Faust Standard Libraries

Contents

Faust Libraries

NOTE: this documentation was automatically generated.

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 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:

- `ma`: `math.lib`
- `ba`: `basic.lib`
- `de`: `delay.lib`
- `en`: `envelope.lib`
- `ro`: `route.lib`
- `si`: `signal.lib`
- `an`: `analyzer.lib`
- `fi`: `filter.lib`
- `os`: `miscoscillator.lib`
- `no`: `noise.lib`
- `ho`: `hoa.lib`
- `sp`: `spat.lib`
- `sy`: `synth.lib`
- `ef`: `misceffect.lib`
- `ve`: `vaeffect.lib`
- `co`: `compressor.lib`
- `pf`: `phafla.lib`

- `re`: `reverb.lib`
- `de`: `demo.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 `miscoscillator.lib`.

Alternatively, environments can be created by hand:

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

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

```
import("miscoscillator.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 descirption
//-------------------------------------------------
````

- 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 exisiting 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:

- `analyzer.lib`
- `basic.lib`
- `delay.lib`
- `misceffect.lib`
- `compressor.lib`
- `vaeffect.lib`
- `phafla.lib`
- `reverb.lib`
- `envelope.lib`
- `filter.lib`
- `miscoscillator.lib`
- `noise.lib`
- `hoa.lib`
- `math.lib`
- `pm.lib`
- `route.lib`
- `signal.lib`
- `spat.lib`
- `synth.lib`
- `demo.lib`
- `tonestack.lib` (not documented but example in `/examples/misc`)
- `tube.lib` (not documented but example in `/examples/misc`)

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

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

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 `basic.lib` for an example).
- Sections in function documentation should be declared as `####` markdown title.
- Each function documentation provides a "Usage" section (see `basic.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:

```
ma = library("math.lib");
ba = library("basic.lib");
de = library("delay.lib");
en = library("envelope.lib");
ro = library("route.lib");
si = library("signal.lib");
an = library("analyzer.lib");
fi = library("filter.lib");
os = library("miscoscillator.lib");
no = library("noise.lib");
ho = library("hoa.lib");
sp = library("spat.lib");
sy = library("synth.lib");
ef = library("misceffect.lib");
ve = library("vaeffect.lib");
co = library("compressor.lib");
pf = library("phafla.lib");
re = library("reverb.lib");
```

For example, if we wanted to use the `smooth` function which is now declared in `signal.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 `signal.lib` directly and call `smooth` without `ro.`, etc.

"Demo" Functions

All the functions that were present in the libraries and that contained any kind of UI elements declaration (mostly JOS "`demo`" functions) were turned into independant `.dsp` files that were placed in the `/examples` folder. Thus, Faust libraries now only contain "pure" function declarations which should make them more legible. Also, "`demo`" functions make great examples...

For practicality, the "`demo`" functions are still declared and are available in `demo.lib` as "components" pointing at the `/examples` folder (which is why that folder will have to be installed on the system during the installation process of the Faust distribution).

The question of licensing/authoring/copyrigth

Now that Faust libraries are not author specific, each function will be able to have its own licence/author declaration. This means that some libraries wont have a global licence/author/copyright declaration like it used to be the case.

analyzer.lib

This library contains a collection of tools to analyze signals.

It should be used using the `an` environment:

```
an = library("analyzer.lib");
process = an.functionCall;
```

Another option is to import `stdfaust.lib` which already contains the `an` environment:

```
import("stdfaust.lib");
process = an.functionCall;
```

Amplitude Tracking

`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.

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

---

`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

---

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) : _;
```

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 `filter.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/

`mth_octave_analyzer[N]`

Octave analyzer.

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.

`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);
```

`[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`.

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.

