Contents

Faust Libraries

NOTE: this documentation was automatically generated using the script `generateDoc`. This script depends on pandoc and html-xml-utils.

This page provides information on how to use the Faust libraries.

The `/libraries` folder contains the different Faust libraries. If you wish to add your own functions to this library collection, you can refer to the "Contributing" section providing a set of coding conventions.

WARNING: These libraries replace the "old" Faust libraries. They are still being beta tested so you might encounter bugs while using them. If your codes still use the "old" Faust libraries, you might want to try to use Bart Brouns' script that automatically makes an old Faust code compatible with the new libraries: https://github.com/magnetophon/faustCompressors/blob/master/newlib.sh. If

you find a bug, please report it at rmichon_at_ccrma_dot_stanford_dot_edu. Thanks ;)!

Using the Faust Libraries

The easiest and most standard way to use the Faust libraries is to import `stdfaust.lib` in your Faust code:

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

This will give you access to all the Faust libraries through a series of environments:

- `sf`: `all.lib`
- `an`: `analyzers.lib`
- `ba`: `basics.lib`
- `co`: `compressors.lib`
- `de`: `delays.lib`
- `dm`: `demos.lib`
- `dx`: `dx7.lib`
- `en`: `envelopes.lib`
- `fi`: `filters.lib`
- `ho`: `hoa.lib`
- `it`: `interpolators.lib`
- `ma`: `maths.lib`
- `ef`: `misceffects.lib`
- `os`: `oscillators.lib`
- `no`: `noises.lib`
- `pf`: `phaflangers.lib`
- `pm`: `physmodels.lib`
- `rm`: `reducemaps.lib`
- `re`: `reverbs.lib`
- `ro`: `routes.lib`
- `si`: `signals.lib`
- `so`: `soundfiles.lib`
- `sp`: `spats.lib`
- `sy`: `synths.lib`
- `ve`: `vaeffects.lib`
- `wa`: `webaudio.lib`
- `vl`: `version.lib`

Environments can then be used as follows in your Faust code:

```
import("stdfaust.lib");
process = os.osc(440);
```

In this case, we're calling the `osc` function from `oscillators.lib`.

You can also access all the functions of all the libraries directly using the `sf` environment:

```
import("stdfaust.lib");
process = sf.osc(440);
```

Alternatively, environments can be created by hand:

```
os = library("oscillators.lib");
process = os.osc(440);
```

Finally, libraries can be simply imported in the Faust code (not recommended):

```
import("oscillators.lib");
process = osc(440);
```

Contributing

If you wish to add a function to any of these libraries or if you plan to add a new library, make sure that you follow the following conventions:

New Functions

- All functions must be preceded by a markdown documentation header respecting the following format (open the source code of any of the libraries for an example):

```
//----------------functionName--------------------
// Description
//
// #### Usage
//
// ```
// Usage Example
// ```
//
// Where:
//
// * argument1: argument 1 description
//------------------------------------------------
```

- Every time a new function is added, the documentation should be updated simply by running `make doclib`.
- The environment system (e.g. `os.osc`) should be used when calling a function declared in another library (see the section on *Using the Faust Libraries*).
- Try to reuse existing functions as much as possible.
- If you have any question, send an e-mail to rmichon_at_ccrma_dot_stanford_dot_edu.

New Libraries

- Any new "standard" library should be declared in `stdfaust.lib` with its own environment (2 letters - see `stdfaust.lib`).
- Any new "standard" library must be added to `generateDoc`.
- Functions must be organized by sections.
- Any new library should at least `declare` a `name` and a `version`.
- The comment based markdown documentation of each library must respect the following format (open the source code of any of the libraries for an example):

```
//############### libraryName ##################
// Description
//
// * Section Name 1
// * Section Name 2
// * ...
//
// It should be used using the `[...]` environment:
//
// ```
// [...] = library("libraryName");
// process = [...].functionCall;
// ```
//
// Another option is to import `stdfaust.lib` which already contains the `[...]`
// environment:
//
// ```
// import("stdfaust.lib");
// process = [...].functionCall;
// ```
//############################################

//================= Section Name ===============
// Description
//==============================================
```

- If you have any question, send an e-mail to rmichon_at_ccrma_dot_stanford_dot_edu.

General Organization

Only the libraries that are considered to be "standard" are documented:

- `analyzers.lib`
- `basics.lib`
- `compressors.lib`
- `delays.lib`

- demos.lib
- dx7.lib
- envelopes.lib
- filters.lib
- hoa.lib
- interpolators.lib
- maths.lib
- misceffects.lib
- oscillators.lib
- noises.lib
- phaflangers.lib
- physmodels.lib
- reducemaps.lib
- reverbs.lib
- routes.lib
- signals.lib
- soundfiles.lib
- spats.lib
- synths.lib
- tonestacks.lib (not documented but example in /examples/misc)
- tubes.lib (not documented but example in /examples/misc)
- vaeffects.lib
- webaudio.lib
- version.lib

Other deprecated libraries such as music.lib, etc. are present but are not documented to not confuse new users.

The documentation of each library can be found in /documentation/library.html or in /documentation/library.pdf.

A global version number for the standard libraries is defined in version.lib. It follows the semantic versioning structure: MAJOR, MINOR, PATCH. The MAJOR number is increased when we make incompatible changes. The MINOR number is increased when we add functionality in a backwards compatible manner, and the PATCH number when we make backwards compatible bug fixes. By looking at the generated code or the diagram of process = vl.version; one can see the current version of the libraries.

The /examples directory contains all the examples from the /examples folder of the Faust distribution as well as new ones. Most of them were updated to reflect the coding conventions described in the next section. Examples are organized by types in different folders. The /old folder contains examples that are fully deprecated, probably because they were integrated to the libraries and fully rewritten (see freeverb.dsp for example). Examples using deprecated libraries were integrated to the general tree but a warning comment was added at their beginning to point readers to the right library and function.

Coding Conventions

In order to have a uniformized library system, we established the following conventions (that hopefully will be followed by others when making modifications to them :-)).

Documentation

- All the functions that we want to be "public" are documented.
- We used the `faust2md` "standards" for each library: `//###` for main title (library name - equivalent to `#` in markdown), `//===` for section declarations (equivalent to `##` in markdown) and `//---` for function declarations (equivalent to `####` in markdown - see `basics.lib` for an example).
- Sections in function documentation should be declared as `####` markdown title.
- Each function documentation provides a "Usage" section (see `basics.lib`).

Library Import

To prevent cross-references between libraries we generalized the use of the `library("")` system for function calls in all the libraries. This means that everytime a function declared in another library is called, the environment corresponding to this library needs to be called too. To make things easier, a `stdfaust.lib` library was created and is imported by all the libraries:

```
an = library("analyzers.lib");
ba = library("basics.lib");
co = library("compressors.lib");
de = library("delays.lib");
dm = library("demos.lib");
dx = library("dx7.lib");
en = library("envelopes.lib");
fi = library("filters.lib");
ho = library("hoa.lib");
it = library("interpolators.lib");
ma = library("maths.lib");
ef = library("misceffects.lib");
os = library("oscillators.lib");
no = library("noises.lib");
pf = library("phaflangers.lib");
pm = library("physmodels.lib");
rm = library("reducemaps.lib");
re = library("reverbs.lib");
ro = library("routes.lib");
sp = library("spats.lib");
si = library("signals.lib");
so = library("soundfiles.lib");
```

```
sy = library("synths.lib");
ve = library("vaeffects.lib");
wa = library("webaudio.lib");
vl = library("version.lib");
```

For example, if we wanted to use the `smooth` function which is now declared in `signals.lib`, we would do the following:

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

process = si.smooth(0.999);
```

This standard is only used within the libraries: nothing prevents coders to still import `signals.lib` directly and call `smooth` without `ro.`, etc. It means symbols and function names defined within a library have to be unique to not collide with symbols of any other libraries.

"Demo" Functions

"Demo" functions are placed in `demos.lib` and have a built-in user interface (UI). Their name ends with the `_demo` suffix. Each of these function have a `.dsp` file associated to them in the `/examples` folder.

Any function containing UI elements should be placed in this library and respect these standards.

"Standard" Functions

"Standard" functions are here to simplify the life of new (or not so new) Faust coders. They are declared in `/libraries/doc/standardFunctions.md` and allow to point programmers to preferred functions to carry out a specific task. For example, there are many different types of lowpass filters declared in `filters.lib` and only one of them is considered to be standard, etc.

Copyright / License

Now that Faust libraries are less author specific, each function will normally have its own copyright-and-license line in the library source (the `.lib` file, such as `analyzers.lib`). If not, see if the function is defined within a section of the `.lib` file stating the license in source-code comments. If not, then the copyright and license given at the beginning of the `.lib` file may be assumed, when present. If not, run `git blame` on the `.lib` file and ask the person who last edited the function!

Note that it is presently possible for a library function released under one license to utilize another library function having some different license. There is presently no indication of this situation in the Faust compiler output, but such notice is planned. For now, library contributors should strive to use only library

functions having compatible licenses, and concerned end-users must manually determine the union of licenses applicable to the library functions they are using.

Standard Functions

Dozens of functions are implemented in the Faust libraries and many of them are very specialized and not useful to beginners or to people who only need to use Faust for basic applications. This section offers an index organized by categories of the "standard Faust functions" (basic filters, effects, synthesizers, etc.). This index only contains functions without a user interface (UI). Faust functions with a built-in UI can be found in `demos.lib`.

Analysis Tools

Function Type	Function Name	Description
Amplitude Follower	`an.amp_follower`	Classic analog audio envelope follower
Octave Analyzers	`an.mth_octave_analyzer[N]`	Octave analyzers

Basic Elements

Function Type	Function Name	Description
Beats	`ba.beat`	Pulses at a specific tempo
Block	`si.block`	Terminate n signals
Break Point Function	`ba.bpf`	Beak Point Function (BPF)
Bus	`si.bus`	Bus of n signals
Bypass (Mono)	`ba.bypass1`	Mono bypass
Bypass (Stereo)	`ba.bypass2`	Stereo bypass
Count Elements	`ba.count`	Count elements in a list
Count Down	`ba.countdown`	Samples count down
Count Up	`ba.countup`	Samples count up
Delay (Integer)	`de.delay`	Integer delay
Delay (Float)	`de.fdelay`	Fractional delay
Down Sample	`ba.downSample`	Down sample a signal
Impulsify	`ba.impulsify`	Turns a signal into an impulse
Sample and Hold	`ba.sAndH`	Sample and hold
Signal Crossing	`ro.cross`	Cross n signals
Smoother (Default)	`si.smoo`	Exponential smoothing
Smoother	`si.smooth`	Exponential smoothing with controllable pole
Take Element	`ba.take`	Take en element from a list
Time	`ba.time`	A simple timer

Conversion

Function Type	Function Name	Description
dB to Linear	`ba.db2linear`	Converts dB to linear values
Linear to dB	`ba.linear2db`	Converts linear values to dB
MIDI Key to Hz	`ba.midikey2hz`	Converts a MIDI key number into a frequency
Hz to MIDI Key	`ba.hz2midikey`	Converts a frequency into MIDI key number
Pole to T60	`ba.pole2tau`	Converts a pole into a time constant (t60)
Samples to Seconds	`ba.samp2sec`	Converts samples to seconds
Seconds to Samples	`ba.sec2samp`	Converts seconds to samples
T60 to Pole	`ba.tau2pole`	Converts a time constant (t60) into a pole

Effects

Function Type	Function Name	Description
Auto Wah	`ve.autowah`	Auto-Wah effect
Compressor	`co.compressor_mono`	Dynamic range compressor
Distortion	`ef.cubicnl`	Cubic nonlinearity distortion
Crybaby	`ve.crybaby`	Crybaby wah pedal
Echo	`ef.echo`	Simple echo
Flanger	`pf.flanger_stereo`	Flanging effect
Gate	`ef.gate_mono`	Mono signal gate
Limiter	`co.limiter_1176_R4_mono`	Limiter
Phaser	`pf.phaser2_stereo`	Phaser effect
Reverb (FDN)	`re.fdnrev0`	Feedback delay network reverberator
Reverb (Freeverb)	`re.mono_freeverb`	Most “famous” Schroeder reverberator
Reverb (Simple)	`re.jcrev`	Simple Schroeder reverberator
Reverb (Zita)	`re.zita_rev1_stereo`	High quality FDN reverberator
Panner	`sp.panner`	Linear stereo panner
Pitch Shift	`ef.transpose`	Simple pitch shifter
Panner	`sp.spat`	N outputs spatializer
Speaker Simulator	`ef.speakerbp`	Simple speaker simulator
Stereo Width	`ef.stereo_width`	Stereo width effect
Vocoder	`ve.vocoder`	Simple vocoder
Wah	`ve.wah4`	Wah effect

Envelope Generators

Function Type	Function Name	Description
ADSR	`en.adsr`	Attack/Decay/Sustain/Release envelope generator
AR	`en.ar`	Attack/Release envelope generator
ASR	`en.asr`	Attack/Sustain/Release envelope generator

Function Type	Function Name	Description
Exponential	`en.smoothEnvelope`	Exponential envelope generator

Filters

Function Type	Function Name	Description
Bandpass (Butterworth)	`fi.bandpass`	Generic butterworth bandpass
Bandpass (Resonant)	`fi.resonbp`	Virtual analog resonant bandpass
Bandstop (Butterworth)	`fi.bandstop`	Generic butterworth bandstop
Biquad	`fi.tf2`	"Standard" biquad filter
Comb (Allpass)	`fi.allpass_fcomb`	Schroeder allpass comb filter
Comb (Feedback)	`fi.fb_fcomb`	Feedback comb filter
Comb (Feedforward)	`fi.ff_fcomb`	Feed-forward comb filter.
DC Blocker	`fi.dcblocker`	Default dc blocker
Filterbank	`fi.filterbank`	Generic filter bank
FIR (Arbitrary Order)	`fi.fir`	Nth-order FIR filter
High Shelf	`fi.high_shelf`	High shelf
Highpass (Butterworth)	`fi.highpass`	Nth-order Butterworth highpass
Highpass (Resonant)	`fi.resonhp`	Virtual analog resonant highpass
IIR (Arbitrary Order)	`fi.iir`	Nth-order IIR filter
Level Filter	`fi.levelfilter`	Dynamic level lowpass
Low Shelf	`fi.low_shelf`	Low shelf
Lowpass (Butterworth)	`fi.lowpass`	Nth-order Butterworth lowpass
Lowpass (Resonant)	`fi.resonlp`	Virtual analog resonant lowpass
Notch Filter	`fi.notchw`	Simple notch filter
Peak Equalizer	`fi.peak_eq`	Peaking equalizer section

Oscillators/Sound Generators

Function Type	Function Name	Description
Impulse	`os.impulse`	Generate an impulse on start-up
Impulse Train	`os.imptrain`	Band-limited impulse train
Phasor	`os.phasor`	Simple phasor
Pink Noise	`no.pink_noise`	Pink noise generator
Pulse Train	`os.pulsetrain`	Band-limited pulse train
Pulse Train (Low Frequency)	`os.lf_imptrain`	Low-frequency pulse train
Sawtooth	`os.sawtooth`	Band-limited sawtooth wave
Sawtooth (Low Frequency)	`os.lf_saw`	Low-frequency sawtooth wave
Sine (Filter-Based)	`os.oscs`	Sine oscillator (filter-based)
Sine (Table-Based)	`os.osc`	Sine oscillator (table-based)
Square	`os.square`	Band-limited square wave
Square (Low Frequency)	`os.lf_squarewave`	Low-frequency square wave

Function Type	Function Name	Description
Triangle	`os.triangle`	Band-limited triangle wave
Triangle (Low Frequency)	`os.lf_triangle`	Low-frequency triangle wave
White Noise	`no.noise`	White noise generator

Synths

Function Type	Function Name	Description
Additive Drum	`sy.additiveDrum`	Additive synthesis drum
Bandpassed Sawtooth	`sy.dubDub`	Sawtooth through resonant bandpass
Comb String	`sy.combString`	String model based on a comb filter
FM	`sy.fm`	Frequency modulation synthesizer
Lowpassed Sawtooth	`sy.sawTrombone`	"Trombone" based on a filtered sawtooth
Popping Filter	`sy.popFilterPerc`	Popping filter percussion instrument

Primitives

User Interface Primitives

`button`

Creates a button in the user interface. The `button` is a primitive circuit with one output and no input. The signal produced by the `button` is 0 when not pressed and 1 while pressed.

Usage

```
button("play") : _;
```

Where `"play"` is the name of the `button` in the interface.

`checkbox`

Creates a checkbox in the user interface. The `checkbox` is a primitive circuit with one output and no input. The signal produced by the checkbox is 0 when not checked and 1 when checked.

Usage

```
checkbox("play") : _;
```

Where `"play"` is the name of the `checkbox` in the interface.

`hslider`

Creates a horizontal slider in the user interface. The `hslider` is a primitive circuit with one output and no input. `hslider` produces a signal between a minimum and a maximum value based on the position of the slider cursor.

Usage

```
hslider("volume",-10,-70,12,0.1) : _;
```

Where `volume` is the name of the slider in the interface, `-10` the default value of the slider when the program starts, `-70` the minimum value, `12` the maximum value, and `0.1` the step the determines the precision of the control.

`nentry`

Creates a numerical entry in the user interface. The `nentry` is a primitive circuit with one output and no input. `nentry` produces a signal between a minimum and a maximum value based on the user input.

Usage

```
nentry("volume",-10,-70,12,0.1) : _;
```

Where `volume` is the name of the numerical entry in the interface, `-10` the default value of the entry when the program starts, `-70` the minimum value, `12` the maximum value, and `0.1` the step the determines the precision of the control.

`vslider`

Creates a vertical slider in the user interface. The `vslider` is a primitive circuit with one output and no input. `vslider` produces a signal between a minimum and a maximum value based on the position of the slider cursor.

Usage

```
vslider("volume",-10,-70,12,0.1) : _;
```

Where `volume` is the name of the slider in the interface, `-10` the default value of the slider when the program starts, `-70` the minimum value, `12` the maximum value, and `0.1` the step the determines the precision of the control.

analyzers.lib

Faust Analyzers library. Its official prefix is `an`.

Amplitude Tracking

`(an.)amp_follower`

Classic analog audio envelope follower with infinitely fast rise and exponential decay. The amplitude envelope instantaneously follows the absolute value going up, but then floats down exponentially. `amp_follower` is a standard Faust function.

Usage

```
_ : amp_follower(rel) : _
```

Where:

- `rel`: release time = amplitude-envelope time-constant (sec) going down

Reference

- Musical Engineer's Handbook, Bernie Hutchins, Ithaca NY, 1975 Electronotes Newsletter, Bernie Hutchins

---

`(an.)amp_follower_ud`

Envelope follower with different up and down time-constants (also called a "peak detector").

Usage

```
_ : amp_follower_ud(att,rel) : _
```

Where:

- `att`: attack time = amplitude-envelope time constant (sec) going up
- `rel`: release time = amplitude-envelope time constant (sec) going down

Note

We assume rel >> att. Otherwise, consider rel ~ max(rel,att). For audio, att is normally faster (smaller) than rel (e.g., 0.001 and 0.01). Use `amp_follower_ar` below to remove this restriction.

Reference

- "Digital Dynamic Range Compressor Design — A Tutorial and Analysis", by Dimitrios Giannoulis, Michael Massberg, and Joshua D. Reiss http://www.eecs.qmul.ac.uk/~josh/documents/GiannoulisMassbergReiss-dynamicrangecompression-JAES2012.pdf

---

`(an.)amp_follower_ar`

Envelope follower with independent attack and release times. The release can be shorter than the attack (unlike in `amp_follower_ud` above).

Usage

```
_ : amp_follower_ar(att,rel) : _;
```

- Author Jonatan Liljedahl, revised by RM

Spectrum-Analyzers

Spectrum-analyzers split the input signal into a bank of parallel signals, one for each spectral band. They are related to the Mth-Octave Filter-Banks in `filters.lib`. The documentation of this library contains more details about the implementation. The parameters are:

- `M`: number of band-slices per octave (>1)
- `N`: total number of bands (>2)
- `ftop` = upper bandlimit of the Mth-octave bands (<SR/2)

In addition to the Mth-octave output signals, there is a highpass signal containing frequencies from ftop to SR/2, and a "dc band" lowpass signal containing frequencies from 0 (dc) up to the start of the Mth-octave bands. Thus, the N output signals are

```
highpass(ftop), MthOctaveBands(M,N-2,ftop), dcBand(ftop*2^(-M*(N-1)))
```

A Spectrum-Analyzer is defined here as any band-split whose bands span the relevant spectrum, but whose band-signals do not necessarily sum to the original signal, either exactly or to within an allpass filtering. Spectrum analyzer outputs are normally at least nearly "power complementary", i.e., the power spectra of the individual bands sum to the original power spectrum (to within some negligible tolerance).

Increasing Channel Isolation

Go to higher filter orders - see Regalia et al. or Vaidyanathan (cited below) regarding the construction of more aggressive recursive filter-banks using elliptic or Chebyshev prototype filters.

References

- "Tree-structured complementary filter banks using all-pass sections", Regalia et al., IEEE Trans. Circuits & Systems, CAS-34:1470-1484, Dec. 1987
- "Multirate Systems and Filter Banks", P. Vaidyanathan, Prentice-Hall, 1993

- Elementary filter theory: https://ccrma.stanford.edu/~jos/filters/

(an.)mth_octave_analyzer

Octave analyzer. `mth_octave_analyzer[N]` are standard Faust functions.

Usage

```
_ : mth_octave_analyzer(O,M,ftop,N) : par(i,N,_); // Oth-order Butterworth
_ : mth_octave_analyzer6e(M,ftop,N) : par(i,N,_); // 6th-order elliptic
```

Also for convenience:

```
_ : mth_octave_analyzer3(M,ftop,N) : par(i,N,_); // 3d-order Butterworth
_ : mth_octave_analyzer5(M,ftop,N) : par(i,N,_); // 5th-roder Butterworth
mth_octave_analyzer_default = mth_octave_analyzer6e;
```

Where:

- `O`: order of filter used to split each frequency band into two
- `M`: number of band-slices per octave
- `ftop`: highest band-split crossover frequency (e.g., 20 kHz)
- `N`: total number of bands (including dc and Nyquist)

Mth-Octave Spectral Level

Spectral Level: Display (in bar graphs) the average signal level in each spectral band.

(an.)mth_octave_spectral_level6e

Spectral level display.

Usage:

```
_ : mth_octave_spectral_level6e(M,ftop,NBands,tau,dB_offset) : _;
```

Where:

- `M`: bands per octave
- `ftop`: lower edge frequency of top band
- `NBands`: number of passbands (including highpass and dc bands),
- `tau`: spectral display averaging-time (time constant) in seconds,
- `dB_offset`: constant dB offset in all band level meters.

Also for convenience:

```
mth_octave_spectral_level_default = mth_octave_spectral_level6e;
spectral_level = mth_octave_spectral_level(2,10000,20);
```

`(an.)[third|half]_octave_[analyzer|filterbank]`

A bunch of special cases based on the different analyzer functions described above:

```
third_octave_analyzer(N) = mth_octave_analyzer_default(3,10000,N);
third_octave_filterbank(N) = mth_octave_filterbank_default(3,10000,N);
half_octave_analyzer(N) = mth_octave_analyzer_default(2,10000,N);
half_octave_filterbank(N) = mth_octave_filterbank_default(2,10000,N);
octave_filterbank(N) = mth_octave_filterbank_default(1,10000,N);
octave_analyzer(N) = mth_octave_analyzer_default(1,10000,N);
```

Usage

See `mth_octave_spectral_level_demo` in `demos.lib`.

Arbritary-Crossover Filter-Banks and Spectrum Analyzers

These are similar to the Mth-octave analyzers above, except that the band-split frequencies are passed explicitly as arguments.

`(an.)analyzer`

Analyzer.

Usage

```
_ : analyzer(O,freqs) : par(i,N,_); // No delay equalizer
```

Where:

- `O`: band-split filter order (ODD integer required for filterbank[i])
- `freqs`: (fc1,fc2,...,fcNs) [in numerically ascending order], where Ns=N-1 is the number of octave band-splits (total number of bands N=Ns+1).

If frequencies are listed explicitly as arguments, enclose them in parens:

```
_ : analyzer(3,(fc1,fc2)) : _,_,_
```

Fast Fourier Transform (fft) and its Inverse (ifft)

Sliding FFTs that compute a rectangularly windowed FFT each sample.

`(an.)gortzelOpt`

Optimized Goertzel filter.

Usage

```
_ : goertzelOpt(freq,N) : _;
```

Where:

- `freq`: frequency to be analyzed
- `N`: the Goertzel block size

Reference

- https://en.wikipedia.org/wiki/Goertzel_algorithm

`(an.)gortzelComp`

Complex Goertzel filter.

Usage

```
_ : goertzelComp(freq,N) : _;
```

Where:

- `freq`: frequency to be analyzed
- `N`: the Goertzel block size

Reference

- https://en.wikipedia.org/wiki/Goertzel_algorithm

`(an.)goertzel`

Same as `goertzelOpt`.

Usage

```
_ : goertzel(freq,N) : _;
```

Where:

- `freq`: frequency to be analyzed
- `N`: the Goertzel block size

Reference

- https://en.wikipedia.org/wiki/Goertzel_algorithm

`(an.)fft`

Fast Fourier Transform (FFT)

Usage

```
si.cbus(N) : fft(N) : si.cbus(N);
```

Where:

- `si.cbus(N)` is a bus of N complex signals, each specified by real and imaginary parts: (r0,i0), (r1,i1), (r2,i2), ...
- `N` is the FFT size (must be a power of 2: 2,4,8,16,...)
- `fft(N)` performs a length `N` FFT for complex signals (radix 2)
- The output is a bank of N complex signals containing the complex spectrum over time: (R0, I0), (R1,I1), ...
  - The dc component is (R0,I0), where I0=0 for real input signals.

FFTs of Real Signals:

- To perform a sliding FFT over a real input signal, you can say

```
process = signal : an.rtocv(N) : an.fft(N);
```

where `an.rtocv` converts a real (scalar) signal to a complex vector signal having a zero imaginary part.

- See `an.rfft_analyzer_c` (in `analyzers.lib`) and related functions for more detailed usage examples.
- Use `an.rfft_spectral_level(N,tau,dB_offset)` to display the power spectrum of a real signal.
- See `dm.fft_spectral_level_demo(N)` in `demos.lib` for an example GUI driving `an.rfft_spectral_level()`.

Reference

- Decimation-in-time (DIT) Radix-2 FFT

`(an.)ifft`

Inverse Fast Fourier Transform (IFFT).

Usage

```
si.cbus(N) : ifft(N) : si.cbus(N);
```

Where:

- N is the IFFT size (power of 2)
- Input is a complex spectrum represented as interleaved real and imaginary parts: (R0, I0), (R1,I1), (R2,I2), ...
- Output is a bank of N complex signals giving the complex signal in the time domain: (r0, i0), (r1,i1), (r2,i2), ...

---

basics.lib

A library of basic elements. Its official prefix is `ba`.

Conversion Tools

`(ba.)samp2sec`

Converts a number of samples to a duration in seconds. `samp2sec` is a standard Faust function.

Usage

```
samp2sec(n) : _
```

Where:

- `n`: number of samples

`(ba.)sec2samp`

Converts a duration in seconds to a number of samples. `samp2sec` is a standard Faust function.

Usage

```
sec2samp(d) : _
```

Where:

- `d`: duration in seconds

`(ba.)db2linear`

Converts a loudness in dB to a linear gain (0-1). `db2linear` is a standard Faust function.

Usage

```
db2linear(l) : _
```

Where:

- `l`: loudness in dB

`(ba.)linear2db`

Converts a linear gain (0-1) to a loudness in dB. `linear2db` is a standard Faust function.

Usage

```
linear2db(g) : _
```

Where:

- `g`: a linear gain

`(ba.)lin2LogGain`

Converts a linear gain (0-1) to a log gain (0-1).

Usage

```
lin2LogGain(n) : _
```

`(ba.)log2LinGain`

Converts a log gain (0-1) to a linear gain (0-1).

Usage

```
log2LinGain(n) : _
```

`(ba.)tau2pole`

Returns a real pole giving exponential decay. Note that t60 (time to decay 60 dB) is ~6.91 time constants. `tau2pole` is a standard Faust function.

Usage

```
_ : smooth(tau2pole(tau)) : _
```

Where:

- `tau`: time-constant in seconds

`(ba.)pole2tau`

Returns the time-constant, in seconds, corresponding to the given real, positive pole in (0,1). `pole2tau` is a standard Faust function.

Usage

```
pole2tau(pole) : _
```

Where:

- `pole`: the pole

`(ba.)midikey2hz`

Converts a MIDI key number to a frequency in Hz (MIDI key 69 = A440). `midikey2hz` is a standard Faust function.

Usage

```
midikey2hz(mk) : _
```

Where:

- `mk`: the MIDI key number

`(ba.)hz2midikey`

Converts a frequency in Hz to a MIDI key number (MIDI key 69 = A440). `hz2midikey` is a standard Faust function.

Usage

```
hz2midikey(f) : _
```

Where:

- `f`: frequency in Hz

`(ba.)semi2ratio`

Converts semitones in a frequency multiplicative ratio. `semi2ratio` is a standard Faust function.

Usage

```
semi2ratio(semi) : _
```

Where:

- `semi`: number of semitone

`(ba.)ratio2semi`

Converts a frequency multiplicative ratio in semitones. `ratio2semi` is a standard Faust function.

Usage

```
ratio2semi(ratio) : _
```

Where:

- `ratio`: frequency multiplicative ratio

`(ba.)pianokey2hz`

Converts a piano key number to a frequency in Hz (piano key 49 = A440).

Usage

```
pianokey2hz(pk) : _
```

Where:

- `pk`: the piano key number

`(ba.)hz2pianokey`

Converts a frequency in Hz to a piano key number (piano key 49 = A440).

Usage

```
hz2pianokey(f) : _
```

Where:

- `f`: frequency in Hz

Counters and Time/Tempo Tools

`(ba.)countdown`

Starts counting down from n included to 0. While trig is 1 the output is n. The countdown starts with the transition of trig from 1 to 0. At the end of the countdown the output value will remain at 0 until the next trig. `countdown` is a standard Faust function.

Usage

```
countdown(n,trig) : _
```

Where:

- `n`: the starting point of the countdown
- `trig`: the trigger signal (1: start at `n`; 0: decrease until 0)

`(ba.)countup`

Starts counting up from 0 to n included. While trig is 1 the output is 0. The countup starts with the transition of trig from 1 to 0. At the end of the countup the output value will remain at n until the next trig. `countup` is a standard Faust function.

Usage

```
countup(n,trig) : _
```

Where:

- `n`: the maximum count value
- `trig`: the trigger signal (1: start at 0; 0: increase until `n`)

`(ba.)sweep`

Counts from 0 to `period-1` repeatedly, generating a sawtooth waveform, like os.lf_rawsaw, starting at 1 when `run` transitions from 0 to 1. Outputs zero while `run` is 0.

Usage

```
sweep(period,run) : _
```

`(ba.)time`

A simple timer that counts every samples from the beginning of the process. `time` is a standard Faust function.

Usage

```
time : _
```

`(ba.)ramp`

An linear ramp of 'n' samples to reach the next value

Usage

```
_ : ramp(n) : _
```

Where:

- `n`: number of samples to reach the next value

`(ba.)tempo`

Converts a tempo in BPM into a number of samples.

Usage

```
tempo(t) : _
```

Where:

- `t`: tempo in BPM

`(ba.)period`

Basic sawtooth wave of period `p`.

Usage

```
period(p) : _
```

Where:

- `p`: period as a number of samples

`(ba.)pulse`

Pulses (10000) generated at period `p`.

Usage

```
pulse(p) : _
```

Where:

- `p`: period as a number of samples

`(ba.)pulsen`

Pulses (11110000) of length `n` generated at period `p`.

Usage

```
pulsen(n,p) : _
```

Where:

- `n`: pulse length as a number of samples
- `p`: period as a number of samples

`(ba.)cycle`

Split nonzero input values into `n` cycles.

Usage

```
_ : cycle(n) <:
```

Where:

- `n`: the number of cycles/output signals

`(ba.)beat`

Pulses at tempo `t`. `beat` is a standard Faust function.

Usage

```
beat(t) : _
```

Where:

- `t`: tempo in BPM

`(ba.)pulse_countup`

Starts counting up pulses. While trig is 1 the output is counting up, while trig is 0 the counter is reset to 0.

Usage

```
_ : pulse_countup(trig) : _
```

Where:

- `trig`: the trigger signal (1: start at next pulse; 0: reset to 0)

`(ba.)pulse_countdown`

Starts counting down pulses. While trig is 1 the output is counting down, while trig is 0 the counter is reset to 0.

Usage

```
_ : pulse_countdown(trig) : _
```

Where:

- `trig`: the trigger signal (1: start at next pulse; 0: reset to 0)

`(ba.)pulse_countup_loop`

Starts counting up pulses from 0 to n included. While trig is 1 the output is counting up, while trig is 0 the counter is reset to 0. At the end of the countup (n) the output value will be reset to 0.

Usage

```
_ : pulse_countup_loop(n,trig) : _
```

Where:

- `n`: the highest number of the countup (included) before reset to 0.
- `trig`: the trigger signal (1: start at next pulse; 0: reset to 0)

`(ba.)resetCtr`

Function that lets through the mth impulse out of each consecutive group of `n` impulses.

Usage

```
_ : resetCtr(n,m) : _
```

Where:

- `n`: the total number of impulses being split
- `m`: index of impulse to allow to be output

`(ba.)pulse_countdown_loop`

Starts counting down pulses from 0 to n included. While trig is 1 the output is counting down, while trig is 0 the counter is reset to 0. At the end of the countdown (n) the output value will be reset to 0.

Usage

