Contents

Faust Libraries

The Faust libraries implement hundreds of DSP functions for audio processing and synthesis. They are organized by types in a set of `.lib` files (e.g., `envelopes.lib`, `filters.lib`, etc.). Librairies use semantic versioning, so may evolve in a manner where newer versions break compatibility with older ones. The recommended way to solve this issue is to keep *self-contained versions of the DSP code* (that is the DSP program with all needed libraries) as explained in Goals of the Mathdoc.

This website serves as the main documentation of the Faust libraries. The main Faust website can be found at the following URL:

https://faust.grame.fr

Using the Faust Libraries

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

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

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

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

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

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

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

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

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

Alternatively, environments can be created by hand:

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

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

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

Organization of This Documentation

The `Overview` tab in the upper menu provides additional information about the general organization of the libraries, licensing/copyright, and guidelines on how to contribute to the Faust libraries.

The `Libraries` tab contain the actual documentation of the Faust libraries.

General Organization

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

- `aanl.lib`
- `analyzers.lib`
- `basics.lib`
- `compressors.lib`
- `delays.lib`
- `demos.lib`
- `dx7.lib`
- `envelopes.lib`
- `fds.lib`
- `filters.lib`
- `hoa.lib`
- `interpolators.lib`
- `linearalgebra.lib`
- `maths.lib`
- `mi.lib`
- `misceffects.lib`
- `oscillators.lib`
- `noises.lib`
- `phaflangers.lib`
- `physmodels.lib`
- `reducemaps.lib`
- `reverbs.lib`
- `routes.lib`
- `signals.lib`
- `soundfiles.lib`
- `spats.lib`
- `synths.lib`

- `tonestacks.lib` (not documented but example in `/examples/misc`)
- `tubes.lib` (not documented but example in `/examples/misc`)
- `vaeffects.lib`
- `version.lib`
- `wdmodels.lib`
- `webaudio.lib`

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

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

The `all.lib` compatibility library imports all libraries in a same namespace, to be located in a single folder. The `doc.lib` describes the actual locatisation of all libraries, including possible subfolders, and is used to generate the documentation of the Faust standard libraries.

Versioning

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

Examples

The Faust distribution `/examples` directory contains a lot of DSP examples. They are organized by types in different folders. The `/examples/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.

Standard Functions

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

Analysis Tools

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

Basic Elements

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

Conversion

Function Type	Function Name	Description
dB to Linear	`ba.db2linear`	Converts dB to linear values
Linear to dB	`ba.linear2db`	Converts linear values to dB

Function Type	Function Name	Description
MIDI Key to Hz	`ba.midikey2hz`	Converts a MIDI key number into a frequency
Hz to MIDI Key	`ba.hz2midikey`	Converts a frequency into MIDI key number
Pole to T60	`ba.pole2tau`	Converts a pole into a time constant (t60)
T60 to Pole	`ba.tau2pole`	Converts a time constant (t60) into a pole
Samples to Seconds	`ba.samp2sec`	Converts samples to seconds
Seconds to Samples	`ba.sec2samp`	Converts seconds to samples
Semitones to Frequency ratio	`ba.semi2ratio`	Converts semitones in a frequency multiplicative ratio
Frequency ratio to semintones	`ba.ratio2semi`	Converts a frequency multiplicative ratio in semitones

Effects

Function Type	Function Name	Description
Auto Wah	`ve.autowah`	Auto-Wah effect
Compressor	`co.compressor_mono`	Dynamic range compressor
Distortion	`ef.cubicnl`	Cubic nonlinearity distortion
Crybaby	`ve.crybaby`	Crybaby wah pedal
Echo	`ef.echo`	Simple echo
Flanger	`pf.flanger_stereo`	Flanging effect
Gate	`ef.gate_mono`	Mono signal gate
Limiter	`co.limiter_1176_R4_mono`	Limiter
Phaser	`pf.phaser2_stereo`	Phaser effect
Reverb (FDN)	`re.fdnrev0`	Feedback delay network reverberator
Reverb (Freeverb)	`re.mono_freeverb`	Most "famous" Schroeder reverberator
Reverb (Simple)	`re.jcrev`	Simple Schroeder reverberator
Reverb (Zita)	`re.zita_rev1_stereo`	High quality FDN reverberator

Function Type	Function Name	Description
Panner	`sp.panner`	Linear stereo panner
Pitch Shift	`ef.transpose`	Simple pitch shifter
Panner	`sp.spat`	N outputs spatializer
Speaker Simulator	`ef.speakerbp`	Simple speaker simulator
Stereo Width	`ef.stereo_width`	Stereo width effect
Vocoder	`ve.vocoder`	Simple vocoder
Wah	`ve.wah4`	Wah effect

Envelope Generators

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

Filters

Function Type	Function Name	Description
Bandpass (Butterworth)	`fi.bandpass`	Generic butterworth bandpass
Bandpass (Resonant)	`fi.resonbp`	Virtual analog resonant bandpass
Bandstop (Butterworth)	`fi.bandstop`	Generic butterworth bandstop
Biquad	`fi.tf2`	"Standard" biquad filter
Comb (Allpass)	`fi.allpass_fcomb`	Schroeder allpass comb filter
Comb (Feedback)	`fi.fb_fcomb`	Feedback comb filter
Comb (Feedforward)	`fi.ff_fcomb`	Feed-forward comb filter.
DC Blocker	`fi.dcblocker`	Default dc blocker
Filterbank	`fi.filterbank`	Generic filter bank
FIR (Arbitrary Order)	`fi.fir`	Nth-order FIR filter
High Shelf	`fi.high_shelf`	High shelf

Function Type	Function Name	Description
Highpass (Butterworth)	`fi.highpass`	Nth-order Butterworth highpass
Highpass (Resonant)	`fi.resonhp`	Virtual analog resonant highpass
IIR (Arbitrary Order)	`fi.iir`	Nth-order IIR filter
Level Filter	`fi.levelfilter`	Dynamic level lowpass
Low Shelf	`fi.low_shelf`	Low shelf
Lowpass (Butterworth)	`fi.lowpass`	Nth-order Butterworth lowpass
Lowpass (Resonant)	`fi.resonlp`	Virtual analog resonant lowpass
Notch Filter	`fi.notchw`	Simple notch filter
Peak Equalizer	`fi.peak_eq`	Peaking equalizer section

Oscillators/Sound Generators

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

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

Synths

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

Contributing

In general, libraries are organised in a *stacked manner*: the base ones define functions or constants without any dependancies, and additional ones are gradually built on top of simpler ones, layer by layer. **Dependency loops must be avoided as much as possible**. The *resources* folder contains tools to build and visualise the libraries dependencies graphs.

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 observe the following conventions:

New Functions

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

````
//-----------------functionName--------------------
// Description
//
// #### Usage
//
// ```
````

~~~~
// Usage example
// ```
//
// Where:
//
// * argument1: argument 1 description
// * argument2: argument 2 description
//
// #### Example
//
// ```
// Additional example
// ```
//
// #### Test
// ```
// functionName_test = some_dsp_code;
// ```
//
// #### References
//
// * <https://some_url1>
// * <https://some_url2>
//--------------------------------------------------
~~~~

- 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 Library Import).
- Try to reuse existing functions as much as possible.
- The `Usage` line must show the *input/output shape* (the number of inputs and outputs) of the function, like `gen: _` for a mono generator, `_ : filter : _` for a mono effect, etc.
- The `Example` line can be used to provide additional examples.
- The `Test` line can be used to add a DSP program to test the function. The test name must be `functionName_test`. The actual code can be extracted and independantly tested using the `-pn` compiler option (to specify the name of the dsp entry-point instead of process). The test code must import all the needed libraries, like `an = library("analyzers.lib");` if a function from analyzers.lib is used in the test code.
- The `References` line can be used to add links to references
- Some functions use parameters that are constant numerical expressions. The convention is to label them in *capital letters* and document them preferably to be *constant numerical expressions* (or *known at compile time* in existing libraries).
- Functions with several parameters should better be written by putting the

more constant parameters (like control, setup…) at the beginning of the parameter list, and *audio signals to be processed* at the end. This allows to do partial-application. So prefer the following `clip(low, high, x) = min(max(x, low), high);` form where `clip(-1, 1)` partially applied version can be used later on in different contexts, better than `clip(x, low, high) = min(max(x, low), high);` version.

Layering UI-ready variants

Many functions benefit from two public faces so the same DSP can serve both low-level reuse and ready-to-tweak usage:

- *Core function*: exposes all parameters, no UI or side effects; best for reuse, composition, and testing.
- *UI wrapper*: fixes sensible defaults and exposes only runtime-tuned parameters as UI controls; leaves signals that must be provided externally as arguments.

Use the UI-free core for correctness and performance work; build the UI variant when you need something directly tweakable in examples or end-user contexts.

A generic core/UI pair could be:

```
// Core: parameters explicit, no UI
coreEffect(paramA, paramB, mix) =
  fooProcessing(mix, wet)
with {
  wet = *(paramA) : barProcessing(paramB); // barProcessing is your DSP
};

// UI wrapper: binds smoothed controls to the core
coreEffect_ui =
  coreEffect(paramA_ui, paramB_ui, mix_ui)
with {
  paramA_ui = hslider("Param A", 1.0, 0.0, 2.0, 0.01) : si.smoo;
  paramB_ui = hslider("Param B", 0.5, 0.0, 1.0, 0.01) : si.smoo;
  mix_ui    = hslider("Mix", 1.0, 0.0, 1.0, 0.01) : si.smoo;
};

process = coreEffect_ui;
```

This keeps the core reusable (no UI dependencies) and the wrapper ready for immediate tweaking; `process` points to the UI layer for quick testing.

Instrument-specific three-layer pattern Instrument models often add a third, ready-to-play layer. The clarinet model is a reference:

- `pm.clarinetModel(tubeLength, pressure, reedStiffness, bellOpening)`: core DSP with every parameter explicit and no UI.

- `pm.clarinetModel_ui(pressure)`: wraps the core and adds UI sliders for tube length, reed stiffness, bell opening, and output gain; keeps `pressure` as an argument.
- `pm.clarinet_ui_MIDI`: builds a playable instrument by pairing the core with a blower/envelope plus MIDI-mapped UI (pitch bend, sustain, vibrato, gain, etc.).

When adding similar models, start with the UI-free core, add a minimal UI wrapper, then optionally provide a controller-specific wrapper (MIDI or otherwise). Keep the core independent so it remains reusable.

Variables and identifiers scoping

To avoid name clashes between libraries, keep identifiers as local as possible. Prefer defining intermediate constants and helpers inside `with { ... }` blocks or `environment { ... }` sections, and only expose the intended public entry points. This minimizes collisions when several libraries are imported together and keeps global namespace usage limited to documented, public-facing functions.

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`.
- Any new library has to use a prefix declared in the header section with the following kind of syntax: `Its official prefix is 'qu'` (look at an existing library to follow the exact syntax).
- Be sure to add the appropriate kind of `ma = library("maths.lib");` import library line, for each external library function used in the new library (for instance `ma.foo` that would be used somewhre in the code).
- 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
//==============================================
```

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).

Function Naming

[WIP]

JOS proposal: using terms used in the field of digital signal processing, as follows:

- `impulse`: …,0,1,0,…
- `pulse`: …,0,1,1,0,… or longer
- `impulse_train`
- `pulse_train`
- `gate` = pulse controlled externally (e.g., by NoteOn,NoteOff)
- `trigger` = impulse controlled externally (gate - gate' > 0) == gate rising edge

[/WIP]

Variable Argument List

Strictly speaking, there are no lists in Faust. But list operations can be simulated (in part) using the parallel binary composition operation `,` and pattern matching.

Thus functions expecting a variable number of arguments can use this mechanism, like a `foo` function that would be used this way: `foo((a,b,c,d))`. See fi.iir and fi.fir examples.

Documentation

- All the functions that we want to be "public" are documented.
- We used the `faust2md` "standards" for each library: `//###` for main title (library name - equivalent to `#` in markdown), `//===` for section declarations (equivalent to `##` in markdown) and `//---` for function declarations (equivalent to `####` in markdown - see `basics.lib` for an example).
- Sections in function documentation should be declared as `####` markdown title.
- Each function documentation provides a "Usage" section (see `basics.lib`).
- The full documentation can be generated using the doc/Makefile script. Use `make help` to see all possible commands. If you plan to create a pull-request, *do not commit the full generated code* but only the modified .lib files.
- Each function can have `declare author "name";`, `declare copyright "XXX";` and `declare licence "YYY";` declarations.
- Each library has a `declare version "xx.yy.zz";` semantic version number to be raised each time a modification is done. The global `version` number in `version.lib` also has to be adapted according to the change.

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:

```
aa = library("aanl.lib");
sf = library("all.lib");
an = library("analyzers.lib");
ba = library("basics.lib");
co = library("compressors.lib");
de = library("delays.lib");
dm = library("demos.lib");
dx = library("dx7.lib");
en = library("envelopes.lib");
fd = library("fds.lib");
fi = library("filters.lib");
ho = library("hoa.lib");
it = library("interpolators.lib");
la = library("linearalgebra.lib");
ma = library("maths.lib");
mi = library("mi.lib");
```

```
ef = library("misceffects.lib");
os = library("oscillators.lib");
no = library("noises.lib");
pf = library("phaflangers.lib");
pm = library("physmodels.lib");
qu = library("quantizers.lib");
rm = library("reducemaps.lib");
re = library("reverbs.lib");
ro = library("routes.lib");
si = library("signals.lib");
so = library("soundfiles.lib");
sp = library("spats.lib");
sy = library("synths.lib");
ve = library("vaeffects.lib");
vl = library("version.lib");
wa = library("webaudio.lib");
wd = library("wdmodels.lib");
```

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

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

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

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

"Demo" Functions

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

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

"Standard" Functions

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

Testing the library

Before preparing a pull-request, the new library must be carefully tested:

- all functions defined in the library must be tested by preparing a DSP test program, to be added using the `#### Test` syntax
- the compatibility library `all.lib` imports all libraries in a same namespace, so check functions names collisions using the following test program:

```
import("all.lib");
process = _;
```

Library test and deployment

For GRAME maintainers:

- global tests can be done using the `make reference` and `make check` at rool level
- regenerate the PDF documentation using `make pdf` target in the `doc` folder
- update the library submodule in faust, recompile and deploy WebAssembly libfaust in fausteditor, faustplayground and faustide
- update the library submodule in faustlive
- update the library list in this fausteditor page as well as the snippets (using the `faust2atomsnippets` tool).
- update the library list in this faustide page and LIBRARIES folder.
- update the library list in the faustgen~ code
- update the Faust Syntax Highlighting Files
- make an update PR for vscode-faust project

The Faust Project

The Faust Project has started in 2002. It is actively developed by the GRAME-CNCM Research Department.

Many persons are contributing to the Faust project, by providing code for the compiler, architecture files, libraries, examples, documentation, scripts, bug reports, ideas, etc. We would like in particular to thank:

Fons Adriaensen, Karim Barkati, Jérôme Barthélemy, Tim Blechmann, Tiziano Bole, Alain Bonardi, Thomas Charbonnel, Raffaele Ciavarella, Julien Colafrancesco, Damien Cramet, Sarah Denoux, Étienne Gaudrin, Olivier Guillerminet, Pierre Guillot, Albert Gräf, Pierre Jouvelot, Stefan Kersten, Victor Lazzarini, Matthieu Leberre, Mathieu Leroi, Fernando Lopez-Lezcano, Kjetil Matheussen, Hermann Meyer, Rémy Muller, Raphael Panis, Eliott Paris, Reza Payami, Laurent Pottier, Sampo Savolainen, Nicolas Scaringella, Anne Sedes, Priyanka Shekar, Stephen Sinclair, Travis Skare,

Julius Smith, Mike Solomon, Michael Wilson, Bart Brouns, Dirk Roosenburg, Riccardo Russo.

as well as our colleagues at GRAME:

- Dominique Fober
- Christophe Lebreton
- Stéphane Letz
- Romain Michon
- Yann Orlarey

We would like also to thank for their financial support:

- the French Ministry of Culture,
- the Auvergne-Rhône-Alpes Region,
- the City of Lyon,
- the French National Research Agency (ANR).

aanl.lib

A library for antialiased nonlinearities. Its official prefix is `aa`.

This library provides aliasing-suppressed nonlinearities through first-order and second-order approximations of continuous-time signals, functions, and convolution based on antiderivatives. This technique is particularly effective if combined with low-factor oversampling, for example, operating at 96 kHz or 192 kHz sample-rate.

The library contains trigonometric functions as well as other nonlinear functions such as bounded and unbounded saturators.

Due to their limited domains or ranges, some of these functions may not suitable for audio nonlinear processing or waveshaping, although they have been included for completeness. Some other functions, for example, tan() and tanh(), are only available with first-order antialiasing due to the complexity of the antiderivative of the x * f(x) term, particularly because of the necessity of the dilogarithm function, which requires special implementation.

Future improvements to this library may include an adaptive mechanism to set the ill-conditioned cases threshold to improve performance in varying cases.

Note that the antialiasing functions introduce a delay in the path, respectively half and one-sample delay for first and second-order functions.

Also note that due to division by differences, it is vital to use double-precision or more to reduce errors.

The environment identifier for this library is `aa`. After importing the standard libraries in Faust, the functions below can be called as `aa.function_name`.

References

- https://github.com/grame-cncm/faustlibraries/blob/master/aanl.lib
- Reducing the Aliasing in Nonlinear Waveshaping Using Continuous-time Convolution, Julian Parker, Vadim Zavalishin, Efflam Le Bivic, DAFX, 2016
- http://dafx16.vutbr.cz/dafxpapers/20-DAFx-16__paper__41-PN.pdf

Auxiliary Functions

`(aa.)clip`

Clipping function.

`(aa.)Rsqrt`

Real-valued sqrt().

`(aa.)Rlog`

Real-valued log().

`(aa.)Rtan`

Real-valued tan().

`(aa.)Racos`

Real-valued acos().

`(aa.)Rasin`

Real-valued asin().

(aa.)Racosh

Real-valued acosh()

(aa.)Rcosh

Real-valued cosh().

(aa.)Rsinh

Real-valued sinh().

(aa.)Ratanh

Real-valued atanh().

(aa.)ADAA1

Generalised first-order Antiderivative Anti-Aliasing (ADAA) function.

Implements a first-order ADAA approximation to reduce aliasing in nonlinear audio processing.

Usage

`_ : ADAA1(EPS, f, F1) : _`

Where:

- `EPS`: a threshold for switching between safe and ill-conditioned paths
- `f`: a function that we want to process with ADAA
- `F1`: f's first antiderivative #### Test

```
aa = library("aanl.lib");
ba = library("basics.lib");
ma = library("maths.lib");
os = library("oscillators.lib");
ADAA1_test = aa.ADAA1(0.001, f, F1, os.osc(110))
    with {
        f(x) = aa.clip(-1.0, 1.0, x);
        F1(x) = ba.if((x <= 1.0) & (x >= -1.0), 0.5 * x^2, x * ma.signum(x) - 0.5);
    };
```

(aa.)ADAA2

Generalised second-order Antiderivative Anti-Aliasing (ADAA) function.

Implements a second-order ADAA approximation for even better aliasing reduction at the cost of additional computation. #### Usage

```
_ : ADAA2(EPS, f, F1, F2) : _
```

Where:

- `EPS`: a threshold for switching between safe and ill-conditioned paths
- `f`: a function that we want to process with ADAA
- `F1`: f's first antiderivative
- `F2`: f's second antiderivative #### Test

```
aa = library("aanl.lib");
ba = library("basics.lib");
ma = library("maths.lib");
os = library("oscillators.lib");
ADAA2_test = aa.ADAA2(0.001, f, F1, F2, os.osc(110))
    with {
        f(x) = aa.clip(-1.0, 1.0, x);
        F1(x) = ba.if((x <= 1.0) & (x >= -1.0), 0.5 * x^2, x * ma.signum(x) - 0.5);
        F2(x) = ba.if((x <= 1.0) & (x >= -1.0), (1.0 / 3.0) * x^3, ((0.5 * x^2) - 1.0 / 6.0)
    };
```

Main functions

Saturators

These antialiased saturators perform best with high-amplitude input signals. If the input is only slightly saturated, hence producing negligible aliasing, the trivial saturator may result in a better overall output, as noise can be introduced by first and second ADAA at low amplitudes.

Once determining the lowest saturation level for which the antialiased functions perform adequately, it might be sensible to cross-fade between the trivial and the antialiased saturators according to the amplitude profile of the input signal.

(aa.)hardclip

First-order ADAA hard-clip.

The domain of this function is ; its theoretical range is [-1.0; 1.0].

Usage

```
_ : aa.hardclip : _
```

Test

```
aa = library("aanl.lib");
os = library("oscillators.lib");
hardclip_test = aa.hardclip(os.osc(110));
```

(aa.)hardclip2

Second-order ADAA hard-clip.

The domain of this function is ; its theoretical range is [-1.0; 1.0].

Usage

```
_ : aa.hardclip2 : _
```

Test

```
aa = library("aanl.lib");
os = library("oscillators.lib");
hardclip2_test = aa.hardclip2(os.osc(110));
```

(aa.)cubic1

First-order ADAA cubic saturator.

The domain of this function is ; its theoretical range is [-2.0/3.0; 2.0/3.0].

Usage

```
_ : aa.cubic1 : _
```

Test

```
aa = library("aanl.lib");
os = library("oscillators.lib");
cubic1_test = aa.cubic1(os.osc(110));
```

(aa.)parabolic

First-order ADAA parabolic saturator.

The domain of this function is ; its theoretical range is [-1.0; 1.0].

Usage

```
_ : aa.parabolic : _
```

Test

```
aa = library("aanl.lib");
os = library("oscillators.lib");
parabolic_test = aa.parabolic(os.osc(110));
```

`(aa.)parabolic2`

Second-order ADAA parabolic saturator.

The domain of this function is ; its theoretical range is [-1.0; 1.0].

Usage

```
_ : aa.parabolic2 : _
```

Test

```
aa = library("aanl.lib");
os = library("oscillators.lib");
parabolic2_test = aa.parabolic2(os.osc(110));
```

`(aa.)hyperbolic`

First-order ADAA hyperbolic saturator.

The domain of this function is ; its theoretical range is [-1.0; 1.0].

Usage

```
_ : aa.hyperbolic : _
```

Test

```
aa = library("aanl.lib");
os = library("oscillators.lib");
hyperbolic_test = aa.hyperbolic(os.osc(110));
```

(aa.)hyperbolic2

Second-order ADAA hyperbolic saturator.

The domain of this function is ; its theoretical range is [-1.0; 1.0].

Usage

```
_ : aa.hyperbolic2 : _
```

Test

```
aa = library("aanl.lib");
os = library("oscillators.lib");
hyperbolic2_test = aa.hyperbolic2(os.osc(110));
```

(aa.)sinarctan

First-order ADAA sin(atan()) saturator.

The domain of this function is ; its theoretical range is [-1.0; 1.0].

Usage

```
_ : aa.sinarctan : _
```

Test

```
aa = library("aanl.lib");
os = library("oscillators.lib");
sinarctan_test = aa.sinarctan(os.osc(110));
```

(aa.)sinarctan2

Second-order ADAA sin(atan()) saturator.

The domain of this function is ; its theoretical range is [-1.0; 1.0].

Usage

```
_ : aa.sinarctan2 : _
```

Test

```
aa = library("aanl.lib");
os = library("oscillators.lib");
sinarctan2_test = aa.sinarctan2(os.osc(110));
```

(aa.)softclipQuadratic1

First-order ADAA quadratic softclip.

The domain of this function is ; its theoretical range is [-1.0; 1.0].

Usage

```
_ : aa.softclipQuadratic1 : _
```

Test

```
aa = library("aanl.lib");
os = library("oscillators.lib");
softclipQuadratic1_test = aa.softclipQuadratic1(os.osc(110));
```

(aa.)softclipQuadratic2

Second-order ADAA quadratic softclip.

The domain of this function is ; its theoretical range is [-1.0; 1.0].

Usage

```
_ : aa.softclipQuadratic2 : _
```

Test

```
aa = library("aanl.lib");
os = library("oscillators.lib");
softclipQuadratic2_test = aa.softclipQuadratic2(os.osc(110));
```

(aa.)tanh1

First-order ADAA tanh() saturator.

The domain of this function is ; its theoretical range is [-1.0; 1.0].

Usage

```
_ : aa.tanh1 : _
```

Test

```
aa = library("aanl.lib");
os = library("oscillators.lib");
tanh1_test = aa.tanh1(os.osc(110));
```

(aa.)arctan

First-order ADAA atan().

The domain of this function is ; its theoretical range is [- /2.0; /2.0].

Usage

```
_ : aa.arctan : _
```

Test

```
aa = library("aanl.lib");
os = library("oscillators.lib");
arctan_test = aa.arctan(os.osc(110));
```

(aa.)arctan2

Second-order ADAA atan().

The domain of this function is ; its theoretical range is]- /2.0; /2.0[.

Usage

```
_ : aa.arctan2 : _
```

Test

```
aa = library("aanl.lib");
os = library("oscillators.lib");
arctan2_test = aa.arctan2(os.osc(110));
```

(aa.)asinh1

First-order ADAA asinh() saturator (unbounded).

The domain of this function is ; its theoretical range is .

Usage

```
_ : aa.asinh1 : _
```

Test

```
aa = library("aanl.lib");
os = library("oscillators.lib");
asinh1_test = aa.asinh1(os.osc(110));
```

(aa.)asinh2

Second-order ADAA asinh() saturator (unbounded).

The domain of this function is ; its theoretical range is .

Usage

```
_ : aa.asinh2 : _
```

Test

```
aa = library("aanl.lib");
os = library("oscillators.lib");
asinh2_test = aa.asinh2(os.osc(110));
```

Trigonometry

These functions are reliable if input signals are within their domains.

(aa.)cosine1

First-order ADAA cos().

The domain of this function is ; its theoretical range is [-1.0; 1.0].

Usage

```
_ : aa.cosine1 : _
```

Test

```
aa = library("aanl.lib");
os = library("oscillators.lib");
cosine1_test = aa.cosine1(os.osc(110));
```

(aa.)cosine2

Second-order ADAA cos().

The domain of this function is ; its theoretical range is [-1.0; 1.0].

Usage

```
_ : aa.cosine2 : _
```

Test

```
aa = library("aanl.lib");
os = library("oscillators.lib");
cosine2_test = aa.cosine2(os.osc(110));
```

(aa.)arccos

First-order ADAA acos().

The domain of this function is [-1.0; 1.0]; its theoretical range is [; 0.0].

Usage

```
_ : aa.arccos : _
```

Test

```
aa = library("aanl.lib");
os = library("oscillators.lib");
arccos_test = aa.arccos(os.osc(110));
```

(aa.)arccos2

Second-order ADAA acos().

The domain of this function is [-1.0; 1.0]; its theoretical range is [; 0.0].

Note that this function is not accurate for low-amplitude or low-frequency input signals. In that case, the first-order ADAA arccos() can be used.

Usage

```
_ : aa.arccos2 : _
```

Test

```
aa = library("aanl.lib");
os = library("oscillators.lib");
arccos2_test = aa.arccos2(os.osc(110));
```

(aa.)acosh1

First-order ADAA acosh().

The domain of this function is $>= 1.0$; its theoretical range is $>= 0.0$.

Usage

```
_ : aa.acosh1 : _
```

Test

```
aa = library("aanl.lib");
os = library("oscillators.lib");
acosh1_test = aa.acosh1(1.0 + abs(os.osc(110)));
```

(aa.)acosh2

Second-order ADAA acosh().

The domain of this function is $>= 1.0$; its theoretical range is $>= 0.0$.

Note that this function is not accurate for low-frequency input signals. In that case, the first-order ADAA acosh() can be used.

Usage

```
_ : aa.acosh2 : _
```

Test

```
aa = library("aanl.lib");
os = library("oscillators.lib");
acosh2_test = aa.acosh2(1.0 + abs(os.osc(110)));
```

(aa.)sine

First-order ADAA sin().

The domain of this function is ; its theoretical range is .

Usage

```
_ : aa.sine : _
```

Test

```
aa = library("aanl.lib");
os = library("oscillators.lib");
sine_test = aa.sine(os.osc(110));
```

`(aa.)sine2`

Second-order ADAA sin().

The domain of this function is ; its theoretical range is .

Usage

```
_ : aa.sine2 : _
```

Test

```
aa = library("aanl.lib");
os = library("oscillators.lib");
sine2_test = aa.sine2(os.osc(110));
```

`(aa.)arcsin`

First-order ADAA asin().

The domain of this function is [-1.0, 1.0]; its theoretical range is [- /2.0; /2.0].

Usage

```
_ : aa.arcsin : _
```

Test

```
aa = library("aanl.lib");
os = library("oscillators.lib");
arcsin_test = aa.arcsin(os.osc(110));
```

`(aa.)arcsin2`

Second-order ADAA asin().

The domain of this function is [-1.0, 1.0]; its theoretical range is [- /2.0; /2.0].

Note that this function is not accurate for low-frequency input signals. In that case, the first-order ADAA asin() can be used.

Usage

```
_ : aa.arcsin2 : _
```

Test

```
aa = library("aanl.lib");
os = library("oscillators.lib");
arcsin2_test = aa.arcsin2(os.osc(110));
```

`(aa.)tangent`

First-order ADAA tan().

The domain of this function is [- /2.0; /2.0]; its theoretical range is .

Usage

```
_ : aa.tangent : _
```

Test

```
aa = library("aanl.lib");
ma = library("maths.lib");
os = library("oscillators.lib");
tangent_test = aa.tangent(0.25 * ma.PI * os.osc(110));
```

`(aa.)atanh1`

First-order ADAA atanh().

The domain of this function is [-1.0; 1.0]; its theoretical range is .

Usage

```
_ : aa.atanh1 : _
```

Test

```
aa = library("aanl.lib");
os = library("oscillators.lib");
atanh1_test = aa.atanh1(0.8 * os.osc(110));
```

(aa.)atanh2

Second-order ADAA atanh().

The domain of this function is [-1.0; 1.0]; its theoretical range is .

Usage

```
_ : aa.atanh2 : _
```

Test

```
aa = library("aanl.lib");
os = library("oscillators.lib");
atanh2_test = aa.atanh2(0.8 * os.osc(110));
```

analyzers.lib

Analyzers library. Its official prefix is `an`.

This library provides reusable building blocks for audio signal *analysis* and metering. It includes functions and components for measuring levels, extracting features, and computing statistics useful in visualization, diagnostics, adaptive processing, and music information retrieval.

The Analyzers library is organized into 7 sections:

- Amplitude Tracking
- Adaptive Frequency Analysis
- Spectrum-Analyzers
- Mth-Octave Spectral Level
- Arbitrary-Crossover Filter-Banks and Spectrum Analyzers
- Fast Fourier Transform (fft) and its Inverse (ifft)
- Test signal generators

References

- https://github.com/grame-cncm/faustlibraries/blob/master/analyzers.lib

Amplitude Tracking

(an.)abs_envelope_rect

Absolute value average with moving-average algorithm.

Usage

```
_ : abs_envelope_rect(period) : _
```

Where:

- `period`: sets the averaging frame in seconds

Test

```
an = library("analyzers.lib");
os = library("oscillators.lib");
abs_envelope_rect_test = an.abs_envelope_rect(0.05, os.osc(220));
```

(an.)abs_envelope_tau

Absolute value average with one-pole lowpass and tau response (see filters.lib).

Usage

```
_ : abs_envelope_tau(period) : _
```

Where:

- `period`: (time to decay by 1/e) sets the averaging frame in secs

Test

```
an = library("analyzers.lib");
os = library("oscillators.lib");
abs_envelope_tau_test = an.abs_envelope_tau(0.05, os.osc(220));
```

(an.)abs_envelope_t60

Absolute value average with one-pole lowpass and t60 response (see filters.lib).

Usage

```
_ : abs_envelope_t60(period) : _
```

Where:

- `period`: (time to decay by 60 dB) sets the averaging frame in secs

Test

```
an = library("analyzers.lib");
os = library("oscillators.lib");
abs_envelope_t60_test = an.abs_envelope_t60(0.05, os.osc(220));
```

(an.)abs_envelope_t19

Absolute value average with one-pole lowpass and t19 response (see filters.lib).

Usage

```
_ : abs_envelope_t19(period) : _
```

Where:

- `period`: (time to decay by 1/e^2.2) sets the averaging frame in secs

Test

```
an = library("analyzers.lib");
os = library("oscillators.lib");
abs_envelope_t19_test = an.abs_envelope_t19(0.05, os.osc(220));
```

(an.)amp_follower

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

`amp_follower` is a standard Faust function.

Usage

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

Where:

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

Test

```
an = library("analyzers.lib");
os = library("oscillators.lib");
amp_follower_test = os.osc(220) : an.amp_follower(0.05);
```

References

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

(an.)amp_follower_ud

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

Usage

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

Where:

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

Test

```
an = library("analyzers.lib");
os = library("oscillators.lib");
amp_follower_ud_test = os.osc(220) : an.amp_follower_ud(0.002, 0.05);
```

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.

References

- "Digital Dynamic Range Compressor Design — A Tutorial and Analysis", by Dimitrios Giannoulis, Michael Massberg, and Joshua D. Reiss
- https://www.eecs.qmul.ac.uk/~josh/documents/2012/GiannoulisMassbergReiss-dynamicrangecompression-JAES2012.pdf

---

(an.)amp_follower_ar

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

Usage

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

Where:

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

Test

```
an = library("analyzers.lib");
os = library("oscillators.lib");
amp_follower_ar_test = os.osc(220) : an.amp_follower_ar(0.002, 0.05);
```

(an.)ms_envelope_rect

Mean square with moving-average algorithm.

Usage

```
_ : ms_envelope_rect(period) : _
```

Where:

- `period`: sets the averaging frame in secs

Test

```
an = library("analyzers.lib");
os = library("oscillators.lib");
ms_envelope_rect_test = an.ms_envelope_rect(0.05, os.osc(220));
```

(an.)ms_envelope_tau

Mean square average with one-pole lowpass and tau response (see filters.lib).

Usage

```
_ : ms_envelope_tau(period) : _
```

Where:

- `period`: (time to decay by 1/e) sets the averaging frame in secs

Test

```
an = library("analyzers.lib");
os = library("oscillators.lib");
ms_envelope_tau_test = an.ms_envelope_tau(0.05, os.osc(220));
```

(an.)ms_envelope_t60

Mean square with one-pole lowpass and t60 response (see filters.lib).

Usage

```
_ : ms_envelope_t60(period) : _
```

Where:

- `period`: (time to decay by 60 dB) sets the averaging frame in secs

Test

```
an = library("analyzers.lib");
os = library("oscillators.lib");
ms_envelope_t60_test = an.ms_envelope_t60(0.05, os.osc(220));
```

(an.)ms_envelope_t19

Mean square with one-pole lowpass and t19 response (see filters.lib).

Usage

```
_ : ms_envelope_t19(period) : _
```

Where:

- `period`: (time to decay by 1/e^2.2) sets the averaging frame in secs

Test

```
an = library("analyzers.lib");
os = library("oscillators.lib");
ms_envelope_t19_test = an.ms_envelope_t19(0.05, os.osc(220));
```

(an.)rms_envelope_rect

Root mean square with moving-average algorithm.

Usage

```
_ : rms_envelope_rect(period) : _
```

Where:

- `period`: sets the averaging frame in secs

Test

```
an = library("analyzers.lib");
os = library("oscillators.lib");
rms_envelope_rect_test = an.rms_envelope_rect(0.05, os.osc(220));
```

`(an.)rms_envelope_tau`

Root mean square with one-pole lowpass and tau response (see filters.lib).

Usage

```
_ : rms_envelope_tau(period) : _
```

Where:

- `period`: (time to decay by 1/e) sets the averaging frame in secs

Test

```
an = library("analyzers.lib");
os = library("oscillators.lib");
rms_envelope_tau_test = an.rms_envelope_tau(0.05, os.osc(220));
```

`(an.)rms_envelope_t60`

Root mean square with one-pole lowpass and t60 response (see filters.lib).

Usage

```
_ : rms_envelope_t60(period) : _
```

Where:

- `period`: (time to decay by 60 dB) sets the averaging frame in secs

Test

```
an = library("analyzers.lib");
os = library("oscillators.lib");
rms_envelope_t60_test = an.rms_envelope_t60(0.05, os.osc(220));
```

(an.)rms_envelope_t19

Root mean square with one-pole lowpass and t19 response (see filters.lib).

Usage

```
_ : rms_envelope_t19(period) : _
```

Where:

- `period`: (time to decay by 1/e^2.2) sets the averaging frame in secs

Test

```
an = library("analyzers.lib");
os = library("oscillators.lib");
rms_envelope_t19_test = an.rms_envelope_t19(0.05, os.osc(220));
```

(an.)zcr

Zero-crossing rate (ZCR) with one-pole lowpass averaging based on the tau constant. It outputs an index between 0 and 1 at a desired analysis frame. The ZCR of a signal correlates with the noisiness [Gouyon et al. 2000] and the spectral centroid [Herrera-Boyer et al. 2006] of a signal. For sinusoidal signals, the ZCR can be multiplied by ma.SR/2 and used as a frequency detector. For example, it can be deployed as a computationally efficient adaptive mechanism for automatic Larsen suppression.

Usage

```
_ : zcr(tau) : _
```

Where:

- `tau`: (time to decay by e^-1) sets the averaging frame in seconds.

Test

```
an = library("analyzers.lib");
os = library("oscillators.lib");
zcr_test = an.zcr(0.01, os.osc(220));
```

Adaptive Frequency Analysis

(an.)pitchTracker

This function implements a pitch-tracking algorithm by means of zero-crossing rate analysis and adaptive low-pass filtering. The design is based on the algorithm described in this tutorial (section 2.2).

Usage

```
_ : pitchTracker(N, tau) : _
```

Where:

- `N`: a constant numerical expression, sets the order of the low-pass filter, which determines the sensitivity of the algorithm for signals where partials are stronger than the fundamental frequency.
- `tau`: response time in seconds based on exponentially-weighted averaging with tau time-constant. See https://ccrma.stanford.edu/~jos/st/Exponentials.html.

Test

```
an = library("analyzers.lib");
os = library("oscillators.lib");
pitchTracker_test = an.pitchTracker(4, 0.02, os.osc(220));
```

(an.)spectralCentroid

This function implements a time-domain spectral centroid by means of RMS measurements and adaptive crossover filtering. The weight difference of the upper and lower spectral powers are used to recursively adjust the crossover cutoff so that the system (minimally) oscillates around a balancing point.

Unlike block processing techniques such as FFT, this algorithm provides continuous measurements and fast response times. Furthermore, when providing input signals that are spectrally sparse, the algorithm will output a logarithmic measure of the centroid, which is perceptually desirable for musical applications. For example, if the input signal is the combination of three tones at 1000, 2000, and 4000 Hz, the centroid will be the middle octave.

Usage

```
_ : spectralCentroid(nonlinearity, tau) : _
```

Where:

- `nonlinearity`: a boolean to activate or deactivate nonlinear integration. The nonlinear function is useful to improve stability with very short response times such as .001 <= tau <= .005 , otherwise, the nonlinearity may reduce precision.
- `tau`: response time in seconds based on exponentially-weighted averaging with tau time-constant. See https://ccrma.stanford.edu/~jos/st/Exponentials.html.

Example: `process = os.osc(500) + os.osc(1000) + os.osc(2000) + os.osc(4000) + os.osc(8000) : an.spectralCentroid(1, .001);`

Test

```
an = library("analyzers.lib");
os = library("oscillators.lib");
spectralCentroid_test = (os.osc(440) + os.osc(880)) : an.spectralCentroid(1, 0.01);
```

References Sanfilippo, D. (2021). Time-Domain Adaptive Algorithms for Low- and High-Level Audio Information Processing. Computer Music Journal, 45(1), 24-38.

Spectrum-Analyzers

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

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

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

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

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

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

References

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

`(an.)mth_octave_analyzer`

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

Usage

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

Also for convenience:

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

Where:

- `O`: (odd) 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)

Test

```
an = library("analyzers.lib");
os = library("oscillators.lib");
mth_octave_analyzer_test = os.osc(440) : an.mth_octave_analyzer(3, 3, 8000, 5);
```

Mth-Octave Spectral Level

Spectral Level: display (in bargraphs) the average signal level in each spectral band.

(an.)mth_octave_spectral_level6e

Spectral level display.

Usage:

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

Where:

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

Also for convenience:

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

Test

```
an = library("analyzers.lib");
os = library("oscillators.lib");
mth_octave_spectral_level6e_test = os.osc(440) : an.mth_octave_spectral_level6e(3, 8000, 5,
```

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

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

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

Usage See `mth_octave_spectral_level_demo` in `demos.lib`.

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.

`(an.)analyzer`

Analyzer.

Usage

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

Where:

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

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

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

Test

```
an = library("analyzers.lib");
os = library("oscillators.lib");
analyzer_test = os.osc(440) : an.analyzer(3, (500, 2000));
```

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

Sliding FFTs that compute a rectangularly windowed FFT each sample.

`(an.)goertzelOpt`

Optimized Goertzel filter.

Usage

```
_ : goertzelOpt(freq,n) : _
```

Where:

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

Test

```
an = library("analyzers.lib");
os = library("oscillators.lib");
goertzelOpt_test = an.goertzelOpt(440, 128, os.osc(440));
```

References

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

(an.)goertzelComp

Complex Goertzel filter.

Usage

```
_ : goertzelComp(freq,n) : _
```

Where:

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

Test

```
an = library("analyzers.lib");
os = library("oscillators.lib");
goertzelComp_test = an.goertzelComp(440, 128, os.osc(440));
```

References

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

(an.)goertzel

Same as `goertzelOpt`.

Usage

```
_ : goertzel(freq,n) : _
```

Where:

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

Test

```
an = library("analyzers.lib");
os = library("oscillators.lib");
goertzel_test = an.goertzel(440, 128, os.osc(440));
```

References

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

(an.)resonator

Efficient low-latency single-frequency resonator. It estimates the magnitude and phase of a single target frequency `f` in real time, with minimal memory and CPU usage, without the need for FFT or windowing.

Usage

```
_ : resonator(N,f) : _,_ // magnitude, phase
```

Where:

- `N`: smoothing filter order (compile-time constant). - N > 1: smoother magnitude/phase estimates, but slower response at low f - N = 1: faster response at low f, less stable at any f
- `f`: frequency to be analyzed (Hz).

Test

```
an = library("analyzers.lib");
os = library("oscillators.lib");
resonator_test = os.osc(440) : an.resonator(2, 440);
```

Algorithm Internally, the resonator maintains a quadrature oscillator at `f` and accumulates the projection of the input signal onto its sine and cosine components. These projections are smoothed using an exponential moving average (EWMA) whose decay factor depends on `f`:

```
sf(f) = 1 - exp(-f / (log(1+f) * SR))
```

Magnitude and phase are then computed from the smoothed projections:

```
magnitude = sqrt(so² + co²) * 2
phase     = atan2(so, co)
```

Example

```
F = nentry("F", 1000, 0, 10000, 0.001);
process = ba.line(ma.SR, ma.SR/2) : os.oscrs
         <: par(i, 4, resonator(i+1, F) : _,!);
```

Advantages

- Ultra-low latency: single-sample recursive update
- No FFT or windowing required
- Frequency-dependent smoothing for better stability at low f
- Scales linearly with the number of resonators

References

- https://alexandrefrancois.org/assets/publications/FrancoisARJ-ICMC2025.pdf

---

`(an.)fft`

Fast Fourier Transform (FFT).

Usage

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

Where:

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

FFTs of Real Signals:

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

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

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

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

Test

```
an = library("analyzers.lib");
os = library("oscillators.lib");
fft_test = an.rtocv(8, os.osc(220)) : an.fft(8);
```

References

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

`(an.)ifft`

Inverse Fast Fourier Transform (IFFT).

Usage

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

Where:

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

Test

```
an = library("analyzers.lib");
os = library("oscillators.lib");
ifft_test = (an.rtocv(8, os.osc(220)) : an.fft(8)) : an.ifft(8);
```

Test signal generators

Signal generators for testing purposes.

`(an.)logsweep`

Logarithmic sine sweep generator.

Usage

```
logsweep(fs,fe,dur) : _
```

Where:

- `fs`: start frequency in Hz

- `fe`: end frequency in Hz
- `dur`: duration of the sweep in seconds

Test

```
an = library("analyzers.lib");
logsweep_test = an.logsweep(20, 2000, 5);
```

`(an.)linsweep`

Linear sine sweep generator.

Usage

```
linsweep(fs,fe,dur) : _
```

Where:

- `fs`: start frequency in Hz
- `fe`: end frequency in Hz
- `dur`: duration of the sweep in seconds

Test

```
an = library("analyzers.lib");
linsweep_test = an.linsweep(20, 2000, 5);
```

basics.lib

Basics library. Its official prefix is `ba`.

This library provides reusable building blocks for core DSP and Faust programming. It typically includes low-level utilities for math, routing, signal conditioning, timing, control, and helper components used across higher-level libraries.

The Basics library is organized into 8 sections:

- Conversion Tools
- Counters and Time/Tempo Tools
- Array Processing/Pattern Matching
- Function tabulation
- Selectors (Conditions)
- Other
- Sliding Reduce
- Parallel Operators

References

- https://github.com/grame-cncm/faustlibraries/blob/master/basics.lib

Conversion Tools

(ba.)samp2sec

Converts a number of samples to a duration in seconds at the current sampling rate (see `ma.SR`). `samp2sec` is a standard Faust function.

Usage

`samp2sec(n) : _`

Where:

- `n`: number of samples

Test

```
ba = library("basics.lib");
samp2sec_test = ba.samp2sec(512);
```

(ba.)sec2samp

Converts a duration in seconds to a number of samples at the current sampling rate (see `ma.SR`). `samp2sec` is a standard Faust function.

Usage

`sec2samp(d) : _`

Where:

- `d`: duration in seconds

Test

```
ba = library("basics.lib");
sec2samp_test = ba.sec2samp(0.01);
```

(ba.)db2linear

dB-to-linear value converter. It can be used to convert an amplitude in dB to a linear gain]0-N]. `db2linear` is a standard Faust function.

Usage

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

Where:

- `l`: amplitude in dB

Test

```
ba = library("basics.lib");
db2linear_test = ba.db2linear(-6);
```

(ba.)linear2db

linea-to-dB value converter. It can be used to convert a linear gain]0-N] to an amplitude in dB. `linear2db` is a standard Faust function.

Usage

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

Where:

- `g`: a linear gain

Test

```
ba = library("basics.lib");
linear2db_test = ba.linear2db(0.5);
```

(ba.)lin2LogGain

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

Usage

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

Where:

- `n`: the linear gain

Test

```
ba = library("basics.lib");
lin2LogGain_test = ba.lin2LogGain(0.5);
```

(ba.)log2LinGain

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

Usage

`log2LinGain(n) : _`

Where:

- `n`: the log gain

Test

```
ba = library("basics.lib");
log2LinGain_test = ba.log2LinGain(0.25);
```

(ba.)tau2pole

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

Usage

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

Where:

- `tau`: time-constant in seconds

tau2pole(tau) = exp(-1.0/(tau*ma.SR));

Test

```
ba = library("basics.lib");
tau2pole_test = ba.tau2pole(0.01);
```

(ba.)pole2tau

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

Usage

`pole2tau(pole) : _`

Where:

- `pole`: the pole

Test

```
ba = library("basics.lib");
pole2tau_test = ba.pole2tau(0.9);
```

(ba.)midikey2hz

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

Usage

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

Where:

- `mk`: the MIDI key number

Test

```
ba = library("basics.lib");
midikey2hz_test = ba.midikey2hz(60);
```

(ba.)hz2midikey

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

Usage

```
hz2midikey(freq) : _
```

Where:

- `freq`: frequency in Hz

Test

```
ba = library("basics.lib");
hz2midikey_test = ba.hz2midikey(440);
```

(ba.)semi2ratio

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

Usage

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

Where:

- `semi`: number of semitone

Test

```
ba = library("basics.lib");
semi2ratio_test = ba.semi2ratio(7);
```

(ba.)ratio2semi

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

Usage

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

Where:

- `ratio`: frequency multiplicative ratio

Test

```
ba = library("basics.lib");
ratio2semi_test = ba.ratio2semi(2.0);
```

(ba.)cent2ratio

Converts cents in a frequency multiplicative ratio.

Usage

```
cent2ratio(cent) : _
```

Where:

- `cent`: number of cents

Test

```
ba = library("basics.lib");
cent2ratio_test = ba.cent2ratio(100);
```

(ba.)ratio2cent

Converts a frequency multiplicative ratio in cents.

Usage

```
ratio2cent(ratio) : _
```

Where:

- `ratio`: frequency multiplicative ratio

Test

```
ba = library("basics.lib");
ratio2cent_test = ba.ratio2cent(1.5);
```

(ba.)pianokey2hz

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

Usage

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

Where:

- `pk`: the piano key number

Test

```
ba = library("basics.lib");
pianokey2hz_test = ba.pianokey2hz(49);
```

(ba.)hz2pianokey

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

Usage

```
hz2pianokey(freq) : _
```

Where:

- `freq`: frequency in Hz

Test

```
ba = library("basics.lib");
hz2pianokey_test = ba.hz2pianokey(440);
```

Counters and Time/Tempo Tools

(ba.)counter

Starts counting 0, 1, 2, 3…, and raise the current integer value at each upfront of the trigger.

Usage

```
counter(trig) : _
```

Where:

- `trig`: the trigger signal, each upfront will move the counter to the next integer

Test

```
ba = library("basics.lib");
counter_test = ba.counter(button("trig"));
```

(ba.)countdown

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

Usage

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

Where:

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

Test

```
ba = library("basics.lib");
countdown_test = ba.countdown(8, button("trig"));
```

`(ba.)countup`

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

Usage

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

Where:

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

Test

```
ba = library("basics.lib");
countup_test = ba.countup(8, button("trig"));
```

`(ba.)sweep`

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

Usage

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

Test

```
ba = library("basics.lib");
sweep_test = ba.sweep(64, checkbox("run"));
```

`(ba.)time`

A simple counter that produces the sequence of 0,1,2...N integer values. `time` is a standard Faust function.

Usage

```
time : _
```

Test

```
ba = library("basics.lib");
time_test = ba.time;
```

(ba.)ramp

A linear ramp with a slope of '(+/-)1/n' samples to reach the next target value.

Usage

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

Where:

- `n`: number of samples to increment/decrement the value by one

Test

```
ba = library("basics.lib");
os = library("oscillators.lib");
ramp_test = os.osc(1) : ba.ramp(256);
```

(ba.)line

A ramp interpolator that generates a linear transition to reach a target value:

- the interpolation process restarts each time a new and distinct input value is received
- it utilizes 'n' samples to achieve the transition to the target value
- after reaching the target value, the output value is maintained.

Usage

```
_ : line(n) : _
```

Where:

- `n`: number of samples to reach the new target received at its input

Test

```
ba = library("basics.lib");
os = library("oscillators.lib");
line_test = os.osc(1) : ba.line(256);
```

(ba.)tempo

Converts a tempo in BPM into a number of samples.

Usage

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

Where:

- `t`: tempo in BPM

Test

```
ba = library("basics.lib");
tempo_test = ba.tempo(120);
```

(ba.)period

Basic sawtooth wave of period `p`.

Usage

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

Where:

- `p`: period as a number of samples

NOTE: may be this should go in oscillators.lib #### Test

```
ba = library("basics.lib");
period_test = ba.period(64);
```

(ba.)spulse

Produces a single pulse of n samples when trig goes from 0 to 1.

Usage

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

Where:

- `n`: pulse length as a number of samples
- `trig`: the trigger signal (1: start the pulse)

Test

```
ba = library("basics.lib");
spulse_test = ba.spulse(32, button("trig"));
```

(ba.)pulse

Pulses (like 10000) generated at period `p`.

Usage

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

Where:

- `p`: period as a number of samples

NOTE: may be this should go in oscillators.lib #### Test

```
ba = library("basics.lib");
pulse_test = ba.pulse(64);
```

(ba.)pulsen

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

Usage

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

Where:

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

NOTE: may be this should go in oscillators.lib #### Test

```
ba = library("basics.lib");
pulsen_test = ba.pulsen(8, 64);
```

(ba.)cycle

Split nonzero input values into `n` cycles.

Usage

```
_ : cycle(n) : si.bus(n)
```

Where:

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

Test

```
ba = library("basics.lib");
cycle_test = button("gate") : ba.cycle(3);
```

`(ba.)beat`

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

Usage

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

Where:

- `t`: tempo in BPM

Test

```
ba = library("basics.lib");
beat_test = ba.beat(120);
```

`(ba.)pulse_countup`

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

Usage

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

Where:

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

Test

```
ba = library("basics.lib");
pulse_countup_test = ba.pulse_countup(button("run"));
```

(ba.)pulse_countdown

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

Usage

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

Where:

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

Test

```
ba = library("basics.lib");
pulse_countdown_test = ba.pulse_countdown(button("run"));
```

(ba.)pulse_countup_loop

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

Usage

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

Where:

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

Test

```
ba = library("basics.lib");
pulse_countup_loop_test = ba.pulse_countup_loop(4, button("run"));
```

(ba.)pulse_countdown_loop

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

Usage

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

Where:

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

Test

```
ba = library("basics.lib");
pulse_countdown_loop_test = ba.pulse_countdown_loop(4, button("run"));
```

(ba.)resetCtr

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

Usage

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

Where:

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

Test

```
ba = library("basics.lib");
resetCtr_test = ba.pulse(16) : ba.resetCtr(4, 2);
```

Array Processing/Pattern Matching

(ba.)count

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

Usage

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

Where:

- `l`: list of elements

Test

```
ba = library("basics.lib");
count_test = ba.count((10,20,30,40));
```

(ba.)take

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

Usage

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

Where:

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

Test

```
ba = library("basics.lib");
take_test = ba.take(3, (10,20,30,40));
```

(ba.)pick

Pick the nth element from a list. Similar to `ba.take(n+1,l)` but faster and more powerful.

Usage

```
pick(l,n) : _
```

Where:

- `l`: list of elements
- `n`: index of element to pick, compile time constant. if n < 0 or n >= length of `l`, `pick()` outputs 0.

Example test program

```
pick((10,20,30,40), 2) -> 30
```

```
pick(si.bus(3), 1) // same as !,_,!
```

while `ba.take(2, si.bus(3))` acts as `_`.

Unlike `take()`, `pick()` always flattens the list, so `pick((10,(20,30),40), 1)` outputs 20, not `20,30`.

Test

```
ba = library("basics.lib");
pick_test = ba.pick((10,20,30,40), 2);
```

`(ba.)pickN`

Select the inputs listed in `O` among `N` at compile time.

Usage

```
si.bus(N) : pickN(N,O) : si.bus(outputs(O))
```

Where:

- `N`: number of inputs, compile time constant
- `O`: list of the inputs to select, compile time constants

Example test program

```
pickN(4,2) : _  // same as selector(2,4) but faster
```

```
pick(4,(1,3)) : _,_ // same as !,_,!,_
```

```
pickN(4,(1,3), (10,20,30,40)) -> (20,40)
```

```
process = pickN(2, (1,0,0,1)) // same as `process(x,y) = y,x,x,y`
```

Test

```
ba = library("basics.lib");
pickN_test = (1,2,3,4) : ba.pickN(4, (0,2));
```

`(ba.)subseq`

Extract a part of a list.

Usage

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

Where:

- `l`: list
- `P`: start point (int, known at compile time, 0: begin of list)
- `N`: number of elements (int, known at compile time)

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

Test

```
ba = library("basics.lib");
subseq_test = ba.subseq((10,20,30,40,50), 1, 3);
```

Function tabulation

The purpose of function tabulation is to speed up the computation of heavy functions over an interval, so that the computation at runtime can be faster than directly using the function. Two techniques are implemented:

- `tabulate` computes the function in a table and read the points using interpolation. `tabulateNd` is the N dimensions version of `tabulate`
- `tabulate_chebychev` uses Chebyshev polynomial approximation

Comparison program example

```
process = line(50000, r0, r1) <: FX-tb,FX-ch : par(i, 2, maxerr)
with {
   C = 0;
   FX = sin;
   NX = 50;
   CD = 3;
   r0 = 0;
   r1 = ma.PI;
   tb(x) = ba.tabulate(C, FX, NX*(CD+1), r0, r1, x).cub;
   ch(x) = ba.tabulate_chebychev(C, FX, NX, CD, r0, r1, x);
   maxerr = abs : max ~ _;
   line(n, x0, x1) = x0 + (ba.time%n)/n * (x1-x0);
};
```

`(ba.)tabulate`

Tabulate a 1D function over the range [r0, r1] for access via nearest-value, linear, cubic interpolation. In other words, the uniformly tabulated function can be evaluated using interpolation of order 0 (none), 1 (linear), or 3 (cubic).

Usage

```
tabulate(C, FX, S, r0, r1, x).(val|lin|cub) : _
```

- `C`: whether to dynamically force the `x` value to the range [r0, r1]: 1 forces the check, 0 deactivates it (constant numerical expression)

- `FX`: unary function Y=F(X) with one output (scalar function of one variable)
- `S`: size of the table in samples (constant numerical expression)
- `r0`: minimum value of argument x
- `r1`: maximum value of argument x

`tabulate(C, FX, S, r0, r1, x).val uses the value in the table closest to x`

`tabulate(C, FX, S, r0, r1, x).lin evaluates at x using linear interpolation between the clos`

`tabulate(C, FX, S, r0, r1, x).cub evaluates at x using cubic interpolation between the close`

Example test program

```
midikey2hz(mk) = ba.tabulate(1, ba.midikey2hz, 512, 0, 127, mk).lin;
process = midikey2hz(ba.time), ba.midikey2hz(ba.time);
```

Test

```
ba = library("basics.lib");
tabulate_test = ba.tabulate(1, ba.midikey2hz, 128, 0, 127, 60).lin;
```

`(ba.)tabulate_chebychev`

Tabulate a 1D function over the range [r0, r1] for access via Chebyshev polynomial approximation. In contrast to `(ba.)tabulate`, which interpolates only between tabulated samples, `(ba.)tabulate_chebychev` stores coefficients of Chebyshev polynomials that are evaluated to provide better approximations in many cases. Two new arguments controlling this are NX, the number of segments into which [r0, r1] is divided, and CD, the maximum Chebyshev polynomial degree to use for each segment. A `rdtable` of size NX*(CD+1) is internally used.

Note that processing `r1` the last point in the interval is not safe. So either be sure the input stays in [r0, r1[or use `C = 1`.

Usage

```
_ : tabulate_chebychev(C, FX, NX, CD, r0, r1) : _
```

- `C`: whether to dynamically force the value to the range [r0, r1]: 1 forces the check, 0 deactivates it (constant numerical expression)
- `FX`: unary function Y=F(X) with one output (scalar function of one variable)
- `NX`: number of segments for uniformly partitioning [r0, r1] (constant numerical expression)
- `CD`: maximum polynomial degree for each Chebyshev polynomial (constant numerical expression)

- `r0`: minimum value of argument x
- `r1`: maximum value of argument x

Example test program