`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)) : _,_,_
```

---

basic.lib

A library of basic elements for Faust organized in 5 sections:

- Conversion Tools
- Counters and Time/Tempo Tools
- Array Processing/Pattern Matching
- Selectors (Conditions)
- Other Tools (Misc)

It should be used using the `ba` environment:

```
ba = library("basic.lib");
process = ba.functionCall;
```

Another option is to import `stdfaust.lib` which already contains the `ba` environment:

```
import("stdfaust.lib");
process = ba.functionCall;
```

Conversion Tools

`samp2sec`

Converts a number of samples to a duration in seconds.

Usage

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

Where:

- `n`: number of samples

`sec2samp`

Converts a duration in seconds to a number of samples.

Usage

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

Where:

- `d`: duration in seconds

`db2linear`

Converts a loudness in dB to a linear gain (0-1).

Usage

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

Where:

- `l`: loudness in dB

`linear2db`

Converts a linear gain (0-1) to a loudness in dB.

Usage

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

Where:

- `g`: a linear gain

`lin2LogGain`

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

Usage

```
_ : lin2LogGain : _
```

---

log2LinGain

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

Usage

`_ : log2LinGain : _`

---

tau2pole

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

Usage

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

Where:

- `tau`: time-constant in seconds

---

pole2tau

Returns the time-constant, in seconds, corresponding to the given real, positive pole in (0,1).

Usage

`pole2tau(pole) : _`

Where:

- `pole`: the pole

---

midikey2hz

Converts a MIDI key number to a frequency in Hz (MIDI key 69 = A440).

Usage

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

Where:

- `mk`: the MIDI key number

`pianokey2hz`

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

Usage

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

Where:

- `pk`: the piano key number

`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

`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.

Usage

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

Where:

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

`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.

Usage

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

Where:

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

`sweep`

Counts from 0 to 'period' samples repeatedly, while 'run' is 1. Outsputs zero while 'run' is 0.

Usage

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

`time`

A simple timer that counts every samples from the beginning of the process.

Usage

```
time : _
```

`tempo`

Converts a tempo in BPM into a number of samples.

Usage

`tempo(t) : _`

Where:

- `t`: tempo in BPM

`period`

Basic sawtooth wave of period `p`.

Usage

`period(p) : _`

Where:

- `p`: period as a number of samples

`pulse`

Pulses (10000) generated at period `p`.

Usage

`pulse(p) : _`

Where:

- `p`: period as a number of samples

`pulsen`

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

Usage

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

Where:

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

`beat`

Pulses at tempo `t`.

Usage

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

Where:

- `t`: tempo in BPM

Array Processing/Pattern Matching

`count`

Count the number of elements of list l.

Usage

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

Where:

- `l`: list of elements

`take`

Take an element from a list.

Usage

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

Where:

- `p`: position (starting at 1)
- `l`: list of elements

`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)

`if`

if-then-else implemented with a select2.

Usage

- `if(c, t, e) : _`

Where:

- `c`: condition

- `t`: signal selected while c is true
- `e`: signal selected while c is false

`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`)

`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)));
```

`select2stereo`

Select between 2 stereo signals.

Usage

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

Where:

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

Other

`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)

`sAndH`

Sample And Hold.

Usage

`_ : sAndH(t) : _`

Where:

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

`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.

`peakholder`

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

Usage

```
_ : peakholder(holdtime) : _;
```

`impulsify`

Turns the signal from a button into an impulse (1,0,0,... when button turns on).

Usage

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

`automat`

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

Usage

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

Break Point Functions

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}`

`bypass1`

Takes a mono input signal, route it to `e` and bypass it if `bpc = 1`.

Usage

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

Where:

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

`bypass2`

Takes a stereo input signal, route it to `e` and bypass it if `bpc = 1`.

Usage

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

Where:

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

---

compressor.lib

A library of compressor effects.

It should be used using the `co` environment:

```
co = library("compressor.lib");
process = co.functionCall;
```

Another option is to import `stdfaust.lib` which already contains the `co` environment:

```
import("stdfaust.lib");
process = co.functionCall;
```

Functions Reference

`compressor_mono` and `compressor_stereo`

Mono and stereo dynamic range compressors.

Usage

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

---

`limiter_*`

A limiter guards against hard-clipping. It can be 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_mono : _;
_,_ : limiter_1176_R4_stereo : _,_;
```

Reference:

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

---

delay.lib

This library contains a collection of delay functions.

It should be used using the `de` environment:

```
de = library("delay.lib");
process = de.functionCall;
```

Another option is to import `stdfaust.lib` which already contains the `de` environment:

```
import("stdfaust.lib");
process = de.functionCall;
```

Basic Delay Functions

`delay`

Simple `d` samples delay where `n` is the maximum delay length as a number of samples (it needs to be a power of 2). Unlike the `@` delay operator, this function allows to preallocate memory which means that `d` can be changed dynamically at run time as long as it remains smaller than `n`.

Usage

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

Where:

- `n`: the max delay length as a power of 2
- `d`: the delay length as a number of samples (integer)

`fdelay`

Simple `d` samples fractional delay based on 2 interpolated delay lines where `n` is the maximum delay length as a number of samples (it needs to be a power of 2 - see `delay()`).

Usage

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

Where:

- `n`: the max delay length as a power of 2
- `d`: the delay length as a number of samples (float)

`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 (must be a constant power of 2, for example 65536)
- `it`: interpolation time (in samples) for example 1024
- `dt`: delay time (in samples)

Lagrange Interpolation

`fdelaylti` and `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
- 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.

`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

`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]

---

demo.lib

This library contains a set of demo functions based on examples located in the `/examples` folder.

It should be used using the `dm` environment:

```
dm = library("demo.lib");
process = dm.functionCall;
```

Another option is to import `stdfaust.lib` which already contains the `dm` environment:

```
import("stdfaust.lib");
process = dm.functionCall;
```

Analyzers

`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

`parametric_eq_demo`

A parametric equalizer application.

Usage:

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

`spectral_tilt_demo`

A spectral tilt application.

Usage

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

Where:

- `N`: filter order (integer)

All other parameters interactive

`mth_octave_filterbank_demo` and `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

`cubicnl_demo`

Distortion demo application.

Usage:

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

`gate_demo`

Gate demo application.

Usage

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

`compressor_demo`

Compressor demo application.

Usage

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

---

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

---

moog_vcf_demo

Illustrate and compare all three Moog VCF implementations above.

Usage

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

---

wah4_demo

Wah pedal application.

Usage

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

---

crybaby_demo

Crybaby effect application.

Usage

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

---

vocoder_demo

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

Usage

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

---

flanger_demo

Flanger effect application.

Usage

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

---

phaser2_demo

Phaser effect demo application.

Usage

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

---

freeverb_demo

Freeverb demo application.

Usage

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

---

stereo_reverb_tester

Handy test inputs for reverberator demos below.

Usage

```
_ : stereo_reverb_tester : _
```

---

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

---

zita_rev_fdn_demo

Reverb demo application based on `zita_rev_fdn`.

Usage

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

---

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

`sawtooth_demo`

An application demonstrating the different sawtooth oscillators of Faust.

Usage

```
sawtooth_demo : _
```

`virtual_analog_oscillator_demo`

Virtual analog oscillator demo application.

Usage

```
virtual_analog_oscillator_demo : _
```

`oscrs_demo`

Simple application demoing filter based oscillators.

Usage

```
oscrs_demo : _
```

---

envelope.lib

This library contains a collection of envelope generators.

It should be used using the `en` environment:

```
en = library("envelope.lib");
process = en.functionCall;
```

Another option is to import `stdfaust.lib` which already contains the `en` environment:

```
import("stdfaust.lib");
process = en.functionCall;
```

Functions Reference

`smoothEnvelope`

An envelope with an exponential attack and release.

Usage

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

- `ar`: attack and release duration (s)
- `t`: trigger signal (0-1)

`ar`

AR (Attack, Release) envelope generator (useful to create percussion envelopes).

Usage

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

Where:

- `a`: attack (sec)
- `r`: release (sec)
- `t`: trigger signal (0 or 1)

`asr`

ASR (Attack, Sustain, Release) envelope generator.

Usage

`asr(a,s,r,t) : _`

Where:

- `a`, `s`, `r`: attack (sec), sustain (percentage of t), release (sec)
- `t`: trigger signal (>0 for attack, then release is when t back to 0)

`adsr`

ADSR (Attack, Decay, Sustain, Release) envelope generator.

Usage

`adsr(a,d,s,r,t) : _`

Where:

- `a`, `d`, `s`, `r`: attack (sec), decay (sec), sustain (percentage of t), release (sec)
- `t`: trigger signal (>0 for attack, then release is when t back to 0)

---

filter.lib

A library of filters and of more advanced filter-based sound processor organized in 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
- Arbritary-Crossover Filter-Banks and Spectrum Analyzers

It should be used using the `fi` environment:

```
fi = library("filter.lib");
process = fi.functionCall;
```

Another option is to import `stdfaust.lib` which already contains the `fi` environment:

```
import("stdfaust.lib");
process = fi.functionCall;
```

Basic Filters

`zero`

One zero filter. Difference equation: y(n) = x(n) - z * x(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

`pole`

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

Usage