```
_ : pulse_countdown_loop(n,trig) : _
```

Where:

- `n`: the highest number of the countup (included) before reset to 0.
- `trig`: the trigger signal (1: start at next pulse; 0: reset to 0)

Array Processing/Pattern Matching

`(ba.)count`

Count the number of elements of list l. `count` is a standard Faust function.

Usage

```
count(l)
count((10,20,30,40)) -> 4
```

Where:

- `l`: list of elements

`(ba.)take`

Take an element from a list. `take` is a standard Faust function.

Usage

```
take(P,l)
take(3,(10,20,30,40)) -> 30
```

Where:

- `P`: position (int, known at compile time, P > 0)
- `l`: list of elements

`(ba.)subseq`

Extract a part of a list.

Usage

```
subseq(l, p, n)
subseq((10,20,30,40,50,60), 1, 3) -> (20,30,40)
subseq((10,20,30,40,50,60), 4, 1) -> 50
```

Where:

- `l`: list
- `p`: start point (0: begin of list)
- `n`: number of elements

Note:

Faust doesn't have proper lists. Lists are simulated with parallel compositions and there is no empty list.

Selectors (Conditions)

`(ba.)if`

if-then-else implemented with a select2. WARNING : since select2 is strict (always evaluating both branches), the resulting if does not have the usual "lazy" semantic of the C if form, and thus cannot be used to protect against forbidden computations like division-by-zero for instance.

Usage

- `if(cond, then, else) : _`

Where:

- `cond`: condition
- `cond`: signal selected while c is true
- `else`: signal selected while c is false

`(ba.)selector`

Selects the ith input among n at compile time.

Usage

```
selector(I,N)
_,_,_,_ : selector(2,4) : _  // selects the 3rd input among 4
```

Where:

- `I`: input to select (int, numbered from 0, known at compile time)
- `N`: number of inputs (int, known at compile time, N > I)

There is also cselector for selecting among complex input signals of the form (real,imag).

`(ba.)select2stereo`

Select between 2 stereo signals.

Usage

```
_,_,_,_ : select2stereo(bpc) : _,_
```

Where:

- `bpc`: the selector switch (0/1)

`(ba.)selectn`

Selects the ith input among N at run time.

Usage

```
selectn(N,i)
_,_,_,_ : selectn(4,2) : _  // selects the 3rd input among 4
```

Where:

- `N`: number of inputs (int, known at compile time, N > 0)
- `i`: input to select (int, numbered from 0)

Example test program

```
N = 64;
process = par(n, N, (par(i,N,i) : selectn(N,n)));
```

`(ba.)selectmulti`

Selects the ith circuit among N at run time (all should have the same number of inputs and outputs) with a crossfade.

Usage

```
selectmulti(n,lgen,id)
```

Where:

- `n`: crossfade in samples
- `lgen`: list of circuits
- `id`: circuit to select (int, numbered from 0)

Example test program

```
process = selectmulti(ma.SR/10, ((3,9),(2,8),(5,7)), nentry("choice", 0, 0, 2, 1));
process = selectmulti(ma.SR/10, ((_*3,_*9),(_*2,_*8),(_*5,_*7)), nentry("choice", 0, 0, 2, 1
```

Other

`(ba.)latch`

Latch input on positive-going transition of “clock” (“sample-and-hold”).

Usage

```
_ : latch(clocksig) : _
```

Where:

- `clocksig`: hold trigger (0 for hold, 1 for bypass)

`(ba.)sAndH`

Sample And Hold. `sAndH` is a standard Faust function.

Usage

```
_ : sAndH(t) : _
```

Where:

- `t`: hold trigger (0 for hold, 1 for bypass)

`(ba.)downSample`

Down sample a signal. WARNING: this function doesn't change the rate of a signal, it just holds samples... `downSample` is a standard Faust function.

Usage

```
_ : downSample(freq) : _
```

Where:

- `freq`: new rate in Hz

`(ba.)peakhold`

Outputs current max value above zero.

Usage

```
_ : peakhold(mode) : _;
```

Where:

`mode` means: 0 - Pass through. A single sample 0 trigger will work as a reset. 1 - Track and hold max value.

`(ba.)peakholder`

Tracks abs peak and holds peak for 'n' samples.

Usage

```
_ : peakholder(n) : _;
```

Where:

- `n`: number of samples

`(ba.)impulsify`

Turns the signal from a button into an impulse (1,0,0,... when button turns on). `impulsify` is a standard Faust function.

Usage

```
button("gate") : impulsify;
```

`(ba.)automat`

Record and replay to the values the input signal in a loop.

Usage

```
hslider(...) : automat(bps, size, init) : _
```

`(ba.)bpf`

bpf is an environment (a group of related definitions) that can be used to create break-point functions. It contains three functions:

- `start(x,y)` to start a break-point function
- `end(x,y)` to end a break-point function
- `point(x,y)` to add intermediate points to a break-point function

A minimal break-point function must contain at least a start and an end point:

```
f = bpf.start(x0,y0) : bpf.end(x1,y1);
```

A more involved break-point function can contains any number of intermediate points:

```
f = bpf.start(x0,y0) : bpf.point(x1,y1) : bpf.point(x2,y2) : bpf.end(x3,y3);
```

In any case the `x_{i}` must be in increasing order (for all `i`, `x_{i} < x_{i+1}`). For example the following definition :

```
f = bpf.start(x0,y0) : ... : bpf.point(xi,yi) : ... : bpf.end(xn,yn);
```

implements a break-point function f such that:

- `f(x) = y_{0}` when `x < x_{0}`
- `f(x) = y_{n}` when `x > x_{n}`
- `f(x) = y_{i} + (y_{i+1}-y_{i})*(x-x_{i})/(x_{i+1}-x_{i})` when `x_{i} <= x` and `x < x_{i+1}`

`bpf` is a standard Faust function.

`(ba.)listInterp`

Linearly interpolates between the elements of a list.

Usage

```
index = 1.69; // range is 0-4
process = listInterp((800,400,350,450,325),index);
```

Where:

- `index`: the index (float) to interpolate between the different values. The range of `index` depends on the size of the list.

`(ba.)bypass1`

Takes a mono input signal, route it to `e` and bypass it if `bpc = 1`. `bypass1` is a standard Faust function.

Usage

```
_ : bypass1(bpc,e) : _
```

Where:

- `bpc`: bypass switch (0/1)
- `e`: a mono effect

`(ba.)bypass2`

Takes a stereo input signal, route it to `e` and bypass it if `bpc = 1`. `bypass2` is a standard Faust function.

Usage

```
_,_ : bypass2(bpc,e) : _,_
```

Where:

- `bpc`: bypass switch (0/1)
- `e`: a stereo effect

`(ba.)bypass1to2`

Bypass switch for effect `e` having mono input signal and stereo output. Effect `e` is bypassed if `bpc = 1`. `bypass1to2` is a standard Faust function.

Usage

```
_ : bypass1(bpc,e) : _,_
```

Where:

- `bpc`: bypass switch (0/1)
- `e`: a mono-to-stereo effect

`(ba.)bypass_fade`

Bypass an arbitrary (N x N) circuit with 'n' samples crossfade. Once bypassed the
effect is replaced by par(i,N,_). Bypassed circuits can be chained.

Usage

```
_ : bypass_fade(n,b,e) : _
or
_,_ : bypass_fade(n,b,e) : _,_
```

- `n`: number of samples for the crossfade
- `b`: bypass switch (0/1)
- `e`: N x N circuit

Examples

```
process = bypass_fade(ma.SR/10, checkbox("bypass echo"), echo);
process = bypass_fade(ma.SR/10, checkbox("bypass reverb"), freeverb);
```

`(ba.)toggle`

Triggered by the change of 0 to 1, it toggles the output value between 0 and 1.

Usage

```
_ : toggle : _
```

Examples

```
button("toggle") : toggle : vbargraph("output", 0, 1)
(an.amp_follower(0.1) > 0.01) : toggle : vbargraph("output", 0, 1) // takes audio input
```

`(ba.)on_and_off`

The first channel set the output to 1, the second channel to 0.

Usage

```
_ , _ : on_and_off : _
```

Example

```
button("on"), button("off") : on_and_off : vbargraph("output", 0, 1)
```

`(ba.)selectoutn`

Route input to the output among N at run time.

Usage

```
_ : selectoutn(N, i) : _,_,...N
```

Where:

- `N`: number of outputs (int, known at compile time, N > 0)
- `i`: output number to route to (int, numbered from 0) (i.e. slider)

Example

```
process = 1 : selectoutn(3, sel) : par(i, 3, vbargraph("v.bargraph %i", 0, 1));
sel = hslider("volume", 0, 0, 2, 1) : int;
```

Sliding Reduce

Provides various operations on the last N samples using a high order 'slidingReduce(op,N,maxN,disabledVal,x)" fold-like function:

- `slidingSum(n)`: the sliding sum of the last n input samples, CPU-light
- `slidingSump(n,maxn)`: the sliding sum of the last n input samples, numerically stable "forever"
- `slidingMax(n,maxn)`: the sliding max of the last n input samples
- `slidingMin(n,maxn)`: the sliding min of the last n input samples
- `slidingMean(n)`: the sliding mean of the last n input samples, CPU-light
- `slidingMeanp(n,maxn)`: the sliding mean of the last n input samples, numerically stable "forever"
- `slidingRMS(n)`: the sliding RMS of the last n input samples, CPU-light
- `slidingRMSp(n,maxn)`: the sliding RMS of the last n input samples, numerically stable "forever"

Working Principle

If we want the maximum of the last 8 values, we can do that as:

```
simpleMax(x) =
 (
   (
     max(x@0,x@1),
     max(x@2,x@3)
   ) :max
 ),
 (
   (
     max(x@4,x@5),
     max(x@6,x@7)
   ) :max
 )
 :max;
```

`max(x@2,x@3)` is the same as `max(x@0,x@1)@2` but the latter re-uses a value we already computed,so is more efficient. Using the same trick for values 4 trough 7, we can write:

```
efficientMax(x)=
 (
   (
     max(x@0,x@1),
     max(x@0,x@1)@2
   ) :max
 ),
 (
```

```
   (
     max(x@0,x@1),
     max(x@0,x@1)@2
   ) :max@4
 )
 :max;
```

We can rewrite it recursively, so it becomes possible to get the maximum at have any number of values, as long as it's a power of 2.

```
recursiveMax =
 case {
   (1,x) => x;
   (N,x) =>  max(recursiveMax(N/2,x) , recursiveMax(N/2,x)@(N/2));
 };
```

What if we want to look at a number of values that's not a power of 2? For each value, we will have to decide whether to use it or not. If N is bigger than the index of the value, we use it, otherwise we replace it with (`0-(ma.INFINITY)`):

```
variableMax(N,x) =
 max(
   max(
     (
       (x@0 : useVal(0)),
       (x@1 : useVal(1))
     ):max,
     (
       (x@2 : useVal(2)),
       (x@3 : useVal(3))
     ):max
   ),
   max(
     (
       (x@4 : useVal(4)),
       (x@5 : useVal(5))
     ):max,
     (
       (x@6 : useVal(6)),
       (x@7 : useVal(7))
     ):max
   )
 )
 with {
 useVal(i) = select2((N>=i) , (0-(ma.INFINITY)),_);
};
```

Now it becomes impossible to re-use any values. To fix that let's first look at

how we'd implement it using recursiveMax, but with a fixed N that is not a power of 2. For example, this is how you'd do it with `N=3`:

```
binaryMaxThree(x) =
 (
   recursiveMax(1,x)@0, // the first x
   recursiveMax(2,x)@1  // the second and third x
 ):max;
```

`N=6`

```
binaryMaxSix(x) =
 (
   recursiveMax(2,x)@0, // first two
   recursiveMax(4,x)@2  // third trough sixth
 ):max;
```

Note that `recursiveMax(2,x)` is used at a different delay then in `binaryMaxThree`, since it represents 1 and 2, not 2 and 3. Each block is delayed the combined size of the previous blocks.

`N=7`

```
binaryMaxSeven(x) =
 (
   (
     recursiveMax(1,x)@0, // first x
     recursiveMax(2,x)@1  // second and third
   ):max,
   (
     recursiveMax(4,x)@3  // fourth trough seventh
   )
 ):max;
```

To make a variable version, we need to know which powers of two are used, and at which delay time.

Then it becomes a matter of:

- lining up all the different block sizes in parallel: the first `par()` statement
- delaying each the appropriate amount: `sumOfPrevBlockSizes()`
- turning it on or off: `useVal()`
- getting the maximum of all of them: `combine()`

In Faust, we can only do that for a fixed maximum number of values: `maxN`

```
variableBinaryMaxN(N,maxN,x) =
 par(i,maxNrBits,recursiveMax(pow2(i),x)@sumOfPrevBlockSizes(N,maxN,i)  : useVal(i)) : combi
   // The sum of all the sizes of the previous blocks
   sumOfPrevBlockSizes(N,maxN,0) = 0;
   sumOfPrevBlockSizes(N,maxN,i) = (subseq((allBlockSizes(N,maxN)),0,i):>_);
```

```
  allBlockSizes(N,maxN) = par(i, maxNrBits, pow2(i) * isUsed(i) );
  maxNrBits = int2nrOfBits(maxN);
  // get the maximum of all blocks
  combine(2) = max;
  combine(N) = max(combine(N-1),_);
  // Decide wether or not to use a certain value, based on N
  useVal(i) = select2( isUsed(i), (0-(ma.INFINITY)),_);
  isUsed(i) = take(i+1,(int2bin(N,maxN)));
};
```

(ba.)slidingReduce

Fold-like high order function. Apply a commutative binary operation `<op>` to the last `<n>` consecutive samples of a signal `<x>`. For example : `slidingReduce(max,128,128,-(ma.INFINITY))` will compute the maximum of the last 128 samples. The output is updated each sample, unlike reduce, where the output is constant for the duration of a block.

Usage

```
_ : slidingReduce(op,N,maxN,disabledVal) : _
```

Where:

- `N`: the number of values to process
- `maxN`: the maximum number of values to process, needs to be a power of 2
- `op`: the operator. Needs to be a commutative one.
- `disabledVal`: the value to use when we want to ignore a value.

In other words, `op(x,disabledVal)` should equal to `x`. For example, `+(x,0)` equals `x` and `min(x,ma.INFINITY)` equals `x`. So if we want to calculate the sum, we need to give 0 as `disabledVal`, and if we want the minimum, we need to give `ma.INFINITY` as `disabledVal`.

(ba.)slidingSum

The sliding sum of the last n input samples.

It will eventually run into numerical trouble when there is a persistent dc component. If that matters in your application, use the more CPU-intensive (ba.)slidingSump.

Usage

```
_ : slidingSum(N) : _
```

Where:

- `N`: the number of values to process

`(ba.)slidingSump`

The sliding sum of the last n input samples.

It uses a lot more CPU then (ba.)slidingSum(n,maxn), but is numerically stable "forever" in return.

Usage

```
_ : slidingSump(N,maxN) : _
```

Where:

- `N`: the number of values to process
- `maxN`: the maximum number of values to process, needs to be a power of 2

`(ba.)slidingMax`

The sliding maximum of the last n input samples.

Usage

```
_ : slidingMax(N,maxN) : _
```

Where:

- `N`: the number of values to process
- `maxN`: the maximum number of values to process, needs to be a power of 2

`(ba.)slidingMin`

The sliding minimum of the last n input samples.

Usage

```
_ : slidingMin(N,maxN) : _
```

Where:

- `N`: the number of values to process
- `maxN`: the maximum number of values to process, needs to be a power of 2

`(ba.)slidingMean`

The sliding mean of the last n input samples.

It will eventually run into numerical trouble when there is a persistent dc component. If that matters in your application, use the more CPU-intensive (ba.)slidinRMSp.

Usage

```
_ : slidingMean(N,maxN) : _
```

Where:

- `N`: the number of values to process

`(ba.)slidingMeanp`

The sliding mean of the last n input samples.

It uses a lot more CPU then (ba.)slidingMean(n,maxn), but is numerically stable "forever" in return.

Usage

```
_ : slidingMeanp(N,maxN) : _
```

Where:

- `N`: the number of values to process
- `maxN`: the maximum number of values to process, needs to be a power of 2

`(ba.)slidingRMS`

The root mean square of the last n input samples.

It will eventually run into numerical trouble when there is a persistent dc component. If that matters in your application, use the more CPU-intensive (ba.)slidinRMSp.

`(ba.)slidingRMSp`

The root mean square of the last n input samples.

It uses a lot more CPU then (ba.)slidingRMS(n,maxn), but is numerically stable "forever" in return.

Usage

```
_ : slidingRMSp(N,maxN) : _
```

Where:

- `N`: the number of values to process
- `maxN`: the maximum number of values to process, needs to be a power of 2

---

compressors.lib

A library of compressor effects. Its official prefix is `co`.

Functions Reference

`(co.)peak_compression_gain_mono`

Mono dynamic range compressor gain computer. `peak_compression_gain_mono` is a standard Faust function

Usage

```
_ : peak_compression_gain_mono(strength,thresh,att,rel,knee,prePost) : _
```

Where:

- `strength`: strength of the compression (0 = no compression, 1 means hard limiting, >1 means over-compression)
- `thresh`: dB level threshold above which compression kicks in
- `att`: attack time = time constant (sec) when level & compression going up
- `rel`: release time = time constant (sec) coming out of compression
- `knee`: a gradual increase in gain reduction around the threshold: Below thresh-(knee/2) there is no gain reduction, above thresh+(knee/2) there is the same gain reduction as without a knee, and in between there is a gradual increase in gain reduction.
- `prePost`: places the level detector either at the input or after the gain computer; this turns it from a linear return-to-zero detector into a log domain return-to-threshold detector

`(co.)peak_compression_gain_N_chan`

N channel dynamic range compressor gain computer. `peak_compression_gain_N_chan` is a standard Faust function

Usage

```
si.bus(N) : peak_compression_gain_N_chan(strength,thresh,att,rel,knee,prePost,link,N) : si.b
```

Where:

- `strength`: strength of the compression (0 = no compression, 1 means hard limiting, >1 means over-compression)
- `thresh`: dB level threshold above which compression kicks in
- `att`: attack time = time constant (sec) when level & compression going up
- `rel`: release time = time constant (sec) coming out of compression
- `knee`: a gradual increase in gain reduction around the threshold: Below thresh-(knee/2) there is no gain reduction, above thresh+(knee/2) there is the same gain reduction as without a knee, and in between there is a gradual increase in gain reduction.
- `prePost`: places the level detector either at the input or after the gain computer; this turns it from a linear return-to-zero detector into a log domain return-to-threshold detector
- `link`: the amount of linkage between the channels. 0 = each channel is independent, 1 = all channels have the same amount of gain reduction
- `N`: the number of channels of the compressor

`(co.)FFcompressor_N_chan`

feed forward N channel dynamic range compressor. `FFcompressor_N_chan` is a standard Faust function

Usage

```
si.bus(N) : FFcompressor_N_chan(strength,thresh,att,rel,knee,prePost,link,meter,N) : si.bus(
```

Where:

- `strength`: strength of the compression (0 = no compression, 1 means hard limiting, >1 means over-compression)
- `thresh`: dB level threshold above which compression kicks in
- `att`: attack time = time constant (sec) when level & compression going up
- `rel`: release time = time constant (sec) coming out of compression
- `knee`: a gradual increase in gain reduction around the threshold: Below thresh-(knee/2) there is no gain reduction, above thresh+(knee/2) there is the same gain reduction as without a knee, and in between there is a gradual increase in gain reduction.
- `prePost`: places the level detector either at the input or after the gain computer; this turns it from a linear return-to-zero detector into a log domain return-to-threshold detector
- `link`: the amount of linkage between the channels. 0 = each channel is independent, 1 = all channels have the same amount of gain reduction
- `meter`: a gain reduction meter. It can be implemented like so: meter = *<:(*, (ba.linear2db:max(maxGR):meter_group((hbargraph("[1][unit:dB][tooltip: gain reduction in dB]", maxGR, 0))))):attach;
- `N`: the number of channels of the compressor

`(co.)FBcompressor_N_chan`

feed back N channel dynamic range compressor. `FBcompressor_N_chan` is a standard Faust function

Usage

```
si.bus(N) : FBcompressor_N_chan(strength,thresh,att,rel,knee,prePost,link,meter,N) : si.bus(
```

Where:

- `strength`: strength of the compression (0 = no compression, 1 means hard limiting, >1 means over-compression)
- `thresh`: dB level threshold above which compression kicks in
- `att`: attack time = time constant (sec) when level & compression going up
- `rel`: release time = time constant (sec) coming out of compression
- `knee`: a gradual increase in gain reduction around the threshold: Below thresh-(knee/2) there is no gain reduction, above thresh+(knee/2) there is the same gain reduction as without a knee, and in between there is a gradual increase in gain reduction.
- `prePost`: places the level detector either at the input or after the gain computer; this turns it from a linear return-to-zero detector into a log domain return-to-threshold detector
- `link`: the amount of linkage between the channels. 0 = each channel is independent, 1 = all channels have the same amount of gain reduction
- `meter`: a gain reduction meter. It can be implemented like so: meter = *<:(*, (ba.linear2db:max(maxGR):meter_group((hbargraph("[1][unit:dB][tooltip: gain reduction in dB]", maxGR, 0))))):attach;
- `N`: the number of channels of the compressor

`(co.)FFFBcompressor_N_chan`

feed forward / feed back N channel dynamic range compressor. the feedback part has a much higher strength, so they end up sounding similar `FFFBcompressor_N_chan` is a standard Faust function

Usage

```
si.bus(N) : FFFBcompressor_N_chan(strength,thresh,att,rel,knee,prePost,link,FBFF,meter,N) :
```

Where:

- `strength`: strength of the compression (0 = no compression, 1 means hard limiting, >1 means over-compression)
- `thresh`: dB level threshold above which compression kicks in
- `att`: attack time = time constant (sec) when level & compression going up
- `rel`: release time = time constant (sec) coming out of compression
- `knee`: a gradual increase in gain reduction around the threshold: Below thresh-(knee/2) there is no gain reduction, above thresh+(knee/2) there

is the same gain reduction as without a knee, and in between there is a gradual increase in gain reduction.
- `prePost`: places the level detector either at the input or after the gain computer; this turns it from a linear return-to-zero detector into a log domain return-to-threshold detector
- `link`: the amount of linkage between the channels. 0 = each channel is independent, 1 = all channels have the same amount of gain reduction
- `FFFB`: fade between feed forward (0) and feed back (1) compression.
- `meter`: a gain reduction meter. It can be implemented like so: meter = *<:(*, (ba.linear2db:max(maxGR):meter_group((hbargraph("[1][unit:dB][tooltip: gain reduction in dB]", maxGR, 0))))):attach;
- `N`: the number of channels of the compressor

`(co.)RMS_compression_gain_mono`

Mono RMS dynamic range compressor gain computer. `RMS_compression_gain_mono` is a standard Faust function

Usage

```
_ : RMS_compression_gain_mono(strength,thresh,att,rel,knee,prePost) : _
```

Where:

- `strength`: strength of the compression (0 = no compression, 1 means hard limiting, >1 means over-compression)
- `thresh`: dB level threshold above which compression kicks in
- `att`: attack time = time constant (sec) when level & compression going up
- `rel`: release time = time constant (sec) coming out of compression
- `knee`: a gradual increase in gain reduction around the threshold: Below thresh-(knee/2) there is no gain reduction, above thresh+(knee/2) there is the same gain reduction as without a knee, and in between there is a gradual increase in gain reduction.
- `prePost`: places the level detector either at the input or after the gain computer; this turns it from a linear return-to-zero detector into a log domain return-to-threshold detector

`(co.)RMS_compression_gain_N_chan`

RMS N channel dynamic range compressor gain computer. `RMS_compression_gain_N_chan` is a standard Faust function

Usage

```
si.bus(N) : RMS_compression_gain_N_chan(strength,thresh,att,rel,knee,prePost,link,N) : si.bu
```

Where:

- `strength`: strength of the compression (0 = no compression, 1 means hard limiting, >1 means over-compression)
- `thresh`: dB level threshold above which compression kicks in
- `att`: attack time = time constant (sec) when level & compression going up
- `rel`: release time = time constant (sec) coming out of compression
- `knee`: a gradual increase in gain reduction around the threshold: Below thresh-(knee/2) there is no gain reduction, above thresh+(knee/2) there is the same gain reduction as without a knee, and in between there is a gradual increase in gain reduction.
- `prePost`: places the level detector either at the input or after the gain computer; this turns it from a linear return-to-zero detector into a log domain return-to-threshold detector
- `link`: the amount of linkage between the channels. 0 = each channel is independent, 1 = all channels have the same amount of gain reduction
- `N`: the number of channels of the compressor

`(co.)RMS_FFFBcompressor_N_chan`

RMS feed forward / feed back N channel dynamic range compressor. the feedback part has a much higher strength, so they end up sounding similar `RMS_FFFBcompressor_N_chan` is a standard Faust function

Usage

```
si.bus(N) : RMS_FFFBcompressor_N_chan(strength,thresh,att,rel,knee,prePost,link,FBFF,meter,N
```

Where:

- `strength`: strength of the compression (0 = no compression, 1 means hard limiting, >1 means over-compression)
- `thresh`: dB level threshold above which compression kicks in
- `att`: attack time = time constant (sec) when level & compression going up
- `rel`: release time = time constant (sec) coming out of compression
- `knee`: a gradual increase in gain reduction around the threshold: Below thresh-(knee/2) there is no gain reduction, above thresh+(knee/2) there is the same gain reduction as without a knee, and in between there is a gradual increase in gain reduction.
- `prePost`: places the level detector either at the input or after the gain computer; this turns it from a linear return-to-zero detector into a log domain return-to-threshold detector
- `link`: the amount of linkage between the channels. 0 = each channel is independent, 1 = all channels have the same amount of gain reduction
- `FFFB`: fade between feed forward (0) and feed back (1) compression.
- `meter`: a gain reduction meter. It can be implemented like so: meter = *<:(*, (ba.linear2db:max(maxGR):meter_group((hbargraph("[1][unit:dB][tooltip: gain reduction in dB]", maxGR, 0))))):attach;
- `N`: the number of channels of the compressor

`(co.)RMS_FBcompressor_peak_limiter_N_chan`

N channel RMS feed back compressor into peak limiter feeding back into the FB compressor. By combining them this way, they complement each other optimally: The RMS compressor doesn't have to deal with the peaks, and the peak limiter get's spared from the steady state signal. the feedback part has a much higher strength, so they end up sounding similar `RMS_FBcompressor_peak_limiter_N_chan` is a standard Faust function

Usage

```
si.bus(N) : RMS_FBcompressor_peak_limiter_N_chan(strength,thresh,threshLim,att,rel,knee,link
```

Where:

- `strength`: strength of the compression (0 = no compression, 1 means hard limiting, >1 means over-compression)
- `thresh`: dB level threshold above which compression kicks in
- `threshLim`: dB level threshold above which the brick wall limiter kicks in
- `att`: attack time = time constant (sec) when level & compression going up this is also used as the release time of the limiter
- `rel`: release time = time constant (sec) coming out of compression
- `knee`: a gradual increase in gain reduction around the threshold: Below thresh-(knee/2) there is no gain reduction, above thresh+(knee/2) there is the same gain reduction as without a knee, and in between there is a gradual increase in gain reduction. the limiter uses a knee half this size
- `link`: the amount of linkage between the channels. 0 = each channel is independent, 1 = all channels have the same amount of gain reduction
- `meter`: a gain reduction meter. It can be implemented like so: meter = *<:(*, (ba.linear2db:max(maxGR):meter_group((hbargraph("[1][unit:dB][tooltip: gain reduction in dB]", maxGR, 0))))):attach;
- `N`: the number of channels of the compressor

Backward compatibility section

Functions Reference

`(co.)compressor_mono`

Mono dynamic range compressors. `compressor_mono` is a standard Faust function.

Usage

```
_ : compressor_mono(ratio,thresh,att,rel) : _
```

Where:

- `ratio`: compression ratio (1 = no compression, >1 means compression)

- `thresh`: dB level threshold above which compression kicks in (0 dB = max level)
- `att`: attack time = time constant (sec) when level & compression going up
- `rel`: release time = time constant (sec) coming out of compression

References

- http://en.wikipedia.org/wiki/Dynamic_range_compression
- https://ccrma.stanford.edu/~jos/filters/Nonlinear_Filter_Example_Dynamic.html
- Albert Graef's "faust2pd"/examples/synth/compressor_.dsp
- More features: https://github.com/magnetophon/faustCompressors

---

`(co.)compressor_stereo`

Stereo dynamic range compressors.

Usage

```
_,_ : compressor_stereo(ratio,thresh,att,rel) : _,_
```

Where:

- `ratio`: compression ratio (1 = no compression, >1 means compression)
- `thresh`: dB level threshold above which compression kicks in (0 dB = max level)
- `att`: attack time = time constant (sec) when level & compression going up
- `rel`: release time = time constant (sec) coming out of compression

References

- http://en.wikipedia.org/wiki/Dynamic_range_compression
- https://ccrma.stanford.edu/~jos/filters/Nonlinear_Filter_Example_Dynamic.html
- Albert Graef's "faust2pd"/examples/synth/compressor_.dsp
- More features: https://github.com/magnetophon/faustCompressors

---

`(co.)limiter_1176_R4_mono`

A limiter guards against hard-clipping. It can be implemented as a compressor having a high threshold (near the clipping level), fast attack and release, and high ratio. Since the ratio is so high, some knee smoothing is desirable ("soft limiting"). This example is intended to get you started using compressor_* as a limiter, so all parameters are hardwired to nominal values here. Ratios: 4 (moderate compression), 8 (severe compression), 12 (mild limiting), or 20

to 1 (hard limiting) Att: 20-800 MICROseconds (Note: scaled by ratio in the 1176) Rel: 50-1100 ms (Note: scaled by ratio in the 1176) Mike Shipley likes 4:1 (Grammy-winning mixer for Queen, Tom Petty, etc.) Faster attack gives "more bite" (e.g. on vocals) He hears a bright, clear eq effect as well (not implemented here). `limiter_1176_R4_mono` is a standard Faust function.

Usage

```
_ : limiter_1176_R4_mono : _;
```

Reference:

http://en.wikipedia.org/wiki/1176_Peak_Limiter

`(co.)limiter_1176_R4_stereo`

A limiter guards against hard-clipping. It can be implemented as a compressor having a high threshold (near the clipping level), fast attack and release, and high ratio. Since the ratio is so high, some knee smoothing is desirable ("soft limiting"). This example is intended to get you started using compressor_* as a limiter, so all parameters are hardwired to nominal values here. Ratios: 4 (moderate compression), 8 (severe compression), 12 (mild limiting), or 20 to 1 (hard limiting) Att: 20-800 MICROseconds (Note: scaled by ratio in the 1176) Rel: 50-1100 ms (Note: scaled by ratio in the 1176) Mike Shipley likes 4:1 (Grammy-winning mixer for Queen, Tom Petty, etc.) Faster attack gives "more bite" (e.g. on vocals) He hears a bright, clear eq effect as well (not implemented here)

Usage

```
_,_ : limiter_1176_R4_stereo : _,_;
```

Reference:

http://en.wikipedia.org/wiki/1176_Peak_Limiter

---

delays.lib

This library contains a collection of delay functions. Its official prefix is `de`.

Basic Delay Functions

`(de.)delay`

Simple `d` samples delay where `n` is the maximum delay length as a number of samples. Unlike the `@` delay operator, here the delay signal `d` is explicitly bounded to the interval [0..n]. The consequence is that delay will compile even if the interval of d can't be computed by the compiler. `delay` is a standard Faust function.

Usage

```
_ : delay(n,d) : _
```

Where:

- `n`: the max delay length (in samples)
- `d`: the delay length as a number of samples (integer)

`(de.)fdelay`

Simple `d` samples fractional delay based on 2 interpolated delay lines where `n` is the maximum delay length as a number of samples.

`(de.)sdelay`

s(mooth)delay: a mono delay that doesn't click and doesn't transpose when the delay time is changed.

Usage

```
_ : sdelay(N,it,dt) : _
```

Where :

- `N`: maximal delay in samples
- `it`: interpolation time (in samples) for example 1024
- `dt`: delay time (in samples)

Lagrange Interpolation

`(de.)fdelaylti` and `(de.)fdelayltv`

Fractional delay line using Lagrange interpolation.

Usage

```
_ : fdelaylt[i|v](order, maxdelay, delay, inputsignal) : _
```

Where `order=1,2,3,...` is the order of the Lagrange interpolation polynomial.

`fdelaylti` is most efficient, but designed for constant/slowly-varying delay. `fdelayltv` is more expensive and more robust when the delay varies rapidly.

NOTE: The requested delay should not be less than `(N-1)/2`.

References

- https://ccrma.stanford.edu/~jos/pasp/Lagrange_Interpolation.html
  - (fixed-delay case)(https://ccrma.stanford.edu/~jos/Interpolation/Efficient_Time_Invariant_Lagrang
  - (variable-delay case)(https://ccrma.stanford.edu/~jos/Interpolation/Time_Varying_Lagrange_Interp
- Timo I. Laakso et al., "Splitting the Unit Delay - Tools for Fractional Delay Filter Design", IEEE Signal Processing Magazine, vol. 13, no. 1, pp. 30-60, Jan 1996.
- Philippe Depalle and Stephan Tassart, "Fractional Delay Lines using Lagrange Interpolators", ICMC Proceedings, pp. 341-343, 1996.

`(de.)fdelay[n]`

For convenience, `fdelay1`, `fdelay2`, `fdelay3`, `fdelay4`, `fdelay5` are also available where n is the order of the interpolation.

Thiran Allpass Interpolation

Thiran Allpass Interpolation

Reference

https://ccrma.stanford.edu/~jos/pasp/Thiran_Allpass_Interpolators.html

`(de.)fdelay[n]a`

Delay lines interpolated using Thiran allpass interpolation.

Usage

