# The Cloud Generator: documentation on the synthesiser

by Visstup (Rittik Wystup)

Content:

Additional information ...1
Section 1: About the synthesiser ...2
Section 2: Some helpful tips on Faust ...8
References ...13
Recommended reading list ...13

*Note: this documentation is an excerpt from a dissertation I wrote, which I will soon publish, which also includes more discussion on Faust as an open-source software.*
*The title of the dissertation: "Des Pudels Kern: How open source software shapes music and its associated technology".*
*Course: Creative Music Technology (BA[Hons]) at Anglia Ruskin University, Cambridge, UK*

Mathematical functions are presented in "Times New Roman"

$$F(h,s) = h / 3^s + 0.34$$

and writings of code are presented in "Menlo"

```
process = + ~ <: _,_' :> _@(19);
```

If you find any issues feel free to contact me:
rittik.wystup@gmail.com
github.com/visstup
visstup.wordpress.com
soundcloud.com/wystup

*This documentation is aimed at Faust beginners. The README file provides a shorter documentation and installation instructions.*

# Section 1

The “Cloud Generator” 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 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*: 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, increases the gain, 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.2 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.

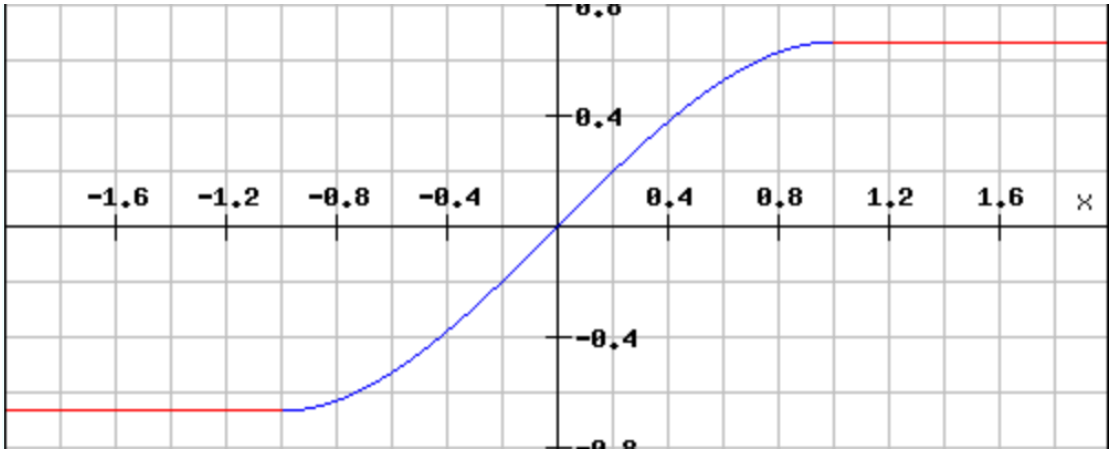


*Fig.2*: Cubic distortion

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.

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.3.

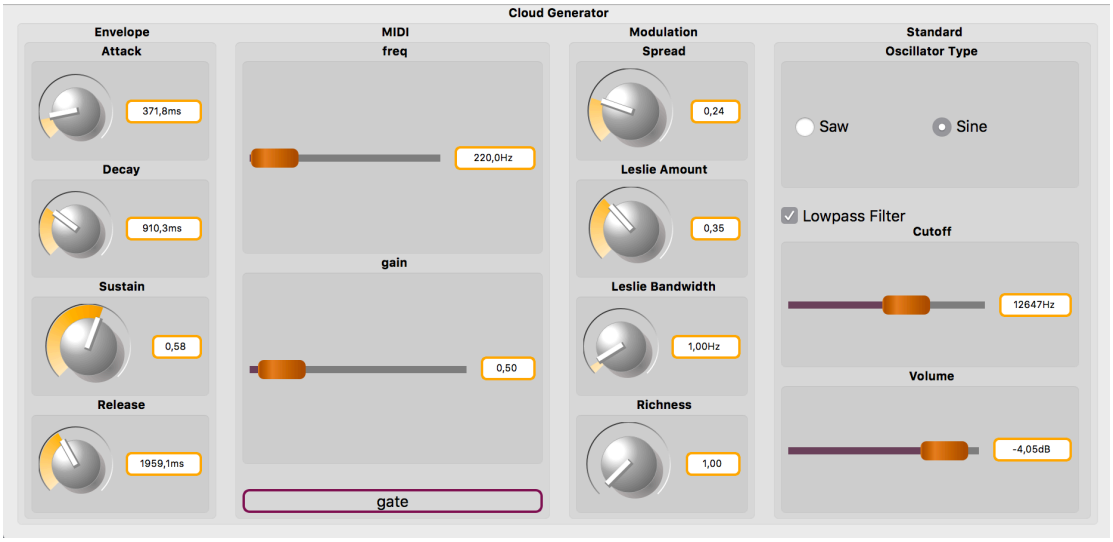


*Fig.3*: User interface of the Cloud Generator in FaustLive

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!
The architecture "faust2au" is unfortunately no longer supported. Information on how to overcome this problem is outlined in section 2 (part 5).