```
_ : pole(z) : _
```

Where:

- `p`: pole location = feedback coefficient

Reference

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

`integrator`

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

`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) = 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

---

`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).

Usage

```
_ : dcblocker : _
```

Comb Filters

`ff_comb` and `ff_fcomb`

Feed-Forward Comb Filter. Note that `ff_comb` requires integer delays (uses `delay()` internally) while `ff_fcomb` takes floating-point delays (uses `fdelay()` internally).

Usage

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

---

`ffcombfilter`

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

`fb_comb` and `fb_fcomb`

Feed-Back Comb Filter.

Usage

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

---

`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 `reverb.lib` for usage examples.

`fbcombfilter` and `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

---

`allpass_comb` and `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.

Usage

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

---

`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 `reverb.lib` for usage examples.

`allpass_fcomb5` and `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

`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".

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

---

`fir`

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

Usage

```
_ : fir(bv) : _
```

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));
```

`conv` and `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,...))
```

`tf1`, `tf2` and `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

`notchw`

Simple notch filter based on a biquad (`tf2`).

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

`tf21`, `tf22`, `tf22t` and `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.

`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).

`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

---

`iir_lat2`

Two-multiply latice IIR filter or arbitrary order.

Usage

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

Where:

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

`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 reflexion coefficients (-1 1)

`iir_kl`

Kelly-Lochbaum ladder IIR filter or arbitrary order.

Usage

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

Where:

- bv: zeros as a bank of parallel signals

- av: poles as a bank of parallel signals

---

allpassnklt

Kelly-Lochbaum ladder allpass.

Usage:

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

Where:

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

---

iir_lat1

One-multiply latice IIR filter or arbitrary order.

Usage

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

Where:

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

---

allpassn1mt

One-multiply lattice allpass with tap lines.

Usage

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

Where:

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

`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

---

`allpassnnlt`

Normalized ladder allpass filter of arbitrary order.

Usage:

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

Where:

- `n`: the order of the filter
- `sv`: the reflexion 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

`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

`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

---

`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

---

`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

`allpassn`

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

Usage:

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

Where:

- `n`: the order of the filter
- `sv`: the reflexion 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.

`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 reflexion coefficients (-PI PI)

`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 reflexion coefficients (-1 1)

`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 reflexion coefficients (-1 1)

Digital Filter Sections Specified as Analog Filter Sections

`tf2s` and `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

---

`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) : _
```

`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

`1`

H(s) = ––- 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

---

`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 prototpe 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) : _
```

`tf1sb`

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

Usage

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

Simple Resonator Filters

`resonlp`, `resonhp` and `resonbp`

Simple resonant lowpass, highpass and bandpass filters based on `tf2s`.

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

`lowpass` and `highpass`

Nth-order Butterworth lowpass or highpass filters.

Usage

```
_ : lowpass(N,fc) : _
_ : 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');"`)

`lowpass0_highpass1`

TODO

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).

`lowpass_plus|minus_highpass`

TODO

Elliptic (Cauer) Lowpass Filters

Elliptic (Cauer) Lowpass Filters

References

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

`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
```

`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

`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

`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

`bandpass` and `bandstop`

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. A notch-like "bandstop" filter is similarly made from `highpass(Nh)`.

Usage

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

Where:

- `Nh`: HALF the desired bandpass/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

`bandpass6e`

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

`bandpass12e`

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

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

`low_shelf` and `lowshelf_other_freq`

First-order "low shelf" filter (gain boost|cut between dc and some frequency)

Usage

```
_ : lowshelf(N,L0,fx) : _
_ : lowshelf_other_freq(N,L0,fx) : _
```

Where: * `N`: filter order 1, 3, 5, ... (odd only). * `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`.

`high_shelf` and `highshelf_other_freq`

First-order "high shelf" filter (gain boost|cut above some frequency).

Usage

```
_ : highshelf(N,Lpi,fx) : _
_ : 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.

`peak_eq`

Second order "peaking equalizer" section (gain boost or cut near some frequency) Also called a "parametric equalizer" section.

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

`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

---

`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.)

`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
- `bw`: bandwidth of desired roll-off band
- `alpha`: slope of roll-off desired in nepers per neper (ln mag / ln radian freq)

Examples

See `spectral_tilt_demo`.