```
_ : fdelay[N]a(maxdelay, delay, inputsignal) : _
```

(exactly like `fdelay`)

Where:

- `N`=1,2,3, or 4 is the order of the Thiran interpolation filter, and the delay argument is at least N - 1/2.

Note

The interpolated delay should not be less than `N - 1/2`. (The allpass delay ranges from `N - 1/2` to `N + 1/2`.) This constraint can be alleviated by altering the code, but be aware that allpass filters approach zero delay by means of pole-zero cancellations. The delay range `[N-1/2,N+1/2]` is not optimal. What is?

Delay arguments too small will produce an UNSTABLE allpass!

Because allpass interpolation is recursive, it is not as robust as Lagrange interpolation under time-varying conditions. (You may hear clicks when changing the delay rapidly.)

First-order allpass interpolation, delay d in [0.5,1.5]

---

demos.lib

This library contains a set of demo functions based on examples located in the `/examples` folder. Its official prefix is `dm`.

Analyzers

`(dm.)mth_octave_spectral_level_demo`

Demonstrate mth_octave_spectral_level in a standalone GUI.

Usage

```
_ : mth_octave_spectral_level_demo(BandsPerOctave);
_ : spectral_level_demo : _; // 2/3 octave
```

Filters

`(dm.)parametric_eq_demo`

A parametric equalizer application.

Usage:

```
_ : parametric_eq_demo : _ ;
```

`(dm.)spectral_tilt_demo`

A spectral tilt application.

Usage

```
_ : spectral_tilt_demo(N) : _ ;
```

Where:

- `N`: filter order (integer)

All other parameters interactive

`(dm.)mth_octave_filterbank_demo` and `(dm.)filterbank_demo`

Graphic Equalizer: Each filter-bank output signal routes through a fader.

Usage

```
_ : mth_octave_filterbank_demo(M) : _
_ : filterbank_demo : _
```

Where:

- `N`: number of bands per octave

Effects

`(dm.)cubicnl_demo`

Distortion demo application.

Usage:

```
_ : cubicnl_demo : _;
```

`(dm.)gate_demo`

Gate demo application.

Usage

```
_,_ : gate_demo : _,_;
```

`(dm.)compressor_demo`

Compressor demo application.

Usage

```
_,_ : compressor_demo : _,_;
```

`(dm.)moog_vcf_demo`

Illustrate and compare all three Moog VCF implementations above.

Usage

```
_ : moog_vcf_demo : _;
```

`(dm.)wah4_demo`

Wah pedal application.

Usage

```
_ : wah4_demo : _;
```

`(dm.)crybaby_demo`

Crybaby effect application.

Usage

```
_ : crybaby_demo : _ ;
```

`(dm.)flanger_demo`

Flanger effect application.

Usage

```
_,_ : flanger_demo : _,_;
```

`(dm.)phaser2_demo`

Phaser effect demo application.

Usage

```
_,_ : phaser2_demo : _,_;
```

`(dm.)freeverb_demo`

Freeverb demo application.

Usage

```
_,_ : freeverb_demo : _,_;
```

`(dm.)stereo_reverb_tester`

Handy test inputs for reverberator demos below.

Usage

```
_ : stereo_reverb_tester : _
```

`(dm.)fdnrev0_demo`

A reverb application using `fdnrev0`.

Usage

```
_,_ : fdnrev0_demo(N,NB,BBSO) : _,_
```

Where:

- `n`: Feedback Delay Network (FDN) order / number of delay lines used = order of feedback matrix / 2, 4, 8, or 16 [extend primes array below for 32, 64, ...]
- `nb`: Number of frequency bands / Number of (nearly) independent T60 controls / Integer 3 or greater
- `bbso` = Butterworth band-split order / order of lowpass/highpass bandsplit used at each crossover freq / odd positive integer

---

(dm.)zita_rev_fdn_demo

Reverb demo application based on `zita_rev_fdn`.

Usage

```
si.bus(8) : zita_rev_fdn_demo : si.bus(8)
```

(dm.)zita_light

Light version of `dm.zita_rev1` with only 2 UI elements.

Usage

```
_,_ : zita_light : _,_
```

(dm.)zita_rev1

Example GUI for `zita_rev1_stereo` (mostly following the Linux `zita-rev1` GUI).

Only the dry/wet and output level parameters are "dezippered" here. If parameters are to be varied in real time, use `smooth(0.999)` or the like in the same way.

Usage

```
_,_ : zita_rev1 : _,_
```

Reference

http://www.kokkinizita.net/linuxaudio/zita-rev1-doc/quickguide.html

---

Generators

(dm.)sawtooth_demo

An application demonstrating the different sawtooth oscillators of Faust.

Usage

```
sawtooth_demo : _
```

(dm.)virtual_analog_oscillator_demo

Virtual analog oscillator demo application.

Usage

```
virtual_analog_oscillator_demo : _
```

(dm.)oscrs_demo

Simple application demoing filter based oscillators.

Usage

```
oscrs_demo : _
```

(dm.)velvet_noise_demo

Listen to velvet_noise!

Usage

```
velvet_noise_demo : _
```

(dm.)latch_demo

Illustrate latch operation

Usage

```
echo 'import("stdfaust.lib");' > latch_demo.dsp
echo 'process = dm.latch_demo;' >> latch_demo.dsp
faust2octave latch_demo.dsp
Octave:1> plot(faustout);
```

(dm.)envelopes_demo

Illustrate various envelopes overlaid, including their gate * 1.1.

Usage

```
echo 'import("stdfaust.lib");' > envelopes_demo.dsp
echo 'process = dm.envelopes_demo;' >> envelopes_demo.dsp
faust2octave envelopes_demo.dsp
Octave:1> plot(faustout);
```

(dm.)fft_spectral_level_demo

Make a real-time spectrum analyzer using FFT from analyzers.lib

Usage

```
echo 'import("stdfaust.lib");' > fft_spectral_level_demo.dsp
echo 'process = dm.fft_spectral_level_demo;' >> fft_spectral_level_demo.dsp
Mac:
  faust2caqt fft_spectral_level_demo.dsp
  open fft_spectral_level_demo.app
Linux GTK:
  faust2jack fft_spectral_level_demo.dsp
  ./fft_spectral_level_demo
Linux QT:
  faust2jaqt fft_spectral_level_demo.dsp
  ./fft_spectral_level_demo
```

(dm.)reverse_echo_demo(nChans)

Multichannel echo effect with reverse delays

Usage

```
echo 'import("stdfaust.lib");' > reverse_echo_demo.dsp
echo 'nChans = 3; // Any integer > 1 should work here' >> reverse_echo_demo.dsp
echo 'process = dm.reverse_echo_demo(nChans);' >> reverse_echo_demo.dsp
Mac:
  faust2caqt reverse_echo_demo.dsp
  open reverse_echo_demo.app
Linux GTK:
  faust2jack reverse_echo_demo.dsp
  ./reverse_echo_demo
Linux QT:
  faust2jaqt reverse_echo_demo.dsp
  ./reverse_echo_demo
Etc.
```

`(dm.)pospass_demo`

Use Positive-Pass Filter pospass() to frequency-shift a sine tone. First, a real sinusoid is converted to its analytic-signal form using pospass() to filter out its negative frequency component. Next, it is multiplied by a modulating complex sinusoid at the shifting frequency to create the frequency-shifted result. The real and imaginary parts are output to channels 1 & 2. For a more interesting frequency-shifting example, check the "Use Mic" checkbox to replace the input sinusoid by mic input. Note that frequency shifting is not the same as frequency scaling. A frequency-shifted harmonic signal is usually not harmonic. Very small frequency shifts give interesting chirp effects when there is feedback around the frequency shifter.

Usage

```
echo 'import("stdfaust.lib");' > pospass_demo.dsp
echo 'process = dm.pospass_demo;' >> pospass_demo.dsp
Mac:
  faust2caqt pospass_demo.dsp
  open pospass_demo.app
Linux GTK:
  faust2jack pospass_demo.dsp
  ./pospass_demo
Linux QT:
  faust2jaqt pospass_demo.dsp
  ./pospass_demo
Etc.
```

`(dm.)exciter`

Psychoacoustic harmonic exciter, with GUI.

Usage

```
_ : exciter : _
```

References

- https://secure.aes.org/forum/pubs/ebriefs/?elib=16939
- https://www.researchgate.net/publication/258333577_Modeling_the_Harmonic_Exciter

---

`(dm.)vocoder_demo`

Use example of the vocoder function where an impulse train is used as excitation.

Usage

```
_ : vocoder_demo : _;
```

---

dx7.lib

Yamaha DX7 emulation library. Its official prefix is `dx`.

`(dx.)dx7_ampf`

DX7 amplitude conversion function. 3 versions of this function are available:

- `dx7_amp_bpf`: BPF version (same as in the CSOUND toolkit)
- `dx7_amp_func`: estimated mathematical equivalent of `dx7_amp_bpf`
- `dx7_ampf`: default (sugar for `dx7_amp_func`)

Usage:

```
dx7AmpPreset : dx7_ampf_bpf : _
```

Where:

- `dx7AmpPreset`: DX7 amplitude value (0-99)

`(dx.)dx7_egraterisef`

DX7 envelope generator rise conversion function. 3 versions of this function are available:

- `dx7_egraterise_bpf`: BPF version (same as in the CSOUND toolkit)
- `dx7_egraterise_func`: estimated mathematical equivalent of `dx7_egraterise_bpf`
- `dx7_egraterisef`: default (sugar for `dx7_egraterise_func`)

Usage:

```
dx7envelopeRise : dx7_egraterisef : _
```

Where:

- `dx7envelopeRise`: DX7 envelope rise value (0-99)

`(dx.)dx7_egraterisepercf`

DX7 envelope generator percussive rise conversion function. 3 versions of this function are available:

- `dx7_egraterisepercf_bpf`: BPF version (same as in the CSOUND toolkit)
- `dx7_egraterisepercf_func`: estimated mathematical equivalent of `dx7_egraterisepercf_bpf`
- `dx7_egraterisepercf`: default (sugar for `dx7_egraterisepercf_func`)

Usage:

```
dx7envelopePercRise : dx7_egraterisepercf : _
```

Where:

- `dx7envelopePercRise`: DX7 envelope percussive rise value (0-99)

`(dx.)dx7_egratedecayf`

DX7 envelope generator decay conversion function. 3 versions of this function are available:

- `dx7_egratedecay_bpf`: BPF version (same as in the CSOUND toolkit)
- `dx7_egratedecay_func`: estimated mathematical equivalent of `dx7_egratedecay_bpf`
- `dx7_egratedecayf`: default (sugar for `dx7_egratedecay_func`)

Usage:

```
dx7envelopeDecay : dx7_egratedecayf : _
```

Where:

- `dx7envelopeDecay`: DX7 envelope decay value (0-99)

`(dx.)dx7_egratedecaypercf`

DX7 envelope generator percussive decay conversion function. 3 versions of this function are available:

- `dx7_egratedecayperc_bpf`: BPF version (same as in the CSOUND toolkit)
- `dx7_egratedecayperc_func`: estimated mathematical equivalent of `dx7_egratedecayperc_bpf`
- `dx7_egratedecaypercf`: default (sugar for `dx7_egratedecayperc_func`)

Usage:

```
dx7envelopePercDecay : dx7_egratedecaypercf : _
```

Where:

- `dx7envelopePercDecay`: DX7 envelope decay value (0-99)

`(dx.)dx7_eglv2peakf`

DX7 envelope level to peak conversion function. 3 versions of this function are available:

- `dx7_eglv2peak_bpf`: BPF version (same as in the CSOUND toolkit)
- `dx7_eglv2peak_func`: estimated mathematical equivalent of `dx7_eglv2peak_bpf`
- `dx7_eglv2peakf`: default (sugar for `dx7_eglv2peak_func`)

Usage:

```
dx7Level : dx7_eglv2peakf : _
```

Where:

- `dx7Level`: DX7 level value (0-99)

`(dx.)dx7_velsensf`

DX7 velocity sensitivity conversion function.

Usage:

```
dx7Velocity  : dx7_velsensf : _
```

Where:

- `dx7Velocity`: DX7 level value (0-8)

`(dx.)dx7_fdbkscalef`

DX7 feedback scaling conversion function.

Usage:

```
dx7Feedback  : dx7_fdbkscalef : _
```

Where:

- `dx7Feedback`: DX7 feedback value

`(dx.)dx7_op`

DX7 Operator. Implements a phase-modulable sine wave oscillator connected to a DX7 envelope generator.

Usage:

```
dx7_op(freq,phaseMod,outLev,R1,R2,R3,R4,L1,L2,L3,L4,keyVel,rateScale,type,gain,gate) : _
```

Where:

- `freq`: frequency of the oscillator
- `phaseMod`: phase deviation (-1 - 1)
- `outLev`: preset output level (0-99)
- `R1`: preset envelope rate 1 (0-99)
- `R2`: preset envelope rate 2 (0-99)
- `R3`: preset envelope rate 3 (0-99)
- `R4`: preset envelope rate 4 (0-99)
- `L1`: preset envelope level 1 (0-99)
- `L2`: preset envelope level 2 (0-99)
- `L3`: preset envelope level 3 (0-99)
- `L4`: preset envelope level 4 (0-99)
- `keyVel`: preset key velocity sensitivity (0-99)
- `rateScale`: preset envelope rate scale
- `type`: preset operator type
- `gain`: general gain
- `gate`: trigger signal

`(dx.)dx7_algo`

DX7 algorithms. Implements the 32 DX7 algorithms (a quick Google search should give your more details on this). Each algorithm uses 6 operators.

Usage:

```
dx7_algo(algN,egR1,egR2,egR3,egR4,egL1,egL2,egL3,egL4,outLevel,keyVelSens,ampModSens,opMode
```

Where:

- `algN`: algorithm number (0-31, should be an int...)
- `egR1`: preset envelope rates 1 (a list of 6 values between 0-99)
- `egR2`: preset envelope rates 2 (a list of 6 values between 0-99)
- `egR3`: preset envelope rates 3 (a list of 6 values between 0-99)
- `egR4`: preset envelope rates 4 (a list of 6 values between 0-99)
- `egL1`: preset envelope levels 1 (a list of 6 values between 0-99)

- `egL2`: preset envelope levels 2 (a list of 6 values between 0-99)
- `egL3`: preset envelope levels 3 (a list of 6 values between 0-99)
- `egL4`: preset envelope levels 4 (a list of 6 values between 0-99)
- `outLev`: preset output levels (a list of 6 values between 0-99)
- `keyVel`: preset key velocity sensitivities (a list of 6 values between 0-99)
- `ampModSens`: preset amplitude sensitivities (a list of 6 values between 0-99)
- `opMode`: preset operator mode (a list of 6 values between 0-1)
- `opFreq`: preset operator frequencies (a list of 6 values between 0-99)
- `opDetune`: preset operator detuning (a list of 6 values between 0-99)
- `opRateScale`: preset operator rate scale (a list of 6 values between 0-99)
- `feedback`: preset operator feedback (a list of 6 values between 0-99)
- `lfoDelay`: preset LFO delay (a list of 6 values between 0-99)
- `lfoDepth`: preset LFO depth (a list of 6 values between 0-99)
- `lfoSpeed`: preset LFO speed (a list of 6 values between 0-99)
- `freq`: fundamental frequency
- `gain`: general gain
- `gate`: trigger signal

`(dx.)dx7_ui`

Generic DX7 function where all parameters are controllable using UI elements. The `master-with-mute` branch must be used for this function to work... This function is MIDI-compatible.

Usage

```
dx7_ui : _
```

---

envelopes.lib

This library contains a collection of envelope generators. Its official prefix is `en`.

Functions Reference

`(en.)smoothEnvelope`

An envelope with an exponential attack and release. `smoothEnvelope` is a standard Faust function.

Usage

```
smoothEnvelope(ar,t) : _
```

- `ar`: attack and release duration (s)

- `t`: trigger signal (attack is triggered when `t>0`, release is triggered when `t=0`)

`(en.)ar`

AR (Attack, Release) envelope generator (useful to create percussion envelopes). `ar` is a standard Faust function.

Usage

```
ar(at,rt,t) : _
```

Where:

- `at`: attack (sec)
- `rt`: release (sec)
- `t`: trigger signal (attack is triggered when `t>0`, release is triggered when `t=0`)

`(en.)arfe`

ARFE (Attack and Release-to-Final-value Exponentially) envelope generator. Approximately equal to smoothEnvelope(Attack/6.91) when Attack == Release.

Usage

```
arfe(a,r,f,t) : _
```

Where:

- `a`, `r`: attack (sec), release (sec)
- `f`: final value to approach upon release (such as 0)
- `t`: trigger signal (attack is triggered when `t>0`, release is triggered when `t=0`)

`(en.)are`

ARE (Attack, Release) envelope generator with Exponential segments. Approximately equal to smoothEnvelope(Attack/6.91) when Attack == Release.

Usage

```
are(a,r,t) : _
```

Where:

- `a`: attack (sec)
- `r`: release (sec)
- `t`: trigger signal (attack is triggered when `t>0`, release is triggered when `t=0`)

`(en.)asr`

ASR (Attack, Sustain, Release) envelope generator. `asr` is a standard Faust function.

Usage

```
asr(at,sl,rt,t) : _
```

Where:

- `at`: attack (sec)
- `sl`: sustain level (between 0..1)
- `r`: release (sec)
- `t`: trigger signal (attack is triggered when `t>0`, release is triggered when `t=0`)

`(en.)adsr`

ADSR (Attack, Decay, Sustain, Release) envelope generator. `adsr` is a standard Faust function.

Usage

```
adsr(at,dt,sl,rt,gate) : _
```

Where:

- `at`: attack time (sec)
- `dt`: decay time (sec)
- `sl`: sustain level (between 0..1)
- `rt`: release time (sec)
- `gate`: trigger signal (attack is triggered when `gate>0`, release is triggered when `gate=0`)

`(en.)adsre`

ADSRE (Attack, Decay, Sustain, Release) envelope generator with Exponential segments.

Usage

`adsre(a,d,s,r,g) : _`

Where:

- `a`: attack (sec)
- `d`: decay (sec)
- `s`: sustain (fraction of `t`: 0-1)
- `r`: release (sec)
- `t`: trigger signal (attack is triggered when `t>0`, release is triggered when `t=0`)

`(en.)asre`

ASRE (Attack, Sustain, Release) envelope generator with Exponential segments.

Usage

`asre(a,s,r,g) : _`

Where:

- `a`: attack (sec)
- `s`: sustain (fraction of `t`: 0-1)
- `r`: release (sec)
- `t`: trigger signal (attack is triggered when `t>0`, release is triggered when `t=0`)

`(en.)dx7envelope`

DX7 operator envelope generator with 4 independent rates and levels. It is essentially a 4 points BPF.

Usage

`dx7_envelope(R1,R2,R3,R4,L1,L2,L3,L4,t) : _`

Where:

- `RN`: rates in seconds
- `LN`: levels (0-1)
- `t`: trigger signal

---

filters.lib

Faust Filters library; Its official prefix is `fi`.

The Filters library is organized into 18 sections:

- Basic Filters
- Comb Filters
- Direct-Form Digital Filter Sections
- Direct-Form Second-Order Biquad Sections
- Ladder/Lattice Digital Filters
- Useful Special Cases
- Ladder/Lattice Allpass Filters
- Digital Filter Sections Specified as Analog Filter Sections
- Simple Resonator Filters
- Butterworth Lowpass/Highpass Filters
- Special Filter-Bank Delay-Equalizing Allpass Filters
- Elliptic (Cauer) Lowpass Filters
- Elliptic Highpass Filters
- Butterworth Bandpass/Bandstop Filters
- Elliptic Bandpass Filters
- Parametric Equalizers (Shelf, Peaking)
- Mth-Octave Filter-Banks
- Arbitrary-Crossover Filter-Banks and Spectrum Analyzers

For more information, see ../documentation/library.pdf

Basic Filters

`(fi.)zero`

One zero filter. Difference equation: $y(n) = x(n) - zx(n-1)$.

Usage

```
_ : zero(z) : _
```

Where:

- `z`: location of zero along real axis in z-plane

Reference

https://ccrma.stanford.edu/~jos/filters/One_Zero.html

`(fi.)pole`

One pole filter. Could also be called a "leaky integrator". Difference equation: $y(n) = x(n) + py(n-1)$.

Usage

```
_ : pole(p) : _
```

Where:

- `p`: pole location = feedback coefficient

Reference

https://ccrma.stanford.edu/~jos/filters/One_Pole.html

`(fi.)integrator`

Same as `pole(1)` [implemented separately for block-diagram clarity].

`(fi.)dcblockerat`

DC blocker with configurable break frequency. The amplitude response is substantially flat above fb, and sloped at about +6 dB/octave below fb. Derived from the analog transfer function $H(s) = \frac{s}{(s+2\pi fb)}$ by the low-frequency-matching bilinear transform method (i.e., the standard frequency-scaling constant 2*SR).

Usage

```
_ : dcblockerat(fb) : _
```

Where:

- `fb`: "break frequency" in Hz, i.e., -3 dB gain frequency.

Reference

https://ccrma.stanford.edu/~jos/pasp/Bilinear_Transformation.html

---

`(fi.)dcblocker`

DC blocker. Default dc blocker has -3dB point near 35 Hz (at 44.1 kHz) and high-frequency gain near 1.0025 (due to no scaling). `dcblocker` is as standard Faust function.

Usage

```
_ : dcblocker : _
```

Comb Filters

`(fi.)ff_comb`

Feed-Forward Comb Filter. Note that `ff_comb` requires integer delays (uses `delay` internally). `ff_comb` is a standard Faust function.

Usage

```
_ : ff_comb(maxdel,intdel,b0,bM) : _
```

Where:

- `maxdel`: maximum delay (a power of 2)
- `intdel`: current (integer) comb-filter delay between 0 and maxdel
- `del`: current (float) comb-filter delay between 0 and maxdel
- `b0`: gain applied to delay-line input
- `bM`: gain applied to delay-line output and then summed with input

Reference

https://ccrma.stanford.edu/~jos/pasp/Feedforward_Comb_Filters.html

---

`(fi.)ff_fcomb`

Feed-Forward Comb Filter. Note that `ff_fcomb` takes floating-point delays (uses `fdelay` internally). `ff_fcomb` is a standard Faust function.

Usage

```
_ : ff_fcomb(maxdel,del,b0,bM) : _
```

Where:

- `maxdel`: maximum delay (a power of 2)
- `intdel`: current (integer) comb-filter delay between 0 and maxdel
- `del`: current (float) comb-filter delay between 0 and maxdel
- `b0`: gain applied to delay-line input
- `bM`: gain applied to delay-line output and then summed with input

Reference

https://ccrma.stanford.edu/~jos/pasp/Feedforward_Comb_Filters.html

---

`(fi.)ffcombfilter`

Typical special case of `ff_comb()` where: `b0 = 1`.

`(fi.)fb_comb`

Feed-Back Comb Filter (integer delay).

Usage

```
_ : fb_comb(maxdel,intdel,b0,aN) : _
```

Where:

- `maxdel`: maximum delay (a power of 2)
- `intdel`: current (integer) comb-filter delay between 0 and maxdel
- `del`: current (float) comb-filter delay between 0 and maxdel
- `b0`: gain applied to delay-line input and forwarded to output
- `aN`: minus the gain applied to delay-line output before summing with the input and feeding to the delay line

Reference

https://ccrma.stanford.edu/~jos/pasp/Feedback_Comb_Filters.html

---

`(fi.)fb_fcomb`

Feed-Back Comb Filter (floating point delay).

Usage

```
_ : fb_fcomb(maxdel,del,b0,aN) : _
```

Where:

- `maxdel`: maximum delay (a power of 2)
- `intdel`: current (integer) comb-filter delay between 0 and maxdel
- `del`: current (float) comb-filter delay between 0 and maxdel
- `b0`: gain applied to delay-line input and forwarded to output
- `aN`: minus the gain applied to delay-line output before summing with the input and feeding to the delay line

Reference

https://ccrma.stanford.edu/~jos/pasp/Feedback_Comb_Filters.html

---

`(fi.)rev1`

Special case of `fb_comb` (`rev1(maxdel,N,g)`). The "rev1 section" dates back to the 1960s in computer-music reverberation. See the `jcrev` and `brassrev` in `reverbs.lib` for usage examples.

`(fi.)fbcombfilter` and `(fi.)ffbcombfilter`

Other special cases of Feed-Back Comb Filter.

Usage

```
_ : fbcombfilter(maxdel,intdel,g) : _
_ : ffbcombfilter(maxdel,del,g) : _
```

Where:

- `maxdel`: maximum delay (a power of 2)
- `intdel`: current (integer) comb-filter delay between 0 and maxdel
- `del`: current (float) comb-filter delay between 0 and maxdel
- `g`: feedback gain

Reference

https://ccrma.stanford.edu/~jos/pasp/Feedback_Comb_Filters.html

---

`(fi.)allpass_comb`

Schroeder Allpass Comb Filter. Note that

```
allpass_comb(maxlen,len,aN) = ff_comb(maxlen,len,aN,1) : fb_comb(maxlen,len-1,1,aN);
```

which is a direct-form-1 implementation, requiring two delay lines. The implementation here is direct-form-2 requiring only one delay line.

Usage

```
_ : allpass_comb(maxdel,intdel,aN) : _
```

Where:

- `maxdel`: maximum delay (a power of 2)
- `intdel`: current (integer) comb-filter delay between 0 and maxdel
- `del`: current (float) comb-filter delay between 0 and maxdel
- `aN`: minus the feedback gain

References

- https://ccrma.stanford.edu/~jos/pasp/Allpass_Two_Combs.html
- https://ccrma.stanford.edu/~jos/pasp/Schroeder_Allpass_Sections.html
- https://ccrma.stanford.edu/~jos/filters/Four_Direct_Forms.html

---

`(fi.)allpass_fcomb`

Schroeder Allpass Comb Filter. Note that

```
allpass_comb(maxlen,len,aN) = ff_comb(maxlen,len,aN,1) : fb_comb(maxlen,len-1,1,aN);
```

which is a direct-form-1 implementation, requiring two delay lines. The implementation here is direct-form-2 requiring only one delay line.

`allpass_fcomb` is a standard Faust library.

Usage

```
_ : allpass_comb(maxdel,intdel,aN) : _
_ : allpass_fcomb(maxdel,del,aN) : _
```

Where:

- `maxdel`: maximum delay (a power of 2)
- `intdel`: current (float) comb-filter delay between 0 and maxdel
- `del`: current (float) comb-filter delay between 0 and maxdel
- `aN`: minus the feedback gain

References

- https://ccrma.stanford.edu/~jos/pasp/Allpass_Two_Combs.html
- https://ccrma.stanford.edu/~jos/pasp/Schroeder_Allpass_Sections.html
- https://ccrma.stanford.edu/~jos/filters/Four_Direct_Forms.html

---

`(fi.)rev2`

Special case of `allpass_comb` (`rev2(maxlen,len,g)`). The "rev2 section" dates back to the 1960s in computer-music reverberation. See the `jcrev` and `brassrev` in `reverbs.lib` for usage examples.

`(fi.)allpass_fcomb5` and `(fi.)allpass_fcomb1a`

Same as `allpass_fcomb` but use `fdelay5` and `fdelay1a` internally (Interpolation helps - look at an fft of faust2octave on

`1-1' <: allpass_fcomb(1024,10.5,0.95), allpass_fcomb5(1024,10.5,0.95);`).

Direct-Form Digital Filter Sections

`(fi.)iir`

Nth-order Infinite-Impulse-Response (IIR) digital filter, implemented in terms of the Transfer-Function (TF) coefficients. Such filter structures are termed "direct form".

`iir` is a standard Faust function.

Usage

```
_ : iir(bcoeffs,acoeffs) : _
```

Where:

- `order`: filter order (int) = max(#poles,#zeros)
- `bcoeffs`: (b0,b1,...,b_order) = TF numerator coefficients
- `acoeffs`: (a1,...,a_order) = TF denominator coeffs (a0=1)

Reference

https://ccrma.stanford.edu/~jos/filters/Four_Direct_Forms.html

---

`(fi.)fir`

FIR filter (convolution of FIR filter coefficients with a signal)

Usage

`_ : fir(bv) : _`

`fir` is standard Faust function.

Where:

- `bv` = b0,b1,...,bn is a parallel bank of coefficient signals.

Note

`bv` is processed using pattern-matching at compile time, so it must have this normal form (parallel signals).

Example

Smoothing white noise with a five-point moving average:

```
bv = .2,.2,.2,.2,.2;
process = noise : fir(bv);
```

Equivalent (note double parens):

```
process = noise : fir((.2,.2,.2,.2,.2));
```

`(fi.)conv` and `(fi.)convN`

Convolution of input signal with given coefficients.

Usage

```
_ : conv((k1,k2,k3,...,kN)) : _; // Argument = one signal bank
_ : convN(N,(k1,k2,k3,...)) : _; // Useful when N < count((k1,...))
```

`(fi.)tf1`, `(fi.)tf2` and `(fi.)tf3`

tfN = N'th-order direct-form digital filter.

Usage

```
_ : tf1(b0,b1,a1) : _
_ : tf2(b0,b1,b2,a1,a2) : _
_ : tf3(b0,b1,b2,b3,a1,a2,a3) : _
```

Where:

- `a`: the poles
- `b`: the zeros

Reference

https://ccrma.stanford.edu/~jos/fp/Direct_Form_I.html

`(fi.)notchw`

Simple notch filter based on a biquad (`tf2`). `notchw` is a standard Faust function.

Usage:

```
_ : notchw(width,freq) : _
```

Where:

- `width`: "notch width" in Hz (approximate)
- `freq`: "notch frequency" in Hz

Reference

https://ccrma.stanford.edu/~jos/pasp/Phasing_2nd_Order_Allpass_Filters.html

---

Direct-Form Second-Order Biquad Sections

Direct-Form Second-Order Biquad Sections

Reference

https://ccrma.stanford.edu/~jos/filters/Four_Direct_Forms.html

`(fi.)tf21`, `(fi.)tf22`, `(fi.)tf22t` and `(fi.)tf21t`

tfN = N'th-order direct-form digital filter where:

- `tf21` is tf2, direct-form 1
- `tf22` is tf2, direct-form 2
- `tf22t` is tf2, direct-form 2 transposed
- `tf21t` is tf2, direct-form 1 transposed

Usage

```
_ : tf21(b0,b1,b2,a1,a2) : _
_ : tf22(b0,b1,b2,a1,a2) : _
_ : tf22t(b0,b1,b2,a1,a2) : _
_ : tf21t(b0,b1,b2,a1,a2) : _
```

Where:

- `a`: the poles
- `b`: the zeros

Reference

https://ccrma.stanford.edu/~jos/fp/Direct_Form_I.html

Ladder/Lattice Digital Filters

Ladder and lattice digital filters generally have superior numerical properties relative to direct-form digital filters. They can be derived from digital waveguide filters, which gives them a physical interpretation.

`(fi.)av2sv`

Compute reflection coefficients sv from transfer-function denominator av.

Usage

```
sv = av2sv(av)
```

Where:

- `av`: parallel signal bank `a1,...,aN`
- `sv`: parallel signal bank `s1,...,sN`

where `ro = ith` reflection coefficient, and `ai` = coefficient of `z^(-i)` in the filter transfer-function denominator `A(z)`.

Reference

https://ccrma.stanford.edu/~jos/filters/Step_Down_Procedure.html (where reflection coefficients are denoted by k rather than s).

`(fi.)bvav2nuv`

Compute lattice tap coefficients from transfer-function coefficients.

Usage

```
nuv = bvav2nuv(bv,av)
```

Where:

- `av`: parallel signal bank `a1,...,aN`
- `bv`: parallel signal bank `b0,b1,...,aN`
- `nuv`: parallel signal bank `nu1,...,nuN`

where `nui` is the i'th tap coefficient, `bi` is the coefficient of `z^(-i)` in the filter numerator, `ai` is the coefficient of `z^(-i)` in the filter denominator

---

`(fi.)iir_lat2`

Two-multiply latice IIR filter of arbitrary order.

Usage

```
_ : iir_lat2(bv,av) : _
```

Where:

- bv: zeros as a bank of parallel signals
- av: poles as a bank of parallel signals

(fi.)allpassnt

Two-multiply lattice allpass (nested order-1 direct-form-ii allpasses).

Usage

```
_ : allpassnt(n,sv) : _
```

Where:

- `n`: the order of the filter
- `sv`: the reflection coefficients (-1 1)

(fi.)iir_kl

Kelly-Lochbaum ladder IIR filter of arbitrary order.

Usage

```
_ : iir_kl(bv,av) : _
```

Where:

- bv: zeros as a bank of parallel signals
- av: poles as a bank of parallel signals

(fi.)allpassnklt

Kelly-Lochbaum ladder allpass.

Usage:

```
_ : allpassklt(n,sv) : _
```

Where:

- `n`: the order of the filter
- `sv`: the reflection coefficients (-1 1)

`(fi.)iir_lat1`

One-multiply latice IIR filter of arbitrary order.

Usage

```
_ : iir_lat1(bv,av) : _
```

Where:

- bv: zeros as a bank of parallel signals
- av: poles as a bank of parallel signals

`(fi.)allpassn1mt`

One-multiply lattice allpass with tap lines.

Usage

```
_ : allpassn1mt(n,sv) : _
```

Where:

- `n`: the order of the filter
- `sv`: the reflection coefficients (-1 1)

`(fi.)iir_nl`

Normalized ladder filter of arbitrary order.

Usage

```
_ : iir_nl(bv,av) : _
```

Where:

- bv: zeros as a bank of parallel signals
- av: poles as a bank of parallel signals

References

- J. D. Markel and A. H. Gray, Linear Prediction of Speech, New York: Springer Verlag, 1976.
- https://ccrma.stanford.edu/~jos/pasp/Normalized_Scattering_Junctions.html

---

`(fi.)allpassnnlt`

Normalized ladder allpass filter of arbitrary order.

Usage:

