Building a synthesiser Audio Unit plug-in modelling the Synthesis Technology E340 Cloud Generator with Faust

By Rittik Wystup

Published 10th May 2018 as part of the dissertation with the title:

Des Pudels Kern:

How open source software shapes music and its associated technology

Course: Creative Music Technology

Anglia Ruskin University, Cambridge, UK

- “The elements of software, its functions and variables, are at bottom simple things; as equally, in the faith of scientists and philosophers, must be the elements and principles which make up the world.” [Derek Robinson]

Disclaimer:

Faust has changed over the years. This article was written in 2018. A few functionalities in Faust have been modified.

This project won the Faust Award 2018.

A video demonstration of the project can be found on YouTube.

Section 1

Faust is an open-source project developed at GRAME in Lyon, France, with contributions from people internationally. In essence, the software Faust provides DSP (Digital Signal Processing) functionality to C, C++, Rust, LLVM IR, JAVA, JavaScript, asm.js, and WebAssembly (wast/wasm) interpreting and compiling, whereas the language Faust is a functional programming language specifically designed for real-time audio signal processing and synthesis which allows the user to write their desired code in a text editor of their choice and compile the .dsp file (extension for Faust code) to many different targets, including Audio Unit plug-ins (.component), VST plug-ins (.vst, relating to Steinberg's "Virtual Studio Technology") and many more. Faust has features of functional programming because it allows the user to create discrete functions within the code, which then can be called back elsewhere in the code. The main function in Faust is called "process" – this function connects the Faust code to CoreAudio (on OS X).
The compilation usually involves the Faust code being compiled into a lower-level programming language (most commonly C++) and then being compiled to the desired target. These targets range from more commonly known formats such as Audio Units or Android applications to more experimental formats, not excluding ALSA GTK standalone applications and MaxMSP externals.

The software itself does not come with an interface; the code is written in any compatible text editor (I recommend Atom because Faust comes with a code-colouring package for Atom) and compiled using a command-line compiler ("Terminal" on OS X).

The reason I chose Faust as my tool for developing my plug-in is that it is the first programming language ever designed specifically for real-time audio signal processing and synthesis. Apart from Faust, audio plug-in developers use other languages (like C++ or MathLab), which do not have a syntax build for DSP. This is where Faust becomes quite powerful – algorithms that would take many lines to code in C++ can be done in a few lines in Faust (especially algorithms involving recursion). Also, Faust allows the user to compile the code into a block diagram (with help of FaustLive, which will be discussed later), which is similar to the signal chart of an analog circuit, making it fairly simple to graphically represent the signal flow. Furthermore, the signal flow is from left to right (one-dimensional), unlike other DSP-creation programs, where the signal flow can be a vectorised plain (two-dimensional). Also, everything in Faust is a signal. For example, even a mathematical function as in

```
process = (0.1+0.3-0.1)*2
```

represents a stream of 0.4, resulting in a direct current offset of 0.4. Fig.1.1 depicts the signal flow in form of a block diagram. Essentially, when developing a tool which produces or processes audio, the user will want to consider the alternating current of the speakers. Everything being a signal is closer to audio signal processing than in e.g. Max, where there are audio signals and data signals, the latter which cannot be used to modulate audio without first converting the data to audio rate.

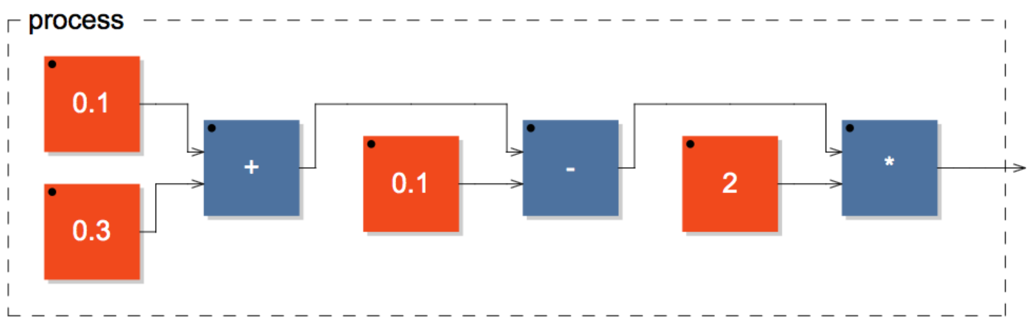


Fig.1.1: signal flow in Faust, shown with the block diagram

Faust has two branches: Faust and Faust2. Faust is the command-line compiler - the software that "converts" the Faust code to a lower-level programming language, and Faust2 is the compiler with specific targets (e.g. compiling to a Max/MSP external or a Jack application), which is the software that converts Faust code to an application-like target.