In the package there is a file named "Fraction.wav". This is the premaster of a track I composed, produced, and mixed, acting as a demonstration of the Cloud Generator. The track consists of percussion, vocals, and pitched synthesis. All of the pitched synthesis is from the Cloud Generator, and most of it is only slightly edited. The track will be available on my upcoming EP "Harmonic Adder", out on Primitive Music in late 2018 (vinyl and digital, catalogue #PRIMUS001). Visit facebook.com/primitivemusicofficial for more details.

## Section 2

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 the recommended reading list 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

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 recommended reading list for details).*

## 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.

## Recommended reading list:

Brice, R., 2001. *Music Engineering*. 2nd ed. Oxford: Newnes.

Bolton, W., 2002. *Control Systems*. Oxford: Newnes.

Collins, N., 2010. *Introduction to Computer Music*. Chichester: John Wiley & Sons Ltd.

Cook, P. R., 2002. *Real Sound Synthesis for Interactive Applications*. Natick: A K Peters, Ltd.

Cooper, M., 2015. Plug-ins for Nothing. *Electronic Musician*, 31(12), pp.58–60,62,64.

D'Angelo, S. and Valimaki, V., 2013. An improved virtual analog model of the Moog ladder filter. *Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference*, pp.729–733.

Davis, D., Patronis E. Jr. and Brown, P., 2013. *Sound System Engineering*. 4th ed. Burlington: Focal Press.

Fuller, M., 2008. *Software Studies: A Lexicon*. Cambridge, Massachusetts: The MIT Press.

Gaudrain, E. and Orlarey, Y., 2003. A Faust Tutorial. *GRAME* [online]. Available at: <http://www.grame.fr/ressources/publications/faust_tutorial.pdf> [Accessed 4 March 2018].

Hofmann, D., 2013. Creating Audio Unit Plugins with SuperCollider. *Dave's Blog*, [online] 14 July. Available at: <http://www.davehofmann.de/?p=118> [Accessed 24 November 2017].

Kates, J. M. and Arehart, K. H., 2005. Multichannel Dynamic-Range Compression Using Digital Frequency Warping. *EURASIP Journal on Applied Signal Processing*, 2005(18), pp.1-12.

Kleimola, J. and Välimäki, V., 2012. Reducing Aliasing from Synthetic Audio Signals using Polynomial Transition Regions. *IEEE Signal Processing Letters*, 19(2), pp.67-70.

Langford, S., 2014. *Digital Audio Editing: Correcting and Enhancing Audio with DAWs*. Burlington: Focal Press.

Mano, M. M. and Kime, C. R., 2008. *Logic and Computer Design Fundamentals*. 4th ed. Upper Saddle River: Pearson Education, Inc.

Michon, R., 2016. Faust Workshop 2016: Audio Plug-ins Designed with Faust. *Romain Michon @ CCRMA*, [online]. Available at: <https://ccrma.stanford.edu/~rmichon/faustWorkshops/2016/> [Accessed 14 January 2018].

Michon, R., 2015. Faust Workshop 2015: Online Course. *Romain Michon @ CCRMA*, [online]. Available at: <https://ccrma.stanford.edu/~rmichon/faustWorkshops/course2015/> [Accessed 10 January – 16 April 2018].

Michon, R., n.d. Faust Tutorials. *Romain Michon @ CCRMA*, [online]. Available at: <https://ccrma.stanford.edu/~rmichon/faustTutorials/> [Accessed 3 February 2018].

Millward, S., 2002. *Sound Synthesis with VST Instruments*. Kent: PC Publishing.

Pirkle, W. C., 2012. *Designing Audio Effect Plug-Ins in C++: With Digital Audio Signal Processing Theory*. Waltham: Focal Press.

Rohrhuber, J., Hall, T. and de Campo, A., n.d. Dialects, Constraints, and Systems within Systems. In: S. Wilson, D. Cottle and N. Collins, ed. 2011. *The SuperCollider Book*. Massachusetts: Massachusetts Institute of Technology.

Smith, J.O. III, 2007. *Introduction to Digital Filters: With Audio Applications*. Stanford: Center for Computer Research in Music and Acoustics (CCRMA).

Smith, O.J. III and Wang, A., 1997. On Fast FIR Filters Implemented as Tail-Canceling Filters. *IEEE Transactions on Signal Processing*, 45(6), pp.1415-1427.

Smith, J. O. III, n.d. Audio Signal Processing in Faust. *Center for Computer Research in Music and Acoustics (CCRMA),* [online]. Available at: <https://ccrma.stanford.edu/~jos/aspf/aspf.pdf> [Accessed 20 December 2017].

Smith, J. O. III and Berdahl, E., n.d. An Introduction to the Synth-A-Modeler Compiler: Modular and Open-Source Sound Synthesis using Physical Models. *Linuxaudio*, [online]. Available at: http://lac.linuxaudio.org/2012/papers/34.pdf [Accessed 20 April 2018].

Stowell, D., n.d. Writing Unit Generator Plug-ins. In: S. Wilson, D. Cottle and N. Collins, ed. 2011. *The SuperCollider Book*. Massachusetts: Massachusetts Institute of Technology.

Walker, J. S. and Don, G. W., 2013. *Mathematics and Music*. Boca Raton: CRC Press.