```
_ : allpassnnlt(n,sv) : _
```

Where:

- `n`: the order of the filter
- `sv`: the reflection coefficients (-1,1)

References

- J. D. Markel and A. H. Gray, Linear Prediction of Speech, New York: Springer Verlag, 1976.
- https://ccrma.stanford.edu/~jos/pasp/Normalized_Scattering_Junctions.html

---

Useful Special Cases

`(fi.)tf2np`

Biquad based on a stable second-order Normalized Ladder Filter (more robust to modulation than `tf2` and protected against instability).

Usage

```
_ : tf2np(b0,b1,b2,a1,a2) : _
```

Where:

- `a`: the poles
- `b`: the zeros

`(fi.)wgr`

Second-order transformer-normalized digital waveguide resonator.

Usage

```
_ : wgr(f,r) : _
```

Where:

- `f`: resonance frequency (Hz)

- `r`: loss factor for exponential decay (set to 1 to make a numerically stable oscillator)

References

- https://ccrma.stanford.edu/~jos/pasp/Power_Normalized_Waveguide_Filters.html
- https://ccrma.stanford.edu/~jos/pasp/Digital_Waveguide_Oscillator.html

---

`(fi.)nlf2`

Second order normalized digital waveguide resonator.

Usage

```
_ : nlf2(f,r) : _
```

Where:

- `f`: resonance frequency (Hz)
- `r`: loss factor for exponential decay (set to 1 to make a sinusoidal oscillator)

Reference

https://ccrma.stanford.edu/~jos/pasp/Power_Normalized_Waveguide_Filters.html

---

`(fi.)apnl`

Passive Nonlinear Allpass based on Pierce switching springs idea. Switch between allpass coefficient `a1` and `a2` at signal zero crossings.

Usage

```
_ : apnl(a1,a2) : _
```

Where:

- `a1` and `a2`: allpass coefficients

Reference

- "A Passive Nonlinear Digital Filter Design ..." by John R. Pierce and Scott A. Van Duyne, JASA, vol. 101, no. 2, pp. 1120-1126, 1997

Ladder/Lattice Allpass Filters

An allpass filter has gain 1 at every frequency, but variable phase. Ladder/lattice allpass filters are specified by reflection coefficients. They are defined here as nested allpass filters, hence the names allpassn*.

References

- https://ccrma.stanford.edu/~jos/pasp/Conventional_Ladder_Filters.html
- https://ccrma.stanford.edu/~jos/pasp/Nested_Allpass_Filters.html
- Linear Prediction of Speech, Markel and Gray, Springer Verlag, 1976

`(fi.)allpassn`

Two-multiply lattice - each section is two multiply-adds.

Usage:

```
_ : allpassn(n,sv) : _
```

Where:

- `n`: the order of the filter
- `sv`: the reflection coefficients (-1 1)

References

- J. O. Smith and R. Michon, "Nonlinear Allpass Ladder Filters in FAUST", in Proceedings of the 14th International Conference on Digital Audio Effects (DAFx-11), Paris, France, September 19-23, 2011.

`(fi.)allpassnn`

Normalized form - four multiplies and two adds per section, but coefficients can be time varying and nonlinear without "parametric amplification" (modulation of signal energy).

Usage:

```
_ : allpassnn(n,tv) : _
```

Where:

- `n`: the order of the filter
- `tv`: the reflection coefficients (-PI PI)

`(fi.)allpasskl`

Kelly-Lochbaum form - four multiplies and two adds per section, but all signals have an immediate physical interpretation as traveling pressure waves, etc.

Usage:

```
_ : allpassnkl(n,sv) : _
```

Where:

- `n`: the order of the filter
- `sv`: the reflection coefficients (-1 1)

`(fi.)allpass1m`

One-multiply form - one multiply and three adds per section. Normally the most efficient in special-purpose hardware.

Usage:

```
_ : allpassn1m(n,sv) : _
```

Where:

- `n`: the order of the filter
- `sv`: the reflection coefficients (-1 1)

Digital Filter Sections Specified as Analog Filter Sections

`(fi.)tf2s` and `(fi.)tf2snp`

Second-order direct-form digital filter, specified by ANALOG transfer-function polynomials B(s)/A(s), and a frequency-scaling parameter. Digitization via the bilinear transform is built in.

Usage

```
_ : tf2s(b2,b1,b0,a1,a0,w1) : _
```

Where:

```
        b2 s^2 + b1 s + b0
H(s) = --------------------
           s^2 + a1 s + a0
```

and `w1` is the desired digital frequency (in radians/second) corresponding to analog frequency 1 rad/sec (i.e., `s = j`).

Example

A second-order ANALOG Butterworth lowpass filter, normalized to have cutoff frequency at 1 rad/sec, has transfer function

```
             1
H(s) = ------------------
        s^2 + a1 s + 1
```

where `a1 = sqrt(2)`. Therefore, a DIGITAL Butterworth lowpass cutting off at `SR/4` is specified as `tf2s(0,0,1,sqrt(2),1,PI*SR/2);`

Method

Bilinear transform scaled for exact mapping of w1.

Reference

https://ccrma.stanford.edu/~jos/pasp/Bilinear_Transformation.html

---

`(fi.)tf3slf`

Analogous to tf2s above, but third order, and using the typical low-frequency-matching bilinear-transform constant 2/T ("lf" series) instead of the specific-frequency-matching value used in tf2s and tf1s. Note the lack of a "w1" argument.

Usage

```
_ : tf3slf(b3,b2,b1,b0,a3,a2,a1,a0) : _
```

`(fi.)tf1s`

First-order direct-form digital filter, specified by ANALOG transfer-function polynomials B(s)/A(s), and a frequency-scaling parameter.

Usage

`tf1s(b1,b0,a0,w1)`

Where:

`b1 s + b0`

H(s) = ———- s + a0

and `w1` is the desired digital frequency (in radians/second) corresponding to analog frequency 1 rad/sec (i.e., `s = j`).

Example

A first-order ANALOG Butterworth lowpass filter, normalized to have cutoff frequency at 1 rad/sec, has transfer function

$$H(s) = \frac{1}{s + 1}$$

so `b0 = a0 = 1` and `b1 = 0`. Therefore, a DIGITAL first-order Butterworth lowpass with gain -3dB at `SR/4` is specified as

```
tf1s(0,1,1,PI*SR/2); // digital half-band order 1 Butterworth
```

Method

Bilinear transform scaled for exact mapping of w1.

Reference

https://ccrma.stanford.edu/~jos/pasp/Bilinear_Transformation.html

---

`(fi.)tf2sb`

Bandpass mapping of `tf2s`: In addition to a frequency-scaling parameter `w1` (set to HALF the desired passband width in rad/sec), there is a desired center-frequency parameter wc (also in rad/s). Thus, `tf2sb` implements a fourth-order digital bandpass filter section specified by the coefficients of a second-order analog lowpass prototype section. Such sections can be combined in series for higher orders. The order of mappings is (1) frequency scaling (to set lowpass cutoff w1), (2) bandpass mapping to wc, then (3) the bilinear transform, with the usual scale parameter `2*SR`. Algebra carried out in maxima and pasted here.

Usage

```
_ : tf2sb(b2,b1,b0,a1,a0,w1,wc) : _
```

`(fi.)tf1sb`

First-to-second-order lowpass-to-bandpass section mapping, analogous to tf2sb above.

Usage

```
_ : tf1sb(b1,b0,a0,w1,wc) : _
```

Simple Resonator Filters

(fi.)resonlp

Simple resonant lowpass filter based on `tf2s` (virtual analog). `resonlp` is a standard Faust function.

Usage

```
_ : resonlp(fc,Q,gain) : _
_ : resonhp(fc,Q,gain) : _
_ : resonbp(fc,Q,gain) : _
```

Where:

- `fc`: center frequency (Hz)
- `Q`: q
- `gain`: gain (0-1)

(fi.)resonhp

Simple resonant highpass filters based on `tf2s` (virtual analog). `resonhp` is a standard Faust function.

Usage

```
_ : resonlp(fc,Q,gain) : _
_ : resonhp(fc,Q,gain) : _
_ : resonbp(fc,Q,gain) : _
```

Where:

- `fc`: center frequency (Hz)
- `Q`: q
- `gain`: gain (0-1)

(fi.)resonbp

Simple resonant bandpass filters based on `tf2s` (virtual analog). `resonbp` is a standard Faust function.

Usage

```
_ : resonlp(fc,Q,gain) : _
_ : resonhp(fc,Q,gain) : _
_ : resonbp(fc,Q,gain) : _
```

Where:

- `fc`: center frequency (Hz)
- `Q`: q
- `gain`: gain (0-1)

Butterworth Lowpass/Highpass Filters

`(fi.)lowpass`

Nth-order Butterworth lowpass filter. `lowpass` is a standard Faust function.

Usage

```
_ : lowpass(N,fc) : _
```

Where:

- `N`: filter order (number of poles) [nonnegative constant integer]
- `fc`: desired cut-off frequency (-3dB frequency) in Hz

References

- https://ccrma.stanford.edu/~jos/filters/Butterworth_Lowpass_Design.html
- `butter` function in Octave `("[z,p,g] = butter(N,1,'s');")`

`(fi.)highpass`

Nth-order Butterworth highpass filters. `highpass` is a standard Faust function.

Usage

```
_ : highpass(N,fc) : _
```

Where:

- `N`: filter order (number of poles) [nonnegative constant integer]
- `fc`: desired cut-off frequency (-3dB frequency) in Hz

References

- https://ccrma.stanford.edu/~jos/filters/Butterworth_Lowpass_Design.html
- `butter` function in Octave `("[z,p,g] = butter(N,1,'s');")`

`(fi.)lowpass0_highpass1`

Special Filter-Bank Delay-Equalizing Allpass Filters

These special allpass filters are needed by filterbank et al. below. They are equivalent to (`lowpass(N,fc)` +|- `highpass(N,fc))/2`, but with canceling pole-zero pairs removed (which occurs for odd N).

`(fi.)lowpass_plus|minus_highpass`

Elliptic (Cauer) Lowpass Filters

Elliptic (Cauer) Lowpass Filters

References

- <http://en.wikipedia.org/wiki/Elliptic_filter
- functions `ncauer` and `ellip` in Octave

`(fi.)lowpass3e`

Third-order Elliptic (Cauer) lowpass filter.

Usage

```
_ : lowpass3e(fc) : _
```

Where:

- `fc`: -3dB frequency in Hz

Design

For spectral band-slice level display (see `octave_analyzer3e`):

```
[z,p,g] = ncauer(Rp,Rs,3);  % analog zeros, poles, and gain, where
Rp = 60  % dB ripple in stopband
Rs = 0.2 % dB ripple in passband
```

`(fi.)lowpass6e`

Sixth-order Elliptic/Cauer lowpass filter.

Usage

```
_ : lowpass6e(fc) : _
```

Where:

- `fc`: -3dB frequency in Hz

Design

For spectral band-slice level display (see octave_analyzer6e):

```
[z,p,g] = ncauer(Rp,Rs,6);  % analog zeros, poles, and gain, where
 Rp = 80  % dB ripple in stopband
 Rs = 0.2 % dB ripple in passband
```

Elliptic Highpass Filters

`(fi.)highpass3e`

Third-order Elliptic (Cauer) highpass filter. Inversion of `lowpass3e` wrt unit circle in s plane (s <- 1/s)

Usage

```
_ : highpass3e(fc) : _
```

Where:

- `fc`: -3dB frequency in Hz

`(fi.)highpass6e`

Sixth-order Elliptic/Cauer highpass filter. Inversion of lowpass3e wrt unit circle in s plane (s <- 1/s)

Usage

```
_ : highpass6e(fc) : _
```

Where:

- `fc`: -3dB frequency in Hz

Butterworth Bandpass/Bandstop Filters

`(fi.)bandpass`

Order 2*Nh Butterworth bandpass filter made using the transformation `s <- s + wc^2/s` on `lowpass(Nh)`, where `wc` is the desired bandpass center frequency. The `lowpass(Nh)` cutoff `w1` is half the desired bandpass width. `bandpass` is a standard Faust function.

Usage

```
_ : bandpass(Nh,fl,fu) : _
```

Where:

- `Nh`: HALF the desired bandpass order (which is therefore even)
- `fl`: lower -3dB frequency in Hz
- `fu`: upper -3dB frequency in Hz Thus, the passband width is `fu-fl`, and its center frequency is `(fl+fu)/2`.

Reference

http://cnx.org/content/m16913/latest/

`(fi.)bandstop`

Order 2*Nh Butterworth bandstop filter made using the transformation `s <- s + wc^2/s` on `highpass(Nh)`, where `wc` is the desired bandpass center frequency. The `highpass(Nh)` cutoff `w1` is half the desired bandpass width. `bandstop` is a standard Faust function.

Usage

```
_ : bandstop(Nh,fl,fu) : _
```

Where:

- `Nh`: HALF the desired bandstop order (which is therefore even)
- `fl`: lower -3dB frequency in Hz
- `fu`: upper -3dB frequency in Hz Thus, the passband (stopband) width is `fu-fl`, and its center frequency is `(fl+fu)/2`.

Reference

http://cnx.org/content/m16913/latest/

Elliptic Bandpass Filters

`(fi.)bandpass6e`

Order 12 elliptic bandpass filter analogous to `bandpass(6)`.

`(fi.)bandpass12e`

Order 24 elliptic bandpass filter analogous to `bandpass(6)`.

`(fi.)pospass`

Positive-Pass Filter (single-side-band filter)

Usage

```
_ : pospass(N,fc) : _,_
```

where

- `N`: filter order (Butterworth bandpass for positive frequencies).
- `fc`: lower bandpass cutoff frequency in Hz.
  - Highpass cutoff frequency at ma.SR/2 - fc Hz.

Example

- See dm.pospass_demo
- Look at frequency response:

Method

A filter passing only positive frequencies can be made from a half-band lowpass by modulating it up to the positive-frequency range. Equivalently, down-modulate the input signal using a complex sinusoid at -SR/4 Hz, lowpass it with a half-band filter, and modulate back up by SR/4 Hz. In Faust/math notation: pospass(N) $= *(e^{-j\frac{\pi}{2}n})$: lowpass(N,SR/4) : $*(e^{j\frac{\pi}{2}n})$

An approximation to the Hilbert transform is given by the imaginary output signal:

```
hilbert(N) = pospass(N) : !,*(2);
```

References

- https://ccrma.stanford.edu/~jos/mdft/Analytic_Signals_Hilbert_Transform.html
- https://ccrma.stanford.edu/~jos/sasp/Comparison_Optimal_Chebyshev_FIR_I.html
- https://ccrma.stanford.edu/~jos/sasp/Hilbert_Transform.html

---

Parametric Equalizers (Shelf, Peaking)

Parametric Equalizers (Shelf, Peaking).

References

- http://en.wikipedia.org/wiki/Equalization
- http://www.musicdsp.org/files/Audio-EQ-Cookbook.txt
- Digital Audio Signal Processing, Udo Zolzer, Wiley, 1999, p. 124
- https://ccrma.stanford.edu/~jos/filters/Low_High_Shelving_Filters.html>
- https://ccrma.stanford.edu/~jos/filters/Peaking_Equalizers.html>
- maxmsp.lib in the Faust distribution
- bandfilter.dsp in the faust2pd distribution

`(fi.)low_shelf`

First-order "low shelf" filter (gain boost|cut between dc and some frequency) `low_shelf` is a standard Faust function.

Usage

```
_ : lowshelf(N,L0,fx) : _
_ : low_shelf(L0,fx) : _ // default case (order 3)
_ : lowshelf_other_freq(N,L0,fx) : _
```

Where: * `N`: filter order 1, 3, 5, ... (odd only). (default should be 3) * `L0`: desired level (dB) between dc and fx (boost `L0>0` or cut `L0<0`) * `fx`: -3dB frequency of lowpass band (`L0>0`) or upper band (`L0<0`) (see "SHELF SHAPE" below).

The gain at SR/2 is constrained to be 1. The generalization to arbitrary odd orders is based on the well known fact that odd-order Butterworth band-splits are allpass-complementary (see filterbank documentation below for references).

Shelf Shape

The magnitude frequency response is approximately piecewise-linear on a log-log plot ("BODE PLOT"). The Bode "stick diagram" approximation L(lf) is easy to state in dB versus dB-frequency lf = dB(f):

- L0 > 0:
- L(lf) = L0, f between 0 and fx = 1st corner frequency;
- L(lf) = L0 - N * (lf - lfx), f between fx and f2 = 2nd corner frequency;
- L(lf) = 0, lf > lf2.
- lf2 = lfx + L0/N = dB-frequency at which level gets back to 0 dB.
- L0 < 0:
- L(lf) = L0, f between 0 and f1 = 1st corner frequency;
- L(lf) = - N * (lfx - lf), f between f1 and lfx = 2nd corner frequency;
- L(lf) = 0, lf > lfx.
- lf1 = lfx + L0/N = dB-frequency at which level goes up from L0.

See `lowshelf_other_freq`.

(fi.)high_shelf

First-order "high shelf" filter (gain boost|cut above some frequency). `high_shelf` is a standard Faust function.

Usage

```
_ : highshelf(N,Lpi,fx) : _
_ : high_shelf(L0,fx) : _ // default case (order 3)
_ : highshelf_other_freq(N,Lpi,fx) : _
```

Where:

- `N`: filter order 1, 3, 5, ... (odd only).
- `Lpi`: desired level (dB) between fx and SR/2 (boost Lpi>0 or cut Lpi<0)
- `fx`: -3dB frequency of highpass band (L0>0) or lower band (L0<0) (Use highshelf_other_freq() below to find the other one.)

The gain at dc is constrained to be 1. See `lowshelf` documentation above for more details on shelf shape.

(fi.)peak_eq

Second order "peaking equalizer" section (gain boost or cut near some frequency) Also called a "parametric equalizer" section. `peak_eq` is a standard Faust function.

Usage

```
_ : peak_eq(Lfx,fx,B) : _;
```

Where:

- `Lfx`: level (dB) at fx (boost Lfx>0 or cut Lfx<0)
- `fx`: peak frequency (Hz)
- `B`: bandwidth (B) of peak in Hz

(fi.)peak_eq_cq

Constant-Q second order peaking equalizer section.

Usage

```
_ : peak_eq_cq(Lfx,fx,Q) : _;
```

Where:

- `Lfx`: level (dB) at fx

- `fx`: boost or cut frequency (Hz)
- `Q`: "Quality factor" = fx/B where B = bandwidth of peak in Hz

---

`(fi.)peak_eq_rm`

Regalia-Mitra second order peaking equalizer section.

Usage

```
_ : peak_eq_rm(Lfx,fx,tanPiBT) : _;
```

Where:

- `Lfx`: level (dB) at fx
- `fx`: boost or cut frequency (Hz)
- `tanPiBT`: `tan(PI*B/SR)`, where B = -3dB bandwidth (Hz) when 10^(Lfx/20) = 0 ~ PI*B/SR for narrow bandwidths B

Reference

P.A. Regalia, S.K. Mitra, and P.P. Vaidyanathan, "The Digital All-Pass Filter: A Versatile Signal Processing Building Block" Proceedings of the IEEE, 76(1):19-37, Jan. 1988. (See pp. 29-30.)

`(fi.)spectral_tilt`

Spectral tilt filter, providing an arbitrary spectral rolloff factor alpha in (-1,1), where -1 corresponds to one pole (-6 dB per octave), and +1 corresponds to one zero (+6 dB per octave). In other words, alpha is the slope of the ln magnitude versus ln frequency. For a "pinking filter" (e.g., to generate 1/f noise from white noise), set alpha to -1/2.

Usage

```
_ : spectral_tilt(N,f0,bw,alpha) : _
```

Where:

- `N`: desired integer filter order (fixed at compile time)
- `f0`: lower frequency limit for desired roll-off band > 0
- `bw`: bandwidth of desired roll-off band
- `alpha`: slope of roll-off desired in nepers per neper, between -1 and 1 (ln mag / ln radian freq)

Examples

See `spectral_tilt_demo`.

Reference

J.O. Smith and H.F. Smith, "Closed Form Fractional Integration and Differentiation via Real Exponentially Spaced Pole-Zero Pairs", arXiv.org publication arXiv:1606.06154 [cs.CE], June 7, 2016, http://arxiv.org/abs/1606.06154

`(fi.)levelfilter`

Dynamic level lowpass filter. `levelfilter` is a standard Faust function.

Usage

```
_ : levelfilter(L,freq) : _
```

Where:

- `L`: desired level (in dB) at Nyquist limit (SR/2), e.g., -60
- `freq`: corner frequency (-3dB point) usually set to fundamental freq
- `N`: Number of filters in series where L = L/N

Reference

https://ccrma.stanford.edu/realsimple/faust_strings/Dynamic_Level_Lowpass_Filter.html

---

`(fi.)levelfilterN`

Dynamic level lowpass filter.

Usage

```
_ : levelfilterN(N,freq,L) : _
```

Where:

- `L`: desired level (in dB) at Nyquist limit (SR/2), e.g., -60
- `freq`: corner frequency (-3dB point) usually set to fundamental freq
- `N`: Number of filters in series where L = L/N

Reference

https://ccrma.stanford.edu/realsimple/faust_strings/Dynamic_Level_Lowpass_Filter.html

---

Mth-Octave Filter-Banks

Mth-octave filter-banks split the input signal into a bank of parallel signals, one for each spectral band. They are related to the Mth-Octave Spectrum-Analyzers in `analysis.lib`. The documentation of this library contains more details about the implementation. The parameters are:

- `M`: number of band-slices per octave (>1)
- `N`: total number of bands (>2)
- `ftop`: upper bandlimit of the Mth-octave bands (<SR/2)

In addition to the Mth-octave output signals, there is a highpass signal containing frequencies from ftop to SR/2, and a "dc band" lowpass signal containing frequencies from 0 (dc) up to the start of the Mth-octave bands. Thus, the N output signals are

```
highpass(ftop), MthOctaveBands(M,N-2,ftop), dcBand(ftop*2^(-M*(N-1)))
```

A Filter-Bank is defined here as a signal bandsplitter having the property that summing its output signals gives an allpass-filtered version of the filter-bank input signal. A more conventional term for this is an "allpass-complementary filter bank". If the allpass filter is a pure delay (and possible scaling), the filter bank is said to be a "perfect-reconstruction filter bank" (see Vaidyanathan-1993 cited below for details). A "graphic equalizer", in which band signals are scaled by gains and summed, should be based on a filter bank.

The filter-banks below are implemented as Butterworth or Elliptic spectrum-analyzers followed by delay equalizers that make them allpass-complementary.

Increasing Channel Isolation

Go to higher filter orders - see Regalia et al. or Vaidyanathan (cited below) regarding the construction of more aggressive recursive filter-banks using elliptic or Chebyshev prototype filters.

References

- "Tree-structured complementary filter banks using all-pass sections", Regalia et al., IEEE Trans. Circuits & Systems, CAS-34:1470-1484, Dec. 1987
- "Multirate Systems and Filter Banks", P. Vaidyanathan, Prentice-Hall, 1993
- Elementary filter theory: https://ccrma.stanford.edu/~jos/filters/

`(fi.)mth_octave_filterbank[n]`

Allpass-complementary filter banks based on Butterworth band-splitting. For Butterworth band-splits, the needed delay equalizer is easily found.

Usage

```
_ : mth_octave_filterbank(O,M,ftop,N) : par(i,N,_); // Oth-order
_ : mth_octave_filterbank_alt(O,M,ftop,N) : par(i,N,_); // dc-inverted version
```

Also for convenience:

```
_ : mth_octave_filterbank3(M,ftop,N) : par(i,N,_); // 3rd-order Butterworth
_ : mth_octave_filterbank5(M,ftop,N) : par(i,N,_); // 5th-order Butterworth
mth_octave_filterbank_default = mth_octave_filterbank5;
```

Where:

- `O`: order of filter used to split each frequency band into two
- `M`: number of band-slices per octave
- `ftop`: highest band-split crossover frequency (e.g., 20 kHz)
- `N`: total number of bands (including dc and Nyquist)

Arbitrary-Crossover Filter-Banks and Spectrum Analyzers

These are similar to the Mth-octave analyzers above, except that the band-split frequencies are passed explicitly as arguments.

`(fi.)filterbank`

Filter bank. `filterbank` is a standard Faust function.

Usage

```
_ : filterbank (O,freqs) : par(i,N,_); // Butterworth band-splits
```

Where:

- `O`: band-split filter order (ODD integer required for filterbank[i])
- `freqs`: (fc1,fc2,. . . ,fcNs) [in numerically ascending order], where Ns=N-1 is the number of octave band-splits (total number of bands N=Ns+1).

If frequencies are listed explicitly as arguments, enclose them in parens:

```
_ : filterbank(3,(fc1,fc2)) : _,_,_
```

`(fi.)filterbanki`

Inverted-dc filter bank.

Usage

```
_ : filterbanki(O,freqs) : par(i,N,_); // Inverted-dc version
```

Where:

- `O`: band-split filter order (ODD integer required for `filterbank[i]`)
- `freqs`: (fc1,fc2,...,fcNs) [in numerically ascending order], where Ns=N-1 is the number of octave band-splits (total number of bands N=Ns+1).

If frequencies are listed explicitly as arguments, enclose them in parens:

```
_ : filterbanki(3,(fc1,fc2)) : _,_,_
```

---

hoa.lib

Faust library for high order ambisonic. Its official prefix is `ho`.

`(ho.)encoder`

Ambisonic encoder. Encodes a signal in the circular harmonics domain depending on an order of decomposition and an angle.

Usage

```
encoder(n, x, a) : _
```

Where:

- `n`: the order
- `x`: the signal
- `a`: the angle

`(ho.)decoder`

Decodes an ambisonics sound field for a circular array of loudspeakers.

Usage

```
_ : decoder(n, p) : _
```

Where:

- `n`: the order
- `p`: the number of speakers

Note

Number of loudspeakers must be greater or equal to 2n+1. It's preferable to use 2n+2 loudspeakers.

`(ho.)decoderStereo`

Decodes an ambisonic sound field for stereophonic configuration. An "home made" ambisonic decoder for stereophonic restitution (30° - 330°) : Sound field lose energy around 180°. You should use `inPhase` optimization with ponctual sources. #### Usage

```
_ : decoderStereo(n) : _
```

Where:

- `n`: the order

Optimization Functions

Functions to weight the circular harmonics signals depending to the ambisonics optimization. It can be `basic` for no optimization, `maxRe` or `inPhase`.

`(ho.)optimBasic`

The basic optimization has no effect and should be used for a perfect circle of loudspeakers with one listener at the perfect center loudspeakers array.

Usage

```
_ : optimBasic(n) : _
```

Where:

- `n`: the order

`(ho.)optimMaxRe`

The maxRe optimization optimize energy vector. It should be used for an auditory confined in the center of the loudspeakers array.

Usage

```
_ : optimMaxRe(n) : _
```

Where:

- `n`: the order

`(ho.)optimInPhase`

The inPhase Optimization optimize energy vector and put all loudspeakers signals n phase. It should be used for an auditory.

Usage

“ optimInPhase(n) : _ “

here:

`n`: the order

`(ho.)wider`

Can be used to wide the diffusion of a localized sound. The order depending signals are weighted and appear in a logarithmic way to have linear changes.

Usage

```
_ : wider(n,w) : _
```

Where:

- `n`: the order
- `w`: the width value between 0 - 1

`(ho.)map`

It simulate the distance of the source by applying a gain on the signal and a wider processing on the soundfield.

Usage

```
map(n, x, r, a)
```

Where:

- `n`: the order
- `x`: the signal
- `r`: the radius
- `a`: the angle in radian

`(ho.)rotate`

Rotates the sound field.

Usage

```
_ : rotate(n, a) : _
```

Where:

- `n`: the order
- `a`: the angle in radian

3D functions

//--//

`(ho.)encoder3D`

Ambisonic encoder. Encodes a signal in the circular harmonics domain depending on an order of decomposition, an angle and an elevation.

Usage

```
encoder3D(n, x, a, e) : _
```

Where:

- `n`: the order
- `x`: the signal
- `a`: the angle
- `e`: the elevation

`(ho.)optimBasic3D`

The basic optimization has no effect and should be used for a perfect sphere of loudspeakers with one listener at the perfect center loudspeakers array.

Usage

```
_ : optimBasic3D(n) : _
```

Where:

- `n`: the order

`(ho.)optimMaxRe3D`

The maxRe optimization optimize energy vector. It should be used for an auditory confined in the center of the loudspeakers array.

Usage

```
_ : optimMaxRe3D(n) : _
```

Where:

- `n`: the order

`(ho.)optimInPhase3D`

The inPhase Optimization optimize energy vector and put all loudspeakers signals n phase. It should be used for an auditory.

Usage

“ optimInPhase3D(n) : _ “

here:

`n`: the order

---

interpolators.lib

A library to handle interpolation in Faust. Its official prefix is `it`.

`(it.)interpolate_linear`

Linear interpolation between 2 values

Usage

```
interpolate_linear(dv,v0,v1) : _
```

Where:

- `dv`: in the fractional value in [0..1] range
- `v0`: is the first value
- `v1`: is the second value

Reference:

https://github.com/jamoma/JamomaCore/blob/master/Foundation/library/includes/TTInterpolate.h

---

`(it.)interpolate_cosine`

Cosine interpolation between 2 values

Usage

`interpolate_cosine(dv,v0,v1) : _`

Where:

- `dv`: in the fractional value in [0..1] range
- `v0`: is the first value
- `v1`: is the second value

Reference:

https://github.com/jamoma/JamomaCore/blob/master/Foundation/library/includes/TTInterpolate.h

---

`(it.)interpolate_cubic`

Cubic interpolation between 4 values

Usage

`interpolate_cubic(dv,v0,v1,v2,v3) : _`

Where:

- `dv`: in the fractional value in [0..1] range
- `v0`: is the first value
- `v1`: is the second value
- `v2`: is the third value
- `v3`: is the fourth value

Reference:

https://www.paulinternet.nl/?page=bicubic

`(it.)interpolator_linear`

Linear interpolator for a 'gen' circuit triggered by an 'idv' input to generate values

Usage

```
interpolator_linear(gen, idv) : _,_... (equal to N = outputs(gen))
```

Where:

- `gen`: a circuit with an 'idv' reader input that produces N outputs
- `idv`: a fractional read index expressed as a float value, or a (int,frac) pair (see float.lib and double.lib)

`(it.)interpolator_cosine`

Cosine interpolator for a 'gen' circuit triggered by an 'idv' input to generate values

Usage

```
interpolator_cosine(gen, idv) : _,_... (equal to N = outputs(gen))
```

Where:

- `gen`: a circuit with an 'idv' reader input that produces N outputs
- `idv`: a fractional read index expressed as a float value, or a (int,frac) pair (see float.lib and double.lib)

`(it.)interpolator_cubic`

Cubic interpolator for a 'gen' circuit triggered by an 'idv' input to generate values

Usage

```
interpolator_cubic(gen, idv) : _,_... (equal to N = outputs(gen))
```

Where:

- `gen`: a circuit with an 'idv' reader input that produces N outputs
- `idv`: a fractional read index expressed as a float value, or a (int,frac) pair (see float.lib and double.lib)

`(it.)interpolator_select`

Generic configurable interpolator (with selector between in [0..3]). The value 3 is used for no interpolation.

Usage

```
interpolator_select(gen, idv, sel) : _,_... (equal to N = outputs(gen))
```

Where:

- `gen`: a circuit with an 'idv' reader input that produces N outputs
- `idv`: a fractional read index expressed as a float value, or a (int,frac) pair (see float.lib and double.lib)
- `sel`: an interpolation algorithm selector in [0..3] (0 = linear, 1 = cosine, 2 = cubic, 3 = nointerp)

---

maths.lib

Mathematic library for Faust. Its official prefix is `ma`.

Functions Reference

`(ma.)SR`

Current sampling rate. Constant during program execution.

Usage

```
SR : _
```

`(ma.)BS`

Current block-size. Can change during the execution.

Usage

```
BS : _
```

`(ma.)PI`

Constant PI in double precision.

Usage

```
PI : _
```

(ma.)EPSILON

Constant EPSILON in simple/double/quad precision.

Usage

```
EPSILON : _
```

(ma.)MIN

Constant MIN in simple/double/quad precision (minimal positive value).

Usage

```
MIN : _
```

(ma.)INFINITY

Constant INFINITY in simple/double/quad precision (maximal positive value).

Usage

```
INFINITY : _
```

(ma.)FTZ

Flush to zero: force samples under the "maximum subnormal number" to be zero. Usually not needed in C++ because the architecture file take care of this, but can be useful in JavaScript for instance.

Usage

```
_ : ftz : _
```

See : http://docs.oracle.com/cd/E19957-01/806-3568/ncg_math.html

---

(ma.)neg

Invert the sign (-x) of a signal.

Usage

```
_ : neg : _
```

(ma.)sub(x,y)

Subtract `x` and `y`.

(ma.)inv

Compute the inverse (1/x) of the input signal.

Usage

```
_ : inv : _
```

(ma.)cbrt

Computes the cube root of of the input signal.

Usage

```
_ : cbrt : _
```

(ma.)hypot

Computes the euclidian distance of the two input signals sqrt(xx+yy) without undue overflow or underflow.

Usage

```
_,_ : hypot : _
```

(ma.)ldexp

Takes two input signals: x and n, and multiplies x by 2 to the power n.

Usage

```
_,_ : ldexp : _
```

`(ma.)scalb`

Takes two input signals: x and n, and multiplies x by 2 to the power n.

Usage

```
_,_ : scalb : _
```

`(ma.)log1p`

Computes log(1 + x) without undue loss of accuracy when x is nearly zero.

Usage

```
_ : log1p : _
```

`(ma.)logb`

Return exponent of the input signal as a floating-point number.

Usage

```
_ : logb : _
```

`(ma.)ilogb`

Return exponent of the input signal as an integer number.

Usage

```
_ : ilogb : _
```

`(ma.)log2`

Returns the base 2 logarithm of x.

Usage

```
_ : log2 : _
```

(ma.)expm1

Return exponent of the input signal minus 1 with better precision.

Usage

```
_ : expm1 : _
```

(ma.)acosh

Computes the principle value of the inverse hyperbolic cosine of the input signal.

Usage

```
_ : acosh : _
```

(ma.)asinh

Computes the inverse hyperbolic sine of the input signal.

Usage

```
_ : asinh : _
```

(ma.)atanh

Computes the inverse hyperbolic tangent of the input signal.

Usage

```
_ : atanh : _
```

(ma.)sinh

Computes the hyperbolic sine of the input signal.

Usage

```
_ : sinh : _
```

(ma.)cosh

Computes the hyperbolic cosine of the input signal.

Usage

```
_ : cosh : _
```

(ma.)tanh

Computes the hyperbolic tangent of the input signal.

Usage

```
_ : tanh : _
```

(ma.)erf

Computes the error function of the input signal.

Usage

```
_ : erf : _
```

(ma.)erfc

Computes the complementary error function of the input signal.

Usage

```
_ : erfc : _
```

(ma.)gamma

Computes the gamma function of the input signal.

Usage

```
_ : gamma : _
```

(ma.)lgamma

Calculates the natural logorithm of the absolute value of the gamma function of the input signal.

Usage

```
_ : lgamma : _
```

(ma.)J0

Computes the Bessel function of the first kind of order 0 of the input signal.

Usage

```
_ : J0 : _
```

(ma.)J1

Computes the Bessel function of the first kind of order 1 of the input signal.

Usage

```
_ : J1 : _
```

(ma.)Jn

Computes the Bessel function of the first kind of order n (first input signal) of the second input signal.

Usage

```
_,_ : Jn : _
```

(ma.)Y0

Computes the linearly independent Bessel function of the second kind of order 0 of the input signal.

Usage

```
_ : Y0 : _
```

`(ma.)Y1`

Computes the linearly independent Bessel function of the second kind of order 1 of the input signal.

Usage

```
_ : Y0 : _
```

`(ma.)Yn`

Computes the linearly independent Bessel function of the second kind of order n (first input signal) of the second input signal.

Usage

```
_,_ : Yn : _
```

`(ma.)fabs`, `(ma.)fmax`, `(ma.)fmin`

Just for compatibility...