```
midikey2hz_chebychev(mk) = ba.tabulate_chebychev(1, ba.midikey2hz, 100, 4, 0, 127, mk);
process = midikey2hz_chebychev(ba.time), ba.midikey2hz(ba.time);
```

Test

```
ba = library("basics.lib");
tabulate_chebychev_test = ba.tabulate_chebychev(1, ba.midikey2hz, 32, 4, 0, 127, 60);
```

`(ba.)tabulateNd`

Tabulate an nD function for access via nearest-value or linear or cubic interpolation. In other words, the tabulated function can be evaluated using interpolation of order 0 (none), 1 (linear), or 3 (cubic).

The table size and parameter range of each dimension can and must be separately specified. You can use it anywhere you have an expensive function with multiple parameters with known ranges. You could use it to build a wavetable synth, for example.

The number of dimensions is deduced from the number of parameters you give, see below.

Note that processing the last point in each interval is not safe. So either be sure the inputs stay in their respective ranges, or use `C = 1`. Similarly for the first point when doing cubic interpolation.

Usage

```
tabulateNd(C, function, (parameters) ).(val|lin|cub) : _
```

- `C`: whether to dynamically force the parameter values for each dimension to the ranges specified in parameters: 1 forces the check, 0 deactivates it (constant numerical expression)
- `function`: the function we want to tabulate. Can have any number of inputs, but needs to have just one output.
- `(parameters)`: sizes, ranges and read values. Note: these need to be in brackets, to make them one entity.

  If N is the number of dimensions, we need:

  - N times `S`: number of values to store for this dimension (constant numerical expression)

- N times `r0`: minimum value of this dimension
- N times `r1`: maximum value of this dimension
- N times `x`: read value of this dimension

By providing these parameters, you indirectly specify the number of dimensions; it's the number of parameters divided by 4.

The user facing functions are:

`tabulateNd(C, function, S, parameters).val`

- Uses the value in the table closest to x.

`tabulateNd(C, function, S, parameters).lin`

- Evaluates at x using linear interpolation between the closest stored values.

`tabulateNd(C, function, S, parameters).cub`

- Evaluates at x using cubic interpolation between the closest stored values.

Example test program

```
powSin(x,y) = sin(pow(x,y)); // The function we want to tabulate
powSinTable(x,y) = ba.tabulateNd(1, powSin, (sizeX,sizeY, rx0,ry0, rx1,ry1, x,y) ).lin;
sizeX = 512; // table size of the first parameter
sizeY = 512; // table size of the second parameter
rx0 = 2; // start of the range of the first parameter
ry0 = 2; // start of the range of the second parameter
rx1 = 10; // end of the range of the first parameter
ry1 = 10; // end of the range of the second parameter
x = hslider("x", rx0, rx0, rx1, 0.001):si.smoo;
y = hslider("y", ry0, ry0, ry1, 0.001):si.smoo;
process = powSinTable(x,y), powSin(x,y);
```

Working principle The `.val` function just outputs the closest stored value. The `.lin` and `.cub` functions interpolate in N dimensions.

Multi dimensional interpolation To understand what it means to interpolate in N dimensions, here's a quick reminder on the general principle of 2D linear interpolation:

- We have a grid of values, and we want to find the value at a point (x, y) within this grid.

- We first find the four closest points (A, B, C, D) in the grid surrounding the point (x, y).

Then, we perform linear interpolation in the x-direction between points A and B, and between points C and D. This gives us two new points E and F. Finally, we perform linear interpolation in the y-direction between points E and F to get our value.

To implement this in Faust, we need N sequential groups of interpolators, where N is the number of dimensions.
Each group feeds into the next, with the last "group" being a single interpolator, and the group before it containing one interpolator for each input of the group it's feeding.

Some examples:

- Our 2D linear example has two interpolators feeding into one.
- A 3D linear interpolator has four interpolators feeding into two, feeding into one.
- A 2D cubic interpolater has four interpolators feeding into one.
- A 3D cubic interpolator has sixteen interpolators feeding into four, feeding into one.

To understand which values we need to look up, let's consider the 2D linear example again. The four values going into the first group represent the four closest points (A, B, C, D) mentioned above.

1) The first interpolator gets:

- The closest value that is stored (A)
- The next value in the x dimension, keeping y fixed (B)

2) The second interpolator gets:

- One step over in the y dimension, keeping x fixed (C)
- One step over in both the x dimension and the y dimension (D)

The outputs of these two interpolators are points E and F. In other words: the interpolated x values and, respectively, the following y values:

- The closest stored value of the y dimension
- One step forward in the y dimension

The last interpolator takes these two values and interpolates them in the y dimension.

To generalize for N dimensions and linear interpolation:

- The first group has 2^(n-1) parallel interpolators interpolating in the first dimension.
- The second group has 2^(n-2) parallel interpolators interpolating in the second dimension.
- The process continues until the n-th group, which has a single interpolator interpolating in the n-th dimension.

The same principle applies to the cubic interpolation in nD. The only difference is that there would be 4^(n-1) parallel interpolators in the first group, compared to 2^(n-1) for linear interpolation.

This is what the `mixers` function does.

Besides the values, each interpolator also needs to know the weight of each value in it's output.
Let's call this `d`, like in `ba.interpolate`. It is the same for each group of interpolators, since it correlates to a dimension.
It's value is calculated the similarly to `ba.interpolate`:

- First we prepare a "float table read-index" for that dimension (`id` in `ba.tabulate`)
- If the table only had that dimension and it could read a float index, what would it be.
- Then we `int` the float index to get the value we have stored that is closest to, but lower than the input value; the actual index for that dimension. Our `d` is the difference between the float index and the actual index.

The `ids` function calculates the `id` for each dimension and inside the `mixer` function they get turned into `d`s.

Storage method The elephant in the room is: how do we get these indexes? For that we need to know how the values are stored. We use one big table to store everything.

To understand the concept, let's look at the 2D example again, and then we'll extend it to 3d and the general nD case.

Let's say we have a 2D table with dimensions A and B where: A has 3 values between 0 and 5 and B has 4 values between 0 and 1. The 1D array representation of this 2D table will have a size of 3 * 4 = 12.

The values are stored in the following way:

- First 3 values: A is 0, then 3, then 5 while B is at 0.
- Next 3 values: A changes from 0 to 5 while B is at 1/3.
- Next 3 values: A changes from 0 to 5 while B is at 2/3.
- Last 3 values: A changes from 0 to 5 while B is at 1.

For the 3D example, let's extend the 2D example with an additional dimension C having 2 values between 0 and 2. The total size will be 3 * 4 * 2 = 24.

The values are stored like so:

- First 3 values: A changes from 0 to 5, B is at 0, and C is at 0.
- Next 3 values: A changes from 0 to 5, B is at 1/3, and C is at 0.
- Next 3 values: A changes from 0 to 5, B is at 2/3, and C is at 0.
- Next 3 values: A changes from 0 to 5, B is at 1, and C is at 0.

The last 12 values are the same as the first 12, but with C at 2.

For the general n-dimensional case, we iterate through all dimensions, changing the values of the innermost dimension first, then moving towards the outer dimensions.

Read indexes To get the float read index (`id`) corresponding to a particular dimension, we scale the function input value to be between 0 and 1, and multiply it by the size of that dimension minus one.

To understand how we get the `readIndex`for `.val`, let's work trough how we'd do it in our 2D linear example.
For simplicity's sake, the ranges of the inputs to our `function` are both 0 to 1. Say we wanted to read the value closest to `x=0.5` and `y=0`, so the `id` of `x` is `1` (the second value) and the `id` of `y` is 0 (first value). In this case, the read index is just the `id` of `x`, rounded to the nearest integer, just like in `ba.tabulate`.

If we want to read the value belonging to `x=0.5` and `y=2/3`, things get more complicated. The `id` for `y` is now `2`, the third value. For each step in the `y` direction, we need to increase the index by `3`, the number of values that are stored for `x`. So the influence of the `y` is: the size of `x` times the rounded `id` of `y`. The final read index is the rounded `id` of `x` plus the influence of `y`.

For the general nD case, we need to do the same operation N times, each feeding into the next. This operation is the `riN` function. We take four parameters: the size of the dimension before it `prevSize`, the index of the previous dimension `prevIX`, the current size `sizeX` and the current id `idX`. `riN` has 2 outputs, the size, for feeding into the next dimension's `prevSize`, and the read index feeding into the next dimension's `prevIX`.
The size is the `sizeX` times `prevSize`. The read index is the rounded `idX` times `prevSize` added to the `prevIX`. Our final `readIndex` is the read index output of the last dimension.

To get the read values for the interpolators need a pattern of offsets in each dimension, since we are looking for the read indexes surrounding the point of interest. These offsets are best explained by looking at the code of `tabulate2d`, the hardcoded 2D version:

```
tabulate2d(C,function, sizeX,sizeY, rx0,ry0, rx1,ry1, x,y) =
  environment {
    size = sizeX*sizeY;
    // Maximum X index to access
    midX = sizeX-1;
    // Maximum Y index to access
    midY = sizeY-1;
    // Maximum total index to access
    mid = size-1;
    // Create the table
    wf = function(wfX,wfY);
    // Prepare the 'float' table read index for X
    idX = (x-rx0)/(rx1-rx0)*midX;
    // Prepare the 'float' table read index for Y
```

```
idY = ((y-ry0)/(ry1-ry0))*midY;
// table creation X:
wfX =
  rx0+float(ba.time%sizeX)*(rx1-rx0)
  /float(midX);
// table creation Y:
wfY =
  ry0+
  ((float(ba.time-(ba.time%sizeX))
    /float(sizeX))
   *(ry1-ry0))
  /float(midY);

// Limit the table read index in [0, mid] if C = 1
rid(x,mid, 0) = x;
rid(x,mid, 1) = max(0, min(x, mid));

// Tabulate a binary 'FX' function on a range [rx0, rx1] [ry0, ry1]
val(x,y) =
  rdtable(size, wf, readIndex);
readIndex =
  rid(
    rid(int(idX+0.5),midX, C)
    +yOffset
  , mid, C);
yOffset = sizeX*rid(int(idY),midY,C);

// Tabulate a binary 'FX' function over the range [rx0, rx1] [ry0, ry1] with linear inte
lin =
  it.interpolate_linear(
    dy
  , it.interpolate_linear(dx,v0,v1)
  , it.interpolate_linear(dx,v2,v3))
with {
  i0 = rid(int(idX), midX, C)+yOffset;
  i1 = i0+1;
  i2 = i0+sizeX;
  i3 = i1+sizeX;
  dx  = idX-int(idX);
  dy  = idY-int(idY);
  v0 = rdtable(size, wf, rid(i0, mid, C));
  v1 = rdtable(size, wf, rid(i1, mid, C));
  v2 = rdtable(size, wf, rid(i2, mid, C));
  v3 = rdtable(size, wf, rid(i3, mid, C));
};
```

```
// Tabulate a binary 'FX' function over the range [rx0, rx1] [ry0, ry1] with cubic inter
cub =
  it.interpolate_cubic(
    dy
  , it.interpolate_cubic(dx,v0,v1,v2,v3)
  , it.interpolate_cubic(dx,v4,v5,v6,v7)
  , it.interpolate_cubic(dx,v8,v9,v10,v11)
  , it.interpolate_cubic(dx,v12,v13,v14,v15)
  )
with {
  i0  = i4-sizeX;
  i1  = i5-sizeX;
  i2  = i6-sizeX;
  i3  = i7-sizeX;

  i4  = i5-1;
  i5  = rid(int(idX), midX, C)+yOffset;
  i6  = i5+1;
  i7  = i6+1;

  i8  = i4+sizeX;
  i9  = i5+sizeX;
  i10 = i6+sizeX;
  i11 = i7+sizeX;

  i12 = i4+(2*sizeX);
  i13 = i5+(2*sizeX);
  i14 = i6+(2*sizeX);
  i15 = i7+(2*sizeX);

  dx  = idX-int(idX);
  dy  = idY-int(idY);
  v0  = rdtable(size, wf, rid(i0 , mid, C));
  v1  = rdtable(size, wf, rid(i1 , mid, C));
  v2  = rdtable(size, wf, rid(i2 , mid, C));
  v3  = rdtable(size, wf, rid(i3 , mid, C));
  v4  = rdtable(size, wf, rid(i4 , mid, C));
  v5  = rdtable(size, wf, rid(i5 , mid, C));
  v6  = rdtable(size, wf, rid(i6 , mid, C));
  v7  = rdtable(size, wf, rid(i7 , mid, C));
  v8  = rdtable(size, wf, rid(i8 , mid, C));
  v9  = rdtable(size, wf, rid(i9 , mid, C));
  v10 = rdtable(size, wf, rid(i10, mid, C));
  v11 = rdtable(size, wf, rid(i11, mid, C));
  v12 = rdtable(size, wf, rid(i12, mid, C));
  v13 = rdtable(size, wf, rid(i13, mid, C));
```

```
      v14 = rdtable(size, wf, rid(i14, mid, C));
      v15 = rdtable(size, wf, rid(i15, mid, C));
    };
  };
```

In the interest of brevity, we'll stop explaining here. If you have any more questions, feel free to open an issue on faustlibraries and tag @magnetophon.

Test

```
ba = library("basics.lib");
powSin(x,y) = sin(pow(x,y));
tabulateNd_test = ba.tabulateNd(1, powSin, (8,8, 2.0,2.0, 8.0,8.0, 3.0,4.0)).lin;
```

Selectors (Conditions)

`(ba.)if`

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

Usage

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

Where:

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

Test

```
ba = library("basics.lib");
if_test = ba.if(1, 0.5, -0.5);
```

`(ba.)ifNc`

if-then-elseif-then-…elsif-then-else implemented on top of `ba.if`.

Usage

```
ifNc((cond1,then1, cond2,then2, ... condN,thenN, else)) : _
```

or

```
ifNc(Nc, cond1,then1, cond2,then2, ... condN,thenN, else) : _
```

or

```
cond1,then1, cond2,then2, ... condN,thenN, else : ifNc(Nc) : _
```

Where:

- `Nc` : number of branches/conditions (constant numerical expression)
- `condX`: condition
- `thenX`: signal selected if condX is the 1st true condition
- `else`: signal selected if all the cond1-condN conditions are false

Example test program

```
process(x,y) = ifNc((x<y,-1, x>y,+1, 0));
```

or

```
process(x,y) = ifNc(2, x<y,-1, x>y,+1, 0);
```

or

```
process(x,y) = x<y,-1, x>y,+1, 0 : ifNc(2);
```

outputs `-1` if `x<y`, `+1` if `x>y`, `0` otherwise.

Test

```
ba = library("basics.lib");
ifNc_test = ba.ifNc((1, 10, 0, 20, 30));
```

`(ba.)ifNcNo`

`ifNcNo(Nc,No)` is similar to `ifNc(Nc)` above but then/else branches have `No` outputs.

Usage

```
ifNcNo(Nc,No, cond1,then1, cond2,then2, ... condN,thenN, else) : sig.bus(No)
```

Where:

- `Nc` : number of branches/conditions (constant numerical expression)
- `No` : number of outputs (constant numerical expression)
- `condX`: condition
- `thenX`: list of No signals selected if condX is the 1st true condition
- `else`: list of No signals selected if all the cond1-condN conditions are false

Example test program

```
process(x) = ifNcNo(2,3, x<0, -1,-1,-1, x>0, 1,1,1, 0,0,0);
```

outputs `-1,-1,-1` if `x<0`, `1,1,1` if `x>0`, `0,0,0` otherwise.

Test

```
ba = library("basics.lib");
ifNcNo_test = (1, 10, 0, 20, 30) : ba.ifNcNo(2, 1);
```

`(ba.)selector`

Selects the ith input among n at compile time.

Usage

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

Where:

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

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

Test

```
ba = library("basics.lib");
selector_test = (0.1, 0.2, 0.3, 0.4) : ba.selector(2, 4);
```

`(ba.)select2stereo`

Select between 2 stereo signals.

Usage

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

Where:

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

Test

```
ba = library("basics.lib");
select2stereo_test = ba.select2stereo(1, (0.1,0.2, 0.3,0.4));
```

(ba.)selectn

Selects the ith input among N at run time.

Usage

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

Where:

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

Example test program

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

Test

```
ba = library("basics.lib");
selectn_test = (1,2,3,4) : ba.selectn(4, 2);
```

(ba.)selectbus

Select a bus among `NUM_BUSES` buses, where each bus has `BUS_SIZE` outputs. The order of the signal inputs should be the signals of the first bus, the signals of the second bus, and so on.

Usage

```
process = si.bus(BUS_SIZE*NUM_BUSES) : selectbus(BUS_SIZE, NUM_BUSES, id) : si.bus(BUS_SIZE)
```

Where:

- `BUS_SIZE`: number of outputs from each bus (int, known at compile time).
- `NUM_BUSES`: number of buses (int, known at compile time).
- `id`: index of the bus to select (int, `0<=id<NUM_BUSES`)

Test

```
ba = library("basics.lib");
selectbus_test = (1,2,3,4) : ba.selectbus(2, 2, 1);
```

(ba.)selectxbus

Like `ba.selectbus`, but with a cross-fade when selecting the bus using the same technique than `ba.selectmulti`.

Usage

```
process = si.bus(BUS_SIZE*NUM_BUSES) : selectbus(BUS_SIZE, NUM_BUSES, FADE, id) : si.bus(BUS
```

Where:

- `BUS_SIZE`: number of outputs from each bus (int, known at compile time).
- `NUM_BUSES`: number of buses (int, known at compile time).
- `fade`: number of samples for the crossfade.
- `id`: index of the bus to select (int, `0<=id<NUM_BUSES`)

Test

```
ba = library("basics.lib");
selectxbus_test = (1,2,3,4) : ba.selectxbus(2, 2, 16, checkbox("bus"));
```

(ba.)selectmulti

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

Usage

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

Where:

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

Example test program

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

Test

```
ba = library("basics.lib");
os = library("oscillators.lib");
ma = library("maths.lib");
effects = ((_*0.5,_*0.5),(_*0.25,_*0.25));
choice = int(checkbox("choice"));
selectmulti_test = (os.osc(440), os.osc(660)) : ba.selectmulti(ma.SR/100, effects, choice);
```

(ba.)selectoutn

Route input to the output among N at run time.

Usage

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

Where:

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

Example test program

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

Test

```
ba = library("basics.lib");
selectoutn_test = 1 : ba.selectoutn(3, 1);
```

Other

(ba.)latch

Latch input on the rising edge of trig. Captures ("records") the input x whenever trig crosses from 0 to >0, and holds the last captured value at all other times.

Usage

```
_ : latch(trig) : _
```

Where:

- `trig`: trigger signal. A rising edge (0 → >0) samples the input.

Test

```
ba = library("basics.lib");
os = library("oscillators.lib");
latch_test = os.osc(2) : ba.latch(button("hold"));
```

(ba.)sAndH

Sample And Hold: "records" the input when trig is 1, outputs a frozen value when trig is 0. `sAndH` is a standard Faust function.

Usage

```
_ : sAndH(trig) : _
```

Where:

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

Test

```
ba = library("basics.lib");
os = library("oscillators.lib");
sAndH_test = os.osc(2) : ba.sAndH(button("hold"));
```

(ba.)tAndH

Test And Hold: "records" the input when pred(input) is true, outputs a frozen value otherwise.

Usage

```
_ : tAndH(pred) : _
```

Where:

- `pred`: predicate to test the input

Test

```
ba = library("basics.lib");
os = library("oscillators.lib");
isPositive(x) = x > 0.0;
tAndH_test = os.osc(2) : ba.tAndH(isPositive);
```

`(ba.)downSample`

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

Usage

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

Where:

- `freq`: new rate in Hz

Test

```
ba = library("basics.lib");
os = library("oscillators.lib");
downSample_test = os.osc(440) : ba.downSample(11025);
```

`(ba.)downSampleCV`

A version of `ba.downSample` where the frequency parameter has been replaced by an `amount` parameter that is in the range zero to one. WARNING: this function doesn't change the rate of a signal, it just holds samples…

Usage

```
_ : downSampleCV(amount) : _
```

Where:

- `amount`: The amount of down-sampling to perform [0..1]

Test

```
ba = library("basics.lib");
os = library("oscillators.lib");
downSampleCV_test = os.osc(440) : ba.downSampleCV(0.5);
```

`(ba.)peakhold`

Outputs current max value above zero.

Usage

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

Where:

`mode` means:

0 - Pass through. A single sample 0 trigger will work as a reset.

1 - Track and hold max value.

Test

```
ba = library("basics.lib");
os = library("oscillators.lib");
peakhold_test = os.osc(440) : ba.peakhold(1);
```

`(ba.)peakholder`

While peak-holder functions are scarcely discussed in the literature (please do send me an email if you know otherwise), common sense tells that the expected behaviour should be as follows: the absolute value of the input signal is compared with the output of the peak-holder; if the input is greater or equal to the output, a new peak is detected and sent to the output; otherwise, a timer starts and the current peak is held for N samples; once the timer is out and no new peaks have been detected, the absolute value of the current input becomes the new peak.

Usage

```
_ : peakholder(holdTime) : _
```

Where:

- `holdTime`: hold time in samples

Test

```
ba = library("basics.lib");
os = library("oscillators.lib");
peakholder_test = os.osc(440) : ba.peakholder(ba.sec2samp(0.1));
```

`(ba.)kr2ar`

Force a control rate signal to be used as an audio rate signal.

Usage

```
hslider("freq", 200, 200, 2000, 0.1) : kr2ar;
```

Test

```
ba = library("basics.lib");
kr2ar_test = button("gate") : ba.kr2ar;
```

(ba.)impulsify

Turns a signal into an impulse with the value of the current sample (0.3,0.2,0.1 becomes 0.3,0.0,0.0). This function is typically used with a `button` to turn its output into an impulse. `impulsify` is a standard Faust function.

Usage

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

Test

```
ba = library("basics.lib");
os = library("oscillators.lib");
impulsify_test = os.osc(440) : ba.impulsify;
```

(ba.)automat

Record and replay in a loop the successives values of the input signal.

Usage

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

Where:

- `t`: tempo in BPM
- `size`: number of items in the loop
- `init`: init value in the loop

Test

```
ba = library("basics.lib");
autoControl = hslider("autoControl", 0.2, 0, 1, 0.01);
automat_test = autoControl : ba.automat(120, 4, 0.0);
```

`(ba.)bpf`

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

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

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}`

In addition to `bpf.point`, there are also `step` and `curve` functions:

- `step(x,y)` to add a flat section
- `step_end(x,y)` to end with a flat section
- `curve(B,x,y)` to add a curved section
- `curve_end(B,x,y)` to end with a curved section

These functions can be combined with the other `bpf` functions.

Here's an example using `bpf.step`:

```
f(x) = x : bpf.start(0,0) : bpf.step(.2,.3) : bpf.step(.4,.6) :
bpf.step_end(1,1);
```

For `x < 0.0`, the output is 0.0. For `0.0 <= x < 0.2`, the output is 0.0. For `0.2 <= x < 0.4`, the output is 0.3. For `0.4 <= x < 1.0`, the output is 0.6. For `1.0 <= x`, the output is 1.0

For the `curve` functions, `B` (compile-time constant) is a "bias" value strictly greater than zero and less than or equal to 1. When `B` is 0.5, the output curve is exactly linear and equivalent to `bpf.point`. When `B` is less than 0.5, the output is biased towards the `y` value of the previous breakpoint. When `B` is greater than

0.5, the output is biased towards the `y` value of the curve breakpoint. Here's an example:

```
f = bpf.start(0,0) : bpf.curve(.15,.5,.5) : bpf.curve_end(.85,1,1);
```

In the following example, the output is biased towards zero (the latter y value) instead of being a linear ramp from 1 to 0.

```
f = bpf.start(0,1) : bpf.curve_end(.9,1,0);
```

`bpf` is a standard Faust function.

`(ba.)listInterp`

Linearly interpolates between the elements of a list.

Usage

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

Where:

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

Test

```
ba = library("basics.lib");
listInterp_test = ba.listInterp((800,400,350,450,325), 1.5);
```

`(ba.)bypass1`

Takes a mono input signal, route it to `e` and bypass it if `bpc = 1`. When bypassed, `e` is feed with zeros so that its state is cleanup up. `bypass1` is a standard Faust function.

Usage

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

Where:

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

Test

```
ba = library("basics.lib");
os = library("oscillators.lib");
bypass1_test = os.osc(440) : ba.bypass1(button("bypass"), *(0.5));
```

(ba.)bypass2

Takes a stereo input signal, route it to `e` and bypass it if `bpc = 1`. When bypassed, `e` is feed with zeros so that its state is cleanup up. `bypass2` is a standard Faust function.

Usage

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

Where:

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

Test

```
ba = library("basics.lib");
os = library("oscillators.lib");
bypass2_test = (os.osc(440), os.osc(660)) : ba.bypass2(button("bypass"), par(i,2, *(0.5)));
```

(ba.)bypass1to2

Bypass switch for effect `e` having mono input signal and stereo output. Effect `e` is bypassed if `bpc = 1`.When bypassed, `e` is feed with zeros so that its state is cleanup up. `bypass1to2` is a standard Faust function.

Usage

```
_ : bypass1to2(bpc,e) : _,_
```

Where:

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

Test

```
ba = library("basics.lib");
os = library("oscillators.lib");
monoToStereo(x) = (x*0.5, x*0.25);
```

```
bypass1to2_test = os.osc(440) : ba.bypass1to2(button("bypass"), monoToStereo);
```

(ba.)bypass_fade

Bypass an arbitrary (N x N) circuit with 'n' samples crossfade. Inputs and outputs signals are faded out when 'e' is bypassed, so that 'e' state is cleanup up. Once bypassed the effect is replaced by `par(i,N,_)`. Bypassed circuits can be chained.

Usage

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

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

Example test program

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

Test

```
ba = library("basics.lib");
os = library("oscillators.lib");
bypass_fade_test = (os.osc(440), os.osc(660)) : ba.bypass_fade(128, button("bypass"), par(i
```

(ba.)toggle

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

Usage

```
_ : toggle : _
```

Example test program

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

Test

```
ba = library("basics.lib");
toggle_test = ba.toggle(button("trig"));
```

(ba.)on_and_off

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

Usage

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

Example test program

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

Test

```
ba = library("basics.lib");
on_and_off_test = button("on"), button("off") : ba.on_and_off;
```

(ba.)bitcrusher

Produce distortion by reduction of the signal resolution.

Usage

```
_ : bitcrusher(nbits) : _
```

Where:

- `nbits`: the number of bits of the wanted resolution

Test

```
ba = library("basics.lib");
os = library("oscillators.lib");
bitcrusher_test = os.osc(440) : ba.bitcrusher(8);
```

(ba.)mulaw_bitcrusher

Produce distortion by reducing the signal resolution using -law compression.

Usage

```
_ : mulaw_bitcrusher(mu,nbits) : _
```

Where:

- `mu`: controls the degree of -law compression, larger values result in stronger compression
- `nbits`: the number of bits of the wanted resolution

Description The `mulaw_bitcrusher` applies a combination of -law compression, quantization, and expansion to create a non-linear bitcrushed effect. This method retains finer detail in lower-amplitude signals compared to linear bitcrushing, making it suitable for creative sound design.

Theory

1. **-law Compression**: emphasizes lower-amplitude signals by applying a logarithmic curve to the signal. The formula used is:

   ```
   F(x) = ma.signum(x) * log(1 + mu * abs(x)) / log(1 + mu);
   ```

2. **Quantization**: reduces the signal resolution to `nbits` by rounding values to the nearest step within the specified bit depth.
3. **-law Expansion**: reverses the compression applied earlier to restore the signal to its original dynamic range:

   ```
   F 1(y) = ma.signum(y) * (pow(1 + mu, abs(y)) - 1) / mu;
   ```

Example test program

```
process = os.osc(440) : mulaw_bitcrusher(255, 8);
```

In this example, a sine wave at 440 Hz is passed through the -law bitcrusher, with a compression parameter `mu` of 255 and 8-bit quantization. This creates a distorted, “lo-fi” effect.

Test

```
ba = library("basics.lib");
os = library("oscillators.lib");
mulaw_bitcrusher_test = os.osc(440) : ba.mulaw_bitcrusher(2.0, 8);
```

References

- https://en.wikipedia.org/wiki/M-law_algorithm

Sliding Reduce

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

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

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

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

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

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

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

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

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

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

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

Now it becomes impossible to re-use any values. To fix that let's first look at how we'd implement it using recursiveMax, but with a fixed n that is not a power of 2. For example, this is how you'd do it with `n=3`:

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

```
n=6
```

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

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

```
n=7
```

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

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

Then it becomes a matter of:

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

In Faust, we can only do that for a fixed maximum number of values: `maxN`, known at compile time.

(ba.)slidingReduce

Fold-like high order function. Apply a commutative binary operation `op` to the last `n` consecutive samples of a signal `x`. For example : `slidingReduce(max,128,128,0-(ma.MAX))` will compute the maximum of the last 128 samples. The output is updated each sample, unlike reduce, where the output is constant for the duration of a block.

Usage

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

Where:

- `n`: the number of values to process
- `maxN`: the maximum number of values to process (int, known at compile time, maxN > 0)
- `op`: the operator. Needs to be a commutative one.
- `disabledVal`: the value to use when we want to ignore a value.

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

Test

```
ba = library("basics.lib");
os = library("oscillators.lib");
ma = library("maths.lib");
slidingReduce_test = os.osc(440) : ba.slidingReduce(max, 64, 64, 0 - ma.MAX);
```

(ba.)slidingSum

The sliding sum of the last n input samples.

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

Usage

```
_ : slidingSum(n) : _
```

Where:

- `n`: the number of values to process

Test

```
ba = library("basics.lib");
os = library("oscillators.lib");
slidingSum_test = os.osc(440) : ba.slidingSum(64);
```

(ba.)slidingSump

The sliding sum of the last n input samples.

It uses a lot more CPU than `ba.slidingSum`, but is numerically stable "forever" in return.

Usage

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

Where:

- `n`: the number of values to process
- `maxN`: the maximum number of values to process (int, known at compile time, maxN > 0)

Test

```
ba = library("basics.lib");
os = library("oscillators.lib");
slidingSump_test = os.osc(440) : ba.slidingSump(64, 128);
```

(ba.)slidingMax

The sliding maximum of the last n input samples.

Usage

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

Where:

- `n`: the number of values to process
- `maxN`: the maximum number of values to process (int, known at compile time, maxN > 0)

Test

```
ba = library("basics.lib");
os = library("oscillators.lib");
ma = library("maths.lib");
slidingMax_test = os.osc(440) : ba.slidingMax(64, 128);
```

(ba.)slidingMin

The sliding minimum of the last n input samples.

Usage

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

Where:

- `n`: the number of values to process
- `maxN`: the maximum number of values to process (int, known at compile time, maxN > 0)

Test

```
ba = library("basics.lib");
os = library("oscillators.lib");
ma = library("maths.lib");
slidingMin_test = os.osc(440) : ba.slidingMin(64, 128);
```

(ba.)slidingMean

The sliding mean of the last n input samples.

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

Usage

```
_ : slidingMean(n) : _
```

Where:

- `n`: the number of values to process

Test

```
ba = library("basics.lib");
os = library("oscillators.lib");
slidingMean_test = os.osc(440) : ba.slidingMean(64);
```

(ba.)slidingMeanp

The sliding mean of the last n input samples.

It uses a lot more CPU than `ba.slidingMean`, but is numerically stable "forever" in return.

Usage

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

Where:

- `n`: the number of values to process
- `maxN`: the maximum number of values to process (int, known at compile time, maxN > 0)

Test

```
ba = library("basics.lib");
os = library("oscillators.lib");
slidingMeanp_test = os.osc(440) : ba.slidingMeanp(64, 128);
```

(ba.)slidingRMS

The root mean square of the last n input samples.

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

Usage

```
_ : slidingRMS(n) : _
```

Where:

- `n`: the number of values to process

Test

```
ba = library("basics.lib");
os = library("oscillators.lib");
slidingRMS_test = os.osc(440) : ba.slidingRMS(64);
```

(ba.)slidingRMSp

The root mean square of the last n input samples.

It uses a lot more CPU than `ba.slidingRMS`, but is numerically stable "forever" in return.

Usage

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

Where:

- `n`: the number of values to process
- `maxN`: the maximum number of values to process (int, known at compile time, maxN > 0)

Test

```
ba = library("basics.lib");
os = library("oscillators.lib");
slidingRMSp_test = os.osc(440) : ba.slidingRMSp(64, 128);
```

Parallel Operators

Provides various operations on N parallel inputs using a high order `parallelOp(op,N,x)` function:

- `parallelMax(N)`: the max of n parallel inputs
- `parallelMin(N)`: the min of n parallel inputs
- `parallelMean(N)`: the mean of n parallel inputs
- `parallelRMS(N)`: the RMS of n parallel inputs

(ba.)parallelOp

Apply a commutative binary operation `op` to N parallel inputs.

usage

```
si.bus(N) : parallelOp(op,N) : _
```

where:

- `N`: the number of parallel inputs known at compile time
- `op`: the operator which needs to be commutative

Test

```
ba = library("basics.lib");
parallelOp_test = (0.2, 0.5, 0.1) : ba.parallelOp(max, 3);
```

`(ba.)parallelMax`

The maximum of N parallel inputs.

Usage

```
si.bus(N) : parallelMax(N) : _
```

Where:

- `N`: the number of parallel inputs known at compile time

Test

```
ba = library("basics.lib");
parallelMax_test = (0.2, 0.5, 0.1) : ba.parallelMax(3);
```

`(ba.)parallelMin`

The minimum of N parallel inputs.

Usage

```
si.bus(N) : parallelMin(N) : _
```

Where:

- `N`: the number of parallel inputs known at compile time

Test

```
ba = library("basics.lib");
parallelMin_test = (0.2, 0.5, 0.1) : ba.parallelMin(3);
```

(ba.)parallelMean

The mean of N parallel inputs.

Usage

```
si.bus(N) : parallelMean(N) : _
```

Where:

- `N`: the number of parallel inputs known at compile time

Test

```
ba = library("basics.lib");
parallelMean_test = (0.2, 0.5, 0.1) : ba.parallelMean(3);
```

(ba.)parallelRMS

The RMS of N parallel inputs.

Usage

```
si.bus(N) : parallelRMS(N) : _
```

Where:

- `N`: the number of parallel inputs known at compile time

Test

```
ba = library("basics.lib");
parallelRMS_test = (0.2, 0.5, 0.1) : ba.parallelRMS(3);
```

compressors.lib

Compressors library. Its official prefix is `co`.

This library provides building blocks and complete dynamic processors including compressors, limiters, expanders, and gates.

The Compressors library is organized into 6 sections:

- Conversion Tools
- Functions Reference
- Linear gain computer section
- Original versions section
- Expanders
- Lookahead Limiters

References

- https://github.com/grame-cncm/faustlibraries/blob/master/compressors.lib

Conversion Tools

Most compressors have a ratio parameter to define the amount of compression. A ratio of 1 means no compression, a ratio of 2 means that for every dB the input goes above the threshold, the output gets turned down half a dB. To use a compressor as a brick wall limiter, the ratio needs to be infinity. This is hard to express in a Faust UI element, and overcompression can not be expressed at all, therefore most compressors in this library use a strength parameter instead, where 0 means no compression, 1 means hard limiting and bigger than 1 means over-compression.

`(co.)ratio2strength`

This utility converts a ratio to a strength.

Usage

```
ratio2strength(ratio) : _
```

Where:

- `ratio`: compression ratio, between 1 and infinity (1=no compression, infinity means hard limiting)

Test

```
co = library("compressors.lib");
ratio2strength_test = co.ratio2strength(4);
```

`(co.)strength2ratio`

This utility converts a strength to a ratio.

Usage

```
strength2ratio(strength) : _
```

Where:

- `strength`: strength of the compression (0 = no compression, 1 means hard limiting, >1 means over-compression)

Test

```
co = library("compressors.lib");
strength2ratio_test = co.strength2ratio(0.75);
```

Functions Reference

(co.)peak_compression_gain_mono_db

Mono dynamic range compressor gain computer with dB output. `peak_compression_gain_mono_db` is a standard Faust function.

Usage

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

Where:

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

It uses a strength parameter instead of the traditional ratio, in order to be able to function as a hard limiter. For that you'd need a ratio of infinity:1, and you cannot express that in Faust.

Sometimes even bigger ratios are useful: for example a group recording where one instrument is recorded with both a close microphone and a room microphone, and the instrument is loud enough in the room mic when playing loud, but you want to boost it when it is playing soft.

Test

```
co = library("compressors.lib");
os = library("oscillators.lib");
peak_compression_gain_mono_db_test = os.osc(440) : co.peak_compression_gain_mono_db(0.5, -12
```

References

- http://en.wikipedia.org/wiki/Dynamic_range_compression
- Digital Dynamic Range Compressor Design: A Tutorial and Analysis, Dimitrios GIANNOULIS (Dimitrios.Giannoulis@eecs.qmul.ac.uk), Michael MASSBERG (michael@massberg.org), and Josuah D.REISS (josh.reiss@eecs.qmul.ac.uk)

(co.)peak_compression_gain_N_chan_db

N channels dynamic range compressor gain computer with dB output. `peak_compression_gain_N_chan_db` is a standard Faust function.

Usage

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

Where:

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

It uses a strength parameter instead of the traditional ratio, in order to be able to function as a hard limiter. For that you'd need a ratio of infinity:1, and you cannot express that in Faust.

Sometimes even bigger ratios are useful: for example a group recording where one instrument is recorded with both a close microphone and a room microphone, and the instrument is loud enough in the room mic when playing loud, but you want to boost it when it is playing soft.

Test

```
co = library("compressors.lib");
os = library("oscillators.lib");
peak_compression_gain_N_chan_db_test = (os.osc(440), os.osc(660)) : co.peak_compression_gain
```

References

- http://en.wikipedia.org/wiki/Dynamic_range_compression
- Digital Dynamic Range Compressor Design: A Tutorial and Analysis, Dimitrios GIANNOULIS (Dimitrios.Giannoulis@eecs.qmul.ac.uk), Michael MASSBERG (michael@massberg.org), and Josuah D.REISS (josh.reiss@eecs.qmul.ac.uk)

(co.)FFcompressor_N_chan

Feed forward N channels dynamic range compressor. `FFcompressor_N_chan` is a standard Faust function.

Usage

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

Where:

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

It uses a strength parameter instead of the traditional ratio, in order to be able to function as a hard limiter. For that you'd need a ratio of infinity:1, and you cannot express that in Faust.

Sometimes even bigger ratios are useful: for example a group recording where one instrument is recorded with both a close microphone and a room microphone, and the instrument is loud enough in the room mic when playing loud, but you want to boost it when it is playing soft.

Test

```
co = library("compressors.lib");
os = library("oscillators.lib");
meter(x) = x;
FFcompressor_N_chan_test = (os.osc(440), os.osc(660)) : co.FFcompressor_N_chan(0.5, -12, 0.0
```

References

- http://en.wikipedia.org/wiki/Dynamic_range_compression
- Digital Dynamic Range Compressor Design: A Tutorial and Analysis, Dimitrios GIANNOULIS (Dimitrios.Giannoulis@eecs.qmul.ac.uk), Michael MASSBERG (michael@massberg.org), and Josuah D.REISS (josh.reiss@eecs.qmul.ac.uk)

`(co.)FBcompressor_N_chan`

Feed back N channels dynamic range compressor. `FBcompressor_N_chan` is a standard Faust function.

Usage

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

Where:

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

- `meter`: a gain reduction meter. It can be implemented with: `meter = _ <: (_,(ba.linear2db:max(maxGR):meter_group((hbargraph("[1][unit:dB][tooltip: gain reduction in dB]", maxGR, 0))))):attach;` or it can be omitted by defining `meter = _;`.
- `N`: the number of channels of the compressor, known at compile time

It uses a strength parameter instead of the traditional ratio, in order to be able to function as a hard limiter. For that you'd need a ratio of infinity:1, and you cannot express that in Faust.

Sometimes even bigger ratios are useful: for example a group recording where one instrument is recorded with both a close microphone and a room microphone, and the instrument is loud enough in the room mic when playing loud, but you want to boost it when it is playing soft.

Test

```
co = library("compressors.lib");
os = library("oscillators.lib");
meter(x) = x;
FBcompressor_N_chan_test = (os.osc(440), os.osc(660)) : co.FBcompressor_N_chan(0.5, -12, 0.0
```

References

- http://en.wikipedia.org/wiki/Dynamic_range_compression
- Digital Dynamic Range Compressor Design: A Tutorial and Analysis, Dimitrios GIANNOULIS (Dimitrios.Giannoulis@eecs.qmul.ac.uk), Michael MASSBERG (michael@massberg.org), and Josuah D.REISS (josh.reiss@eecs.qmul.ac.uk)

`(co.)FBFFcompressor_N_chan`

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

Usage

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

Where:

- `strength`: strength of the compression (0 = no compression, 1 means hard limiting, >1 means over-compression)
- `thresh`: dB level threshold above which compression kicks in
- `att`: attack time = time constant (sec) when level & compression going up

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

It uses a strength parameter instead of the traditional ratio, in order to be able to function as a hard limiter. For that you'd need a ratio of infinity:1, and you cannot express that in Faust.

Sometimes even bigger ratios are useful: for example a group recording where one instrument is recorded with both a close microphone and a room microphone, and the instrument is loud enough in the room mic when playing loud, but you want to boost it when it is playing soft.

Test

```
co = library("compressors.lib");
os = library("oscillators.lib");
meter(x) = x;
FBFFcompressor_N_chan_test = (os.osc(440), os.osc(660)) : co.FBFFcompressor_N_chan(0.4, -12,
```

References

- http://en.wikipedia.org/wiki/Dynamic_range_compression
- Digital Dynamic Range Compressor Design: A Tutorial and Analysis, Dimitrios GIANNOULIS (Dimitrios.Giannoulis@eecs.qmul.ac.uk), Michael MASSBERG (michael@massberg.org), and Josuah D.REISS (josh.reiss@eecs.qmul.ac.uk)

`(co.)RMS_compression_gain_mono_db`

Mono RMS dynamic range compressor gain computer with dB output. `RMS_compression_gain_mono_db` is a standard Faust function.

Usage

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

Where:

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

It uses a strength parameter instead of the traditional ratio, in order to be able to function as a hard limiter. For that you'd need a ratio of infinity:1, and you cannot express that in Faust.

Sometimes even bigger ratios are useful: for example a group recording where one instrument is recorded with both a close microphone and a room microphone, and the instrument is loud enough in the room mic when playing loud, but you want to boost it when it is playing soft.

Test

```
co = library("compressors.lib");
os = library("oscillators.lib");
RMS_compression_gain_mono_db_test = os.osc(330) : co.RMS_compression_gain_mono_db(0.5, -18,
```

References

- http://en.wikipedia.org/wiki/Dynamic_range_compression
- Digital Dynamic Range Compressor Design: A Tutorial and Analysis, Dimitrios GIANNOULIS (Dimitrios.Giannoulis@eecs.qmul.ac.uk), Michael MASSBERG (michael@massberg.org), and Josuah D.REISS (josh.reiss@eecs.qmul.ac.uk)

(co.)RMS_compression_gain_N_chan_db

RMS N channels dynamic range compressor gain computer with dB output. `RMS_compression_gain_N_chan_db` is a standard Faust function.

Usage

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

Where:

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

It uses a strength parameter instead of the traditional ratio, in order to be able to function as a hard limiter. For that you'd need a ratio of infinity:1, and you cannot express that in Faust.

Sometimes even bigger ratios are useful: for example a group recording where one instrument is recorded with both a close microphone and a room microphone, and the instrument is loud enough in the room mic when playing loud, but you want to boost it when it is playing soft.

Test

```
co = library("compressors.lib");
os = library("oscillators.lib");
RMS_compression_gain_N_chan_db_test = (os.osc(330), os.osc(550)) : co.RMS_compression_gain_N
```

References

- http://en.wikipedia.org/wiki/Dynamic_range_compression
- Digital Dynamic Range Compressor Design: A Tutorial and Analysis, Dimitrios GIANNOULIS (Dimitrios.Giannoulis@eecs.qmul.ac.uk), Michael MASSBERG (michael@massberg.org), and Josuah D.REISS (josh.reiss@eecs.qmul.ac.uk)

(co.)RMS_FBFFcompressor_N_chan

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

Usage

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

Where:

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

It uses a strength parameter instead of the traditional ratio, in order to be able to function as a hard limiter. For that you'd need a ratio of infinity:1, and you cannot express that in Faust.

Sometimes even bigger ratios are useful: for example a group recording where one instrument is recorded with both a close microphone and a room microphone, and the instrument is loud enough in the room mic when playing loud, but you want to boost it when it is playing soft.

To save CPU we cheat a bit, in a similar way as in the original libs: instead of crosfading between two sets of gain calculators as above, we take the `abs` of the audio from both the FF and FB, and crossfade between those, and feed that into one set of gain calculators again the strength is much higher when in FB mode, but implemented differently.

Test

```
co = library("compressors.lib");
os = library("oscillators.lib");
meter(x) = x;
RMS_FBFFcompressor_N_chan_test = (os.osc(330), os.osc(550)) : co.RMS_FBFFcompressor_N_chan(0
```

References

- http://en.wikipedia.org/wiki/Dynamic_range_compression
- Digital Dynamic Range Compressor Design: A Tutorial and Analysis, Dimitrios GIANNOULIS (Dimitrios.Giannoulis@eecs.qmul.ac.uk), Michael MASSBERG (michael@massberg.org), and Josuah D.REISS (josh.reiss@eecs.qmul.ac.uk)

(co.)RMS_FBcompressor_peak_limiter_N_chan

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

Usage

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

Where:

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

- `meterLim`: brickwall limiter gain reduction meter. It can be implemented with: `meterLim = _<:(_,(max(maxGR):meter_group((hbargraph("[1][unit:dB][tooltip: gain reduction in dB]", maxGR, 0))))):attach;`
- `N`: the number of channels of the compressor, known at compile time

It uses a strength parameter instead of the traditional ratio, in order to be able to function as a hard limiter. For that you'd need a ratio of infinity:1, and you cannot express that in Faust.

Sometimes even bigger ratios are useful: for example a group recording where one instrument is recorded with both a close microphone and a room microphone, and the instrument is loud enough in the room mic when playing loud, but you want to boost it when it is playing soft.

Test

```
co = library("compressors.lib");
os = library("oscillators.lib");
meter(x) = x;
meterLim(x) = x;
RMS_FBcompressor_peak_limiter_N_chan_test = (os.osc(330), os.osc(550)) : co.RMS_FBcompressor
```

References

- http://en.wikipedia.org/wiki/Dynamic_range_compression
- Digital Dynamic Range Compressor Design: A Tutorial and Analysis, Dimitrios GIANNOULIS (Dimitrios.Giannoulis@eecs.qmul.ac.uk), Michael MASSBERG (michael@massberg.org), and Josuah D.REISS (josh.reiss@eecs.qmul.ac.uk)

Linear gain computer section

The gain computer functions in this section have been replaced by a version that outputs dBs, but we retain the linear output version for backward compatibility.

`(co.)peak_compression_gain_mono`

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

Usage

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

Where:

- `strength`: strength of the compression (0 = no compression, 1 means hard limiting, >1 means over-compression)

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

It uses a strength parameter instead of the traditional ratio, in order to be able to function as a hard limiter. For that you'd need a ratio of infinity:1, and you cannot express that in Faust.

Sometimes even bigger ratios are useful: for example a group recording where one instrument is recorded with both a close microphone and a room microphone, and the instrument is loud enough in the room mic when playing loud, but you want to boost it when it is playing soft.

Test

```
co = library("compressors.lib");
os = library("oscillators.lib");
peak_compression_gain_mono_test = os.osc(440) : co.peak_compression_gain_mono(0.5, -12, 0.0
```

References

- http://en.wikipedia.org/wiki/Dynamic_range_compression
- Digital Dynamic Range Compressor Design: A Tutorial and Analysis, Dimitrios GIANNOULIS (Dimitrios.Giannoulis@eecs.qmul.ac.uk), Michael MASSBERG (michael@massberg.org), and Josuah D.REISS (josh.reiss@eecs.qmul.ac.uk)

`(co.)peak_compression_gain_N_chan`

N channels dynamic range compressor gain computer with linear output. `peak_compression_gain_N_chan` is a standard Faust function.

Usage

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

Where:

- `strength`: strength of the compression (0 = no compression, 1 means hard limiting, >1 means over-compression)

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

It uses a strength parameter instead of the traditional ratio, in order to be able to function as a hard limiter. For that you'd need a ratio of infinity:1, and you cannot express that in Faust.

Sometimes even bigger ratios are useful: for example a group recording where one instrument is recorded with both a close microphone and a room microphone, and the instrument is loud enough in the room mic when playing loud, but you want to boost it when it is playing soft.

Test

```
co = library("compressors.lib");
os = library("oscillators.lib");
peak_compression_gain_N_chan_test = (os.osc(440), os.osc(660)) : co.peak_compression_gain_N_
```

References

- http://en.wikipedia.org/wiki/Dynamic_range_compression
- Digital Dynamic Range Compressor Design: A Tutorial and Analysis, Dimitrios GIANNOULIS (Dimitrios.Giannoulis@eecs.qmul.ac.uk), Michael MASSBERG (michael@massberg.org), and Josuah D.REISS (josh.reiss@eecs.qmul.ac.uk)

`(co.)RMS_compression_gain_mono`

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

Usage

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

Where:

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

It uses a strength parameter instead of the traditional ratio, in order to be able to function as a hard limiter. For that you'd need a ratio of infinity:1, and you cannot express that in Faust.

Sometimes even bigger ratios are useful: for example a group recording where one instrument is recorded with both a close microphone and a room microphone, and the instrument is loud enough in the room mic when playing loud, but you want to boost it when it is playing soft.

Test

```
co = library("compressors.lib");
os = library("oscillators.lib");
RMS_compression_gain_mono_test = os.osc(330) : co.RMS_compression_gain_mono(0.5, -18, 0.02,
```

References

- http://en.wikipedia.org/wiki/Dynamic_range_compression
- Digital Dynamic Range Compressor Design: A Tutorial and Analysis, Dimitrios GIANNOULIS (Dimitrios.Giannoulis@eecs.qmul.ac.uk), Michael MASSBERG (michael@massberg.org), and Josuah D.REISS (josh.reiss@eecs.qmul.ac.uk)

(co.)RMS_compression_gain_N_chan

RMS N channels dynamic range compressor gain computer with linear output. `RMS_compression_gain_N_chan` is a standard Faust function.

Usage

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

Where:

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

It uses a strength parameter instead of the traditional ratio, in order to be able to function as a hard limiter. For that you'd need a ratio of infinity:1, and you cannot express that in Faust.

Sometimes even bigger ratios are useful: for example a group recording where one instrument is recorded with both a close microphone and a room microphone, and the instrument is loud enough in the room mic when playing loud, but you want to boost it when it is playing soft.

Test

```
co = library("compressors.lib");
os = library("oscillators.lib");
RMS_compression_gain_N_chan_test = (os.osc(330), os.osc(550)) : co.RMS_compression_gain_N_ch
```

References

- http://en.wikipedia.org/wiki/Dynamic_range_compression
- Digital Dynamic Range Compressor Design: A Tutorial and Analysis, Dimitrios GIANNOULIS (Dimitrios.Giannoulis@eecs.qmul.ac.uk), Michael MASSBERG (michael@massberg.org), and Josuah D.REISS (josh.reiss@eecs.qmul.ac.uk)

Original versions section

The functions in this section are largely superseded by the limiters above, but we retain them for backward compatibility and for situations in which a more permissive, MIT-style license is required.

(co.)compressor_lad_mono

Mono dynamic range compressor with lookahead delay. `compressor_lad_mono` is a standard Faust function.

Usage

```
_ : compressor_lad_mono(lad,ratio,thresh,att,rel) : _
```

Where:

- `lad`: lookahead delay in seconds (nonnegative) - gets rounded to nearest sample. The effective attack time is a good setting
- `ratio`: compression ratio (1 = no compression, >1 means compression) Ratios: 4 is moderate compression, 8 is strong compression, 12 is mild limiting, and 20 is pretty hard limiting at the threshold
- `thresh`: dB level threshold above which compression kicks in (0 dB = max level)
- `att`: attack time = time constant (sec) when level & compression are going up
- `rel`: release time = time constant (sec) coming out of compression

Test

```
co = library("compressors.lib");
os = library("oscillators.lib");
compressor_lad_mono_test = os.osc(440) : co.compressor_lad_mono(0.005, 4, -9, 0.01, 0.1);
```

References

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

---

(co.)compressor_mono

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

Usage

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

Where:

- `ratio`: compression ratio (1 = no compression, >1 means compression) Ratios: 4 is moderate compression, 8 is strong compression, 12 is mild limiting, and 20 is pretty hard limiting at the threshold
- `thresh`: dB level threshold above which compression kicks in (0 dB = max level)
- `att`: attack time = time constant (sec) when level & compression are going up
- `rel`: release time = time constant (sec) coming out of compression

Test

```
co = library("compressors.lib");
os = library("oscillators.lib");
compressor_mono_test = os.osc(440) : co.compressor_mono(4, -9, 0.01, 0.2);
```

References

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

---

`(co.)compressor_stereo`

Stereo dynamic range compressors.

Usage

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

Where:

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

Test

```
co = library("compressors.lib");
os = library("oscillators.lib");
compressor_stereo_test = (os.osc(440), os.osc(660)) : co.compressor_stereo(4, -9, 0.01, 0.2)
```

References

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

---

(co.)compression_gain_mono

Compression-gain calculation for dynamic range compressors.

Usage

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

Where:

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

Test

```
co = library("compressors.lib");
os = library("oscillators.lib");
compression_gain_mono_test = os.osc(440) : co.compression_gain_mono(4, -9, 0.01, 0.2);
```

References

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

---

(co.)limiter_1176_R4_mono

A limiter guards against hard-clipping. It can be implemented as a compressor having a high threshold (near the clipping level), fast attack, and high ratio. Since the compression ratio is so high, some knee smoothing is desirable (for softer limiting). This example is intended to get you started using compressors as limiters, so all parameters are hardwired here to nominal values.

`ratio`: 4 (moderate compression). See `compressor_mono` comments for a guide to other choices. Mike Shipley likes this (lowest) setting on the 1176. (Grammy award-winning mixer for Queen, Tom Petty, etc.).

`thresh`: -6 dB, meaning 4:1 compression begins at amplitude 1/2.

`att`: 800 MICROseconds (Note: scaled by ratio in the 1176) The 1176 range is said to be 20-800 microseconds. Faster attack gives "more bite" (e.g. on vocals), and makes hard-clipping less likely on fast overloads.

`rel`: 0.5 s (Note: scaled by ratio in the 1176) The 1176 range is said to be 50-1100 ms.

The 1176 also has a "bright, clear eq effect" (use filters.lib if desired). `limiter_1176_R4_mono` is a standard Faust function.

Usage

```
_ : limiter_1176_R4_mono : _
```

Test

```
co = library("compressors.lib");
os = library("oscillators.lib");
limiter_1176_R4_mono_test = os.osc(440) : co.limiter_1176_R4_mono;
```

References

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

`(co.)limiter_1176_R4_stereo`

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

`ratio`: 4 (moderate compression), 8 (severe compression), 12 (mild limiting), or 20 to 1 (hard limiting).

`att`: 20-800 MICROseconds (Note: scaled by ratio in the 1176).

`rel`: 50-1100 ms (Note: scaled by ratio in the 1176).

Mike Shipley likes 4:1 (Grammy-winning mixer for Queen, Tom Petty, etc.) Faster attack gives "more bite" (e.g. on vocals). He hears a bright, clear eq effect as well (not implemented here).

Usage

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

Test

```
co = library("compressors.lib");
os = library("oscillators.lib");
limiter_1176_R4_stereo_test = (os.osc(440), os.osc(660)) : co.limiter_1176_R4_stereo;
```

References

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

Expanders

(co.)peak_expansion_gain_N_chan_db

N channels dynamic range expander gain computer. `peak_expansion_gain_N_chan_db` is a standard Faust function.

Usage

```
si.bus(N) : peak_expansion_gain_N_chan_db(strength,thresh,range,att,hold,rel,knee,prePost,li
```

Where:

- `strength`: strength of the expansion (0 = no expansion, 100 means gating, <1 means upward compression)
- `thresh`: dB level threshold below which expansion kicks in
- `range`: maximum amount of expansion in dB
- `att`: attack time = time constant (sec) coming out of expansion
- `hold` : hold time (sec)
- `rel`: release time = time constant (sec) going into expansion
- `knee`: a gradual increase in gain reduction around the threshold: above thresh+(knee/2) there is no gain reduction, below thresh-(knee/2) there is the same gain reduction as without a knee, and in between there is a gradual increase in gain reduction
- `prePost`: places the level detector either at the input or after the gain computer; this turns it from a linear return-to-zero detector into a log domain return-to-range detector
- `link`: the amount of linkage between the channels: 0 = each channel is independent, 1 = all channels have the same amount of gain reduction
- `maxHold`: the maximum hold time in samples, known at compile time
- `N`: the number of channels of the gain computer, known at compile time

Test

```
co = library("compressors.lib");
os = library("oscillators.lib");
peak_expansion_gain_N_chan_db_test = (os.osc(220), os.osc(330)) : co.peak_expansion_gain_N_c
```

(co.)expander_N_chan

Feed forward N channels dynamic range expander. `expander_N_chan` is a standard Faust function.

Usage

```
si.bus(N) : expander_N_chan(strength,thresh,range,att,hold,rel,knee,prePost,link,meter,maxHo
```

Where:

- `strength`: strength of the expansion (0 = no expansion, 100 means gating, <1 means upward compression)
- `thresh`: dB level threshold below which expansion kicks in
- `range`: maximum amount of expansion in dB
- `att`: attack time = time constant (sec) coming out of expansion
- `hold` : hold time
- `rel`: release time = time constant (sec) going into expansion
- `knee`: a gradual increase in gain reduction around the threshold: above thresh+(knee/2) there is no gain reduction, below thresh-(knee/2) there is the same gain reduction as without a knee, and in between there is a gradual increase in gain reduction
- `prePost`: places the level detector either at the input or after the gain computer; this turns it from a linear return-to-zero detector into a log domain return-to-range detector
- `link`: the amount of linkage between the channels: 0 = each channel is independent, 1 = all channels have the same amount of gain reduction
- `meter`: a gain reduction meter. It can be implemented like so: `meter = _<:(_, (ba.linear2db:max(maxGR):meter_group((hbargraph("[1][unit:dB][tooltip: gain reduction in dB]", maxGR, 0))))):attach;`
- `maxHold`: the maximum hold time in samples, known at compile time
- `N`: the number of channels of the expander, known at compile time

Test

```
co = library("compressors.lib");
os = library("oscillators.lib");
meter(x) = x;
expander_N_chan_test = (os.osc(220), os.osc(330)) : co.expander_N_chan(0.5, -40, 20, 0.05, 0
```

(co.)expanderSC_N_chan

Feed forward N channels dynamic range expander with sidechain. `expanderSC_N_chan` is a standard Faust function.

Usage