Prototyping Faust code can be done with FaustLive, a dynamic on-the-fly compiler. It allows an extremely fast edit-compile-run cycle, which makes prototyping easier than compiling to the set target, placing the compiled file to the desired directory, and testing it from possibly a host. FaustLive eliminates that process and lets the user test their code fast, which is why it is an essential tool for using Faust at its highest potential.
The Faust website offers a compiler similar to FaustLive called "Faust PlayGround". The website also provides a compiler (Faust Editor) where code can be imported and compiled to all the available targets in Faust.

The user-friendliness in Faust manifests itself in the libraries. Faust comes with an extensive library, where certain functions and operations are predefined. This means the user can utilise a predefined filter instead of building the filter from scratch. This implies Faust can be used by programmers who perhaps are unaware of how to code a third-order elliptic “Cauer” filter. The libraries also include envelopes, effects (like reverberation), analysers, oscillators, and many other practical functions. They could be compared with UGen classes in SuperCollider. I’ve included further material on the Faust libraries in the appendix (section 3).

Section 2

The “Cloud Generator” (name of the plug-in I developed) offers nine sawtooth and nine sinusoidal oscillators with a switch to toggle between the two waveforms, a spread function which controls the detuning of the individual oscillators, a sinusoidal LFO modulating the frequency of each oscillator, a cubic distortion unit, an elliptic third-order “Cauer” lowpass filter, and an envelope modulating the amplitude as well as MIDI-compatibility and polyphony.

The spread function is what makes the plug-in powerful. It involves the detuning of eight oscillators (pitched up and down) to an extent that at full spread, the two most detuned oscillators are set at half the given input frequency and double the input frequency, respectively. Many commercial plug-ins feature a detuning on multiple sawtooth oscillators (“Supersaw”), but very few offer detuning with sinusoidal oscillators. Naturally, when slightly detuning multiple sine waves, phasing quickly becomes an issue. By adding the ninth oscillator, which, no matter how much spread is applied, always stays at the centre frequency, complete wave cancellation is eliminated. Within the parallel operator (which will be discussed later) I indexed the oscillators with i. “The index, (…) [is a] one-to-many mapping that goes from a single “effect” (the input key) to the set of its multiple possible “causes” (the key’s inverted file)” (Robinson, 2006, p.107), with the input key being the number assignment of the oscillator and the inverted file being the actual oscillator and the associated spread value. Unfortunately, the spread parameter of each oscillator cannot be modulated. The frequency of each oscillator is multiplied by the spread factor (accessible via the user interface) and a fixed index (a number), which determines the amount of detuning for the specific oscillator. If the detuning index could be modulated, two separate spread functions that would have to merge into one function would need to exist: one function for tuning up and another function for tuning down. This is due to the perception of pitch to frequency being exponential. The spread function could then be:

$$S(i,t) = lin*i(t) + exp*i(t)$$

with: S = output frequency, i = iteration index, t = detuning, lin = linear function, exp = exponential function

This would allow changing the characteristics of the spread. The problem here is that the function would need to disregard its first summand for odd numbers of t and disregard its second summand for even numbers of t, which makes this version of S(i,t) inoperative.
The solution is simplifying the spread function to:

$$S(i) = V(i)^A$$

with: S = spread, V = fixed spread float, A = spread amount $[0 \leq A \leq 1]$, i = iteration index

This simple function and will output 1 for A=0 and will output V(i) for A = 1 (because of A^0=1, for $A \in R$).
Fig.1.2 shows a list of V(i) (listed in the source code as “spreadvox(i)”).