```
fabs = abs
fmax = max
fmin = min
```

`(ma.)np2`

Gives the next power of 2 of x.

Usage

```
np2(n) : _
```

Where:

- `n`: an integer

(ma.)frac

Gives the fractional part of n.

Usage

```
frac(n) : _
```

Where:

- `n`: a decimal number

(ma.)modulo

Modulus operation.

Usage

```
modulo(x,N) : _
```

Where:

- `x`: the numerator
- `N`: the denominator

(ma.)isnan

Return non-zero if x is a NaN.

Usage

```
isnan(x)
_ : isnan : _
```

Where:

- `x`: signal to analyse

(ma.)isinf

Return non-zero if x is a positive or negative infinity.

Usage

```
isinf(x)
_ : isinf : _
```

Where:

- `x`: signal to analyse

`(ma.)chebychev`

Chebychev transformation of order n.

Usage

```
_ : chebychev(n) : _
```

Where:

- `n`: the order of the polynomial

Semantics

```
T[0](x) = 1,
T[1](x) = x,
T[n](x) = 2x*T[n-1](x) - T[n-2](x)
```

Reference

http://en.wikipedia.org/wiki/Chebyshev_polynomial

`(ma.)chebychevpoly`

Linear combination of the first Chebyshev polynomials.

Usage

```
_ : chebychevpoly((c0,c1,...,cn)) : _
```

Where:

- `cn`: the different Chebychevs polynomials such that: chebychevpoly((c0,c1,...,cn)) = Sum of chebychev(i)*ci

Reference

http://www.csounds.com/manual/html/chebyshevpoly.html

`(ma.)diffn`

Negated first-order difference.

Usage

```
_ : diffn : _
```

`(ma.)signum`

The signum function signum(x) is defined as -1 for x<0, 0 for x==0, and 1 for x>0.

Usage

```
_ : signum : _
```

`(ma.)nextpow2`

The nextpow2(x) returns the lowest integer m such that 2^m >= x.

Usage

```
2^nextpow2(n)
```

Useful for allocating delay lines, e.g.,

```
delay(2^nextpow2(maxDelayNeeded), currentDelay);
```

---

misceffects.lib

This library contains a collection of audio effects. Its official prefix is `ef`.

Dynamic

`(ef.)cubicnl`

Cubic nonlinearity distortion. `cubicnl` is a standard Faust library.

Usage:

```
_ : cubicnl(drive,offset) : _
_ : cubicnl_nodc(drive,offset) : _
```

Where:

- `drive`: distortion amount, between 0 and 1
- `offset`: constant added before nonlinearity to give even harmonics. Note: offset can introduce a nonzero mean - feed cubicnl output to dcblocker to remove this.

References:

- https://ccrma.stanford.edu/~jos/pasp/Cubic_Soft_Clipper.html
- https://ccrma.stanford.edu/~jos/pasp/Nonlinear_Distortion.html

---

`(ef.)gate_mono`

Mono signal gate. `gate_mono` is a standard Faust function.

Usage

```
_ : gate_mono(thresh,att,hold,rel) : _
```

Where:

- `thresh`: dB level threshold above which gate opens (e.g., -60 dB)
- `att`: attack time = time constant (sec) for gate to open (e.g., 0.0001 s = 0.1 ms)
- `hold`: hold time = time (sec) gate stays open after signal level < thresh (e.g., 0.1 s)
- `rel`: release time = time constant (sec) for gate to close (e.g., 0.020 s = 20 ms)

References

- http://en.wikipedia.org/wiki/Noise_gate
- http://www.soundonsound.com/sos/apr01/articles/advanced.asp
- http://en.wikipedia.org/wiki/Gating_(sound_engineering)

`(ef.)gate_stereo`

Stereo signal gates. `gate_stereo` is a standard Faust function.

Usage

```
_,_ : gate_stereo(thresh,att,hold,rel) : _,_
```

Where:

- `thresh`: dB level threshold above which gate opens (e.g., -60 dB)
- `att`: attack time = time constant (sec) for gate to open (e.g., 0.0001 s = 0.1 ms)

- `hold`: hold time = time (sec) gate stays open after signal level < thresh (e.g., 0.1 s)
- `rel`: release time = time constant (sec) for gate to close (e.g., 0.020 s = 20 ms)

References

- http://en.wikipedia.org/wiki/Noise_gate
- http://www.soundonsound.com/sos/apr01/articles/advanced.asp
- http://en.wikipedia.org/wiki/Gating_(sound_engineering)

Filtering

`(ef.)speakerbp`

Dirt-simple speaker simulator (overall bandpass eq with observed roll-offs above and below the passband).

Low-frequency speaker model = +12 dB/octave slope breaking to flat near f1. Implemented using two dc blockers in series.

High-frequency model = -24 dB/octave slope implemented using a fourth-order Butterworth lowpass.

Example based on measured Celestion G12 (12" speaker):

`speakerbp` is a standard Faust function

Usage

```
speakerbp(f1,f2)
_ : speakerbp(130,5000) : _
```

`(ef.)piano_dispersion_filter`

Piano dispersion allpass filter in closed form.

Usage

```
piano_dispersion_filter(M,B,f0)
_ : piano_dispersion_filter(1,B,f0) : +(totalDelay),_ : fdelay(maxDelay) : _
```

Where:

- `M`: number of first-order allpass sections (compile-time only) Keep below 20. 8 is typical for medium-sized piano strings.
- `B`: string inharmonicity coefficient (0.0001 is typical)
- `f0`: fundamental frequency in Hz

Outputs

- MINUS the estimated delay at `f0` of allpass chain in samples, provided in negative form to facilitate subtraction from delay-line length.
- Output signal from allpass chain

Reference

- "Dispersion Modeling in Waveguide Piano Synthesis Using Tunable Allpass Filters", by Jukka Rauhala and Vesa Valimaki, DAFX-2006, pp. 71-76
- http://www.dafx.ca/proceedings/papers/p_071.pdf (An erratum in Eq. (7) is corrected in Dr. Rauhala's encompassing dissertation (and below).)
- http://www.acoustics.hut.fi/research/asp/piano/

`(ef.)stereo_width`

Stereo Width effect using the Blumlein Shuffler technique. `stereo_width` is a standard Faust function.

Usage

```
_,_ : stereo_width(w) : _,_
```

Where:

- `w`: stereo width between 0 and 1

At `w=0`, the output signal is mono ((left+right)/2 in both channels). At `w=1`, there is no effect (original stereo image). Thus, w between 0 and 1 varies stereo width from 0 to "original".

Reference

- "Applications of Blumlein Shuffling to Stereo Microphone Techniques" Michael A. Gerzon, JAES vol. 42, no. 6, June 1994

Meshes

`(ef.)mesh_square`

Square Rectangular Digital Waveguide Mesh.

Usage

```
bus(4*N) : mesh_square(N) : bus(4*N);
```

Where:

- `N`: number of nodes along each edge - a power of two (1,2,4,8,...)

Reference

https://ccrma.stanford.edu/~jos/pasp/Digital_Waveguide_Mesh.html

Signal Order In and Out

The mesh is constructed recursively using 2x2 embeddings. Thus, the top level of `mesh_square(M)` is a block 2x2 mesh, where each block is a `mesh(M/2)`. Let these blocks be numbered 1,2,3,4 in the geometry NW,NE,SW,SE, i.e., as 1 2 3 4 Each block has four vector inputs and four vector outputs, where the length of each vector is `M/2`. Label the input vectors as Ni,Ei,Wi,Si, i.e., as the inputs from the North, East South, and West, and similarly for the outputs. Then, for example, the upper left input block of M/2 signals is labeled 1Ni. Most of the connections are internal, such as 1Eo -> 2Wi. The `8*(M/2)` input signals are grouped in the order 1Ni 2Ni 3Si 4Si 1Wi 3Wi 2Ei 4Ei and the output signals are 1No 1Wo 2No 2Eo 3So 3Wo 4So 4Eo or

In: 1No 1Wo 2No 2Eo 3So 3Wo 4So 4Eo

Out: 1Ni 2Ni 3Si 4Si 1Wi 3Wi 2Ei 4Ei

Thus, the inputs are grouped by direction N,S,W,E, while the outputs are grouped by block number 1,2,3,4, which can also be interpreted as directions NW, NE, SW, SE. A simple program illustrating these orderings is `process = mesh_square(2);`.

Example

Reflectively terminated mesh impulsed at one corner:

```
mesh_square_test(N,x) = mesh_square(N)~(busi(4*N,x)) // input to corner
with { busi(N,x) = bus(N) : par(i,N,*(-1)) : par(i,N-1,_), +(x); };
process = 1-1' : mesh_square_test(4); // all modes excited forever
```

In this simple example, the mesh edges are connected as follows:

1No -> 1Ni, 1Wo -> 2Ni, 2No -> 3Si, 2Eo -> 4Si,

3So -> 1Wi, 3Wo -> 3Wi, 4So -> 2Ei, 4Eo -> 4Ei

A routing matrix can be used to obtain other connection geometries.

`(ef.)reverseEchoN(nChans,delay)`

Reverse echo effect

Usage

```
_ : ef.reverseEchoN(N,delay) : si.bus(N)
```

Where:

- `` `N` : Number of channels desired (1 or more) ``
- `delay`: echo delay (integer power of 2)

Demo

```
_ : dm.reverseEchoN(N) : _,_
```

Description

The effect uses N instances of reverseDelayRamped at different phases.

`(ef.)reverseDelayRamped(delay,phase)`

Reverse delay with amplitude ramp

Usage

```
_ : ef.reverseDelayRamped(delay,phase) : _
```

Where:

- `delay`: echo delay (integer power of 2)
- `phase`: float between 0 and 1 giving ramp delay phase*delay

Demo

```
_ : dm.reverseEchoN(N) : _,_
```

`(ef.)uniformPanToStereo(nChans)`

Pan nChans channels to the stereo field, spread uniformly left to right

Usage

```
si.bus(N) : ef.uniformPanToStereo(N) : _,_
```

Where:

- `N`: Number of input channels to pan down to stereo

Demo

```
_ : dm.reverseEchoN(N) : _,_
```

Time Based

`(ef.)echo`

A simple echo effect.

`echo` is a standard Faust function

Usage

```
_ : echo(maxDuration,duration,feedback) : _
```

Where:

- `maxDuration`: the max echo duration in seconds
- `duration`: the echo duration in seconds
- `feedback`: the feedback coefficient

Pitch Shifting

`(ef.)transpose`

A simple pitch shifter based on 2 delay lines. `transpose` is a standard Faust function.

Usage

```
_ : transpose(w, x, s) : _
```

Where:

- `w`: the window length (samples)
- `x`: crossfade duration duration (samples)
- `s`: shift (semitones)

---

noises.lib

Faust Noise Generator Library. Its official prefix is `no`.

Functions Reference

`(no.)noise`

White noise generator (outputs random number between -1 and 1). `Noise` is a standard Faust function.

Usage

```
noise : _
```

`(no.)multirandom`

Generates multiple decorrelated random numbers in parallel.

Usage

```
multirandom(n) : si.bus(n)
```

Where:

- `n`: the number of decorrelated random numbers in parallel

`(no.)multinoise`

Generates multiple decorrelated noises in parallel.

Usage

```
multinoise(n) : si.bus(n)
```

Where:

- `n`: the number of decorrelated random numbers in parallel

`(no.)noises`

TODO.

`(no.)pink_noise`

Pink noise (1/f noise) generator (third-order approximation) `pink_noise` is a standard Faust function.

Usage

```
pink_noise : _;
```

Reference:

https://ccrma.stanford.edu/~jos/sasp/Example_Synthesis_1_F_Noise.html

---

`(no.)pink_noise_vm`

Multi pink noise generator.

Usage

```
pink_noise_vm(N) : _;
```

Where:

- `N`: number of latched white-noise processes to sum, not to exceed sizeof(int) in C++ (typically 32).

References

- http://www.dsprelated.com/showarticle/908.php
- http://www.firstpr.com.au/dsp/pink-noise/#Voss-McCartney

`(no.)lfnoise`, `(no.)lfnoise0` and `(no.)lfnoiseN`

Low-frequency noise generators (Butterworth-filtered downsampled white noise).

Usage

```
lfnoise0(rate) : _;   // new random number every int(SR/rate) samples or so
lfnoiseN(N,rate) : _; // same as "lfnoise0(rate) : lowpass(N,rate)" [see filters.lib]
lfnoise(rate) : _;    // same as "lfnoise0(rate) : seq(i,5,lowpass(N,rate))" (no overshoot)
```

Example

(view waveforms in faust2octave):

```
rate = SR/100.0; // new random value every 100 samples (SR from music.lib)
process = lfnoise0(rate),   // sampled/held noise (piecewise constant)
          lfnoiseN(3,rate), // lfnoise0 smoothed by 3rd order Butterworth LPF
          lfnoise(rate);    // lfnoise0 smoothed with no overshoot
```

`(no.)sparse_noise_vm`

sparse noise generator.

Usage

```
sparse_noise(f0) : _;
```

Where:

- `f0`: average frequency of noise impulses per second

Random impulses in the amplitude range -1 to 1 are generated at an average rate of f0 impulses per second.

Reference

- See velvet_noise

`(no.)velvet_noise_vm`

velvet noise generator.

Usage

```
velvet_noise(amp,f0) : _;
```

Where:

- `amp`: amplitude of noise impulses (positive and negative)
- `f0`: average frequency of noise impulses per second

Reference

- Matti Karjalainen and Hanna Jarvelainen, "Reverberation Modeling Using Velvet Noise", in Proc. 30th Int. Conf. Intelligent Audio Environments (AES07), March 2007.

`(no.)gnoise`

approximate zero-mean, unit-variance Gaussian white noise generator.

Usage

```
gnoise(N) : _;
```

Where:

- `N`: number of uniform random numbers added to approximate Gaussian white noise

Reference

- See Central Limit Theorem

---

oscillators.lib

This library contains a collection of sound generators. Its official prefix is `os`.

Wave-Table-Based Oscillators

`(os.)sinwaveform`

Sine waveform ready to use with a `rdtable`.

Usage

```
sinwaveform(tablesize) : _
```

Where:

- `tablesize`: the table size

`(os.)coswaveform`

Cosine waveform ready to use with a `rdtable`.

Usage

```
coswaveform(tablesize) : _
```

Where:

- `tablesize`: the table size

`(os.)phasor`

A simple phasor to be used with a `rdtable`. `phasor` is a standard Faust function.

Usage

```
phasor(tablesize,freq) : _
```

Where:

- `tablesize`: the table size
- `freq`: the frequency of the phasor (Hz)

`(os.)hs_phasor`

Hardsyncing phasor to be used with an `rdtable`.

Usage

```
hs_phasor(tablesize,freq,c) :  _
```

Where:

- `tablesize`: the table size
- `freq`: the frequency of the phasor (Hz)
- `c`: a clock signal, `c>0` resets phase to 0

`(os.)oscsin`

Sine wave oscillator. `oscsin` is a standard Faust function.

Usage

```
oscsin(freq) : _
```

Where:

- `freq`: the frequency of the wave (Hz)

`(os.)hs_oscsin`

Sin lookup table with hardsyncing phase.

Usage

```
hs_oscsin(freq,c) : _
```

Where:

- `freq`: the fundamental frequency of the phasor
- `c`: a clock signal, `c>0` resets phase to 0

`(os.)osccos`

Cosine wave oscillator.

Usage

```
osccos(freq) : _
```

Where:

- `freq`: the frequency of the wave (Hz)

`(os.)oscp`

A sine wave generator with controllable phase.

Usage

```
oscp(freq,p) : _
```

Where:

- `freq`: the frequency of the wave (Hz)
- `p`: the phase in radian

`(os.)osci`

Interpolated phase sine wave oscillator.

Usage

```
osci(freq) : _
```

Where:

- `freq`: the frequency of the wave (Hz)

LFOs

Low-Frequency Oscillators (LFOs) have prefix `lf_` (no aliasing suppression, which is not audible at LF).

(os.)lf_imptrain

Unit-amplitude low-frequency impulse train. `lf_imptrain` is a standard Faust function.

Usage

```
lf_imptrain(freq) : _
```

Where:

- `freq`: frequency in Hz

(os.)lf_pulsetrainpos

Unit-amplitude nonnegative LF pulse train, duty cycle between 0 and 1.

Usage

```
lf_pulsetrainpos(freq,duty) : _
```

Where:

- `freq`: frequency in Hz
- `duty`: duty cycle between 0 and 1

(os.)lf_pulsetrain

Unit-amplitude zero-mean LF pulse train, duty cycle between 0 and 1.

Usage

```
lf_pulsetrain(freq,duty) : _
```

Where:

- `freq`: frequency in Hz
- `duty`: duty cycle between 0 and 1

(os.)lf_squarewavepos

Positive LF square wave in [0,1]

Usage

```
lf_squarewavepos(freq) : _
```

Where:

- `freq`: frequency in Hz

`(os.)lf_squarewave`

Zero-mean unit-amplitude LF square wave. `lf_squarewave` is a standard Faust function.

Usage

```
lf_squarewave(freq) : _
```

Where:

- `freq`: frequency in Hz

`(os.)lf_trianglepos`

Positive unit-amplitude LF positive triangle wave.

Usage

```
lf_trianglepos(freq) : _
```

Where:

- `freq`: frequency in Hz

`(os.)lf_triangle`

Positive unit-amplitude LF triangle wave `lf_triangle` is a standard Faust function.

Usage

```
lf_triangle(freq) : _
```

Where:

- `freq`: frequency in Hz

Low Frequency Sawtooths

Sawtooth waveform oscillators for virtual analog synthesis et al. The 'simple' versions (`lf_rawsaw`, `lf_sawpos` and `saw1`), are mere samplings of the ideal continuous-time ("analog") waveforms. While simple, the aliasing due to sampling is quite audible. The differentiated polynomial waveform family (`saw2`, `sawN`, and derived functions) do some extra processing to suppress aliasing (not audible for very low fundamental frequencies). According to Lehtonen et al. (JASA 2012), the aliasing of `saw2` should be inaudible at fundamental frequencies below 2 kHz or so, for a 44.1 kHz sampling rate and 60 dB SPL presentation level; fundamentals 415 and below required no aliasing suppression (i.e., `saw1` is ok).

`(os.)lf_rawsaw`

Simple sawtooth waveform oscillator between 0 and period in samples.

Usage

```
lf_rawsaw(periodsamps)
```

Where:

- `periodsamps`: number of periods per samples

`(os.)lf_sawpos_phase`

Simple sawtooth waveform oscillator between 0 and 1 with phase control.

Usage

```
lf_sawpos_phase(freq,phase)
```

Where:

- `freq`: frequency
- `phase`: phase

`(os.)lf_sawpos`

Simple sawtooth waveform oscillator between 0 and 1.

Usage

```
lf_sawpos(freq)
```

Where:

- `freq`: frequency

`(os.)lf_saw`

Simple sawtooth waveform. `lf_saw` is a standard Faust function.

Usage

`lf_saw(freq)`

Where:

- `freq`: frequency

Bandlimited Sawtooth

//————————`(os.)sawN`———————— Bandlimited Sawtooth

`sawN(N,freq)`, `sawNp`, `saw2dpw(freq)`, `saw2(freq)`, `saw3(freq)`, `saw4(freq)`, `saw5(freq)`, `saw6(freq)`, `sawtooth(freq)`, `saw2f2(freq)` `saw2f4(freq)`

Method 1 (`saw2`)

Polynomial Transition Regions (PTR) (for aliasing suppression).

References

- Kleimola, J.; Valimaki, V., "Reducing Aliasing from Synthetic Audio Signals Using Polynomial Transition Regions," in Signal Processing Letters, IEEE , vol.19, no.2, pp.67-70, Feb. 2012
- https://aaltodoc.aalto.fi/bitstream/handle/123456789/7747/publication6.pdf?sequence=9
- http://research.spa.aalto.fi/publications/papers/spl-ptr/

Method 2 (`sawN`)

Differentiated Polynomial Waves (DPW) (for aliasing suppression).

Reference

"Alias-Suppressed Oscillators based on Differentiated Polynomial Waveforms", Vesa Valimaki, Juhan Nam, Julius Smith, and Jonathan Abel, IEEE Tr. Acoustics, Speech, and Language Processing (IEEE-ASLP), Vol. 18, no. 5, May 2010.

Other Cases

Correction-filtered versions of `saw2`: `saw2f2`, `saw2f4` The correction filter compensates "droop" near half the sampling rate. See reference for sawN.

Usage

```
sawN(N,freq) : _
sawNp(N,freq,phase) : _
saw2dpw(freq) : _
saw2(freq) : _
saw3(freq) : _ // based on sawN
saw4(freq) : _ // based on sawN
saw5(freq) : _ // based on sawN
saw6(freq) : _ // based on sawN
sawtooth(freq) : _ // = saw2
saw2f2(freq) : _
saw2f4(freq) : _
```

Where:

- `N`: polynomial order
- `freq`: frequency in Hz
- `phase`: phase

`(os.)sawNp`

TODO: MarkDown doc in comments

`(os.)saw2dpw`

TODO: MarkDown doc in comments

`(os.)saw3`

TODO: MarkDown doc in comments

`(os.)sawtooth`

Alias-free sawtooth wave. 2nd order interpolation (based on `saw2`). `sawtooth` is a standard Faust function.

Usage

```
sawtooth(freq) : _
```

Where:

- `freq`: frequency

`(os.)saw2f2`

TODO: MarkDown doc in comments

`(os.)saw2f4`

TODO: MarkDown doc in comments

Bandlimited Pulse, Square, and Impulse Trains

Bandlimited Pulse, Square, and Impulse Trains.

`pulsetrainN`, `pulsetrain`, `squareN`, `square`, `imptrain`, `imptrainN`, `triangle`, `triangleN`

All are zero-mean and meant to oscillate in the audio frequency range. Use simpler sample-rounded lf_* versions above for LFOs.

Usage

```
pulsetrainN(N,freq,duty) : _
pulsetrain(freq, duty) : _ // = pulsetrainN(2)
squareN(N, freq) : _
square : _ // = squareN(2)
imptrainN(N,freq) : _
imptrain : _ // = imptrainN(2)
triangleN(N,freq) : _
triangle : _ // = triangleN(2)
```

Where:

- `N`: polynomial order
- `freq`: frequency in Hz

`(os.)pulsetrainN`

TODO: MarkDown doc in comments

`(os.)pulsetrain`

Bandlimited pulse train oscillator. Based on `pulsetrainN(2)`. `pulsetrain` is a standard Faust function.

Usage

```
pulsetrain(freq, duty) : _
```

Where:

- `freq`: frequency
- `duty`: duty cycle between 0 and 1

`(os.)squareN`

TODO: MarkDown doc in comments

`(os.)square`

Bandlimited square wave oscillator. Based on `squareN(2)`. `square` is a standard Faust function.

Usage

```
square(freq) : _
```

Where:

- `freq`: frequency

`(os.)impulse`

One-time impulse generated when the Faust process is started. `impulse` is a standard Faust function.

Usage

```
impulse : _
```

`(os.)imptrainN`

TODO: MarkDown doc in comments

`(os.)imptrain`

Bandlimited impulse train generator. Based on `imptrainN(2)`. `imptrain` is a standard Faust function.

Usage

```
imptrain(freq) : _
```

Where:

- `freq`: frequency

`(os.)triangleN`

TODO: MarkDown doc in comments

`(os.)triangle`

Bandlimited triangle wave oscillator. Based on `triangleN(2)`. `triangle` is a standard Faust function.

Usage

```
triangle(freq) : _
```

Where:

- `freq`: frequency

Filter-Based Oscillators

Filter-Based Oscillators

Usage

```
osc[b|r|rs|rc|s|w](f), where f = frequency in Hz.
```

References

- http://lac.linuxaudio.org/2012/download/lac12-slides-jos.pdf
- https://ccrma.stanford.edu/~jos/pdf/lac12-paper-jos.pdf

`(os.)oscb`

Sinusoidal oscillator based on the biquad.

Usage

```
oscb(freq) : _
```

Where:

- `freq`: frequency

`(os.)oscrq`

Sinusoidal (sine and cosine) oscillator based on 2D vector rotation, = undamped "coupled-form" resonator = lossless 2nd-order normalized ladder filter.

Usage

```
oscrq(freq) : _,_
```

Where:

- `freq`: frequency

Reference

- https://ccrma.stanford.edu/~jos/pasp/Normalized_Scattering_Junctions.html

---

`(os.)oscrs`

Sinusoidal (sine) oscillator based on 2D vector rotation, = undamped "coupled-form" resonator = lossless 2nd-order normalized ladder filter.

Usage

```
oscrs(freq) : _
```

Where:

- `freq`: frequency

Reference

- https://ccrma.stanford.edu/~jos/pasp/Normalized_Scattering_Junctions.html

---

`(os.)oscrc`

Sinusoidal (cosine) oscillator based on 2D vector rotation, = undamped "coupled-form" resonator = lossless 2nd-order normalized ladder filter.

Usage

```
oscrc(freq) : _
```

Where:

- `freq`: frequency

Reference

- https://ccrma.stanford.edu/~jos/pasp/Normalized_Scattering_Junctions.html

---

`(os.)oscs`

Sinusoidal oscillator based on the state variable filter = undamped "modified-coupled-form" resonator = "magic circle" algorithm used in graphics.

`(os.)osc`

Default sine wave oscillator (same as oscsin). `osc` is a standard Faust function.

Usage

```
osc(freq) : _
```

Where:

- `freq`: the frequency of the wave (Hz)

Waveguide-Resonator-Based Oscillators

Sinusoidal oscillator based on the waveguide resonator `wgr`.

`(os.)oscw`

Sinusoidal oscillator based on the waveguide resonator `wgr`. Unit-amplitude cosine oscillator.

Usage

```
oscwc(freq) : _
```

Where:

- `freq`: frequency

Reference

- https://ccrma.stanford.edu/~jos/pasp/Digital_Waveguide_Oscillator.html

---

`(os.)oscws`

Sinusoidal oscillator based on the waveguide resonator `wgr`. Unit-amplitude sine oscillator.

Usage

```
oscws(freq) : _
```

Where:

- `freq`: frequency

Reference

- https://ccrma.stanford.edu/~jos/pasp/Digital_Waveguide_Oscillator.html

---

`(os.)oscwq`

Sinusoidal oscillator based on the waveguide resonator `wgr`. Unit-amplitude cosine and sine (quadrature) oscillator.

Usage

```
oscwq(freq) : _
```

Where:

- `freq`: frequency

Reference

- https://ccrma.stanford.edu/~jos/pasp/Digital_Waveguide_Oscillator.html

---

`(os.)oscw`

Sinusoidal oscillator based on the waveguide resonator `wgr`. Unit-amplitude cosine oscillator (default).

Usage

```
oscw(freq) : _
```

Where:

- `freq`: frequency

Reference

- https://ccrma.stanford.edu/~jos/pasp/Digital_Waveguide_Oscillator.html

---

Casio CZ Oscillators

Oscillators that mimics some of the Casio CZ oscillators.

`(os.)CZsaw`

Oscillator that mimics the Casio CZ saw oscillator `CZsaw` is a standard Faust function.

Usage

```
CZsaw(fund,index) : _
```

Where:

- `fund`: a saw-tooth waveform between 0 and 1 that the oscillator slaves to
- `index`: the brightness of the oscillator, 0 to 1. 0 = sine-wave, 1 = saw-wave

---

`(os.)CZsquare`

Oscillator that mimics the Casio CZ square oscillator `CZsquare` is a standard Faust function.

Usage

```
CZsquare(fund,index) : _
```

Where:

- `fund`: a saw-tooth waveform between 0 and 1 that the oscillator slaves to
- `index`: the brightness of the oscillator, 0 to 1. 0 = sine-wave, 1 = square-wave

---

`(os.)CZpulse`

Oscillator that mimics the Casio CZ pulse oscillator `CZpulse` is a standard Faust function.

Usage

```
CZpulse(fund,index) : _
```

Where:

- `fund`: a saw-tooth waveform between 0 and 1 that the oscillator slaves to
- `index`: the brightness of the oscillator, 0 gives a sine-wave, 1 is closer to a pulse

---

`(os.)CZsinePulse`

Oscillator that mimics the Casio CZ sine/pulse oscillator `CZsinePulse` is a standard Faust function.

Usage

```
CZsinePulse(fund,index) : _
```

Where:

- `fund`: a saw-tooth waveform between 0 and 1 that the oscillator slaves to
- `index`: the brightness of the oscillator, 0 gives a sine-wave, 1 is a sine minus a pulse

---

`(os.)CZhalfSine`

Oscillator that mimics the Casio CZ half sine oscillator `CZhalfSine` is a standard Faust function.

Usage

```
CZhalfSine(fund,index) : _
```

Where:

- `fund`: a saw-tooth waveform between 0 and 1 that the oscillator slaves to
- `index`: the brightness of the oscillator, 0 gives a sine-wave, 1 is somewhere between a saw and a square

---

`(os.)CZresSaw`

Oscillator that mimics the Casio CZ resonant saw-tooth oscillator `CZresSaw` is a standard Faust function.

Usage

```
CZresSaw(fund,res) : _
```

Where:

- `fund`: a saw-tooth waveform between 0 and 1 that the oscillator slaves to
- `res`: the frequency of resonance as a factor of the fundamental pitch.

`(os.)CZresTriangle`

Oscillator that mimics the Casio CZ resonant triangle oscillator `CZresTriangle` is a standard Faust function.

Usage

```
CZresTriangle(fund,res) : _
```

Where:

- `fund`: a saw-tooth waveform between 0 and 1 that the oscillator slaves to
- `res`: the frequency of resonance as a factor of the fundamental pitch.

`(os.)CZresTrap`

Oscillator that mimics the Casio CZ resonant trapeze oscillator `CZresTrap` is a standard Faust function.

Usage

```
CZresTrap(fund,res) : _
```

Where:

- `fund`: a saw-tooth waveform between 0 and 1 that the oscillator slaves to
- `res`: the frequency of resonance as a factor of the fundamental pitch.

PolyBLEP-Based Oscillators

`(os.)polyblep`

PolyBLEP residual function - used for smoothing steps in the audio signal.

Usage

```
polyblep(Q, phase) : _
```

Where:

- `Q`: smoothing factor between 0 and 0.5. Determines how far from the ends of the phase interval the quadratic function is used.
- `phase`: normalised phase (between 0 and 1)

`(os.)polyblep_saw`

Sawtooth oscillator with suppressed aliasing (using polyBLEP)

Usage

```
polyblep_saw(f) : _
```

Where:

- `f`: frequency in Hz

`(os.)polyblep_square`

Square wave oscillator with suppressed aliasing (using polyBLEP)

Usage

```
polyblep_square(f) : _
```

Where:

- `f`: frequency in Hz

`(os.)polyblep_triangle`

Triangle wave oscillator with suppressed aliasing (using polyBLEP)

Usage

```
polyblep_triangle(f) : _
```

Where:

- `f`: frequency in Hz

Filter-Based Oscillators

`(os.)quadosc`

Sinusoidal oscillator based on QuadOsc by Martin Vicanek

Usage

```
quadosc(freq) : _
```

where

- `freq`: frequency in Hz

Reference

- https://vicanek.de/articles/QuadOsc.pdf

---

phaflangers.lib

A library of phasor and flanger effects. Its official prefix is `pf`.

Functions Reference

`(pf.)flanger_mono`

Mono flanging effect.

Usage:

