear Filters" in *Proceedings of the 2007 International Computer Music Conference*, Vol 1, pp. 283-286. Copenhagen, Denmark.

[6] N. Collins, "Errant Sound Synthesis" in *Proceedings of the 2008 International Computer Music Conference*, pp. 575-578. Belfast.

[7] R. Dobson and J. Ffitch, "Experiments with Non-Linear Filters" in *Proceedings of the 1996 International Computer Music Conference*, pp. 405-408. Hong Kong.

[8] Post by N. Virgo on the sc-users mailing list https://www.listarc.bham.ac.uk/lists/sc-users-2009/msg56802.html (accessed 2020-02-20)

[9] Post by J. Harkins on the sc-users mailing list https://www.listarc.bham.ac.uk/lists/sc-users-2011/msg01363.html (accessed 2020-02-20)

[10] D. Pirrò, *Composing Interactions*. Dissertation, Institute of Electronic Music and Acoustics, University of Music and Performing Arts Graz, 2017, pp.135-146. https://pirro.mur.at/media/pirro_david_composing_interactions_print.pdf (accessed 2020-02-20)