```
si.bus(N) : expanderSC_N_chan(strength,thresh,range,att,hold,rel,knee,prePost,link,meter,max
```

Where:

- `strength`: strength of the expansion (0 = no expansion, 100 means gating, <1 means upward compression)
- `thresh`: dB level threshold below which expansion kicks in
- `range`: maximum amount of expansion in dB
- `att`: attack time = time constant (sec) coming out of expansion
- `hold` : hold time
- `rel`: release time = time constant (sec) going into expansion
- `knee`: a gradual increase in gain reduction around the threshold: above thresh+(knee/2) there is no gain reduction, below thresh-(knee/2) there is the same gain reduction as without a knee, and in between there is a gradual increase in gain reduction
- `prePost`: places the level detector either at the input or after the gain computer; this turns it from a linear return-to-zero detector into a log domain return-to-range detector
- `link`: the amount of linkage between the channels: 0 = each channel is independent, 1 = all channels have the same amount of gain reduction
- `meter`: a gain reduction meter. It can be implemented like so: `meter = _<:(_, (ba.linear2db:max(maxGR):meter_group((hbargraph("[1][unit:dB][tooltip: gain reduction in dB]", maxGR, 0))))):attach;`
- `maxHold`: the maximum hold time in samples, known at compile time
- `N`: the number of channels of the expander, known at compile time
- `SCfunction` : a function that get's placed before the level-detector, needs to have a single input and output
- `SCswitch` : use either the regular audio input or the SCsignal as the input for the level detector
- `SCsignal` : an audio signal, to be used as the input for the level detector when SCswitch is 1

Test

```
co = library("compressors.lib");
os = library("oscillators.lib");
meter(x) = x;
SCfunction(x) = x;
expanderSC_N_chan_test = (os.osc(220), os.osc(330)) : co.expanderSC_N_chan(0.5, -40, 20, 0.0
```

Lookahead Limiters

(co.)limiter_lad_N

N-channels lookahead limiter inspired by IOhannes Zmölnig's post, which is in turn based on the thesis by Peter Falkner "Entwicklung eines digitalen Stereo-Limiters mit Hilfe des Signalprozessors DSP56001". This version of the limiter uses a peak-holder with smoothed attack and release based on tau time constant filters.

It is also possible to use a time constant that is `2PI*tau` by dividing the attack and release times by `2PI`. This time constant allows for the amplitude profile to reach `1 - e^(-2PI)` of the final peak after the attack time. The input path can be delayed by the same amount as the attack time to synchronise input and amplitude profile, realising a system that is particularly effective as a colourless (ideally) brickwall limiter.

Note that the effectiveness of the ceiling settings are dependent on the other parameters, especially the time constant used for the smoothing filters and the lookahead delay.

Similarly, the colourless characteristics are also dependent on attack, hold, and release times. Since fluctuations above ~15 Hz are perceived as timbral effects, [Vassilakis and Kendall 2010] it is reasonable to set the attack time to 1/15 seconds for a smooth amplitude modulation. On the other hand, the hold time can be set to the peak-to-peak period of the expected lowest frequency in the signal, which allows for minimal distortion of the low frequencies. The release time can then provide a perceptually linear and gradual gain increase determined by the user for any specific application.

The scaling factor for all the channels is determined by the loudest peak between them all, so that amplitude ratios between the signals are kept.

Usage

```
si.bus(N) : limiter_lad_N(N, LD, ceiling, attack, hold, release) : si.bus(N)
```

Where:

- `N`: is the number of channels, known at compile-time
- `LD`: is the lookahead delay in seconds, known at compile-time
- `ceiling`: is the linear amplitude output limit
- `attack`: is the attack time in seconds
- `hold`: is the hold time in seconds
- `release`: is the release time in seconds

Example for a stereo limiter: `limiter_lad_N(2, .01, 1, .01, .1, 1);`

Test

```
co = library("compressors.lib");
os = library("oscillators.lib");
limiter_lad_N_test = (os.osc(440), os.osc(660)) : co.limiter_lad_N(2, 0.01, 1, 0.01, 0.05, 0
```

References

- http://iem.at/~zmoelnig/publications/limiter

(co.)limiter_lad_mono

Specialised case of `limiter_lad_N` mono limiter.

Usage

```
_ : limiter_lad_mono(LD, ceiling, attack, hold, release) : _
```

Where:

- `LD`: is the lookahead delay in seconds, known at compile-time
- `ceiling`: is the linear amplitude output limit
- `attack`: is the attack time in seconds
- `hold`: is the hold time in seconds
- `release`: is the release time in seconds

Test

```
co = library("compressors.lib");
os = library("oscillators.lib");
limiter_lad_mono_test = os.osc(440) : co.limiter_lad_mono(0.01, 1, 0.01, 0.05, 0.2);
```

References

- http://iem.at/~zmoelnig/publications/limiter

(co.)limiter_lad_stereo

Specialised case of `limiter_lad_N` stereo limiter.

Usage

```
_,_ : limiter_lad_stereo(LD, ceiling, attack, hold, release) : _,_
```

Where:

- `LD`: is the lookahead delay in seconds, known at compile-time

- `ceiling`: is the linear amplitude output limit
- `attack`: is the attack time in seconds
- `hold`: is the hold time in seconds
- `release`: is the release time in seconds

Test

```
co = library("compressors.lib");
os = library("oscillators.lib");
limiter_lad_stereo_test = (os.osc(440), os.osc(660)) : co.limiter_lad_stereo(0.01, 1, 0.01,
```

References

- http://iem.at/~zmoelnig/publications/limiter

(co.)limiter_lad_quad

Specialised case of `limiter_lad_N` quadraphonic limiter.

Usage

```
si.bus(4) : limiter_lad_quad(LD, ceiling, attack, hold, release) : si.bus(4)
```

Where:

- `LD`: is the lookahead delay in seconds, known at compile-time
- `ceiling`: is the linear amplitude output limit
- `attack`: is the attack time in seconds
- `hold`: is the hold time in seconds
- `release`: is the release time in seconds

Test

```
co = library("compressors.lib");
os = library("oscillators.lib");
limiter_lad_quad_test = (os.osc(220), os.osc(330), os.osc(440), os.osc(550)) : co.limiter_la
```

References

- http://iem.at/~zmoelnig/publications/limiter

(co.)limiter_lad_bw

Specialised case of `limiter_lad_N` and ready-to-use unit-amplitude mono limiting function. This implementation, in particular, uses `2PI*tau` time constant filters for attack and release smoothing with synchronised input and gain signals.

This function's best application is to be used as a brickwall limiter with the least colouring artefacts while keeping a not-so-slow release curve. Tests have shown that, given a pop song with 60 dB of amplification and a 0-dB-ceiling, the loudest peak recorded was ~0.38 dB.

Usage

```
_ : limiter_lad_bw : _
```

Test

```
co = library("compressors.lib");
os = library("oscillators.lib");
limiter_lad_bw_test = os.osc(440) : co.limiter_lad_bw;
```

References

- http://iem.at/~zmoelnig/publications/limiter

delays.lib

Delays library. Its official prefix is `de`.

This library provides reusable building blocks for delay-based processing: single and multi-tap delays, fractional delays and utilities for echo and spatial effects.

The Delays library is organized into 4 sections:

- Basic Delay Functions
- Lagrange Interpolation
- Thiran Allpass Interpolation
- Others

References

- https://github.com/grame-cncm/faustlibraries/blob/master/delays.lib

Basic Delay Functions

`(de.)delay`

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

Usage

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

Where:

- `n`: the max delay length in samples
- `d`: the delay length in samples (integer) #### Test

```
de = library("delays.lib");
os = library("oscillators.lib");
delay_test = os.osc(440) : de.delay(44100, 22050);
```

TODO: add MBH np2

(de.)fdelay

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

`fdelay` is a standard Faust function.

Usage

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

Where:

- `n`: the max delay length in samples
- `d`: the delay length in samples (float)

Test

```
de = library("delays.lib");
os = library("oscillators.lib");
fdelay_test = os.osc(440) : de.fdelay(44100, 22050.5);
```

(de.)sdelay

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

Usage

```
_ : sdelay(n,it,d) : _
```

Where:

- `n`: the max delay length in samples

- `it`: interpolation time (in samples), for example 1024
- `d`: the delay length in samples (float)

Test

```
de = library("delays.lib");
os = library("oscillators.lib");
sdelay_test = os.osc(440) : de.sdelay(44100, 1024, 22050.5);
```

`(de.)prime_power_delays`

Prime Power Delay Line Lengths.

Usage

```
si.bus(N) : prime_power_delays(N,pathmin,pathmax) : si.bus(N);
```

Where:

- `N`: positive integer up to 16 (for higher powers of 2, extend 'primes' array below)
- `pathmin`: minimum acoustic ray length in the reverberator (in meters)
- `pathmax`: maximum acoustic ray length (meters) - think "room size"

Test

```
de = library("delays.lib");
prime_power_delays_test = de.prime_power_delays(4, 1, 10);
```

References

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

Lagrange Interpolation

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

Fractional delay line using Lagrange interpolation.

Usage

```
_ : fdelaylt[i|v](N, n, d) : _
```

Where:

- `N=1,2,3,...` is the order of the Lagrange interpolation polynomial (constant numerical expression)

- `n`: the max delay length in samples
- `d`: the delay length in samples

`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`.

Test

```
de = library("delays.lib");
os = library("oscillators.lib");
fdelaylti_test = os.osc(440) : de.fdelaylti(3, 44100, 22050.5);
fdelayltv_test = os.osc(440) : de.fdelayltv(3, 44100, 22050.5);
```

References

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

`(de.)fdelay[N]`

For convenience, `fdelay1`, `fdelay2`, `fdelay3`, `fdelay4`, `fdelay5` are also available where `N` is the order of the interpolation, built using `fdelayltv`.

Thiran Allpass Interpolation

Thiran Allpass Interpolation.

References

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

---

`(de.)fdelay[N]a`

Delay lines interpolated using Thiran allpass interpolation.

Usage

`_ : fdelay[N]a(n, d) : _`

(exactly like `fdelay`)

Where:

- `N=1,2,3, or 4` is the order of the Thiran interpolation filter (constant numerical expression), and the delay argument is at least `N-1/2`. First-order: `d` at least 0.5, second-order: `d` at least 1.5, third-order: `d` at least 2.5, fourth-order: `d` at least 3.5.
- `n`: the max delay length in samples
- `d`: the delay length in samples

Test

```
de = library("delays.lib");
os = library("oscillators.lib");
fdelay2a_test = os.osc(440) : de.fdelay2a(44100, 22050.5);
```

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.

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).

Others

`(de.)multiTapSincDelay`

Variable delay line using multi-tap sinc interpolation.

This function implements a continuously variable delay line by superposing (2K+2) auxiliary delayed signals whose positions and gains are determined by a sinc-based interpolation method. It extends the traditional crossfade delay technique to significantly reduce spectral coloration artifacts, which are problematic in applications like Wave Field Synthesis (WFS) and auralization.

Operation:

- If tau1 and tau2 are very close (|tau2 - tau1| 0), a simple fixed fractional delay is applied

- Otherwise, a variable delay is synthesized by:
  - Computing (2K+2) taps symmetrically distributed around tau1 and tau2
  - Applying sinc-based weighting to each tap, based on its offset from the target interpolated delay tau
  - Summing all the weighted taps to produce the output

Features:

- Smooth delay variation without introducing Doppler pitch shifts
- Significant reduction of comb-filter coloration compared to classical crossfading
- Switching between fixed and variable delay modes to ensure stability

Usage

```
_ : multiTapSincDelay(K, MaxDelay, tau1, tau2, alpha) : _
```

Where:

- `K (integer)`: number of auxiliary tap pairs (a constant numerical expression). Total number of taps = 2*K + 2
- `MaxDelay`: maximum allowable delay in samples (buffer size)
- `tau1`: initial delay in samples (can be fractional)
- `tau2`: target delay in samples (can be fractional)
- `alpha`: interpolation factor between tau1 and tau2 (in [0,1] with 0 = tau1, 1 = tau2)

Test

```
de = library("delays.lib");
os = library("oscillators.lib");
multiTapSincDelay_test = os.osc(440) : de.multiTapSincDelay(2, 4096, 1024.0, 1536.0, 0.5);
```

References

- T. Carpentier, "Implementation of a continuously variable delay line by crossfading between several tap delays", 2024: https://hal.science/hal-04646939

demos.lib

Demos library. Its official prefix is `dm`.

This library provides a collection of example DSP algorithms and demonstrations used to illustrate Faust features, syntax, and best practices. It includes simple oscillators, filters, effects, and synthesis examples useful for learning and testing.

The Demos library is organized into 5 sections:

- Analyzers
- Filters
- Effects
- Reverbs
- Generators

References

- https://github.com/grame-cncm/faustlibraries/blob/master/demos.lib

Analyzers

`(dm.)mth_octave_spectral_level_demo`

Demonstrate mth_octave_spectral_level in a standalone GUI.

Usage

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

Test

```
dm = library("demos.lib");
no = library("noises.lib");
mth_octave_spectral_level_demo_test = no.noise : dm.mth_octave_spectral_level_demo(1.5);
spectral_level_demo_test = no.noise : dm.spectral_level_demo;
```

Filters

`(dm.)parametric_eq_demo`

A parametric equalizer application.

Usage:

```
_ : parametric_eq_demo : _
```

Test

```
dm = library("demos.lib");
no = library("noises.lib");
parametric_eq_demo_test = no.noise : dm.parametric_eq_demo;
```

(dm.)spectral_tilt_demo

A spectral tilt application.

Usage

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

Where:

- `N`: filter order (integer)

Test

```
dm = library("demos.lib");
no = library("noises.lib");
spectral_tilt_demo_test = no.noise : dm.spectral_tilt_demo(4);
```

All other parameters interactive

(dm.)mth_octave_filterbank_demo and (dm.)filterbank_demo

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

Usage

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

Where:

- `M`: number of bands per octave

Test

```
dm = library("demos.lib");
no = library("noises.lib");
mth_octave_filterbank_demo_test = no.noise : dm.mth_octave_filterbank_demo(1);
filterbank_demo_test = no.noise : dm.filterbank_demo;
```

Effects

(dm.)cubicnl_demo

Distortion demo application.

Usage:

```
_ : cubicnl_demo : _
```

Test

```
dm = library("demos.lib");
no = library("noises.lib");
cubicnl_demo_test = no.noise : dm.cubicnl_demo;
```

(dm.)gate_demo

Gate demo application.

Usage

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

Test

```
dm = library("demos.lib");
no = library("noises.lib");
gate_demo_test = no.noise, no.noise : dm.gate_demo;
```

(dm.)compressor_demo

Compressor demo application.

Usage

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

Test

```
dm = library("demos.lib");
no = library("noises.lib");
compressor_demo_test = no.noise, no.noise : dm.compressor_demo;
```

(dm.)moog_vcf_demo

Illustrate and compare all three Moog VCF implementations above.

Usage

```
_ : moog_vcf_demo : _
```

Test

```
dm = library("demos.lib");
os = library("oscillators.lib");
moog_vcf_demo_test = os.osc(440) : dm.moog_vcf_demo;
```

(dm.)wah4_demo

Wah pedal application.

Usage

```
_ : wah4_demo : _
```

Test

```
dm = library("demos.lib");
os = library("oscillators.lib");
wah4_demo_test = os.osc(440) : dm.wah4_demo;
```

(dm.)crybaby_demo

Crybaby effect application.

Usage

```
_ : crybaby_demo : _
```

Test

```
dm = library("demos.lib");
os = library("oscillators.lib");
crybaby_demo_test = os.osc(440) : dm.crybaby_demo;
```

(dm.)flanger_demo

Flanger effect application.

Usage

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

Test

```
dm = library("demos.lib");
os = library("oscillators.lib");
flanger_demo_test = os.osc(440), os.osc(442) : dm.flanger_demo;
```

(dm.)phaser2_demo

Phaser effect demo application.

Usage

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

Test

```
dm = library("demos.lib");
os = library("oscillators.lib");
phaser2_demo_test = os.osc(440), os.osc(442) : dm.phaser2_demo;
```

(dm.)tapeStop_demo

Stereo tape-stop effect.

Usage

```
_,_ : tapeStop_demo : _,_
```

Test

```
dm = library("demos.lib");
os = library("oscillators.lib");
tapeStop_demo_test = os.osc(440), os.osc(442) : dm.tapeStop_demo;
```

Reverbs

(dm.)freeverb_demo

Freeverb demo application.

Usage

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

Test

```
dm = library("demos.lib");
os = library("oscillators.lib");
freeverb_demo_test = os.osc(440), os.osc(442) : dm.freeverb_demo;
```

(dm.)stereo_reverb_tester

Handy test inputs for reverberator demos below.

Usage

```
_,_ : stereo_reverb_tester(gui_group) : _,_
```

For suppressing the `gui_group` input, pass it as `!`. (See `(dm.)fdnrev0_demo` for an example of its use).

Test

```
dm = library("demos.lib");
no = library("noises.lib");
stereo_reverb_tester_test = no.noise, no.noise : dm.stereo_reverb_tester(!);
```

(dm.)fdnrev0_demo

A reverb application using `fdnrev0`.

Usage

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

Where:

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

Test

```
dm = library("demos.lib");
no = library("noises.lib");
fdnrev0_demo_test = no.noise, no.noise : dm.fdnrev0_demo(16, 5, 3);
```

(dm.)zita_rev_fdn_demo

Reverb demo application based on `zita_rev_fdn`.

Usage

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

Test

```
dm = library("demos.lib");
os = library("oscillators.lib");
zita_rev_fdn_demo_test = par(i, 8, os.osc(440 + i)) : dm.zita_rev_fdn_demo;
```

(dm.)zita_light

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

Usage

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

Test

```
dm = library("demos.lib");
os = library("oscillators.lib");
zita_light_test = os.osc(440), os.osc(442) : dm.zita_light;
```

(dm.)zita_rev1

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

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

Usage

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

Test

```
dm = library("demos.lib");
os = library("oscillators.lib");
zita_rev1_test = os.osc(440), os.osc(442) : dm.zita_rev1;
```

References

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

---

(dm.)vital_rev_demo

Example GUI for `vital_rev` with all parameters exposed.

Usage

```
_,_ : vital_rev_demo : _,_
```

Test

```
dm = library("demos.lib");
os = library("oscillators.lib");
vital_rev_demo_test = os.osc(440), os.osc(442) : dm.vital_rev_demo;
```

(dm.)reverbTank_demo

This is a stereo reverb following the "ReverbTank" example in [1], although some parameter ranges and scaling have been adjusted. It is an unofficial version of the Spin Semiconductor® Reverb. Other relevant instructional material can be found in [2-4].

Usage

```
_,_ : reverbTank_demo : _,_
```

Test

```
dm = library("demos.lib");
os = library("oscillators.lib");
reverbTank_demo_test = os.osc(440), os.osc(442) : dm.reverbTank_demo;
```

References

- [1] Pirkle, W. C. (2019). Designing audio effect plugins in C++ (2nd ed.). Chapter 17.14.

- [2] Spin Semiconductor. (n.d.). Reverberation. Retrieved 2024-04-16, from http://www.spinsemi.com/knowledge_base/effects.html#Reverberation
- [3] Zölzer, U. (2022). Digital audio signal processing (3rd ed.). Chapter 7, Figure 7.39.
- [4] Valhalla DSP. (2010, August 25). RIP Keith Barr. Retrieved 2024-04-16, from https://valhalladsp.com/2010/08/25/rip-keith-barr/

(dm.)kb_rom_rev1_demo

Keith Barr reverb effect rom_rev1 demo application.

Usage

```
_,_ : kb_rom_rev1_demo : _,_
```

Test

```
dm = library("demos.lib");
os = library("oscillators.lib");
kb_rom_rev1_demo_test = os.osc(440), os.osc(442) : dm.kb_rom_rev1_demo;
```

(dm.)dattorro_rev_demo

Example GUI for `dattorro_rev` with all parameters exposed and additional dry/wet and output gain control.

Usage

```
_,_ : dattorro_rev_demo : _,_
```

Test

```
dm = library("demos.lib");
os = library("oscillators.lib");
dattorro_rev_demo_test = os.osc(440), os.osc(442) : dm.dattorro_rev_demo;
```

(dm.)jprev_demo

Example GUI for `jprev` with all parameters exposed.

Usage

```
_,_ : jprev_demo : _,_
```

Test

```
dm = library("demos.lib");
os = library("oscillators.lib");
jprev_demo_test = os.osc(440), os.osc(442) : dm.jprev_demo;
```

(dm.)greyhole_demo

Example GUI for `greyhole` with all parameters exposed.

Usage

```
_,_ : greyhole_demo : _,_
```

Test

```
dm = library("demos.lib");
os = library("oscillators.lib");
greyhole_demo_test = os.osc(440), os.osc(442) : dm.greyhole_demo;
```

Generators

(dm.)sawtooth_demo

An application demonstrating the different sawtooth oscillators of Faust.

Usage

```
sawtooth_demo : _
```

Test

```
dm = library("demos.lib");
sawtooth_demo_test = dm.sawtooth_demo;
```

(dm.)virtual_analog_oscillator_demo

Virtual analog oscillator demo application.

Usage

```
virtual_analog_oscillator_demo : _
```

Test

```
dm = library("demos.lib");
virtual_analog_oscillator_demo_test = dm.virtual_analog_oscillator_demo;
```

(dm.)oscrs_demo

Simple application demoing filter based oscillators.

Usage

```
oscrs_demo : _
```

Test

```
dm = library("demos.lib");
oscrs_demo_test = dm.oscrs_demo;
```

(dm.)velvet_noise_demo

Listen to velvet_noise!

Usage

```
velvet_noise_demo : _
```

Test

```
dm = library("demos.lib");
velvet_noise_demo_test = dm.velvet_noise_demo;
```

(dm.)latch_demo

Illustrate latch operation.

Usage

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

Test

```
dm = library("demos.lib");
latch_demo_test = dm.latch_demo;
```

(dm.)envelopes_demo

Illustrate various envelopes overlaid, including their gate * 1.1.

Usage

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

Test

```
dm = library("demos.lib");
envelopes_demo_test = dm.envelopes_demo;
```

(dm.)fft_spectral_level_demo

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

Usage

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

Test

```
dm = library("demos.lib");
fft_spectral_level_demo_test = dm.fft_spectral_level_demo(256);
```

(dm.)reverse_echo_demo(nChans)

Multichannel echo effect with reverse delays.

Usage

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

Test

```
dm = library("demos.lib");
no = library("noises.lib");
reverse_echo_demo_test = no.noise : dm.reverse_echo_demo(3);
```

(dm.)pospass_demo

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

Usage

```
echo 'import("stdfaust.lib");' > pospass_demo.dsp
echo 'process = dm.pospass_demo;' >> pospass_demo.dsp
Mac:
  faust2caqt pospass_demo.dsp
  open pospass_demo.app
```

```
Linux GTK:
  faust2jack pospass_demo.dsp
  ./pospass_demo
Linux QT:
  faust2jaqt pospass_demo.dsp
  ./pospass_demo
Etc.
```

Test

```
dm = library("demos.lib");
os = library("oscillators.lib");
pospass_demo_test = os.osc(440) : dm.pospass_demo;
```

(dm.)exciter

Psychoacoustic harmonic exciter, with GUI.

Usage

```
_ : exciter : _
```

Test

```
dm = library("demos.lib");
no = library("noises.lib");
exciter_test = no.noise : dm.exciter;
```

References

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

---

(dm.)vocoder_demo

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

Usage

```
_ : vocoder_demo : _
```

Test

```
dm = library("demos.lib");
os = library("oscillators.lib");
no = library("noises.lib");
vocoder_demo_test = no.noise : dm.vocoder_demo;
```

(dm.)colored_noise_demo

A coloured noise signal generator.

Usage

```
colored_noise_demo : _
```

Test

```
dm = library("demos.lib");
colored_noise_demo_test = dm.colored_noise_demo;
```

envelopes.lib

Envelopes library. Its official prefix is `en`.

This library provides envelope generators and control functions for shaping signal amplitude, pitch, or other parameters. It includes ADSR, AR, and percussive models, as well as exponential, linear, and segmented envelope types used in both synthesis and dynamic processing contexts.

The Envelopes library is organized into 3 sections:

- Envelopes with linear segments
- Envelopes with exponential segments
- Others

References

- https://github.com/grame-cncm/faustlibraries/blob/master/envelopes.lib

Envelopes with linear segments

(en.)ar

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

Usage

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

Where:

- `at`: attack (sec)
- `rt`: release (sec)
- `t`: trigger signal (attack is triggered when `t>0`, release is triggered when the envelope value reaches 1)

Test

```
en = library("envelopes.lib");
no = library("noises.lib");
ar_test = no.noise * en.ar(0.02, 0.3, button("gate"));
```

`(en.)asr`

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

Usage

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

Where:

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

Test

```
en = library("envelopes.lib");
no = library("noises.lib");
asr_test = no.noise * en.asr(0.05, 0.7, 0.4, button("gate"));
```

`(en.)adsr`

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

Usage

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

Where:

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

Test

```
en = library("envelopes.lib");
no = library("noises.lib");
adsr_test = no.noise * en.adsr(0.05, 0.1, 0.6, 0.3, button("gate"));
```

`(en.)adsrf_bias`

ADSR (Attack, Decay, Sustain, Release, Final) envelope generator with control over bias on each segment, and toggle for legato.

Usage

```
adsrf_bias(at,dt,sl,rt,final,b_att,b_dec,b_rel,legato,t) : _
```

Where:

- `at`: attack time (sec)
- `dt`: decay time (sec)
- `sl`: sustain level (between 0..1)
- `rt`: release time (sec)
- `final`: final level (between 0..1) but less than or equal to `sl`
- `b_att`: bias during attack (between 0..1) where 0.5 is no bias.
- `b_dec`: bias during decay (between 0..1) where 0.5 is no bias.
- `b_rel`: bias during release (between 0..1) where 0.5 is no bias.
- `legato`: toggle for legato. If disabled, envelopes "re-trigger" from zero.
- `t`: trigger signal (attack is triggered when `t>0`, release is triggered when `t=0`)

Test

```
en = library("envelopes.lib");
no = library("noises.lib");
adsrf_bias_test = no.noise * en.adsrf_bias(
  0.05, 0.1, 0.6, 0.4, 0.2,
```

```
  0.4, 0.6, 0.5,
  checkbox("legato"), button("gate")
);
```

(en.)adsr_bias

ADSR (Attack, Decay, Sustain, Release) envelope generator with control over bias on each segment, and toggle for legato.

Usage

```
adsr_bias(at,dt,sl,rt,b_att,b_dec,b_rel,legato,t) : _
```

Where:

- `at`: attack time (sec)
- `dt`: decay time (sec)
- `sl`: sustain level (between 0..1)
- `rt`: release time (sec)
- `b_att`: bias during attack (between 0..1) where 0.5 is no bias.
- `b_dec`: bias during decay (between 0..1) where 0.5 is no bias.
- `b_rel`: bias during release (between 0..1) where 0.5 is no bias.
- `legato`: toggle for legato. If disabled, envelopes "re-trigger" from zero.
- `t`: trigger signal (attack is triggered when `t>0`, release is triggered when `t=0`)

Test

```
en = library("envelopes.lib");
no = library("noises.lib");
adsr_bias_test = no.noise * en.adsr_bias(
  0.05, 0.1, 0.6, 0.4,
  0.4, 0.6, 0.5,
  checkbox("legato"), button("gate")
);
```

(en.)ahdsrf_bias

AHDSR (Attack, Hold, Decay, Sustain, Release, Final) envelope generator with control over bias on each segment, and toggle for legato.

Usage

```
ahdsrf_bias(at,ht,dt,sl,rt,final,b_att,b_dec,b_rel,legato,t) : _
```

Where:

- `at`: attack time (sec)
- `ht`: hold time (sec)
- `dt`: decay time (sec)
- `sl`: sustain level (between 0..1)
- `rt`: release time (sec)
- `final`: final level (between 0..1) but less than or equal to `sl`
- `b_att`: bias during attack (between 0..1) where 0.5 is no bias.
- `b_dec`: bias during decay (between 0..1) where 0.5 is no bias.
- `b_rel`: bias during release (between 0..1) where 0.5 is no bias.
- `legato`: toggle for legato. If disabled, envelopes "re-trigger" from zero.
- `t`: trigger signal (attack is triggered when `t>0`, release is triggered when `t=0`)

Test

```
en = library("envelopes.lib");
no = library("noises.lib");
ahdsrf_bias_test = no.noise * en.ahdsrf_bias(
  0.05, 0.05, 0.1, 0.6, 0.4, 0.2,
  0.4, 0.6, 0.5,
  checkbox("legato"), button("gate")
);
```

`(en.)ahdsr_bias`

AHDSR (Attack, Hold, Decay, Sustain, Release) envelope generator with control over bias on each segment, and toggle for legato.

Usage

```
ahdsr_bias(at,ht,dt,sl,rt,final,b_att,b_dec,b_rel,legato,t) : _
```

Where:

- `at`: attack time (sec)
- `ht`: hold time (sec)
- `dt`: decay time (sec)
- `sl`: sustain level (between 0..1)
- `rt`: release time (sec)
- `final`: final level (between 0..1) but less than or equal to `sl`
- `b_att`: bias during attack (between 0..1) where 0.5 is no bias.
- `b_dec`: bias during decay (between 0..1) where 0.5 is no bias.
- `b_rel`: bias during release (between 0..1) where 0.5 is no bias.
- `legato`: toggle for legato. If disabled, envelopes "re-trigger" from zero.
- `t`: trigger signal (attack is triggered when `t>0`, release is triggered when `t=0`)

Test

```
en = library("envelopes.lib");
no = library("noises.lib");
ahdsr_bias_test = no.noise * en.ahdsr_bias(
  0.05, 0.05, 0.1, 0.6, 0.4,
  0.4, 0.6, 0.5,
  checkbox("legato"), button("gate")
);
```

Envelopes with exponential segments

(en.)smoothEnvelope

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

Usage

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

Where:

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

Test

```
en = library("envelopes.lib");
no = library("noises.lib");
smoothEnvelope_test = no.noise * en.smoothEnvelope(0.2, button("gate"));
```

(en.)arfe

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

Usage

```
arfe(at,rt,fl,t) : _
```

Where:

- `at`: attack (sec)

- `rt`: release (sec)
- `fl`: final level to approach upon release (such as 0)
- `t`: trigger signal (attack is triggered when `t>0`, release is triggered when `t=0`)

Test

```
en = library("envelopes.lib");
no = library("noises.lib");
arfe_test = no.noise * en.arfe(0.2, 0.4, 0, button("gate"));
```

`(en.)are`

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

Usage

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

Where:

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

Test

```
en = library("envelopes.lib");
no = library("noises.lib");
are_test = no.noise * en.are(0.2, 0.4, button("gate"));
```

`(en.)asre`

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

Usage

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

Where:

- `at`: attack (sec)
- `sl`: sustain level (between 0..1)
- `rt`: release (sec)

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

Test

```
en = library("envelopes.lib");
no = library("noises.lib");
asre_test = no.noise * en.asre(0.2, 0.6, 0.4, button("gate"));
```

(en.)adsre

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

Usage

```
adsre(at,dt,sl,rt,t) : _
```

Where:

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

Test

```
en = library("envelopes.lib");
no = library("noises.lib");
adsre_test = no.noise * en.adsre(0.2, 0.1, 0.6, 0.4, button("gate"));
```

(en.)ahdsre

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

Usage

```
ahdsre(at,ht,dt,sl,rt,t) : _
```

Where:

- `at`: attack (sec)
- `ht`: hold (sec)
- `dt`: decay (sec)

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

Test

```
en = library("envelopes.lib");
no = library("noises.lib");
ahdsre_test = no.noise * en.ahdsre(0.2, 0.05, 0.1, 0.6, 0.4, button("gate"));
```

Others

`(en.)dx7envelope`

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

Usage

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

Where:

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

Test

```
en = library("envelopes.lib");
os = library("oscillators.lib");
dx7envelope_test = en.dx7envelope(
  0.05, 0.1, 0.1, 0.2,
  1, 0.8, 0.6, 0,
  button("gate")
) * os.osc(440);
```

fds.lib

This library allows to build linear, explicit finite difference schemes physical models in 1 or 2 dimensions using an approach based on the cellular automata formalism. Its official prefix is `fd`.

In order to use the library, one needs to discretize the linear partial differential equation of the desired system both at boundaries and in-between them, thus obtaining a set of explicit recursion relations. Each one of these will provide, for each spatial point the scalar coefficients to be multiplied by the states of the current and past neighbour points.

Coefficients need to be stacked in parallel in order to form a coefficients matrix for each point in the mesh. It is necessary to provide one matrix for coefficients matrices are defined, they need to be placed in parallel and ordered following the desired mesh structure (i.e., coefficients for the top left boundaries will come first, while bottom right boundaries will come last), to form a *coefficients scheme*, which can be used with the library functions.

Sources

Here are listed some works on finite difference schemes and cellular automata thet were the basis for the implementation of this library

- S. Bilbao, Numerical Sound Synthesis.Chichester, UK: John Wiley Sons, Ltd, 2009
- P. Narbel, "Qualitative and quantitative cellular automata from differential equations," Lecture Notes in Computer Science, vol. 4173, pp. 112–121, 10 2006
- X.-S. Yang and Y. Young, Cellular Automata, PDEs, and Pattern Formation. Chapman & Hall/CRC, 092005, ch. 18, pp. 271–282.

The FDS library is organized into 5 sections:

- Model Construction
- Interpolation
- Routing
- Scheme Operations
- Interaction Models

References

- https://github.com/grame-cncm/faustlibraries/blob/master/fds.lib

Model Construction

Once the coefficients scheme is defined, the user can simply call one of these functions to obtain a fully working physical model. They expect to receive a force input signal for each mesh point and output the state of each point. Interpolation operators can be used to drive external forces to the desired points, and to get the signal only from a certain area of the mesh.

`(fd.)model1D`

This function can be used to obtain a physical model in 1 dimension. Takes a force input signal for each point and outputs the state of each point.

Usage

```
si.bus(points) : model1D(points,R,T,scheme) : si.bus(points)
```

Where:

- `points`: size of the mesh in points
- `R`: neighbourhood radius, indicates how many side points are needed (i.e. if R=1 the mesh depends on one point on the left and one on the right)
- `T`: time coefficient, indicates how much steps back in time are needed (i. e. if T=1 the maximum delay needed for a neighbour state is 1 sample)
- `scheme`: coefficients scheme

Test

```
fd = library("fds.lib");
si = library("signals.lib");

scheme = 0, 0;
model1D_test = si.bus(2)
  : fd.model1D(2, 0, 0, scheme)
  : si.bus(2);
```

`(fd.)model2D`

This function can be used to obtain a physical model in 2 dimension. Takes a force input signal for each point and outputs the state of each point. IMPORTANT: 2D models with more than 30x20 points might crash the c++ compiler. 2D models need to be compiled with the command line compiler, the online one presents some issues.

Usage

```
si.bus(pointsX*pointsY) : model2D(pointsX,pointsY,R,T,scheme) :
   si.bus(pointsX*pointsY)
```

Where:

- `pointsX`: horizontal size of the mesh in points
- `pointsY`: vertical size of the mesh in points
- `R`: neighbourhood radius, indicates how many side points are needed (i.e. if R=1 the mesh depends on one point on the left and one on the right)

- `T`: time coefficient, indicates how much steps back in time are needed (i. e. if T=1 the maximum delay needed for a neighbour state is 1 sample)
- `scheme`: coefficients scheme

Test

```
fd = library("fds.lib");
si = library("signals.lib");

scheme = 0, 0, 0, 0;
model2D_test = si.bus(4)
  : fd.model2D(2, 2, 0, 0, scheme)
  : si.bus(4);
```

Interpolation

Interpolation functions can be used to drive the input signals to the correct mesh points, or to get the output signal from the desired points. All the interpolation functions allow to change the input/output points at run time. In general, all these functions get in input a number of connections, and output the same number of connections, where each signal is multiplied by zero except the ones specified by the arguments.

`(fd.)stairsInterp1D`

Stairs interpolator in 1 dimension. Takes a number of signals and outputs the same number of signals, where each one is multiplied by zero except the one specified by the argument. This can vary at run time (i.e. a slider), but must be an integer.

Usage

```
si.bus(points) : stairsInterp1D(points,point) : si.bus(points)
```

Where:

- `points`: total number of points in the mesh
- `point`: number of the desired nonzero signal

Test

```
fd = library("fds.lib");
si = library("signals.lib");

stairsInterp1D_test = si.bus(4)
  : fd.stairsInterp1D(4, 1);
```

(fd.)stairsInterp2D

Stairs interpolator in 2 dimensions. Similar to the 1-D version.

Usage

```
si.bus(pointsX*pointsY) : stairsInterp2D(pointsX,pointsY,pointX,pointY) :
   si.bus(pointsX*pointsY)
```

Where:

- `pointsX`: total number of points in the X direction
- `pointsY`: total number of points in the Y direction
- `pointX`: horizontal index of the desired nonzero signal
- `pointY`: vertical index of the desired nonzero signal

Test

```
fd = library("fds.lib");
si = library("signals.lib");

stairsInterp2D_test = si.bus(4)
  : fd.stairsInterp2D(2, 2, 1, 0);
```

(fd.)linInterp1D

Linear interpolator in 1 dimension. Takes a number of signals and outputs the same number of signals, where each one is multiplied by zero except two signals around a floating point index. This is essentially a Faust implementation of the $J(x_i)$ operator, not scaled by the spatial step. (see Stefan Bilbao's book, Numerical Sound Synthesis). The index can vary at run time.

Usage

```
si.bus(points) : linInterp1D(points,point) : si.bus(points)
```

Where:

- `points`: total number of points in the mesh
- `point`: floating point index

Test

```
fd = library("fds.lib");
si = library("signals.lib");
```

```
linInterp1D_test = si.bus(4)
  : fd.linInterp1D(4, 1.25);
```

(fd.)linInterp2D

Linear interpolator in 2 dimensions. Similar to the 1 D version.

Usage

```
si.bus(pointsX*pointsY) : linInterp2D(pointsX,pointsY,pointX,pointY) :
   si.bus(pointsX*pointsY)
```

Where:

- `pointsX`: total number of points in the X direction
- `pointsY`: total number of points in the Y direction
- `pointX`: horizontal float index
- `pointY`: vertical float index

Test

```
fd = library("fds.lib");
si = library("signals.lib");

linInterp2D_test = si.bus(4)
  : fd.linInterp2D(2, 2, 0.6, 1.2);
```

(fd.)stairsInterp1DOut

Stairs interpolator in 1 dimension. Similar to `stairsInterp1D`, except it outputs only the desired signal.

Usage

```
si.bus(points) : stairsInterp1DOut(points,point) : _
```

Where:

- `points`: total number of points in the mesh
- `point`: number of the desired nonzero signal

Test

```
fd = library("fds.lib");
si = library("signals.lib");

stairsInterp1DOut_test = si.bus(4)
```

```
  : fd.stairsInterp1DOut(4, 2);
```

(fd.)stairsInterp2DOut

Stairs interpolator in 2 dimensions which outputs only one signal.

Usage

```
si.bus(pointsX*pointsY) : stairsInterp2DOut(pointsX,pointsY,pointX,pointY) : _
```

Where:

- `pointsX`: total number of points in the X direction
- `pointsY`: total number of points in the Y direction
- `pointX`: horizontal index of the desired nonzero signal
- `pointY`: vertical index of the desired nonzero signal

Test

```
fd = library("fds.lib");
si = library("signals.lib");

stairsInterp2DOut_test = si.bus(4)
  : fd.stairsInterp2DOut(2, 2, 1, 0);
```

(fd.)linInterp1DOut

Linear interpolator in 1 dimension. Similar to `stairsInterp1D`, except it sums each output signal and provides only one output value.

Usage

```
si.bus(points) : linInterp1DOut(points,point) : _
```

Where:

- `points`: total number of points in the mesh
- `point`: floating point index

Test

```
fd = library("fds.lib");
si = library("signals.lib");

linInterp1DOut_test = si.bus(4)
  : fd.linInterp1DOut(4, 1.5);
```

`(fd.)stairsInterp2DOut`

Linear interpolator in 2 dimensions which outputs only one signal.

Usage

```
si.bus(pointsX*pointsY) : linInterp2DOut(pointsX,pointsY,pointX,pointY) : _
```

Where:

- `pointsX`: total number of points in the X direction
- `pointsY`: total number of points in the Y direction
- `pointX`: horizontal float index
- `pointY`: vertical float index

Test

```
fd = library("fds.lib");
si = library("signals.lib");

linInterp2DOut_test = si.bus(4)
  : fd.linInterp2DOut(2, 2, 0.6, 1.2);
```

Routing

The routing functions are used internally by the model building functions, but can also be taken separately. These functions route the forces, the coefficients scheme and the neighbours' signals into the correct scheme points and take as input, in this order: the coefficients block, the feedback signals and the forces. In output they provide, in order, for each scheme point: the force signal, the coefficient matrices and the neighbours' signals. These functions are based on the Faust route primitive.

`(fd.)route1D`

Routing function for 1 dimensional schemes.

Usage

```
si.bus((2*R+1)*(T+1)*points),si.bus(points*2) : route1D(points, R, T) :
   si.bus((1 + ((2*R+1)*(T+1)) + (2*R+1))*points)
```

Where:

- `points`: total number of points in the mesh
- `R`: neighbourhood radius

- T: time coefficient

Test

```
fd = library("fds.lib");
si = library("signals.lib");

route1D_test = par(i, 3, 0)
  : fd.route1D(1, 0, 0)
  : si.bus(3);
```

(fd.)route2D

Routing function for 2 dimensional schemes.

Usage

```
si.bus((2*R+1)^2*(T+1)*pointsX*pointsY),si.bus(pointsX*pointsY*2) :
   route2D(pointsX, pointsY, R, T) :
       si.bus((1 + ((2*R+1)^2*(T+1)) + (2*R+1)^2)*pointsX*pointsY)
```

Where:

- pointsX: total number of points in the X direction
- pointsY: total number of points in the Y direction
- R: neighbourhood radius
- T: time coefficient

Test

```
fd = library("fds.lib");
si = library("signals.lib");

route2D_test = par(i, 3, 0)
  : fd.route2D(1, 1, 0, 0)
  : si.bus(3);
```

Scheme Operations

The scheme operation functions are used internally by the model building functions but can also be taken separately. The schemePoint function is where the update equation is actually calculated. The `buildScheme` functions are used to stack in parallel several schemePoint blocks, according to the choosed mesh size.

(fd.)schemePoint

This function calculates the next state for each mesh point, in order to form a scheme, several of these blocks need to be stacked in parallel. This function takes in input, in order, the force, the coefficient matrices and the neighbours' signals and outputs the next point state.

Usage

```
_,si.bus((2*R+1)^D*(T+1)),si.bus((2*R+1)^D) : schemePoint(R,T,D) : _
```

Where:

- `R`: neighbourhood radius
- `T`: time coefficient
- `D`: scheme spatial dimensions (i.e. 1 if 1-D, 2 if 2-D)

Test

```
fd = library("fds.lib");
si = library("signals.lib");

schemePoint_test = par(i, 3, 0)
  : fd.schemePoint(0, 0, 1);
```

(fd.)buildScheme1D

This function is used to stack in parallel several schemePoint functions in 1 dimension, according to the number of points.

Usage

```
si.bus((1 + ((2*R+1)*(T+1)) + (2*R+1))*points) : buildScheme1D(points,R,T) :
   si.bus(points)
```

Where:

- `points`: total number of points in the mesh
- `R`: neighbourhood radius
- `T`: time coefficient

Test

```
fd = library("fds.lib");
si = library("signals.lib");

buildScheme1D_test = par(i, 3, 0)
  : fd.buildScheme1D(1, 0, 0);
```

`(fd.)buildScheme2D`

This function is used to stack in parallel several schemePoint functions in 2 dimensions, according to the number of points in the X and Y directions.

Usage

```
si.bus((1 + ((2*R+1)^2*(T+1)) + (2*R+1)^2)*pointsX*pointsY) :
   buildScheme2D(pointsX,pointsY,R,T) : si.bus(pointsX*pointsY)
```

Where:

- `pointsX`: total number of points in the X direction
- `pointsY`: total number of points in the Y direction
- `R`: neighbourhood radius
- `T`: time coefficient

Test

```
fd = library("fds.lib");
si = library("signals.lib");

buildScheme2D_test = par(i, 3, 0)
  : fd.buildScheme2D(1, 1, 0, 0);
```

Interaction Models

Here are defined two physically based interaction algorithms: a hammer and a bow. These functions need to be coupled to the mesh pde, in the point where the interaction happens: to do so, the mesh output signals can be fed back and driven into the force block using the interpolation operators. The latters can be also used to drive the single force output signal to the correct scheme points.

`(fd.)hammer`

Implementation of a nonlinear collision model. The hammer is essentially a finite difference scheme of a linear damped oscillator, which is coupled with the mesh through the collision model (see Stefan Bilbao's book, Numerical Sound Synthesis).

Usage

```
_ :hammer(coeff,omega0Sqr,sigma0,kH,alpha,k,offset,fIn) : _
```

Where:

- `coeff`: output force scaling coefficient
- `omega0Sqr`: squared angular frequency of the hammer oscillator
- `sigma0`: damping coefficient of the hammer oscillator
- `kH`: hammer stiffness coefficient
- `alpha`: nonlinearity parameter
- `k`: time sampling step (the same as for the mesh)
- `offset`: distance between the string and the hammer at rest in meters
- `fIn`: hammer excitation signal (i.e. a button)

Test

```
fd = library("fds.lib");
os = library("oscillators.lib");

hammer_test = os.osc(5)
  : fd.hammer(
      0.1,
      1000,
      0.01,
      1e5,
      2.0,
      1.0/48000,
      0.001,
      button("hammer:trigger")
    );
```

(fd.)bow

Implementation of a nonlinear friction based interaction model that induces Helmholtz motion. (see Stefan Bilbao's book, Numerical Sound Synthesis).

Usage

```
_ :bow(coeff,alpha,k,vb) : _
```

Where:

- `coeff`: output force scaling coefficient
- `alpha`: nonlinearity parameter
- `k`: time sampling step (the same as for the mesh)
- `vb`: bow velocity [m/s]

Test

```
fd = library("fds.lib");
os = library("oscillators.lib");
```

```
bow_test = os.osc(5)
  : fd.bow(0.05, 2.0, 1.0/48000, 0.1);
```

filters.lib

Filters library. Its official prefix is `fi`.

This library provides a comprehensive collection of linear and nonlinear filters used in audio and signal processing. It includes low-pass, high-pass, band-pass, allpass, shelving, equalizer, and crossover filters, as well as advanced analog and digital filter design sections for both educational and production use.

The Filters library is organized into 23 sections:

- Basic Filters
- Comb Filters
- Direct-Form Digital Filter Sections
- Direct-Form Second-Order Biquad Sections
- Ladder/Lattice Digital Filters
- Useful Special Cases
- Ladder/Lattice Allpass Filters
- Digital Filter Sections Specified as Analog Filter Sections
- Simple Resonator Filters
- Butterworth Lowpass/Highpass Filters
- Special Filter-Bank Delay-Equalizing Allpass Filters
- Elliptic (Cauer) Lowpass Filters
- Elliptic Highpass Filters
- Butterworth Bandpass/Bandstop Filters
- Elliptic Bandpass Filters
- Parametric Equalizers (Shelf, Peaking)
- Mth-Octave Filter-Banks
- Arbitrary-Crossover Filter-Banks and Spectrum Analyzers
- State Variable Filters (SVF)
- Linkwitz-Riley 4th-order 2-way, 3-way, and 4-way crossovers
- Standardized Filters
- Averaging Functions
- Kalman Filters

References

- https://github.com/grame-cncm/faustlibraries/blob/master/filters.lib

Basic Filters

`(fi.)zero`

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

Usage

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

Where:

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

Test

```
fi = library("filters.lib");
os = library("oscillators.lib");
zero_test = os.osc(440) : fi.zero(0.5);
```

References

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

`(fi.)pole`

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

Usage

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

Where:

- `p`: pole location = feedback coefficient

Test

```
fi = library("filters.lib");
os = library("oscillators.lib");
pole_test = os.osc(440) : fi.pole(0.9);
```

References

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

`(fi.)integrator`

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

Usage

```
_ : integrator : _
```

Test

```
fi = library("filters.lib");
os = library("oscillators.lib");
integrator_test = os.osc(440) : fi.integrator;
```

(fi.)dcblockerat

DC blocker with configurable "break frequency". The amplitude response is substantially flat above `fb`, and sloped at about +6 dB/octave below `fb`. Derived from the analog transfer function:

$$H(s) = \frac{s}{(s + 2\pi f_b)}$$

(which can be seen as a 1st-order Butterworth highpass filter) by the low-frequency-matching bilinear transform method (i.e., using the typical frequency-scaling constant `2*SR`).

Usage

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

Where:

- `fb`: "break frequency" in Hz, i.e., -3 dB gain frequency (see 2nd reference below)

Test

```
fi = library("filters.lib");
os = library("oscillators.lib");
dcblockerat_test = os.osc(440) : fi.dcblockerat(30);
```

References

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

(fi.)dcblocker

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

Usage

```
_ : dcblocker : _
```

Test

```
fi = library("filters.lib");
os = library("oscillators.lib");
dcblocker_test = os.osc(440) : fi.dcblocker;
```

(fi.)lptN

One-pole lowpass filter with arbitrary dis/charging factors set in dB and times set in seconds.

Usage

```
_ : lptN(N, tN) : _
```

Where:

- `N`: is the attenuation factor in dB
- `tN`: is the filter period in seconds, that is, the time for the impulse response to decay by `N` dB

Test

```
fi = library("filters.lib");
os = library("oscillators.lib");
lptN_test = os.osc(440) : fi.lptN(60, 0.1);
```

References

- https://ccrma.stanford.edu/~jos/mdft/Exponentials.html

(fi.)lptau

One-pole lowpass with a tau time constant (1/e attenuation after `tN` seconds).

Usage

```
_ : lptau(tN) : _
```

Test

```
fi = library("filters.lib");
os = library("oscillators.lib");
lptau_test = os.osc(440) : fi.lptau(0.1);
```

(fi.)lpt60

One-pole lowpass with a T60 time constant (60 dB attenuation after `tN` seconds).

Usage

```
_ : lpt60(tN) : _
```

Test

```
fi = library("filters.lib");
os = library("oscillators.lib");
lpt60_test = os.osc(440) : fi.lpt60(0.3);
```

(fi.)lpt19

One-pole lowpass with a T19 time constant (approx. 19 dB attenuation after `tN` seconds).

Usage

```
_ : lpt19(tN) : _
```

Test

```
fi = library("filters.lib");
os = library("oscillators.lib");
lpt19_test = os.osc(440) : fi.lpt19(0.2);
```

Comb Filters

(fi.)ff_comb

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

Usage

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

Where:

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

Test

```
fi = library("filters.lib");
os = library("oscillators.lib");
ff_comb_test = os.osc(440) : fi.ff_comb(2048, 64, 1, 0.7);
```

References

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

---

(fi.)ff_fcomb

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

Usage

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

Where:

- `maxdel`: maximum delay (a power of 2)
- `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

Test

```
fi = library("filters.lib");
os = library("oscillators.lib");
ff_fcomb_test = os.osc(440) : fi.ff_fcomb(2048, 64.5, 1, 0.7);
```

References

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

---

(fi.)ffcombfilter

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

Test

```
fi = library("filters.lib");
os = library("oscillators.lib");
ffcombfilter_test = os.osc(440) : fi.ffcombfilter(2048, 64, 0.7);
```

(fi.)fb_comb_common

A generic feedback comb filter.

Usage

```
_ : fb_comb_common(dop,N,b0,aN) : _
```

Where

- `dop`: delay operator, e.g. `@` or `de.fdelay4a(2048)`
- `N`: current delay
- `b0`: gain applied to input
- `aN`: gain applied to delay-line output

Example test program

```
process = fb_comb_common(@,N,b0,aN);
```

implements the following difference equation:

```
y[n] = b0 x[n] + aN y[n - N]
```

See more examples in `filters.lib` below.

Test

```
fi = library("filters.lib");
os = library("oscillators.lib");
fb_comb_common_test = os.osc(440) : fi.fb_comb_common(@, 64, 0.8, 0.6);
```

(fi.)fb_comb

Feed-Back Comb Filter (integer delay).

Usage

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

Where:

- `maxdel`: maximum delay (a power of 2)
- `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

Test

```
fi = library("filters.lib");
os = library("oscillators.lib");
fb_comb_test = os.osc(440) : fi.fb_comb(2048, 64, 0.7, 0.6);
```

References

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

---

(fi.)fb_fcomb

Feed-Back Comb Filter (floating point delay).

Usage

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

Where:

- `maxdel`: maximum delay (a power of 2)
- `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

Test

```
fi = library("filters.lib");
os = library("oscillators.lib");
fb_fcomb_test = os.osc(440) : fi.fb_fcomb(2048, 64.5, 0.7, 0.6);
```

References

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

---

(fi.)rev1

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

Test

```
fi = library("filters.lib");
os = library("oscillators.lib");
rev1_test = os.osc(440) : fi.rev1(2048, 64, 0.6);
```

(fi.)fbcombfilter and (fi.)ffbcombfilter

Other special cases of Feed-Back Comb Filter.

Usage

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

Where:

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

Test

```
fi = library("filters.lib");
os = library("oscillators.lib");
fbcombfilter_test = os.osc(440) : fi.fbcombfilter(2048, 64, 0.6);
ffbcombfilter_test = os.osc(440) : fi.ffbcombfilter(2048, 64.5, 0.6);
```

References

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

---

(fi.)allpass_comb

Schroeder Allpass Comb Filter. Note that:

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

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

Usage

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

Where:

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

Test

```
fi = library("filters.lib");
os = library("oscillators.lib");
allpass_comb_test = os.osc(440) : fi.allpass_comb(2048, 64, 0.6);
```

References

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

---

`(fi.)allpass_fcomb`

Schroeder Allpass Comb Filter. Note that:

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

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

`allpass_fcomb` is a standard Faust library.

Usage

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

Where:

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

Test

```
fi = library("filters.lib");
os = library("oscillators.lib");
allpass_fcomb_test = os.osc(440) : fi.allpass_fcomb(2048, 64.5, 0.6);
```

References

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

---

(fi.)rev2

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

Test

```
fi = library("filters.lib");
os = library("oscillators.lib");
rev2_test = os.osc(440) : fi.rev2(2048, 64, 0.6);
```

(fi.)allpass_fcomb5 and (fi.)allpass_fcomb1a

Same as `allpass_fcomb` but use `fdelay5` and `fdelay1a` internally (Interpolation helps - look at an fft of faust2octave on: `1-1' <: allpass_fcomb(1024,10.5,0.95), allpass_fcomb5(1024,10.5,0.95);`)

Test

```
fi = library("filters.lib");
os = library("oscillators.lib");
allpass_fcomb5_test = os.osc(440) : fi.allpass_fcomb5(2048, 64.5, 0.6);
allpass_fcomb1a_test = os.osc(440) : fi.allpass_fcomb1a(2048, 64.5, 0.6);
```

Direct-Form Digital Filter Sections

(fi.)iir

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

`iir` is a standard Faust function.

Usage

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

Where:

- `bcoeffs`: (b0,b1,…,b_order) = TF numerator coefficients
- `acoeffs`: (a1,…,a_order) = TF denominator coeffs (a0=1)

Test

```
fi = library("filters.lib");
os = library("oscillators.lib");
iir_test = os.osc(440) : fi.iir((0.5, 0.5), (0.3));
```

References

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

---

(fi.)fir

FIR filter (convolution of FIR filter coefficients with a signal). `fir` is standard Faust function.

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 test program Smoothing white noise with a five-point moving average:

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

Equivalent (note double parens):

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

(fi.)conv and (fi.)convN

Convolution of input signal with given coefficients.

Usage

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

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

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

Usage

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

Where:

- `b`: transfer-function numerator
- `a`: transfer-function denominator (monic)

Test

```
fi = library("filters.lib");
os = library("oscillators.lib");
tf1_test = os.osc(440) : fi.tf1(0.5, 0.25, -0.4);
tf2_test = os.osc(440) : fi.tf2(0.1, 0.2, 0.1, -0.5, 0.06);
tf3_test = os.osc(440) : fi.tf3(0.1, 0.3, 0.3, 0.1, -0.9, 0.26, -0.024);
```

References

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

(fi.)notchw

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

Usage:

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

Where:

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

Test

```
fi = library("filters.lib");
os = library("oscillators.lib");
notchw_test = os.osc(440) : fi.notchw(200, 1000);
```

References

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

Direct-Form Second-Order Biquad Sections

Direct-Form Second-Order Biquad Sections

References

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

---

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

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

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

Usage

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

Where:

- `b`: transfer-function numerator
- `a`: transfer-function denominator (monic)

Test

```
fi = library("filters.lib");
os = library("oscillators.lib");
tf21_test = os.osc(440) : fi.tf21(0.1, 0.2, 0.1, -0.5, 0.06);
tf22_test = os.osc(440) : fi.tf22(0.1, 0.2, 0.1, -0.5, 0.06);
tf22t_test = os.osc(440) : fi.tf22t(0.1, 0.2, 0.1, -0.5, 0.06);
tf21t_test = os.osc(440) : fi.tf21t(0.1, 0.2, 0.1, -0.5, 0.06);
```

References

- 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. #### References

- F. Itakura and S. Saito: "Digital Filtering Techniques for Speech Analysis and Synthesis", 7th Int. Cong. Acoustics, Budapest, 25 C 1, 1971.
- J. D. Markel and A. H. Gray: Linear Prediction of Speech, New York: Springer Verlag, 1976.
- https://ccrma.stanford.edu/~jos/pasp/Conventional_Ladder_Filters.html

---

`(fi.)av2sv`

Compute reflection coefficients sv from transfer-function denominator av.

Usage

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

Where:

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

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

Test

```
fi = library("filters.lib");
si = library("signals.lib");
av2sv_test = fi.av2sv((-0.4, 0.1)) : si.bus(2);
```

References

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

`(fi.)bvav2nuv`

Compute lattice tap coefficients from transfer-function coefficients.

Usage

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

Where:

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

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

Test

```
fi = library("filters.lib");
si = library("signals.lib");
bvav2nuv_test = fi.bvav2nuv((0.1, 0.2, 0.3), (-0.4, 0.1)) : si.bus(3);
```

`(fi.)iir_lat2`

Two-multiply lattice IIR filter of arbitrary order.

Usage

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

Where:

- `bv`: transfer-function numerator
- `av`: transfer-function denominator (monic)

Test

```
fi = library("filters.lib");
os = library("oscillators.lib");
iir_lat2_test = os.osc(440) : fi.iir_lat2((0.1, 0.2, 0.3), (-0.4, 0.1));
```

(fi.)allpassnt

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

Usage