```
_ : flanger_mono(dmax,curdel,depth,fb,invert) : _;
```

Where:

- `dmax`: maximum delay-line length (power of 2) - 10 ms typical
- `curdel`: current dynamic delay (not to exceed dmax)
- `depth`: effect strength between 0 and 1 (1 typical)
- `fb`: feedback gain between 0 and 1 (0 typical)
- `invert`: 0 for normal, 1 to invert sign of flanging sum

Reference

https://ccrma.stanford.edu/~jos/pasp/Flanging.html

`(pf.)flanger_stereo`

Stereo flanging effect. `flanger_stereo` is a standard Faust function.

Usage:

```
_,_ : flanger_stereo(dmax,curdel1,curdel2,depth,fb,invert) : _,_;
```

Where:

- `dmax`: maximum delay-line length (power of 2) - 10 ms typical
- `curdel`: current dynamic delay (not to exceed dmax)
- `depth`: effect strength between 0 and 1 (1 typical)
- `fb`: feedback gain between 0 and 1 (0 typical)
- `invert`: 0 for normal, 1 to invert sign of flanging sum

Reference

https://ccrma.stanford.edu/~jos/pasp/Flanging.html

`(pf.)phaser2_mono`

Mono phasing effect.

Phaser

```
_ : phaser2_mono(Notches,phase,width,frqmin,fratio,frqmax,speed,depth,fb,invert) : _;
```

Where:

- `Notches`: number of spectral notches (MACRO ARGUMENT - not a signal)

- `phase`: phase of the oscillator (0-1)
- `width`: approximate width of spectral notches in Hz
- `frqmin`: approximate minimum frequency of first spectral notch in Hz
- `fratio`: ratio of adjacent notch frequencies
- `frqmax`: approximate maximum frequency of first spectral notch in Hz
- `speed`: LFO frequency in Hz (rate of periodic notch sweep cycles)
- `depth`: effect strength between 0 and 1 (1 typical) (aka "intensity") when depth=2, "vibrato mode" is obtained (pure allpass chain)
- `fb`: feedback gain between -1 and 1 (0 typical)
- `invert`: 0 for normal, 1 to invert sign of flanging sum

Reference:

- https://ccrma.stanford.edu/~jos/pasp/Phasing.html
- http://www.geofex.com/Article_Folders/phasers/phase.html
- 'An Allpass Approach to Digital Phasing and Flanging', Julius O. Smith III, Proc. Int. Computer Music Conf. (ICMC-84), pp. 103-109, Paris, 1984.
- CCRMA Tech. Report STAN-M-21: https://ccrma.stanford.edu/STANM/stanms/stanm21/

`(pf.)phaser2_stereo`

Stereo phasing effect. `phaser2_stereo` is a standard Faust function.

Phaser

```
_ : phaser2_stereo(Notches,phase,width,frqmin,fratio,frqmax,speed,depth,fb,invert) : _;
```

Where:

- `Notches`: number of spectral notches (MACRO ARGUMENT - not a signal)
- `phase`: phase of the oscillator (0-1)
- `width`: approximate width of spectral notches in Hz
- `frqmin`: approximate minimum frequency of first spectral notch in Hz
- `fratio`: ratio of adjacent notch frequencies
- `frqmax`: approximate maximum frequency of first spectral notch in Hz
- `speed`: LFO frequency in Hz (rate of periodic notch sweep cycles)
- `depth`: effect strength between 0 and 1 (1 typical) (aka "intensity") when depth=2, "vibrato mode" is obtained (pure allpass chain)
- `fb`: feedback gain between -1 and 1 (0 typical)
- `invert`: 0 for normal, 1 to invert sign of flanging sum

Reference:

- https://ccrma.stanford.edu/~jos/pasp/Phasing.html
- http://www.geofex.com/Article_Folders/phasers/phase.html

- 'An Allpass Approach to Digital Phasing and Flanging', Julius O. Smith III, Proc. Int. Computer Music Conf. (ICMC-84), pp. 103-109, Paris, 1984.
- CCRMA Tech. Report STAN-M-21: https://ccrma.stanford.edu/STANM/stanms/stanm21/

---

physmodels.lib

Faust physical modeling library; Its official prefix is `pm`.

This library provides an environment to facilitate physical modeling of musical instruments. It contains dozens of functions implementing low and high level elements going from a simple waveguide to fully operational models with built-in UI, etc.

It is organized as follows:

- Global Variables: Useful pre-defined variables for physical modeling (e.g., speed of sound, etc.).
- Conversion Tools: Conversion functions specific to physical modeling (e.g., length to frequency, etc.).
- Bidirectional Utilities: Functions to create bidirectional block diagrams for physical modeling.
- Basic Elements: waveguides, specific types of filters, etc.
- String Instruments: various types of strings (e.g., steel, nylon, etc.), bridges, guitars, etc.
- Bowed String Instruments: parts and models specific to bowed string instruments (e.g., bows, bridges, violins, etc.).
- Wind Instrument: parts and models specific to wind string instruments (e.g., reeds, mouthpieces, flutes, clarinets, etc.).
- Exciters: pluck generators, "blowers", etc.
- Modal Percussions: percussion instruments based on modal models.
- Vocal Synthesis: functions for various vocal synthesis techniques (e.g., fof, source/filter, etc.) and vocal synthesizers.
- Misc Functions: any other functions that don't fit in the previous category (e.g., nonlinear filters, etc.).

This library is part of the Faust Physical Modeling ToolKit. More information on how to use this library can be found on this page: https://ccrma.stanford.edu/~rmichon/pmFaust. Tutorials on how to make physical models of musical instruments using Faust can be found here as well.

Global Variables

Useful pre-defined variables for physical modeling.

`(pm.)speedOfSound`

Speed of sound in meters per second (340m/s).

`(pm.)maxLength`

The default maximum length (3) in meters of strings and tubes used in this library. This variable should be overriden to allow longer strings or tubes.

Conversion Tools

Useful conversion tools for physical modeling.

`(pm.)f2l`

Frequency to length in meters.

Usage

```
f2l(freq) : distanceInMeters
```

Where:

- `freq`: the frequency

`(pm.)l2f`

Length in meters to frequency.

Usage

```
l2f(length) : freq
```

Where:

- `length`: length/distance in meters

`(pm.)l2s`

Length in meters to number of samples.

Usage

```
l2s(l) : numberOfSamples
```

Where:

- `l`: length in meters

Bidirectional Utilities

Set of fundamental functions to create bi-directional block diagrams in Faust. These elements are used as the basis of this library to connect high level elements (e.g., mouthpieces, strings, bridge, instrument body, etc.). Each block has 3 inputs and 3 outputs. The first input/output carry left going waves, the second input/output carry right going waves, and the third input/output is used to carry any potential output signal to the end of the algorithm.

`(pm.)basicBlock`

Empty bidirectional block to be used with `chain`: 3 signals ins and 3 signals out.

Usage

```
chain(basicBlock : basicBlock : etc.)
```

`(pm.)chain`

Creates a chain of bidirectional blocks. Blocks must have 3 inputs and outputs. The first input/output carry left going waves, the second input/output carry right going waves, and the third input/output is used to carry any potential output signal to the end of the algorithm. The implied one sample delay created by the `~` operator is generalized to the left and right going waves. Thus, `n` blocks in `chain()` will add an `n` samples delay to both left and right going waves.

Usage

```
leftGoingWaves,rightGoingWaves,mixedOutput : chain( A : B ) : leftGoingWaves,rightGoingWaves
with{
     A = _,_,_;
     B = _,_,_;
};
```

`(pm.)inLeftWave`

Adds a signal to left going waves anywhere in a `chain` of blocks.

Usage

```
model(x) = chain(A : inLeftWave(x) : B)
```

Where `A` and `B` are bidirectional blocks and `x` is the signal added to left going waves in that chain.

`(pm.)inRightWave`

Adds a signal to right going waves anywhere in a `chain` of blocks.

Usage

```
model(x) = chain(A : inRightWave(x) : B)
```

Where `A` and `B` are bidirectional blocks and `x` is the signal added to right going waves in that chain.

`(pm.)in`

Adds a signal to left and right going waves anywhere in a `chain` of blocks.

Usage

```
model(x) = chain(A : in(x) : B)
```

Where `A` and `B` are bidirectional blocks and `x` is the signal added to left and right going waves in that chain.

`(pm.)outLeftWave`

Sends the signal of left going waves to the output channel of the `chain`.

Usage

```
chain(A : outLeftWave : B)
```

Where `A` and `B` are bidirectional blocks.

`(pm.)outRightWave`

Sends the signal of right going waves to the output channel of the `chain`.

Usage

```
chain(A : outRightWave : B)
```

Where `A` and `B` are bidirectional blocks.

`(pm.)out`

Sends the signal of right and left going waves to the output channel of the `chain`.

Usage

```
chain(A : out : B)
```

Where `A` and `B` are bidirectional blocks.

`(pm.)terminations`

Creates terminations on both sides of a `chain` without closing the inputs and outputs of the bidirectional signals chain. As for `chain`, this function adds a 1 sample delay to the bidirectional signal, both ways. Of courses, this function can be nested within a `chain`.

Usage

```
terminations(a,b,c)
with{
     a = *(-1); // left termination
     b = chain(D : E : F); // bidirectional chain of blocks (D, E, F, etc.)
     c = *(-1); // right termination
};
```

`(pm.)lTermination`

Creates a termination on the left side of a `chain` without closing the inputs and outputs of the bidirectional signals chain. This function adds a 1 sample delay near the termination and can be nested within another `chain`.

Usage

```
lTerminations(a,b)
with{
     a = *(-1); // left termination
     b = chain(D : E : F); // bidirectional chain of blocks (D, E, F, etc.)
};
```

(pm.)rTermination

Creates a termination on the right side of a `chain` without closing the inputs and outputs of the bidirectional signals chain. This function adds a 1 sample delay near the termination and can be nested within another `chain`.

Usage

```
rTerminations(b,c)
with{
     b = chain(D : E : F); // bidirectional chain of blocks (D, E, F, etc.)
     c = *(-1); // right termination
};
```

(pm.)closeIns

Closes the inputs of a bidirectional chain in all directions.

Usage

```
closeIns : chain(...) :  _,_,_
```

(pm.)closeOuts

Closes the outputs of a bidirectional chain in all directions except for the main signal output (3d output).

Usage

```
_,_,_ : chain(...) : _
```

(pm.)endChain

Closes the inputs and outputs of a bidirectional chain in all directions except for the main signal output (3d output).

Usage

```
endChain(chain(...)) : _
```

Basic Elements

Basic elements for physical modeling (e.g., waveguides, specific filters, etc.).

`(pm.)waveguideN`

A series of waveguide functions based on various types of delays (see `fdelay[n]`).

List of functions

- `waveguideUd`: unit delay waveguide
- `waveguideFd`: fractional delay waveguide
- `waveguideFd2`: second order fractional delay waveguide
- `waveguideFd4`: fourth order fractional delay waveguide

Usage

```
chain(A : waveguideUd(nMax,n) : B)
```

Where:

- `nMax`: the maximum length of the delays in the waveguide
- `n`: the length of the delay lines in samples.

`(pm.)waveguide`

Standard `pm.lib` waveguide (based on `waveguideFd4`).

Usage

```
chain(A : waveguide(nMax,n) : B)
```

Where:

- `nMax`: the maximum length of the delays in the waveguide
- `n`: the length of the delay lines in samples.

`(pm.)bridgeFilter`

Generic two zeros bridge FIR filter (as implemented in the STK) that can be used to implement the reflectance violin, guitar, etc. bridges.

Usage

```
_ : bridge(brightness,absorption) : _
```

Where:

- `brightness`: controls the damping of high frequencies (0-1)
- `absorption`: controls the absorption of the brige and thus the t60 of the string plugged to it (0-1) (1 = 20 seconds)

`(pm.)modeFilter`

Resonant bandpass filter that can be used to implement a single resonance (mode).

Usage

```
_ : modeFilter(freq,t60,gain) : _
```

Where:

- `freq`: mode frequency
- `t60`: mode resonance duration (in seconds)
- `gain`: mode gain (0-1)

String Instruments

Low and high level string instruments parts. Most of the elements in this section can be used in a bidirectional chain.

`(pm.)stringSegment`

A string segment without terminations (just a simple waveguide).

Usage

```
chain(A : stringSegment(maxLength,length) : B)
```

Where:

- `maxLength`: the maximum length of the string in meters (should be static)
- `length`: the length of the string in meters

`(pm.)openString`

A bidirectional block implementing a basic "generic" string with a selectable excitation position. Lowpass filters are built-in and allow to simulate the effect of dispersion on the sound and thus to change the "stiffness" of the string.

Usage

```
chain(... : openString(length,stiffness,pluckPosition,excitation) : ...)
```

Where:

- `length`: the length of the string in meters
- `stiffness`: the stiffness of the string (0-1) (1 for max stiffness)
- `pluckPosition`: excitation position (0-1) (1 is bottom)
- `excitation`: the excitation signal

`(pm.)nylonString`

A bidirectional block implementing a basic nylon string with selectable excitation position. This element is based on `openString` and has a fix stiffness corresponding to that of a nylon string.

Usage

```
chain(... : nylonString(length,pluckPosition,excitation) : ...)
```

Where:

- `length`: the length of the string in meters
- `pluckPosition`: excitation position (0-1) (1 is bottom)
- `excitation`: the excitation signal

`(pm.)steelString`

A bidirectional block implementing a basic steel string with selectable excitation position. This element is based on `openString` and has a fix stiffness corresponding to that of a steel string.

Usage

```
chain(... : steelString(length,pluckPosition,excitation) : ...)
```

Where:

- `length`: the length of the string in meters
- `pluckPosition`: excitation position (0-1) (1 is bottom)
- `excitation`: the excitation signal

`(pm.)openStringPick`

A bidirectional block implementing a "generic" string with selectable excitation position. It also has a built-in pickup whose position is the same as the excitation position. Thus, moving the excitation position will also move the pickup.

Usage

```
chain(... : openStringPick(length,stiffness,pluckPosition,excitation) : ...)
```

Where:

- `length`: the length of the string in meters
- `stiffness`: the stiffness of the string (0-1) (1 for max stiffness)
- `pluckPosition`: excitation position (0-1) (1 is bottom)
- `excitation`: the excitation signal

`(pm.)openStringPickUp`

A bidirectional block implementing a "generic" string with selectable excitation position and stiffness. It also has a built-in pickup whose position can be independenly selected. The only constraint is that the pickup has to be placed after the excitation position.

Usage

```
chain(... : openStringPickUp(length,stiffness,pluckPosition,excitation) : ...)
```

Where:

- `length`: the length of the string in meters
- `stiffness`: the stiffness of the string (0-1) (1 for max stiffness)
- `pluckPosition`: pluck position between the top of the string and the pickup (0-1) (1 for same as pickup position)
- `pickupPosition`: position of the pickup on the string (0-1) (1 is bottom)
- `excitation`: the excitation signal

`(pm.)openStringPickDown`

A bidirectional block implementing a "generic" string with selectable excitation position and stiffness. It also has a built-in pickup whose position can be independenly selected. The only constraint is that the pickup has to be placed before the excitation position.

Usage

```
chain(... : openStringPickDown(length,stiffness,pluckPosition,excitation) : ...)
```

Where:

- `length`: the length of the string in meters
- `stiffness`: the stiffness of the string (0-1) (1 for max stiffness)
- `pluckPosition`: pluck position on the string (0-1) (1 is bottom)
- `pickupPosition`: position of the pickup between the top of the string and the excitation position (0-1) (1 is excitation position)
- `excitation`: the excitation signal

`(pm.)ksReflexionFilter`

The "typical" one-zero Karplus-strong feedforward reflexion filter. This filter will be typically used in a termination (see below).

Usage

```
terminations(_,chain(...),ksReflexionFilter)
```

`(pm.)rStringRigidTermination`

Bidirectional block implementing a right rigid string termination (no damping, just phase inversion).

Usage

```
chain(rStringRigidTermination : stringSegment : ...)
```

`(pm.)lStringRigidTermination`

Bidirectional block implementing a left rigid string termination (no damping, just phase inversion).

Usage

```
chain(... : stringSegment : lStringRigidTermination)
```

`(pm.)elecGuitarBridge`

Bidirectional block implementing a simple electric guitar bridge. This block is based on `bridgeFilter`. The bridge doesn't implement transmittance since it is not meant to be connected to a body (unlike acoustic guitar). It also partially sets the resonance duration of the string with the nuts used on the other side.

Usage

```
chain(... : stringSegment : elecGuitarBridge)
```

`(pm.)elecGuitarNuts`

Bidirectional block implementing a simple electric guitar nuts. This block is based on `bridgeFilter` and does essentially the same thing as `elecGuitarBridge`, but on the other side of the chain. It also partially sets the resonance duration of the string with the bridge used on the other side.

Usage

```
chain(elecGuitarNuts : stringSegment : ...)
```

`(pm.)guitarBridge`

Bidirectional block implementing a simple acoustic guitar bridge. This bridge damps more hight frequencies than `elecGuitarBridge` and implements a transmittance filter. It also partially sets the resonance duration of the string with the nuts used on the other side.

Usage

```
chain(... : stringSegment : guitarBridge)
```

`(pm.)guitarNuts`

Bidirectional block implementing a simple acoustic guitar nuts. This nuts damps more hight frequencies than `elecGuitarNuts` and implements a transmittance filter. It also partially sets the resonance duration of the string with the bridge used on the other side.

Usage

```
chain(guitarNuts : stringSegment : ...)
```

`(pm.)idealString`

An "ideal" string with rigid terminations and where the plucking position and the pick-up position are the same. Since terminations are rigid, this string will ring forever.

Usage

```
1-1' : idealString(length,reflexion,xPosition,excitation)
```

With: * `length`: the length of the string in meters * `pluckPosition`: the plucking position (0.001-0.999) * `excitation`: the input signal for the excitation.

`(pm.)ks`

A Karplus-Strong string (in that case, the string is implemented as a one dimension waveguide).

Usage

```
ks(length,damping,excitation) : _
```

Where:

- `length`: the length of the string in meters
- `damping`: string damping (0-1)
- `excitation`: excitation signal

`(pm.)ks_ui_MIDI`

Ready-to-use, MIDI-enabled Karplus-Strong string with buil-in UI.

Usage

```
ks_ui_MIDI : _
```

`(pm.)elecGuitarModel`

A simple electric guitar model (without audio effects, of course) with selectable pluck position. This model implements a single string. Additional strings should be created by making a polyphonic applications out of this function. Pitch is changed by changing the length of the string and not through a finger model.

Usage

```
elecGuitarModel(length,pluckPosition,mute,excitation) : _
```

Where:

- `length`: the length of the string in meters
- `pluckPosition`: pluck position (0-1) (1 is on the bridge)
- `mute`: mute coefficient (1 for no mute and 0 for instant mute)
- `excitation`: excitation signal

`(pm.)elecGuitar`

A simple electric guitar model with steel strings (based on `elecGuitarModel`) implementing an excitation model. This model implements a single string. Additional strings should be created by making a polyphonic applications out of this function.

Usage

```
elecGuitar(length,pluckPosition,trigger) : _
```

Where:

- `length`: the length of the string in meters
- `pluckPosition`: pluck position (0-1) (1 is on the bridge)
- `mute`: mute coefficient (1 for no mute and 0 for instant mute)
- `gain`: gain of the pluck (0-1)
- `trigger`: trigger signal (1 for on, 0 for off)

`(pm.)elecGuitar_ui_MIDI`

Ready-to-use MIDI-enabled electric guitar physical model with built-in UI.

Usage

```
elecGuitar_ui_MIDI : _
```

`(pm.)guitarBody`

WARNING: not implemented yet! Bidirectional block implementing a simple acoustic guitar body.

Usage

```
chain(... : guitarBody)
```

(pm.)guitarModel

A simple acoustic guitar model with steel strings and selectable excitation position. This model implements a single string. Additional strings should be created by making a polyphonic applications out of this function. Pitch is changed by changing the length of the string and not through a finger model. WARNING: this function doesn't currently implement a body (just strings and bridge).

Usage

```
guitarModel(length,pluckPosition,excitation) : _
```

Where:

- `length`: the length of the string in meters
- `pluckPosition`: pluck position (0-1) (1 is on the bridge)
- `excitation`: excitation signal

(pm.)guitar

A simple acoustic guitar model with steel strings (based on `guitarModel`) implementing an excitation model. This model implements a single string. Additional strings should be created by making a polyphonic applications out of this function.

Usage

```
guitar(length,pluckPosition,trigger) : _
```

Where:

- `length`: the length of the string in meters
- `pluckPosition`: pluck position (0-1) (1 is on the bridge)
- `gain`: gain of the excitation
- `trigger`: trigger signal (1 for on, 0 for off)

(pm.)guitar_ui_MIDI

Ready-to-use MIDI-enabled steel strings acoustic guitar physical model with built-in UI.

Usage

```
guitar_ui_MIDI : _
```

(pm.)nylonGuitarModel

A simple acoustic guitar model with nylon strings and selectable excitation position. This model implements a single string. Additional strings should be created by making a polyphonic applications out of this function. Pitch is changed by changing the length of the string and not through a finger model. WARNING: this function doesn't currently implement a body (just strings and bridge).

Usage

```
nylonGuitarModel(length,pluckPosition,excitation) : _
```

Where:

- `length`: the length of the string in meters
- `pluckPosition`: pluck position (0-1) (1 is on the bridge)
- `excitation`: excitation signal

(pm.)nylonGuitar

A simple acoustic guitar model with steel strings (based on `nylonGuitarModel`) implementing an excitation model. This model implements a single string. Additional strings should be created by making a polyphonic applications out of this function.

Usage

```
nylonGuitar(length,pluckPosition,trigger) : _
```

Where:

- `length`: the length of the string in meters
- `pluckPosition`: pluck position (0-1) (1 is on the bridge)
- `gain`: gain of the excitation (0-1)
- `trigger`: trigger signal (1 for on, 0 for off)

(pm.)nylonGuitar_ui_MIDI

Ready-to-use MIDI-enabled nylon strings acoustic guitar physical model with built-in UI.

Usage

```
nylonGuitar_ui_MIDI : _
```

(pm.)modeInterpRes

Modular string instrument resonator based on IR measurements made on 3D printed models. The 2D space allowing for the control of the shape and the scale of the model is enabled by interpolating between modes parameters. More information about this technique/project can be found here: https://ccrma.stanford.edu/~rmichon/3dPrintingModeling/.

Usage

```
_ : modeInterpRes(nModes,x,y) : _
```

Where:

- `nModes`: number of modeled modes (40 max)
- `x`: shape of the resonator (0: square, 1: square with rounded corners, 2: round)
- `y`: scale of the resonator (0: small, 1: medium, 2: large)

(pm.)modularInterpBody

Bidirectional block implementing a modular string instrument resonator (see `modeInterpRes`).

Usage

```
chain(... : modularInterpBody(nModes,shape,scale) : ...)
```

Where:

- `nModes`: number of modeled modes (40 max)
- `shape`: shape of the resonator (0: square, 1: square with rounded corners, 2: round)
- `scale`: scale of the resonator (0: small, 1: medium, 2: large)

(pm.)modularInterpStringModel

String instrument model with a modular body (see `modeInterpRes` and https://ccrma.stanford.edu/~rmichon/3dPrintingModeling/).

Usage

```
modularInterpStringModel(length,pluckPosition,shape,scale,bodyExcitation,stringExcitation) :
```

Where:

- `stringLength`: the length of the string in meters
- `pluckPosition`: pluck position (0-1) (1 is on the bridge)
- `shape`: shape of the resonator (0: square, 1: square with rounded corners, 2: round)
- `scale`: scale of the resonator (0: small, 1: medium, 2: large)
- `bodyExcitation`: excitation signal for the body
- `stringExcitation`: excitation signal for the string

`(pm.)modularInterpInstr`

String instrument with a modular body (see `modeInterpRes` and https://ccrma.stanford.edu/~rmichon/3dPrintingModeling/).

Usage

```
modularInterpInstr(stringLength,pluckPosition,shape,scale,gain,tapBody,triggerString) : _
```

Where:

- `stringLength`: the length of the string in meters
- `pluckPosition`: pluck position (0-1) (1 is on the bridge)
- `shape`: shape of the resonator (0: square, 1: square with rounded corners, 2: round)
- `scale`: scale of the resonator (0: small, 1: medium, 2: large)
- `gain`: of the string excitation
- `tapBody`: send an impulse in the body of the instrument where the string is connected (1 for on, 0 for off)
- `triggerString`: trigger signal for the string (1 for on, 0 for off)

`(pm.)modularInterpInstr_ui_MIDI`

Ready-to-use MIDI-enabled string instrument with a modular body (see `modeInterpRes` and https://ccrma.stanford.edu/~rmichon/3dPrintingModeling/) with built-in UI.

Usage

```
modularInterpInstr_ui_MIDI : _
```

Bowed String Instruments

Low and high level basic string instruments parts. Most of the elements in this section can be used in a bidirectional chain.

`(pm.)bowTable`

Extremely basic bow table that can be used to implement a wide range of bow types for many different bowed string instruments (violin, cello, etc.).

Usage

```
excitation : bowTable(offeset,slope) : _
```

Where:

- `excitation`: an excitation signal
- `offset`: table offset
- `slope`: table slope

`(pm.)violinBowTable`

Violin bow table based on `bowTable`.

Usage

```
bowVelocity : violinBowTable(bowPressure) : _
```

Where:

- `bowVelocity`: velocity of the bow/excitation signal (0-1)
- `bowPressure`: bow pressure on the string (0-1)

`(pm.)bowInteraction`

Bidirectional block implementing the interaction of a bow in a `chain`.

Usage

```
chain(... : stringSegment : bowInteraction(bowTable) : stringSegment : ...)
```

Where:

- `bowTable`: the bow table

`(pm.)violinBow`

Bidirectional block implementing a violin bow and its interaction with a string.

Usage

```
chain(... : stringSegment : violinBow(bowPressure,bowVelocity) : stringSegment : ...)
```

Where:

- `bowVelocity`: velocity of the bow / excitation signal (0-1)
- `bowPressure`: bow pressure on the string (0-1)

`(pm.)violinBowedString`

Violin bowed string bidirectional block with controllable bow position. Terminations are not implemented in this model.

Usage

```
chain(nuts : violinBowedString(stringLength,bowPressure,bowVelocity,bowPosition) : bridge)
```

Where:

- `stringLength`: the length of the string in meters
- `bowVelocity`: velocity of the bow / excitation signal (0-1)
- `bowPressure`: bow pressure on the string (0-1)
- `bowPosition`: the position of the bow on the string (0-1)

`(pm.)violinNuts`

Bidirectional block implementing simple violin nuts. This function is based on `bridgeFilter`.

Usage

```
chain(violinNuts : stringSegment : ...)
```

`(pm.)violinBridge`

Bidirectional block implementing a simple violin bridge. This function is based on `bridgeFilter`.

Usage

```
chain(... : stringSegment : violinBridge
```

(pm.)violinBody

Bidirectional block implementing a simple violin body (just a simple resonant lowpass filter).

Usage

```
chain(... : stringSegment : violinBridge : violinBody)
```

(pm.)violinModel

Ready-to-use simple violin physical model. This model implements a single string. Additional strings should be created by making a polyphonic applications out of this function. Pitch is changed by changing the length of the string (and not through a finger model).

Usage

```
violinModel(stringLength,bowPressure,bowVelocity,bridgeReflexion,
bridgeAbsorption,bowPosition) : _
```

Where:

- `stringLength`: the length of the string in meters
- `bowVelocity`: velocity of the bow / excitation signal (0-1)
- `bowPressure`: bow pressure on the string (0-1))
- `bowPosition`: the position of the bow on the string (0-1)

(pm.)violin_ui

Ready-to-use violin physical model with built-in UI.

Usage

```
violinModel_ui : _
```

(pm.)violin_ui_MIDI

Ready-to-use MIDI-enabled violin physical model with built-in UI.

Usage

```
violin_ui_MIDI : _
```

Wind Instruments

Low and high level basic wind instruments parts. Most of the elements in this section can be used in a bidirectional chain.

`(pm.)openTube`

A tube segment without terminations (same as `stringSegment`).

Usage

```
chain(A : openTube(maxLength,length) : B)
```

Where:

- `maxLength`: the maximum length of the tube in meters (should be static)
- `length`: the length of the tube in meters

`(pm.)reedTable`

Extremely basic reed table that can be used to implement a wide range of single reed types for many different instruments (saxophone, clarinet, etc.).

Usage

```
excitation : reedTable(offeset,slope) : _
```

Where:

- `excitation`: an excitation signal
- `offset`: table offset
- `slope`: table slope

`(pm.)fluteJetTable`

Extremely basic flute jet table.

Usage

```
excitation : fluteJetTable : _
```

Where:

- `excitation`: an excitation signal

`(pm.)brassLipsTable`

Simple brass lips/mouthpiece table. Since this implementation is very basic and that the lips and tube of the instrument are coupled to each other, the length of that tube must be provided here.

Usage

```
excitation : brassLipsTable(tubeLength,lipsTension) : _
```

Where:

- `excitation`: an excitation signal (can be DC)
- `tubeLength`: length in meters of the tube connected to the mouthpiece
- `lipsTension`: tension of the lips (0-1) (default: 0.5)

`(pm.)clarinetReed`

Clarinet reed based on `reedTable` with controllable stiffness.

Usage

```
excitation : clarinetReed(stiffness) : _
```

Where:

- `excitation`: an excitation signal
- `stiffness`: reed stiffness (0-1)

`(pm.)clarinetMouthPiece`

Bidirectional block implementing a clarinet mouthpiece as well as the various interactions happening with traveling waves. This element is ready to be plugged to a tube...

Usage

```
chain(clarinetMouthPiece(reedStiffness,pressure) : tube : etc.)
```

Where:

- `pressure`: the pressure of the air flow (DC) created by the virtual performer (0-1). This can also be any kind of signal that will directly injected in the mouthpiece (e.g., breath noise, etc.).

- `reedStiffness`: reed stiffness (0-1)

`(pm.)brassLips`

Bidirectional block implementing a brass mouthpiece as well as the various interactions happening with traveling waves. This element is ready to be plugged to a tube...

Usage

```
chain(brassLips(tubeLength,lipsTension,pressure) : tube : etc.)
```

Where:

- `tubeLength`: length in meters of the tube connected to the mouthpiece
- `lipsTension`: tension of the lips (0-1) (default: 0.5)
- `pressure`: the pressure of the air flow (DC) created by the virtual performer (0-1). This can also be any kind of signal that will directly injected in the mouthpiece (e.g., breath noise, etc.).

`(pm.)fluteEmbouchure`

Bidirectional block implementing a flute embouchure as well as the various interactions happening with traveling waves. This element is ready to be plugged between tubes segments...

Usage

```
chain(... : tube : fluteEmbouchure(pressure) : tube : etc.)
```

Where:

- `pressure`: the pressure of the air flow (DC) created by the virtual performer (0-1). This can also be any kind of signal that will directly injected in the mouthpiece (e.g., breath noise, etc.).

`(pm.)wBell`

Generic wind instrument bell bidirectional block that should be placed at the end of a `chain`.

Usage

```
chain(... : wBell(opening))
```

Where:

- `opening`: the "opening" of bell (0-1)

`(pm.)fluteHead`

Simple flute head implementing waves reflexion.

Usage

```
chain(fluteHead : tube : ...)
```

`(pm.)fluteFoot`

Simple flute foot implementing waves reflexion and dispersion.

Usage

```
chain(... : tube : fluteFoot)
```

`(pm.)clarinetModel`

A simple clarinet physical model without tone holes (pitch is changed by changing the length of the tube of the instrument).

Usage

```
clarinetModel(length,pressure,reedStiffness,bellOpening) : _
```

Where:

- `tubeLength`: the length of the tube in meters
- `pressure`: the pressure of the air flow created by the virtual performer (0-1). This can also be any kind of signal that will directly injected in the mouthpiece (e.g., breath noise, etc.).
- `reedStiffness`: reed stiffness (0-1)
- `bellOpening`: the opening of bell (0-1)

`(pm.)clarinetModel_ui`

Same as `clarinetModel` but with a built-in UI. This function doesn't implement a virtual "blower", thus `pressure` remains an argument here.

Usage

```
clarinetModel_ui(pressure) : _
```

Where:

- `pressure`: the pressure of the air flow created by the virtual performer (0-1). This can also be any kind of signal that will be directly injected in the mouthpiece (e.g., breath noise, etc.).

`(pm.)clarinet_ui`

Ready-to-use clarinet physical model with built-in UI based on `clarinetModel`.

Usage

```
clarinet_ui : _
```

`(pm.)clarinet_ui_MIDI`

Ready-to-use MIDI compliant clarinet physical model with built-in UI.

Usage

```
clarinet_ui_MIDI : _
```

`(pm.)brassModel`

A simple generic brass instrument physical model without pistons (pitch is changed by changing the length of the tube of the instrument). This model is kind of hard to control and might not sound very good if bad parameters are given to it...

Usage

```
brassModel(tubeLength,lipsTension,mute,pressure) : _
```

Where:

- `tubeLength`: the length of the tube in meters
- `lipsTension`: tension of the lips (0-1) (default: 0.5)
- `mute`: mute opening at the end of the instrument (0-1) (default: 0.5)
- `pressure`: the pressure of the air flow created by the virtual performer (0-1). This can also be any kind of signal that will directly injected in the mouthpiece (e.g., breath noise, etc.).

`(pm.)brassModel_ui`

Same as `brassModel` but with a built-in UI. This function doesn't implement a virtual "blower", thus `pressure` remains an argument here.

Usage

```
brassModel_ui(pressure) : _
```

Where:

- `pressure`: the pressure of the air flow created by the virtual performer (0-1). This can also be any kind of signal that will be directly injected in the mouthpiece (e.g., breath noise, etc.).

`(pm.)brass_ui`

Ready-to-use brass instrument physical model with built-in UI based on `brassModel`.

Usage