Reference

Link to appear here when write up is done

`levelfilter` and `levelfilterN`

Dynamic level lowpass filter.

Usage

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

`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,_); // 3d-order Butterworth
_ : mth_octave_filterbank5(M,ftop,N) : par(i,N,_); // 5th-roder Butterworth
mth_octave_filterbank_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)

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.

`filterbank`

Filter bank.

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)) : _,_,_
```

`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.

It should be used using the `ho` environment:

```
ho = library("ho.lib");
process = ho.functionCall;
```

Another option is to import `stdfaust.lib` which already contains the `ho` environment:

```
import("stdfaust.lib");
process = ho.functionCall;
```

`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

`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.

`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`.

`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

`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

`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

`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

`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

`rotate`

Rotates the sound field.

Usage

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

Where:

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

---

math.lib

Mathematic library for Faust. Some functions are implemenented as Faust foreign functions of `math.h` functions that are not part of Faust's primitives. Defines also various constants and several utilities.

It should be used using the `fi` environment:

```
ma = library("math.lib");
process = ma.functionCall;
```

Another option is to import `stdfaust.lib` which already contains the `ma` environment:

```
import("stdfaust.lib");
process = ma.functionCall;
```

Functions Reference

`SR`

Current sampling rate (between 1Hz and 192000Hz). Constant during program execution.

Usage

```
SR : _
```

`BS`

Current block-size. Can change during the execution.

Usage

```
BS : _
```

`PI`

Constant PI in double precisio.n

Usage

```
PI : _
```

`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

---

`neg`

Invert the sign (-x) of a signal.

Usage

```
_ : neg : _
```

`sub(x,y)`

Subtract `x` and `y`.

`inv`

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

Usage

```
_ : inv : _
```

`cbrt`

Computes the cube root of of the input signal.

Usage

```
_ : cbrt : _
```

`hypot`

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

Usage

`_,_ : hypot : _`

`ldexp`

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

Usage

`_,_ : ldexp : _`

`scalb`

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

Usage

`_,_ : scalb : _`

`log1p`

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

Usage

`_ : log1p : _`

`logb`

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

Usage

`_ : logb : _`

`ilogb`

Return exponent of the input signal as an integer number.

Usage

`_ : ilogb : _`

`log2`

Returns the base 2 logarithm of x.

Usage

`_ : log2 : _`

`expm1`

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

Usage

`_ : expm1 : _`

`acosh`

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

Usage

`_ : acosh : _`

`asinh`

Computes the inverse hyperbolic sine of the input signal.

Usage

```
_ : asinh : _
```

---

atanh

Computes the inverse hyperbolic tangent of the input signal.

Usage

```
_ : atanh : _
```

---

sinh

Computes the hyperbolic sine of the input signal.

Usage

```
_ : sinh : _
```

---

cosh

Computes the hyperbolic cosine of the input signal.

Usage

```
_ : cosh : _
```

---

tanh

Computes the hyperbolic tangent of the input signal.

Usage

```
_ : tanh : _
```

`erf`

Computes the error function of the input signal.

Usage

`_ : erf : _`

`erfc`

Computes the complementary error function of the input signal.

Usage

`_ : erfc : _`

`gamma`

Computes the gamma function of the input signal.

Usage

`_ : gamma : _`

`lgamma`

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

Usage

`_ : lgamma : _`

`J0`

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

Usage

`_ : J0 : _`

`J1`

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

Usage

`_ : J1 : _`

`Jn`

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

Usage

`_,_ : Jn : _`

`Y0`

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

Usage

`_ : Y0 : _`

`Y1`

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

Usage

```
_ : Y0 : _
```

`Yn`

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

Usage

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

`fabs`, `fmax`, `fmin`

Just for compatibility…

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

`np2`

Gives the next power of 2 of x.

Usage

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

Where:

- `n`: an integer

`frac`

Gives the fractional part of n.

Usage

```
frac(n) : _
```

Where:

- `n`: a decimal number

`isnan`

Return non-zero if and only if x is a NaN.

Usage

```
isnan(x)
_ : isnan : _
```

Where:

- `x`: signal to analyse

`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

`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

`diffn`

Negated first-roder difference.

Usage

```
_ : diffn : _
```

---

misceffect.lib

This library contains a collection of audio effects.

It should be used using the `ef` environment:

```
ef = library("misceffect.lib");
process = ef.functionCall;
```

Another option is to import `stdfaust.lib` which already contains the `ef` environment:

```
import("stdfaust.lib");
process = ef.functionCall;
```

Dynamic

`cubicnl`

Cubic nonlinearity distortion.

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

---

`gate_mono` and `gate_stereo`

Mono and stereo signal gates.

Usage

```
_ : gate_mono(thresh,att,hold,rel) : _
```

or

```
 _,_ : 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

`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(130,5000);

Usage

```
speakerbp(f1,f2)
_ : speakerbp(130,5000) : _
```

`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

`stereo_width`

Stereo Width effect using the Blumlein Shuffler technique.