V(0) = 1;

V(1) = 1.25;

V(2) = 0.8;

V(3) = 1.5;

V(4) = 0.66;

V(5) = 1.75;

V(6) = 0.57;

$$V(7) = 2;$$
$$V(8) = 0.5;$$

with V(even): detuning by decreasing the frequency and V(odd): detuning by increasing the frequency

Fig.1.2: fixed detuning multipliers of each oscillator

Considering the carefully chosen spread multipliers, the oscillator osc(i) will go from the centre frequency f defined by the MIDI input to f*V(i), which is exactly what is needed. V(i) are values chosen so that when A = 1, all oscillators combined create a chord of nine stacked minor thirds (ranging two octaves).

Because V(i) is constant, S(i) is a simple function, resulting in it being computationally inexpensive, which is something worth considering when developing any kind of software.

Apart from the spread function, a sinusoidal LFO also modulates the frequency of each oscillator. The modulation function is also quite simple:

$$C(c,b) = c * \sin(b) * 0.01 + 1$$

with: C = modulation output, c = modulation amount, sin(b) = sinusoidal oscillator with frequency b (in radians),

This results in the final function for frequency modulation for oscillator i:

$$F(f,i,c,b) = (f * S(i)) \wedge C(c,b)$$

with: F = output frequency, f = input frequency (converted from MIDI note number), S(i) = spread value, C(b) = LFO,

or:

$$F(f,i,A,c,b) = (f * V(i) \wedge A) \wedge (c * \sin(b) * 0.01 + 1)$$

with: F = output frequency, f = input frequency, i = iteration index, A = spread amount, c = LFO modulation amount, b = LFO frequency (in radians), V(i) = spread multiplier

This function needs to be called within the code as the main function for the synthesis frequency modulation.
This function is called back within the parallel operator

```
par(i, osc(F(f,i,c,b)), 9)
```

osc(h) = oscillator function; h = oscillator input frequency

which splits the incoming signal f into (in this case) nine parallel streams, each indexed with i (so, oscillator 1 is allocated index i=0, oscillator 2 is allocated index i=1, and so on). And since the allocation index works everywhere within the parallel operator, it can be used to set V(i) for every single oscillator, setting them in parallel but with slightly different frequencies.
Of course, it would be possible to write out every single oscillator as a line of code, but using the parallel operator arranges the code in a clearer fashion.

Next in the signal chain is a simple cubic distortion followed by an elliptic third-order filter. The inspiration for adding distortion comes from the classic Moog 4-pole ladder filter where the input is driven by 10dB. For that reason, I decided to variably drive the output signal of my synthesis function and route that signal into an elliptic filter ("(…) the elliptic filter (…) has ripple in both the passband and the stopband. Elliptic filters provide the fastest roll-off for a given number of poles (…)" (Smith, 1997, p.334); they are a combination of Chebyshev Type I and a Chebyshev Type II filters) which is available in the Faust libraries. The library "filters.lib" contains many filters unknown to the standard plug-in user, and the elliptic Cauer filter has a unique characteristic.
A cubic distortion simply takes in input, sets the output to be within 1 and -1, and smooths out the clipping through a cubic function:

```
process = x – x*x*x/3
```

(normalises the signal to -2/3 and 2/3).

Fig.1.3 depicts the cubic distortion function, with the x-axis representing the input amplitude and the y-axis representing the output amplitude after the signal is clipped.