```
brass_ui : _
```

`(pm.)brass_ui_MIDI`

Ready-to-use MIDI-controllable brass instrument physical model with built-in UI.

Usage

```
brass_ui_MIDI : _
```

`(pm.)fluteModel`

A simple generic flute instrument physical model without tone holes (pitch is changed by changing the length of the tube of the instrument).

Usage

```
fluteModel(tubeLength,mouthPosition,pressure) : _
```

Where:

- `tubeLength`: the length of the tube in meters
- `mouthPosition`: position of the mouth on the embouchure (0-1) (default: 0.5)
- `pressure`: the pressure of the air flow created by the virtual performer (0-1). This can also be any kind of signal that will directly injected in the mouthpiece (e.g., breath noise, etc.).

`(pm.)fluteModel_ui`

Same as `fluteModel` but with a built-in UI. This function doesn't implement a virtual "blower", thus `pressure` remains an argument here.

Usage

```
fluteModel_ui(pressure) : _
```

Where:

- `pressure`: the pressure of the air flow created by the virtual performer (0-1). This can also be any kind of signal that will be directly injected in the mouthpiece (e.g., breath noise, etc.).

`(pm.)flute_ui`

Ready-to-use flute physical model with built-in UI based on `fluteModel`.

Usage

```
flute_ui : _
```

`(pm.)flute_ui_MIDI`

Ready-to-use MIDI-controllable flute physical model with built-in UI.

Usage

```
flute_ui_MIDI : _
```

Exciters

Various kind of excitation signal generators.

`(pm.)impulseExcitation`

Creates an impulse excitation of one sample.

Usage

```
gate = button('gate');
impulseExcitation(gate) : chain;
```

Where:

- `gate`: a gate button

`(pm.)strikeModel`

Creates a filtered noise excitation.

Usage

```
gate = button('gate');
strikeModel(LPcutoff,HPcutoff,sharpness,gain,gate) : chain;
```

Where:

- `HPcutoff`: highpass cutoff frequency
- `LPcutoff`: lowpass cutoff frequency
- `sharpness`: sharpness of the attack and release (0-1)
- `gain`: gain of the excitation
- `gate`: a gate button/trigger signal (0/1)

`(pm.)strike`

Strikes generator with controllable excitation position.

Usage

```
gate = button('gate');
strike(exPos,sharpness,gain,gate) : chain;
```

Where:

- `exPos`: excitation position wiht 0: for max low freqs and 1: for max high freqs. So, on membrane for example, 0 would be the middle and 1 the edge
- `sharpness`: sharpness of the attack and release (0-1)
- `gain`: gain of the excitation
- `gate`: a gate button/trigger signal (0/1)

`(pm.)pluckString`

Creates a plucking excitation signal.

Usage

```
trigger = button('gate');
pluckString(stringLength,cutoff,maxFreq,sharpness,trigger)
```

Where:

- `stringLength`: length of the string to pluck
- `cutoff`: cutoff ratio (1 for default)
- `maxFreq`: max frequency ratio (1 for default)
- `sharpness`: sharpness of the attack and release (1 for default)
- `gain`: gain of the excitation (0-1)
- `trigger`: trigger signal (1 for on, 0 for off)

`(pm.)blower`

A virtual blower creating a DC signal with some breath noise in it.

Usage

```
blower(pressure,breathGain,breathCutoff) : _
```

Where:

- `pressure`: pressure (0-1)
- `breathGain`: breath noise gain (0-1) (recommended: 0.005)
- `breathCutoff`: breath cuttoff frequency (Hz) (recommended: 2000)

`(pm.)blower_ui`

Same as `blower` but with a built-in UI.

Usage

```
blower : somethingToBeBlown
```

Modal Percussions

High and low level functions for modal synthesis of percussion instruments.

`(pm.)djembeModel`

Dirt-simple djembe modal physical model. Mode parameters are empirically calculated and don't correspond to any measurements or 3D model. They kind of sound good though :).

Usage

```
excitation : djembeModel(freq)
```

Where:

- `excitation`: excitation signal
- `freq`: fundamental frequency of the bar

`(pm.)djembe`

Dirt-simple djembe modal physical model. Mode parameters are empirically calculated and don't correspond to any measurements or 3D model. They kind of sound good though :).

This model also implements a virtual "exciter".

Usage

```
djembe(freq,strikePosition,strikeSharpness,gain,trigger)
```

Where:

- `freq`: fundamental frequency of the model
- `strikePosition`: strike position (0 for the middle of the membrane and 1 for the edge)
- `strikeSharpness`: sharpness of the strike (0-1, default: 0.5)
- `gain`: gain of the strike
- `trigger`: trigger signal (0: off, 1: on)

`(pm.)djembe_ui_MIDI`

Simple MIDI controllable djembe physical model with built-in UI.

Usage

```
djembe_ui_MIDI : _
```

`(pm.)marimbaBarModel`

Generic marimba tone bar modal model.

This model was generated using `mesh2faust` from a 3D CAD model of a marimba tone bar (`libraries/modalmodels/marimbaBar`). The corresponding CAD model is that of a C2 tone bar (original fundamental frequency: ~65Hz). While `marimbaBarModel` allows to translate the harmonic content of the generated sound by providing a frequency (`freq`), mode transposition has limits and the model will sound less and less like a marimba tone bar as it diverges from C2. To make an accurate model of a marimba, we'd want to have an independent model for each bar...

This model contains 5 excitation positions going linearly from the center bottom to the center top of the bar. Obviously, a model with more excitation position could be regenerated using `mesh2faust`.

Usage

```
excitation : marimbaBarModel(freq,exPos,t60,t60DecayRatio,t60DecaySlope)
```

Where:

- `excitation`: excitation signal
- `freq`: fundamental frequency of the bar
- `exPos`: excitation position (0-4)
- `t60`: T60 in seconds (recommended value: 0.1)
- `t60DecayRatio`: T60 decay ratio (recommended value: 1)
- `t60DecaySlope`: T60 decay slope (recommended value: 5)

`(pm.)marimbaResTube`

Simple marimba resonance tube.

Usage

```
marimbaResTube(tubeLength,excitation)
```

Where:

- `tubeLength`: the length of the tube in meters
- `excitation`: the excitation signal (audio in)

`(pm.)marimbaModel`

Simple marimba physical model implementing a single tone bar connected to tube. This model is scalable and can be adapted to any size of bar/tube (see `marimbaBarModel` to know more about the limitations of this type of system).

Usage

```
excitation : marimbaModel(freq,exPos) : _
```

Where:

- `freq`: the frequency of the bar/tube couple
- `exPos`: excitation position (0-4)

`(pm.)marimba`

Simple marimba physical model implementing a single tone bar connected to tube. This model is scalable and can be adapted to any size of bar/tube (see `marimbaBarModel` to know more about the limitations of this type of system).

This function also implement a virtual exciter to drive the model.

Usage

```
excitation : marimba(freq,strikePosition,strikeCutoff,strikeSharpness,gain,trigger) : _
```

Where:

- `excitation`: the excitation signal
- `freq`: the frequency of the bar/tube couple
- `strikePosition`: strike position (0-4)
- `strikeCutoff`: cuttoff frequency of the strike genarator (recommended: ~7000Hz)
- `strikeSharpness`: shaarpness of the strike (recommened: ~0.25)
- `gain`: gain of the strike (0-1)
- `trigger` signal (0: off, 1: on)

`(pm.)marimba_ui_MIDI`

Simple MIDI controllable marimba physical model with built-in UI implementing a single tone bar connected to tube. This model is scalable and can be adapted to any size of bar/tube (see `marimbaBarModel` to know more about the limitations of this type of system).

Usage

```
marimba_ui_MIDI : _
```

`(pm.)churchBellModel`

Generic church bell modal model generated by `mesh2faust` from `libraries/modalmodels/churchBell`.

Modeled after T. Rossing and R. Perrin, Vibrations of Bells, Applied Acoustics 2, 1987.

Model height is 301 mm.

This model contains 7 excitation positions going linearly from the bottom to the top of the bell. Obviously, a model with more excitation position could be regenerated using `mesh2faust`.

Usage

```
excitation : churchBellModel(nModes,exPos,t60,t60DecayRatio,t60DecaySlope)
```

Where:

- `excitation`: the excitation signal
- `nModes`: number of synthesized modes (max: 50)
- `exPos`: excitation position (0-6)
- `t60`: T60 in seconds (recommended value: 0.1)
- `t60DecayRatio`: T60 decay ratio (recommended value: 1)
- `t60DecaySlope`: T60 decay slope (recommended value: 5)

`(pm.)churchBell`

Generic church bell modal model.

Modeled after T. Rossing and R. Perrin, Vibrations of Bells, Applied Acoustics 2, 1987.

Model height is 301 mm.

This model contains 7 excitation positions going linearly from the bottom to the top of the bell. Obviously, a model with more excitation position could be regenerated using `mesh2faust`.

This function also implement a virtual exciter to drive the model.

Usage

```
excitation : churchBell(strikePosition,strikeCutoff,strikeSharpness,gain,trigger) : _
```

Where:

- `excitation`: the excitation signal
- `strikePosition`: strike position (0-6)
- `strikeCutoff`: cuttoff frequency of the strike genarator (recommended: ~7000Hz)

- `strikeSharpness`: shaarpness of the strike (recommened: ~0.25)
- `gain`: gain of the strike (0-1)
- `trigger` signal (0: off, 1: on)

`(pm.)churchBell_ui`

Church bell physical model based on `churchBell` with built-in UI.

Usage

```
churchBell_ui : _
```

`(pm.)englishBellModel`

English church bell modal model generated by `mesh2faust` from `libraries/modalmodels/englishBell`.

Modeled after D. Bartocha and . Baron, Influence of Tin Bronze Melting and Pouring Parameters on Its Properties and Bell' Tone, Archives of Foundry Engineering, 2016.

Model height is 1 m.

This model contains 7 excitation positions going linearly from the bottom to the top of the bell. Obviously, a model with more excitation position could be regenerated using `mesh2faust`.

Usage

```
excitation : englishBellModel(nModes,exPos,t60,t60DecayRatio,t60DecaySlope)
```

Where:

- `excitation`: the excitation signal
- `nModes`: number of synthesized modes (max: 50)
- `exPos`: excitation position (0-6)
- `t60`: T60 in seconds (recommended value: 0.1)
- `t60DecayRatio`: T60 decay ratio (recommended value: 1)
- `t60DecaySlope`: T60 decay slope (recommended value: 5)

`(pm.)englishBell`

English church bell modal model.

Modeled after D. Bartocha and . Baron, Influence of Tin Bronze Melting and Pouring Parameters on Its Properties and Bell' Tone, Archives of Foundry Engineering, 2016.

Model height is 1 m.

This model contains 7 excitation positions going linearly from the bottom to the top of the bell. Obviously, a model with more excitation position could be regenerated using `mesh2faust`.

This function also implement a virtual exciter to drive the model.

Usage

```
excitation : englishBell(strikePosition,strikeCutoff,strikeSharpness,gain,trigger) : _
```

Where:

- `excitation`: the excitation signal
- `strikePosition`: strike position (0-6)
- `strikeCutoff`: cuttoff frequency of the strike genarator (recommended: ~7000Hz)
- `strikeSharpness`: shaarpness of the strike (recommened: ~0.25)
- `gain`: gain of the strike (0-1)
- `trigger` signal (0: off, 1: on)

`(pm.)englishBell_ui`

English church bell physical model based on `englishBell` with built-in UI.

Usage

```
englishBell_ui : _
```

`(pm.)frenchBellModel`

French church bell modal model generated by `mesh2faust` from `libraries/modalmodels/frenchBell`.

Modeled after D. Bartocha and . Baron, Influence of Tin Bronze Melting and Pouring Parameters on Its Properties and Bell' Tone, Archives of Foundry Engineering, 2016.

Model height is 1 m.

This model contains 7 excitation positions going linearly from the bottom to the top of the bell. Obviously, a model with more excitation position could be regenerated using `mesh2faust`.

Usage

```
excitation : frenchBellModel(nModes,exPos,t60,t60DecayRatio,t60DecaySlope)
```

Where:

- `excitation`: the excitation signal
- `nModes`: number of synthesized modes (max: 50)
- `exPos`: excitation position (0-6)
- `t60`: T60 in seconds (recommended value: 0.1)
- `t60DecayRatio`: T60 decay ratio (recommended value: 1)
- `t60DecaySlope`: T60 decay slope (recommended value: 5)

`(pm.)frenchBell`

French church bell modal model.

Modeled after D. Bartocha and . Baron, Influence of Tin Bronze Melting and Pouring Parameters on Its Properties and Bell' Tone, Archives of Foundry Engineering, 2016.

Model height is 1 m.

This model contains 7 excitation positions going linearly from the bottom to the top of the bell. Obviously, a model with more excitation position could be regenerated using `mesh2faust`.

This function also implement a virtual exciter to drive the model.

Usage

```
excitation : frenchBell(strikePosition,strikeCutoff,strikeSharpness,gain,trigger) : _
```

Where:

- `excitation`: the excitation signal
- `strikePosition`: strike position (0-6)
- `strikeCutoff`: cuttoff frequency of the strike genarator (recommended: ~7000Hz)
- `strikeSharpness`: shaarpness of the strike (recommened: ~0.25)
- `gain`: gain of the strike (0-1)
- `trigger` signal (0: off, 1: on)

`(pm.)frenchBell_ui`

French church bell physical model based on `frenchBell` with built-in UI.

Usage

```
frenchBell_ui : _
```

`(pm.)germanBellModel`

German church bell modal model generated by `mesh2faust` from `libraries/modalmodels/germanBell`.

Modeled after D. Bartocha and . Baron, Influence of Tin Bronze Melting and Pouring Parameters on Its Properties and Bell' Tone, Archives of Foundry Engineering, 2016.

Model height is 1 m.

This model contains 7 excitation positions going linearly from the bottom to the top of the bell. Obviously, a model with more excitation position could be regenerated using `mesh2faust`.

Usage

```
excitation : germanBellModel(nModes,exPos,t60,t60DecayRatio,t60DecaySlope)
```

Where:

- `excitation`: the excitation signal
- `nModes`: number of synthesized modes (max: 50)
- `exPos`: excitation position (0-6)
- `t60`: T60 in seconds (recommended value: 0.1)
- `t60DecayRatio`: T60 decay ratio (recommended value: 1)
- `t60DecaySlope`: T60 decay slope (recommended value: 5)

`(pm.)germanBell`

German church bell modal model.

Modeled after D. Bartocha and . Baron, Influence of Tin Bronze Melting and Pouring Parameters on Its Properties and Bell' Tone, Archives of Foundry Engineering, 2016.

Model height is 1 m.

This model contains 7 excitation positions going linearly from the bottom to the top of the bell. Obviously, a model with more excitation position could be regenerated using `mesh2faust`.

This function also implement a virtual exciter to drive the model.

Usage

```
excitation : germanBell(strikePosition,strikeCutoff,strikeSharpness,gain,trigger) : _
```

Where:

- `excitation`: the excitation signal
- `strikePosition`: strike position (0-6)

- `strikeCutoff`: cuttoff frequency of the strike genarator (recommended: ~7000Hz)
- `strikeSharpness`: shaarpness of the strike (recommened: ~0.25)
- `gain`: gain of the strike (0-1)
- `trigger` signal (0: off, 1: on)

`(pm.)germanBell_ui`

German church bell physical model based on `germanBell` with built-in UI.

Usage

```
germanBell_ui : _
```

`(pm.)russianBellModel`

Russian church bell modal model generated by `mesh2faust` from `libraries/modalmodels/russianBell`.

Modeled after D. Bartocha and . Baron, Influence of Tin Bronze Melting and Pouring Parameters on Its Properties and Bell' Tone, Archives of Foundry Engineering, 2016.

Model height is 2 m.

This model contains 7 excitation positions going linearly from the bottom to the top of the bell. Obviously, a model with more excitation position could be regenerated using `mesh2faust`.

Usage

```
excitation : russianBellModel(nModes,exPos,t60,t60DecayRatio,t60DecaySlope)
```

Where:

- `excitation`: the excitation signal
- `nModes`: number of synthesized modes (max: 50)
- `exPos`: excitation position (0-6)
- `t60`: T60 in seconds (recommended value: 0.1)
- `t60DecayRatio`: T60 decay ratio (recommended value: 1)
- `t60DecaySlope`: T60 decay slope (recommended value: 5)

`(pm.)russianBell`

Russian church bell modal model.

Modeled after D. Bartocha and . Baron, Influence of Tin Bronze Melting and Pouring Parameters on Its Properties and Bell' Tone, Archives of Foundry Engineering, 2016.

Model height is 2 m.

This model contains 7 excitation positions going linearly from the bottom to the top of the bell. Obviously, a model with more excitation position could be regenerated using `mesh2faust`.

This function also implement a virtual exciter to drive the model.

Usage

```
excitation : russianBell(strikePosition,strikeCutoff,strikeSharpness,gain,trigger) : _
```

Where:

- `excitation`: the excitation signal
- `strikePosition`: strike position (0-6)
- `strikeCutoff`: cuttoff frequency of the strike genarator (recommended: ~7000Hz)
- `strikeSharpness`: shaarpness of the strike (recommened: ~0.25)
- `gain`: gain of the strike (0-1)
- `trigger` signal (0: off, 1: on)

`(pm.)russianBell_ui`

Russian church bell physical model based on `russianBell` with built-in UI.

Usage

```
russianBell_ui : _
```

`(pm.)standardBellModel`

Standard church bell modal model generated by `mesh2faust` from `libraries/modalmodels/standardBell`.

Modeled after T. Rossing and R. Perrin, Vibrations of Bells, Applied Acoustics 2, 1987.

Model height is 1.8 m.

This model contains 7 excitation positions going linearly from the bottom to the top of the bell. Obviously, a model with more excitation position could be regenerated using `mesh2faust`.

Usage

```
excitation : standardBellModel(nModes,exPos,t60,t60DecayRatio,t60DecaySlope)
```

Where:

- `excitation`: the excitation signal
- `nModes`: number of synthesized modes (max: 50)
- `exPos`: excitation position (0-6)
- `t60`: T60 in seconds (recommended value: 0.1)
- `t60DecayRatio`: T60 decay ratio (recommended value: 1)
- `t60DecaySlope`: T60 decay slope (recommended value: 5)

`(pm.)standardBell`

Standard church bell modal model.

Modeled after T. Rossing and R. Perrin, Vibrations of Bells, Applied Acoustics 2, 1987.

Model height is 1.8 m.

This model contains 7 excitation positions going linearly from the bottom to the top of the bell. Obviously, a model with more excitation position could be regenerated using `mesh2faust`.

This function also implement a virtual exciter to drive the model.

Usage

```
excitation : standardBell(strikePosition,strikeCutoff,strikeSharpness,gain,trigger) : _
```

Where:

- `excitation`: the excitation signal
- `strikePosition`: strike position (0-6)
- `strikeCutoff`: cuttoff frequency of the strike genarator (recommended: ~7000Hz)
- `strikeSharpness`: shaarpness of the strike (recommened: ~0.25)
- `gain`: gain of the strike (0-1)
- `trigger` signal (0: off, 1: on)

`(pm.)standardBell_ui`

Standard church bell physical model based on `standardBell` with built-in UI.

Usage

```
standardBell_ui : _
```

Vocal Synthesis

Vocal synthesizer functions (source/filter, fof, etc.).

`(pm.)formantValues`

Formant data values.

The formant data used here come from the CSOUND manual http://www.csounds.com/manual/html/.

Usage

```
ba.take(j+1,formantValues.f(i)) : _
ba.take(j+1,formantValues.g(i)) : _
ba.take(j+1,formantValues.bw(i)) : _
```

Where:

- `i`: formant number
- `j`: (voiceType*nFormants)+vowel
- `voiceType`: the voice type (0: alto, 1: bass, 2: countertenor, 3: soprano, 4: tenor)
- `vowel`: the vowel (0: a, 1: e, 2: i, 3: o, 4: u)

`(pm.)voiceGender`

Calculate the gender for the provided `voiceType` value. (0: male, 1: female)

Usage

```
voiceGender(voiceType) : _
```

Where:

- `voiceType`: the voice type (0: alto, 1: bass, 2: countertenor, 3: soprano, 4: tenor)

`(pm.)skirtWidthMultiplier`

Calculates value to multiply bandwidth to obtain `skirtwidth` for a Fof filter.

Usage

```
skirtWidthMultiplier(vowel,freq,gender) : _
```

Where:

- `vowel`: the vowel (0: a, 1: e, 2: i, 3: o, 4: u)
- `freq`: the fundamental frequency of the excitation signal
- `gender`: gender of the voice used in the fof filter (0: male, 1: female)

`(pm.)autobendFreq`

Autobends the center frequencies of formants 1 and 2 based on the fundamental frequency of the excitation signal and leaves all other formant frequencies unchanged. Ported from `chant-lib`. Reference: https://ccrma.stanford.edu/~rmichon/chantLib/.

Usage

```
_ : autobendFreq(n,freq,voiceType) : _
```

Where:

- `n`: formant index
- `freq`: the fundamental frequency of the excitation signal
- `voiceType`: the voice type (0: alto, 1: bass, 2: countertenor, 3: soprano, 4: tenor)
- input is the center frequency of the corresponding formant

`(pm.)vocalEffort`

Changes the gains of the formants based on the fundamental frequency of the excitation signal. Higher formants are reinforced for higher fundamental frequencies. Ported from `chant-lib`. Reference: https://ccrma.stanford.edu/~rmichon/chantLib/.

Usage

```
_ : vocalEffort(freq,gender) : _
```

Where:

- `freq`: the fundamental frequency of the excitation signal
- `gender`: the gender of the voice type (0: male, 1: female)
- input is the linear amplitude of the formant

`(pm.)fof`

Function to generate a single Formant-Wave-Function. Reference: https://ccrma.stanford.edu/~mjolsen/pdfs/smc2016_MOlsenFOF.pdf.

Usage

```
_ : fof(fc,bw,a,g) : _
```

Where:

- `fc`: formant center frequency,
- `bw`: formant bandwidth (Hz),
- `sw`: formant skirtwidth (Hz)
- `g`: linear scale factor (g=1 gives 0dB amplitude response at fc)
- input is an impulse signal to excite filter

`(pm.)fofSH`

FOF with sample and hold used on `bw` and a parameter used in the filter-cycling FOF function `fofCycle`. Reference: https://ccrma.stanford.edu/~mjolsen/pdfs/smc2016_MOlsenFOF.pdf.

Usage

```
_ : fofSH(fc,bw,a,g) : _
```

Where: all parameters same as for `fof`

`(pm.)fofCycle`

FOF implementation where time-varying filter parameter noise is mitigated by using a cycle of `n` sample and hold FOF filters. Reference: https://ccrma.stanford.edu/~mjolsen/pdfs/smc2016_MOlsenFOF.pdf.

Usage

```
_ : fofCycle(fc,bw,a,g,n) : _
```

Where:

- `n`: the number of FOF filters to cycle through
- all other parameters are same as for `fof`

`(pm.)fofSmooth`

FOF implementation where time-varying filter parameter noise is mitigated by lowpass filtering the filter parameters `bw` and `a` with smooth.

Usage

```
_ : fofSmooth(fc,bw,sw,g,tau) : _
```

Where:

- `tau`: the desired smoothing time constant in seconds
- all other parameters are same as for `fof`

`(pm.)formantFilterFofCycle`

Formant filter based on a single FOF filter. Formant parameters are linearly interpolated allowing to go smoothly from one vowel to another. A cycle of `n` fof filters with sample-and-hold is used so that the fof filter parameters can be varied in realtime. This technique is more robust but more computationally expensive than `formantFilterFofSmooth`.Voice type can be selected but must correspond to the frequency range of the provided source to be realistic.

Usage

```
_ : formantFilterFofCycle(voiceType,vowel,nFormants,i,freq) : _
```

Where:

- `voiceType`: the voice type (0: alto, 1: bass, 2: countertenor, 3: soprano, 4: tenor)
- `vowel`: the vowel (0: a, 1: e, 2: i, 3: o, 4: u)
- `nFormants`: number of formant regions in frequency domain, typically 5
- `i`: formant number (i.e. 0 - 4) used to index formant data value arrays
- `freq`: fundamental frequency of excitation signal. Used to calculate rise time of envelope

---

`(pm.)formantFilterFofSmooth`

Formant filter based on a single FOF filter. Formant parameters are linearly interpolated allowing to go smoothly from one vowel to another. Fof filter parameters are lowpass filtered to mitigate possible noise from varying them in realtime. Voice type can be selected but must correspond to the frequency range of the provided source to be realistic.

Usage

```
_ : formantFilterFofSmooth(voiceType,vowel,nFormants,i,freq) : _
```

Where:

- `voiceType`: the voice type (0: alto, 1: bass, 2: countertenor, 3: soprano, 4: tenor)
- `vowel`: the vowel (0: a, 1: e, 2: i, 3: o, 4: u)
- `nFormants`: number of formant regions in frequency domain, typically 5
- `i`: formant number (i.e. 1 - 5) used to index formant data value arrays
- `freq`: fundamental frequency of excitation signal. Used to calculate rise time of envelope

---

`(pm.)formantFilterBP`

Formant filter based on a single resonant bandpass filter. Formant parameters are linearly interpolated allowing to go smoothly from one vowel to another. Voice type can be selected but must correspond to the frequency range of the provided source to be realistic.

Usage

```
_ : formantFilterBP(voiceType,vowel,nFormants,i,freq) : _
```

Where:

- `voiceType`: the voice type (0: alto, 1: bass, 2: countertenor, 3: soprano, 4: tenor)
- `vowel`: the vowel (0: a, 1: e, 2: i, 3: o, 4: u)
- `nFormants`: number of formant regions in frequency domain, typically 5
- `i`: formant index used to index formant data value arrays
- `freq`: fundamental frequency of excitation signal.

`(pm.)formantFilterbank`

Formant filterbank which can use different types of filterbank functions and different excitation signals. Formant parameters are linearly interpolated allowing to go smoothly from one vowel to another. Voice type can be selected but must correspond to the frequency range of the provided source to be realistic.

Usage

```
_ : formantFilterbank(voiceType,vowel,formantGen,freq) : _
```

Where:

- `voiceType`: the voice type (0: alto, 1: bass, 2: countertenor, 3: soprano, 4: tenor)
- `vowel`: the vowel (0: a, 1: e, 2: i, 3: o, 4: u)
- `formantGen`: the specific formant filterbank function (i.e. FormantFilterbankBP, FormantFilterbankFof,...)
- `freq`: fundamental frequency of excitation signal. Needed for FOF version to calculate rise time of envelope

---

`(pm.)formantFilterbankFofCycle`

Formant filterbank based on a bank of fof filters. Formant parameters are linearly interpolated allowing to go smoothly from one vowel to another. Voice type can be selected but must correspond to the frequency range of the provided source to be realistic.

Usage

```
_ : formantFilterbankFofCycle(voiceType,vowel,freq) : _
```

Where:

- `voiceType`: the voice type (0: alto, 1: bass, 2: countertenor, 3: soprano, 4: tenor)
- `vowel`: the vowel (0: a, 1: e, 2: i, 3: o, 4: u)
- `freq`: the fundamental frequency of the excitation signal. Needed to calculate the skirtwidth of the FOF envelopes and for the autobendFreq and vocalEffort functions

---

`(pm.)formantFilterbankFofSmooth`

Formant filterbank based on a bank of fof filters. Formant parameters are linearly interpolated allowing to go smoothly from one vowel to another. Voice type can be selected but must correspond to the frequency range of the provided source to be realistic.

Usage

```
_ : formantFilterbankFofSmooth(voiceType,vowel,freq) : _
```

Where:

- `voiceType`: the voice type (0: alto, 1: bass, 2: countertenor, 3: soprano, 4: tenor)
- `vowel`: the vowel (0: a, 1: e, 2: i, 3: o, 4: u)

- `freq`: the fundamental frequency of the excitation signal. Needed to calculate the skirtwidth of the FOF envelopes and for the autobendFreq and vocalEffort functions

---

`(pm.)formantFilterbankBP`

Formant filterbank based on a bank of resonant bandpass filters. Formant parameters are linearly interpolated allowing to go smoothly from one vowel to another. Voice type can be selected but must correspond to the frequency range of the provided source to be realistic.

Usage

```
_ : formantFilterbankBP(voiceType,vowel) : _
```

Where:

- `voiceType`: the voice type (0: alto, 1: bass, 2: countertenor, 3: soprano, 4: tenor)
- `vowel`: the vowel (0: a, 1: e, 2: i, 3: o, 4: u)
- `freq`: the fundamental frequency of the excitation signal. Needed for the autobendFreq and vocalEffort functions

---

`(pm.)SFFormantModel`

Simple formant/vocal synthesizer based on a source/filter model. The `source` and `filterbank` must be specified by the user. `filterbank` must take the same input parameters as `formantFilterbank` (`BP`/`FofCycle` /`FofSmooth`). Formant parameters are linearly interpolated allowing to go smoothly from one vowel to another. Voice type can be selected but must correspond to the frequency range of the synthesized voice to be realistic.

Usage

```
SFFormantModel(voiceType,vowel,exType,freq,gain,source,filterbank,isFof) : _
```

Where:

- `voiceType`: the voice type (0: alto, 1: bass, 2: countertenor, 3: soprano, 4: tenor)
- `vowel`: the vowel (0: a, 1: e, 2: i, 3: o, 4: u
- `exType`: voice vs. fricative sound ratio (0-1 where 1 is 100% fricative)
- `freq`: the fundamental frequency of the source signal
- `gain`: linear gain multiplier to multiply the source by
- `isFof`: whether model is FOF based (0: no, 1: yes)

`(pm.)SFFormantModelFofCycle`

Simple formant/vocal synthesizer based on a source/filter model. The source is just a periodic impulse and the "filter" is a bank of FOF filters. Formant parameters are linearly interpolated allowing to go smoothly from one vowel to another. Voice type can be selected but must correspond to the frequency range of the synthesized voice to be realistic. This model does not work with noise in the source signal so exType has been removed and model does not depend on SFFormantModel function.

Usage

```
SFFormantModelFofCycle(voiceType,vowel,freq,gain) : _
```

Where:

- `voiceType`: the voice type (0: alto, 1: bass, 2: countertenor, 3: soprano, 4: tenor)
- `vowel`: the vowel (0: a, 1: e, 2: i, 3: o, 4: u
- `freq`: the fundamental frequency of the source signal
- `gain`: linear gain multiplier to multiply the source by

`(pm.)SFFormantModelFofSmooth`

Simple formant/vocal synthesizer based on a source/filter model. The source is just a periodic impulse and the "filter" is a bank of FOF filters. Formant parameters are linearly interpolated allowing to go smoothly from one vowel to another. Voice type can be selected but must correspond to the frequency range of the synthesized voice to be realistic.

Usage

```
SFFormantModelFofSmooth(voiceType,vowel,freq,gain) : _
```

Where:

- `voiceType`: the voice type (0: alto, 1: bass, 2: countertenor, 3: soprano, 4: tenor)
- `vowel`: the vowel (0: a, 1: e, 2: i, 3: o, 4: u
- `freq`: the fundamental frequency of the source signal
- `gain`: linear gain multiplier to multiply the source by

`(pm.)SFFormantModelBP`

Simple formant/vocal synthesizer based on a source/filter model. The source is just a sawtooth wave and the "filter" is a bank of resonant bandpass filters.

Formant parameters are linearly interpolated allowing to go smoothly from one vowel to another. Voice type can be selected but must correspond to the frequency range of the synthesized voice to be realistic.

The formant data used here come from the CSOUND manual http://www.csounds.com/manual/html/.

Usage

```
SFFormantModelBP(voiceType,vowel,exType,freq,gain) : _
```

Where:

- `voiceType`: the voice type (0: alto, 1: bass, 2: countertenor, 3: soprano, 4: tenor)
- `vowel`: the vowel (0: a, 1: e, 2: i, 3: o, 4: u
- `exType`: voice vs. fricative sound ratio (0-1 where 1 is 100% fricative)
- `freq`: the fundamental frequency of the source signal
- `gain`: linear gain multiplier to multiply the source by

`(pm.)SFFormantModelFofCycle_ui`

Ready-to-use source-filter vocal synthesizer with built-in user interface.

Usage

```
SFFormantModelFofCycle_ui : _
```

`(pm.)SFFormantModelFofSmooth_ui`

Ready-to-use source-filter vocal synthesizer with built-in user interface.

Usage

```
SFFormantModelFofSmooth_ui : _
```

`(pm.)SFFormantModelBP_ui`

Ready-to-use source-filter vocal synthesizer with built-in user interface.

Usage

```
SFFormantModelBP_ui : _
```

(pm.)SFFormantModelFofCycle_ui_MIDI

Ready-to-use MIDI-controllable source-filter vocal synthesizer.

Usage

```
SFFormantModelFofCycle_ui_MIDI : _
```

(pm.)SFFormantModelFofSmooth_ui_MIDI

Ready-to-use MIDI-controllable source-filter vocal synthesizer.

Usage

```
SFFormantModelFofSmooth_ui_MIDI : _
```

(pm.)SFFormantModelBP_ui_MIDI

Ready-to-use MIDI-controllable source-filter vocal synthesizer.

Usage

```
SFFormantModelBP_ui_MIDI : _
```

Misc Functions

Various miscellaneous functions.

(pm.)allpassNL

Bidirectional block adding nonlinearities in both directions in a chain. Nonlinearities are created by modulating the coefficients of a passive allpass filter by the signal it is processing.

Usage

```
chain(... : allpassNL(nonlinearity) : ...)
```

Where:

- `nonlinearity`: amount of nonlinearity to be added (0-1)

`modalModel`

// Implement multiple resonance modes using resonant bandpass filters.

Usage