```
_ : allpassnt(n,sv) : si.bus(n+1)
```

Where:

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

The first output is the n-th order allpass output, while the remaining outputs are taps taken from the input of each delay element from the input to the output. See (fi.)allpassn for the single-output case.

Test

```
fi = library("filters.lib");
os = library("oscillators.lib");
si = library("signals.lib");
allpassnt_test = os.osc(440) : fi.allpassnt(2, (0.3, -0.2)) : si.bus(3);
```

(fi.)iir_kl

Kelly-Lochbaum ladder IIR filter of arbitrary order.

Usage

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

Where:

- `bv`: transfer-function numerator
- `av`: transfer-function denominator (monic)

Test

```
fi = library("filters.lib");
os = library("oscillators.lib");
iir_kl_test = os.osc(440) : fi.iir_kl((0.1, 0.2, 0.3), (-0.4, 0.1));
```

(fi.)allpassnklt

Kelly-Lochbaum ladder allpass.

Usage:

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

Where:

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

Test

```
fi = library("filters.lib");
os = library("oscillators.lib");
si = library("signals.lib");
allpassnklt_test = os.osc(440) : fi.allpassnklt(2, (0.3, -0.2)) : si.bus(3);
```

`(fi.)iir_lat1`

One-multiply lattice IIR filter of arbitrary order.

Usage

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

Where:

- bv: transfer-function numerator as a bank of parallel signals
- av: transfer-function denominator as a bank of parallel signals

Test

```
fi = library("filters.lib");
os = library("oscillators.lib");
iir_lat1_test = os.osc(440) : fi.iir_lat1((0.1, 0.2, 0.3), (-0.4, 0.1));
```

`(fi.)allpassn1mt`

One-multiply lattice allpass with tap lines.

Usage

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

Where:

- `N`: the order of the filter (fixed at compile time)
- `sv`: the reflection coefficients (-1 1)

Test

```
fi = library("filters.lib");
os = library("oscillators.lib");
si = library("signals.lib");
allpassn1mt_test = os.osc(440) : fi.allpassn1mt(2, (0.3, -0.2)) : si.bus(3);
```

(fi.)iir_nl

Normalized ladder filter of arbitrary order.

Usage

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

Where:

- `bv`: transfer-function numerator
- `av`: transfer-function denominator (monic)

Test

```
fi = library("filters.lib");
os = library("oscillators.lib");
iir_nl_test = os.osc(440) : fi.iir_nl((0.1, 0.2, 0.3), (-0.4, 0.1));
```

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

---

(fi.)allpassnnlt

Normalized ladder allpass filter of arbitrary order.

Usage:

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

Where:

- `N`: the order of the filter (fixed at compile time)
- `sv`: the reflection coefficients (-1 1)

Test

```
fi = library("filters.lib");
os = library("oscillators.lib");
si = library("signals.lib");
allpassnnlt_test = os.osc(440) : fi.allpassnnlt(2, (0.3, -0.2)) : si.bus(3);
```

References

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

Useful Special Cases

`(fi.)tf2np`

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

Usage

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

Where:

- `b`: transfer-function numerator
- `a`: transfer-function denominator (monic)

Test

```
fi = library("filters.lib");
os = library("oscillators.lib");
tf2np_test = os.osc(440) : fi.tf2np(0.6, 0.3, 0.2, -0.5, 0.2);
```

`(fi.)wgr`

Second-order transformer-normalized digital waveguide resonator.

Usage

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

Where:

- `f`: resonance frequency (Hz)

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

Test

```
fi = library("filters.lib");
os = library("oscillators.lib");
wgr_test = fi.wgr(440, 0.995, os.osc(440));
```

References

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

---

`(fi.)nlf2`

Second order normalized digital waveguide resonator.

Usage

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

Where:

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

Test

```
fi = library("filters.lib");
os = library("oscillators.lib");
nlf2_test = fi.nlf2(440, 0.995, os.osc(440));
```

References

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

---

`(fi.)apnl`

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

Usage

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

Where:

- `a1` and `a2`: allpass coefficients

Test

```
fi = library("filters.lib");
os = library("oscillators.lib");
apnl_test = fi.apnl(0.5, -0.5, os.osc(440));
```

References

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

Ladder/Lattice Allpass Filters

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

References

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

`(fi.)scatN`

N-port scattering junction.

Usage

```
si.bus(N) : scatN(N,av,filter) : si.bus(N)
```

Where:

- `N`: number of incoming/outgoing waves
- `av`: vector (list) of `N` alpha parameters (each between 0 and 2, and normally summing to 2): https://ccrma.stanford.edu/~jos/pasp/Alpha_Parameters.html
- `filter` : optional junction filter to apply (`_` for none, see below)

With no filter:

- The junction is *lossless* when the alpha parameters sum to 2 (“allpass”).
- The junction is *passive* but lossy when the alpha parameters sum to less than 2 (“resistive loss”).
- Dynamic and reactive junctions are obtained using the `filter` argument. For guaranteed stability, the filter should be *positive real*. (See 2nd ref. below).

For (N=2) (two-port scattering), the reflection coefficient ( ) corresponds to alpha parameters (1±).

Example: Whacky echo chamber made of 16 lossless “acoustic tubes”:

```
process = _ : *(1.0/sqrt(N)) <: daisyRev(16,2,0.9999) :> _,_ with {
  daisyRev(N,Dp2,G) = si.bus(N) : (si.bus(2*N) :> si.bus(N)
    : fi.scatN(N, par(i,N,2*G/float(N)), fi.lowpass(1,5000.0))
    : par(i,N,de.delay(DS(i),DS(i)-1))) ~ si.bus(N) with { DS(i) = 2^(Dp2+i); };
};
```

Test

```
fi = library("filters.lib");
os = library("oscillators.lib");
scatN_test = (os.osc(440), os.osc(660)) : fi.scatN(2, (1, 1), _);
```

References

- https://ccrma.stanford.edu/~jos/pasp/Loaded_Waveguide_Junctions.html
- https://ccrma.stanford.edu/~jos/pasp/Passive_String_Terminations.html
- https://ccrma.stanford.edu/~jos/pasp/Unloaded_Junctions_Alpha_Parameters.html

---

`(fi.)scat`

Scatter off of reflectance r with reflection coefficient s.

Usage:

```
_ : scat(s,r) : _
```

Where:

- `s`: reflection coefficient between -1 and 1 for stability
- `r`: single-input, single-output block diagram, having gain less than 1 at all frequencies for stability.

Example: the following program should produce all zeros:

```
process = fi.allpassn(3,(.3,.2,.1)), fi.scat(.1, fi.scat(.2, fi.scat(.3, _)))
          :> - : ^(2) : +~_;
```

Test

```
fi = library("filters.lib");
os = library("oscillators.lib");
scat_test = os.osc(440) : fi.scat(0.5, _);
```

References

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

---

(fi.)allpassn

Two-multiply lattice filter.

Usage:

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

Where:

- `n`: the order of the filter
- `sv`: the reflection coefficients (-1 1)
- `sv`: the reflection coefficients (s1,s2,…,sN), each between -1 and 1.

Equivalent to `fi.allpassnt(n,sv) : _, par(i,n,!);`

Equivalent to `fi.scat( s(n), fi.scat( s(n-1), ..., fi.scat( s(1), _ )))               with { s(k) = ba.take(k,sv); } ;`

Identical to `allpassn` in `old/filter.lib`.

Test

```
fi = library("filters.lib");
os = library("oscillators.lib");
allpassn_test = os.osc(440) : fi.allpassn(3, (0.3, 0.2, 0.1));
```

References

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

(fi.)allpassnn

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

Usage:

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

Where:

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

Test

```
fi = library("filters.lib");
os = library("oscillators.lib");
allpassnn_test = os.osc(440) : fi.allpassnn(3, (0.3, 0.2, 0.1));
```

(fi.)allpassnkl

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

Usage:

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

Where:

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

Test

```
fi = library("filters.lib");
os = library("oscillators.lib");
allpassnkl_test = os.osc(440) : fi.allpassnkl(3, (0.3, 0.2, 0.1));
```

(fi.)allpass1m

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

Usage:

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

Where:

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

Test

```
fi = library("filters.lib");
os = library("oscillators.lib");
allpassn1m_test = os.osc(440) : fi.allpassn1m(3, (0.3, 0.2, 0.1));
```

Digital Filter Sections Specified as Analog Filter Sections

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

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

Usage

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

Where:

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

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

Example test program 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.

Test

```
fi = library("filters.lib");
os = library("oscillators.lib");
ma = library("maths.lib");
tf2s_test = os.osc(440) : fi.tf2s(0, 0, 1, sqrt(2), 1, ma.PI*ma.SR/2);
```

References

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

---

(fi.)tf1snp

First-order special case of tf2snp above.

Usage

```
_ : tf1snp(b1,b0,a0) : _
```

Test

```
fi = library("filters.lib");
os = library("oscillators.lib");
ma = library("maths.lib");
tf1snp_test = os.osc(440) : fi.tf1snp(0, 1, 1, ma.PI*ma.SR/2);
```

(fi.)tf3slf

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

Usage

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

Test

```
fi = library("filters.lib");
os = library("oscillators.lib");
tf3slf_test = os.osc(440) : fi.tf3slf(0, 0, 0, 1, 1, 2, 2, 1);
```

`(fi.)tf1s`

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

Usage

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

Where:

```
b1 s + b0
```

H(s) = ——— s + a0

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

Example test program 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.

Test

```
fi = library("filters.lib");
os = library("oscillators.lib");
ma = library("maths.lib");
tf1s_test = os.osc(440) : fi.tf1s(0, 1, 1, ma.PI*ma.SR/2);
```

References

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

---

`(fi.)tf2sb`

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

Usage

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

Test

```
fi = library("filters.lib");
os = library("oscillators.lib");
ma = library("maths.lib");
tf2sb_test = os.osc(440) : fi.tf2sb(0, 0, 1, sqrt(2), 1, 2*ma.PI*200, 2*ma.PI*1000);
```

`(fi.)tf1sb`

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

Usage

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

Test

```
fi = library("filters.lib");
os = library("oscillators.lib");
ma = library("maths.lib");
tf1sb_test = os.osc(440) : fi.tf1sb(0, 1, 1, 2*ma.PI*200, 2*ma.PI*1000);
```

Simple Resonator Filters

`(fi.)resonlp`

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

Usage

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

Where:

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

Test

```
fi = library("filters.lib");
os = library("oscillators.lib");
resonlp_test = os.osc(440) : fi.resonlp(1000, 2, 0.8);
```

(fi.)resonhp

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

Usage

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

Where:

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

Test

```
fi = library("filters.lib");
os = library("oscillators.lib");
resonhp_test = fi.resonhp(1000, 2, 0.8, os.osc(440));
```

(fi.)resonbp

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

Usage

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

Where:

- `fc`: center frequency (Hz)

- `Q`: q
- `gain`: gain (0-1)

Test

```
fi = library("filters.lib");
os = library("oscillators.lib");
resonbp_test = os.osc(440) : fi.resonbp(1000, 2, 0.8);
```

Butterworth Lowpass/Highpass Filters

(fi.)lowpass

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

Usage

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

Where:

- `N`: filter order (number of poles), nonnegative constant numerical expression
- `fc`: desired cut-off frequency (-3dB frequency) in Hz

Test

```
fi = library("filters.lib");
os = library("oscillators.lib");
lowpass_test = os.osc(440) : fi.lowpass(4, 2000);
```

References

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

(fi.)highpass

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

Usage

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

Where:

- `N`: filter order (number of poles), nonnegative constant numerical expression
- `fc`: desired cut-off frequency (-3dB frequency) in Hz

Test

```
fi = library("filters.lib");
os = library("oscillators.lib");
highpass_test = os.osc(440) : fi.highpass(4, 500);
```

References

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

(fi.)lowpass0_highpass1

Test

```
fi = library("filters.lib");
os = library("oscillators.lib");
lowpass0_highpass1_test = os.osc(440) : fi.lowpass0_highpass1(0, 2, 1000);
```

Special Filter-Bank Delay-Equalizing Allpass Filters

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

(fi.)highpass_plus_lowpass

Test

```
fi = library("filters.lib");
os = library("oscillators.lib");
highpass_plus_lowpass_test = os.osc(440) : fi.highpass_plus_lowpass(3, 1000);
```

(fi.)highpass_minus_lowpass

Test

```
fi = library("filters.lib");
os = library("oscillators.lib");
highpass_minus_lowpass_test = os.osc(440) : fi.highpass_minus_lowpass(3, 1000);
```

`(fi.)highpass_plus_lowpass_even`

Test

```
fi = library("filters.lib");
os = library("oscillators.lib");
highpass_plus_lowpass_even_test = os.osc(440) : fi.highpass_plus_lowpass_even(4, 1000);
```

`(fi.)highpass_plus_lowpass_even`

Test

```
fi = library("filters.lib");
os = library("oscillators.lib");
highpass_minus_lowpass_even_test = os.osc(440) : fi.highpass_minus_lowpass_even(4, 1000);
```

`(fi.)highpass_minus_lowpass_odd`

Test

```
fi = library("filters.lib");
os = library("oscillators.lib");
highpass_plus_lowpass_odd_test = os.osc(440) : fi.highpass_plus_lowpass_odd(3, 1000);
```

FIXME: Rewrite the following, as for orders 3 and 5 above, to eliminate pole-zero cancellations:

`(fi.)highpass_minus_lowpass_odd`

Test

```
fi = library("filters.lib");
os = library("oscillators.lib");
highpass_minus_lowpass_odd_test = os.osc(440) : fi.highpass_minus_lowpass_odd(3, 1000);
```

FIXME: Rewrite the following, as for orders 3 and 5 above, to eliminate pole-zero cancellations/

Elliptic (Cauer) Lowpass Filters

Elliptic (Cauer) Lowpass Filters

References

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

(fi.)lowpass3e

Third-order Elliptic (Cauer) lowpass filter.

Usage

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

Where:

- `fc`: -3dB frequency in Hz

Test

```
fi = library("filters.lib");
os = library("oscillators.lib");
lowpass3e_test = os.osc(440) : fi.lowpass3e(1000);
```

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

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

(fi.)lowpass6e

Sixth-order Elliptic/Cauer lowpass filter.

Usage

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

Where:

- `fc`: -3dB frequency in Hz

Test

```
fi = library("filters.lib");
os = library("oscillators.lib");
lowpass6e_test = os.osc(440) : fi.lowpass6e(1000);
```

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

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

Elliptic Highpass Filters

(fi.)highpass3e

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

Usage

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

Where:

- `fc`: -3dB frequency in Hz

Test

```
fi = library("filters.lib");
os = library("oscillators.lib");
highpass3e_test = os.osc(440) : fi.highpass3e(1000);
```

(fi.)highpass6e

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

Usage

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

Where:

- `fc`: -3dB frequency in Hz

Test

```
fi = library("filters.lib");
os = library("oscillators.lib");
highpass6e_test = os.osc(440) : fi.highpass6e(1000);
```

Butterworth Bandpass/Bandstop Filters

`(fi.)bandpass`

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

Usage

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

Where:

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

Test

```
fi = library("filters.lib");
os = library("oscillators.lib");
bandpass_test = os.osc(440) : fi.bandpass(2, 500, 1500);
```

`(fi.)bandstop`

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

Usage

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

Where:

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

Test

```
fi = library("filters.lib");
os = library("oscillators.lib");
bandstop_test = os.osc(440) : fi.bandstop(2, 500, 1500);
```

(fi.)bandstop

Test

```
fi = library("filters.lib");
os = library("oscillators.lib");
bandpass0_bandstop1_test = os.osc(440) : fi.bandpass0_bandstop1(0, 2, 500, 1500);
```

Elliptic Bandpass Filters

(fi.)bandpass6e

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

(fi.)bandpass12e

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

(fi.)pospass

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

Usage

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

where

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

Example test program

- See `dm.pospass_demo`
- Look at frequency response

Method A filter passing only positive frequencies can be made from a half-band lowpass by modulating it up to the positive-frequency range. Equivalently, down-modulate the input signal using a complex sinusoid at -SR/4 Hz, lowpass it with a half-band filter, and modulate back up by SR/4 Hz. In Faust/math notation:

$$pospass(N) = *(e^{-j\frac{\pi}{2}n}) : \text{lowpass(N,SR/4)} : *(e^{j\frac{\pi}{2}n})$$

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

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

References

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

Parametric Equalizers (Shelf, Peaking)

Parametric Equalizers (Shelf, Peaking).

References

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

`(fi.)lowshelf`

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

Usage

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

Where:

- `N`: filter order 1, 3, 5, … (odd only, default should be 3, a constant numerical expression)
- `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).

Test

```
fi = library("filters.lib");
os = library("oscillators.lib");
lowshelf_test = os.osc(440) : fi.lowshelf(3, 6, 500);
```

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`.

References

- See "Parametric Equalizers" above for references regarding `low_shelf`, `high_shelf`, and `peak_eq`.

`(fi.)low_shelf`

Test

```
fi = library("filters.lib");
os = library("oscillators.lib");
low_shelf_test = os.osc(440) : fi.low_shelf(6, 500);
```

(fi.)low_shelf1_l

Test

```
fi = library("filters.lib");
os = library("oscillators.lib");
low_shelf1_l_test = fi.low_shelf1_l(2, 500, os.osc(440));
```

(fi.)low_shelf1_l

Test

```
fi = library("filters.lib");
os = library("oscillators.lib");
low_shelf1_l_test = fi.low_shelf1_l(2, 500, os.osc(440));
```

(fi.)lowshelf_other_freq

Test

```
fi = library("filters.lib");
lowshelf_other_freq_test = fi.lowshelf_other_freq(3, 6, 500);
```

(fi.)high_shelf

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

Usage

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

Where:

- `N`: filter order 1, 3, 5, … (odd only, a constant numerical expression).
- `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.

Test

```
fi = library("filters.lib");
os = library("oscillators.lib");
highshelf_test = os.osc(440) : fi.highshelf(3, 6, 2000);
```

References

- See "Parametric Equalizers" above for references regarding `low_shelf`, `high_shelf`, and `peak_eq`.

(fi.)high_shelf

Test

```
fi = library("filters.lib");
os = library("oscillators.lib");
high_shelf_test = os.osc(440) : fi.high_shelf(6, 2000);
```

(fi.)high_shelf1

Test

```
fi = library("filters.lib");
os = library("oscillators.lib");
high_shelf1_test = fi.high_shelf1(6, 2000, os.osc(440));
```

(fi.)high_shelf1_l

Test

```
fi = library("filters.lib");
os = library("oscillators.lib");
high_shelf1_l_test = fi.high_shelf1_l(2, 2000, os.osc(440));
```

(fi.)highshelf_other_freq

Test

```
fi = library("filters.lib");
highshelf_other_freq_test = fi.highshelf_other_freq(3, 6, 2000);
```

(fi.)peak_eq

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

Usage

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

Where:

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

Test

```
fi = library("filters.lib");
os = library("oscillators.lib");
peak_eq_test = os.osc(440) : fi.peak_eq(6, 1000, 200);
```

References

- See "Parametric Equalizers" above for references regarding `low_shelf`, `high_shelf`, and `peak_eq`.

(fi.)peak_eq_cq

Constant-Q second order peaking equalizer section.

Usage

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

Where:

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

Test

```
fi = library("filters.lib");
os = library("oscillators.lib");
peak_eq_cq_test = os.osc(440) : fi.peak_eq_cq(6, 1000, 4);
```

References

- See “Parametric Equalizers” above for references regarding `low_shelf`, `high_shelf`, and `peak_eq`.

(fi.)peak_eq_rm

Regalia-Mitra second order peaking equalizer section.

Usage

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

Where:

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

Test

```
fi = library("filters.lib");
os = library("oscillators.lib");
ma = library("maths.lib");
peak_eq_rm_test = os.osc(440) : fi.peak_eq_rm(6, 1000, ma.tan(ma.PI*200/ma.SR));
```

References 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.) See also “Parametric Equalizers” above for references on shelf and peaking equalizers in general.

(fi.)spectral_tilt

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

Usage

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

Where:

- `N`: desired integer filter order (fixed at compile time)

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

Test

```
fi = library("filters.lib");
os = library("oscillators.lib");
spectral_tilt_test = os.osc(440) : fi.spectral_tilt(4, 200, 2000, -0.5);
```

Example test program See `dm.spectral_tilt_demo` and the documentation for `no.pink_noise`.

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

`(fi.)levelfilter`

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

Usage

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

Where:

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

Test

```
fi = library("filters.lib");
os = library("oscillators.lib");
levelfilter_test = fi.levelfilter(0.1, 200, os.osc(440));
```

References

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

---

(fi.)levelfilterN

Dynamic level lowpass filter.

Usage

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

Where:

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

Test

```
fi = library("filters.lib");
os = library("oscillators.lib");
levelfilterN_test = os.osc(440) : fi.levelfilterN(3, 200, 0.1);
```

References

- 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), a constant numerical expression
- `N`: total number of bands (>2), a constant numerical expression
- `ftop`: upper bandlimit of the Mth-octave bands (<SR/2)

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

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

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

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

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

References

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

(fi.)mth_octave_filterbank[n]

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

Usage

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

Also for convenience:

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

Where:

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

Test

```
fi = library("filters.lib");
os = library("oscillators.lib");
```

```
mth_octave_filterbank_test = os.osc(440) : fi.mth_octave_filterbank(3, 2, 8000, 2);
```

(fi.)mth_octave_filterbank_alt

Test

```
fi = library("filters.lib");
os = library("oscillators.lib");
mth_octave_filterbank_alt_test = os.osc(440) : fi.mth_octave_filterbank_alt(3, 2, 8000, 2);
```

(fi.)mth_octave_filterbank3

Test

```
fi = library("filters.lib");
os = library("oscillators.lib");
mth_octave_filterbank3_test = os.osc(440) : fi.mth_octave_filterbank3(2, 8000, 2);
```

(fi.)mth_octave_filterbank5

Test

```
fi = library("filters.lib");
os = library("oscillators.lib");
mth_octave_filterbank5_test = os.osc(440) : fi.mth_octave_filterbank5(2, 8000, 2);
```

(fi.)mth_octave_filterbank_default

Test

```
fi = library("filters.lib");
os = library("oscillators.lib");
mth_octave_filterbank_default_test = os.osc(440) : fi.mth_octave_filterbank_default(2, 8000,
```

Arbitrary-Crossover Filter-Banks and Spectrum Analyzers

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

(fi.)filterbank

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

Usage

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

Where:

- `O`: band-split filter order (odd integer required for filterbank[i], a constant numerical expression)
- `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)) : _,_,_
```

Test

```
fi = library("filters.lib");
os = library("oscillators.lib");
filterbank_test = os.osc(440) : fi.filterbank(3, (500, 2000));
```

(fi.)filterbanki

Inverted-dc filter bank.

Usage

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

Where:

- `O`: band-split filter order (odd integer required for `filterbank[i]`, a constant numerical expression)
- `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)) : _,_,_
```

Test

```
fi = library("filters.lib");
os = library("oscillators.lib");
filterbanki_test = os.osc(440) : fi.filterbanki(3, (500, 2000));
```

State Variable Filters

References

- Solving the continuous SVF equations using trapezoidal integration

- https://cytomic.com/files/dsp/SvfLinearTrapOptimised2.pdf

(fi.)svf

An environment with `lp`, `bp`, `hp`, `notch`, `peak`, `ap`, `bell`, `ls`, `hs` SVF based filters. All filters have `freq` and `Q` parameters, the `bell`, `ls`, `hs` ones also have a `gain` third parameter.

Usage

```
_ : svf.xx(freq, Q, [gain]) : _
```

Where:

- `freq`: cut frequency
- `Q`: quality factor
- `[gain]`: gain in dB

Test

```
fi = library("filters.lib");
os = library("oscillators.lib");
svf_lp_test = fi.svf.lp(1000, 0.707, os.osc(440));
svf_bp_test = fi.svf.bp(1000, 0.707, os.osc(440));
svf_hp_test = fi.svf.hp(1000, 0.707, os.osc(440));
svf_notch_test = fi.svf.notch(1000, 0.707, os.osc(440));
svf_peak_test = fi.svf.peak(1000, 0.707, os.osc(440));
svf_ap_test = fi.svf.ap(1000, 0.707, os.osc(440));
svf_bell_test = fi.svf.bell(1000, 0.707, 6, os.osc(440));
svf_ls_test = fi.svf.ls(500, 0.707, 6, os.osc(440));
svf_hs_test = fi.svf.hs(3000, 0.707, 6, os.osc(440));
```

(fi.)svf_morph

An SVF-based filter that can smoothly morph between being lowpass, bandpass, and highpass.

Usage

```
_ : svf_morph(freq, Q, blend) : _
```

Where:

- `freq`: cutoff frequency
- `Q`: quality factor

- `blend`: [0..2] continuous, where 0 is `lowpass`, 1 is `bandpass`, and 2 is `highpass`. For performance, the value is not clamped to [0..2].

Test

```
fi = library("filters.lib");
os = library("oscillators.lib");
svf_morph_test = fi.svf_morph(1000, 0.707, 1, os.osc(440));
```

Example test program

```
process = no.noise : svf_morph(freq, q, blend)
with {
  blend = hslider("Blend", 0, 0, 2, .01) : si.smoo;
  q = hslider("Q", 1, 0.1, 10, .01) : si.smoo;
  freq = hslider("freq", 5000, 100, 18000, 1) : si.smoo;
};
```

References

- https://github.com/mtytel/vital/blob/636ca0ef517a4db087a6a08a6a8a5e704e21f836/src/synthesis/filters/digital_svf.cpp#L292-L295

`(fi.)svf_notch_morph`

An SVF-based notch-filter that can smoothly morph between being lowpass, notch, and highpass.

Usage

```
_ : svf_notch_morph(freq, Q, blend) : _
```

Where:

- `freq`: cutoff frequency
- `Q`: quality factor
- `blend`: [0..2] continuous, where 0 is `lowpass`, 1 is `notch`, and 2 is `highpass`. For performance, the value is not clamped to [0..2].

Test

```
fi = library("filters.lib");
os = library("oscillators.lib");
svf_notch_morph_test = fi.svf_notch_morph(1000, 0.707, 1, os.osc(440));
```

Example test program

```
process = no.noise : svf_notch_morph(freq, q, blend)
with {
  blend = hslider("Blend", 0, 0, 2, .01) : si.smoo;
  q = hslider("Q", 1, 0.1, 10, .01) : si.smoo;
  freq = hslider("freq", 5000, 100, 18000, 1) : si.smoo;
};
```

References

- https://github.com/mtytel/vital/blob/636ca0ef517a4db087a6a08a6a8a5e704e21f836/src/synthesis/filters/digital_svf.cpp#L256C36-L263

`(fi.)SVFTPT`

Topology-preserving transform implementation following Zavalishin's method.

Outputs: lowpass, highpass, bandpass, normalised bandpass, notch, allpass, peaking.

Each individual output can be recalled with its name in the environment as in: `SVFTPT.LP2(1000.0, .707)`.

The 7 outputs can be recalled by using `SVF` name as in: `SVFTPT.SVF(1000.0, .707)`.

Even though the implementation is different, the characteristics of this filter are comparable to those of the `svf` environment in this library.

Usage:

```
_ : SVFTPT.xxx(CF, Q) : _
```

Where:

- `xxx` can be one of the following: `LP2`, `HP2`, `BP2`, `BP2Norm`, `Notch2`, `AP2`, `Peaking2`
- `CF`: cutoff in Hz
- `Q`: resonance

Test

```
fi = library("filters.lib");
os = library("oscillators.lib");
SVFTPT_SVF_test = fi.SVFTPT.SVF(1000, 0.707, os.osc(440));
SVFTPT_LP2_test = fi.SVFTPT.LP2(1000, 0.707, os.osc(440));
SVFTPT_HP2_test = fi.SVFTPT.HP2(1000, 0.707, os.osc(440));
SVFTPT_BP2_test = fi.SVFTPT.BP2(1000, 0.707, os.osc(440));
```

```
SVFTPT_BP2Norm_test = fi.SVFTPT.BP2Norm(1000, 0.707, os.osc(440));
SVFTPT_Notch2_test = fi.SVFTPT.Notch2(1000, 0.707, os.osc(440));
SVFTPT_AP2_test = fi.SVFTPT.AP2(1000, 0.707, os.osc(440));
SVFTPT_Peaking2_test = fi.SVFTPT.Peaking2(1000, 0.707, os.osc(440));
```

`(fi.)dynamicSmoothing`

Adaptive smoother based on Andy Simper's paper.

This filter uses both the lowpass and bandpass outputs of a state-variable filter. The lowpass is used to smooth out the input signal, the bandpass, which is a smoothed out version of the highpass, provides information on the rate of change of the input. Hence, the bandpass signal can be used to adjust the cutoff of the filter to quickly follow the input's fast and large variations while effectively filtering out local perturbations.

This implementation does not use an approximation for the CF computation, and it deploys guards to prevent overshooting with extreme sensitivity values.

Usage:

```
_ : dynamicSmoothing(sensitivity, baseCF) : _
```

Where:

- `sensitivity`: sensitivity to changes in the input signal. The range is, theoretically, from 0 to INF, though anything between 0.0 and 1.0 should be reasonable
- `baseCF`: cutoff frequency, in Hz, when there is no variation in the input signal

Test

```
fi = library("filters.lib");
os = library("oscillators.lib");
dynamicSmoothing_test = fi.dynamicSmoothing(0.5, 500, os.osc(440));
```

References

- https://cytomic.com/files/dsp/DynamicSmoothing.pdf

`(fi.)oneEuro`

The One Euro Filter (1€ Filter) is an adaptive lowpass filter. This kind of filter is commonly used in object-tracking, not necessarily audio processing.

Usage

```
_ : oneEuro(derivativeCutoff, beta, minCutoff) : _
```

Where:

- `derivativeCutoff`: Used to filter the first derivative of the input. 1 Hz is a good default.
- `beta`: "Speed" parameter where higher values reduce latency.
- `minCutoff`: Minimum cutoff frequency in Hz. Lower values remove more jitter.

Test

```
fi = library("filters.lib");
os = library("oscillators.lib");
oneEuro_test = os.osc(440) : fi.oneEuro(1, 0.5, 5);
```

References

- https://gery.casiez.net/1euro/

Linkwitz-Riley 4th-order 2-way, 3-way, and 4-way crossovers

The Linkwitz-Riley (LR) crossovers are designed to produce a fully-flat magnitude response when their outputs are combined. The 4th-order LR filters (LR4) have a 24dB/octave slope and they are rather popular audio crossovers used in multi-band processing.

The LR4 can be constructed by cascading two second-order Butterworth filters. For the second-order Butterworth filters, we will use the SVF filter implemented above by setting the Q-factor to 1.0 / sqrt(2.0). These will be cascaded in pairs to build the LR4 highpass and lowpass. For the phase correction, we will use the 2nd-order Butterworth allpass.

References Zavalishin, Vadim. "The art of VA filter design." Native Instruments, Berlin, Germany (2012).

`(fi.)lowpassLR4`

4th-order Linkwitz-Riley lowpass.

Usage

```
_ : lowpassLR4(cf) : _
```

Where:

- `cf` is the lowpass cutoff in Hz

Test

```
fi = library("filters.lib");
os = library("oscillators.lib");
lowpassLR4_test = os.osc(440) : fi.lowpassLR4(1000);
```

`(fi.)highpassLR4`

4th-order Linkwitz-Riley highpass.

Usage

```
_ : highpassLR4(cf) : _
```

Where:

- `cf` is the highpass cutoff in Hz

Test

```
fi = library("filters.lib");
os = library("oscillators.lib");
highpassLR4_test = os.osc(440) : fi.highpassLR4(1000);
```

`(fi.)crossover2LR4`

Two-way 4th-order Linkwitz-Riley crossover.

Usage

```
_ : crossover2LR4(cf) : si.bus(2)
```

Where:

- `cf` is the crossover split cutoff in Hz

Test

```
fi = library("filters.lib");
os = library("oscillators.lib");
crossover2LR4_test = os.osc(440) : fi.crossover2LR4(1000);
```

(fi.)crossover3LR4

Three-way 4th-order Linkwitz-Riley crossover.

Usage

```
_ : crossover3LR4(cf1, cf2) : si.bus(3)
```

Where:

- `cf1` is the crossover lower split cutoff in Hz
- `cf2` is the crossover upper split cutoff in Hz

Test

```
fi = library("filters.lib");
os = library("oscillators.lib");
crossover3LR4_test = os.osc(440) : fi.crossover3LR4(500, 2000);
```

(fi.)crossover4LR4

Four-way 4th-order Linkwitz-Riley crossover.

Usage

```
_ : crossover4LR4(cf1, cf2, cf3) : si.bus(4)
```

Where:

- `cf1` is the crossover lower split cutoff in Hz
- `cf2` is the crossover mid split cutoff in Hz
- `cf3` is the crossover upper split cutoff in Hz

Test

```
fi = library("filters.lib");
os = library("oscillators.lib");
crossover4LR4_test = os.osc(440) : fi.crossover4LR4(300, 1000, 3000);
```

(fi.)crossover8LR4

Eight-way 4th-order Linkwitz-Riley crossover.

Usage

```
_ : crossover8LR4(cf1, cf2, cf3, cf4, cf5, cf6, cf7) : si.bus(8)
```

Where:

- `cf1-cf7` are the crossover cutoff frequencies in Hz

Test

```
fi = library("filters.lib");
os = library("oscillators.lib");
crossover8LR4_test = os.osc(440) : fi.crossover8LR4(100, 200, 400, 800, 1600, 3200, 6400);
```

Standardized Filters

`(fi.)itu_r_bs_1770_4_kfilter`

The prefilter from Recommendation ITU-R BS.1770-4 for loudness measurement. Also known as "K-filter". The recommendation defines biquad filter coefficients for a fixed sample rate of 48kHz (page 4-5). Here, we construct biquads for arbitrary samplerates. The resulting filter is normalized, such that the magnitude at 997Hz is unity gain 1.0.

Please note, the ITU-recommendation handles the normalization in equation (2) by subtracting 0.691dB, which is not needed with `itu_r_bs_1770_4_kfilter`.

One option for future improvement might be, to round those filter coefficients, that are almost equal to one. Second, the maximum magnitude difference at 48kHz between the ITU-defined filter and `itu_r_bs_1770_4_kfilter` is 0.001dB, which obviously could be less.

Usage

```
_ : itu_r_bs_1770_4_kfilter : _
```

Test

```
fi = library("filters.lib");
os = library("oscillators.lib");
itu_r_bs_1770_4_kfilter_test = os.osc(440) : fi.itu_r_bs_1770_4_kfilter;
```

References

- https://www.itu.int/rec/R-REC-BS.1770
- https://gist.github.com/jkbd/07521a98f7873a2dc3dbe16417930791

Averaging Functions

(fi.)avg_rect

Moving average.

Usage

```
_ : avg_rect(period) : _
```

Where:

- `period` is the averaging frame in seconds

(fi.)avg_tau

Averaging function based on a one-pole filter and the tau response time. Tau represents the effective length of the one-pole impulse response, that is, tau is the integral of the filter's impulse response. This response is slower to reach the final value but has less ripples in non-steady signals.

Usage

```
_ : avg_tau(period) : _
```

Where:

- `period` is the time, in seconds, for the system to decay by 1/e, or to reach 1-1/e of its final value.

References

- https://ccrma.stanford.edu/~jos/mdft/Exponentials.html

(fi.)avg_t60

Averaging function based on a one-pole filter and the t60 response time. This response is particularly useful when the system is required to reach the final value after about `period` seconds.

Usage

```
_ : avg_t60(period) : _
```

Where:

- `period` is the time, in seconds, for the system to decay by 1/1000, or to reach 1-1/1000 of its final value.

References

- https://ccrma.stanford.edu/~jos/mdft/Audio_Decay_Time_T60.html

---

`(fi.)avg_t19`

Averaging function based on a one-pole filter and the t19 response time. This response is close to the moving-average algorithm as it roughly reaches the final value after `period` seconds and shows about the same oscillations for non-steady signals.

Usage

```
_ : avg_t19(period) : _
```

Where:

- `period` is the time, in seconds, for the system to decay by 1/e^2.2, or to reach 1-1/e^2.2 of its final value.

References Zölzer, U. (2008). Digital audio signal processing (Vol. 9). New York: Wiley.

Kalman Filters

`(fi.)kalman`

The Kalman filter. It returns the state (a bus of size `N`). Note that the only compile-time constant arguments are `N` and `M`. Other arguments are capitalized because they're matrices, and it makes reading them much easier.

Usage

```
kalman(N, M, B, R, H, Q, F, reset, u, z) : si.bus(N)
```

Where:

- `N`: State size (constant int)
- `M`: Measurement size (constant int)
- `B`: Control input matrix (NxM)
- `R`: Measurement noise covariance matrix (MxM)
- `H`: Observation matrix (MxN)
- `Q`: Process noise covariance matrix (NxN)

- `F`: State transition matrix (NxN)
- `reset`: Reset trigger. Whenever `reset>0`, the internal state `x` and covariance matrix `P` are reset.
- `u`: Control input (Mx1)
- `z`: Measurement signal (Mx1)

Example test programs Demo 1 `(N=1, M=1)` (don't listen, just use oscilloscope):

```
process = fi.kalman(N, M, B, R, H, Q, F, reset, u, z) : it.interpolate_linear(filteredAmt, z
with {
   B = 1.;
   R = 0.1;
   H = 1;
   Q = .01;
   F = la.identity(N);
   reset = button("reset");

   // Dimensions
   N = 1; // State size
   M = 1; // Measurement size

   freq = hslider("Freq", 1, 0.01, 10, .01);
   u = 0.; // constant input
   trueState = os.osc(freq)*.5 + u;
   noiseGain = hslider("Noise Gain", .1, 0, 1, .01);

   filteredAmt = hslider("Filter Amount", 1, 0, 1, .01) : si.smoo;

   measurementNoise = no.noise*noiseGain;
   z = trueState + measurementNoise; // Observed state
};
```

Demo 2 `(N=2, M=1)` (don't listen, just use oscilloscope)

```
process = fi.kalman(N, M, B, R, H, Q, F, reset, u, z)
with {
    B = par(i, N, 0);
    R = (0.1);
    H = (1, 0);
    Q = la.diag(2, par(i, N, .1));
    F = la.identity(N);
    reset = 0;
    u = si.bus(M);
    z = si.bus(M);

    // Dimensions
```

```
    N = 2; // State size
    M = 1; // Measurement size
};
```

References

- https://en.wikipedia.org/wiki/Kalman_filter
- https://www.cs.unc.edu/~welch/kalman/index.html

hoa.lib

Higher-Order Ambisonics (HOA) library. Its official prefix is `ho`.

The HOA library provides functions and components for spatial audio rendering and analysis using Higher-Order Ambisonics. It includes encoders, decoders, rotators, and utilities for spherical harmonics and spatial transformations. The library supports both 2D and 3D HOA processing workflows for immersive audio.

The HOA library is organized into 4 sections:

- Encoding/decoding Functions
- Optimization Functions
- Spatial Sound Processes
- 3D Functions

References

- https://github.com/grame-cncm/faustlibraries/blob/master/hoa.lib

Encoding/decoding Functions

`(ho.)encoder`

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

Usage

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

Where:

- `N`: the ambisonic order (constant numerical expression)
- `x`: the signal
- `a`: the angle

Test

```
ho = library("hoa.lib");
os = library("oscillators.lib");
encoder_test = ho.encoder(1, os.osc(440), 0.0);
```

(ho.)rEncoder

Ambisonic encoder in 2D including source rotation. A mono signal is encoded at a certain ambisonic order with two possible modes: either rotation with an angular speed, or static with a fixed angle (when speed is zero).

Usage

```
_ : rEncoder(N, sp, a, it) : _,_, ...
```

Where:

- `N`: the ambisonic order (constant numerical expression)
- `sp`: the azimuth speed expressed as angular speed (2PI/sec), positive or negative
- `a`: the fixed azimuth when the rotation stops (sp = 0) in radians
- `it` : interpolation time (in milliseconds) between the rotation and the fixed modes

Test

```
ho = library("hoa.lib");
os = library("oscillators.lib");
rEncoder_test = os.osc(440) : ho.rEncoder(1, 0.5, 0.0, 0.05);
```

(ho.)stereoEncoder

Encoding of a stereo pair of channels with symetric angles (a/2, -a/2).

Usage

```
_,_ : stereoEncoder(N, a) : _,_, ...
```

Where:

- `N`: the ambisonic order (constant numerical expression)
- `a` : opening angle in radians, left channel at a/2 angle, right channel at -a/2 angle

Test

```
ho = library("hoa.lib");
os = library("oscillators.lib");
stereoEncoder_test = os.osc(440), os.osc(660) : ho.stereoEncoder(1, 1.0);
```

(ho.)multiEncoder

Encoding of a set of P signals distributed on the unit circle according to a list of P speeds and P angles.

Usage

```
_,_, ... : multiEncoder(N, lspeed, langle, it) : _,_, ...
```

Where:

- `N`: the ambisonic order (constant numerical expression)
- `lspeed` : a list of P speeds in turns by second (one speed per input signal, positive or negative)
- `langle` : a list of P angles in radians on the unit circle to localize the sources (one angle per input signal)
- `it` : interpolation time (in milliseconds) between the rotation and the fixed modes.

Test

```
ho = library("hoa.lib");
os = library("oscillators.lib");
multiEncoder_test = os.osc(440), os.osc(660) : ho.multiEncoder(1, (0.0, 0.0), (0.0, 1.57), (
```

(ho.)decoder

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

Usage

```
_ : decoder(N, P) : _
```

Where:

- `N`: the ambisonic order (constant numerical expression)
- `P`: the number of speakers (constant numerical expression)

Test

```
ho = library("hoa.lib");
os = library("oscillators.lib");
ambi = ho.encoder(1, os.osc(440), 0.0);
decoder_test = ambi : ho.decoder(1, 4);
```

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

`(ho.)decoderStereo`

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

```
_ : decoderStereo(N) : _
```

Where:

- `N`: the ambisonic order (constant numerical expression)

Test

```
ho = library("hoa.lib");
os = library("oscillators.lib");
ambi = ho.encoder(1, os.osc(440), 0.0);
decoderStereo_test = ambi : ho.decoderStereo(1);
```

`(ho.)iBasicDecoder`

The irregular basic decoder is a simple decoder that projects the incoming ambisonic situation to the loudspeaker situation (P loudspeakers) whatever it is, without compensation. When there is a strong irregularity, there can be some discontinuity in the sound field.

Usage

```
_,_, ... : iBasicDecoder(N,la, direct, shift) : _,_, ...
```

Where:

- `N`: the ambisonic order (there are 2*N+1 inputs to this function)
- `la` : the list of P angles in degrees, for instance (0, 85, 182, 263) for four loudspeakers

- `direct`: 1 for direct mode, -1 for the indirect mode (changes the rotation direction)
- `shift` : angular shift in degrees to easily adjust angles

Test

```
ho = library("hoa.lib");
os = library("oscillators.lib");
ambi = ho.encoder(1, os.osc(440), 0.0);
iBasicDecoder_test = ambi : ho.iBasicDecoder(1, (0, 120, 240), 1, 0);
```

`(ho.)circularScaledVBAP`

The function provides a circular scaled VBAP with all loudspeakers and the virtual source on the unit-circle.

Usage

```
_ : circularScaledVBAP(l, t) : _,_, ...
```

Where:

- `l` : the list of angles of the loudspeakers in degrees, for instance (0, 85, 182, 263) for four loudspeakers
- `t` : the current angle of the virtual source in degrees

Test

```
ho = library("hoa.lib");
os = library("oscillators.lib");
circularScaledVBAP_test = os.osc(440) : ho.circularScaledVBAP((0, 120, 240), 60);
```

`(ho.)imlsDecoder`

Irregular decoder in 2D for an irregular configuration of P loudspeakers using 2D VBAP for compensation.

Usage

```
_,_, ... : imlsDecoder(N,la, direct, shift) : _,_, ...
```

Where:

- `N`: the ambisonic order (constant numerical expression)
- `la` : the list of P angles in degrees, for instance (0, 85, 182, 263) for four loudspeakers

- `direct`: 1 for direct mode, -1 for the indirect mode (changes the rotation direction)
- `shift` : angular shift in degrees to easily adjust angles

Test

```
ho = library("hoa.lib");
os = library("oscillators.lib");
ambi = ho.encoder(1, os.osc(440), 0.0);
imlsDecoder_test = ambi : ho.imlsDecoder(1, (0, 90, 180, 270), 1, 0);
```

`(ho.)iDecoder`

General decoder in 2D enabling an irregular multi-loudspeaker configuration and to switch between multi-channel and stereo.

Usage

```
_,_, ... : iDecoder(N, la, direct, st, g) : _,_, ...
```

Where:

- `N`: the ambisonic order (constant numerical expression)
- `la`: the list of angles in degrees
- `direct`: 1 for direct mode, -1 for the indirect mode (changes the rotation direction)
- `shift` : angular shift in degrees to easily adjust angles
- `st`: 1 for stereo, 0 for multi-loudspeaker configuration. When 1, stereo sounds goes through the first two channels
- `g` : gain between 0 and 1

Test

```
ho = library("hoa.lib");
os = library("oscillators.lib");
ambi = ho.encoder(1, os.osc(440), 0.0);
iDecoder_test = (ambi, 0.0) : ho.iDecoder(1, (0, 120, 240), 1, 0, 0.8);
```

Optimization Functions

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

(ho.)optimBasic

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

Usage

```
_ : optimBasic(N) : _
```

Where:

- `N`: the ambisonic order (constant numerical expression)

Test

```
ho = library("hoa.lib");
os = library("oscillators.lib");
ambi = ho.encoder(1, os.osc(440), 0.0);
optimBasic_test = ambi : ho.optimBasic(1);
```

(ho.)optimMaxRe

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

Usage

```
_ : optimMaxRe(N) : _
```

Where:

- `N`: the ambisonic order (constant numerical expression)

Test

```
ho = library("hoa.lib");
os = library("oscillators.lib");
ambi = ho.encoder(1, os.osc(440), 0.0);
optimMaxRe_test = ambi : ho.optimMaxRe(1);
```

(ho.)optimInPhase

The inPhase optimization optimizes energy vector and put all loudspeakers signals in phase. It should be used for an auditory.

Usage

```
_ : optimInPhase(N) : _
```

Where:

- `N`: the ambisonic order (constant numerical expression)

Test

```
ho = library("hoa.lib");
os = library("oscillators.lib");
ambi = ho.encoder(1, os.osc(440), 0.0);
optimInPhase_test = ambi : ho.optimInPhase(1);
```

(ho.)optim

Ambisonic optimizer including the three elementary optimizers: `(ho).optimBasic`, `(ho).optimMaxRe` and `(ho.)optimInPhase`.

Usage

```
_,_, ... : optim(N, ot) : _,_, ...
```

Where:

- `N`: the ambisonic order (constant numerical expression)
- `ot` : optimization type (0 for `optimBasic`, 1 for `optimMaxRe`, 2 for `optimInPhase`)

Test

```
ho = library("hoa.lib");
os = library("oscillators.lib");
ambi = ho.encoder(1, os.osc(440), 0.0);
optim_test = ambi : ho.optim(1, 1);
```

(ho.)wider

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

Usage

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

Where:

- `N`: the ambisonic order (constant numerical expression)
- `w`: the width value between 0 - 1

Test

```
ho = library("hoa.lib");
os = library("oscillators.lib");
ambi = ho.encoder(1, os.osc(440), 0.0);
wider_test = ambi : ho.wider(1, 0.5);
```

(ho.)mirror

Mirroring effect on the sound field.

Usage

```
_,_, ... : mirror(N, fa) : _,_, ...
```

Where:

- `N`: the ambisonic order (constant numerical expression)
- `fa` : mirroring type (1 = original sound field, 0 = original+mirrored sound field, -1 = mirrored sound field)

Test

```
ho = library("hoa.lib");
os = library("oscillators.lib");
ambi = ho.encoder(1, os.osc(440), 0.0);
mirror_test = ambi : ho.mirror(1, -1);
```

(ho.)map

It simulates 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 ambisonic order (constant numerical expression)
- `x`: the signal
- `r`: the radius
- `a`: the angle in radian

Test

```
ho = library("hoa.lib");
os = library("oscillators.lib");
map_test = ho.map(1, os.osc(440), 0.5, 0.0);
```

(ho.)rotate

Rotates the sound field.

Usage

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

Where:

- `N`: the ambisonic order (constant numerical expression)
- `a`: the angle in radian

Test

```
ho = library("hoa.lib");
os = library("oscillators.lib");
rotate_test = ho.encoder(1, os.osc(440), 0.0) : ho.rotate(1, 0.78);
```

(ho.)scope

Produces an XY pair of signals representing the ambisonic sound field.

Usage

```
_,_, ... : scope(N, rt) : _,_
```

Where:

- `N`: the ambisonic order (constant numerical expression)
- `rt` : refreshment time in milliseconds

Test

```
ho = library("hoa.lib");
os = library("oscillators.lib");
scope_test = ho.encoder(1, os.osc(440), 0.0) : ho.scope(1, 0.1);
```

Spatial Sound Processes

We propose implementations of processes intricated to the ambisonic model. The process is implemented using as many instances as the number of harmonics at at certain order. The key control parameters of these instances are computed thanks to distribution functions (th functions below) and to a global driving factor.

`(ho.).fxDecorrelation`

Spatial ambisonic decorrelation in fx mode.

`fxDecorrelation` applies decorrelations to spatial components already created. The decorrelation is defined for each #i spatial component among P=2*N+1 at the ambisonic order `N` as a delay of 0 if factor `fa` is under a certain value 1-(i+1)/P and d*F((i+1)/p) in the contrary case, where `d` is the maximum delay applied (in samples) and F is a distribution function for durations. The user can choose this delay time distribution among 22 different ones. The delay increases according to the index of ambisonic components. But it increases at each step and it is modulated by a threshold. Therefore, delays are progressively revealed when the factor increases:

- when the factor is close to 0, only upper components are delayed;
- when the factor increases, more and more components are delayed.

Usage

```
_,_, ... : fxDecorrelation(N, d, wf, fa, fd, tf) : _,_, ...
```

Where:

- `N`: the ambisonic order (constant numerical expression)
- `d`: the maximum delay applied (in samples)
- `wf`: window frequency (in Hz) for the overlapped delay
- `fa`: decorrelation factor (between 0 and 1)
- `fd`: feedback / level of reinjection (between 0 and 1)
- `tf`: type of function of delay distribution (integer, between 0 and 21)

Test

```
ho = library("hoa.lib");
os = library("oscillators.lib");
fxDecorrelation_test = ho.encoder(1, os.osc(440), 0.0) : ho.fxDecorrelation(1, 64, 5, 0.5, 0
```

(ho.).synDecorrelation

Spatial ambisonic decorrelation in syn mode.

`synDecorrelation` generates spatial decorrelated components in ambisonics from one mono signal. The decorrelation is defined for each #i spatial component among P=2*N+1 at the ambisonic order `N` as a delay of 0 if factor `fa` is under a certain value 1-(i+1)/P and d*F((i+1)/p) in the contrary case, where `d` is the maximum delay applied (in samples) and F is a distribution function for durations. The user can choose this delay time distribution among 22 different ones. The delay increases according to the index of ambisonic components. But it increases at each step and it is modulated by a threshold. Therefore, delays are progressively revealed when the factor increases:

- when the factor is close to 0, only upper components are delayed;
- when the factor increases, more and more components are delayed.

When the factor is between [0; 1/P], upper harmonics are progressively faded and the level of the H0 component is compensated to avoid source localization and to produce a large mono.

Usage

```
_,_, ... : synDecorrelation(N, d, wf, fa, fd, tf) : _,_, ...
```

Where:

- `N`: the ambisonic order (constant numerical expression)
- `d`: the maximum delay applied (in samples)
- `wf`: window frequency (in Hz) for the overlapped delay
- `fa`: decorrelation factor (between 0 and 1)
- `fd`: feedback / level of reinjection (between 0 and 1)
- `tf`: type of function of delay distribution (integer, between 0 and 21)

Test

```
ho = library("hoa.lib");
os = library("oscillators.lib");
synDecorrelation_test = os.osc(440) : ho.synDecorrelation(1, 64, 5, 0.5, 0.2, 0);
```

(ho.).fxRingMod

Spatial ring modulation in syn mode.

`fxRingMod` applies ring modulation to spatial components already created. The ring modulation is defined for each spatial component among P=2*n+1 at the ambisonic order `N`. For each spatial component #i, the result is either the original signal or a ring modulated signal according to a threshold that is i/P.

The general process is drive by a factor `fa` between 0 and 1 and a modulation frequency `f0`. If `fa` is greater than theshold (P-i-1)/P, the ith ring modulator is on with carrier frequency of f0*(i+1)/P. On the contrary, it provides the original signal.

Therefore ring modulators are progressively revealed when `fa` increases.

Usage

```
_,_, ... : fxRingMod(N, f0, fa, tf) : _,_, ...
```

Where:

- `N`: the ambisonic order (constant numerical expression)
- `f0`: the maximum delay applied (in samples)
- `fa`: decorrelation factor (between 0 and 1)
- `tf`: type of function of delay distribution (integer, between 0 and 21)

Test

```
ho = library("hoa.lib");
os = library("oscillators.lib");
fxRingMod_test = ho.encoder(1, os.osc(440), 0.0) : ho.fxRingMod(1, 200, 0.5, 0);
```

`(ho.).synRingMod`

Spatial ring modulation in syn mode.

`synRingMod` generates spatial components in ambisonics from one mono signal thanks to ring modulation. The ring modulation is defined for each spatial component among P=2*n+1 at the ambisonic order `N`. For each spatial component #i, the result is either the original signal or a ring modulated signal according to a threshold that is i/P.

The general process is drive by a factor `fa` between 0 and 1 and a modulation frequency `f0`. If `fa` is greater than theshold (P-i-1)/P, the ith ring modulator is on with carrier frequency of f0*(i+1)/P. On the contrary, it provides the original signal.

Therefore ring modulators are progressively revealed when `fa` increases. When the factor is between [0; 1/P], upper harmonics are progressively faded and the level of the H0 component is compensated to avoid source localization and to produce a large mono.

Usage

```
_,_, ... : synRingMod(N, f0, fa, tf) : _,_, ...
```

Where:

- `N`: the ambisonic order (constant numerical expression)
- `f0`: the maximum delay applied (in samples)
- `fa`: decorrelation factor (between 0 and 1)
- `tf`: type of function of delay distribution (integer, between 0 and 21)

Test

```
ho = library("hoa.lib");
os = library("oscillators.lib");
synRingMod_test = os.osc(440) : ho.synRingMod(1, 200, 0.5, 0);
```

3D Functions

`(ho.)encoder3D`

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

Usage

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

Where:

- `N`: the ambisonic order (constant numerical expression)
- `x`: the signal
- `a`: the angle
- `e`: the elevation

Test

```
ho = library("hoa.lib");
os = library("oscillators.lib");
encoder3D_test = ho.encoder3D(1, os.osc(440), 0.0, 0.0);
```

`(ho.)rEncoder3D`

Ambisonic encoder in 3D including source rotation. A mono signal is encoded at at certain ambisonic order with two possible modes: either rotation with 2 angular speeds (azimuth and elevation), or static with a fixed pair of angles.

`rEncoder3D` is a standard Faust function.

Usage

```
_ : rEncoder3D(N, azsp, elsp, az, el, it) : _,_, ...
```

Where:

- `N`: the ambisonic order (constant numerical expression)
- `azsp`: the azimuth speed expressed as angular speed (2PI/sec), positive or negative
- `elsp`: the elevation speed expressed as angular speed (2PI/sec), positive or negative
- `az`: the fixed azimuth when the azimuth rotation stops (azsp = 0) in radians
- `el`: the fixed elevation when the elevation rotation stops (elsp = 0) in radians
- `it` : interpolation time (in milliseconds) between the rotation and the fixed modes

Test

```
ho = library("hoa.lib");
os = library("oscillators.lib");
rEncoder3D_test = os.osc(440) : ho.rEncoder3D(1, 0.5, 0.3, 0.0, 0.0, 0.05);
```

`(ho.)optimBasic3D`

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

Usage

```
_ : optimBasic3D(N) : _
```

Where:

- `N`: the ambisonic order (constant numerical expression)

Test

```
ho = library("hoa.lib");
os = library("oscillators.lib");
ambi3D = ho.encoder3D(1, os.osc(440), 0.0, 0.0);
optimBasic3D_test = ambi3D : ho.optimBasic3D(1);
```

(ho.)optimMaxRe3D

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

Usage

```
_ : optimMaxRe3D(N) : _
```

Where:

- `N`: the ambisonic order (constant numerical expression)

Test

```
ho = library("hoa.lib");
os = library("oscillators.lib");
ambi3D = ho.encoder3D(1, os.osc(440), 0.0, 0.0);
optimMaxRe3D_test = ambi3D : ho.optimMaxRe3D(1);
```

(ho.)optimInPhase3D

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

Usage

```
_ : optimInPhase3D(N) : _
```

Where:

- `N`: the ambisonic order (constant numerical expression)

Test

```
ho = library("hoa.lib");
os = library("oscillators.lib");
ambi3D = ho.encoder3D(1, os.osc(440), 0.0, 0.0);
optimInPhase3D_test = ambi3D : ho.optimInPhase3D(1);
```

(ho.)optim3D

Ambisonic optimizer including the three elementary optimizers: `(ho).optimBasic3D`, `(ho).optimMaxRe3D` and `(ho.)optimInPhase3D`.

Usage

```
_,_, ... : optim3D(N, ot) : _,_, ...
```

Where:

- `N`: the ambisonic order (constant numerical expression)
- `ot` : optimization type (0 for optimBasic, 1 for optimMaxRe, 2 for optimInPhase)

Test