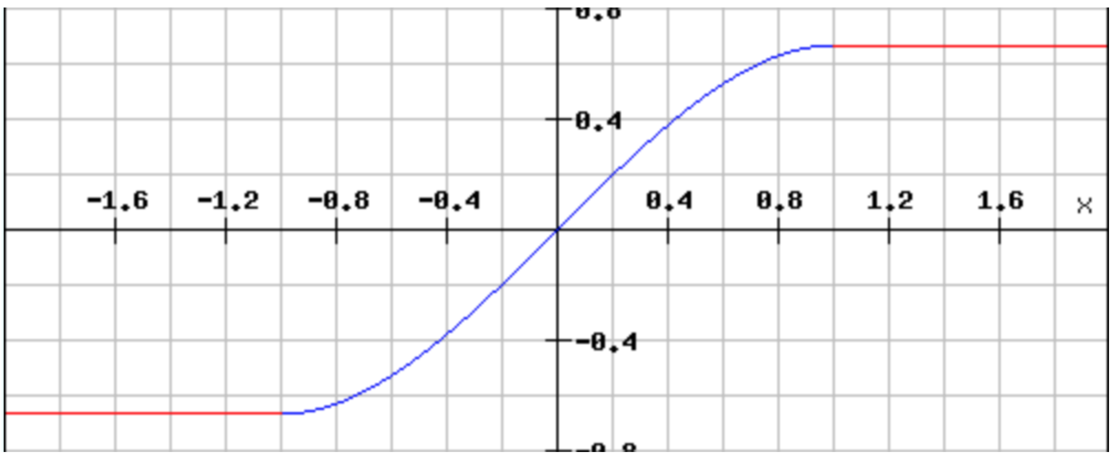
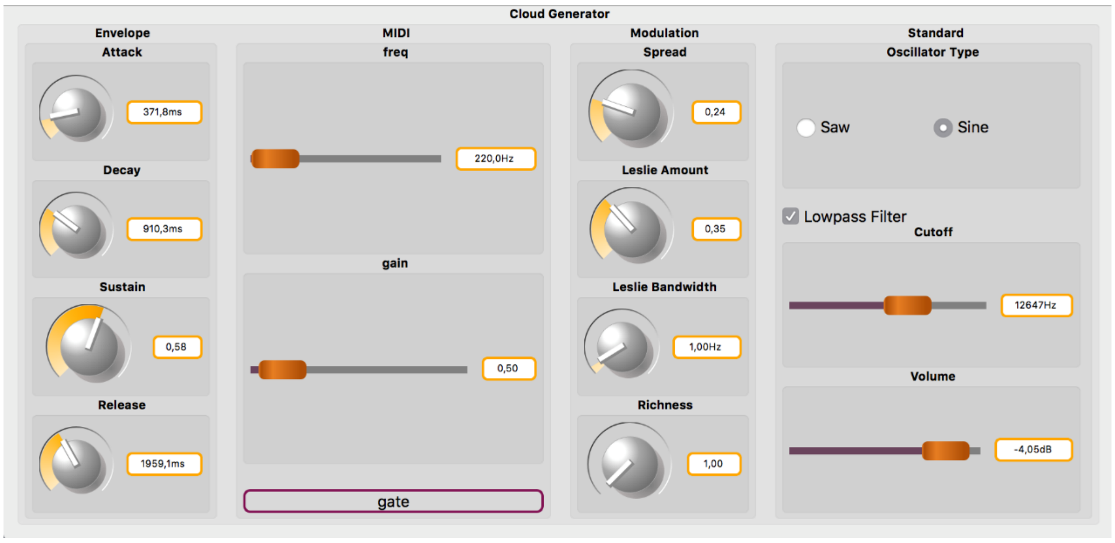


Fig.1.3: Cubic distortion

Unfortunately, by clipping the signal (especially at high frequencies), aliasing cannot be prevented, also because the clipping is applied prior to the smoothing. Upsampling the distortion unit and downsampling at the end of the “process” function could minimize aliasing.
The combination of the driven input and the “unmusical” characteristic of the filter is what gives the plug-in its unique sound.
I have included an unmodified audio example of the Cloud Generator showcasing its features.

Enabling MIDI support in Faust is fairly simple: the code needs to include UI (user interface) elements named “freq”, “gain”, and “gate”, for setting the note number, the velocity, and triggering the envelope, respectively. The UI elements need to control the input frequency of the oscillators, a gain function, and the triggering of the envelope. More information on MIDI can be found in the appendix (section 4).

Creating a user-friendly interface is important to making the plug-in more accessible. Unfortunately, the VST compiler in Faust does not create a user interface; instead, the host of the VST uses its built-in graphical display function. The design for the UI displayed in FaustLive is depicted in Fig.1.4.