```
_ : modalModel(n, freqs, t60s, gains) : _
```

Where:

- `n`: number of given modes
- `freqs` : list of filter center freqencies
- `t60s` : list of mode resonance durations (in seconds)
- `gains` : list of mode gains (0-1)

For example, to generate a model with 2 modes (440 Hz and 660 Hz, a fifth) where the higher one decays faster and is attenuated:

```
os.impulse : modalModel(2, (440, 660),
                           (0.5, 0.25),
                           (ba.db2linear(-1), ba.db2linear(-6)) : _
```

Further reading: Grumiaux et. al., 2017: Impulse-Response and CAD-Mod// el-Based Physical Modeling in Faust

---

platform.lib

A library to handle platform specific code in Faust. Its official prefix is `pl`.

`(pl.)SR`

Current sampling rate (between 1Hz and 192000Hz). Constant during program execution.

`(pl.)tablesize`

Oscillator table size

---

reducemaps.lib

A library to handle reduce/map kind of operation in Faust. Its official prefix is `rm`.

`(rm.)reduce`

Fold-like high order function. Apply a binary operation on a block of consecutive samples of a signal . For example : reduce(max,128) will compute the maximun of each block of 128 samples. Please note that the resulting value, while produced continuously, will be constant for the duration of a block. A new value is only produced at the end of a block. Note also that blocks should be of at least one sample (n>0).

Usage

```
reduce(op, n, x)
```

`(rm.)reducemap`

Like reduce but a foo function is applied to the result. From a mathematical point of view : reducemap(op,foo,n) is equivalent to reduce(op,n):foo but more efficient.

Usage

```
reducemap (op, foo, n, x)
```

---

reverbs.lib

A library of reverb effects. Its official prefix is `re`.

Schroeder Reverberators

`(re.)jcrev`

This artificial reverberator take a mono signal and output stereo (`satrev`) and quad (`jcrev`). They were implemented by John Chowning in the MUS10 computer-music language (descended from Music V by Max Mathews). They are Schroeder Reverberators, well tuned for their size. Nowadays, the more expensive freeverb is more commonly used (see the Faust examples directory).

`jcrev` reverb below was made from a listing of "RV", dated April 14, 1972, which was recovered from an old SAIL DART backup tape. John Chowning thinks this might be the one that became the well known and often copied JCREV.

`jcrev` is a standard Faust function

Usage

```
_ : jcrev : _,_,_,_
```

(re.)satrev

This artificial reverberator take a mono signal and output stereo (`satrev`) and quad (`jcrev`). They were implemented by John Chowning in the MUS10 computer-music language (descended from Music V by Max Mathews). They are Schroeder Reverberators, well tuned for their size. Nowadays, the more expensive freeverb is more commonly used (see the Faust examples directory).

`satrev` was made from a listing of "SATREV", dated May 15, 1971, which was recovered from an old SAIL DART backup tape. John Chowning thinks this might be the one used on his often-heard brass canon sound examples, one of which can be found at https://ccrma.stanford.edu/~jos/wav/FM_BrassCanon2.wav.

Usage

```
_ : satrev : _,_
```

Feedback Delay Network (FDN) Reverberators

(re.)fdnrev0

Pure Feedback Delay Network Reverberator (generalized for easy scaling). `fdnrev0` is a standard Faust function.

Usage

```
<1,2,4,...,N signals> <:
fdnrev0(MAXDELAY,delays,BBSO,freqs,durs,loopgainmax,nonl) :>
<1,2,4,...,N signals>
```

Where:

- `N`: 2, 4, 8, ... (power of 2)
- `MAXDELAY`: power of 2 at least as large as longest delay-line length
- `delays`: N delay lines, N a power of 2, lengths perferably coprime
- `BBSO`: odd positive integer = order of bandsplit desired at freqs
- `freqs`: NB-1 crossover frequencies separating desired frequency bands
- `durs`: NB decay times (t60) desired for the various bands
- `loopgainmax`: scalar gain between 0 and 1 used to "squelch" the reverb
- `nonl`: nonlinearity (0 to 0.999..., 0 being linear)

Reference

https://ccrma.stanford.edu/~jos/pasp/FDN_Reverberation.html

`(re.)zita_rev_fdn`

Internal 8x8 late-reverberation FDN used in the FOSS Linux reverb zita-rev1 by Fons Adriaensen fons@linuxaudio.org. This is an FDN reverb with allpass comb filters in each feedback delay in addition to the damping filters.

Usage

```
bus(8) : zita_rev_fdn(f1,f2,t60dc,t60m,fsmax) : bus(8)
```

Where:

- `f1`: crossover frequency (Hz) separating dc and midrange frequencies
- `f2`: frequency (Hz) above f1 where T60 = t60m/2 (see below)
- `t60dc`: desired decay time (t60) at frequency 0 (sec)
- `t60m`: desired decay time (t60) at midrange frequencies (sec)
- `fsmax`: maximum sampling rate to be used (Hz)

Reference

- http://www.kokkinizita.net/linuxaudio/zita-rev1-doc/quickguide.html
- https://ccrma.stanford.edu/~jos/pasp/Zita_Rev1.html

`(re.)zita_rev1_stereo`

Extend `zita_rev_fdn` to include `zita_rev1` input/output mapping in stereo mode. `zita_rev1_stereo` is a standard Faust function.

Usage

```
_,_ : zita_rev1_stereo(rdel,f1,f2,t60dc,t60m,fsmax) : _,_
```

Where:

`rdel` = delay (in ms) before reverberation begins (e.g., 0 to ~100 ms) (remaining args and refs as for `zita_rev_fdn` above)

`(re.)zita_rev1_ambi`

Extend zita_rev_fdn to include zita_rev1 input/output mapping in "ambisonics mode", as provided in the Linux C++ version.

Usage

```
_,_ : zita_rev1_ambi(rgxyz,rdel,f1,f2,t60dc,t60m,fsmax) : _,_,_,_
```

Where:

`rgxyz` = relative gain of lanes 1,4,2 to lane 0 in output (e.g., -9 to 9) (remaining args and references as for zita_rev1_stereo above)

Freeverb

`(re.)mono_freeverb`

A simple Schroeder reverberator primarily developed by "Jezar at Dreampoint" that is extensively used in the free-software world. It uses four Schroeder allpasses in series and eight parallel Schroeder-Moorer filtered-feedback comb-filters for each audio channel, and is said to be especially well tuned.

`mono_freeverb` is a standard Faust function.

Usage

```
_ : mono_freeverb(fb1, fb2, damp, spread) : _;
```

Where:

- `fb1`: coefficient of the lowpass comb filters (0-1)
- `fb2`: coefficient of the allpass comb filters (0-1)
- `damp`: damping of the lowpass comb filter (0-1)
- `spread`: spatial spread in number of samples (for stereo)

License

While this version is licensed LGPL (with exception) along with other GRAME library functions, the file freeverb.dsp in the examples directory of older Faust distributions, such as faust-0.9.85, was released under the BSD license, which is less restrictive.

`(re.)stereo_freeverb`

A simple Schroeder reverberator primarily developed by "Jezar at Dreampoint" that is extensively used in the free-software world. It uses four Schroeder allpasses in series and eight parallel Schroeder-Moorer filtered-feedback comb-filters for each audio channel, and is said to be especially well tuned.

Usage

```
_,_ : stereo_freeverb(fb1, fb2, damp, spread) : _,_;
```

Where:

- `fb1`: coefficient of the lowpass comb filters (0-1)
- `fb2`: coefficient of the allpass comb filters (0-1)
- `damp`: damping of the lowpass comb filter (0-1)
- `spread`: spatial spread in number of samples (for stereo)

---

routes.lib

A library to handle signal routing in Faust. Its official prefix is `ro`.

Functions Reference

`(ro.)cross`

Cross n signals: `(x1,x2,..,xn) -> (xn,..,x2,x1)`. `cross` is a standard Faust function.

Usage

```
cross(n)
_,_,_ : cross(3) : _,_,_
```

Where:

- `n`: number of signals (int, must be known at compile time)

Note

Special case: `cross2`:

```
cross2 = _,cross(2),_;
```

`(ro.)crossnn`

Cross two `bus(n)`s.

Usage

```
_,_,... : crossmm(n) : _,_,...
```

Where:

- `n`: the number of signals in the `bus`

`(ro.)crossn1`

Cross bus(n) and bus(1).

Usage

```
_,_,... : crossn1(n) : _,_,...
```

Where:

- `n`: the number of signals in the first `bus`

`(ro.)interleave`

Interleave row*col cables from column order to row order. input : x(0), x(1), x(2) ..., x(row*col-1) output: x(0+0*row), x(0+1*row), x(0+2*row), ..., x(1+0*row), x(1+1*row), x(1+2*row), ...

Usage

```
_,_,_,_,_,_ : interleave(row,column) : _,_,_,_,_,_
```

Where:

- `row`: the number of row (int, known at compile time)
- `column`: the number of column (int, known at compile time)

`(ro.)butterfly`

Addition (first half) then substraction (second half) of interleaved signals.

Usage

```
_,_,_,_ : butterfly(n) : _,_,_,_
```

Where:

- `n`: size of the butterfly (n is int, even and known at compile time)

`(ro.)hadamard`

Hadamard matrix function of size `n = 2^k`.

Usage

```
_,_,_,_ : hadamard(n) : _,_,_,_
```

Where:

- `n`: `2^k`, size of the matrix (int, must be known at compile time)

Note:

Implementation contributed by Remy Muller.

`(ro.)recursivize`

Create a recursion from two arbitrary processors p and q.

Usage

```
_,_ : recursivize(p,q) : _,_
```

Where:

- `p`: the forward arbitrary processor
- `q`: the feedback arbitrary processor

---

signals.lib

A library of basic elements to handle signals in Faust. Its official prefix is `si`.

Functions Reference

`(si.)bus`

n parallel cables. `bus` is a standard Faust function.

Usage

```
bus(n)
bus(4) : _,_,_,_
```

Where:

- `n`: is an integer known at compile time that indicates the number of parallel cables.

`(si.)block`

Block - terminate n signals. `block` is a standard Faust function.

Usage

```
_,_,... : block(n) : _,...
```

Where:

- `n`: the number of signals to be blocked

`(si.)interpolate`

Linear interpolation between two signals.

Usage

```
_,_ : interpolate(i) : _
```

Where:

- `i`: interpolation control between 0 and 1 (0: first input; 1: second input)

`(si.)smoo`

Smoothing function based on `smooth` ideal to smooth UI signals (sliders, etc.) down. `smoo` is a standard Faust function.

Usage

```
hslider(...) : smoo;
```

`(si.)polySmooth`

A smoothing function based on `smooth` that doesn't smooth when a trigger signal is given. This is very useful when making polyphonic synthesizer to make sure that the value of the parameter is the right one when the note is started.

Usage

```
hslider(...) : polySmooth(g,s,d) : _
```

Where:

- `g`: the gate/trigger signal used when making polyphonic synths
- `s`: the smoothness (see `smooth`)

- `d`: the number of samples to wait before the signal start being smoothed after `g` switched to 1

`(si.)smoothAndH`

A smoothing function based on `smooth` that holds its output signal when a trigger is sent to it. This feature is convenient when implementing polyphonic instruments to prevent some smoothed parameter to change when a note-off event is sent.

Usage

```
hslider(...) : smoothAndH(g,s) : _
```

Where:

- `g`: the hold signal (0 for hold, 1 for bypass)
- `s`: the smoothness (see `smooth`)

`(si.)bsmooth`

Block smooth linear interpolation during a block of samples.

Usage

```
hslider(...) : bsmooth : _
```

`(si.)dot`

Dot product for two vectors of size n.

Usage

```
_,_,_,_,_,_ : dot(n) : _
```

Where:

- `n`: size of the vectors (int, must be known at compile time)

`(si.)smooth`

Exponential smoothing by a unity-dc-gain one-pole lowpass. `smooth` is a standard Faust function.

Usage:

```
_ : smooth(tau2pole(tau)) : _
```

Where:

- `tau`: desired smoothing time constant in seconds, or

```
hslider(...) : smooth(s) : _
```

Where:

- `s`: smoothness between 0 and 1. s=0 for no smoothing, s=0.999 is "very smooth", s>1 is unstable, and s=1 yields the zero signal for all inputs. The exponential time-constant is approximately 1/(1-s) samples, when s is close to (but less than) 1.

Reference:

https://ccrma.stanford.edu/~jos/mdft/Convolution_Example_2_ADSR.html

---

`(si.)cbus`

n parallel cables for complex signals. `cbus` is a standard Faust function.

Usage

```
cbus(n)
cbus(4) : (r0,i0), (r1,i1), (r2,i2), (r3,i3)
```

Where:

- `n`: is an integer known at compile time that indicates the number of parallel cables.
- each complex number is represented by two real signals as (real,imag)

`(si.)cmul`

multiply two complex signals pointwise. `cmul` is a standard Faust function.

Usage

```
(r1,i1) : cmul(r2,i2) : (_,_);
```

Where:

- Each complex number is represented by two real signals as (real,imag), so
- `(r1,i1)` = real and imaginary parts of signal 1
- `(r2,i2)` = real and imaginary parts of signal 2

`(si.)cconj`

complex conjugation of a (complex) signal. `cconj` is a standard Faust function.

Usage

```
(r1,i1) : cconj : (_,_);
```

Where:

- Each complex number is represented by two real signals as (real,imag), so
- `(r1,i1)` = real and imaginary parts of the input signal
- `(r1,-i1)` = real and imaginary parts of the output signal

`(si.)lag_ud`

Lag filter with separate times for up and down.

Usage

```
_ : lag_ud(up, dn) : _;
```

`(si.)rev`

Reverse the input signal by blocks of N>0 samples. `rev(1)` is the indentity function. `rev(N)` has a latency of `N-1` samples.

Usage

```
_ : rev(N) : _;
```

Where:

- `N`: the block size

---

soundfiles.lib

A library to handle soundfiles in Faust. Its official prefix is `so`.

Functions Reference

`(so.)loop`

Play a soundfile in a loop taking into account its sampling rate `loop` is a standard Faust function.

Usage

```
loop(sf, part)
```

Where:

- `sf`: the soundfile
- `part`: the part in the soundfile list of sounds

`(so.)loop_speed`

Play a soundfile in a loop taking into account its sampling rate, with speed control `loop_speed` is a standard Faust function.

Usage

```
loop_speed(sf, part, speed)
```

Where:

- `sf`: the soundfile
- `part`: the part in the soundfile list of sounds
- `speed`: the speed between 0 and n

`(so.)loop_speed_level`

Play a soundfile in a loop taking into account its sampling rate, with speed and level controls `loop_speed_level` is a standard Faust function.

Usage

```
loop_speed_level(sf, part, speed, level)
```

Where:

- `sf`: the soundfile
- `part`: the part in the soundfile list of sounds
- `speed`: the speed between 0 and n
- `level`: the volume between 0 and n

---

spats.lib

This library contains a collection of tools for sound spatialization. Its official prefix is `sp`.

`(sp.)panner`

A simple linear stereo panner. `panner` is a standard Faust function.

Usage

```
_ : panner(g) : _,_
```

Where:

- `g`: the panning (0-1)

`(sp.)spat`

GMEM SPAT: n-outputs spatializer. `spat` is a standard Faust function.

Usage

```
_ : spat(n,r,d) : _,_,...
```

Where:

- `n`: number of outputs
- `r`: rotation (between 0 et 1)
- `d`: distance of the source (between 0 et 1)

`(sp.)stereoize`

Transform an arbitrary processor `p` into a stereo processor with 2 inputs and 2 outputs.

Usage

```
_,_ : stereoize(p) : _,_
```

Where:

- `p`: the arbitrary processor

---

synths.lib

This library contains a collection of synthesizers. Its official prefix is `sy`.

`(sy.)popFilterPerc`

A simple percussion instrument based on a "popped" resonant bandpass filter. `popFilterPerc` is a standard Faust function.

Usage

```
popFilterDrum(freq,q,gate) : _;
```

Where:

- `freq`: the resonance frequency of the instrument
- `q`: the q of the res filter (typically, 5 is a good value)
- `gate`: the trigger signal (0 or 1)

`(sy.)dubDub`

A simple synth based on a sawtooth wave filtered by a resonant lowpass. `dubDub` is a standard Faust function.

Usage

```
dubDub(freq,ctFreq,q,gate) : _;
```

Where:

- `freq`: frequency of the sawtooth
- `ctFreq`: cutoff frequency of the filter
- `q`: Q of the filter
- `gate`: the trigger signal (0 or 1)

`(sy.)sawTrombone`

A simple trombone based on a lowpassed sawtooth wave. `sawTrombone` is a standard Faust function.

Usage

```
sawTrombone(att,freq,gain,gate) : _
```

Where:

- `att`: exponential attack duration in s (typically 0.01)
- `freq`: the frequency

- `gain`: the gain (0-1)
- `gate`: the gate (0 or 1)

`(sy.)combString`

Simplest string physical model ever based on a comb filter. `combString` is a standard Faust function.

Usage

```
combString(freq,res,gate) : _;
```

Where:

- `freq`: the frequency of the string
- `res`: string T60 (resonance time) in second
- `gate`: trigger signal (0 or 1)

`(sy.)additiveDrum`

A simple drum using additive synthesis. `additiveDrum` is a standard Faust function.

Usage

```
additiveDrum(freq,freqRatio,gain,harmDec,att,rel,gate) : _
```

Where:

- `freq`: the resonance frequency of the drum
- `freqRatio`: a list of ratio to choose the frequency of the mode in function of `freq` e.g.(1 1.2 1.5 ...). The first element should always be one (fundamental).
- `gain`: the gain of each mode as a list (1 0.9 0.8 ...). The first element is the gain of the fundamental.
- `harmDec`: harmonic decay ratio (0-1): configure the speed at which higher modes decay compare to lower modes.
- `att`: attack duration in second
- `rel`: release duration in second
- `gate`: trigger signal (0 or 1)

`(sy.)fm`

An FM synthesizer with an arbitrary number of modulators connected as a sequence. `fm` is a standard Faust function.

Usage

```
freqs = (300,400,...);
indices = (20,...);
fm(freqs,indices) : _
```

Where:

- `freqs`: a list of frequencies where the first one is the frequency of the carrier and the others, the frequency of the modulator(s)
- `indices`: the indices of modulation (Nfreqs-1)

---

vaeffects.lib

A library of virtual analog filter effects. Its official prefix is `ve`.

Moog Filters

`(ve.)moog_vcf`

Moog "Voltage Controlled Filter" (VCF) in "analog" form. Moog VCF implemented using the same logical block diagram as the classic analog circuit. As such, it neglects the one-sample delay associated with the feedback path around the four one-poles. This extra delay alters the response, especially at high frequencies (see reference [1] for details). See `moog_vcf_2b` below for a more accurate implementation.

Usage

```
moog_vcf(res,fr)
```

Where:

- `res`: normalized amount of corner-resonance between 0 and 1 (0 is no resonance, 1 is maximum)
- `fr`: corner-resonance frequency in Hz (less than SR/6.3 or so)

References

- https://ccrma.stanford.edu/~stilti/papers/moogvcf.pdf
- https://ccrma.stanford.edu/~jos/pasp/vegf.html

`(ve.)moog_vcf_2b[n]`

Moog "Voltage Controlled Filter" (VCF) as two biquads. Implementation of the ideal Moog VCF transfer function factored into second-order sections. As

a result, it is more accurate than `moog_vcf` above, but its coefficient formulas are more complex when one or both parameters are varied. Here, res is the fourth root of that in `moog_vcf`, so, as the sampling rate approaches infinity, `moog_vcf(res,fr)` becomes equivalent to `moog_vcf_2b[n](res^4,fr)` (when res and fr are constant). `moog_vcf_2b` uses two direct-form biquads (`tf2`). `moog_vcf_2bn` uses two protected normalized-ladder biquads (`tf2np`).

Usage

```
moog_vcf_2b(res,fr)
moog_vcf_2bn(res,fr)
```

Where:

- `res`: normalized amount of corner-resonance between 0 and 1 (0 is min resonance, 1 is maximum)
- `fr`: corner-resonance frequency in Hz

`(ve.)moogLadder`

Virtual analog model of the 4th-order Moog Ladder, which is arguably the most well-known ladder filter in analog synthesizers. Several 1st-order filters are cascaded in series. Feedback is then used, in part, to control the cut-off frequency and the resonance.

This filter was implemented in Faust by Eric Tarr during the 2019 Embedded DSP With Faust Workshop.

References

- https://www.willpirkle.com/706-2/
- http://www.willpirkle.com/Downloads/AN-4VirtualAnalogFilters.pdf

Usage

```
_ : moogLadder(normFreq,Q) : _
```

Where:

- `normFreq`: normalized frequency (0-1)
- `Q`: q

`(ve.)moogHalfLadder`

Virtual analog model of the 2nd-order Moog Half Ladder (simplified version of `(ve.)moogLadder`). Several 1st-order filters are cascaded in series. Feedback is then used, in part, to control the cut-off frequency and the resonance.

This filter was implemented in Faust by Eric Tarr during the 2019 Embedded DSP With Faust Workshop.

References

- https://www.willpirkle.com/app-notes/virtual-analog-moog-half-ladder-filter
- http://www.willpirkle.com/Downloads/AN-8MoogHalfLadderFilter.pdf

Usage

```
_ : moogHalfLadder(normFreq,Q) : _
```

Where:

- `normFreq`: normalized frequency (0-1)
- `Q`: q

`(ve.)diodeLadder`

4th order virtual analog diode ladder filter. In addition to the individual states used within each independent 1st-order filter, there are also additional feedback paths found in the block diagram. These feedback paths are labeled as connecting states. Rather than separately storing these connecting states in the Faust implementation, they are simply implicitly calculated by tracing back to the other states (s1,s2,s3,s4) each recursive step.

This filter was implemented in Faust by Eric Tarr during the 2019 Embedded DSP With Faust Workshop.

References

- https://www.willpirkle.com/virtual-analog-diode-ladder-filter/
- http://www.willpirkle.com/Downloads/AN-6DiodeLadderFilter.pdf

Usage

```
_ : diodeLadder(normFreq,Q) : _
```

Where:

- `normFreq`: normalized frequency (0-1)
- `Q`: q

Korg 35 Filters

The following filters are virtual analog models of the Korg 35 low-pass filter and high-pass filter found in the MS-10 and MS-20 synthesizers. The virtual analog models for the LPF and HPF are different, making these filters more interesting than simply tapping different states of the same circuit.

These filters were implemented in Faust by Eric Tarr during the 2019 Embedded DSP With Faust Workshop.

Filter history:

https://secretlifeofsynthesizers.com/the-korg-35-filter/

`(ve.)korg35LPF`

Virtual analog models of the Korg 35 low-pass filter found in the MS-10 and MS-20 synthesizers.

Usage

```
_ : korg35LPF(normFreq,Q) : _
```

Where:

- `normFreq`: normalized frequency (0-1)
- `Q`: q

`(ve.)korg35HPF`

Virtual analog models of the Korg 35 high-pass filter found in the MS-10 and MS-20 synthesizers.

Usage

```
_ : korg35HPF(normFreq,Q) : _
```

Where:

- `normFreq`: normalized frequency (0-1)
- `Q`: q

Oberheim Filters

The following filter (4 types) is an implementation of the virtual analog model described in Section 7.2 of the Will Pirkle book, "Designing Software Synthesizer Plug-ins in C++. It is based on the block diagram in Figure 7.5.

The Oberheim filter is a state-variable filter with soft-clipping distortion within the circuit.

In many VA filters, distortion is accomplished using the "tanh" function. For this Faust implementation, that distortion function was replaced with the `(ef.)cubicnl` function.

`(ve.)oberheim`

Generic multi-outputs Oberheim filter (see description above).

Usage

```
_ : oberheim(normFreq,Q) : _,_,_,_
```

Where:

- `normFreq`: normalized frequency (0-1)
- `Q`: q

`(ve.)oberheimBSF`

Band-Stop Oberheim filter (see description above).

Usage

```
_ : oberheimBSF(normFreq,Q) : _
```

Where:

- `normFreq`: normalized frequency (0-1)
- `Q`: q

`(ve.)oberheimBPF`

Band-Pass Oberheim filter (see description above).

Usage

```
_ : oberheimBPF(normFreq,Q) : _
```

Where:

- `normFreq`: normalized frequency (0-1)
- `Q`: q

`(ve.)oberheimHPF`

High-Pass Oberheim filter (see description above).

Usage

```
_ : oberheimHPF(normFreq,Q) : _
```

Where:

- `normFreq`: normalized frequency (0-1)
- `Q`: q

`(ve.)oberheimLPF`

Low-Pass Oberheim filter (see description above).

Usage

```
_ : oberheimLPF(normFreq,Q) : _
```

Where:

- `normFreq`: normalized frequency (0-1)
- `Q`: q

Sallen Key Filters

The following filters were implemented based on VA models of synthesizer filters.

The modeling approach is based on a Topology Preserving Transform (TPT) to resolve the delay-free feedback loop in the corresponding analog filters.

The primary processing block used to build other filters (Moog, Korg, etc.) is based on a 1st-order Sallen-Key filter.

The filters included in this script are 1st-order LPF/HPF and 2nd-order state-variable filters capable of LPF, HPF, and BPF.

Resources:

- Vadim Zavalishin (2018) "The Art of VA Filter Design", v2.1.0 https://www.native-instruments.com/fileadmin/ni_media/downloads/pdf/VAFilterDesign_2.1.0.pdf
- Will Pirkle (2014) "Resolving Delay-Free Loops in Recursive Filters Using the Modified Härmä Method", AES 137 http://www.aes.org/e-lib/browse.cfm?elib=17517

- Description and diagrams of 1st- and 2nd-order TPT filters: https://www.willpirkle.com/706-2/

`(ve.)sallenKeyOnePole`

Sallen-Key generic One Pole filter (see description above).

For the Faust implementation of this filter, recursion (`letrec`) is used for storing filter "states". The output (e.g. `y`) is calculated by using the input signal and the previous states of the filter. During the current recursive step, the states of the filter (e.g. `s`) for the next step are also calculated. Admittedly, this is not an efficient way to implement a filter because it requires independently calculating the output and each state during each recursive step. However, it works as a way to store and use "states" within the constraints of Faust.

`(ve.)sallenKeyOnePoleLPF`

Sallen-Key One Pole lowpass filter (see description above).

Usage

```
_ : sallenKeyOnePoleLPF(normFreq) : _
```

Where:

-

`normFreq`: normalized frequency (0-1)

`(ve.)sallenKeyOnePoleHPF`

Sallen-Key One Pole Highpass filter (see description above). The dry input signal is routed in parallel to the output. The LPF'd signal is subtracted from the input so that the HPF remains.

Usage

```
_ : sallenKeyOnePoleHPF(normFreq) : _
```

Where:

- `normFreq`: normalized frequency (0-1)

`(ve.)sallenKey2ndOrder`

Sallen-Key generic multi-outputs 2nd order filter.

This is a 2nd-order Sallen-Key state-variable filter. The idea is that by "tapping" into different points in the circuit, different filters (LPF,BPF,HPF) can be achieved. See Figure 4.6 of https://www.willpirkle.com/706-2/

This is also a good example of the next step for generalizing the Faust programming approach used for all these VA filters. In this case, there are three things to calculate each recursive step (y,s1,s2). For each thing, the circuit is only calculated up to that point.

Comparing the LPF to BPF, the output signal (y) is calculated similarly. Except, the output of the BPF stops earlier in the circuit. Similarly, the states (s1 and s2) only differ in that s2 includes a couple more terms beyond what is used for s1.

Usage

```
_ : sallenKey2ndOrder(normFreq,Q) : _,_,_
```

Where:

- `normFreq`: normalized frequency (0-1)
- `Q`: q

`(ve.)sallenKey2ndOrderLPF`

Sallen-Key 2nd order lowpass filter (see description above).

Usage

```
_ : sallenKey2ndOrderLPF(normFreq,Q) : _
```

Where:

- `normFreq`: normalized frequency (0-1)
- `Q`: q

`(ve.)sallenKey2ndOrderBPF`

Sallen-Key 2nd order bandpass filter (see description above).

Usage

```
_ : sallenKey2ndOrderBPF(normFreq,Q) : _
```

Where:

- `normFreq`: normalized frequency (0-1)
- `Q`: q

`(ve.)sallenKey2ndOrderHPF`

Sallen-Key 2nd order highpass filter (see description above).

Usage

```
_ : sallenKey2ndOrderHPF(normFreq,Q) : _
```

Where:

- `normFreq`: normalized frequency (0-1)
- `Q`: q

Effects

`(ve.)wah4`

Wah effect, 4th order. `wah4` is a standard Faust function.

Usage

```
_ : wah4(fr) : _
```

Where:

- `fr`: resonance frequency in Hz

Reference

https://ccrma.stanford.edu/~jos/pasp/vegf.html

`(ve.)autowah`

Auto-wah effect. `autowah` is a standard Faust function.

Usage

```
_ : autowah(level) : _
```

Where:

- `level`: amount of effect desired (0 to 1).

`(ve.)crybaby`

Digitized CryBaby wah pedal. `crybaby` is a standard Faust function.

Usage

```
_ : crybaby(wah) : _
```

Where:

- `wah`: "pedal angle" from 0 to 1

Reference

https://ccrma.stanford.edu/~jos/pasp/vegf.html

`(ve.)vocoder`

A very simple vocoder where the spectrum of the modulation signal is analyzed using a filter bank. `vocoder` is a standard Faust function.

Usage

```
_ : vocoder(nBands,att,rel,BWRatio,source,excitation) : _;
```

Where:

- `nBands`: Number of vocoder bands
- `att`: Attack time in seconds
- `rel`: Release time in seconds
- `BWRatio`: Coefficient to adjust the bandwidth of each band (0.1 - 2)
- `source`: Modulation signal
- `excitation`: Excitation/Carrier signal

---

version.lib

Semantic versioning for the Faust libraries. Its official prefix is `vl`.

`(vl.)version`

Return the version number of the Faust standard libraries.

Usage

```
version : _,_,_
```

---

webaudio.lib

An implementation of the web audio API filters. Its official prefix is `wa`.

`(wa.)lowpass2`

Standard second-order resonant lowpass filter with 12dB/octave rolloff. Frequencies below the cutoff pass through; frequencies above it are attenuated.

Usage

```
_: lowpass2(f0, Q, dtune) :_
```

Where:

- `f0`: cutoff frequency in Hz
- `Q`: the quality factor
- `dtune`: detuning of the frequency in cents

Reference

https://searchfox.org/mozilla-central/source/dom/media/webaudio/blink/Biquad.cpp#98

`(wa.)highpass2`

Standard second-order resonant highpass filter with 12dB/octave rolloff. Frequencies below the cutoff are attenuated; frequencies above it pass through.

Usage

```
_: highpass2(f0, Q, dtune) :_
```

Where:

- `f0`: cutoff frequency in Hz
- `Q`: the quality factor
- `dtune`: detuning of the frequency in cents

Reference

https://searchfox.org/mozilla-central/source/dom/media/webaudio/blink/Biquad.cpp#127

`(wa.)bandpass2`

Standard second-order bandpass filter. Frequencies outside the given range of frequencies are attenuated; the frequencies inside it pass through.

Usage

```
_: bandpass2(f0, Q, dtune) :_
```

Where:

- `f0`: cutoff frequency in Hz
- `Q`: the quality factor
- `dtune`: detuning of the frequency in cents

Reference

https://searchfox.org/mozilla-central/source/dom/media/webaudio/blink/Biquad.cpp#334

`(wa.)notch2`

Standard notch filter, also called a band-stop or band-rejection filter. It is the opposite of a bandpass filter: frequencies outside the give range of frequencies pass through, frequencies inside it are attenuated.

Usage

```
_: notch2(f0, Q, dtune) :_
```

Where:

- `f0`: cutoff frequency in Hz
- `Q`: the quality factor
- `dtune`: detuning of the frequency in cents

Reference

https://searchfox.org/mozilla-central/source/dom/media/webaudio/blink/Biquad.cpp#301

`(wa.)allpass2`

Standard second-order allpass filter. It lets all frequencies through, but changes the phase-relationship between the various frequencies.

Usage

```
_: allpass2(f0, Q, dtune) :_
```

Where:

- `f0`: cutoff frequency in Hz
- `Q`: the quality factor
- `dtune`: detuning of the frequency in cents

Reference

https://searchfox.org/mozilla-central/source/dom/media/webaudio/blink/Biquad.cpp#268

`(wa.)peaking2`

Frequencies inside the range get a boost or an attenuation; frequencies outside it are unchanged.

Usage

```
_: peaking2(f0, gain, Q, dtune) :_
```

Where:

- `f0`: cutoff frequency in Hz
- `gain`: the gain in dB
- `Q`: the quality factor
- `dtune`: detuning of the frequency in cents

Reference

https://searchfox.org/mozilla-central/source/dom/media/webaudio/blink/Biquad.cpp#233

`(wa.)lowshelf2`

Standard second-order lowshelf filter. Frequencies lower than the frequency get a boost, or an attenuation, frequencies over it are unchanged.

```
_: lowshelf2(f0, gain, dtune) :_
```

Where:

- `f0`: cutoff frequency in Hz
- `gain`: the gain in dB
- `dtune`: detuning of the frequency in cents

Reference

https://searchfox.org/mozilla-central/source/dom/media/webaudio/blink/Biquad.cpp#169

`(wa.)highshelf2`

Standard second-order highshelf filter. Frequencies higher than the frequency get a boost or an attenuation, frequencies lower than it are unchanged.

`_: highshelf2(f0, gain, dtune) :_`

Where:

- `f0`: cutoff frequency in Hz
- `gain`: the gain in dB
- `dtune`: detuning of the frequency in cents

Reference

https://searchfox.org/mozilla-central/source/dom/media/webaudio/blink/Biquad.cpp#201

Licenses

STK 4.3 License

PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

LGPL License

This program is free software; you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation; either version 2.1 of the License, or (at your option) any later version.

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

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