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

Time Based

`echo`

A simple echo effect.

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

`transpose`

A simple pitch shifter based on 2 delay lines.

Usage

```
_ : transpose(w, x, s) : _
```

Where:

- `w`: the window length (samples)
- `x`: crossfade duration duration (samples)
- `s`: shift (semitones)

Meshes

`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.

---

miscoscillator.lib

This library contains a collection of sound generators.

It should be used using the `os` environment:

```
os = library("miscoscillator.lib");
process = os.functionCall;
```

Another option is to import `stdfaust.lib` which already contains the `os` environment:

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

Wave-Table-Based Oscillators

`sinwaveform`

Sine waveform ready to use with a `rdtable`.

Usage

```
sinwaveform(tablesize) : _
```

Where:

- `tablesize`: the table size

`coswaveform`

Cosine waveform ready to use with a `rdtable`.

Usage

```
coswaveform(tablesize) : _
```

Where:

- `tablesize`: the table size

`phasor`

A simple phasor to be used with a `rdtable`.

Usage

```
phasor(tablesize,freq) : _
```

Where:

- `tablesize`: the table size
- `freq`: the frequency of the wave (Hz)

`oscsin`

Sine wave oscillator.

Usage

```
oscsin(freq) : _
```

Where:

* `freq`: the frequency of the wave (Hz)

`osc`

Default sine wave oscillator (same as `oscrs`).

Usage

```
osc(freq) : _
```

Where:

* `freq`: the frequency of the wave (Hz)

`oscos`

Cosine wave oscillator.

Usage

```
osccos(freq) : _
```

Where:

* `freq`: the frequency of the wave (Hz)

`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

`osci`

Interpolated phase sine wave oscillator.

Usage

```
osci(freq) : _
```

Where:

- `freq`: the frequency of the wave (Hz)

Virtual Analog Oscillators

Mostly elements from old "oscillator.lib".

Virtual analog oscillators and filter-based oscillators.

Low-frequency oscillators have prefix `lf_` (no aliasing suppression, signal-means not necessarily zero)

Low Frequency Impulse and Pulse Trains, Square and Triangle Waves

Low Frequency Impulse and Pulse Trains, Square and Triangle Waves

`lf_imptrain`, `lf_pulsetrainpos`, `lf_squarewavepos`, `lf_squarewave`, `lf_trianglepos`

Usage

```
lf_imptrain(freq) : _
lf_pulsetrainpos(freq,duty) : _
lf_squarewavepos(freq) : _
lf_squarewave(freq) : _
lf_trianglepos(freq) : _
```

Where:

- `freq`: frequency in Hz
- `duty`: duty cycle between 0 and 1

Notes

- Suffix 'pos' means the function is nonnegative, otherwise ~ zero mean
- All impulse and pulse trains jump to 1 at time 0

Low Frequency Sawtooths

Low Frequency Sawtooths

`lf_rawsaw`, `lf_sawpos`, `lf_sawpos_phase`

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).

Usage

```
lf_rawsaw(periodsamps) : _
lf_sawpos(freq) : _
lf_sawpos_phase(phase,freq) : _
saw1(freq) : _
```

Bandlimited Sawtooth

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)

Reference

- 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

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

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

`oscb`

Sinusoidal oscillator based on the biquad

`oscr`,`oscrs` and `oscs`

Sinusoidal oscillator based on 2D vector rotation, = undamped "coupled-form" resonator = lossless 2nd-order normalized ladder filter.

`oscr` = `oscrs`, `oscrs` generates a sine wave and `oscs` a cosine.

Reference:

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

---

`oscs`

Sinusoidal oscillator based on the state variable filter = undamped "modified-coupled-form" resonator = "magic circle" algorithm used in graphics

---

`oscw`, `oscwq`, `oscwc` and `oscws`

Sinusoidal oscillator based on the waveguide resonator wgr.