```
ho = library("hoa.lib");
os = library("oscillators.lib");
ambi3D = ho.encoder3D(1, os.osc(440), 0.0, 0.0);
optim3D_test = ambi3D : ho.optim3D(1, 2);
```

Faust Libraries Index

---

aanl

(aa.)clip (aa.)Rsqrt (aa.)Rlog (aa.)Rtan (aa.)Racos (aa.)Rasin (aa.)Racosh (aa.)Rcosh (aa.)Rsinh (aa.)Ratanh (aa.)ADAA1 (aa.)ADAA2 (aa.)hardclip (aa.)hardclip2 (aa.)cubic1 (aa.)parabolic (aa.)parabolic2 (aa.)hyperbolic (aa.)hyperbolic2 (aa.)sinarctan (aa.)sinarctan2 (aa.)softclipQuadratic1 (aa.)softclipQuadratic2 (aa.)tanh1 (aa.)arctan (aa.)arctan2 (aa.)asinh1 (aa.)asinh2 (aa.)cosine1 (aa.)cosine2 (aa.)arccos (aa.)arccos2 (aa.)acosh1 (aa.)acosh2 (aa.)sine (aa.)sine2 (aa.)arcsin (aa.)arcsin2 (aa.)tangent (aa.)atanh1 (aa.)atanh2

analyzers

(an.)abs_envelope_rect (an.)abs_envelope_tau (an.)abs_envelope_t60 (an.)abs_envelope_t19 (an.)amp_follower (an.)amp_follower_ud (an.)amp_follower_ar (an.)ms_envelope_rect (an.)ms_envelope_tau (an.)ms_envelope_t60 (an.)ms_envelope_t19 (an.)rms_envelope_rect (an.)rms_envelope_tau (an.)rms_envelope_t60 (an.)rms_envelope_t19 (an.)zcr (an.)pitchTracker (an.)spectralCentroid (an.)mth_octave_analyzer (an.)mth_octave_spectral_level6e (an.)[third|half]*octave*[analyzer|filterbank] (an.)analyzer (an.)goertzelOpt (an.)goertzelComp (an.)goertzel (an.)resonator (an.)fft (an.)ifft (an.)logsweep (an.)linsweep

basics

(ba.)samp2sec (ba.)sec2samp (ba.)db2linear (ba.)linear2db (ba.)lin2LogGain (ba.)log2LinGain (ba.)tau2pole (ba.)pole2tau (ba.)midikey2hz (ba.)hz2midikey (ba.)semi2ratio (ba.)ratio2semi (ba.)cent2ratio (ba.)ratio2cent (ba.)pianokey2hz (ba.)hz2pianokey (ba.)counter (ba.)countdown (ba.)countup (ba.)sweep (ba.)time (ba.)ramp (ba.)line (ba.)tempo (ba.)period (ba.)spulse (ba.)pulse (ba.)pulsen (ba.)cycle (ba.)beat (ba.)pulse_countup (ba.)pulse_countdown (ba.)pulse_countup_loop (ba.)pulse_countdown_loop (ba.)resetCtr (ba.)count (ba.)take (ba.)pick (ba.)pickN (ba.)subseq (ba.)tabulate (ba.)tabulate_chebychev (ba.)tabulateNd (ba.)if (ba.)ifNc (ba.)ifNcNo (ba.)selector (ba.)select2stereo (ba.)selectn (ba.)selectbus (ba.)selectxbus (ba.)selectmulti (ba.)selectoutn (ba.)latch (ba.)sAndH (ba.)tAndH (ba.)downSample (ba.)downSampleCV (ba.)peakhold (ba.)peakholder (ba.)kr2ar (ba.)impulsify (ba.)automat (ba.)bpf (ba.)listInterp (ba.)bypass1 (ba.)bypass2 (ba.)bypass1to2 (ba.)bypass_fade (ba.)toggle (ba.)on_and_off (ba.)bitcrusher (ba.)mulaw_bitcrusher (ba.)slidingReduce (ba.)slidingSum (ba.)slidingSump (ba.)slidingMax (ba.)slidingMin (ba.)slidingMean (ba.)slidingMeanp (ba.)slidingRMS (ba.)slidingRMSp (ba.)parallelOp (ba.)parallelMax (ba.)parallelMin (ba.)parallelMean (ba.)parallelRMS

compressors

(co.)ratio2strength (co.)strength2ratio (co.)peak_compression_gain_mono_db (co.)peak_compression_gain_N_chan_db (co.)FFcompressor_N_chan (co.)FBcompressor_N_chan (co.)FBFFcompressor_N_chan (co.)RMS_compression_gain_mono_db (co.)RMS_compression_gain_N_chan_db (co.)RMS_FBFFcompressor_N_chan (co.)RMS_FBcompressor_peak_limiter_N_chan (co.)peak_compression_gain_mono (co.)peak_compression_gain_N_chan (co.)RMS_compression_gain_mono (co.)RMS_compression_gain_N_chan (co.)compressor_lad_mono (co.)compressor_mono (co.)compressor_stereo (co.)compression_gain_mono (co.)limiter_1176_R4_mono (co.)limiter_1176_R4_stereo (co.)peak_expansion_gain_N_chan_db (co.)expander_N_chan (co.)expanderSC_N_chan (co.)limiter_lad_N (co.)limiter_lad_mono (co.)limiter_lad_stereo (co.)limiter_lad_quad (co.)limiter_lad_bw

delays

(de.)delay (de.)fdelay (de.)sdelay (de.)prime_power_delays (de.)fdelaylti and (de.)fdelayltv (de.)fdelay[N] (de.)fdelay[N]a (de.)multiTapSincDelay

demos

(dm.)mth_octave_spectral_level_demo (dm.)parametric_eq_demo (dm.)spectral_tilt_demo (dm.)mth_octave_filterbank_demo and (dm.)filterbank_demo (dm.)cubicnl_demo (dm.)gate_demo (dm.)compressor_demo (dm.)moog_vcf_demo (dm.)wah4_demo (dm.)crybaby_demo (dm.)flanger_demo (dm.)phaser2_demo (dm.)tapeStop_demo (dm.)freeverb_demo (dm.)stereo_reverb_tester (dm.)fdnrev0_demo (dm.)zita_rev_fdn_demo (dm.)zita_light (dm.)zita_rev1 (dm.)vital_rev_demo (dm.)reverbTank_demo (dm.)kb_rom_rev1_demo (dm.)dattorro_rev_demo (dm.)jprev_demo (dm.)greyhole_demo (dm.)sawtooth_demo (dm.)virtual_analog_oscillator_demo (dm.)oscrs_demo (dm.)velvet_noise_demo (dm.)latch_demo (dm.)envelopes_demo (dm.)fft_spectral_level_demo (dm.)reverse_echo_demo(nChans) (dm.)pospass_demo (dm.)exciter (dm.)vocoder_demo (dm.)colored_noise_demo

dx7/dx7

(dx.)fdbkscalef (dx.)fdbkscalef2 (dx.)algorithms (dx.)algorithm

dx7/env

(dx.)env

dx7/lfo

(dx.)lfo

dx7/operator

(dx.)operator

dx7/pitchenv

(dx.)pitchenv

envelopes

(en.)ar (en.)asr (en.)adsr (en.)adsrf_bias (en.)adsr_bias (en.)ahdsrf_bias (en.)ahdsr_bias (en.)smoothEnvelope (en.)arfe (en.)are (en.)asre (en.)adsre (en.)ahdsre (en.)dx7envelope

fds

(fd.)model1D (fd.)model2D (fd.)stairsInterp1D (fd.)stairsInterp2D (fd.)linInterp1D (fd.)linInterp2D (fd.)stairsInterp1DOut (fd.)stairsInterp2DOut (fd.)linInterp1DOut (fd.)stairsInterp2DOut (fd.)route1D (fd.)route2D (fd.)schemePoint (fd.)buildScheme1D (fd.)buildScheme2D (fd.)hammer (fd.)bow

filters

(fi.)zero (fi.)pole (fi.)integrator (fi.)dcblockerat (fi.)dcblocker (fi.)lptN (fi.)lptau (fi.)lpt60 (fi.)lpt19 (fi.)ff_comb (fi.)ff_fcomb (fi.)ffcombfilter (fi.)fb_comb_common (fi.)fb_comb (fi.)fb_fcomb (fi.)rev1 (fi.)fbcombfilter and (fi.)ffbcombfilter (fi.)allpass_comb (fi.)allpass_fcomb (fi.)rev2 (fi.)allpass_fcomb5 and (fi.)allpass_fcomb1a (fi.)iir (fi.)fir (fi.)conv and (fi.)convN (fi.)tf1, (fi.)tf2 and (fi.)tf3 (fi.)notchw (fi.)tf21, (fi.)tf22, (fi.)tf22t and (fi.)tf21t (fi.)av2sv (fi.)bvav2nuv (fi.)iir_lat2 (fi.)allpassnt (fi.)iir_kl (fi.)allpassnklt (fi.)iir_lat1 (fi.)allpassn1mt (fi.)iir_nl (fi.)allpassnnlt (fi.)tf2np (fi.)wgr (fi.)nlf2 (fi.)apnl (fi.)scatN (fi.)scat (fi.)allpassn (fi.)allpassnn (fi.)allpassnkl (fi.)allpass1m (fi.)tf2s and (fi.)tf2snp (fi.)tf1snp (fi.)tf3slf (fi.)tf1s (fi.)tf2sb (fi.)tf1sb (fi.)resonlp (fi.)resonhp (fi.)resonbp (fi.)lowpass (fi.)highpass (fi.)lowpass0_highpass1 (fi.)highpass_plus_lowpass (fi.)highpass_minus_lowpass (fi.)highpass_plus_lowpass_even (fi.)highpass_plus_lowpass_even (fi.)highpass_minus_lowpass_odd (fi.)highpass_minus_lowpass_odd (fi.)lowpass3e (fi.)lowpass6e (fi.)highpass3e (fi.)highpass6e (fi.)bandpass (fi.)bandstop (fi.)bandstop (fi.)bandpass6e (fi.)bandpass12e (fi.)pospass (fi.)lowshelf (fi.)low_shelf (fi.)low_shelf1_l (fi.)low_shelf1_l (fi.)lowshelf_other_freq (fi.)high_shelf (fi.)high_shelf (fi.)high_shelf1 (fi.)high_shelf1_l (fi.)highshelf_other_freq (fi.)peak_eq (fi.)peak_eq_cq (fi.)peak_eq_rm (fi.)spectral_tilt (fi.)levelfilter (fi.)levelfilterN (fi.)mth_octave_filterbank[n] (fi.)mth_octave_filterbank_alt (fi.)mth_octave_filterbank3 (fi.)mth_octave_filterbank5 (fi.)mth_octave_filterbank_default (fi.)filterbank (fi.)filterbanki (fi.)svf (fi.)svf_morph (fi.)svf_notch_morph (fi.)SVFTPT (fi.)dynamicSmoothing (fi.)oneEuro (fi.)lowpassLR4 (fi.)highpassLR4 (fi.)crossover2LR4 (fi.)crossover3LR4 (fi.)crossover4LR4 (fi.)crossover8LR4 (fi.)itu_r_bs_1770_4_kfilter (fi.)avg_rect (fi.)avg_tau (fi.)avg_t60 (fi.)avg_t19 (fi.)kalman

hoa

(ho.)encoder (ho.)rEncoder (ho.)stereoEncoder (ho.)multiEncoder (ho.)decoder (ho.)decoderStereo (ho.)iBasicDecoder (ho.)circularScaledVBAP (ho.)imlsDecoder (ho.)iDecoder (ho.)optimBasic (ho.)optimMaxRe (ho.)optimInPhase (ho.)optim (ho.)wider (ho.)mirror (ho.)map (ho.)rotate (ho.)scope (ho.).fxDecorrelation (ho.).synDecorrelation (ho.).fxRingMod (ho.).synRingMod (ho.)encoder3D (ho.)rEncoder3D (ho.)optimBasic3D (ho.)optimMaxRe3D (ho.)optimInPhase3D (ho.)optim3D

interpolators

(it.)interpolate_linear (it.)interpolate_cosine (it.)interpolate_cubic (it.)interpolator_two_points (it.)interpolator_linear (it.)interpolator_cosine (it.)interpolator_four_points (it.)interpolator_cubic (it.)interpolator_select (it.)lerp (it.)piecewise (it.)lagrangeCoeffs (it.)lagrangeInterpolation (it.)frdtable (it.)frwtable (it.)remap

linearalgebra

(la.)determinant (la.)minor (la.)inverse (la.)transpose2 (la.)matMul (la.)identity (la.)diag

maths

(ma.)SR (ma.)T (ma.)BS (ma.)PI (ma.)deg2rad (ma.)rad2deg (ma.)E (ma.)EPSILON (ma.)MIN (ma.)MAX (ma.)FTZ (ma.)copysign (ma.)neg (ma.)not (ma.)sub(x,y) (ma.)inv (ma.)cbrt (ma.)hypot (ma.)ldexp (ma.)scalb (ma.)log1p (ma.)logb (ma.)ilogb (ma.)log2 (ma.)expm1 (ma.)acosh (ma.)asinh (ma.)atanh (ma.)sinh (ma.)cosh (ma.)tanh (ma.)erf (ma.)erfc (ma.)gamma (ma.)lgamma (ma.)J0 (ma.)J1 (ma.)Jn (ma.)Y0 (ma.)Y1 (ma.)Yn (ma.)fabs, (ma.)fmax, (ma.)fmin (ma.)np2 (ma.)frac (ma.)modulo (ma.)isnan (ma.)isinf (ma.)chebychev (ma.)chebychevpoly (ma.)diffn (ma.)signum (ma.)nextpow2 (ma.)zc (ma.)unwrap (ma.)primes

mi

(mi.)initState (mi.)mass (mi.)oscil (mi.)ground (mi.)posInput (mi.)spring (mi.)damper (mi.)springDamper (mi.)nlSpringDamper2 (mi.)nlSpringDamper3 (mi.)nlSpringDamperClipped (mi.)nlPluck (mi.)nlBow (mi.)collision (mi.)nlCollisionClipped

misceffects

(ef.)cubicnl (ef.)gate_mono (ef.)gate_stereo (ef.)fibonacci (ef.)fibonacciGeneral (ef.)fibonacciSeq (ef.)speakerbp (ef.)piano_dispersion_filter (ef.)stereo_width (ef.)mesh_square (ef.)dryWetMixer (ef.)dryWetMixerConstantPower (ef.)mixLinearClamp (ef.)mixLinearLoop (ef.)mixPowerClamp (ef.)mixPowerLoop (ef.)echo (ef.)reverseEchoN (ef.)reverseDelayRamped (ef.)uniformPanToStereo (ef.)tapeStop (ef.)transpose (ef.)softclipQuadratic (ef.)wavefold

oscillators

(os.)sinwaveform (os.)coswaveform (os.)phasor (os.)hs_phasor (os.)hsp_phasor (os.)oscsin (os.)hs_oscsin (os.)osccos (os.)hs_osccos (os.)oscp (os.)osci (os.)osc (os.)m_oscsin (os.)m_osccos (os.)lf_imptrain (os.)lf_pulsetrainpos (os.)lf_pulsetrain (os.)lf_squarewavepos (os.)lf_squarewave (os.)lf_trianglepos (os.)lf_triangle (os.)lf_rawsaw (os.)lf_sawpos (os.)lf_sawpos_phase (os.)lf_sawpos_reset (os.)lf_sawpos_phase_reset (os.)lf_saw (os.)sawN (os.)sawNp (os.)saw2, (os.)saw3, (os.)saw4 (os.)saw2ptr (os.)saw2dpw (os.)sawtooth (os.)saw2f2, (os.)saw2f4 (os.)impulse (os.)pulsetrainN (os.)pulsetrain (os.)squareN (os.)square (os.)imptrainN (os.)imptrain (os.)triangleN (os.)triangle (os.)oscb (os.)oscrq (os.)oscrs (os.)oscrc (os.)oscs (os.)quadosc (os.)sidebands (os.)sidebands_list (os.)dsf (os.)oscwc (os.)oscws (os.)oscq (os.)oscw (os.)CZsaw (os.)CZsawP (os.)CZsquare (os.)CZsquareP (os.)CZpulse (os.)CZpulseP (os.)CZsinePulse (os.)CZsinePulseP (os.)CZhalfSine (os.)CZhalfSineP (os.)CZresSaw (os.)CZresTriangle (os.)CZresTrap (os.)polyblep (os.)polyblep_saw (os.)polyblep_square (os.)polyblep_triangle

noises

(no.)noise (no.)multirandom (no.)multinoise (no.)noises (no.)dnoise (no.)randomseed (no.)rnoise (no.)rmultirandom (no.)rmultinoise (no.)rnoises (no.)pink_noise (no.)pink_noise_vm (no.)lfnoise, (no.)lfnoise0 and (no.)lfnoiseN (no.)sparse_noise (no.)velvet_noise_vm (no.)gnoise (no.)colored_noise

phaflangers

(pf.)flanger_mono (pf.)flanger_stereo (pf.)phaser2_mono (pf.)phaser2_stereo

physmodels

(pm.)speedOfSound (pm.)maxLength (pm.)f2l (pm.)l2f (pm.)l2s (pm.)basicBlock (pm.)chain (pm.)inLeftWave (pm.)inRightWave (pm.)in (pm.)outLeftWave (pm.)outRightWave (pm.)out (pm.)terminations (pm.)lTermination (pm.)rTermination (pm.)closeIns (pm.)closeOuts (pm.)endChain (pm.)waveguideN (pm.)waveguide (pm.)bridgeFilter (pm.)modeFilter (pm.)stringSegment (pm.)openString (pm.)nylonString (pm.)steelString (pm.)openStringPick (pm.)openStringPickUp (pm.)openStringPickDown (pm.)ksReflexionFilter (pm.)rStringRigidTermination (pm.)lStringRigidTermination (pm.)elecGuitarBridge (pm.)elecGuitarNuts (pm.)guitarBridge (pm.)guitarNuts (pm.)idealString (pm.)ks (pm.)ks_ui_MIDI (pm.)elecGuitarModel (pm.)elecGuitar (pm.)elecGuitar_ui_MIDI (pm.)guitarBody (pm.)guitarModel (pm.)guitar (pm.)guitar_ui_MIDI (pm.)nylonGuitarModel (pm.)nylonGuitar (pm.)nylonGuitar_ui_MIDI (pm.)modeInterpRes (pm.)modularInterpBody (pm.)modularInterpStringModel (pm.)modularInterpInstr (pm.)modularInterpInstr_ui_MIDI (pm.)bowTable (pm.)violinBowTable (pm.)bowInteraction (pm.)violinBow (pm.)violinBowedString (pm.)violinNuts (pm.)violinBridge (pm.)violinBody (pm.)violinModel (pm.)violin_ui (pm.)violin_ui_MIDI (pm.)openTube (pm.)reedTable (pm.)fluteJetTable (pm.)brassLipsTable (pm.)clarinetReed (pm.)clarinetMouthPiece (pm.)brassLips (pm.)fluteEmbouchure (pm.)wBell (pm.)fluteHead (pm.)fluteFoot (pm.)clarinetModel (pm.)clarinetModel_ui (pm.)clarinet_ui (pm.)clarinet_ui_MIDI (pm.)brassModel (pm.)brassModel_ui (pm.)brass_ui (pm.)brass_ui_MIDI (pm.)fluteModel (pm.)fluteModel_ui (pm.)flute_ui (pm.)flute_ui_MIDI (pm.)impulseExcitation (pm.)strikeModel (pm.)strike (pm.)pluckString (pm.)blower (pm.)blower_ui (pm.)djembeModel (pm.)djembe (pm.)djembe_ui_MIDI (pm.)marimbaBarModel (pm.)marimbaResTube (pm.)marimbaModel (pm.)marimba (pm.)marimba_ui_MIDI (pm.)churchBellModel (pm.)churchBell (pm.)churchBell_ui (pm.)englishBellModel (pm.)englishBell (pm.)englishBell_ui (pm.)frenchBellModel (pm.)frenchBell (pm.)frenchBell_ui (pm.)germanBellModel (pm.)germanBell (pm.)germanBell_ui (pm.)russianBellModel (pm.)russianBell (pm.)russianBell_ui (pm.)standardBellModel (pm.)standardBell (pm.)standardBell_ui (pm.)formantValues (pm.)voiceGender (pm.)skirtWidthMultiplier (pm.)autobendFreq (pm.)vocalEffort (pm.)fof (pm.)fofSH (pm.)fofCycle (pm.)fofSmooth (pm.)formantFilterFofCycle (pm.)formantFilterFofSmooth (pm.)formantFilterBP (pm.)formantFilterbank (pm.)formantFilterbankFofCycle (pm.)formantFilterbankFofSmooth (pm.)formantFilterbankBP (pm.)SFFormantModel (pm.)SFFormantModelFofCycle (pm.)SFFormantModelFofSmooth (pm.)SFFormantModelBP (pm.)SFFormantModelFofCycle_ui (pm.)SFFormantModelFofSmooth_ui (pm.)SFFormantModelBP_ui (pm.)SFFormantModelFofCycle_ui_MIDI (pm.)SFFormantModelFofSmooth_ui_MIDI (pm.)SFFormantModelBP_ui_MIDI (pm.)allpassNL (pm.)modalModel (pm.)rk_solve

quantizers

(qu.)quantize (qu.)quantizeSmoothed (qu.)ionian (qu.)dorian (qu.)phrygian (qu.)lydian (qu.)mixo (qu.)eolian (qu.)locrian (qu.)pentanat (qu.)kumoi (qu.)natural (qu.)dodeca (qu.)dimin (qu.)penta

reducemaps

(rm.)parReduce (rm.)topReduce (rm.)botReduce (rm.)reduce (rm.)reducemap

reverbs

(re.)jcrev (re.)satrev (re.)fdnrev0 (re.)zita_rev_fdn (re.)zita_rev1_stereo (re.)zita_rev1_ambi (re.)vital_rev (re.)mono_freeverb (re.)stereo_freeverb (re.)dattorro_rev (re.)dattorro_rev_default (re.)jpverb (re.)greyhole (re.)kb_rom_rev1

routes

(ro.)cross (ro.)crossnn (ro.)crossn1 (ro.)cross1n (ro.)crossNM (ro.)interleave (ro.)butterfly (ro.)hadamard (ro.)recursivize (ro.)bubbleSort

signals

(si.)bus (si.)block (si.)interpolate (si.)repeat (si.)smoo (si.)polySmooth (si.)smoothAndH (si.)bsmooth (si.)dot (si.)smooth (si.)smoothq (si.)cbus (si.)cmul (si.)cconj (si.)onePoleSwitching (si.)rev (si.)vecOp (si.)bpar (si.)bsum (si.)bprod

soundfiles

(so.)loop (so.)loop_speed (so.)loop_speed_level

spats

(sp.)panner (sp.)constantPowerPan (sp.)spat (sp.)wfs (sp.)wfs_ui (sp.)stereoize

synths

(sy.)popFilterDrum (sy.)dubDub (sy.)sawTrombone (sy.)combString (sy.)additiveDrum (sy.)fm (sy.)kick (sy.)clap (sy.)hat

vaeffects

(ve.)moog_vcf (ve.)moog_vcf_2b[n] (ve.)moogLadder (ve.)lowpassLadder4 (ve.)moogHalfLadder (ve.)diodeLadder (ve.)korg35LPF (ve.)korg35HPF (ve.)oberheim (ve.)oberheimBSF (ve.)oberheimBPF (ve.)oberheimHPF (ve.)oberheimLPF (ve.)sallenKeyOnePole (ve.)sallenKeyOnePoleLPF (ve.)sallenKeyOnePoleHPF (ve.)sallenKey2ndOrder (ve.)sallenKey2ndOrderLPF (ve.)sallenKey2ndOrderBPF (ve.)sallenKey2ndOrderHPF (ve.)biquad (ve.)lowpass2Matched (ve.)highpass2Matched (ve.)bandpass2Matched (ve.)peaking2Matched (ve.)lowshelf2Matched (ve.)highshelf2Matched (ve.)wah4 (ve.)autowah (ve.)crybaby (ve.)vocoder

version

(vl.)version

wdmodels

(wd.)resistor (wd.)resistor_Vout (wd.)resistor_Iout (wd.)u_voltage (wd.)u_current (wd.)resVoltage (wd.)resVoltage_Vout (wd.)u_resVoltage (wd.)resCurrent (wd.)u_resCurrent (wd.)u_switch (wd.)capacitor (wd.)capacitor_Vout (wd.)inductor (wd.)inductor_Vout (wd.)u_idealDiode (wd.)u_chua (wd.)lambert (wd.)u_diodePair (wd.)u_diodeSingle (wd.)u_diodeAntiparallel (wd.)u_parallel2Port (wd.)parallel2Port (wd.)u_series2Port (wd.)series2Port (wd.)parallelCurrent (wd.)seriesVoltage (wd.)u_transformer (wd.)transformer (wd.)u_transformerActive (wd.)transformerActive (wd.)parallel (wd.)series (wd.)u_sixportPassive (wd.)genericNode (wd.)genericNode_Vout (wd.)genericNode_Iout (wd.)u_genericNode (wd.)builddown (wd.)buildup (wd.)getres (wd.)parres (wd.)buildout (wd.)buildtree

webaudio

(wa.)lowpass2 (wa.)highpass2 (wa.)bandpass2 (wa.)notch2 (wa.)allpass2 (wa.)peaking2 (wa.)lowshelf2 (wa.)highshelf2

interpolators.lib

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

This library provides interpolation algorithms for signal and control processing. It includes linear, polynomial, spline, and higher-order interpolation methods used in delay lines, envelope shaping, resampling, and parameter smoothing.

The Interpolators library is organized into 7 sections:

- Two points interpolation functions
- Four points interpolation functions
- Two points interpolators
- Four points interpolators
- Generic piecewise linear interpolation
- Lagrange based interpolators
- Misc functions

The first four sections provide several basic interpolation functions, as well as interpolators taking a `gen` circuit of N outputs producing values to be interpolated, triggered by a `idv` read index signal. Two points and four points interpolations are implemented.

The idv parameter is to be used as a read index. In -single (= singleprecision) mode, a technique based on 2 signals with the pure integer index and a fractional part in the [0,1] range is used to avoid accumulating errors. In -double (= doubleprecision) or -quad (= quadprecision) modes, a standard implementation with a single fractional index signal is used. Three functions int_part, frac_part and mak_idv are available to manipulate the read index signal.

Here is a use-case with waveform. Here the signal given to interpolator_XXX uses the idv model.

```
waveform_interpolator(wf, step, interp) = interp(gen, idv)
with {
   gen(idx) = wf, (idx:max(0):min(size-1)) : rdtable with { size = wf:(_,!); };   /* wavefor
   index = (+(step)~_)-step;  /* starting from 0 */
   idv = it.make_idv(index);  /* build the signal for interpolation in a generic way */
};

waveform_linear(wf, step) = waveform_interpolator(wf, step, it.interpolator_linear);
waveform_cosine(wf, step) = waveform_interpolator(wf, step, it.interpolator_cosine);
waveform_cubic(wf, step) = waveform_interpolator(wf, step, it.interpolator_cubic);

waveform_interp(wf, step, selector) = waveform_interpolator(wf, step, interp_select(selector
with {
   /* adapts the argument order */
   interp_select(sel, gen, idv) = it.interpolator_select(gen, idv, sel);
};

waveform and index
waveform_interpolator1(wf, idv, interp) = interp(gen, idv)
with {
   gen(idx) = wf, (idx:max(0):min(size-1)) : rdtable with { size = wf:(_,!); };   /* wavefor
};

waveform_linear1(wf, idv) = waveform_interpolator1(wf, idv, it.interpolator_linear);
waveform_cosine1(wf, idv) = waveform_interpolator1(wf, idv, it.interpolator_cosine);
waveform_cubic1(wf, idv) = waveform_interpolator1(wf, idv, it.interpolator_cubic);

waveform_interp1(wf, idv, selector) = waveform_interpolator1(wf, idv, interp_select(selector
with {
   /* adapts the argument order */
   interp_select(sel, gen, idv) = it.interpolator_select(gen, idv, sel);
};
```

Some tests here:

```
wf = waveform {0.0, 10.0, 20.0, 30.0, 40.0, 50.0, 60.0, 50.0, 40.0, 30.0, 20.0, 10.0, 0.0};

process = waveform_linear(wf, step), waveform_cosine(wf, step), waveform_cubic(wf, step) wit
```

```
process = waveform_interp(wf, 0.25, nentry("algo", 0, 0, 3, 1));

process = waveform_interp1(wf, idv, nentry("algo", 0, 0, 3, 1))
with {
   step = 0.1;
   idv_aux = (+(step)~_)-step;  /* starting from 0 */
   idv = it.make_idv(idv_aux);  /* build the signal for interpolation in a generic way */
};

/* Test linear interpolation between 2 samples with a `(idx,dv)` signal built using a wavefo
linear_test = (idx,dv), it.interpolator_linear(gen, (idx,dv))
with {
   /* signal to interpolate (only 2 points here) */
   gen(id) = waveform {3.0, -1.0}, (id:max(0)) : rdtable;
   dv = waveform {0.0, 0.25, 0.50, 0.75, 1.0}, index : rdtable;
   idx = 0;
   /* test index signal */
   index = (+(1)~_)-1;   /* starting from 0 */
};

/* Test cosine interpolation between 2 samples with a `(idx,dv)` signal built using a wavefo
cosine_test = (idx,dv), it.interpolator_cosine(gen, (idx,dv))
with {
   /* signal to interpolate (only 2 points here) */
   gen(id) = waveform {3.0, -1.0}, (id:max(0)) : rdtable;
   dv = waveform {0.0, 0.25, 0.50, 0.75, 1.0}, index : rdtable;
   idx = 0;
   /* test index signal */
   index = (+(1)~_)-1;   /* starting from 0 */
};

/* Test cubic interpolation between 4 samples with a `(idx,dv)` signal built using a wavefor
cubic_test = (idx,dv), it.interpolator_cubic(gen, (idx,dv))
with {
   /* signal to interpolate (only 4 points here) */
   gen(id) = waveform {-1.0, 2.0, 1.0, 4.0}, (id:max(0)) : rdtable;
   dv = waveform {0.0, 0.25, 0.50, 0.75, 1.0}, index : rdtable;
   idx = 0;
   /* test index signal */
   index = (+(1)~_)-1;   /* starting from 0 */
};
```

References

- https://github.com/grame-cncm/faustlibraries/blob/master/interpolator s.lib

Two points interpolation functions

(it.)interpolate_linear

Linear interpolation between 2 values.

Usage

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

Where:

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

Test

```
it = library("interpolators.lib");
interpolate_linear_test = it.interpolate_linear(0.5, 0.0, 1.0);
```

References

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

---

(it.)interpolate_cosine

Cosine interpolation between 2 values.

Usage

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

Where:

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

Test

```
it = library("interpolators.lib");
interpolate_cosine_test = it.interpolate_cosine(0.5, 0.0, 1.0);
```

References

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

Four points interpolation functions

`(it.)interpolate_cubic`

Cubic interpolation between 4 values.

Usage

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

Where:

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

Test

```
it = library("interpolators.lib");
interpolate_cubic_test = it.interpolate_cubic(0.5, -1.0, 2.0, 1.0, 4.0);
```

References

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

Two points interpolators

`(it.)interpolator_two_points`

Generic interpolator on two points (current and next index), assuming an increasing index.

Usage

```
interpolator_two_points(gen, idv, interpolate_two_points) : si.bus(outputs(gen))
```

Where:

- `gen`: a circuit with an 'idv' reader input that produces N outputs

- `idv`: a fractional read index expressed as a float value, or a (int,frac) pair
- `interpolate_two_points`: a two points interpolation function

Test

```
it = library("interpolators.lib");
ma = library("maths.lib");
interpolator_two_points_test = it.interpolator_two_points(gen, idv, it.interpolate_linear)
with {
    gen(idx) = waveform {0.0, 1.0, 4.0, 9.0, 16.0}, int(ma.modulo(idx, 5)) : rdtable;
    step = 0.25;
    idxFloat = ma.modulo((+(step)~_) - step, 4.0);
    idv = it.make_idv(idxFloat);
};
```

`(it.)interpolator_linear`

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

Usage

```
interpolator_linear(gen, idv) : si.bus(outputs(gen))
```

Where:

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

Test

```
it = library("interpolators.lib");
ma = library("maths.lib");
interpolator_linear_test = it.interpolator_linear(gen, idv)
with {
    gen(idx) = waveform {0.0, 1.0, 4.0, 9.0, 16.0}, int(ma.modulo(idx, 5)) : rdtable;
    step = 0.25;
    idxFloat = ma.modulo((+(step)~_) - step, 4.0);
    idv = it.make_idv(idxFloat);
};
```

`(it.)interpolator_cosine`

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

Usage

```
interpolator_cosine(gen, idv) : si.bus(outputs(gen))
```

Where:

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

Test

```
it = library("interpolators.lib");
ma = library("maths.lib");
interpolator_cosine_test = it.interpolator_cosine(gen, idv)
with {
    gen(idx) = waveform {0.0, 1.0, 4.0, 9.0, 16.0}, int(ma.modulo(idx, 5)) : rdtable;
    step = 0.25;
    idxFloat = ma.modulo((+(step)~_) - step, 4.0);
    idv = it.make_idv(idxFloat);
};
```

Four points interpolators

(it.)interpolator_four_points

Generic interpolator on interpolator_four_points points (previous, current and two next indexes), assuming an increasing index.

Usage

```
interpolator_four_points(gen, idv, interpolate_four_points) : si.bus(outputs(gen))
```

Where:

- `gen`: a circuit with an 'idv' reader input that produces N outputs
- `idv`: a fractional read index expressed as a float value, or a (int,frac) pair
- `interpolate_four_points`: a four points interpolation function

Test

```
it = library("interpolators.lib");
ma = library("maths.lib");
interpolator_four_points_test = it.interpolator_four_points(gen, idv, it.interpolate_cubic)
with {
    gen(idx) = waveform {-1.0, 2.0, 1.0, 4.0, 7.0, 3.0}, int(ma.modulo(idx, 6)) : rdtable;
    step = 0.25;
    idxFloat = ma.modulo((+(step)~_) - step, 5.0);
    idv = it.make_idv(idxFloat);
```

```
};
```

(it.)interpolator_cubic

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

Usage

```
interpolator_cubic(gen, idv) : si.bus(outputs(gen))
```

Where:

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

Test

```
it = library("interpolators.lib");
ma = library("maths.lib");
interpolator_cubic_test = it.interpolator_cubic(gen, idv)
with {
    gen(idx) = waveform {-1.0, 2.0, 1.0, 4.0, 7.0, 3.0}, int(ma.modulo(idx, 6)) : rdtable;
    step = 0.25;
    idxFloat = ma.modulo((+(step)~_) - step, 5.0);
    idv = it.make_idv(idxFloat);
};
```

(it.)interpolator_select

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

Usage

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

Where:

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

Test

```
it = library("interpolators.lib");
ma = library("maths.lib");
interpolator_select_test = it.interpolator_select(gen, idv, 2)
with {
    gen(idx) = waveform {-1.0, 2.0, 1.0, 4.0, 7.0, 3.0}, int(ma.modulo(idx, 6)) : rdtable;
    step = 0.25;
    idxFloat = ma.modulo((+(step)~_) - step, 5.0);
    idv = it.make_idv(idxFloat);
};
```

Generic piecewise linear interpolation

(it.)lerp

Linear interpolation between two points.

Usage

```
lerp(x0, x1, y0, y1, x) : si.bus(1);
```

Where:

- `x0`: x-coordinate origin
- `x1`: x-coordinate destination
- `y0`: y-coordinate origin
- `y1`: y-coordinate destination
- `x`: x-coordinate input

Test

```
it = library("interpolators.lib");
lerp_test = it.lerp(0.0, 10.0, -5.0, 5.0, 2.5);
```

(it.)piecewise

Linear piecewise interpolation between N points.

Usage

```
piecewise(xList, yList, x) : si.bus(1);
```

Where:

- `xList`: x-coordinates list

- `yList`: y-coordinates list
- `x`: x-coordinate input

Example test program The code below will output the values of linear segments going through the y coordinates as the input goes from -5 to 5:

```
x = hslider("x", -5, -5.0, 5.0, .001);
process = it.piecewise((-5, -3, 0, 3, 5), (2, 0, 3, -3, -2), x);
```

Test

```
it = library("interpolators.lib");
piecewise_test = it.piecewise((-5, -2, 0, 3), (1, 0, 4, -1), os.osc(0.1));
```

Lagrange based interpolators

`(it.)lagrangeCoeffs`

This is a function to generate N + 1 coefficients for an Nth-order Lagrange basis polynomial with arbitrary spacing of the points.

Usage

```
lagrangeCoeffs(N, xCoordsList, x) : si.bus(N + 1)
```

Where:

- `N`: order of the interpolation filter, known at compile-time
- `xCoordsList`: a list of N + 1 elements determining the x-axis coordinates of N + 1 values, known at compile-time
- `x`: a fractional position on the x-axis to obtain the interpolated y-value

Test

```
it = library("interpolators.lib");
lagrangeCoeffs_test = it.lagrangeCoeffs(2, (0.0, 0.5, 1.0), 0.25);
```

References

- https://ccrma.stanford.edu/~jos/pasp/Lagrange_Interpolation.html
- https://en.wikipedia.org/wiki/Lagrange_polynomial

`(it.)lagrangeInterpolation`

Nth-order Lagrange interpolator to interpolate between a set of arbitrarily spaced N + 1 points.

Usage

```
x , yCoords : lagrangeInterpolation(N, xCoordsList) : _
```

Where:

- `N`: order of the interpolator, known at compile-time
- `xCoordsList`: a list of N + 1 elements determining the x-axis spacing of the points, known at compile-time
- `x`: an x-axis position to interpolate between the y-values
- `yCoords`: N + 1 elements determining the values of the interpolation points

Example: find the centre position of a four-point set using an order-3 Lagrange function fitting the equally-spaced points [2, 5, -1, 3]:

```
N = 3;
xCoordsList = (0, 1, 2, 3);
x = N / 2.0;
yCoords = 2, 5, -1, 3;
process = x, yCoords : it.lagrangeInterpolation(N, xCoordsList);
```

which outputs ~1.938.

- Example: output the dashed curve showed on the Wikipedia page (top figure in https://en.wikipedia.org/wiki/Lagrange_polynomial):

```
N = 3;
xCoordsList = (-9, -4, -1, 7);
x = os.phasor(16, 1) - 9;
yCoords = 5, 2, -2, 9;
process = x, yCoords : it.lagrangeInterpolation(N, xCoordsList);
```

Test

```
it = library("interpolators.lib");
lagrangeInterpolation_test = (lagrange_x, lagrange_y0, lagrange_y1, lagrange_y2, lagrange_y3
with {
    lagrange_x = 1.5;
    lagrange_y0 = 2.0;
    lagrange_y1 = 5.0;
    lagrange_y2 = -1.0;
    lagrange_y3 = 3.0;
};
```

References

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

- Sanfilippo and Parker 2021, "Combining zeroth and first-order analysis with Lagrange polynomials to reduce artefacts in live concatenative granular processing." Proceedings of the DAFx conference 2021, Vienna, Austria.
- https://dafx2020.mdw.ac.at/proceedings/papers/DAFx20in21_paper_38.pdf

---

(it.)frdtable

Look-up circular table with Nth-order Lagrange interpolation for fractional indexes. The index is wrapped-around and the table is cycles for an index span of size S, which is the table size in samples.

Usage

```
frdtable(N, S, init, idx) : _
```

Where:

- `N`: Lagrange interpolation order, known at compile-time
- `S`: table size in samples, known at compile-time
- `init`: the initial table content, known at compile-time
- `idx`: fractional index wrapped-around 0 and S

Example test program Test the effectiveness of the 5th-order interpolation scheme by creating a table look-up oscillator using only 16 points of a sinewave; compare the result with a non-interpolated version:

```
N = 5;
S = 16;
index = os.phasor(S, 1000);
process = rdtable(S, os.sinwaveform(S), int(index)) ,
          it.frdtable(N, S, os.sinwaveform(S), index);
```

Test

```
it = library("interpolators.lib");
os = library("oscillators.lib");
frdtable_test = it.frdtable(3, 16, os.sinwaveform(16), os.phasor(16, 200));
```

(it.)frwtable

Look-up updatable circular table with Nth-order Lagrange interpolation for fractional indexes. The index is wrapped-around and the table is circular indexes ranging from 0 to S, which is the table size in samples.

Usage

```
frwtable(N, S, init, w_idx, x, r_idx) : _
```

Where:

- `N`: Lagrange interpolation order, known at compile-time
- `S`: table size in samples, known at compile-time
- `init`: the initial table content, known at compile-time
- `w_idx`: it should be an INT between 0 and S - 1
- `x`: input signal written on the w_idx positions
- `r_idx`: fractional index wrapped-around 0 and S

Example test program Test the effectiveness of the 5th-order interpolation scheme by creating a table look-up oscillator using only 16 points of a sinewave; compare the result with a non-interpolated version:

```
N = 5;
S = 16;
rIdx = os.phasor(S, 300);
wIdx = ba.period(S);
process = rwtable(S, os.sinwaveform(S), wIdx, os.sinwaveform(S), int(rIdx)) ,
          it.frwtable(N, S, os.sinwaveform(S), wIdx, os.sinwaveform(S), rIdx);
```

Test

```
it = library("interpolators.lib");
os = library("oscillators.lib");
ba = library("basics.lib");
frwtable_test = it.frwtable(3, 16, os.sinwaveform(16), ba.period(16), os.osc(220), os.phaso
```

Misc functions

(it.)remap

Linearly map from an input domain to an output range.

Usage

```
_ : remap(from1, from2, to1, to2) : _
```

Where:

- `from1`: the domain's lower bound.
- `from2`: the domain's upper bound.
- `to1`: the range's lower bound.
- `to2`: the range's upper bound.

Note that having `from1` == `from2` in the mapping will cause a division by zero that has to be taken in account.

Example test program An oscillator remapped from [-1., 1.] to [100., 1000.]:

```
os.osc(440) : it.remap(-1., 1., 100., 1000.)
```

Test

```
it = library("interpolators.lib");
os = library("oscillators.lib");
remap_test = it.remap(-1.0, 1.0, 100.0, 1000.0, os.osc(0.5));
```

linearalgebra.lib

Linear Algebra library. Its official prefix is `la`.

This library provides mathematical tools for matrix and vector operations in Faust. It includes basic arithmetic, dot products, outer products, matrix inversion, determinant computation, and utilities for linear transformations and numerical analysis.

This library adds some new linear algebra functions:

`determinant`

`minor`

`inverse`

`transpose2`

`matMul` matrix multiplication

`identity`

`diag`

How does it work? An `NxM` matrix can be flattened into a bus `si.bus(N*M)`. These buses can be passed to functions as long as `N` and sometimes `M` (if the matrix need not be square) are passed too.

Some things to think about going forward

Implications for ML in Faust Next step of making a "Dense"/"Linear" layer from machine learning. Where in the libraries should `ReLU` go? What about 3D tensors instead of 2D matrices? Image convolutions take place on 3D tensors shaped `HxWxC`.

#####Design of matMul

Currently the design is `matMul(J, K, L, M, leftHandMat, rightHandMat)` where `leftHandMat` is `JxK` and `rightHandMat` is `LxM`.

It would also be neat to have `matMul(J, K, rightHandMat, L, M, leftHandMat)`.

Then a "packed" matrix could be consistently stored as a combination of a 2-channel "header" `N, M` and the values `si.bus(N*M)`.

This would ultimately enable `result = packedLeftHand : matMul(packedRightHand);` for the equivalent numpy code: `result = packedLeftHand @ packedRightHand;`.

References

- https://github.com/grame-cncm/faustlibraries/blob/master/linearalgebra.lib

---

`(la.)determinant`

Calculates the determinant of a bus that represents an `NxN` matrix.

Usage

```
si.bus(N*N) : determinant(N) : _
```

Where:

- `N`: the size of each axis of the matrix.

Test

```
la = library("linearalgebra.lib");
determinant_test = (1, 2, 3, 4) : la.determinant(2);
```

`(la.)minor`

An utility for finding the matrix minor when inverting a matrix. It returns the determinant of the submatrix formed by deleting the row at index `ROW` and column at index `COL`. The following implementation doesn't work but looks simple.

```
minor(N, ROW, COL) = par(r, N, par(c, N, select2((ROW==r)||(COL==c),_,!))) : determinant(N-1
```

Usage

```
si.bus(N*N) : minor(N, ROW, COL) : _
```

Where:

- `N`: the size of each axis of the matrix.
- `ROW`: the selected position on 0th dimension of the matrix (`0 <= ROW < N`)
- `COL`: the selected position on the 1st dimension of the matrix (`0 <= COL < N`)

Test

```
la = library("linearalgebra.lib");
minor_test = (1, 2, 3, 0, 4, 5, 7, 8, 9) : la.minor(3, 1, 1);
```

References

- https://en.wikipedia.org/wiki/Minor_(linear_algebra)#First_minor

---

`(la.)inverse`

Inverts a matrix. The incoming bus represents an `NxN` matrix. Note, this is an unsafe operation since not all matrices are invertible.

Usage

```
si.bus(N*N) : inverse(N) : si.bus(N*N)
```

Where:

- `N`: the size of each axis of the matrix.

Test

```
la = library("linearalgebra.lib");
inverse_test = (4, 7, 2, 6) : la.inverse(2);
```

`(la.)transpose2`

Transposes an `NxM` matrix stored in row-major order, resulting in an `MxN` matrix stored in row-major order.

Usage

```
si.bus(N*M) : transpose2(N, M) : si.bus(M*N)
```

Where:

- `N`: the number of rows in the input matrix
- `M`: the number of columns in the input matrix

Test

```
la = library("linearalgebra.lib");
transpose2_test = (1, 2, 3, 4, 5, 6) : la.transpose2(2, 3);
```

(la.)matMul

Multiply a JxK matrix (mat1) and an LxM matrix (mat2) to produce a JxM matrix. Note that K==L. Both matrices should use row-major order. In terms of numpy, this function is mat1 @ mat2.

Usage

```
matMul(J, K, L, M, si.bus(J*K), si.bus(L*M)) : si.bus(J*M)
```

Where:

- J: the number of rows in mat1
- K: the number of columns in mat1
- L: the number of rows in mat2
- M: the number of columns in mat2

Test

```
la = library("linearalgebra.lib");
matMul_test = (1, 2, 3, 4), (5, 6, 7, 8) : la.matMul(2, 2, 2, 2);
```

(la.)identity

Creates an NxN identity matrix.

Usage

```
identity(N) : si.bus(N*N)
```

Where:

- N: The size of each axis of the identity matrix.

Test

```
la = library("linearalgebra.lib");
identity_test = la.identity(3);
```

(la.)diag

Creates a diagonal matrix of size `NxN` with specified values along the diagonal.

Usage

```
si.bus(N) : diag(N) : si.bus(N*N)
```

Where:

- `N`: The size of each axis of the matrix.

Test

```
la = library("linearalgebra.lib");
diag_test = (1, 2, 3) : la.diag(3);
```

maths.lib

Maths library. Its official prefix is `ma`.

This library provides mathematical functions and utilities for numerical computations in Faust. It includes trigonometric, exponential, logarithmic, and statistical functions, constants, and complex-number operations used throughout Faust DSP and control code.

The Maths library is organized into 1 section:

- Functions Reference

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

References

- https://github.com/grame-cncm/faustlibraries/blob/master/maths.lib

Functions Reference

(ma.)SR

Current sampling rate given at init time. Constant during program execution.

Usage

```
SR : _
```

Where:

- `SR`: initialization-time sampling rate constant

Test

```
ma = library("maths.lib");
SR_test = ma.SR;
```

(ma.)T

Current sample duration in seconds computed from the sampling rate given at init time. Constant during program execution.

Usage

```
T : _
```

Where:

- `T`: sample duration (`1/SR`) constant

Test

```
ma = library("maths.lib");
T_test = ma.T;
```

(ma.)BS

Current block-size. Can change during the execution at each block.

Usage

```
BS : _
```

Where:

- `BS`: current processing block size

Test

```
ma = library("maths.lib");
BS_test = ma.BS;
```

`(ma.)PI`

Constant PI in double precision.

Usage

`PI : _`

Where:

- `PI`: double-precision constant

Test

```
ma = library("maths.lib");
PI_test = ma.PI;
```

`(ma.)deg2rad`

Convert degrees to radians.

Usage

`45. : deg2rad`

Where:

- input: angle in degrees to convert

Test

```
ma = library("maths.lib");
deg2rad_test = 45.0 : ma.deg2rad;
```

`(ma.)rad2deg`

Convert radians to degrees.

Usage

`ma.PI : rad2deg`

Where:

- input: angle in radians to convert

Test

```
ma = library("maths.lib");
rad2deg_test = ma.PI : ma.rad2deg;
```

(ma.)E

Constant e in double precision.

Usage

```
E : _
```

Where:

- `E`: double-precision Euler's number constant

Test

```
ma = library("maths.lib");
E_test = ma.E;
```

(ma.)EPSILON

Constant EPSILON available in simple/double/quad precision, as defined in the floating-point standard and machine epsilon, that is smallest positive number such that `1.0 + EPSILON != 1.0`.

Usage

```
EPSILON : _
```

Where:

- `EPSILON`: machine epsilon constant for the current floating-point precision

Test

```
ma = library("maths.lib");
EPSILON_test = ma.EPSILON;
```

(ma.)MIN

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

Usage

```
MIN : _
```

Where:

- `MIN`: minimal positive normalized value for the current precision

Test

```
ma = library("maths.lib");
MIN_test = ma.MIN;
```

(ma.)MAX

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

Usage

```
MAX : _
```

Where:

- `MAX`: maximal finite value for the current precision

Test

```
ma = library("maths.lib");
MAX_test = ma.MAX;
```

(ma.)FTZ

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

Usage

```
_ : FTZ : _
```

Where:

- `x`: input signal to flush if its magnitude is subnormal

Test

```
ma = library("maths.lib");
FTZ_test = (ma.MIN * 0.5) : ma.FTZ;
```

References

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

`(ma.)copysign`

Changes the sign of x (first input) to that of y (second input).

Usage

```
_,_ : copysign : _
```

Where:

- `x`: value whose magnitude is preserved
- `y`: value providing the sign

Test

```
ma = library("maths.lib");
copysign_test = (-1.0, 2.0) : ma.copysign;
```

`(ma.)neg`

Invert the sign (-x) of a signal.

Usage

```
_ : neg : _
```

Where:

- `x`: value to negate

Test

```
ma = library("maths.lib");
neg_test = 3.5 : ma.neg;
```

`(ma.)not`

Bitwise `not` implemented with xor as `not(x) = x xor -1;`. So working regardless of the size of the integer, assuming negative numbers in two's complement.

Usage

```
_ : not : _
```

Where:

- `x`: integer input value

Test

```
ma = library("maths.lib");
not_test = 5 : ma.not;
```

`(ma.)sub(x,y)`

Subtract `x` and `y`.

Usage

```
_,_ : sub : _
```

Where:

- `x`: first operand
- `y`: second operand

Test

```
ma = library("maths.lib");
sub_test = (3, 10) : ma.sub;
```

`(ma.)inv`

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

Usage

```
_ : inv : _
```

Where:

- `x`: denominator input (non-zero)

Test

```
ma = library("maths.lib");
inv_test = 4.0 : ma.inv;
```

(ma.)cbrt

Computes the cube root of of the input signal.

Usage

```
_ : cbrt : _
```

Where:

- `x`: value whose cube root is computed

Test

```
ma = library("maths.lib");
cbrt_test = 8.0 : ma.cbrt;
```

(ma.)hypot

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

Usage

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

Where:

- `x`: first operand
- `y`: second operand

Test

```
ma = library("maths.lib");
hypot_test = (3.0, 4.0) : ma.hypot;
```

(ma.)ldexp

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

Usage

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

Where:

- `x`: significand input
- `n`: exponent (integer) input

Test

```
ma = library("maths.lib");
ldexp_test = (1.5, 3) : ma.ldexp;
```

`(ma.)scalb`

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

Usage

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

Where:

- `x`: significand input
- `n`: exponent (integer) input

Test

```
ma = library("maths.lib");
scalb_test = (2.0, -1) : ma.scalb;
```

`(ma.)log1p`

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

Usage

```
_ : log1p : _
```

Where:

- `x`: offset used in `log(1 + x)` (must be greater than -1)

Test

```
ma = library("maths.lib");
log1p_test = 0.5 : ma.log1p;
```

`(ma.)logb`

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

Usage

```
_ : logb : _
```

Where:

- `x`: positive value whose exponent part is returned

Test

```
ma = library("maths.lib");
logb_test = 8.0 : ma.logb;
```

`(ma.)ilogb`

Return exponent of the input signal as an integer number.

Usage

```
_ : ilogb : _
```

Where:

- `x`: positive value whose exponent part is returned

Test

```
ma = library("maths.lib");
ilogb_test = 8.0 : ma.ilogb;
```

`(ma.)log2`

Returns the base 2 logarithm of x.

Usage

```
_ : log2 : _
```

Where:

- `x`: positive value whose base-2 logarithm is computed

Test

```
ma = library("maths.lib");
log2_test = 8.0 : ma.log2;
```

(ma.)expm1

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

Usage

```
_ : expm1 : _
```

Where:

- `x`: input value used for the `exp(x) - 1` computation

Test

```
ma = library("maths.lib");
expm1_test = 0.5 : ma.expm1;
```

(ma.)acosh

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

Usage

```
_ : acosh : _
```

Where:

- `x`: input value (greater than or equal to 1)

Test

```
ma = library("maths.lib");
acosh_test = 1.5 : ma.acosh;
```

(ma.)asinh

Computes the inverse hyperbolic sine of the input signal.

Usage

```
_ : asinh : _
```

Where:

- `x`: input value

Test

```
ma = library("maths.lib");
asinh_test = 0.5 : ma.asinh;
```

(ma.)atanh

Computes the inverse hyperbolic tangent of the input signal.

Usage

```
_ : atanh : _
```

Where:

- `x`: input value in (-1, 1)

Test

```
ma = library("maths.lib");
atanh_test = 0.5 : ma.atanh;
```

(ma.)sinh

Computes the hyperbolic sine of the input signal.

Usage

```
_ : sinh : _
```

Where:

- `x`: input value

Test

```
ma = library("maths.lib");
sinh_test = 0.5 : ma.sinh;
```

(ma.)cosh

Computes the hyperbolic cosine of the input signal.

Usage

```
_ : cosh : _
```

Where:

- `x`: input value

Test

```
ma = library("maths.lib");
cosh_test = 0.5 : ma.cosh;
```

(ma.)tanh

Computes the hyperbolic tangent of the input signal.

Usage

```
_ : tanh : _
```

Where:

- `x`: input value

Test

```
ma = library("maths.lib");
tanh_test = 0.5 : ma.tanh;
```

(ma.)erf

Computes the error function of the input signal.

Usage

```
_ : erf : _
```

Where:

- `x`: input value

Test

```
ma = library("maths.lib");
erf_test = 0.5 : ma.erf;
```

`(ma.)erfc`

Computes the complementary error function of the input signal.

Usage

```
_ : erfc : _
```

Where:

- `x`: input value

Test

```
ma = library("maths.lib");
erfc_test = 0.5 : ma.erfc;
```

`(ma.)gamma`

Computes the gamma function of the input signal.

Usage

```
_ : gamma : _
```

Where:

- `x`: positive input value

Test

```
ma = library("maths.lib");
gamma_test = 3.0 : ma.gamma;
```

`(ma.)lgamma`

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

Usage

```
_ : lgamma : _
```

Where:

- `x`: positive input value

Test

```
ma = library("maths.lib");
lgamma_test = 3.0 : ma.lgamma;
```

(ma.)J0

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

Usage

```
_ : J0 : _
```

Where:

- `x`: input value

Test

```
ma = library("maths.lib");
J0_test = 1.0 : ma.J0;
```

(ma.)J1

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

Usage

```
_ : J1 : _
```

Where:

- `x`: input value

Test

```
ma = library("maths.lib");
J1_test = 1.0 : ma.J1;
```

(ma.)Jn

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

Usage

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

Where:

- `n`: integer order
- `x`: input value

Test

```
ma = library("maths.lib");
Jn_test = (2, 1.0) : ma.Jn;
```

(ma.)Y0

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

Usage

```
_ : Y0 : _
```

Where:

- `x`: positive input value

Test

```
ma = library("maths.lib");
Y0_test = 1.0 : ma.Y0;
```

(ma.)Y1

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

Usage

```
_ : Y0 : _
```

Where:

- `x`: positive input value

Test

```
ma = library("maths.lib");
Y1_test = 1.0 : ma.Y1;
```

`(ma.)Yn`

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

Usage

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

Where:

- `n`: integer order
- `x`: positive input value

Test

```
ma = library("maths.lib");
Yn_test = (2, 1.0) : ma.Yn;
```

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

Just for compatibility…

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

`(ma.)np2`

Gives the next power of 2 of x.

Usage

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

Where:

- `n`: an integer

Test

```
ma = library("maths.lib");
np2_test = 5 : ma.np2;
```

(ma.)frac

Gives the fractional part of n.

Usage

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

Where:

- `n`: a decimal number

Test

```
ma = library("maths.lib");
frac_test = 3.75 : ma.frac;
```

(ma.)modulo

Modulus operation using the `(x%y+y)%y` formula to ensures the result is always non-negative, even if `x` is negative.

Usage

```
modulo(x,y) : _
```

Where:

- `x`: the numerator
- `y`: the denominator

Test

```
ma = library("maths.lib");
modulo_test = (-3, 4) : ma.modulo;
```

(ma.)isnan

Return non-zero if x is a NaN.

Usage

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

Where:

- `x`: signal to analyse

Test

```
ma = library("maths.lib");
isnan_test = 1.0 : ma.isnan;
```

(ma.)isinf

Return non-zero if x is a positive or negative infinity.

Usage

```
isinf(x)
_ : isinf : _
```

Where:

- `x`: signal to analyse

Test

```
ma = library("maths.lib");
isinf_test = 1.0 : ma.isinf;
```

(ma.)chebychev

Chebychev transformation of order N.

Usage

```
_ : chebychev(N) : _
```

Where:

- `N`: the order of the polynomial, a constant numerical expression

Semantics

```
T[0](x) = 1,
T[1](x) = x,
T[n](x) = 2x*T[n-1](x) - T[n-2](x)
```

Test

```
ma = library("maths.lib");
chebychev_test = 0.5 : ma.chebychev(3);
```

References

- http://en.wikipedia.org/wiki/Chebyshev_polynomial

(ma.)chebychevpoly

Linear combination of the first Chebyshev polynomials.

Usage

```
_ : chebychevpoly((c0,c1,...,cn)) : _
```

Where:

- `cn`: the different Chebychevs polynomials such that: chebychevpoly((c0,c1,...,cn)) = Sum of chebychev(i)*ci

Test

```
ma = library("maths.lib");
chebychevpoly_test = 0.5 : ma.chebychevpoly((1, 0, 1));
```

References

- http://www.csounds.com/manual/html/chebyshevpoly.html

(ma.)diffn

Negated first-order difference.

Usage

```
_ : diffn : _
```

Where:

- `x`: input signal

Test

```
ma = library("maths.lib");
os = library("oscillators.lib");
diffn_test = os.osc(440) : ma.diffn;
```

`(ma.)signum`

The signum function signum(x) is defined as -1 for x<0, 0 for x==0, and 1 for x>0.

Usage

```
_ : signum : _
```

Where:

- `x`: input value

Test

```
ma = library("maths.lib");
signum_test = (-5.0) : ma.signum;
```

`(ma.)nextpow2`

The nextpow2(x) returns the lowest integer m such that 2^m >= x.

Usage

```
2^nextpow2(n) : _
```

Useful for allocating delay lines, e.g.,

```
delay(2^nextpow2(maxDelayNeeded), currentDelay);
```

Where:

- `n`: positive value whose next power-of-two exponent is computed

Test

```
ma = library("maths.lib");
nextpow2_test = 10.0 : ma.nextpow2;
```

(ma.)zc

Indicator function for zero-crossing: it returns 1 if a zero-crossing occurs, 0 otherwise.

Usage

```
_ : zc : _
```

Where:

- `x`: input signal to monitor for zero crossings

Test

```
ma = library("maths.lib");
os = library("oscillators.lib");
zc_test = os.osc(440) : ma.zc;
```

(ma.)unwrap

Unwrap the input signal so that successive output values never differ by more than `pi`, switching to a 2*pi-complementary value when needed.

Usage