Cloud Generator

Envelope
Attack 371,8ms
Decay 910,3ms
Sustain 0,58
Release 1959,1ms

MIDI
freq 220,0Hz
gain 0,50
gate

Modulation
Spread 0,24
Leslie Amount 0,35
Leslie Bandwidth 1,00Hz
Richness 1,00

Standard
Oscillator Type: Saw [✓ Sine]
[✓] Lowpass Filter
Cutoff 12647Hz
Volume -4,05dB

Fig.1.4: User interface of the Cloud Generator in FaustLive

The plug-in also contains tooltip metadata, but since the VST version does not exhibit an interface, they are not visible. For example, “Leslie” could be anything, but through hover metadata, it is possible to explain the parameter without having set a compact outline in the title.

Compilation to the targets VSTi and AU can only be possible if all requirements are met (Steinberg VST SDK 2.4 and 3 in the right directory and Xcode 8.0 or later installed; see section 2 of the appendix for more information). Through a simple line in the Terminal from the directory containing the file, the commands

```
faust2au cloudgenerator.dsp
```

or

```
faust2vsti cloudgenerator.dsp
```

generate the specified target (“faust2vsti” creates a VST instrument whereas “faust2vst” creates a VST audio effect). The compiler places the files into the folder where the source code (in this case, “cloudgenerator.dsp”) is located.
After copying the file into the audio plug-in directory, it is possible to use the tool within a plug-in host.

Unfortunately, the compiled Audio Unit does not appear in the plug-in folder within the DAW, so the distribution package will only contain the VST. This is because the architecture of “faust2au” is not maintained anymore. A solution on how to overcome this problem is outlined in the appendix (section 5).

Section 3

Because Faust is specifically designed for real-time audio signal processing, the development with Faust programming language was more successful than with Max (I developed a similar signal flow with Max prior to this project), which contained a much more complex patch with multiple subpatches within subpatches and a computationally expensive output. Furthermore, Faust is open source and free, whereas Max is not (although it would be possible to use Pd, the “free” version of Max). I see Faust as the future in DSP development, and a couple of commercial companies already use Faust for the audio section of their product development (“moForte” used Faust for the synthesis and audio processing of their mobile app “GeoShred 2”, with an interface built by “Wizdom Music”). Also, the C++ code compiled by Faust is usually more efficient than a handwritten C++ code, even code written by seasoned programmers. Faust can be used to generate efficient C++ code, and algorithms which either are too complex or even impossible to do in Faust can then be configured in the C++ code, which makes Faust a very powerful starting tool for DSP development.

References:

Robinson, D., 2006. Function. In: M. Fuller, ed. 2008. *Software Studies: A Lexicon*. Cambridge, Massachusetts: The MIT Press. Ch.14

Smith, W. S., 1997. The Scientist The Scientist and Engineer's Guide to Digital Signal Processing. s.l.: California Technical Pub.

Appendix

1. Faust Installation (some helpful tips)

- Make sure the most up-to-date version of Faust is installed (use the github link from the Faust website)
- Faust updated their library system in 2016, so if you do not want to use the current version, take that into consideration

- Faust is also available through MacPorts, but if you are a beginner, I recommend using the version of the Faust website
- Make sure the latest FaustLive version is installed as well – if not, the communication between Faust and FaustLive can be faulty
- If you do not want to use FaustLive, try out Faust PlayGround (available at: https://faust.grame.fr/faustplayground), which is also a great way of learning about Faust
- If you have multiple versions of Faust installed, remember that the source code installation has a different "uninstall" command than the version from MacPorts ([sudo make uninstall] for source code, [sudo port make uninstall] for MacPorts)
- I invite you to read Julius O. Smith's paper "Audio Signal Processing in Faust" (see bibliography for details) before getting started with Faust; bear in mind that the link for the Faust installation within the paper is outdated
- Information on compiler options can be found by typing

```
faust –-help
```

in the Terminal

If two different versions of Faust are installed (e.g. 0.9.95 and 2.5.10), some of the architecture files of one version are active and looking for libraries within both versions. If an outdated library is declared in the code, the up-to-date architecture will search for the library but will not be able to access it because of the configuration of the architecture (which is not designed to use libraries from an unsupported version). This leads to failure in compiling the code; Terminal will return an error message similar to:

```
ERROR: unable to find "oscillators.lib"
```

If two versions of Faust are installed, just type

```
faust –v
```

in the Terminal and see what the compiler returns. Terminal will not display two versions, but it will return the active version (if it is a version prior to 2016 it could lead to compatibility issues).