`oscwc` - unit-amplitude cosine oscillator

`oscws` - unit-amplitude sine oscillator

`oscq` - unit-amplitude cosine and sine (quadrature) oscillator

`oscw` - default = `oscwc` for maximum speed

Reference

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

---

noise.lib

A library of noise generators.

It should be used using the `no` environment:

```
no = library("noise.lib");
process = no.functionCall;
```

Another option is to import `stdfaust.lib` which already contains the `no` environment:

```
import("stdfaust.lib");
process = no.functionCall;
```

Functions Reference

`noise`

White noise generator (outputs random number between -1 and 1).

Usage

```
noise : _
```

`multirandom`

Generates multiple decorrelated random numbers in parallel.

Usage

```
multirandom(n) : _
```

Where:

- `n`: the number of decorrelated random numbers in parallel

`multinoise`

Generates multiple decorrelated noises in parallel.

Usage

```
multinoise(n) : _
```

Where:

- `n`: the number of decorrelated random numbers in parallel

`noises`

TODO.

`pink_noise`

Pink noise (1/f noise) generator (third-order approximation)

Usage

```
pink_noise : _;
```

Reference:

https://ccrma.stanford.edu/~jos/sasp/Example_Synthesis_1_F_Noise.html

---

`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

`lfnoise`, `lfnoise0` and `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 filter.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
```

---

phafla.lib

A library of compressor effects.

It should be used using the `pf` environment:

```
pf = library("phafla.lib");
process = pf.functionCall;
```

Another option is to import `stdfaust.lib` which already contains the `pf` environment:

```
import("stdfaust.lib");
process = pf.functionCall;
```

Functions Reference

`flanger_mono` and `flanger_stereo`

Flanging effect.

Usage:

```
_ : flanger_mono(dmax,curdel,depth,fb,invert) : _;
_,_ : flanger_stereo(dmax,curdel1,curdel2,depth,fb,invert) : _,_;
_,_ : flanger_demo : _,_;
```

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

`phaser2_mono` and `phaser2_stereo`

Phasing effect.

Phaser

```
_ : phaser2_mono(Notches,phase,width,frqmin,fratio,frqmax,speed,depth,fb,invert) : _;
_,_ : phaser2_stereo(") : _,_;
_,_ : phaser2_demo : _,_;
```

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/

---

reverb.lib

A library of reverb effects.

It should be used using the `re` environment:

```
re = library("reverb.lib");
process = re.functionCall;
```

Another option is to import `stdfaust.lib` which already contains the `re` environment:

```
import("stdfaust.lib");
process = re.functionCall;
```

Functions Reference

`jcrev` and `satrev`

These artificial reverberators 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.

`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

```
_ : jcrev : _,_,_,_
_ : satrev : _,_
```

`mono_freeverb` and `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

```
_ : mono_freeverb(fb1, fb2, damp, spread) : _;
_,_ : 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)

`fdnrev0`

Pure Feedback Delay Network Reverberator (generalized for easy scaling).

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

`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

`zita_rev1_stereo`

Extend `zita_rev_fdn` to include `zita_rev1` input/output mapping in stereo mode.

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)

`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)

route.lib

A library of basic elements to handle signal routing in Faust.

It should be used using the `si` environment:

```
ro = library("route.lib");
process = ro.functionCall;
```

Another option is to import `stdfaust.lib` which already contains the `si` environment:

```
import("stdfaust.lib");
process = ro.functionCall;
```

Functions Reference

`cross`

Cross n signals: `(x1,x2,..,xn) -> (xn,..,x2,x1)`.

Usage

```
cross(n)
_,_,_ : cross(3) : _,_,_
```

Where:

- `n`: number of signals (int, must be known at compile time)

Note

Special case: `cross2`:

```
cross2 = _,cross(2),_;
```

`crossnn`

Cross two `bus(n)`s.

Usage

```
_,_,... : crossmm(n) : _,_,...
```

Where:

- `n`: the number of signals in the `bus`

`crossn1`

Cross bus(n) and bus(1).

Usage

```
_,_,... : crossn1(n) : _,_,...
```

Where:

- `n`: the number of signals in the first `bus`

`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)

`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)

`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.

`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

---

signal.lib

A library of basic elements to handle signals in Faust.