```
_ : unwrap(pi) : _
```

Where:

- `pi`: maximum discontinuity between the output values (typically `ma.PI`)

Test

```
ma = library("maths.lib");
os = library("oscillators.lib");
unwrap_test = os.oscrc(100) : ma.unwrap(ma.PI);
```

Example test program

```
process = 0 - os.oscrc(1000)          // the true phase is either -PI or +PI
        : an.resonator(1,1000) : !,_  // oscillates between -PI and +PI
        : ma.unwrap(ma.PI);           // oscillates near +PI
```

(ma.)primes

Return the n-th prime using a waveform primitive. Note that primes(0) is 2, primes(1) is 3, and so on. The waveform is length 2048, so the largest precomputed prime is primes(2047) which is 17863.

Usage

```
_ : primes : _
```

Where:

- `x`: index of the prime number sequence (0-based).

Test

```
ma = library("maths.lib");
primes_test = 10 : ma.primes;
```

mi.lib

This ongoing work is the fruit of a collaboration between GRAME-CNCM and the ANIS (Arts Numériques et Immersions Sensorielles) research group from GIPSA-Lab (Université Grenoble Alpes).

This library implements basic 1-DoF mass-interaction physics algorithms, allowing to declare and connect physical elements (masses, springs, non linear interactions, etc.) together to form topological networks. Models can be assembled by hand, however in more complex scenarios it is recommended to use a scripting tool (such as MIMS) to generate the FAUST signal routing for a given physical network. Its official prefix is `mi`.

Video introduction to Mass Interaction

LAC 2019 Paper

Sources

The core mass-interaction algorithms implemented in this library are in the public domain and are disclosed in the following scientific publications:

- Claude Cadoz, Annie Luciani, Jean-Loup Florens, Curtis Roads and Françoise Chabade. Responsive Input Devices and Sound Synthesis by Stimulation of Instrumental Mechanisms: The Cordis System. Computer Music Journal, Vol 8. No. 3, 1984.
- Claude Cadoz, Annie Luciani and Jean Loup Florens. CORDIS-ANIMA: A Modeling and Simulation System for Sound and Image Synthesis: The General Formalism. Computer Music Journal. Vol. 17, No. 1, 1993.

- Alexandros Kontogeorgakopoulos and Claude Cadoz. Cordis Anima Physical Modeling and Simulation System Analysis. In Proceedings of the Sound and Music Computing Conference (SMC-07), Lefkada, Greece, 2007.
- Nicolas Castagne, Claude Cadoz, Ali Allaoui and Olivier Tache. G3: Genesis Software Environment Update. In Proceedings of the International Computer Music Conference (ICMC-09), Montreal, Canada, 2009.
- Nicolas Castagné and Claude Cadoz. Genesis 3: Plate-forme pour la création musicale à l'aide des modèles physiques Cordis-Anima. In Proceedings of the Journée de l'Informatique Musicale, Grenoble, France, 2009.
- Edgar Berdahl and Julius O. Smith. An Introduction to the Synth-A-Modeler Compiler: Modular and Open-Source Sound Synthesis using Physical Models. In Proceedings of the Linux Audio Conference (LAC-12), Stanford, USA, 2012.
- James Leonard and Claude Cadoz. Physical Modelling Concepts for a Collection of Multisensory Virtual Musical Instruments. In Proceedings of the New Interfaces for Musical Expression (NIME-15) Conference, Baton Rouge, USA, 2015.

The MI library is organized into 3 sections:

- Utility Functions
- Mass Algorithms
- Interaction Algorithms

References

- https://github.com/grame-cncm/faustlibraries/blob/master/mi.lib

Utility Functions

These utility functions are used to help certain operations (e.g. define initial positions and velocities for physical elements).

`(mi.)initState`

Used to set initial delayed position values that must be initialised at step 0 of the physics simulation.

If you develop any of your own modules, you will need to use this (see `mass` and `springDamper` algorithm codes for examples).

Usage

```
x : initState(x0) : _
```

Where:

- `x`: position value signal
- `x0`: initial value for position

Test

```
mi = library("mi.lib");
initState_test = button("impulse") : mi.initState(1.0);
```

Mass Algorithms

All mass-type physical element functions are declared here. They all expect to receive a force input signal and produce a position signal. All physical parameters are expressed in sample-rate dependant values.

`(mi.)mass`

Implementation of a punctual mass element. Takes an input force and produces output position.

Usage

```
mass(m, grav, x0, xr0),_ : _
```

Where:

- `m`: mass value
- `grav`: gravity force value
- `x0`: initial position
- `xr0`: initial delayed position (inferred from initial velocity)

Test

```
mi = library("mi.lib");
mass_test = 0 : mi.mass(1.0, 0.0, 0.0, 0.0);
```

`(mi.)oscil`

Implementation of a simple linear harmonic oscillator. Takes an input force and produces output position.

Usage

```
oscil(m, k, z, grav, x0, xr0),_ : _
```

Where:

- `m`: mass value

- `k`: stiffness value
- `z`: damping value
- `grav`: gravity force value
- `x0`: initial position
- `xr0`: initial delayed position (inferred from initial velocity)

Test

```
mi = library("mi.lib");
oscil_test = 0 : mi.oscil(1.0, 0.5, 0.1, 0.0, 0.0, 0.0);
```

`(mi.)ground`

Implementation of a fixed point element. The position output produced by this module never changes, however it still expects a force input signal (for compliance with connection rules).

Usage

```
ground(x0),_ : _
```

Where:

- `x0`: initial position

Test

```
mi = library("mi.lib");
ground_test = 0 : mi.ground(0.0);
```

`(mi.)posInput`

Implementation of a position input module (driven by an outside signal). Takes two signal inputs: incoming force (which doesn't affect position) and the driving position signal.

Usage

```
posInput(x0),_,_ : _
```

Where:

- `x0`: initial position

Test

```
mi = library("mi.lib");
os = library("oscillators.lib");
posInput_test = 0, os.osc(1) : mi.posInput(0.0);
```

Interaction Algorithms

All interaction-type physical element functions are declared here. They each expect to receive two position signals (coming from the two mass-elements that they connect) and produce two equal and opposite force signals that must be routed back to the mass elements' inputs. All physical parameters are expressed in sample-rate dependant values.

(mi.)spring

Implementation of a linear elastic spring interaction.

Usage

```
spring(k, x1r, x2r),_,_ : _,_
```

Where:

- `k`: stiffness value
- `x1r`: initial delayed position of mass 1 (unused here)
- `x2r`: initial delayed position of mass 2 (unused here)

Test

```
mi = library("mi.lib");
spring_test = mi.spring(10.0, 0.0, 0.0, 0.1, -0.1);
```

(mi.)damper

Implementation of a linear damper interaction. Beware: in 32bit precision mode, damping forces can become truncated if position values are not centered around zero!

Usage

```
damper(z, x1r, x2r),_,_ : _,_
```

Where:

- `z`: damping value
- `x1r`: initial delayed position of mass 1

- `x2r`: initial delayed position of mass 2

Test

```
mi = library("mi.lib");
damper_test = mi.damper(0.5, 0.0, 0.0, 0.2, -0.2);
```

(mi.)springDamper

Implementation of a linear viscoelastic spring-damper interaction (a combination of the spring and damper modules).

Usage

```
springDamper(k, z, x1r, x2r),_,_ : _,_
```

Where:

- `k`: stiffness value
- `z`: damping value
- `x1r`: initial delayed position of mass 1
- `x2r`: initial delayed position of mass 2

Test

```
mi = library("mi.lib");
springDamper_test = mi.springDamper(5.0, 0.3, 0.0, 0.0, 0.1, -0.1);
```

(mi.)nlSpringDamper2

Implementation of a non-linear viscoelastic spring-damper interaction containing a quadratic term (function of squared distance). Beware: at high displacements, this interaction will break numerical stability conditions ! The `nlSpringDamperClipped` is a safer option.

Usage

```
nlSpringDamper2(k, q, z, x1r, x2r),_,_ : _,_
```

Where:

- `k`: linear stiffness value
- `q`: quadratic stiffness value
- `z`: damping value
- `x1r`: initial delayed position of mass 1
- `x2r`: initial delayed position of mass 2

Test

```
mi = library("mi.lib");
nlSpringDamper2_test = mi.nlSpringDamper2(5.0, 1.0, 0.2, 0.0, 0.0, 0.1, -0.1);
```

(mi.)nlSpringDamper3

Implementation of a non-linear viscoelastic spring-damper interaction containing a cubic term (function of distance^3). Beware: at high displacements, this interaction will break numerical stability conditions ! The `nlSpringDamperClipped` is a safer option.

Usage

```
nlSpringDamper3(k, q, z, x1r, x2r),_,_ : _,_
```

Where:

- `k`: linear stiffness value
- `q`: cubic stiffness value
- `z`: damping value
- `x1r`: initial delayed position of mass 1
- `x2r`: initial delayed position of mass 2

Test

```
mi = library("mi.lib");
nlSpringDamper3_test = mi.nlSpringDamper3(5.0, 0.5, 0.2, 0.0, 0.0, 0.1, -0.1);
```

(mi.)nlSpringDamperClipped

Implementation of a non-linear viscoelastic spring-damper interaction containing a cubic term (function of distance^3), bound by an upper linear stiffness (hard-clipping).

This bounding means that when faced with strong displacements, the interaction profile will "clip" at a given point and never produce forces higher than the bounding equivalent linear spring, stopping models from becoming unstable.

So far the interaction clips "hard" (with no soft-knee spline interpolation, etc.)

Usage

```
nlSpringDamperClipped(s, c, k, z, x1r, x2r),_,_ : _,_
```

Where:

- `s`: linear stiffness value

- `c`: cubic stiffness value
- `k`: upper-bound linear stiffness value
- `z`: (linear) damping value
- `x1r`: initial delayed position of mass 1
- `x2r`: initial delayed position of mass 2

Test

```
mi = library("mi.lib");
nlSpringDamperClipped_test = mi.nlSpringDamperClipped(5.0, 0.5, 8.0, 0.2, 0.0, 0.0, 0.1, -0.
```

(mi.)nlPluck

Implementation of a piecewise linear plucking interaction. The symmetric function provides a repulsive viscoelastic interaction upon contact, until a tipping point is reached (when the plucking occurs). The tipping point depends both on the stiffness and the distance scaling of the interaction.

Usage

```
nlPluck(knl, scale, z, x1r, x2r),_,_ : _,_
```

Where:

- `knl`: stiffness scaling parameter (vertical stretch of the NL function)
- `scale`: distance scaling parameter (horizontal stretch of the NL function)
- `z`: (linear) damping value
- `x1r`: initial delayed position of mass 1
- `x2r`: initial delayed position of mass 2

Test

```
mi = library("mi.lib");
nlPluck_test = mi.nlPluck(5.0, 0.1, 0.2, 0.0, 0.0, 0.05, -0.05);
```

(mi.)nlBow

Implementation of a non-linear friction based interaction that allows for stick-slip bowing behaviour. Two versions are proposed : a piecewise linear function (very similar to the `nlPluck`) or a mathematical approximation (see Stefan Bilbao's book, Numerical Sound Synthesis).

Usage

```
nlBow(znl, scale, type, x1r, x2r),_,_ : _,_
```

Where:

- `znl`: friction scaling parameter (vertical stretch of the NL function)
- `scale`: velocity scaling parameter (horizontal stretch of the NL function)
- `type`: interaction profile (0 = piecewise linear, 1 = smooth function)
- `x1r`: initial delayed position of mass 1
- `x2r`: initial delayed position of mass 2

Test

```
mi = library("mi.lib");
nlBow_test = mi.nlBow(0.5, 0.1, 1.0, 0.0, 0.0, 0.05, -0.05);
```

(mi.)collision

Implementation of a collision interaction, producing linear visco-elastic repulsion forces when two mass elements are interpenetrating.

Usage

```
collision(k, z, thres, x1r, x2r),_,_ : _,_
```

Where:

- `k`: collision stiffness parameter
- `z`: collision damping parameter
- `thres`: threshold distance for the contact between elements
- `x1r`: initial delayed position of mass 1
- `x2r`: initial delayed position of mass 2

Test

```
mi = library("mi.lib");
collision_test = mi.collision(5.0, 0.2, 0.01, 0.0, 0.0, 0.0, -0.02);
```

(mi.)nlCollisionClipped

Implementation of a collision interaction, producing non-linear visco-elastic repulsion forces when two mass elements are interpenetrating. Bound by an upper stiffness value to maintain stability. This interaction is particularly useful for more realistic contact dynamics (greater difference in velocity provides sharper contacts, and reciprocally).

Usage

```
nlCollisionClipped(s, c, k, z, thres, x1r, x2r),_,_ : _,_
```

Where:

- `s`: collision linear stiffness parameter
- `c`: collision cubic stiffness parameter
- `k`: collision upper-bounding stiffness parameter
- `z`: collision damping parameter
- `thres`: threshold distance for the contact between elements
- `x1r`: initial delayed position of mass 1
- `x2r`: initial delayed position of mass 2

Test

```
mi = library("mi.lib");
nlCollisionClipped_test = mi.nlCollisionClipped(3.0, 0.5, 6.0, 0.2, 0.01, 0.0, 0.0, 0.0, -0.
```

misceffects.lib

Miscellaneous Effects library. Its official prefix is `ef`.

This library contains a collection of diverse audio effects and utilities not included in other specialized Faust libraries. It includes filtering, mixing, time based, pitch shifters, and other creative or experimental signal processing components for sound design and musical applications.

The library is organized into 7 sections:

- Dynamic
- Fibonacci
- Filtering
- Meshes
- Mixing
- Time Based
- Pitch Shifting
- Saturators

References

- https://github.com/grame-cncm/faustlibraries/blob/master/misceffects.lib

Dynamic

(ef.)cubicnl

Cubic nonlinearity distortion. `cubicnl` is a standard Faust function.

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.

Test

```
ef = library("misceffects.lib");
os = library("oscillators.lib");
cubicnl_test = os.osc(440) : ef.cubicnl(0.5, 0.0);
cubicnl_nodc_test = os.osc(440) : ef.cubicnl_nodc(0.5, 0.0);
```

References

- https://ccrma.stanford.edu/~jos/pasp/Cubic_Soft_Clipper.html
- https://ccrma.stanford.edu/~jos/pasp/Nonlinear_Distortion.html

---

(ef.)gate_mono

Mono signal gate. `gate_mono` is a standard Faust function.

Usage

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

Where:

- `thresh`: dB level threshold above which gate opens (e.g., -60 dB)
- `att`: attack time = time constant (sec) for gate to open (e.g., 0.0001 s = 0.1 ms)
- `hold`: hold time = time (sec) gate stays open after signal level < thresh (e.g., 0.1 s)
- `rel`: release time = time constant (sec) for gate to close (e.g., 0.020 s = 20 ms)

Test

```
ef = library("misceffects.lib");
os = library("oscillators.lib");
gate_mono_test = os.osc(440) : ef.gate_mono(-60, 0.0001, 0.1, 0.02);
```

References

- http://en.wikipedia.org/wiki/Noise_gate
- http://www.soundonsound.com/sos/apr01/articles/advanced.asp
- http://en.wikipedia.org/wiki/Gating_(sound_engineering)

(ef.)gate_stereo

Stereo signal gates. `gate_stereo` is a standard Faust function.

Usage

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

Where:

- `thresh`: dB level threshold above which gate opens (e.g., -60 dB)
- `att`: attack time = time constant (sec) for gate to open (e.g., 0.0001 s = 0.1 ms)
- `hold`: hold time = time (sec) gate stays open after signal level < thresh (e.g., 0.1 s)
- `rel`: release time = time constant (sec) for gate to close (e.g., 0.020 s = 20 ms)

Test

```
ef = library("misceffects.lib");
os = library("oscillators.lib");
gate_stereo_test = os.osc(440), os.osc(441) : ef.gate_stereo(-60, 0.0001, 0.1, 0.02);
```

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)

Fibonacci

(ef.)fibonacci

Fibonacci system where the current output is the current input plus the sum of the previous N outputs.

Usage

```
_ : fibonacci(N) : _
```

Where:

- `N`: the Fibonacci system's order, where 2 is standard

Test

```
ef = library("misceffects.lib");
fibonacci_test = 0 : ef.fibonacci(2);
```

Example Generate the famous series: [1, 1, 2, 3, 5, 8, 13, ...]

```
1. : ba.impulsify : fibonacci(2)
```

(ef.)fibonacciGeneral

Fibonacci system with customizable coefficients. The order of the system is inferred from the number of coefficients.

Usage

```
_ : fibonacciGeneral(wave) : _
```

Where:

- `wave`: a waveform such as `waveform{1, 1}`

Test

```
ef = library("misceffects.lib");
fibonacciGeneral_test = 0 : ef.fibonacciGeneral(waveform{2, 3});
```

Example: Use the update equation `y = 2*y' + 3*y'' + 4*y'''`

```
1. : ba.impulsify : fibonacciGeneral(waveform{2, 3, 4})
```

(ef.)fibonacciSeq

First N numbers of the Fibonacci sequence [1, 1, 2, 3, 5, 8, ...] as parallel channels.

Usage

```
fibonacciSeq(N) : si.bus(N)
```

Where:

- `N`: The number of Fibonacci numbers to generate as channels.

Test

```
ef = library("misceffects.lib");
fibonacciSeq_test = ef.fibonacciSeq(5);
```

Filtering

(ef.)speakerbp

Dirt-simple speaker simulator (overall bandpass eq with observed roll-offs above and below the passband). `speakerbp` is a standard Faust function.

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.

Usage

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

Test

```
ef = library("misceffects.lib");
os = library("oscillators.lib");
speakerbp_test = os.osc(440) : ef.speakerbp(100.0, 5000.0);
```

Example Based on measured Celestion G12 (12” speaker):

```
speakerbp(130,5000)
```

(ef.)piano_dispersion_filter

Piano dispersion allpass filter in closed form.

Usage

```
piano_dispersion_filter(M,B,f0)
_ : piano_dispersion_filter(1,B,f0) : +(totalDelay),_ : fdelay(maxDelay) : _
```

Where:

- `M`: number of first-order allpass sections (compile-time only) Keep below 20. 8 is typical for medium-sized piano strings.
- `B`: string inharmonicity coefficient (0.0001 is typical)
- `f0`: fundamental frequency in Hz

Test

```
ef = library("misceffects.lib");
os = library("oscillators.lib");
piano_dispersion_filter_test = os.osc(110) : ef.piano_dispersion_filter(4, 0.0001, 110);
```

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

References

- "Dispersion Modeling in Waveguide Piano Synthesis Using Tunable All-pass Filters", by Jukka Rauhala and Vesa Valimaki, DAFX-2006, pp. 71-76
- http://lib.tkk.fi/Diss/2007/isbn9789512290666/article2.pdf An erratum in Eq. (7) is corrected in Dr. Rauhala's encompassing dissertation (and below).
- http://www.acoustics.hut.fi/research/asp/piano/

`(ef.)stereo_width`

Stereo Width effect using the Blumlein Shuffler technique. `stereo_width` is a standard Faust function.

Usage

```
_,_ : stereo_width(w) : _,_
```

Where:

- `w`: stereo width between 0 and 1

Test

```
ef = library("misceffects.lib");
os = library("oscillators.lib");
stereo_width_test = os.osc(440), os.osc(550) : ef.stereo_width(0.5);
```

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".

References

- "Applications of Blumlein Shuffling to Stereo Microphone Techniques" Michael A. Gerzon, JAES vol. 42, no. 6, June 1994

Meshes

`(ef.)mesh_square`

Square Rectangular Digital Waveguide Mesh.

Usage

```
bus(4*N) : mesh_square(N) : bus(4*N)
```

Where:

- `N`: number of nodes along each edge - a power of two (1,2,4,8,…)

Test

```
ef = library("misceffects.lib");
mesh_square_test = (0,0,0,0) : ef.mesh_square(1);
```

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.

References

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

Mixing

`(ef.)dryWetMixer`

Linear dry-wet mixer for a N inputs and N outputs effect.

Usage

```
si.bus(inputs(FX)) : dryWetMixer(wetAmount, FX) : si.bus(inputs(FX))
```

Where:

- `wetAmount`: the wet amount (0-1). 0 produces only the dry signal and 1 produces only the wet signal
- `FX`: an arbitrary effect (N inputs and N outputs) to apply to the input bus

Test

```
ef = library("misceffects.lib");
os = library("oscillators.lib");
dryWetMixer_test = os.osc(440) : ef.dryWetMixer(0.5, fi.dcblocker);
```

(ef.)dryWetMixerConstantPower

Constant-power dry-wet mixer for a N inputs and N outputs effect.

Usage

```
si.bus(inputs(FX)) : dryWetMixerConstantPower(wetAmount, FX) :si.bus(inputs(FX))
```

Where:

- `wetAmount`: the wet amount (0-1). 0 produces only the dry signal and 1 produces only the wet signal
- `FX`: an arbitrary effect (N inputs and N outputs) to apply to the input bus

Test

```
ef = library("misceffects.lib");
os = library("oscillators.lib");
dryWetMixerConstantPower_test = os.osc(440) : ef.dryWetMixerConstantPower(0.5, fi.dcblocker)
```

(ef.)mixLinearClamp

Linear mixer for `N` buses, each with `C` channels. The output will be a sum of 2 buses determined by the mixing index `mix`. 0 produces the first bus, 1 produces the second, and so on. `mix` is clamped automatically. For example, `mixLinearClamp(4, 1, 1)` will weight its 4 inputs by `(0, 1, 0, 0)`. Similarly, `mixLinearClamp(4, 1, 1.1)` will weight its 4 inputs by `(0,.9,.1,0)`.

Usage

```
si.bus(N*C) : mixLinearClamp(N, C, mix) : si.bus(C)
```

Where:

- `N`: the number of input buses
- `C`: the number of channels in each bus
- `mix`: the mixing index, continuous in [0;N-1].

Test

```
ef = library("misceffects.lib");
mixLinearClamp_test = (1,0,0,0) : ef.mixLinearClamp(4, 1, 1.2);
```

(ef.)mixLinearLoop

Linear mixer for `N` buses, each with `C` channels. Refer to `mixLinearClamp`. `mix` will loop for multiples of `N`. For example, `mixLinearLoop(4, 1, 0)` has the same effect as `mixLinearLoop(4, 1, -4)` and `mixLinearLoop(4, 1, 4)`.

Usage

```
si.bus(N*C) : mixLinearLoop(N, C, mix) : si.bus(C)
```

Where:

- `N`: the number of input buses
- `C`: the number of channels in each bus
- `mix`: the mixing index (N-1) selects the last bus, and 0 or N selects the 0th bus.

Test

```
ef = library("misceffects.lib");
mixLinearLoop_test = (1,0,0,0) : ef.mixLinearLoop(4, 1, -0.3);
```

(ef.)mixPowerClamp

Constant-power mixer for `N` buses, each with `C` channels. The output will be a sum of 2 buses determined by the mixing index `mix`. 0 produces the first bus, 1 produces the second, and so on. `mix` is clamped automatically. `mixPowerClamp(4, 1, 1)` will weight its 4 inputs by `(0, 1./sqrt(2), 0, 0)`. Similarly, `mixPowerClamp(4, 1, 1.5)` will weight its 4 inputs by `(0,.5,.5,0)`.

Usage

```
si.bus(N*C) : mixPowerClamp(N, C, mix) : si.bus(C)
```

Where:

- `N`: the number of input buses
- `C`: the number of channels in each bus
- `mix`: the mixing index, continuous in [0;N-1].

Test

```
ef = library("misceffects.lib");
mixPowerClamp_test = (1,0,0,0) : ef.mixPowerClamp(4, 1, 1.5);
```

`(ef.)mixPowerLoop`

Constant-power mixer for `N` buses, each with `C` channels. Refer to `mixPowerClamp`. `mix` will loop for multiples of `N`. For example, `mixPowerLoop(4, 1, 0)` has the same effect as `mixPowerLoop(4, 1, -4)` and `mixPowerLoop(4, 1, 4)`.

Usage

```
si.bus(N*C) : mixPowerLoop(N, C, mix) : si.bus(C)
```

Where:

- `N`: the number of input buses
- `C`: the number of channels in each bus
- `mix`: the mixing index (N-1) selects the last bus, and 0 or N selects the 0th bus.

Test

```
ef = library("misceffects.lib");
mixPowerLoop_test = (1,0,0,0) : ef.mixPowerLoop(4, 1, -0.5);
```

Time Based

`(ef.)echo`

A simple echo effect. `echo` is a standard Faust function.

Usage

```
_ : echo(maxDuration,duration,feedback) : _
```

Where:

- `maxDuration`: the max echo duration in seconds
- `duration`: the echo duration in seconds
- `feedback`: the feedback coefficient #### Test

```
ef = library("misceffects.lib");
os = library("oscillators.lib");
echo_test = os.osc(440) : ef.echo(0.5, 0.25, 0.4);
```

(ef.)reverseEchoN

Reverse echo effect.

Usage

```
_ : ef.reverseEchoN(N,delay) : si.bus(N)
```

Where:

- `N`: Number of output channels desired (1 or more), a constant numerical expression
- `delay`: echo delay (integer power of 2)

Test

```
ef = library("misceffects.lib");
os = library("oscillators.lib");
reverseEchoN_test = os.osc(440) : ef.reverseEchoN(2, 32);
```

Demo

```
_ : dm.reverseEchoN(N) : _,_
```

Description The effect uses N instances of `reverseDelayRamped` at different phases.

(ef.)reverseDelayRamped

Reverse delay with amplitude ramp.

Usage

```
_ : ef.reverseDelayRamped(delay,phase) : _
```

Where:

- `delay`: echo delay (integer power of 2)
- `phase`: float between 0 and 1 giving ramp delay phase*delay

Test

```
ef = library("misceffects.lib");
os = library("oscillators.lib");
reverseDelayRamped_test = os.osc(440) : ef.reverseDelayRamped(32, 0.6);
```

Demo

```
_ : ef.reverseDelayRamped(32,0.6) : _,_
```

(ef.)uniformPanToStereo

Pan nChans channels to the stereo field, spread uniformly left to right.

Usage

```
si.bus(N) : ef.uniformPanToStereo(N) : _,_
```

Where:

- `N`: Number of input channels to pan down to stereo, a constant numerical expression

Test

```
ef = library("misceffects.lib");
os = library("oscillators.lib");
uniformPanToStereo_test = os.osc(440), os.osc(550), os.osc(660) : ef.uniformPanToStereo(3);
```

Demo

```
_,_,_ : ef.uniformPanToStereo(3) : _,_
```

(ef.)tapeStop

A tape-stop effect, like putting a finger on a vinyl record player.

Usage:

```
_,_ : tapeStop(2, LAGRANGE_ORDER, MAX_TIME_SAMP,
              crossfade, gainAlpha, stopAlpha, stopTime, stop) : _,_
```

```
_ : tapeStop(1, LAGRANGE_ORDER, MAX_TIME_SAMP,
             crossfade, gainAlpha, stopAlpha, stopTime, stop) : _
```

Where:

- `C`: The number of input and output channels.
- `LAGRANGE_ORDER`: The order of the Lagrange interpolation on the delay line. [2-3] recommended.
- `MAX_TIME_SAMP`: Maximum stop time in samples
- `crossfade`: A crossfade in samples to apply when resuming normal playback. Crossfade is not applied during the enabling of the tape-stop.
- `gainAlpha`: During the tape-stop, lower alpha stays louder longer. Safe values are in the range [.01,2].
- `stopAlpha`: `stopAlpha==1` represents a linear deceleration (constant force). `stopAlpha<1` represents an initially weaker, then stronger force. `stopAlpha>1` represents an initially stronger, then weaker force. Safe values are in the range [.01,2].
- `stopTime`: Desired duration of the stop time, in samples.
- `stop`: When `stop` becomes positive, the tape-stop effect will start. When `stop` becomes zero, normal audio will resume via crossfade. #### Test

```
ef = library("misceffects.lib");
os = library("oscillators.lib");
tapeStop_test = os.osc(440), os.osc(441) : ef.tapeStop(2, 3, 44100, 128, 1.0, 1.0, 22050, bu
```

Pitch Shifting

`(ef.)transpose`

A simple pitch shifter based on 2 delay lines. `transpose` is a standard Faust function.

Usage

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

Where:

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

```
ef = library("misceffects.lib");
os = library("oscillators.lib");
transpose_test = os.osc(440) : ef.transpose(1024, 512, 7);
```

Saturators

(ef.)softclipQuadratic

Quadratic softclip nonlinearity.

Usage

```
_ : softclipQuadratic : _
```

Test

```
ef = library("misceffects.lib");
os = library("oscillators.lib");
softclipQuadratic_test = os.osc(440) : ef.softclipQuadratic;
```

References

- U. Zölzer: Digital Audio Signal Processing. John Wiley & Sons Ltd, 2022.

(ef.)wavefold

Wavefolding nonlinearity.

Usage

```
_ : wavefold(width) : _
```

Where:

- `width`: The width of the folded section [0..1] (float).

Test

```
ef = library("misceffects.lib");
os = library("oscillators.lib");
wavefold_test = os.osc(440) : ef.wavefold(0.5);
```

noises.lib

Noises library. Its official prefix is `no`.

This library provides various noise generators and stochastic signal sources for audio synthesis and testing. It includes white, pink, brown, and blue noise, as well as pseudo-random number generators and utilities for decorrelated signals and random modulation in Faust DSP programs.

The Noises library is organized into 1 section:

- Functions Reference

References

- https://github.com/grame-cncm/faustlibraries/blob/master/noises.lib

Functions Reference

`(no.)noise`

White noise generator (outputs random number between -1 and 1). `noise` is a standard Faust function.

Usage

```
noise : _
```

Where:

- output: white noise signal in [-1, 1].

Test

```
no = library("noises.lib");
noise_test = no.noise;
```

`(no.)multirandom`

Generates multiple decorrelated random numbers in parallel.

Usage

```
multirandom(N) : si.bus(N)
```

Where:

- `N`: the number of decorrelated random numbers in parallel, a constant numerical expression

Test

```
no = library("noises.lib");
multirandom_test = no.multirandom(4);
```

(no.)multinoise

Generates multiple decorrelated noises in parallel.

Usage

```
multinoise(N) : si.bus(N)
```

Where:

- `N`: the number of decorrelated random numbers in parallel, a constant numerical expression

Test

```
no = library("noises.lib");
multinoise_test = no.multinoise(3);
```

(no.)noises

A convenient wrapper around multinoise.

Usage

```
noises(N,i) : _
```

Where:

- `N`: the number of decorrelated random numbers in parallel, a constant numerical expression
- `i`: the selected random number (i in [0..N[)

Test

```
no = library("noises.lib");
noises_test = no.noises(4, 2);
```

(no.)dnoise

A deterministic noise burst with a dynamically adjustable seed, enabling consistent recall. Useful for noise variation sensitive applications like replicable/recallable percussion sounds and waveguide excitation.

Usage

```
dnoise(t,sx) : _
```

Where:

- `t`: is a noise burst trigger
- `sx`: defines the range of integer seed multipliers.

Test

```
no = library("noises.lib");
ba = library("basics.lib");
dnoise_test = (1 : ba.impulsify, 10.0) : no.dnoise;
```

Example This expression `sx = hslider("seed multiplier",1,1,1000,1)` allows 1000 distinct seed variations. To generate a burst with a fixed length, use `ba.spulse(bLength, t)` (as trigger for the `t` parameter), where `bLength` is the burst duration in samples and `t` is a trigger.

(no.)randomseed

A random seed based on the foreign function `arc4random` (see man arc4random). Used in `rnoise`, `rmultirandom`, etc. to avoid having the same pseudo random sequence at each run.

WARNING: using the foreign function `arc4random`, so only available in C/C++ and LLVM backends.

Usage

```
randomseed : _
```

Where:

- output: platform-specific random seed value.

Test

```
no = library("noises.lib");
randomseed_test = no.randomseed;
```

`(no.)rnoise`

A randomized white noise generator (outputs random number between -1 and 1).

WARNING: using the foreign function `arc4random`, so only available in C/C++ and LLVM backends.

Usage

```
rnoise : _
```

Test

```
no = library("noises.lib");
rnoise_test = no.rnoise;
```

`(no.)rmultirandom`

Generates multiple decorrelated random numbers in parallel.

WARNING: using the foreign function `arc4random`, so only available in C/C++ and LLVM backends.

Usage

```
rmultirandom(N) : _
```

Where:

- `N`: the number of decorrelated random numbers in parallel, a constant numerical expression

Test

```
no = library("noises.lib");
rmultirandom_test = no.rmultirandom(4);
```

`(no.)rmultinoise`

Generates multiple decorrelated noises in parallel.

WARNING: using the foreign function `arc4random`, so only available in C/C++ and LLVM backends.

Usage

```
rmultinoise(N) : _
```

Where:

- `N`: the number of decorrelated random numbers in parallel, a constant numerical expression

Test

```
no = library("noises.lib");
rmultinoise_test = no.rmultinoise(3);
```

(no.)rnoises

A convenient wrapper around rmultinoise.

WARNING: using the foreign function `arc4random`, so only available in C/C++ and LLVM backends.

Usage

```
rnoises(N,i) : _
```

Where:

- `N`: the number of decorrelated random numbers in parallel
- `i`: the selected random number (i in [0..N[)

Test

```
no = library("noises.lib");
rnoises_test = no.rnoises(4, 2);
```

(no.)pink_noise

Pink noise (1/f noise) generator (third-order approximation covering the audio band well). `pink_noise` is a standard Faust function.

Usage

```
pink_noise : _
```

Where:

- output: pink (1/f) noise signal.

Test

```
no = library("noises.lib");
pink_noise_test = no.pink_noise;
```

Alternatives Higher-order approximations covering any frequency band can be obtained using

```
no.noise : fi.spectral_tilt(order,lowerBandLimit,Bandwidth,p)
```

where `p=-0.5` means filter rolloff `f^(-1/2)` which gives 1/f rolloff in the power spectral density, and can be changed to other real values.

Example pink_noise_compare.dsp - compare three pinking filters

```
process = pink_noises with {
    f0 = 35; // Lower bandlimit in Hz
    bw3 = 0.7 * ma.SR/2.0 - f0; // Bandwidth in Hz, 3rd order case
    bw9 = 0.8 * ma.SR/2.0 - f0; // Bandwidth in Hz, 9th order case
    pink_tilt_3 = fi.spectral_tilt(3,f0,bw3,-0.5);
    pink_tilt_9 = fi.spectral_tilt(9,f0,bw9,-0.5);
    pink_noises = 1-1' <:
      no.pink_filter, // original designed by invfreqz in Octave
      pink_tilt_3,    // newer method using the same filter order
      pink_tilt_9;    // newer method using a higher filter order
};
```

Output of Example

```
faust2octave pink_noise_compare.dsp
Octave:1> semilogx(20*log10(abs(fft(faustout,8192))(1:4096,:)));
...
```

References

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

---

`(no.)pink_noise_vm`

Multi pink noise generator.

Usage

```
pink_noise_vm(N) : _
```

Where:

- `N`: number of latched white-noise processes to sum, not to exceed sizeof(int) in C++ (typically 32).

Test

```
no = library("noises.lib");
pink_noise_vm_test = no.pink_noise_vm(4);
```

References

- http://www.dsprelated.com/showarticle/908.php
- http://www.firstpr.com.au/dsp/pink-noise/#Voss-McCartney

(no.)lfnoise, (no.)lfnoise0 and (no.)lfnoiseN

Low-frequency noise generators (Butterworth-filtered downsampled white noise).

Usage

```
lfnoise0(rate) : _   // new random number every int(ma.SR/rate) samples or so
lfnoiseN(N,rate) : _ // same as "lfnoise0(rate) : fi.lowpass(N,rate)" [see filters.lib]
lfnoise(rate) : _    // same as "lfnoise0(rate) : seq(i,5,fi.lowpass(N,rate))" (no overshoot
```

Example (view waveforms in faust2octave):

```
rate = ma.SR/100.0; // new random value every 100 samples (ma.SR from maths.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
```

Test

```
no = library("noises.lib");
lfnoise0_test = no.lfnoise0(10.0);
lfnoiseN_test = no.lfnoiseN(3, 10.0);
lfnoise_test = no.lfnoise(10.0);
```

(no.)sparse_noise

Sparse noise generator.

Usage

```
sparse_noise(f0) : _
```

Where:

- `f0`: average frequency of noise impulses per second

Random impulses in the amplitude range -1 to 1 are generated at an average rate of f0 impulses per second.

Test

```
no = library("noises.lib");
sparse_noise_test = no.sparse_noise(5.0);
```

References

- See velvet_noise

`(no.)velvet_noise_vm`

Velvet noise generator.

Usage

```
velvet_noise(amp, f0) : _
```

Where:

- `amp`: amplitude of noise impulses (positive and negative)
- `f0`: average frequency of noise impulses per second

Test

```
no = library("noises.lib");
velvet_noise_test = no.velvet_noise(0.5, 5.0);
```

References

- Matti Karjalainen and Hanna Jarvelainen, "Reverberation Modeling Using Velvet Noise", in Proc. 30th Int. Conf. Intelligent Audio Environments (AES07), March 2007.

`(no.)gnoise`

Approximate zero-mean, unit-variance Gaussian white noise generator.

Usage

```
gnoise(N) : _
```

Where:

- `N`: number of uniform random numbers added to approximate Gaussian white noise

Test

```
no = library("noises.lib");
gnoise_test = no.gnoise(8);
```

References

- See Central Limit Theorem

(no.)colored_noise

Generates a colored noise signal with an arbitrary spectral roll-off factor (alpha) over the entire audible frequency range (20-20000 Hz). The output is normalized so that an equal RMS level is maintained for different values of alpha.

Usage

```
colored_noise(N,alpha) : _
```

Where:

- `N`: desired integer filter order (constant numerical expression)
- `alpha`: slope of roll-off, between -1 and 1. -1 corresponds to brown/red noise, -1/2 pink noise, 0 white noise, 1/2 blue noise, and 1 violet/azure noise.

Test

```
no = library("noises.lib");
colored_noise_test = no.colored_noise(4, 0.0);
```

Examples See `dm.colored_noise_demo`.

oscillators.lib

Oscillators library. Its official prefix is `os`.

This library provides a wide range of oscillator designs for sound synthesis. It includes classic waveforms (sine, sawtooth, square, triangle), band-limited and anti-aliased oscillators, phase and frequency modulation units, as well as noise-based and physical-model driven oscillators for advanced synthesis techniques in Faust.

The oscillators library is organized into 9 sections:

- Wave-Table-Based Oscillators
- Low Frequency Oscillators
- Low Frequency Sawtooths
- Alias-Suppressed Sawtooth
- Alias-Suppressed Pulse, Square, and Impulse Trains
- Filter-Based Oscillators
- Waveguide-Resonator-Based Oscillators
- Casio CZ Oscillators
- PolyBLEP-Based Oscillators

References

- https://github.com/grame-cncm/faustlibraries/blob/master/oscillators.lib

Oscillators based on mathematical functions

Note that there is a numerical problem with several phasor functions built using the internal `phasor_imp`. The reason is that the incremental step is smaller than `ma.EPSILON`, which happens with very small frequencies, so it will have no effect when summed to 1, but it will be enough to make the fractional function wrap around when summed to 0. An example of this problem can be observed when running the following code:

```
process = os.phasor(1.0, -.001);
```

The output of this program is the sequence 1, 0, 1, 0, 1… This happens because the negative incremental step is greater than `-ma.EPSILON`, which will have no effect when summed to 1, but it will be significant enough to make the fractional function wrap around when summed to 0.

The incremental step can be clipped to guarantee that the phasor will always run correctly for its full cycle, otherwise, for increments smaller than `ma.EPSILON`, phasor would initially run but it'd eventually get stuck once the output gets big enough.

All functions using `phasor_imp` are affected by this problem, but a safer version is implemented, and can be used alternatively by setting `SAFE=1` in the environment using explicit sustitution syntax.

For example: `process = os[SAFE=1;].phasor(1.0, -.001);` will use the safer implementation of `phasor_imp`.

Wave-Table-Based Oscillators

Oscillators using tables. The table size is set by the pl.tablesize constant.

(os.)sinwaveform

Sine waveform ready to use with a `rdtable`.

Usage

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

Where:

- `tablesize`: the table size

Test

```
os = library("oscillators.lib");
sinwaveform_test = os.sinwaveform(1024);
```

(os.)coswaveform

Cosine waveform ready to use with a `rdtable`.

Usage

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

Where:

- `tablesize`: the table size

Test

```
os = library("oscillators.lib");
coswaveform_test = os.coswaveform(1024);
```

(os.)phasor

A simple phasor to be used with a `rdtable`. `phasor` is a standard Faust function.

Usage

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

Where:

- `tablesize`: the table size
- `freq`: the frequency in Hz

Note that `tablesize` is just a multiplier for the output of a unit-amp phasor so `phasor(1.0, freq)` can be used to generate a phasor output in the range [0, 1[.

Test

```
os = library("oscillators.lib");
phasor_test = os.phasor(1024, 440);
```

(os.)hs_phasor

Hardsyncing phasor to be used with a `rdtable`.

Usage

```
hs_phasor(tablesize,freq,reset) :  _
```

Where:

- `tablesize`: the table size
- `freq`: the frequency in Hz
- `reset`: a reset signal, reset phase to 0 when equal to 1

Test

```
os = library("oscillators.lib");
hs_phasor_test = os.hs_phasor(1024, 330, button("reset"));
```

(os.)hsp_phasor

Hardsyncing phasor with selectable phase to be used with a `rdtable`.

Usage

```
hsp_phasor(tablesize,freq,reset,phase)
```

Where:

- `tablesize`: the table size
- `freq`: the frequency in Hz

- `reset`: reset the oscillator to phase when equal to 1
- `phase`: phase between 0 and 1

Test

```
os = library("oscillators.lib");
hsp_phasor_test = os.hsp_phasor(1024, 330, button("reset"), 0.25);
```

`(os.)oscsin`

Sine wave oscillator. `oscsin` is a standard Faust function.

Usage

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

Where:

- `freq`: the frequency in Hz

Test

```
os = library("oscillators.lib");
oscsin_test = os.oscsin(440);
```

`(os.)hs_oscsin`

Sin lookup table with hardsyncing phase.

Usage

```
hs_oscsin(freq,reset) : _
```

Where:

- `freq`: the frequency in Hz
- `reset`: reset the oscillator to 0 when equal to 1

Test

```
os = library("oscillators.lib");
hs_oscsin_test = os.hs_oscsin(440, button("reset"));
```

`(os.)osccos`

Cosine wave oscillator.

Usage

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

Where:

- `freq`: the frequency in Hz

Test

```
os = library("oscillators.lib");
osccos_test = os.osccos(440);
```

(os.)hs_osccos

Cos lookup table with hardsyncing phase.

Usage

```
hs_osccos(freq,reset) : _
```

Where:

- `freq`: the frequency in Hz
- `reset`: reset the oscillator to 0 when equal to 1

Test

```
os = library("oscillators.lib");
hs_osccos_test = os.hs_osccos(440, button("reset"));
```

(os.)oscp

A sine wave generator with controllable phase.

Usage

```
oscp(freq,phase) : _
```

Where:

- `freq`: the frequency in Hz
- `phase`: the phase in radian

Test

```
os = library("oscillators.lib");
ma = library("maths.lib");
oscp_test = os.oscp(440, ma.PI/3);
```

`(os.)osci`

Interpolated phase sine wave oscillator.

Usage

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

Where:

- `freq`: the frequency in Hz

Test

```
os = library("oscillators.lib");
osci_test = os.osci(440);
```

`(os.)osc`

Default sine wave oscillator (same as oscsin). `osc` is a standard Faust function.

Usage

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

Where:

- `freq`: the frequency in Hz

Test

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

`(os.)m_oscsin`

Sine wave oscillator based on the `sin` mathematical function.

Usage

```
m_oscsin(freq) : _
```

Where:

- `freq`: the frequency in Hz

Test

```
os = library("oscillators.lib");
m_oscsin_test = os.m_oscsin(440);
```

(os.)m_osccos

Sine wave oscillator based on the `cos` mathematical function.

Usage

```
m_osccos(freq) : _
```

Where:

- `freq`: the frequency in Hz

Test

```
os = library("oscillators.lib");
m_osccos_test = os.m_osccos(440);
```

Low Frequency Oscillators

Low Frequency Oscillators (LFOs) have prefix `lf_` (no aliasing suppression, since it is inaudible at LF). Use `sawN` and its derivatives for audio oscillators with suppressed aliasing.

(os.)lf_imptrain

Unit-amplitude low-frequency impulse train. `lf_imptrain` is a standard Faust function. #### Usage

```
lf_imptrain(freq) : _
```

Where:

- `freq`: frequency in Hz

Test

```
os = library("oscillators.lib");
lf_imptrain_test = os.lf_imptrain(3);
```

(os.)lf_pulsetrainpos

Unit-amplitude nonnegative LF pulse train, duty cycle between 0 and 1.

Usage

```
lf_pulsetrainpos(freq, duty) : _
```

Where:

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

Test

```
os = library("oscillators.lib");
lf_pulsetrainpos_test = os.lf_pulsetrainpos(3, 0.35);
```

(os.)lf_pulsetrain

Unit-amplitude zero-mean LF pulse train, duty cycle between 0 and 1.

Usage

```
lf_pulsetrain(freq,duty) : _
```

Where:

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

Test

```
os = library("oscillators.lib");
lf_pulsetrain_test = os.lf_pulsetrain(3, 0.35);
```

(os.)lf_squarewavepos

Positive LF square wave in [0,1]

Usage

```
lf_squarewavepos(freq) : _
```

Where:

- `freq`: frequency in Hz

Test

```
os = library("oscillators.lib");
lf_squarewavepos_test = os.lf_squarewavepos(3);
```

(os.)lf_squarewave

Zero-mean unit-amplitude LF square wave. `lf_squarewave` is a standard Faust function.

Usage

```
lf_squarewave(freq) : _
```

Where:

- `freq`: frequency in Hz

Test

```
os = library("oscillators.lib");
lf_squarewave_test = os.lf_squarewave(3);
```

(os.)lf_trianglepos

Positive unit-amplitude LF positive triangle wave.

Usage

```
lf_trianglepos(freq) : _
```

Where:

- `freq`: frequency in Hz

Test

```
os = library("oscillators.lib");
lf_trianglepos_test = os.lf_trianglepos(3);
```

(os.)lf_triangle

Zero-mean unit-amplitude LF triangle wave. `lf_triangle` is a standard Faust function.

Usage

```
lf_triangle(freq) : _
```

Where:

- `freq`: frequency in Hz

Test

```
os = library("oscillators.lib");
lf_triangle_test = os.lf_triangle(3);
```

Low Frequency Sawtooths

Sawtooth waveform oscillators for virtual analog synthesis et al. The 'simple' versions (`lf_rawsaw`, `lf_sawpos` and `saw1`), are mere samplings of the ideal continuous-time ("analog") waveforms. While simple, the aliasing due to sampling is quite audible. The differentiated polynomial waveform family (`saw2`, `sawN`, and derived functions) do some extra processing to suppress aliasing (not audible for very low fundamental frequencies). According to Lehtonen et al. (JASA 2012), the aliasing of `saw2` should be inaudible at fundamental frequencies below 2 kHz or so, for a 44.1 kHz sampling rate and 60 dB SPL presentation level; fundamentals 415 and below required no aliasing suppression (i.e., `saw1` is ok).

(os.)lf_rawsaw

Simple sawtooth waveform oscillator between 0 and period in samples.

Usage

```
lf_rawsaw(periodsamps) : _
```

Where:

- `periodsamps`: number of periods per samples

Test

```
os = library("oscillators.lib");
lf_rawsaw_test = os.lf_rawsaw(128);
```

(os.)lf_sawpos

Simple sawtooth waveform oscillator between 0 and 1.

Usage

```
lf_sawpos(freq) : _
```

Where:

- `freq`: frequency in Hz

Test

```
os = library("oscillators.lib");
lf_sawpos_test = os.lf_sawpos(3);
```

(os.)lf_sawpos_phase

Simple sawtooth waveform oscillator between 0 and 1 with phase control.

Usage

```
lf_sawpos_phase(freq, phase) : _
```

Where:

- `freq`: frequency in Hz
- `phase`: phase between 0 and 1

Test

```
os = library("oscillators.lib");
lf_sawpos_phase_test = os.lf_sawpos_phase(3, 0.25);
```

(os.)lf_sawpos_reset

Simple sawtooth waveform oscillator between 0 and 1 with reset.

Usage

```
lf_sawpos_reset(freq,reset) : _
```

Where:

- `freq`: frequency in Hz
- `reset`: reset the oscillator to 0 when equal to 1

Test

```
os = library("oscillators.lib");
lf_sawpos_reset_test = os.lf_sawpos_reset(3, button("reset"));
```

(os.)lf_sawpos_phase_reset

Simple sawtooth waveform oscillator between 0 and 1 with phase control and reset.

Usage

```
lf_sawpos_phase_reset(freq,phase,reset) : _
```

Where:

- `freq`: frequency in Hz
- `phase`: phase between 0 and 1
- `reset`: reset the oscillator to phase when equal to 1

Test

```
os = library("oscillators.lib");
lf_sawpos_phase_reset_test = os.lf_sawpos_phase_reset(3, 0.75, button("reset"));
```

(os.)lf_saw

Simple sawtooth waveform oscillator between -1 and 1. `lf_saw` is a standard Faust function.

Usage

```
lf_saw(freq) : _
```

Where:

- `freq`: frequency in Hz

Test

```
os = library("oscillators.lib");
lf_saw_test = os.lf_saw(3);
```

Alias-Suppressed Sawtooth

(os.)sawN

Alias-Suppressed Sawtooth Audio-Frequency Oscillator using Nth-order polynomial transitions to reduce aliasing.

`sawN(N,freq)`, `sawNp(N,freq,phase)`, `saw2dpw(freq)`, `saw2(freq)`, `saw3(freq)`, `saw4(freq)`, `sawtooth(freq)`, `saw2f2(freq)`, `saw2f4(freq)`

Usage

```
sawN(N,freq) : _        // Nth-order aliasing-suppressed sawtooth using DPW method (see belo
sawNp(N,freq,phase) : _ // sawN with phase offset feature
saw2dpw(freq) : _       // saw2 using DPW
saw2ptr(freq) : _       // saw2 using the faster, stateless PTR method
saw2(freq) : _          // DPW method, but subject to change if a better method emerges
saw3(freq) : _          // sawN(3)
saw4(freq) : _          // sawN(4)
sawtooth(freq) : _      // saw2
saw2f2(freq) : _        // saw2dpw with 2nd-order droop-correction filtering
saw2f4(freq) : _        // saw2dpw with 4th-order droop-correction filtering
```

Where:

- `N`: polynomial order, a constant numerical expression between 1 and 4
- `freq`: frequency in Hz
- `phase`: phase between 0 and 1

Test

```
os = library("oscillators.lib");
sawN_test = os.sawN(3, 440);
```

Method Differentiated Polynomial Wave (DPW).

Reference “Alias-Suppressed Oscillators based on Differentiated Polynomial Waveforms”, Vesa Valimaki, Juhan Nam, Julius Smith, and Jonathan Abel, IEEE Tr. Audio, Speech, and Language Processing (IEEE-ASLP), Vol. 18, no. 5, pp 786-798, May 2010. 10.1109/TASL.2009.2026507.

Notes The polynomial order `N` is limited to 4 because noise has been observed at very low `freq` values. (LFO sawtooths should of course be generated using `lf_sawpos` instead.)

(os.)sawNp

Same as `(os.)sawN` but with a controllable waveform phase.

Usage

```
sawNp(N,freq,phase) : _
```

where

- `N`: waveform interpolation polynomial order 1 to 4 (constant integer expression)
- `freq`: frequency in Hz
- `phase`: waveform phase as a fraction of one period (rounded to nearest sample)

Test

```
os = library("oscillators.lib");
sawNp_test = os.sawNp(3, 330, 0.5);
```

Implementation Notes The phase offset is implemented by delaying `sawN(N,freq)` by `round(phase*ma.SR/freq)` samples, for up to 8191 samples. The minimum sawtooth frequency that can be delayed a whole period is therefore `ma.SR/8191`, which is well below audibility for normal audio sampling rates.

`(os.)saw2`, `(os.)saw3`, `(os.)saw4`

Alias-Suppressed Sawtooth Audio-Frequency Oscillators of order 2, 3, 4.

Usage

```
saw2(freq) : _
saw3(freq) : _
saw4(freq) : _
```

where

- `freq`: frequency in Hz

Test

```
os = library("oscillators.lib");
saw2_test = os.saw2(220);
saw3_test = os.saw3(220);
saw4_test = os.saw4(220);
```

Implementation Notes Presently, only `saw2` uses the PTR method, while `saw3` and `saw4` use DPW. This is because PTR has been implemented and tested for the 2nd-order case only.

References

- See `sawN` above.

`(os.)saw2ptr`

Alias-Suppressed Sawtooth Audio-Frequency Oscillator using Polynomial Transition Regions (PTR) for order 2.

Usage

```
saw2ptr(freq) : _
```

where

- `freq`: frequency in Hz

Test

```
os = library("oscillators.lib");
saw2ptr_test = os.saw2ptr(220);
```

Implementation Polynomial Transition Regions (PTR) method for aliasing suppression.

Notes Method PTR may be preferred because it requires less computation and is stateless which means that the frequency `freq` can be modulated arbitrarily fast over time without filtering artifacts. For this reason, `saw2` is presently defined as `saw2ptr`.

References

- Kleimola, J.; Valimaki, V., "Reducing Aliasing from Synthetic Audio Signals Using Polynomial Transition Regions," 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/

`(os.)saw2dpw`

Alias-Suppressed Sawtooth Audio-Frequency Oscillator using the Differentiated Polynomial Waveform (DWP) method.

Usage

```
saw2dpw(freq) : _
```

where

- `freq`: frequency in Hz

This is the original Faust `saw2` function using the DPW method. Since `saw2` is now defined as `saw2ptr`, the DPW version is now available as `saw2dwp`.

Test

```
os = library("oscillators.lib");
saw2dpw_test = os.saw2dpw(220);
```

`(os.)sawtooth`

Alias-suppressed aliasing-suppressed sawtooth oscillator, presently defined as `saw2`. `sawtooth` is a standard Faust function.

Usage

```
sawtooth(freq) : _
```

with

- `freq`: frequency in Hz

Test

```
os = library("oscillators.lib");
sawtooth_test = os.sawtooth(220);
```

`(os.)saw2f2, (os.)saw2f4`

Alias-Suppressed Sawtooth Audio-Frequency Oscillator with Order 2 or 4 Droop Correction Filtering.

Usage

```
saw2f2(freq) : _
saw2f4(freq) : _
```

with

- `freq`: frequency in Hz

In return for aliasing suppression, there is some attenuation near half the sampling rate. This can be considered as beneficial, or it can be compensated with a high-frequency boost. The boost filter is second-order for `saw2f2` and fourth-order for `saw2f4`, and both are designed for the DWP case and therefore use `saw2dpw`. See Figure 4(b) in the DPW reference for a plot of the slight droop in the DPW case.

Test

```
os = library("oscillators.lib");
saw2f2_test = os.saw2f2(220);
saw2f4_test = os.saw2f4(220);
```

Alias-Suppressed Pulse, Square, and Impulse Trains

Alias-Suppressed Pulse, Square and Impulse Trains.

`pulsetrainN`, `pulsetrain`, `squareN`, `square`, `imptrainN`, `imptrain`, `triangleN`, `triangle`

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, a constant numerical expression
- `freq`: frequency in Hz

Test

```
os = library("oscillators.lib");
sawNp_test = os.sawNp(3, 330, 0.5);
saw2_test = os.saw2(220);
saw3_test = os.saw3(220);
saw4_test = os.saw4(220);
```

```
saw2ptr_test = os.saw2ptr(220);
saw2dpw_test = os.saw2dpw(220);
sawtooth_test = os.sawtooth(220);
saw2f2_test = os.saw2f2(220);
saw2f4_test = os.saw2f4(220);
pulsetrainN_test = os.pulsetrainN(3, 220, 0.25);
pulsetrain_test = os.pulsetrain(220, 0.25);
squareN_test = os.squareN(3, 220);
square_test = os.square(220);
imptrainN_test = os.imptrainN(3, 220);
imptrain_test = os.imptrain(220);
triangleN_test = os.triangleN(3, 220);
triangle_test = os.triangle(220);
```

(os.)impulse

One-time impulse generated when the Faust process is started. `impulse` is a standard Faust function.

Usage

```
impulse : _
```

Test

```
os = library("oscillators.lib");
impulse_test = os.impulse;
```

(os.)pulsetrainN

Alias-suppressed pulse train oscillator.

Usage

```
pulsetrainN(N,freq,duty) : _
```

Where:

- `N`: order, as a constant numerical expression
- `freq`: frequency in Hz
- `duty`: duty cycle between 0 and 1

Test

```
os = library("oscillators.lib");
pulsetrainN_test = os.pulsetrainN(3, 220, 0.25);
```

(os.)pulsetrain

Alias-suppressed pulse train oscillator. Based on `pulsetrainN(2)`. `pulsetrain` is a standard Faust function.

Usage

```
pulsetrain(freq,duty) : _
```

Where:

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

Test

```
os = library("oscillators.lib");
pulsetrain_test = os.pulsetrain(220, 0.25);
```

(os.)squareN

Alias-suppressed square wave oscillator.

Usage

```
squareN(N,freq) : _
```

Where:

- `N`: order, as a constant numerical expression
- `freq`: frequency in Hz

Test

```
os = library("oscillators.lib");
squareN_test = os.squareN(3, 220);
```

(os.)square

Alias-suppressed square wave oscillator. Based on `squareN(2)`. `square` is a standard Faust function.

Usage

`square(freq) : _`

Where:

- `freq`: frequency in Hz

Test

```
os = library("oscillators.lib");
square_test = os.square(220);
```

(os.)imptrainN

Alias-suppressed impulse train generator.

Usage

`imptrainN(N,freq) : _`

Where:

- `N`: order, as a constant numerical expression
- `freq`: frequency in Hz

Test

```
os = library("oscillators.lib");
imptrainN_test = os.imptrainN(4, 220);
```

(os.)imptrain

Alias-suppressed impulse train generator. Based on `imptrainN(2)`. `imptrain` is a standard Faust function.

Usage

`imptrain(freq) : _`

Where:

- `freq`: frequency in Hz

Test

```
os = library("oscillators.lib");
imptrain_test = os.imptrain(220);
```

(os.)triangleN

Alias-suppressed triangle wave oscillator.

Usage

```
triangleN(N,freq) : _
```

Where:

- `N`: order, as a constant numerical expression
- `freq`: frequency in Hz

Test

```
os = library("oscillators.lib");
triangleN_test = os.triangleN(3, 220);
```

(os.)triangle

Alias-suppressed triangle wave oscillator. Based on `triangleN(2)`. `triangle` is a standard Faust function.

Usage