If an old version of Faust (prior to 2016, which uses libraries that are partly unsupported as of 2018) is used with a recent version of FaustLive, prototyping can fail with an error message in the post window of FaustLive similar to this:

```
ERROR: unable to open file “oscillators.lib”
```

This ties back to the architecture of FaustLive and Faust looking for the library system within their build.

2. Faust2 dependencies

- The “README.md” file gives some information on the dependencies of Faust2
- Install LLVM/CLang and CMake; the dependencies for these are trivial (you do not need to install Python)
- If you want to compile to a VST, download the VST/SDK package from the Steinberg website and place it into the correct directory
(/usr/local/include); if that directory does not exist, create it
- If you want to use any target that includes a QT interface, download and install QT
- If you do not wish to install dependencies, use the online Faust Editor to compile (available at: http://faust.grame.fr/editor)

3. Faust syntax: functions

- When creating a function, always call either all the arguments or none, even when it is nested, otherwise, the compiler will neglect the calculation
- Arguments need to be called in the correct order, otherwise, the calculation will swap the two values
- The last argument of any function is by default its input

Faust syntax: libraries

- Import the libraries at the beginning of the code:

```
import(“filters.lib”);
```

• Although, it makes more sense to import the entire library by importing the standard Faust library:

```
import("stdfaust.lib");
```

• "stdfaust.lib" acts as the root of the library tree and branches out to different libraries
• Using "stdfaust.lib" allows you to mark imported functions with a two-letter prefix
• This, for example, changes a sine wave oscillator from

```
osc(440);
```

to

```
os.osc(440);
```

• Through this method, the code becomes clearer, and it is also the recommended procedure in the quick reference.

4. MIDI in Faust

• The Faust website is very unclear on MIDI support; using "gate", "gain", and "freq" as name attributes in the corresponding UI elements and enabling MIDI at compilation is usually the easiest way to enable MIDI support
• The "midiTester" in FaustLive is not very reliable and it does not offer a "post window" like the "Console" in Max
• Use the "midiin" and "midiout" objects in Max if you want to test MIDI

5. Faust: targets

• A full list of targets is available through the Terminal by typing faust2

from the root of the distribution and pressing the TAB key

- This only provides a list of the commands and not the target names
- A full list of target commands, target names, and dependencies for the targets does not exist
- The target “faust2au” (compiling to an Audio Unit) does not work; the person responsible for maintaining the architecture does not do so anymore
- If you want to compile to an Audio Unit, use faust2juce and use the compiled file within JUCE to compile to an Audio Unit

6.
When using the Cloud Generator with sinusoidal oscillation and minimal spread (as well as preferably long release times) polyphonically, simple Schroeder reverberation appears. Though not originally planned when coding, the reason for this effect taking place is obvious. Schroeder reverberation emulates a “perfect” room with an arbitrary amount of parallel walls, using comb filters as reflectors (a common approach in physical modelling since a comb filter is essentially a delay). Feedback comb filters are recursive algorithm where the signal splits into two signals (for the sake of explanation, signal A and signal B). A just passes through the circuit, whereas B is added back into signal A creating a feedback loop, but with a delay of one sample or more. The amount of recursion sets the cutoff frequency and the delay time the number of combs. In the case of the Cloud Generator, feedforward comb filtering emulates Schroeder reverberation, which is why the reverberation effect is not as prominent as with feedback comb filtering. More information on digital filtering can be found in the Faust 2015 workshop, Julius O. Smith’s book “Introduction to Digital Filters: With Audio Applications”, and Perry R. Cook’s book “Real Sound Synthesis for Interactive Applications” (see the bibliography for details).

The source code and the Audio Unit are downloadable.