It should be used using the `si` environment:

```
si = library("signal.lib");
process = si.functionCall;
```

Another option is to import `stdfaust.lib` which already contains the `si` environment:

```
import("stdfaust.lib");
process = si.functionCall;
```

Functions Reference

`bus`

n parallel cables

Usage

```
bus(n)
bus(4) : _,_,_,_
```

Where:

- `n`: is an integer known at compile time that indicates the number of parallel cables.

`block`

Block - terminate n signals.

Usage

```
_,_,... : block(n) : _,...
```

Where:

- `n`: the number of signals to be blocked

`interpolate`

Linear interpolation between two signals.

Usage

`_,_ : interpolate(i) : _`

Where:

- `i`: interpolation control between 0 and 1 (0: first input; 1: second input)

`smooth`

Exponential smoothing by a unity-dc-gain one-pole lowpass.

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

---

`smoo`

Smoothing function based on `smooth` ideal to smooth UI signals (sliders, etc.) down.

Usage

```
hslider(...) : smoo;
```

`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

`bsmooth`

Block smooth linear interpolation during a block of samples.

Usage

```
hslider(...) : bsmooth : _
```

`lag_ud`

Lag filter with separate times for up and down.

Usage

```
_ : lag_ud(up, dn, signal) : _;
```

`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)

---

spat.lib

This library contains a collection of tools for sound spatialization.

It should be used using the `sp` environment:

```
sp = library("spat.lib");
process = sp.functionCall;
```

Another option is to import `stdfaust.lib` which already contains the `sp` environment:

```
import("stdfaust.lib");
process = sp.functionCall;
```

`panner`

A simple linear gain panner.

Usage

```
_ : panner(g) : _,_
```

Where:

- `g`: the panning (0-1)

`spat`

GMEM SPAT: n-outputs spatializer

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)

`stereoize`

Transform an arbitrary processor `p` into a stereo processor with 2 inputs and 2 outputs.

Usage

```
_,_ : stereoize(p) : _,_
```

Where:

- `p`: the arbitrary processor

---

synth.lib

This library contains a collection of envelope generators.

It should be used using the `sy` environment:

```
sy = library("synth.lib");
process = sy.functionCall;
```

Another option is to import `stdfaust.lib` which already contains the `sy` environment:

```
import("stdfaust.lib");
process = sy.functionCall;
```

`popFilterPerc`

A simple percussion instrument based on a "poped" resonant bandpass filter.

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)

`dubDub`

A simple synth based on a sawtooth wave filtered by a resonant lowpass.

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)

`sawTrombone`

A simple trombone based on a lowpassed sawtooth wave.

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)

`combString`

Simplest string physical model ever based on a comb filter.

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)

`additiveDrum`

A simple drum using additive synthesis.

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)

`additiveDrum`

An FM synthesizer with an arbitrary number of modulators connected as a sequence.

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)

---

vaeffect.lib

A library of virtual analog filter effects.

It should be used using the `ve` environment:

```
ve = library("vaeffect.lib");
process = ve.functionCall;
```

Another option is to import `stdfaust.lib` which already contains the `ve` environment:

```
import("stdfaust.lib");
process = ve.functionCall;
```

Functions Reference

`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:

- `fr`: corner-resonance frequency in Hz (less than SR/6.3 or so)

- `res`: Normalized amount of corner-resonance between 0 and 1 (0 is no resonance, 1 is maximum)

References

- https://ccrma.stanford.edu/~stilti/papers/moogvcf.pdf
- https://ccrma.stanford.edu/~jos/pasp/vegf.html

`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:

- `fr`: corner-resonance frequency in Hz
- `res`: Normalized amount of corner-resonance between 0 and 1 (0 is min resonance, 1 is maximum)

`wah4`

Wah effect, 4th order.

Usage

```
_ : wah4(fr) : _
```

Where:

- `fr`: resonance frequency in Hz

Reference

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

`autowah`

Auto-wah effect.

Usage

```
_ : autowah(level) : _;
```

Where:

- `level`: amount of effect desired (0 to 1).

`crybaby`

Digitized CryBaby wah pedal.

Usage

```
_ : crybaby(wah) : _
```

Where:

- `wah`: "pedal angle" from 0 to 1

Reference

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

`vocoder`

A very simple vocoder where the spectrum of the modulation signal is analyzed using a filter bank.

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