```
triangle(freq) : _
```

Where:

- `freq`: frequency in Hz

Test

```
os = library("oscillators.lib");
triangle_test = os.triangle(220);
```

Filter-Based Oscillators

Filter-Based Oscillators.

Usage

```
osc[b|rq|rs|rc|s](freq), where freq = frequency in Hz.
```

Test

```
os = library("oscillators.lib");
oscb_test = os.oscb(440);
oscrq_test = os.oscrq(440);
oscrs_test = os.oscrs(440);
oscrc_test = os.oscrc(440);
oscs_test = os.oscs(440);
```

References

- http://lac.linuxaudio.org/2012/download/lac12-slides-jos.pdf
- https://ccrma.stanford.edu/~jos/pdf/lac12-paper-jos.pdf

`(os.)oscb`

Sinusoidal oscillator based on the biquad.

Usage

```
oscb(freq) : _
```

Where:

- `freq`: frequency in Hz

Test

```
os = library("oscillators.lib");
oscb_test = os.oscb(440);
oscrq_test = os.oscrq(440);
oscrs_test = os.oscrs(440);
oscrc_test = os.oscrc(440);
oscs_test = os.oscs(440);
```

`(os.)oscrq`

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

Usage

```
oscrq(freq) : _,_
```

Where:

- `freq`: frequency in Hz

Test

```
os = library("oscillators.lib");
oscrq_test = os.oscrq(440);
```

References

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

---

(os.)oscrs

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

Usage

```
oscrs(freq) : _
```

Where:

- `freq`: frequency in Hz

Test

```
os = library("oscillators.lib");
oscrs_test = os.oscrs(440);
```

References

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

---

(os.)oscrc

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

Usage

```
oscrc(freq) : _
```

Where:

- `freq`: frequency in Hz

Test

```
os = library("oscillators.lib");
oscrc_test = os.oscrc(440);
```

References

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

---

(os.)oscs

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

Usage

```
oscs(freq) : _
```

Where:

- `freq`: frequency in Hz

Test

```
os = library("oscillators.lib");
oscs_test = os.oscs(440);
```

(os.)quadosc

Quadrature (cosine and sine) oscillator based on QuadOsc by Martin Vicanek.

Usage

```
quadosc(freq) : _,_
```

where

- `freq`: frequency in Hz

Test

```
os = library("oscillators.lib");
quadosc_test = os.quadosc(440);
```

References

- https://vicanek.de/articles/QuadOsc.pdf

`(os.)sidebands`

Adds harmonics to quad oscillator.

Usage

```
cos(x),sin(x) : sidebands(vs) : _,_
```

Where:

- `vs` : list of amplitudes

Test

```
os = library("oscillators.lib");
sidebands_test = os.quadosc(110) : os.sidebands((1, 0.5, 0.25));
```

Example test program

```
cos(x),sin(x) : sidebands((10,20,30))
```

outputs:

```
10*cos(x) + 20*cos(2*x) + 30*cos(3*x),
10*sin(x) + 20*sin(2*x) + 30*sin(3*x);
```

The following:

```
process = os.quadosc(F) : sidebands((10,20,30))
```

is (modulo floating point issues) the same as:

```
c = os.quadosc : _,!;
s = os.quadosc : !,_;
process =
    10*c(F) + 20*c(2*F) + 30*c(F),
    10*s(F) + 20*s(2*F) + 30*s(F);
```

but much more efficient.

Implementation Notes This is based on the trivial trigonometric identities:

```
cos((n + 1) x) = 2 cos(x) cos(n x) - cos((n - 1) x)
sin((n + 1) x) = 2 cos(x) sin(n x) - sin((n - 1) x)
```

Note that the calculation of the cosine/sine parts do not depend on each other, so if you only need the sine part you can do:

```
process = os.quadosc(F) : sidebands(vs) : !,_;
```

and the compiler will discard the half of the calculations.

(os.)sidebands_list

Creates the list of complex harmonics from quad oscillator.

Similar to `sidebands` but doesn't sum the harmonics, so it is more generic but less convenient for immediate usage.

Usage

```
cos(x),sin(x) : sidebands_list(N) : si.bus(2*N)
```

Where:

- `N` : number of harmonics, compile time constant > 1

Test

```
os = library("oscillators.lib");
sidebands_list_test = os.quadosc(110) : os.sidebands_list(3);
```

Example test program

```
cos(x),sin(x) : sidebands_list(3)
```

outputs:

```
cos(x),sin(x), cos(2*x),sin(2*x), cos(3*x),sin(3*x);
```

The following:

```
process = os.quadosc(F) : sidebands_list(3)
```

is (modulo floating point issues) the same as:

```
process = os.quadosc(F), os.quadosc(2*F), os.quadosc(3*F);
```

but much more efficient.

(os.)dsf

An environment with sine/cosine oscsillators with exponentially decaying harmonics based on direct summation formula.

Usage

```
dsf.xxx(f0, df, a, [n]) : _
```

Where:

- `f0`: base frequency
- `df`: step frequency
- `a`: decaying factor != 1
- `n`: total number of harmonics (`osccN/oscsN` only)

Test

```
os = library("oscillators.lib");
dsf_oscc_test = os.dsf.oscc(220, 110, 0.6);
dsf_oscs_test = os.dsf.oscs(220, 110, 0.6);
dsf_osccN_test = os.dsf.osccN(220, 110, 0.6, 4);
dsf_oscsN_test = os.dsf.oscsN(220, 110, 0.6, 4);
dsf_osccNq_test = os.dsf.osccNq(220, 110, 0.6);
dsf_oscsNq_test = os.dsf.oscsNq(220, 110, 0.6);
```

Variants

- infinite number of harmonics, implies aliasing

```
oscc(f0,df,a) : _;
oscs(f0,df,a) : _;
```

- n harmonics, f0, f0 + df, f0 + 2*df, …, f0 + (n-1)*df

```
osccN(f0,df,a,n) : _;
oscsN(f0,df,a,n) : _;
```

- finite number of harmonics, from f0 to Nyquist

```
osccNq(f0,df,a) : _;
oscsNq(f0,df,a) : _;
```

Example test program

```
process = dsf.osccN(F0,DF,A,N),
          dsf.oscsN(F0,DF,A,N);
```

if `N` is an integer constant, the same (modulo fp issues) as:

```
c = os.quadosc : _,!;
s = os.quadosc : !,_;
process = sum(k,N, A^k * c(F0 + k*DF)),
          sum(k,N, A^k * s(F0 + k*DF));
```

but much more efficient.

References

- https://ccrma.stanford.edu/STANM/stanms/stanm5/stanm5.pdf

Waveguide-Resonator-Based Oscillators

Sinusoidal oscillator based on the waveguide resonator `wgr`.

(os.)oscwc

Sinusoidal oscillator based on the waveguide resonator `wgr`. Unit-amplitude cosine oscillator.

Usage

```
oscwc(freq) : _
```

Where:

- `freq`: frequency in Hz

Test

```
os = library("oscillators.lib");
oscwc_test = os.oscwc(440);
```

References

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

---

(os.)oscws

Sinusoidal oscillator based on the waveguide resonator `wgr`. Unit-amplitude sine oscillator.

Usage

```
oscws(freq) : _
```

Where:

- `freq`: frequency in Hz

Test

```
os = library("oscillators.lib");
oscws_test = os.oscws(440);
```

References

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

---

(os.)oscq

Sinusoidal oscillator based on the waveguide resonator `wgr`. Unit-amplitude cosine and sine (quadrature) oscillator.

Usage

```
oscq(freq) : _,_
```

Where:

- `freq`: frequency in Hz

Test

```
os = library("oscillators.lib");
oscq_test = os.oscq(440);
```

References

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

---

(os.)oscw

Sinusoidal oscillator based on the waveguide resonator `wgr`. Unit-amplitude cosine oscillator (default).

Usage

```
oscw(freq) : _
```

Where:

- `freq`: frequency in Hz

Test

```
os = library("oscillators.lib");
oscw_test = os.oscw(440);
```

References

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

Casio CZ Oscillators

Oscillators that mimic some of the Casio CZ oscillators.

There are two sets:

- a set with an index parameter
- a set with a res parameter

The "index oscillators" outputs a sine wave at index=0 and gets brighter with a higher index. There are two versions of the "index oscillators":

- with P appended to the name: is phase aligned with `fund:sin`
- without P appended to the name: has the phase of the original CZ oscillators

The "res oscillators" have a resonant frequency. "res" is the frequency of resonance as a factor of the fundamental pitch.

For the `fund` waveform, use a low-frequency oscillator without anti-aliasing such as `os.lf_saw`.

`(os.)CZsaw`

Oscillator that mimics the Casio CZ saw oscillator. `CZsaw` is a standard Faust function.

Usage

```
CZsaw(fund,index) : _
```

Where:

- `fund`: a saw-tooth waveform between 0 and 1 that the oscillator slaves to
- `index`: the brightness of the oscillator, 0 to 1. 0 = sine-wave, 1 = saw-wave

Test

```
os = library("oscillators.lib");
CZsaw_test = os.CZsaw(os.lf_sawpos(110), 0.5);
```

(os.)CZsawP

Oscillator that mimics the Casio CZ saw oscillator, with it's phase aligned to `fund:sin`. `CZsawP` is a standard Faust function.

Usage

```
CZsawP(fund,index) : _
```

Where:

- `fund`: a saw-tooth waveform between 0 and 1 that the oscillator slaves to
- `index`: the brightness of the oscillator, 0 to 1. 0 = sine-wave, 1 = saw-wave

Test

```
os = library("oscillators.lib");
CZsawP_test = os.CZsawP(os.lf_sawpos(110), 0.5);
```

(os.)CZsquare

Oscillator that mimics the Casio CZ square oscillator `CZsquare` is a standard Faust function.

Usage

```
CZsquare(fund,index) : _
```

Where:

- `fund`: a saw-tooth waveform between 0 and 1 that the oscillator slaves to
- `index`: the brightness of the oscillator, 0 to 1. 0 = sine-wave, 1 = square-wave

Test

```
os = library("oscillators.lib");
CZsquare_test = os.CZsquare(os.lf_sawpos(110), 0.5);
```

(os.)CZsquareP

Oscillator that mimics the Casio CZ square oscillator, with it's phase aligned to `fund:sin`. `CZsquareP` is a standard Faust function.

Usage

`CZsquareP(fund,index) : _`

Where:

- `fund`: a saw-tooth waveform between 0 and 1 that the oscillator slaves to
- `index`: the brightness of the oscillator, 0 to 1. 0 = sine-wave, 1 = square-wave

Test

```
os = library("oscillators.lib");
CZsquareP_test = os.CZsquareP(os.lf_sawpos(110), 0.5);
```

(os.)CZpulse

Oscillator that mimics the Casio CZ pulse oscillator. `CZpulse` is a standard Faust function.

Usage

`CZpulse(fund,index) : _`

Where:

- `fund`: a saw-tooth waveform between 0 and 1 that the oscillator slaves to
- `index`: the brightness of the oscillator, 0 gives a sine-wave, 1 is closer to a pulse

Test

```
os = library("oscillators.lib");
CZpulse_test = os.CZpulse(os.lf_sawpos(110), 0.5);
```

(os.)CZpulseP

Oscillator that mimics the Casio CZ pulse oscillator, with it's phase aligned to `fund:sin`. `CZpulseP` is a standard Faust function.

Usage

`CZpulseP(fund,index) : _`

Where:

- `fund`: a saw-tooth waveform between 0 and 1 that the oscillator slaves to

- `index`: the brightness of the oscillator, 0 gives a sine-wave, 1 is closer to a pulse

Test

```
os = library("oscillators.lib");
CZpulseP_test = os.CZpulseP(os.lf_sawpos(110), 0.5);
```

(os.)CZsinePulse

Oscillator that mimics the Casio CZ sine/pulse oscillator. `CZsinePulse` is a standard Faust function.

Usage

```
CZsinePulse(fund,index) : _
```

Where:

- `fund`: a saw-tooth waveform between 0 and 1 that the oscillator slaves to
- `index`: the brightness of the oscillator, 0 gives a sine-wave, 1 is a sine minus a pulse

Test

```
os = library("oscillators.lib");
CZsinePulse_test = os.CZsinePulse(os.lf_sawpos(110), 0.5);
```

(os.)CZsinePulseP

Oscillator that mimics the Casio CZ sine/pulse oscillator, with it's phase aligned to `fund:sin`. `CZsinePulseP` is a standard Faust function.

Usage

```
CZsinePulseP(fund,index) : _
```

Where:

- `fund`: a saw-tooth waveform between 0 and 1 that the oscillator slaves to
- `index`: the brightness of the oscillator, 0 gives a sine-wave, 1 is a sine minus a pulse

Test

```
os = library("oscillators.lib");
CZsinePulseP_test = os.CZsinePulseP(os.lf_sawpos(110), 0.5);
```

(os.)CZhalfSine

Oscillator that mimics the Casio CZ half sine oscillator. `CZhalfSine` is a standard Faust function.

Usage

```
CZhalfSine(fund,index) : _
```

Where:

- `fund`: a saw-tooth waveform between 0 and 1 that the oscillator slaves to
- `index`: the brightness of the oscillator, 0 gives a sine-wave, 1 is somewhere between a saw and a square

Test

```
os = library("oscillators.lib");
CZhalfSine_test = os.CZhalfSine(os.lf_sawpos(110), 0.5);
```

(os.)CZhalfSineP

Oscillator that mimics the Casio CZ half sine oscillator, with it's phase aligned to `fund:sin`. `CZhalfSineP` is a standard Faust function.

Usage

```
CZhalfSineP(fund,index) : _
```

Where:

- `fund`: a saw-tooth waveform between 0 and 1 that the oscillator slaves to
- `index`: the brightness of the oscillator, 0 gives a sine-wave, 1 is somewhere between a saw and a square

Test

```
os = library("oscillators.lib");
CZhalfSineP_test = os.CZhalfSineP(os.lf_sawpos(110), 0.5);
```

(os.)CZresSaw

Oscillator that mimics the Casio CZ resonant sawtooth oscillator. `CZresSaw` is a standard Faust function.

Usage

```
CZresSaw(fund,res) : _
```

Where:

- `fund`: a saw-tooth waveform between 0 and 1 that the oscillator slaves to
- `res`: the frequency of resonance as a factor of the fundamental pitch.

Test

```
os = library("oscillators.lib");
CZresSaw_test = os.CZresSaw(os.lf_sawpos(110), 2.5);
```

(os.)CZresTriangle

Oscillator that mimics the Casio CZ resonant triangle oscillator. `CZresTriangle` is a standard Faust function.

Usage

```
CZresTriangle(fund,res) : _
```

Where:

- `fund`: a saw-tooth waveform between 0 and 1 that the oscillator slaves to
- `res`: the frequency of resonance as a factor of the fundamental pitch.

Test

```
os = library("oscillators.lib");
CZresTriangle_test = os.CZresTriangle(os.lf_sawpos(110), 2.5);
```

(os.)CZresTrap

Oscillator that mimics the Casio CZ resonant trapeze oscillator `CZresTrap` is a standard Faust function.

Usage

```
CZresTrap(fund,res) : _
```

Where:

- `fund`: a saw-tooth waveform between 0 and 1 that the oscillator slaves to
- `res`: the frequency of resonance as a factor of the fundamental pitch.

Test

```
os = library("oscillators.lib");
CZresTrap_test = os.CZresTrap(os.lf_sawpos(110), 2.5);
```

PolyBLEP-Based Oscillators

(os.)polyblep

PolyBLEP residual function, used for smoothing steps in the audio signal.

Usage

```
polyblep(Q,phase) : _
```

Where:

- `Q`: smoothing factor between 0 and 0.5. Determines how far from the ends of the phase interval the quadratic function is used.
- `phase`: normalised phase (between 0 and 1)

Test

```
os = library("oscillators.lib");
polyblep_test = os.polyblep(0.2, os.lf_sawpos(220));
```

(os.)polyblep_saw

Sawtooth oscillator with suppressed aliasing (using `polyblep`).

Usage

```
polyblep_saw(freq) : _
```

Where:

- `freq`: frequency in Hz

Test

```
os = library("oscillators.lib");
polyblep_saw_test = os.polyblep_saw(220);
```

(os.)polyblep_square

Square wave oscillator with suppressed aliasing (using `polyblep`).

Usage

```
polyblep_square(freq) : _
```

Where:

- `freq`: frequency in Hz

Test

```
os = library("oscillators.lib");
polyblep_square_test = os.polyblep_square(220);
```

(os.)polyblep_triangle

Triangle wave oscillator with suppressed aliasing (using `polyblep`).

Usage

```
polyblep_triangle(freq) : _
```

Where:

- `freq`: frequency in Hz

Test

```
os = library("oscillators.lib");
polyblep_triangle_test = os.polyblep_triangle(220);
```

phaflangers.lib

Phasers and Flangers library. Its official prefix is `pf`.

This library provides a set of phaser and flanger effects based on delay-line modulation.

The Phaflangers library is organized into 1 section:

- Functions Reference

References

- https://github.com/grame-cncm/faustlibraries/blob/master/phaflangers.lib

Functions Reference

`(pf.)flanger_mono`

Mono flanging effect.

Usage:

```
_ : flanger_mono(dmax,curdel,depth,fb,invert) : _
```

Where:

- `dmax`: maximum delay-line length (power of 2) - 10 ms typical
- `curdel`: current dynamic delay (not to exceed dmax)
- `depth`: effect strength between 0 and 1 (1 typical)
- `fb`: feedback gain between 0 and 1 (0 typical)
- `invert`: 0 for normal, 1 to invert sign of flanging sum

Test

```
pf = library("phaflangers.lib");
os = library("oscillators.lib");
flanger_mono_test = os.osc(440) : pf.flanger_mono(4096, 1024, 0.7, 0.25, 0);
```

References

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

`(pf.)flanger_stereo`

Stereo flanging effect. `flanger_stereo` is a standard Faust function.

Usage:

```
_,_ : flanger_stereo(dmax,curdel1,curdel2,depth,fb,invert) : _,_
```

Where:

- `dmax`: maximum delay-line length (power of 2) - 10 ms typical
- `curdel1`: current dynamic delay for the left channel (not to exceed dmax)

- `curdel2`: current dynamic delay for the right channel (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

Test

```
pf = library("phaflangers.lib");
os = library("oscillators.lib");
flanger_stereo_test = os.osc(440), os.osc(660) : pf.flanger_stereo(4096, 1024, 1536, 0.7, 0.
```

References

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

`(pf.)phaser2_mono`

Mono phasing effect.

Phaser

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

Where:

- `Notches`: number of spectral notches (MACRO ARGUMENT - not a signal)
- `phase`: phase of the oscillator (0-1)
- `width`: approximate width of spectral notches in Hz
- `frqmin`: approximate minimum frequency of first spectral notch in Hz
- `fratio`: ratio of adjacent notch frequencies
- `frqmax`: approximate maximum frequency of first spectral notch in Hz
- `speed`: LFO frequency in Hz (rate of periodic notch sweep cycles)
- `depth`: effect strength between 0 and 1 (1 typical) (aka "intensity") when depth=2, "vibrato mode" is obtained (pure allpass chain)
- `fb`: feedback gain between -1 and 1 (0 typical)
- `invert`: 0 for normal, 1 to invert sign of flanging sum

Test

```
pf = library("phaflangers.lib");
os = library("oscillators.lib");
phaser2_mono_test = os.osc(330) : pf.phaser2_mono(4, 0.0, 50, 200, 1.5, 4000, 0.5, 0.8, 0.2,
```

References

- https://ccrma.stanford.edu/~jos/pasp/Phasing.html
- http://www.geofex.com/Article_Folders/phasers/phase.html
- 'An Allpass Approach to Digital Phasing and Flanging', Julius O. Smith III, Proc. Int. Computer Music Conf. (ICMC-84), pp. 103-109, Paris, 1984.
- CCRMA Tech. Report STAN-M-21: https://ccrma.stanford.edu/STANM/stanms/stanm21/

(pf.)phaser2_stereo

Stereo phasing effect. `phaser2_stereo` is a standard Faust function.

Phaser

```
_,_ : phaser2_stereo(Notches,width,frqmin,fratio,frqmax,speed,depth,fb,invert) : _,_
```

Where:

- `Notches`: number of spectral notches (MACRO ARGUMENT - not a signal)
- `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

Test

```
pf = library("phaflangers.lib");
os = library("oscillators.lib");
phaser2_stereo_test = os.osc(220), os.osc(330) : pf.phaser2_stereo(4, 50, 200, 1.5, 4000, 0.
```

References

- https://ccrma.stanford.edu/~jos/pasp/Phasing.html
- http://www.geofex.com/Article_Folders/phasers/phase.html
- 'An Allpass Approach to Digital Phasing and Flanging', Julius O. Smith III, Proc. Int. Computer Music Conf. (ICMC-84), pp. 103-109, Paris, 1984.
- CCRMA Tech. Report STAN-M-21: https://ccrma.stanford.edu/STANM/stanms/stanm21/

physmodels.lib

Faust physical modeling library. Its official prefix is `pm`.

This library provides an environment to facilitate physical modeling of musical instruments. It includes waveguide, mass-spring, and digital wave models for strings, membranes, bars, and resonant systems used in physical modeling synthesis and acoustic simulation research. It contains dozens of functions implementing low and high level elements going from a simple waveguide to fully operational models with built-in UI, etc.

It is organized as follows:

- Global Variables: useful pre-defined variables for physical modeling (e.g., speed of sound, etc.).
- Conversion Tools: conversion functions specific to physical modeling (e.g., length to frequency, etc.).
- Bidirectional Utilities: functions to create bidirectional block diagrams for physical modeling.
- Basic Elements: waveguides, specific types of filters, etc.
- String Instruments: various types of strings (e.g., steel, nylon, etc.), bridges, guitars, etc.
- Bowed String Instruments: parts and models specific to bowed string instruments (e.g., bows, bridges, violins, etc.).
- Wind Instrument: parts and models specific to wind instruments (e.g., reeds, mouthpieces, flutes, clarinets, etc.).
- Exciters: pluck generators, "blowers", etc.
- Modal Percussions: percussion instruments based on modal models.
- Vocal Synthesis: functions for various vocal synthesis techniques (e.g., fof, source/filter, etc.) and vocal synthesizers.
- Misc Functions: any other functions that don't fit in the previous category (e.g., nonlinear filters, etc.).

This library is part of the Faust Physical Modeling ToolKit. More information on how to use this library can be found on this page or this video. Tutorials on how to make physical models of musical instruments using Faust can be found here as well.

References

- https://github.com/grame-cncm/faustlibraries/blob/master/physmodels.lib

Global Variables

Useful pre-defined variables for physical modeling.

(pm.)speedOfSound

Speed of sound in meters per second (340m/s).

(pm.)maxLength

The default maximum length (3) in meters of strings and tubes used in this library. This variable should be overriden to allow longer strings or tubes.

Conversion Tools

Useful conversion tools for physical modeling.

(pm.)f2l

Frequency to length in meters.

Usage

```
f2l(freq) : distanceInMeters
```

Where:

- `freq`: the frequency

Test

```
pm = library("physmodels.lib");
f2l_test = pm.f2l(440);
```

(pm.)l2f

Length in meters to frequency.

Usage

```
l2f(length) : freq
```

Where:

- `length`: length/distance in meters

Test

```
pm = library("physmodels.lib");
l2f_test = pm.l2f(0.75);
```

(pm.)l2s

Length in meters to number of samples.

Usage

```
l2s(l) : numberOfSamples
```

Where:

- `l`: length in meters

Test

```
pm = library("physmodels.lib");
l2s_test = pm.l2s(1.2);
```

Bidirectional Utilities

Set of fundamental functions to create bi-directional block diagrams in Faust. These elements are used as the basis of this library to connect high level elements (e.g., mouthpieces, strings, bridge, instrument body, etc.). Each block has 3 inputs and 3 outputs. The first input/output carry left going waves, the second input/output carry right going waves, and the third input/output is used to carry any potential output signal to the end of the algorithm.

(pm.)basicBlock

Empty bidirectional block to be used with `chain`: 3 signals ins and 3 signals out.

Usage

```
chain(basicBlock : basicBlock : etc.)
```

Test

```
pm = library("physmodels.lib");
basicBlock_test = 0,0,0 : pm.basicBlock;
```

(pm.)chain

Creates a chain of bidirectional blocks. Blocks must have 3 inputs and outputs. The first input/output carry left going waves, the second input/output carry right going waves, and the third input/output is used to carry any potential output signal to the end of the algorithm. The implied one sample delay created by the ~ operator is generalized to the left and right going waves. Thus, `n` blocks in `chain()` will add an `n` samples delay to both left and right going waves.

Usage

```
leftGoingWaves,rightGoingWaves,mixedOutput : chain( A : B ) : leftGoingWaves,rightGoingWaves
with {
   A = _,_,_;
   B = _,_,_;
};
```

Test

```
pm = library("physmodels.lib");
chain_test = 0,0,0 : pm.chain(pm.in(0.1) : pm.basicBlock);
```

(pm.)inLeftWave

Adds a signal to left going waves anywhere in a `chain` of blocks.

Usage

```
model(x) = chain(A : inLeftWave(x) : B)
```

Where `A` and `B` are bidirectional blocks and `x` is the signal added to left going waves in that chain.

Test

```
pm = library("physmodels.lib");
inLeftWave_test = 0,0,0 : pm.inLeftWave(0.25);
```

(pm.)inRightWave

Adds a signal to right going waves anywhere in a `chain` of blocks.

Usage

```
model(x) = chain(A : inRightWave(x) : B)
```

Where `A` and `B` are bidirectional blocks and `x` is the signal added to right going waves in that chain.

Test

```
pm = library("physmodels.lib");
inRightWave_test = 0,0,0 : pm.inRightWave(0.25);
```

`(pm.)in`

Adds a signal to left and right going waves anywhere in a `chain` of blocks.

Usage

```
model(x) = chain(A : in(x) : B)
```

Where `A` and `B` are bidirectional blocks and `x` is the signal added to left and right going waves in that chain.

Test

```
pm = library("physmodels.lib");
in_test = 0,0,0 : pm.in(0.25);
```

`(pm.)outLeftWave`

Sends the signal of left going waves to the output channel of the `chain`.

Usage

```
chain(A : outLeftWave : B)
```

Where `A` and `B` are bidirectional blocks.

Test

```
pm = library("physmodels.lib");
outLeftWave_test = pm.outLeftWave(0.1, 0.2, 0.3);
```

`(pm.)outRightWave`

Sends the signal of right going waves to the output channel of the `chain`.

Usage

```
chain(A : outRightWave : B)
```

Where `A` and `B` are bidirectional blocks.

Test

```
pm = library("physmodels.lib");
outRightWave_test = pm.outRightWave(0.1, 0.2, 0.3);
```

`(pm.)out`

Sends the signal of right and left going waves to the output channel of the `chain`.

Usage

```
chain(A : out : B)
```

Where `A` and `B` are bidirectional blocks.

Test

```
pm = library("physmodels.lib");
out_test = pm.out(0.1, 0.2, 0.3);
```

`(pm.)terminations`

Creates terminations on both sides of a `chain` without closing the inputs and outputs of the bidirectional signals chain. As for `chain`, this function adds a 1 sample delay to the bidirectional signal, both ways. Of course, this function can be nested within a `chain`.

Usage

```
terminations(a,b,c)
with {
   a = *(-1); // left termination
   b = chain(D : E : F); // bidirectional chain of blocks (D, E, F, etc.)
   c = *(-1); // right termination
};
```

Test

```
pm = library("physmodels.lib");
terminations_test = 0,0,0 : pm.terminations(*(-1), pm.basicBlock, *(-1));
```

(pm.)lTermination

Creates a termination on the left side of a `chain` without closing the inputs and outputs of the bidirectional signals chain. This function adds a 1 sample delay near the termination and can be nested within another `chain`.

Usage

```
lTerminations(a,b)
with {
   a = *(-1); // left termination
   b = chain(D : E : F); // bidirectional chain of blocks (D, E, F, etc.)
};
```

Test

```
pm = library("physmodels.lib");
lTermination_test = 0,0,0 : pm.lTermination(*(-1), pm.basicBlock);
```

(pm.)rTermination

Creates a termination on the right side of a `chain` without closing the inputs and outputs of the bidirectional signals chain. This function adds a 1 sample delay near the termination and can be nested within another `chain`.

Usage

```
rTerminations(b,c)
with {
   b = chain(D : E : F); // bidirectional chain of blocks (D, E, F, etc.)
   c = *(-1); // right termination
};
```

Test

```
pm = library("physmodels.lib");
rTermination_test = 0,0,0 : pm.rTermination(pm.basicBlock, *(-1));
```

(pm.)closeIns

Closes the inputs of a bidirectional chain in all directions.

Usage

```
closeIns : chain(...) : _,_,_
```

Test

```
pm = library("physmodels.lib");
closeIns_test = pm.closeIns;
```

(pm.)closeOuts

Closes the outputs of a bidirectional chain in all directions except for the main signal output (3d output).

Usage

```
_,_,_ : chain(...) : _
```

Test

```
pm = library("physmodels.lib");
closeOuts_test = 0,0,0 : pm.closeOuts;
```

(pm.)endChain

Closes the inputs and outputs of a bidirectional chain in all directions except for the main signal output (3d output).

Usage

```
endChain(chain(...)) : _
```

Test

```
pm = library("physmodels.lib");
endChain_test = 0,0,0 : pm.endChain(pm.basicBlock);
```

Basic Elements

Basic elements for physical modeling (e.g., waveguides, specific filters, etc.).

(pm.)waveguideN

A series of waveguide functions based on various types of delays (see `fdelay[n]`).

List of functions

- `waveguideUd`: unit delay waveguide
- `waveguideFd`: fractional delay waveguide
- `waveguideFd2`: second order fractional delay waveguide
- `waveguideFd4`: fourth order fractional delay waveguide

Usage

```
chain(A : waveguideUd(nMax,n) : B)
```

Where:

- `nMax`: the maximum length of the delays in the waveguide
- `n`: the length of the delay lines in samples.

Test

```
pm = library("physmodels.lib");
waveguideUd_test = 0,0,0 : pm.waveguideUd(512, 32);
waveguideFd_test = 0,0,0 : pm.waveguideFd(512, 32);
waveguideFd2_test = 0,0,0 : pm.waveguideFd2(512, 32);
waveguideFd4_test = 0,0,0 : pm.waveguideFd4(512, 32);
```

`(pm.)waveguide`

Standard `pm.lib` waveguide (based on `waveguideFd4`).

Usage

```
chain(A : waveguide(nMax,n) : B)
```

Where:

- `nMax`: the maximum length of the delays in the waveguide
- `n`: the length of the delay lines in samples.

Test

```
pm = library("physmodels.lib");
waveguide_test = 0,0,0 : pm.waveguide(512, 32);
```

`(pm.)bridgeFilter`

Generic two zeros bridge FIR filter (as implemented in the STK) that can be used to implement the reflectance violin, guitar, etc. bridges.

Usage

```
_ : bridge(brightness,absorption) : _
```

Where:

- `brightness`: controls the damping of high frequencies (0-1)
- `absorption`: controls the absorption of the brige and thus the t60 of the string plugged to it (0-1) (1 = 20 seconds)

Test

```
pm = library("physmodels.lib");
bridgeFilter_test = pm.bridgeFilter(0.6, 0.4, os.osc(110));
```

`(pm.)modeFilter`

Resonant bandpass filter that can be used to implement a single resonance (mode).

Usage

```
_ : modeFilter(freq,t60,gain) : _
```

Where:

- `freq`: mode frequency
- `t60`: mode resonance duration (in seconds)
- `gain`: mode gain (0-1)

Test

```
pm = library("physmodels.lib");
modeFilter_test = pm.modeFilter(440, 1.5, 0.8);
```

String Instruments

Low and high level string instruments parts. Most of the elements in this section can be used in a bidirectional chain.

`(pm.)stringSegment`

A string segment without terminations (just a simple waveguide).

Usage

```
chain(A : stringSegment(maxLength,length) : B)
```

Where:

- `maxLength`: the maximum length of the string in meters (should be static)
- `length`: the length of the string in meters

Test

```
pm = library("physmodels.lib");
stringSegment_test = 0,0,0 : pm.stringSegment(1.0, 0.5);
```

(pm.)openString

A bidirectional block implementing a basic "generic" string with a selectable excitation position. Lowpass filters are built-in and allow to simulate the effect of dispersion on the sound and thus to change the "stiffness" of the string.

Usage

```
chain(... : openString(length,stiffness,pluckPosition,excitation) : ...)
```

Where:

- `length`: the length of the string in meters
- `stiffness`: the stiffness of the string (0-1) (1 for max stiffness)
- `pluckPosition`: excitation position (0-1) (1 is bottom)
- `excitation`: the excitation signal

Test

```
pm = library("physmodels.lib");
openString_test = 0,0,0 : pm.openString(0.8, 0.5, 0.2, pm.impulseExcitation(button("gate")))
```

(pm.)nylonString

A bidirectional block implementing a basic nylon string with selectable excitation position. This element is based on `openString` and has a fix stiffness corresponding to that of a nylon string.

Usage

```
chain(... : nylonString(length,pluckPosition,excitation) : ...)
```

Where:

- `length`: the length of the string in meters
- `pluckPosition`: excitation position (0-1) (1 is bottom)
- `excitation`: the excitation signal

Test

```
pm = library("physmodels.lib");
nylonString_test = 0,0,0 : pm.nylonString(0.8, 0.3, pm.impulseExcitation(button("gate")));
```

`(pm.)steelString`

A bidirectional block implementing a basic steel string with selectable excitation position. This element is based on `openString` and has a fix stiffness corresponding to that of a steel string.

Usage

```
chain(... : steelString(length,pluckPosition,excitation) : ...)
```

Where:

- `length`: the length of the string in meters
- `pluckPosition`: excitation position (0-1) (1 is bottom)
- `excitation`: the excitation signal

Test

```
pm = library("physmodels.lib");
steelString_test = 0,0,0 : pm.steelString(0.8, 0.3, pm.impulseExcitation(button("gate")));
```

`(pm.)openStringPick`

A bidirectional block implementing a "generic" string with selectable excitation position. It also has a built-in pickup whose position is the same as the excitation position. Thus, moving the excitation position will also move the pickup.

Usage

```
chain(... : openStringPick(length,stiffness,pluckPosition,excitation) : ...)
```

Where:

- `length`: the length of the string in meters
- `stiffness`: the stiffness of the string (0-1) (1 for max stiffness)
- `pluckPosition`: excitation position (0-1) (1 is bottom)
- `excitation`: the excitation signal

Test

```
pm = library("physmodels.lib");
openStringPick_test = 0,0,0 : pm.openStringPick(0.8, 0.4, 0.3, pm.impulseExcitation(button('
```

`(pm.)openStringPickUp`

A bidirectional block implementing a "generic" string with selectable excitation position and stiffness. It also has a built-in pickup whose position can be independenly selected. The only constraint is that the pickup has to be placed after the excitation position.

Usage

```
chain(... : openStringPickUp(length,stiffness,pluckPosition,excitation) : ...)
```

Where:

- `length`: the length of the string in meters
- `stiffness`: the stiffness of the string (0-1) (1 for max stiffness)
- `pluckPosition`: pluck position between the top of the string and the pickup (0-1) (1 for same as pickup position)
- `pickupPosition`: position of the pickup on the string (0-1) (1 is bottom)
- `excitation`: the excitation signal

Test

```
pm = library("physmodels.lib");
openStringPickUp_test = 0,0,0 : pm.openStringPickUp(0.8, 0.4, 0.6, 0.7, pm.impulseExcitatio
```

`(pm.)openStringPickDown`

A bidirectional block implementing a "generic" string with selectable excitation position and stiffness. It also has a built-in pickup whose position can be independenly selected. The only constraint is that the pickup has to be placed before the excitation position.

Usage

```
chain(... : openStringPickDown(length,stiffness,pluckPosition,excitation) : ...)
```

Where:

- `length`: the length of the string in meters
- `stiffness`: the stiffness of the string (0-1) (1 for max stiffness)
- `pluckPosition`: pluck position on the string (0-1) (1 is bottom)

- `pickupPosition`: position of the pickup between the top of the string and the excitation position (0-1) (1 is excitation position)
- `excitation`: the excitation signal

Test

```
pm = library("physmodels.lib");
openStringPickDown_test = 0,0,0 : pm.openStringPickDown(0.8, 0.4, 0.6, 0.5, pm.impulseExcita
```

(pm.)ksReflexionFilter

The "typical" one-zero Karplus-strong feedforward reflexion filter. This filter will be typically used in a termination (see below).

Usage

```
terminations(_,chain(...),ksReflexionFilter)
```

Test

```
pm = library("physmodels.lib");
os = library("oscillators.lib");
ksReflexionFilter_test = os.osc(220) : pm.ksReflexionFilter;
```

(pm.)rStringRigidTermination

Bidirectional block implementing a right rigid string termination (no damping, just phase inversion).

Usage

```
chain(rStringRigidTermination : stringSegment : ...)
```

Test

```
pm = library("physmodels.lib");
rStringRigidTermination_test = 0,0,0 : pm.rStringRigidTermination;
```

(pm.)lStringRigidTermination

Bidirectional block implementing a left rigid string termination (no damping, just phase inversion).

Usage

```
chain(... : stringSegment : lStringRigidTermination)
```

Test

```
pm = library("physmodels.lib");
lStringRigidTermination_test = 0,0,0 : pm.lStringRigidTermination;
```

(pm.)elecGuitarBridge

Bidirectional block implementing a simple electric guitar bridge. This block is based on `bridgeFilter`. The bridge doesn't implement transmittance since it is not meant to be connected to a body (unlike acoustic guitar). It also partially sets the resonance duration of the string with the nuts used on the other side.

Usage

```
chain(... : stringSegment : elecGuitarBridge)
```

Test

```
pm = library("physmodels.lib");
elecGuitarBridge_test = 0,0,0 : pm.elecGuitarBridge;
```

(pm.)elecGuitarNuts

Bidirectional block implementing a simple electric guitar nuts. This block is based on `bridgeFilter` and does essentially the same thing as `elecGuitarBridge`, but on the other side of the chain. It also partially sets the resonance duration of the string with the bridge used on the other side.

Usage

```
chain(elecGuitarNuts : stringSegment : ...)
```

Test

```
pm = library("physmodels.lib");
elecGuitarNuts_test = 0,0,0 : pm.elecGuitarNuts;
```

(pm.)guitarBridge

Bidirectional block implementing a simple acoustic guitar bridge. This bridge damps more hight frequencies than `elecGuitarBridge` and implements a transmittance filter. It also partially sets the resonance duration of the string with the nuts used on the other side.

Usage

```
chain(... : stringSegment : guitarBridge)
```

Test

```
pm = library("physmodels.lib");
guitarBridge_test = 0,0,0 : pm.guitarBridge;
```

(pm.)guitarNuts

Bidirectional block implementing a simple acoustic guitar nuts. This nuts damps more hight frequencies than `elecGuitarNuts` and implements a transmittance filter. It also partially sets the resonance duration of the string with the bridge used on the other side.

Usage

```
chain(guitarNuts : stringSegment : ...)
```

Test

```
pm = library("physmodels.lib");
guitarNuts_test = 0,0,0 : pm.guitarNuts;
```

(pm.)idealString

An "ideal" string with rigid terminations and where the plucking position and the pick-up position are the same. Since terminations are rigid, this string will ring forever.

Usage

```
1-1' : idealString(length,reflexion,xPosition,excitation)
```

With:

- `length`: the length of the string in meters
- `pluckPosition`: the plucking position (0.001-0.999)
- `excitation`: the input signal for the excitation.

Test

```
pm = library("physmodels.lib");
idealString_test = 0,0,0 : pm.idealString(0.9, 0.2, pm.impulseExcitation(button("gate")));
```

(pm.)ks

A Karplus-Strong string (in that case, the string is implemented as a one dimension waveguide).

Usage

```
ks(length,damping,excitation) : _
```

Where:

- `length`: the length of the string in meters
- `damping`: string damping (0-1)
- `excitation`: excitation signal

Test

```
pm = library("physmodels.lib");
ks_test = pm.ks(0.9, 0.3, pm.impulseExcitation(button("gate")));
```

(pm.)ks_ui_MIDI

Ready-to-use, MIDI-enabled Karplus-Strong string with buil-in UI.

Usage

```
ks_ui_MIDI : _
```

Test

```
pm = library("physmodels.lib");
ks_ui_MIDI_test = pm.ks_ui_MIDI;
```

(pm.)elecGuitarModel

A simple electric guitar model (without audio effects, of course) with selectable pluck position. This model implements a single string. Additional strings should be created by making a polyphonic application out of this function. Pitch is changed by changing the length of the string and not through a finger model.

Usage

```
elecGuitarModel(length,pluckPosition,mute,excitation) : _
```

Where:

- `length`: the length of the string in meters
- `pluckPosition`: pluck position (0-1) (1 is on the bridge)
- `mute`: mute coefficient (1 for no mute and 0 for instant mute)
- `excitation`: excitation signal

Test

```
pm = library("physmodels.lib");
elecGuitarModel_test = pm.elecGuitarModel(0.9, 0.3, 0.8, pm.impulseExcitation(button("gate")
```

(pm.)elecGuitar

A simple electric guitar model with steel strings (based on `elecGuitarModel`) implementing an excitation model. This model implements a single string. Additional strings should be created by making a polyphonic application out of this function.

Usage

```
elecGuitar(length,pluckPosition,trigger) : _
```

Where:

- `length`: the length of the string in meters
- `pluckPosition`: pluck position (0-1) (1 is on the bridge)
- `mute`: mute coefficient (1 for no mute and 0 for instant mute)
- `gain`: gain of the pluck (0-1)
- `trigger`: trigger signal (1 for on, 0 for off)

Test

```
pm = library("physmodels.lib");
elecGuitar_test = pm.elecGuitar(0.9, 0.3, 0.8, 0.6, button("gate"));
```

(pm.)elecGuitar_ui_MIDI

Ready-to-use MIDI-enabled electric guitar physical model with built-in UI.

Usage

```
elecGuitar_ui_MIDI : _
```

Test

```
pm = library("physmodels.lib");
elecGuitar_ui_MIDI_test = pm.elecGuitar_ui_MIDI;
```

(pm.)guitarBody

WARNING: not implemented yet! Bidirectional block implementing a simple acoustic guitar body.

Usage

```
chain(... : guitarBody)
```

Test

```
pm = library("physmodels.lib");
guitarBody_test = 0,0,0 : pm.guitarBody;
```

(pm.)guitarModel

A simple acoustic guitar model with steel strings and selectable excitation position. This model implements a single string. Additional strings should be created by making a polyphonic application out of this function. Pitch is changed by changing the length of the string and not through a finger model. WARNING: this function doesn't currently implement a body (just strings and bridge).

Usage

```
guitarModel(length,pluckPosition,excitation) : _
```

Where:

- `length`: the length of the string in meters
- `pluckPosition`: pluck position (0-1) (1 is on the bridge)
- `excitation`: excitation signal

Test

```
pm = library("physmodels.lib");
guitarModel_test = pm.guitarModel(0.9, 0.25, pm.impulseExcitation(button("gate")));
```

`(pm.)guitar`

A simple acoustic guitar model with steel strings (based on `guitarModel`) implementing an excitation model. This model implements a single string. Additional strings should be created by making a polyphonic application out of this function.

Usage

```
guitar(length,pluckPosition,trigger) : _
```

Where:

- `length`: the length of the string in meters
- `pluckPosition`: pluck position (0-1) (1 is on the bridge)
- `gain`: gain of the excitation
- `trigger`: trigger signal (1 for on, 0 for off)

Test

```
pm = library("physmodels.lib");
guitar_test = pm.guitar(0.9, 0.25, 0.8, button("gate"));
```

`(pm.)guitar_ui_MIDI`

Ready-to-use MIDI-enabled steel strings acoustic guitar physical model with built-in UI.

Usage

```
guitar_ui_MIDI : _
```

Test

```
pm = library("physmodels.lib");
guitar_ui_MIDI_test = pm.guitar_ui_MIDI;
```

`(pm.)nylonGuitarModel`

A simple acoustic guitar model with nylon strings and selectable excitation position. This model implements a single string. Additional strings should be created by making a polyphonic application out of this function. Pitch is changed by changing the length of the string and not through a finger model. WARNING: this function doesn't currently implement a body (just strings and bridge).

Usage

```
nylonGuitarModel(length,pluckPosition,excitation) : _
```

Where:

- `length`: the length of the string in meters
- `pluckPosition`: pluck position (0-1) (1 is on the bridge)
- `excitation`: excitation signal

Test

```
pm = library("physmodels.lib");
nylonGuitarModel_test = pm.nylonGuitarModel(0.9, 0.25, pm.impulseExcitation(button("gate"))
```

(pm.)nylonGuitar

A simple acoustic guitar model with nylon strings (based on `nylonGuitarModel`) implementing an excitation model. This model implements a single string. Additional strings should be created by making a polyphonic application out of this function.

Usage

```
nylonGuitar(length,pluckPosition,trigger) : _
```

Where:

- `length`: the length of the string in meters
- `pluckPosition`: pluck position (0-1) (1 is on the bridge)
- `gain`: gain of the excitation (0-1)
- `trigger`: trigger signal (1 for on, 0 for off)

Test

```
pm = library("physmodels.lib");
nylonGuitar_test = pm.nylonGuitar(0.9, 0.25, 0.8, button("gate"));
```

(pm.)nylonGuitar_ui_MIDI

Ready-to-use MIDI-enabled nylon strings acoustic guitar physical model with built-in UI.

Usage

```
nylonGuitar_ui_MIDI : _
```

Test

```
pm = library("physmodels.lib");
nylonGuitar_ui_MIDI_test = pm.nylonGuitar_ui_MIDI;
```

(pm.)modeInterpRes

Modular string instrument resonator based on IR measurements made on 3D printed models. The 2D space allowing for the control of the shape and the scale of the model is enabled by interpolating between modes parameters. More information about this technique/project can be found here: * https://ccrma.stanford.edu/~rmichon/3dPrintingModeling/.

Usage

```
_ : modeInterpRes(nModes,x,y) : _
```

Where:

- `nModes`: number of modeled modes (40 max)
- `x`: shape of the resonator (0: square, 1: square with rounded corners, 2: round)
- `y`: scale of the resonator (0: small, 1: medium, 2: large)

Test

```
pm = library("physmodels.lib");
os = library("oscillators.lib");
modeInterpRes_test = os.osc(110) : pm.modeInterpRes(20, 1.0, 1.5);
```

(pm.)modularInterpBody

Bidirectional block implementing a modular string instrument resonator (see `modeInterpRes`).

Usage

```
chain(... : modularInterpBody(nModes,shape,scale) : ...)
```

Where:

- `nModes`: number of modeled modes (40 max)
- `shape`: shape of the resonator (0: square, 1: square with rounded corners, 2: round)
- `scale`: scale of the resonator (0: small, 1: medium, 2: large)

Test

```
pm = library("physmodels.lib");
modularInterpBody_test = 0,0,0 : pm.modularInterpBody(20, 1.0, 1.5);
```

(pm.)modularInterpStringModel

String instrument model with a modular body (see `modeInterpRes` and * https://ccrma.stanford.edu/~rmichon/3dPrintingModeling/).

Usage

```
modularInterpStringModel(length,pluckPosition,shape,scale,bodyExcitation,stringExcitation) :
```

Where:

- `stringLength`: the length of the string in meters
- `pluckPosition`: pluck position (0-1) (1 is on the bridge)
- `shape`: shape of the resonator (0: square, 1: square with rounded corners, 2: round)
- `scale`: scale of the resonator (0: small, 1: medium, 2: large)
- `bodyExcitation`: excitation signal for the body
- `stringExcitation`: excitation signal for the string

Test

```
pm = library("physmodels.lib");
modularInterpStringModel_test = pm.modularInterpStringModel(0.9, 0.3, 1.0, 1.5, pm.impulseEx
```

(pm.)modularInterpInstr

String instrument with a modular body (see `modeInterpRes` and * https://ccrma.stanford.edu/~rmichon/3dPrintingModeling/).

Usage

```
modularInterpInstr(stringLength,pluckPosition,shape,scale,gain,tapBody,triggerString) : _
```

Where:

- `stringLength`: the length of the string in meters
- `pluckPosition`: pluck position (0-1) (1 is on the bridge)
- `shape`: shape of the resonator (0: square, 1: square with rounded corners, 2: round)
- `scale`: scale of the resonator (0: small, 1: medium, 2: large)
- `gain`: of the string excitation

- `tapBody`: send an impulse in the body of the instrument where the string is connected (1 for on, 0 for off)
- `triggerString`: trigger signal for the string (1 for on, 0 for off)

Test

```
pm = library("physmodels.lib");
modularInterpInstr_test = pm.modularInterpInstr(0.9, 0.3, 1.0, 1.5, 0.8, button("body"), but
```

(pm.)modularInterpInstr_ui_MIDI

Ready-to-use MIDI-enabled string instrument with a modular body (see `modeInterpRes` and * https://ccrma.stanford.edu/~rmichon/3dPrintingModeling/) with built-in UI.

Usage

```
modularInterpInstr_ui_MIDI : _
```

Test

```
pm = library("physmodels.lib");
modularInterpInstr_ui_MIDI_test = pm.modularInterpInstr_ui_MIDI;
```

Bowed String Instruments

Low and high level basic string instruments parts. Most of the elements in this section can be used in a bidirectional chain.

(pm.)bowTable

Extremely basic bow table that can be used to implement a wide range of bow types for many different bowed string instruments (violin, cello, etc.).

Usage

```
excitation : bowTable(offset,slope) : _
```

Where:

- `excitation`: an excitation signal
- `offset`: table offset
- `slope`: table slope

Test

```
pm = library("physmodels.lib");
bowTable_test = pm.bowTable(0.4, 0.1);
```

(pm.)violinBowTable

Violin bow table based on `bowTable`.

Usage

```
bowVelocity : violinBowTable(bowPressure) : _
```

Where:

- `bowVelocity`: velocity of the bow/excitation signal (0-1)
- `bowPressure`: bow pressure on the string (0-1)

Test

```
pm = library("physmodels.lib");
violinBowTable_test = pm.violinBowTable(0.4, 0.1);
```

(pm.)bowInteraction

Bidirectional block implementing the interaction of a bow in a `chain`.

Usage

```
chain(... : stringSegment : bowInteraction(bowTable) : stringSegment : ...)
```

Where:

- `bowTable`: the bow table

Test

```
pm = library("physmodels.lib");
bowInteraction_test = pm.bowInteraction((0.4, 0.05));
```

(pm.)violinBow

Bidirectional block implementing a violin bow and its interaction with a string.

Usage

```
chain(... : stringSegment : violinBow(bowPressure,bowVelocity) : stringSegment : ...)
```

Where:

- `bowVelocity`: velocity of the bow / excitation signal (0-1)
- `bowPressure`: bow pressure on the string (0-1)

Test

```
pm = library("physmodels.lib");
violinBow_test = pm.violinBow(0.4, 0.05);
```

(pm.)violinBowedString

Violin bowed string bidirectional block with controllable bow position. Terminations are not implemented in this model.

Usage

```
chain(nuts : violinBowedString(stringLength,bowPressure,bowVelocity,bowPosition) : bridge)
```

Where:

- `stringLength`: the length of the string in meters
- `bowVelocity`: velocity of the bow / excitation signal (0-1)
- `bowPressure`: bow pressure on the string (0-1)
- `bowPosition`: the position of the bow on the string (0-1)

Test

```
pm = library("physmodels.lib");
violinBowedString_test = 0,0,0 : pm.violinBowedString(0.82, 0.35, pm.violinBow(0.4, 0.05), (
```

(pm.)violinNuts

Bidirectional block implementing simple violin nuts. This function is based on `bridgeFilter`.

Usage

```
chain(violinNuts : stringSegment : ...)
```

Test

```
pm = library("physmodels.lib");
violinNuts_test = 0,0,0 : pm.violinNuts;
```

(pm.)violinBridge

Bidirectional block implementing a simple violin bridge. This function is based on `bridgeFilter`.

Usage

```
chain(... : stringSegment : violinBridge
```

Test

```
pm = library("physmodels.lib");
violinBridge_test = 0,0,0 : pm.violinBridge;
```

(pm.)violinBody

Bidirectional block implementing a simple violin body (just a simple resonant lowpass filter).

Usage

```
chain(... : stringSegment : violinBridge : violinBody)
```

Test

```
pm = library("physmodels.lib");
violinBody_test = 0,0,0 : pm.violinBody;
```

(pm.)violinModel

Ready-to-use simple violin physical model. This model implements a single string. Additional strings should be created by making a polyphonic application out of this function. Pitch is changed by changing the length of the string (and not through a finger model).

Usage

```
violinModel(stringLength,bowPressure,bowVelocity,bridgeReflexion,
bridgeAbsorption,bowPosition) : _
```

Where:

- `stringLength`: the length of the string in meters
- `bowVelocity`: velocity of the bow / excitation signal (0-1)
- `bowPressure`: bow pressure on the string (0-1))
- `bowPosition`: the position of the bow on the string (0-1)

Test

```
pm = library("physmodels.lib");
violinModel_test = pm.violinModel(0.82, 0.35, pm.violinBow(0.4, 0.05), 0.15);
```

`(pm.)violin_ui`

Ready-to-use violin physical model with built-in UI.

Usage

```
violinModel_ui : _
```

Test

```
pm = library("physmodels.lib");
violin_ui_test = pm.violin_ui;
```

`(pm.)violin_ui_MIDI`

Ready-to-use MIDI-enabled violin physical model with built-in UI.

Usage

```
violin_ui_MIDI : _
```

Test

```
pm = library("physmodels.lib");
violin_ui_MIDI_test = pm.violin_ui_MIDI;
```

Wind Instruments

Low and high level basic wind instruments parts. Most of the elements in this section can be used in a bidirectional chain.

(pm.)openTube

A tube segment without terminations (same as `stringSegment`).

Usage

```
chain(A : openTube(maxLength,length) : B)
```

Where:

- `maxLength`: the maximum length of the tube in meters (should be static)
- `length`: the length of the tube in meters

Test

```
pm = library("physmodels.lib");
openTube_test = pm.openTube(0.9);
```

(pm.)reedTable

Extremely basic reed table that can be used to implement a wide range of single reed types for many different instruments (saxophone, clarinet, etc.).

Usage

```
excitation : reedTable(offeset,slope) : _
```

Where:

- `excitation`: an excitation signal
- `offset`: table offset
- `slope`: table slope

Test

```
pm = library("physmodels.lib");
reedTable_test = pm.reedTable(0.4, 0.2);
```

(pm.)fluteJetTable

Extremely basic flute jet table.

Usage

```
excitation : fluteJetTable : _
```

Where:

- `excitation`: an excitation signal

Test

```
pm = library("physmodels.lib");
fluteJetTable_test = pm.fluteJetTable(0.5);
```

`(pm.)brassLipsTable`

Simple brass lips/mouthpiece table. Since this implementation is very basic and that the lips and tube of the instrument are coupled to each other, the length of that tube must be provided here.

Usage

```
excitation : brassLipsTable(tubeLength,lipsTension) : _
```

Where:

- `excitation`: an excitation signal (can be DC)
- `tubeLength`: length in meters of the tube connected to the mouthpiece
- `lipsTension`: tension of the lips (0-1) (default: 0.5)

Test

```
pm = library("physmodels.lib");
brassLipsTable_test = pm.brassLipsTable(0.3, 0.2);
```

`(pm.)clarinetReed`

Clarinet reed based on `reedTable` with controllable stiffness.

Usage

```
excitation : clarinetReed(stiffness) : _
```

Where:

- `excitation`: an excitation signal
- `stiffness`: reed stiffness (0-1)

Test

```
pm = library("physmodels.lib");
clarinetReed_test = pm.clarinetReed(0.6, 0.4, 0.1);
```

(pm.)clarinetMouthPiece

Bidirectional block implementing a clarinet mouthpiece as well as the various interactions happening with traveling waves. This element is ready to be plugged to a tube…

Usage

```
chain(clarinetMouthPiece(reedStiffness,pressure) : tube : etc.)
```

Where:

- `pressure`: the pressure of the air flow (DC) created by the virtual performer (0-1). This can also be any kind of signal that will directly injected in the mouthpiece (e.g., breath noise, etc.).
- `reedStiffness`: reed stiffness (0-1)

Test

```
pm = library("physmodels.lib");
clarinetMouthPiece_test = pm.clarinetMouthPiece(0.6, 0.4, 0.1);
```

(pm.)brassLips

Bidirectional block implementing a brass mouthpiece as well as the various interactions happening with traveling waves. This element is ready to be plugged to a tube…

Usage

```
chain(brassLips(tubeLength,lipsTension,pressure) : tube : etc.)
```

Where:

- `tubeLength`: length in meters of the tube connected to the mouthpiece
- `lipsTension`: tension of the lips (0-1) (default: 0.5)
- `pressure`: the pressure of the air flow (DC) created by the virtual performer (0-1). This can also be any kind of signal that will directly injected in the mouthpiece (e.g., breath noise, etc.).

Test

```
pm = library("physmodels.lib");
brassLips_test = pm.brassLips(0.3, 0.2, 0.1);
```

(pm.)fluteEmbouchure

Bidirectional block implementing a flute embouchure as well as the various interactions happening with traveling waves. This element is ready to be plugged between tubes segments...

Usage

```
chain(... : tube : fluteEmbouchure(pressure) : tube : etc.)
```

Where:

- `pressure`: the pressure of the air flow (DC) created by the virtual performer (0-1). This can also be any kind of signal that will directly injected in the mouthpiece (e.g., breath noise, etc.).

Test

```
pm = library("physmodels.lib");
fluteEmbouchure_test = pm.fluteEmbouchure(0.5, 0.3);
```

(pm.)wBell

Generic wind instrument bell bidirectional block that should be placed at the end of a `chain`.

Usage

```
chain(... : wBell(opening))
```

Where:

- `opening`: the "opening" of bell (0-1)

Test

```
pm = library("physmodels.lib");
wBell_test = pm.wBell(0.4, 0.6);
```

(pm.)fluteHead

Simple flute head implementing waves reflexion.

Usage

```
chain(fluteHead : tube : ...)
```

Test

```
pm = library("physmodels.lib");
fluteHead_test = pm.fluteHead(0.8, 0.4, 0.3);
```

(pm.)fluteFoot

Simple flute foot implementing waves reflexion and dispersion.

Usage

```
chain(... : tube : fluteFoot)
```

Test

```
pm = library("physmodels.lib");
fluteFoot_test = pm.fluteFoot(0.8, 0.4, 0.3);
```

(pm.)clarinetModel

A simple clarinet physical model without tone holes (pitch is changed by changing the length of the tube of the instrument).

Usage

```
clarinetModel(length,pressure,reedStiffness,bellOpening) : _
```

Where:

- `tubeLength`: the length of the tube in meters
- `pressure`: the pressure of the air flow created by the virtual performer (0-1). This can also be any kind of signal that will directly injected in the mouthpiece (e.g., breath noise, etc.).
- `reedStiffness`: reed stiffness (0-1)
- `bellOpening`: the opening of bell (0-1)

Test

```
pm = library("physmodels.lib");
clarinetModel_test = pm.clarinetModel(0.9, 0.4, 0.3, 0.2);
```

(pm.)clarinetModel_ui

Same as `clarinetModel` but with a built-in UI. This function doesn't implement a virtual "blower", thus `pressure` remains an argument here.

Usage

```
clarinetModel_ui(pressure) : _
```

Where:

- `pressure`: the pressure of the air flow created by the virtual performer (0-1). This can also be any kind of signal that will be directly injected in the mouthpiece (e.g., breath noise, etc.).

Test

```
pm = library("physmodels.lib");
clarinetModel_ui_test = pm.clarinetModel_ui;
```

(pm.)clarinet_ui

Ready-to-use clarinet physical model with built-in UI based on `clarinetModel`.

Usage

```
clarinet_ui : _
```

Test

```
pm = library("physmodels.lib");
clarinet_ui_test = pm.clarinet_ui;
```

(pm.)clarinet_ui_MIDI

Ready-to-use MIDI compliant clarinet physical model with built-in UI.

Usage

```
clarinet_ui_MIDI : _
```

Test

```
pm = library("physmodels.lib");
clarinet_ui_MIDI_test = pm.clarinet_ui_MIDI;
```

(pm.)brassModel

A simple generic brass instrument physical model without pistons (pitch is changed by changing the length of the tube of the instrument). This model is kind of hard to control and might not sound very good if bad parameters are given to it…

Usage

```
brassModel(tubeLength,lipsTension,mute,pressure) : _
```

Where:

- `tubeLength`: the length of the tube in meters
- `lipsTension`: tension of the lips (0-1) (default: 0.5)
- `mute`: mute opening at the end of the instrument (0-1) (default: 0.5)
- `pressure`: the pressure of the air flow created by the virtual performer (0-1). This can also be any kind of signal that will directly injected in the mouthpiece (e.g., breath noise, etc.).

Test

```
pm = library("physmodels.lib");
brassModel_test = pm.brassModel(0.9, 0.4, 0.2, 0.6);
```

(pm.)brassModel_ui

Same as `brassModel` but with a built-in UI. This function doesn't implement a virtual "blower", thus `pressure` remains an argument here.

Usage

```
brassModel_ui(pressure) : _
```

Where:

- `pressure`: the pressure of the air flow created by the virtual performer (0-1). This can also be any kind of signal that will be directly injected in the mouthpiece (e.g., breath noise, etc.).

Test

```
pm = library("physmodels.lib");
brassModel_ui_test = pm.brassModel_ui;
```

(pm.)brass_ui

Ready-to-use brass instrument physical model with built-in UI based on `brassModel`.

Usage

```
brass_ui : _
```

Test

```
pm = library("physmodels.lib");
brass_ui_test = pm.brass_ui;
```

(pm.)brass_ui_MIDI

Ready-to-use MIDI-controllable brass instrument physical model with built-in UI.

Usage

```
brass_ui_MIDI : _
```

Test

```
pm = library("physmodels.lib");
brass_ui_MIDI_test = pm.brass_ui_MIDI;
```

(pm.)fluteModel

A simple generic flute instrument physical model without tone holes (pitch is changed by changing the length of the tube of the instrument).

Usage

```
fluteModel(tubeLength,mouthPosition,pressure) : _
```

Where:

- `tubeLength`: the length of the tube in meters

- `mouthPosition`: position of the mouth on the embouchure (0-1) (default: 0.5)
- `pressure`: the pressure of the air flow created by the virtual performer (0-1). This can also be any kind of signal that will directly injected in the mouthpiece (e.g., breath noise, etc.).

Test

```
pm = library("physmodels.lib");
fluteModel_test = pm.fluteModel(0.9, 0.4, 0.6);
```

(pm.)fluteModel_ui

Same as `fluteModel` but with a built-in UI. This function doesn't implement a virtual "blower", thus `pressure` remains an argument here.

Usage

```
fluteModel_ui(pressure) : _
```

Where:

- `pressure`: the pressure of the air flow created by the virtual performer (0-1). This can also be any kind of signal that will be directly injected in the mouthpiece (e.g., breath noise, etc.).

Test

```
pm = library("physmodels.lib");
fluteModel_ui_test = pm.fluteModel_ui;
```

(pm.)flute_ui

Ready-to-use flute physical model with built-in UI based on `fluteModel`.

Usage

```
flute_ui : _
```

Test

```
pm = library("physmodels.lib");
flute_ui_test = pm.flute_ui;
```

(pm.)flute_ui_MIDI

Ready-to-use MIDI-controllable flute physical model with built-in UI.

Usage

```
flute_ui_MIDI : _
```

Test

```
pm = library("physmodels.lib");
flute_ui_MIDI_test = pm.flute_ui_MIDI;
```

Exciters

Various kind of excitation signal generators.

(pm.)impulseExcitation

Creates an impulse excitation of one sample.

Usage

```
gate = button('gate');
impulseExcitation(gate) : chain;
```

Where:

- `gate`: a gate button

Test

```
pm = library("physmodels.lib");
impulseExcitation_test = pm.impulseExcitation(button("gate"));
```

(pm.)strikeModel

Creates a filtered noise excitation.

Usage

```
gate = button('gate');
strikeModel(LPcutoff,HPcutoff,sharpness,gain,gate) : chain;
```

Where:

- `HPcutoff`: highpass cutoff frequency

- `LPcutoff`: lowpass cutoff frequency
- `sharpness`: sharpness of the attack and release (0-1)
- `gain`: gain of the excitation
- `gate`: a gate button/trigger signal (0/1)

Test

```
pm = library("physmodels.lib");
strikeModel_test = pm.strikeModel(200, 4000, 0.5, 0.8, button("gate"));
```

`(pm.)strike`

Strikes generator with controllable excitation position.

Usage

```
gate = button('gate');
strike(exPos,sharpness,gain,gate) : chain;
```

Where:

- `exPos`: excitation position wiht 0: for max low freqs and 1: for max high freqs. So, on membrane for example, 0 would be the middle and 1 the edge
- `sharpness`: sharpness of the attack and release (0-1)
- `gain`: gain of the excitation
- `gate`: a gate button/trigger signal (0/1)

Test

```
pm = library("physmodels.lib");
strike_test = pm.strike(0.4, 0.5, 0.8, button("gate"));
```

`(pm.)pluckString`

Creates a plucking excitation signal.

Usage

```
trigger = button('gate');
pluckString(stringLength,cutoff,maxFreq,sharpness,trigger)
```

Where:

- `stringLength`: length of the string to pluck
- `cutoff`: cutoff ratio (1 for default)
- `maxFreq`: max frequency ratio (1 for default)

- `sharpness`: sharpness of the attack and release (1 for default)
- `gain`: gain of the excitation (0-1)
- `trigger`: trigger signal (1 for on, 0 for off)

Test

```
pm = library("physmodels.lib");
pluckString_test = pm.pluckString(0.9, 1, 1, 1, 0.6, button("gate"));
```

`(pm.)blower`

A virtual blower creating a DC signal with some breath noise in it.

Usage

```
blower(pressure,breathGain,breathCutoff) : _
```

Where:

- `pressure`: pressure (0-1)
- `breathGain`: breath noise gain (0-1) (recommended: 0.005)
- `breathCutoff`: breath cuttoff frequency (Hz) (recommended: 2000)

Test

```
pm = library("physmodels.lib");
blower_test = pm.blower(0.5, 0.05, 2000, 5, 0.2);
```

`(pm.)blower_ui`

Same as `blower` but with a built-in UI.

Usage

```
blower : somethingToBeBlown
```

Test

```
pm = library("physmodels.lib");
blower_ui_test = pm.blower_ui;
```

Modal Percussions

High and low level functions for modal synthesis of percussion instruments.

(pm.)djembeModel

Dirt-simple djembe modal physical model. Mode parameters are empirically calculated and don't correspond to any measurements or 3D model. They kind of sound good though :).

Usage

```
excitation : djembeModel(freq)
```

Where:

- `excitation`: excitation signal
- `freq`: fundamental frequency of the bar

Test

```
pm = library("physmodels.lib");
djembeModel_test = pm.djembeModel(110);
```

(pm.)djembe

Dirt-simple djembe modal physical model. Mode parameters are empirically calculated and don't correspond to any measurements or 3D model. They kind of sound good though :).

This model also implements a virtual "exciter".

Usage

```
djembe(freq,strikePosition,strikeSharpness,gain,trigger)
```

Where:

- `freq`: fundamental frequency of the model
- `strikePosition`: strike position (0 for the middle of the membrane and 1 for the edge)
- `strikeSharpness`: sharpness of the strike (0-1, default: 0.5)
- `gain`: gain of the strike
- `trigger`: trigger signal (0: off, 1: on)

Test

```
pm = library("physmodels.lib");
djembe_test = pm.djembe(110, 0.3, 0.5, 0.8, button("gate"));
```

(pm.)djembe_ui_MIDI

Simple MIDI controllable djembe physical model with built-in UI.

Usage

```
djembe_ui_MIDI : _
```

Test

```
pm = library("physmodels.lib");
djembe_ui_MIDI_test = pm.djembe_ui_MIDI;
```

(pm.)marimbaBarModel

Generic marimba tone bar modal model.

This model was generated using `mesh2faust` from a 3D CAD model of a marimba tone bar (`libraries/modalmodels/marimbaBar`). The corresponding CAD model is that of a C2 tone bar (original fundamental frequency: ~65Hz). While `marimbaBarModel` allows to translate the harmonic content of the generated sound by providing a frequency (`freq`), mode transposition has limits and the model will sound less and less like a marimba tone bar as it diverges from C2. To make an accurate model of a marimba, we'd want to have an independent model for each bar…

This model contains 5 excitation positions going linearly from the center bottom to the center top of the bar. Obviously, a model with more excitation position could be regenerated using `mesh2faust`.

Usage

```
excitation : marimbaBarModel(freq,exPos,t60,t60DecayRatio,t60DecaySlope)
```

Where:

- `excitation`: excitation signal
- `freq`: fundamental frequency of the bar
- `exPos`: excitation position (0-4)
- `t60`: T60 in seconds (recommended value: 0.1)
- `t60DecayRatio`: T60 decay ratio (recommended value: 1)
- `t60DecaySlope`: T60 decay slope (recommended value: 5)

Test

```
pm = library("physmodels.lib");
marimbaBarModel_test = pm.marimbaBarModel(220);
```

(pm.)marimbaResTube

Simple marimba resonance tube.

Usage

```
marimbaResTube(tubeLength,excitation)
```

Where:

- `tubeLength`: the length of the tube in meters
- `excitation`: the excitation signal (audio in)

Test

```
pm = library("physmodels.lib");
marimbaResTube_test = pm.marimbaResTube(220);
```

(pm.)marimbaModel

Simple marimba physical model implementing a single tone bar connected to tube. This model is scalable and can be adapted to any size of bar/tube (see `marimbaBarModel` to know more about the limitations of this type of system).

Usage

```
excitation : marimbaModel(freq,exPos) : _
```

Where:

- `excitation`: the excitation signal
- `freq`: the frequency of the bar/tube couple
- `exPos`: excitation position (0-4)

Test

```
pm = library("physmodels.lib");
marimbaModel_test = pm.marimbaModel(220);
```

(pm.)marimba

Simple marimba physical model implementing a single tone bar connected to tube. This model is scalable and can be adapted to any size of bar/tube (see `marimbaBarModel` to know more about the limitations of this type of system).

This function also implement a virtual exciter to drive the model.

Usage

```
marimba(freq,strikePosition,strikeCutoff,strikeSharpness,gain,trigger) : _
```

Where:

- `freq`: the frequency of the bar/tube couple
- `strikePosition`: strike position (0-4)
- `strikeCutoff`: cuttoff frequency of the strike genarator (recommended: ~7000Hz)
- `strikeSharpness`: sharpness of the strike (recommended: ~0.25)
- `gain`: gain of the strike (0-1)
- `trigger` signal (0: off, 1: on)

Test

```
pm = library("physmodels.lib");
marimba_test = pm.marimba(220, 0.4, 1, 0.5, 0.8, button("gate"));
```

(pm.)marimba_ui_MIDI

Simple MIDI controllable marimba physical model with built-in UI implementing a single tone bar connected to tube. This model is scalable and can be adapted to any size of bar/tube (see `marimbaBarModel` to know more about the limitations of this type of system).

Usage

```
marimba_ui_MIDI : _
```

Test

```
pm = library("physmodels.lib");
marimba_ui_MIDI_test = pm.marimba_ui_MIDI;
```

(pm.)churchBellModel

Generic church bell modal model generated by `mesh2faust` from `libraries/modalmodels/churchBell`.

Modeled after T. Rossing and R. Perrin, Vibrations of Bells, Applied Acoustics 2, 1987.

Model height is 301 mm.

This model contains 7 excitation positions going linearly from the bottom to the top of the bell. Obviously, a model with more excitation position could be regenerated using `mesh2faust`.

Usage

```
excitation : churchBellModel(nModes,exPos,t60,t60DecayRatio,t60DecaySlope)
```

Where:

- `excitation`: the excitation signal
- `nModes`: number of synthesized modes (max: 50)
- `exPos`: excitation position (0-6)
- `t60`: T60 in seconds (recommended value: 0.1)
- `t60DecayRatio`: T60 decay ratio (recommended value: 1)
- `t60DecaySlope`: T60 decay slope (recommended value: 5)

Test

```
pm = library("physmodels.lib");
churchBellModel_test = pm.churchBellModel(110);
```

(pm.)churchBell

Generic church bell modal model.

Modeled after T. Rossing and R. Perrin, Vibrations of Bells, Applied Acoustics 2, 1987.

Model height is 301 mm.

This model contains 7 excitation positions going linearly from the bottom to the top of the bell. Obviously, a model with more excitation position could be regenerated using `mesh2faust`.

This function also implement a virtual exciter to drive the model.

Usage

```
churchBell(strikePosition,strikeCutoff,strikeSharpness,gain,trigger) : _
```

Where:

- `strikePosition`: strike position (0-6)
- `strikeCutoff`: cuttoff frequency of the strike genarator (recommended: ~7000Hz)
- `strikeSharpness`: sharpness of the strike (recommended: ~0.25)
- `gain`: gain of the strike (0-1)
- `trigger` signal (0: off, 1: on)

Test

```
pm = library("physmodels.lib");
churchBell_test = pm.churchBell(0.4, 2000, 0.5, 0.8, button("gate"));
```

(pm.)churchBell_ui

Church bell physical model based on `churchBell` with built-in UI.

Usage

```
churchBell_ui : _
```

Test

```
pm = library("physmodels.lib");
churchBell_ui_test = pm.churchBell_ui;
```

(pm.)englishBellModel

English church bell modal model generated by `mesh2faust` from `libraries/modalmodels/englishBell`.

Modeled after D.Bartocha and Baron, Influence of Tin Bronze Melting and Pouring Parameters on Its Properties and Bell' Tone, Archives of Foundry Engineering, 2016.

Model height is 1 m.

This model contains 7 excitation positions going linearly from the bottom to the top of the bell. Obviously, a model with more excitation position could be regenerated using `mesh2faust`.

Usage

```
excitation : englishBellModel(nModes,exPos,t60,t60DecayRatio,t60DecaySlope)
```

Where:

- `excitation`: the excitation signal
- `nModes`: number of synthesized modes (max: 50)
- `exPos`: excitation position (0-6)
- `t60`: T60 in seconds (recommended value: 0.1)
- `t60DecayRatio`: T60 decay ratio (recommended value: 1)
- `t60DecaySlope`: T60 decay slope (recommended value: 5)

Test

```
pm = library("physmodels.lib");
englishBellModel_test = pm.englishBellModel(110);
```

(pm.)englishBell

English church bell modal model.

Modeled after D.Bartocha and Baron, Influence of Tin Bronze Melting and Pouring Parameters on Its Properties and Bell' Tone, Archives of Foundry Engineering, 2016.

Model height is 1 m.

This model contains 7 excitation positions going linearly from the bottom to the top of the bell. Obviously, a model with more excitation position could be regenerated using `mesh2faust`.

This function also implement a virtual exciter to drive the model.

Usage

```
englishBell(strikePosition,strikeCutoff,strikeSharpness,gain,trigger) : _
```

Where:

- `strikePosition`: strike position (0-6)
- `strikeCutoff`: cuttoff frequency of the strike genarator (recommended: ~7000Hz)
- `strikeSharpness`: sharpness of the strike (recommended: ~0.25)
- `gain`: gain of the strike (0-1)
- `trigger` signal (0: off, 1: on)

Test

```
pm = library("physmodels.lib");
englishBell_test = pm.englishBell(0.4, 2000, 0.5, 0.8, button("gate"));
```

(pm.)englishBell_ui

English church bell physical model based on `englishBell` with built-in UI.

Usage

```
englishBell_ui : _
```

Test

```
pm = library("physmodels.lib");
englishBell_ui_test = pm.englishBell_ui;
```

(pm.)frenchBellModel

French church bell modal model generated by `mesh2faust` from `libraries/modalmodels/frenchBell`.

Modeled after D.Bartocha and Baron, Influence of Tin Bronze Melting and Pouring Parameters on Its Properties and Bell' Tone, Archives of Foundry Engineering, 2016.

Model height is 1 m.

This model contains 7 excitation positions going linearly from the bottom to the top of the bell. Obviously, a model with more excitation position could be regenerated using `mesh2faust`.

Usage

```
excitation : frenchBellModel(nModes,exPos,t60,t60DecayRatio,t60DecaySlope)
```

Where:

- `excitation`: the excitation signal
- `nModes`: number of synthesized modes (max: 50)
- `exPos`: excitation position (0-6)
- `t60`: T60 in seconds (recommended value: 0.1)
- `t60DecayRatio`: T60 decay ratio (recommended value: 1)
- `t60DecaySlope`: T60 decay slope (recommended value: 5)

Test

```
pm = library("physmodels.lib");
frenchBellModel_test = pm.frenchBellModel(110);
```

(pm.)frenchBell

French church bell modal model.

Modeled after D.Bartocha and Baron, Influence of Tin Bronze Melting and Pouring Parameters on Its Properties and Bell' Tone, Archives of Foundry Engineering, 2016.

Model height is 1 m.

This model contains 7 excitation positions going linearly from the bottom to the top of the bell. Obviously, a model with more excitation position could be regenerated using `mesh2faust`.

This function also implement a virtual exciter to drive the model.

Usage

Where:

- `strikePosition`: strike position (0-6)
- `strikeCutoff`: cuttoff frequency of the strike genarator (recommended: ~7000Hz)
- `strikeSharpness`: sharpness of the strike (recommended: ~0.25)
- `gain`: gain of the strike (0-1)
- `trigger` signal (0: off, 1: on)

Test

```
pm = library("physmodels.lib");
frenchBell_test = pm.frenchBell(0.4, 2000, 0.5, 0.8, button("gate"));
```

`(pm.)frenchBell_ui`

French church bell physical model based on `frenchBell` with built-in UI.

Usage

```
frenchBell_ui : _
```

Test

```
pm = library("physmodels.lib");
frenchBell_ui_test = pm.frenchBell_ui;
```

`(pm.)germanBellModel`

German church bell modal model generated by `mesh2faust` from `libraries/modalmodels/germanBell`.

Modeled after D.Bartocha and Baron, Influence of Tin Bronze Melting and Pouring Parameters on Its Properties and Bell' Tone, Archives of Foundry Engineering, 2016.

Model height is 1 m.

This model contains 7 excitation positions going linearly from the bottom to the top of the bell. Obviously, a model with more excitation position could be regenerated using `mesh2faust`.

Usage

```
excitation : germanBellModel(nModes,exPos,t60,t60DecayRatio,t60DecaySlope)
```

Where:

- `excitation`: the excitation signal
- `nModes`: number of synthesized modes (max: 50)
- `exPos`: excitation position (0-6)
- `t60`: T60 in seconds (recommended value: 0.1)
- `t60DecayRatio`: T60 decay ratio (recommended value: 1)
- `t60DecaySlope`: T60 decay slope (recommended value: 5)

Test

```
pm = library("physmodels.lib");
germanBellModel_test = pm.germanBellModel(110);
```

`(pm.)germanBell`

German church bell modal model.

Modeled after D.Bartocha and Baron, Influence of Tin Bronze Melting and Pouring Parameters on Its Properties and Bell' Tone, Archives of Foundry Engineering, 2016.

Model height is 1 m.

This model contains 7 excitation positions going linearly from the bottom to the top of the bell. Obviously, a model with more excitation position could be regenerated using `mesh2faust`.

This function also implement a virtual exciter to drive the model.

Usage

```
germanBell(strikePosition,strikeCutoff,strikeSharpness,gain,trigger) : _
```

Where:

- `strikePosition`: strike position (0-6)
- `strikeCutoff`: cuttoff frequency of the strike genarator (recommended: ~7000Hz)
- `strikeSharpness`: sharpness of the strike (recommended: ~0.25)
- `gain`: gain of the strike (0-1)
- `trigger` signal (0: off, 1: on)

Test

```
pm = library("physmodels.lib");
germanBell_test = pm.germanBell(0.4, 2000, 0.5, 0.8, button("gate"));
```

(pm.)germanBell_ui

German church bell physical model based on `germanBell` with built-in UI.

Usage

```
germanBell_ui : _
```

Test

```
pm = library("physmodels.lib");
germanBell_ui_test = pm.germanBell_ui;
```

(pm.)russianBellModel

Russian church bell modal model generated by `mesh2faust` from `libraries/modalmodels/russianBell`.

Modeled after D.Bartocha and Baron, Influence of Tin Bronze Melting and Pouring Parameters on Its Properties and Bell' Tone, Archives of Foundry Engineering, 2016.

Model height is 2 m.

This model contains 7 excitation positions going linearly from the bottom to the top of the bell. Obviously, a model with more excitation position could be regenerated using `mesh2faust`.

Usage

```
excitation : russianBellModel(nModes,exPos,t60,t60DecayRatio,t60DecaySlope)
```

Where:

- `excitation`: the excitation signal
- `nModes`: number of synthesized modes (max: 50)
- `exPos`: excitation position (0-6)
- `t60`: T60 in seconds (recommended value: 0.1)
- `t60DecayRatio`: T60 decay ratio (recommended value: 1)
- `t60DecaySlope`: T60 decay slope (recommended value: 5)

Test

```
pm = library("physmodels.lib");
russianBellModel_test = pm.russianBellModel(110);
```

(pm.)russianBell

Russian church bell modal model.

Modeled after D.Bartocha and Baron, Influence of Tin Bronze Melting and Pouring Parameters on Its Properties and Bell' Tone, Archives of Foundry Engineering, 2016.

Model height is 2 m.

This model contains 7 excitation positions going linearly from the bottom to the top of the bell. Obviously, a model with more excitation position could be regenerated using `mesh2faust`.

This function also implement a virtual exciter to drive the model.

Usage

```
russianBell(strikePosition,strikeCutoff,strikeSharpness,gain,trigger) : _
```

Where:

- `strikePosition`: strike position (0-6)
- `strikeCutoff`: cuttoff frequency of the strike genarator (recommended: ~7000Hz)
- `strikeSharpness`: sharpness of the strike (recommended: ~0.25)
- `gain`: gain of the strike (0-1)
- `trigger` signal (0: off, 1: on)

Test

```
pm = library("physmodels.lib");
russianBell_test = pm.russianBell(0.4, 2000, 0.5, 0.8, button("gate"));
```

(pm.)russianBell_ui

Russian church bell physical model based on `russianBell` with built-in UI.

Usage

```
russianBell_ui : _
```

Test

```
pm = library("physmodels.lib");
russianBell_ui_test = pm.russianBell_ui;
```

(pm.)standardBellModel

Standard church bell modal model generated by `mesh2faust` from `libraries/modalmodels/standardBell`.

Modeled after T. Rossing and R. Perrin, Vibrations of Bells, Applied Acoustics 2, 1987.

Model height is 1.8 m.

This model contains 7 excitation positions going linearly from the bottom to the top of the bell. Obviously, a model with more excitation position could be regenerated using `mesh2faust`.

Usage

```
excitation : standardBellModel(nModes,exPos,t60,t60DecayRatio,t60DecaySlope)
```

Where:

- `excitation`: the excitation signal
- `nModes`: number of synthesized modes (max: 50)
- `exPos`: excitation position (0-6)
- `t60`: T60 in seconds (recommended value: 0.1)
- `t60DecayRatio`: T60 decay ratio (recommended value: 1)
- `t60DecaySlope`: T60 decay slope (recommended value: 5)

Test

```
pm = library("physmodels.lib");
standardBellModel_test = pm.standardBellModel(110);
```

(pm.)standardBell

Standard church bell modal model.

Modeled after T. Rossing and R. Perrin, Vibrations of Bells, Applied Acoustics 2, 1987.

Model height is 1.8 m.

This model contains 7 excitation positions going linearly from the bottom to the top of the bell. Obviously, a model with more excitation position could be regenerated using `mesh2faust`.

This function also implement a virtual exciter to drive the model.

Usage

```
standardBell(strikePosition,strikeCutoff,strikeSharpness,gain,trigger) : _
```

Where:

- `strikePosition`: strike position (0-6)
- `strikeCutoff`: cuttoff frequency of the strike genarator (recommended: ~7000Hz)
- `strikeSharpness`: sharpness of the strike (recommended: ~0.25)
- `gain`: gain of the strike (0-1)
- `trigger` signal (0: off, 1: on)

Test

```
pm = library("physmodels.lib");
standardBell_test = pm.standardBell(0.4, 2000, 0.5, 0.8, button("gate"));
```

`(pm.)standardBell_ui`

Standard church bell physical model based on `standardBell` with built-in UI.

Usage

```
standardBell_ui : _
```

Test

```
pm = library("physmodels.lib");
standardBell_ui_test = pm.standardBell_ui;
```

Vocal Synthesis

Vocal synthesizer functions (source/filter, fof, etc.).

`(pm.)formantValues`

Formant data values in an environment.

The formant data used here come from the CSOUND manual * http://www.csounds.com/manual/html/.

Usage

```
ba.take(j+1,formantValues.f(i)) : _
ba.take(j+1,formantValues.g(i)) : _
ba.take(j+1,formantValues.bw(i)) : _
```

Where:

- `i`: formant number
- `j`: (voiceType*nFormants)+vowel
- `voiceType`: the voice type (0: alto, 1: bass, 2: countertenor, 3: soprano, 4: tenor)
- `vowel`: the vowel (0: a, 1: e, 2: i, 3: o, 4: u)

Test

```
pm = library("physmodels.lib");
formantValues_test = pm.formantValues.f(0);
```

`(pm.)voiceGender`

Calculate the gender for the provided `voiceType` value. (0: male, 1: female)

Usage

```
voiceGender(voiceType) : _
```

Where:

- `voiceType`: the voice type (0: alto, 1: bass, 2: countertenor, 3: soprano, 4: tenor)

Test

```
pm = library("physmodels.lib");
voiceGender_test = pm.voiceGender(0.5);
```

`(pm.)skirtWidthMultiplier`

Calculates value to multiply bandwidth to obtain `skirtwidth` for a Fof filter.

Usage

```
skirtWidthMultiplier(vowel,freq,gender) : _
```

Where:

- `vowel`: the vowel (0: a, 1: e, 2: i, 3: o, 4: u)

- `freq`: the fundamental frequency of the excitation signal
- `gender`: gender of the voice used in the fof filter (0: male, 1: female)

Test

```
pm = library("physmodels.lib");
skirtWidthMultiplier_test = pm.skirtWidthMultiplier(0.5);
```

`(pm.)autobendFreq`

Autobends the center frequencies of formants 1 and 2 based on the fundamental frequency of the excitation signal and leaves all other formant frequencies unchanged. Ported from `chant-lib`.

Usage

```
_ : autobendFreq(n,freq,voiceType) : _
```

Where:

- `n`: formant index
- `freq`: the fundamental frequency of the excitation signal
- `voiceType`: the voice type (0: alto, 1: bass, 2: countertenor, 3: soprano, 4: tenor)
- input is the center frequency of the corresponding formant

Test

```
pm = library("physmodels.lib");
autobendFreq_test = pm.autobendFreq(440, 0.5);
```

References

- https://ccrma.stanford.edu/~rmichon/chantLib/.

`(pm.)vocalEffort`

Changes the gains of the formants based on the fundamental frequency of the excitation signal. Higher formants are reinforced for higher fundamental frequencies. Ported from `chant-lib`.

Usage

```
_ : vocalEffort(freq,gender) : _
```

Where:

- `freq`: the fundamental frequency of the excitation signal
- `gender`: the gender of the voice type (0: male, 1: female)
- input is the linear amplitude of the formant

Test

```
pm = library("physmodels.lib");
vocalEffort_test = pm.vocalEffort(0.6);
```

References

- https://ccrma.stanford.edu/~rmichon/chantLib/.

(pm.)fof

Function to generate a single Formant-Wave-Function.

Usage

```
_ : fof(fc,bw,a,g) : _
```

Where:

- `fc`: formant center frequency,
- `bw`: formant bandwidth (Hz),
- `sw`: formant skirtwidth (Hz)
- `g`: linear scale factor (g=1 gives 0dB amplitude response at fc)
- input is an impulse signal to excite filter

Test

```
pm = library("physmodels.lib");
fof_test = pm.fof(0.3, 440, 880, 0.5);
```

References

- https://ccrma.stanford.edu/~mjolsen/pdfs/smc2016_MOlsenFOF.pdf.

(pm.)fofSH

FOF with sample and hold used on `bw` and a parameter used in the filter-cycling FOF function `fofCycle`.

Usage

```
_ : fofSH(fc,bw,a,g) : _
```

Where: all parameters same as for `fof`

Test

```
pm = library("physmodels.lib");
fofSH_test = pm.fofSH(0.3, 440, 880, 0.5);
```

References

- https://ccrma.stanford.edu/~mjolsen/pdfs/smc2016_MOlsenFOF.pdf.

(pm.)fofCycle

FOF implementation where time-varying filter parameter noise is mitigated by using a cycle of `n` sample and hold FOF filters.

Usage

```
_ : fofCycle(fc,bw,a,g,n) : _
```

Where:

- `n`: the number of FOF filters to cycle through
- all other parameters are same as for `fof`

Test

```
pm = library("physmodels.lib");
fofCycle_test = pm.fofCycle(0.3, 440, 880, 0.5, 0.2);
```

References

- https://ccrma.stanford.edu/~mjolsen/pdfs/smc2016_MOlsenFOF.pdf.

(pm.)fofSmooth

FOF implementation where time-varying filter parameter noise is mitigated by lowpass filtering the filter parameters `bw` and `a` with smooth.

Usage

```
_ : fofSmooth(fc,bw,sw,g,tau) : _
```

Where:

- `tau`: the desired smoothing time constant in seconds
- all other parameters are same as for `fof`

Test

```
pm = library("physmodels.lib");
fofSmooth_test = pm.fofSmooth(0.3, 440, 880, 0.5, 0.2);
```

(pm.)formantFilterFofCycle

Formant filter based on a single FOF filter. Formant parameters are linearly interpolated allowing to go smoothly from one vowel to another. A cycle of `n` fof filters with sample-and-hold is used so that the fof filter parameters can be varied in realtime. This technique is more robust but more computationally expensive than `formantFilterFofSmooth`.Voice type can be selected but must correspond to the frequency range of the provided source to be realistic.

Usage

```
_ : formantFilterFofCycle(voiceType,vowel,nFormants,i,freq) : _
```

Where:

- `voiceType`: the voice type (0: alto, 1: bass, 2: countertenor, 3: soprano, 4: tenor)
- `vowel`: the vowel (0: a, 1: e, 2: i, 3: o, 4: u)
- `nFormants`: number of formant regions in frequency domain, typically 5
- `i`: formant number (i.e. 0 - 4) used to index formant data value arrays
- `freq`: fundamental frequency of excitation signal. Used to calculate rise time of envelope

Test

```
pm = library("physmodels.lib");
formantFilterFofCycle_test = pm.formantFilterFofCycle(0, 0, 5, 0, 200);
```

(pm.)formantFilterFofSmooth

Formant filter based on a single FOF filter. Formant parameters are linearly interpolated allowing to go smoothly from one vowel to another. Fof filter parameters are lowpass filtered to mitigate possible noise from varying them in realtime. Voice type can be selected but must correspond to the frequency range of the provided source to be realistic.

Usage

```
_ : formantFilterFofSmooth(voiceType,vowel,nFormants,i,freq) : _
```

Where:

- `voiceType`: the voice type (0: alto, 1: bass, 2: countertenor, 3: soprano, 4: tenor)
- `vowel`: the vowel (0: a, 1: e, 2: i, 3: o, 4: u)
- `nFormants`: number of formant regions in frequency domain, typically 5
- `i`: formant number (i.e. 1 - 5) used to index formant data value arrays
- `freq`: fundamental frequency of excitation signal. Used to calculate rise time of envelope

Test

```
pm = library("physmodels.lib");
formantFilterFofSmooth_test = pm.formantFilterFofSmooth(0, 0, 5, 0, 200);
```

`(pm.)formantFilterBP`

Formant filter based on a single resonant bandpass filter. Formant parameters are linearly interpolated allowing to go smoothly from one vowel to another. Voice type can be selected but must correspond to the frequency range of the provided source to be realistic.

Usage

```
_ : formantFilterBP(voiceType,vowel,nFormants,i,freq) : _
```

Where:

- `voiceType`: the voice type (0: alto, 1: bass, 2: countertenor, 3: soprano, 4: tenor)
- `vowel`: the vowel (0: a, 1: e, 2: i, 3: o, 4: u)
- `nFormants`: number of formant regions in frequency domain, typically 5
- `i`: formant index used to index formant data value arrays
- `freq`: fundamental frequency of excitation signal.

Test

```
pm = library("physmodels.lib");
formantFilterBP_test = pm.formantFilterBP(0, 0, 5, 0, 200);
```

(pm.)formantFilterbank

Formant filterbank which can use different types of filterbank functions and different excitation signals. Formant parameters are linearly interpolated allowing to go smoothly from one vowel to another. Voice type can be selected but must correspond to the frequency range of the provided source to be realistic.

Usage

```
_ : formantFilterbank(voiceType,vowel,formantGen,freq) : _
```

Where:

- `voiceType`: the voice type (0: alto, 1: bass, 2: countertenor, 3: soprano, 4: tenor)
- `vowel`: the vowel (0: a, 1: e, 2: i, 3: o, 4: u)
- `formantGen`: the specific formant filterbank function (i.e. FormantFilterbankBP, FormantFilterbankFof,…)
- `freq`: fundamental frequency of excitation signal. Needed for FOF version to calculate rise time of envelope

Test

```
pm = library("physmodels.lib");
formantFilterbank_test = pm.formantFilterbank(0, 0, 5, 0);
```

(pm.)formantFilterbankFofCycle

Formant filterbank based on a bank of fof filters. Formant parameters are linearly interpolated allowing to go smoothly from one vowel to another. Voice type can be selected but must correspond to the frequency range of the provided source to be realistic.

Usage

```
_ : formantFilterbankFofCycle(voiceType,vowel,freq) : _
```

Where:

- `voiceType`: the voice type (0: alto, 1: bass, 2: countertenor, 3: soprano, 4: tenor)
- `vowel`: the vowel (0: a, 1: e, 2: i, 3: o, 4: u)
- `freq`: the fundamental frequency of the excitation signal. Needed to calculate the skirtwidth of the FOF envelopes and for the autobendFreq and vocalEffort functions

Test

```
pm = library("physmodels.lib");
formantFilterbankFofCycle_test = pm.formantFilterbankFofCycle(0, 0, 5));
```

(pm.)formantFilterbankFofSmooth

Formant filterbank based on a bank of fof filters. Formant parameters are linearly interpolated allowing to go smoothly from one vowel to another. Voice type can be selected but must correspond to the frequency range of the provided source to be realistic.

Usage

```
_ : formantFilterbankFofSmooth(voiceType,vowel,freq) : _
```

Where:

- `voiceType`: the voice type (0: alto, 1: bass, 2: countertenor, 3: soprano, 4: tenor)
- `vowel`: the vowel (0: a, 1: e, 2: i, 3: o, 4: u)
- `freq`: the fundamental frequency of the excitation signal. Needed to calculate the skirtwidth of the FOF envelopes and for the autobendFreq and vocalEffort functions

Test

```
pm = library("physmodels.lib");
formantFilterbankFofSmooth_test = pm.formantFilterbankFofSmooth(0, 0, 5);
```

(pm.)formantFilterbankBP

Formant filterbank based on a bank of resonant bandpass filters. Formant parameters are linearly interpolated allowing to go smoothly from one vowel to another. Voice type can be selected but must correspond to the frequency range of the provided source to be realistic.

Usage

```
_ : formantFilterbankBP(voiceType,vowel,freq) : _
```

Where:

- `voiceType`: the voice type (0: alto, 1: bass, 2: countertenor, 3: soprano, 4: tenor)
- `vowel`: the vowel (0: a, 1: e, 2: i, 3: o, 4: u)

- `freq`: the fundamental frequency of the excitation signal. Needed for the autobendFreq and vocalEffort functions.

Test

```
pm = library("physmodels.lib");
formantFilterbankBP_test = pm.formantFilterbankBP(0, 0, 5);
```

`(pm.)SFFormantModel`

Simple formant/vocal synthesizer based on a source/filter model. The `source` and `filterbank` must be specified by the user. `filterbank` must take the same input parameters as `formantFilterbank` (BP/`FofCycle` /`FofSmooth`). Formant parameters are linearly interpolated allowing to go smoothly from one vowel to another. Voice type can be selected but must correspond to the frequency range of the synthesized voice to be realistic.

Usage

```
SFFormantModel(voiceType,vowel,exType,freq,gain,source,filterbank,isFof) : _
```

Where:

- `voiceType`: the voice type (0: alto, 1: bass, 2: countertenor, 3: soprano, 4: tenor)
- `vowel`: the vowel (0: a, 1: e, 2: i, 3: o, 4: u
- `exType`: voice vs. fricative sound ratio (0-1 where 1 is 100% fricative)
- `freq`: the fundamental frequency of the source signal
- `gain`: linear gain multiplier to multiply the source by
- `isFof`: whether model is FOF based (0: no, 1: yes)

Test

```
pm = library("physmodels.lib");
SFFormantModel_test = pm.SFFormantModel(0, 0, 0.5, 0.6, 100, 2, 1, 1);
```

`(pm.)SFFormantModelFofCycle`

Simple formant/vocal synthesizer based on a source/filter model. The source is just a periodic impulse and the “filter” is a bank of FOF filters. Formant parameters are linearly interpolated allowing to go smoothly from one vowel to another. Voice type can be selected but must correspond to the frequency range of the synthesized voice to be realistic. This model does not work with noise in the source signal so exType has been removed and model does not depend on SFFormantModel function.

Usage

```
SFFormantModelFofCycle(voiceType,vowel,freq,gain) : _
```

Where:

- `voiceType`: the voice type (0: alto, 1: bass, 2: countertenor, 3: soprano, 4: tenor)
- `vowel`: the vowel (0: a, 1: e, 2: i, 3: o, 4: u
- `freq`: the fundamental frequency of the source signal
- `gain`: linear gain multiplier to multiply the source by

Test

```
pm = library("physmodels.lib");
SFFormantModelFofCycle_test = pm.SFFormantModelFofCycle(0.5, 0.6, 0.7);
```

`(pm.)SFFormantModelFofSmooth`

Simple formant/vocal synthesizer based on a source/filter model. The source is just a periodic impulse and the "filter" is a bank of FOF filters. Formant parameters are linearly interpolated allowing to go smoothly from one vowel to another. Voice type can be selected but must correspond to the frequency range of the synthesized voice to be realistic.

Usage

```
SFFormantModelFofSmooth(voiceType,vowel,freq,gain) : _
```

Where:

- `voiceType`: the voice type (0: alto, 1: bass, 2: countertenor, 3: soprano, 4: tenor)
- `vowel`: the vowel (0: a, 1: e, 2: i, 3: o, 4: u
- `freq`: the fundamental frequency of the source signal
- `gain`: linear gain multiplier to multiply the source by

Test

```
pm = library("physmodels.lib");
SFFormantModelFofSmooth_test = pm.SFFormantModelFofSmooth(0.5, 0.6, 0.7);
```

`(pm.)SFFormantModelBP`

Simple formant/vocal synthesizer based on a source/filter model. The source is just a sawtooth wave and the "filter" is a bank of resonant bandpass filters. Formant parameters are linearly interpolated allowing to go smoothly from one vowel to another. Voice type can be selected but must correspond to the frequency range of the synthesized voice to be realistic.

The formant data used here come from the CSOUND manual * http://www.csounds.com/manual/html/.

Usage

```
SFFormantModelBP(voiceType,vowel,exType,freq,gain) : _
```

Where:

- `voiceType`: the voice type (0: alto, 1: bass, 2: countertenor, 3: soprano, 4: tenor)
- `vowel`: the vowel (0: a, 1: e, 2: i, 3: o, 4: u
- `exType`: voice vs. fricative sound ratio (0-1 where 1 is 100% fricative)
- `freq`: the fundamental frequency of the source signal
- `gain`: linear gain multiplier to multiply the source by

Test

```
pm = library("physmodels.lib");
SFFormantModelBP_test = pm.SFFormantModelBP(0.5, 0.6, 0.7);
```

`(pm.)SFFormantModelFofCycle_ui`

Ready-to-use source-filter vocal synthesizer with built-in user interface.

Usage

```
SFFormantModelFofCycle_ui : _
```

Test

```
pm = library("physmodels.lib");
SFFormantModelFofCycle_ui_test = pm.SFFormantModelFofCycle_ui;
```

`(pm.)SFFormantModelFofSmooth_ui`

Ready-to-use source-filter vocal synthesizer with built-in user interface.

Usage

```
SFFormantModelFofSmooth_ui : _
```

Test

```
pm = library("physmodels.lib");
SFFormantModelFofSmooth_ui_test = pm.SFFormantModelFofSmooth_ui;
```

(pm.)SFFormantModelBP_ui

Ready-to-use source-filter vocal synthesizer with built-in user interface.

Usage

```
SFFormantModelBP_ui : _
```

Test

```
pm = library("physmodels.lib");
SFFormantModelBP_ui_test = pm.SFFormantModelBP_ui;
```

(pm.)SFFormantModelFofCycle_ui_MIDI

Ready-to-use MIDI-controllable source-filter vocal synthesizer.

Usage

```
SFFormantModelFofCycle_ui_MIDI : _
```

Test

```
pm = library("physmodels.lib");
SFFormantModelFofCycle_ui_MIDI_test = pm.SFFormantModelFofCycle_ui_MIDI;
```

(pm.)SFFormantModelFofSmooth_ui_MIDI

Ready-to-use MIDI-controllable source-filter vocal synthesizer.

Usage

```
SFFormantModelFofSmooth_ui_MIDI : _
```

Test

```
pm = library("physmodels.lib");
SFFormantModelFofSmooth_ui_MIDI_test = pm.SFFormantModelFofSmooth_ui_MIDI;
```

`(pm.)SFFormantModelBP_ui_MIDI`

Ready-to-use MIDI-controllable source-filter vocal synthesizer.

Usage

```
SFFormantModelBP_ui_MIDI : _
```

Test

```
pm = library("physmodels.lib");
SFFormantModelBP_ui_MIDI_test = pm.SFFormantModelBP_ui_MIDI;
```

Misc Functions

Various miscellaneous functions.

`(pm.)allpassNL`

Bidirectional block adding nonlinearities in both directions in a chain. Nonlinearities are created by modulating the coefficients of a passive allpass filter by the signal it is processing.

Usage

```
chain(... : allpassNL(nonlinearity) : ...)
```

Where:

- `nonlinearity`: amount of nonlinearity to be added (0-1)

Test

```
pm = library("physmodels.lib");
allpassNL_test = 0,0,0 : pm.allpassNL(0.4);
```

`(pm.)modalModel`

Implement multiple resonance modes using resonant bandpass filters.

Usage

```
_ : modalModel(n, freqs, t60s, gains) : _
```

Where:

- `n`: number of given modes

- `freqs` : list of filter center freqencies
- `t60s` : list of mode resonance durations (in seconds)
- `gains` : list of mode gains (0-1)

For example, to generate a model with 2 modes (440 Hz and 660 Hz, a fifth) where the higher one decays faster and is attenuated:

```
os.impulse : modalModel(2, (440, 660),
                           (0.5, 0.25),
                           (ba.db2linear(-1), ba.db2linear(-6)) : _
```

Test

```
pm = library("physmodels.lib");
os = library("oscillators.lib");
modalModel_test = os.impulse : pm.modalModel(3, (440,660,880), (0.5,0.4,0.3), (0.8,0.6,0.4)
```

Further reading: Grumiaux et. al., 2017: Impulse-Response and CAD-Model-Based Physical Modeling in Faust

---

`(pm.)rk_solve`

Solves the system of ordinary differential equations of any order using the explicit Runge-Kutta methods.

Usage

```
rk_solve(ts,ks, ni,h, eq,iv) : si.bus(outputs(eq))
```

Where:

- `ts,ks` : the Butcher tableau (see below)
- `ni` : number of iterations at each tick, compile time constant ni > 1 can improve accuracy but will degrade performance
- `h` : time step, run time constant, e.g. 1/ma.SR
- `eq` : list of derivative functions
- `iv` : list of initial values

`rk_solve()` with the "standard" 1-4 tableaux and ni = 1:

```
rk_solve_1 = rk_solve((0), (1), 1);
rk_solve_2 = rk_solve((0,1/2), (1/2, 0,1), 1);
rk_solve_3 = rk_solve((0,1/2,1), (1/2,-1,2, 1/6,2/3,1/6), 1);
rk_solve_4 = rk_solve((0,1/2,1/2,1), (1/2,0,1/2,0,0,1, 1/6,1/3,1/3,1/6), 1);
```

Test

```
pm = library("physmodels.lib");
ma = library("maths.lib");
```

```
rk_solve_test = pm.rk_solve((0), (1), 1, 1.0/ma.SR, eq, (1)) with { eq(t,x) = -x; };
```

Example test program Suppose we have a system of differential equations:

```
dx/dt = dx_dt(t,x,y,z)
dy/dt = dy_dt(t,x,y,z)
dz/dt = dz_dt(t,x,y,z)
```

with initial conditions:

```
   x(0) = x0
   y(0) = y0
   z(0) = z0
```

and we want to solve it using this Butcher tableau:

```
 0 |
c2 | a21
c3 | a31 a32
c4 | a41 a42 a43
-------------------
   | b1  b1  b3  b4
```

```
EQ(t,x,y,z) = dx_dt(t,x,y,z),
              dy_dt(t,x,y,z),
              dz_dt(t,x,y,z);

IV = x0, y0, z0;

TS = 0, c2, c3, c4;
KS = a21,
     a31, a32,
     a41, a42, a43,
     b1,  b2,  b3,  b4;

process = rk_solve(TS,KS, 1,1/ma.SR, EQ,IV);
```

Less abstract example which can actually be compiled/tested:

```
// Lotka-Volterra equations parameterized by a,b,c,d:
LV(a,b,c,d, t,x,y) =
    a*x - b*x*y,
    c*x*y - d*y;

// Solved using the "standard" fourth-order method:
process = rk_solve_4(
    0.01,                  // time step
    LV(0.1,0.02,0.03,0.4), // LV() with random parameters
    (3,4)                  // initial values
```

```
);
```

References

- https://wikipedia.org/wiki/Runge%E2%80%93Kutta_methods

quantizers.lib

Quantizers library. Its official prefix is `qu`.

This library provides utilities for pitch and signal quantization in Faust. It includes functions for mapping continuous inputs to discrete musical scales.

The Quantizers library is organized into 1 section:

- Functions Reference

References

- https://github.com/grame-cncm/faustlibraries/blob/master/quantizers.lib

Functions Reference

`(qu.)quantize`

Configurable frequency quantization tool. Snaps input frequencies to exact scale notes. Works for positive audio frequencies.

Usage

```
_ : quantize(rf,nl) : _
```

Where:

- `rf` : frequency of the root note of the scale
- `nl` : list of frequency ratios for each note relative to root

Test

```
qu = library("quantizers.lib");
quantize_test = qu.quantize(440, qu.ionian, hslider("input", 450, 100, 1000, 1));
```

Example

```
process = quantize(440, (1, 1.125, 1.25, 1.333, 1.5));
```

`(qu.)quantizeSmoothed`

Configurable frequency quantization tool. Smoothly transitions between scale notes. Works for positive audio frequencies.

Usage

```
_ : quantizeSmoothed(rf,nl) : _
```

Where:

- `rf` : frequency of the root note of the scale
- `nl` : list of frequency ratios for each note relative to root

Test

```
qu = library("quantizers.lib");
quantizeSmoothed_test = qu.quantizeSmoothed(440, qu.ionian, hslider("input", 450, 100, 1000,
```

Example

```
process = quantizeSmoothed(440, dodeca);
```

`(qu.)ionian`

List of the frequency ratios of the notes of the ionian mode.

Usage

```
_ : quantize(rf,ionian) : _
```

Where:

- `rf`: frequency of the root note of the scale

Test

```
qu = library("quantizers.lib");
ionian_test = qu.quantize(220, qu.ionian, 260);
```

`(qu.)dorian`

List of the frequency ratios of the notes of the dorian mode.

Usage

```
_ : quantize(rf,dorian) : _
```

Where:

- `rf`: frequency of the root note of the scale

Test

```
qu = library("quantizers.lib");
dorian_test = qu.quantize(220, qu.dorian, 260);
```

(qu.)phrygian

List of the frequency ratios of the notes of the phrygian mode.

Usage

```
_ : quantize(rf,phrygian) : _
```

Where:

- `rf`: frequency of the root note of the scale

Test

```
qu = library("quantizers.lib");
phrygian_test = qu.quantize(220, qu.phrygian, 260);
```

(qu.)lydian

List of the frequency ratios of the notes of the lydian mode.

Usage

```
_ : quantize(rf,lydian) : _
```

Where:

- `rf`: frequency of the root note of the scale

Test

```
qu = library("quantizers.lib");
lydian_test = qu.quantize(220, qu.lydian, 260);
```

(qu.)mixo

List of the frequency ratios of the notes of the mixolydian mode.

Usage

```
_ : quantize(rf,mixo) : _
```

Where:

- `rf`: frequency of the root note of the scale

Test

```
qu = library("quantizers.lib");
mixo_test = qu.quantize(220, qu.mixo, 260);
```

(qu.)eolian

List of the frequency ratios of the notes of the eolian mode.

Usage

```
_ : quantize(rf,eolian) : _
```

Where:

- `rf`: frequency of the root note of the scale

Test

```
qu = library("quantizers.lib");
eolian_test = qu.quantize(220, qu.eolian, 260);
```

(qu.)locrian

List of the frequency ratios of the notes of the locrian mode.

Usage

```
_ : quantize(rf,locrian) : _
```

Where:

- `rf`: frequency of the root note of the scale

Test

```
qu = library("quantizers.lib");
locrian_test = qu.quantize(220, qu.locrian, 260);
```

(qu.)pentanat

List of the frequency ratios of the notes of the pythagorean tuning for the minor pentatonic scale.

Usage

```
_ : quantize(rf,pentanat) : _
```

Where:

- `rf`: frequency of the root note of the scale

Test

```
qu = library("quantizers.lib");
pentanat_test = qu.quantize(220, qu.pentanat, 260);
```

(qu.)kumoi

List of the frequency ratios of the notes of the kumoijoshi, the japanese pentatonic scale.

Usage

```
_ : quantize(rf,kumoi) : _
```

Where:

- `rf`: frequency of the root note of the scale

Test

```
qu = library("quantizers.lib");
kumoi_test = qu.quantize(220, qu.kumoi, 260);
```

(qu.)natural

List of the frequency ratios of the notes of the natural major scale.

Usage

```
_ : quantize(rf,natural) : _
```

Where:

- `rf`: frequency of the root note of the scale

Test

```
qu = library("quantizers.lib");
natural_test = qu.quantize(220, qu.natural, 260);
```

(qu.)dodeca

List of the frequency ratios of the notes of the dodecaphonic scale.

Usage

```
_ : quantize(rf,dodeca) : _
```

Where:

- `rf`: frequency of the root note of the scale

Test

```
qu = library("quantizers.lib");
dodeca_test = qu.quantize(220, qu.dodeca, 260);
```

(qu.)dimin

List of the frequency ratios of the notes of the diminished scale.

Usage

```
_ : quantize(rf,dimin) : _
```

Where:

- `rf`: frequency of the root note of the scale

Test

```
qu = library("quantizers.lib");
dimin_test = qu.quantize(220, qu.dimin, 260);
```

(qu.)penta

List of the frequency ratios of the notes of the minor pentatonic scale.

Usage

```
_ : quantize(rf,penta) : _
```

Where:

- `rf`: frequency of the root note of the scale

Test

```
qu = library("quantizers.lib");
penta_test = qu.quantize(220, qu.penta, 260);
```

reducemaps.lib

A library providing reduce/map operations in Faust. Its official prefix is `rm`.

The basic idea behind *reduce* operations is to combine several values into a single one by repeatedly applying a binary operation. A typical example is finding the maximum of a set of values by repeatedly applying the binary operation `max`.

In this reducemaps library, you'll find two types of *reduce*, depending on whether you want to reduce n consecutive samples of the same signal or a set of n parallel signals.

References

- https://github.com/grame-cncm/faustlibraries/blob/master/reducemaps.lib

---

(rm.)parReduce

`parReduce(op,N)` combines a set of `N` parallel signals into a single one using a binary operation `op`.

With `parReduce`, this reduction process simultaneously occurs on each half of the incoming signals. In other words, `parReduce(max,256)` is equivalent to `parReduce(max,128),parReduce(max,128) : max`.

To be used with `parReduce`, binary operation `op` must be associative. Additionally, the concept of a binary operation extends to operations that have `2*n` inputs and `n` outputs. For example, complex signals can be simulated using two signals for the real and imaginary parts. In such case, a binary operation would have 4 inputs and 2 outputs.

Please note also that `parReduce` is faster than `topReduce` or `botReduce` for large number of signals. It is therefore the recommended operation whenever `op` is associative.

Usage

```
_,...,_ : parReduce(op, N) : _
```

Where:

- `op`: is a binary operation
- `N`: is the number of incomming signals (`N>0`). We use a capital letter here to indicate that the number of incomming signals must be constant and known at compile time.

Test

```
rm = library("reducemaps.lib");
parReduce_test = (1,2,3,4) : rm.parReduce(+, 4);
```

(rm.)topReduce

`topReduce(op,N)` involves combining a set of `N` parallel signals into a single one using a binary operation `op`. With `topReduce`, the reduction process starts from the top two incoming signals, down to the bottom. In other words, `topReduce(max,256)` is equivalent to `topReduce(max,255),_ : max`.

Contrary to `parReduce`, the binary operation `op` doesn't have to be associative here. Like with `parReduce` the concept of a binary operation can be extended to operations that have 2*n inputs and n outputs. For example, complex signals can be simulated using two signals representing the real and imaginary parts. In such cases, a binary operation would have 4 inputs and 2 outputs.

Usage

```
 _,...,_ : topReduce(op, N) : _
```

Where:

- `op`: is a binary operation
- `N`: is the number of incomming signals (`N>0`). We use a capital letter here to indicate that the number of incomming signals must be constant and known at compile time.

Test

```
rm = library("reducemaps.lib");
topReduce_test = (1,2,3,4) : rm.topReduce(+, 4);
```

(rm.)botReduce

`botReduce(op,N)` combines a set of `N` parallel signals into a single one using a binary operation `op`. With `botReduce`, the reduction process starts from the bottom two incoming signals, up to the top. In other words, `botReduce(max,256)` is equivalent to `_,botReduce(max,255): max`.

Contrary to `parReduce`, the binary operation `op` doesn't have to be associative here. Like with `parReduce` the concept of a binary operation can be extended to operations that have 2*n inputs and n outputs. For example, complex signals can be simulated using two signals representing the real and imaginary parts. In such cases, a binary operation would have 4 inputs and 2 outputs.

Usage

```
_,...,_ : botReduce(op, N) : _
```

Where:

- op: is a binary operation
- N: is the number of incomming signals (`N>0`). We use a capital letter here to indicate that the number of incomming signals must be constant and known at compile time.

Test

```
rm = library("reducemaps.lib");
botReduce_test = (1,2,3,4) : rm.botReduce(+, 4);
```

(rm.)reduce

Reduce a block of `n` consecutive samples of the incomming signal using a binary operation `op`. For example: `reduce(max,128)` will compute the maximun value of each block of 128 samples. Please note that the resulting value, while computed continuously, will be constant for the duration of a block. A new value is only produced at the end of a block. Note also that blocks should be of at least one sample (n>0).

Usage

```
_ : reduce(op, n) : _
```

Where:

- `op`: is a binary operation
- `n`: is the number of consecutive samples in a block.

Test

```
rm = library("reducemaps.lib");
reduce_test = rm.reduce(max, 4, hslider("reduce:input", 0, -1, 1, 0.01));
```

(rm.)reducemap

Like `reduce` but a `foo` function is applied to the result. From a mathematical point of view: `reducemap(op,foo,n)` is equivalent to `reduce(op,n):foo` but more efficient.

Usage

```
_ : reducemap(op, foo, n) : _
```

Where:

- `op`: is a binary operation
- `foo`: is a function applied to the result of the reduction
- `n`: is the number of consecutive samples in a block.

Test

```
rm = library("reducemaps.lib");
reducemap_test = rm.reducemap(+, /(4), 4, hslider("reducemap:input", 0, -1, 1, 0.01));
```

reverbs.lib

Reverbs library. Its official prefix is `re`.

This library provides a collection of artificial reverberation algorithms in Faust. It includes Schroeder, Moorer, Freeverb, and FDN-based designs. These modules can be used for room simulation, spatialization, and creative ambience design in both mono and multichannel contexts.

The Reverbs library is organized into 6 sections:

- Schroeder Reverberators
- Feedback Delay Network (FDN) Reverberators
- Freeverb
- Dattorro Reverb
- JPverb and Greyhole Reverbs
- Keith Barr Allpass Loop Reverb

References

- https://github.com/grame-cncm/faustlibraries/blob/master/reverbs.lib

Schroeder Reverberators

(re.)jcrev

This artificial reverberator take a mono signal and output stereo (`satrev`) and quad (`jcrev`). They were implemented by John Chowning in the MUS10 computer-music language (descended from Music V by Max Mathews). They are Schroeder Reverberators, well tuned for their size. Nowadays, the more expensive freeverb is more commonly used (see the Faust examples directory).

`jcrev` reverb below was made from a listing of "RV", dated April 14, 1972, which was recovered from an old SAIL DART backup tape. John Chowning thinks this might be the one that became the well known and often copied JCREV.

`jcrev` is a standard Faust function.

Usage

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

Test

```
re = library("reverbs.lib");
os = library("oscillators.lib");
jcrev_test = os.osc(440) : re.jcrev;
```

(re.)satrev

This artificial reverberator take a mono signal and output stereo (`satrev`) and quad (`jcrev`). They were implemented by John Chowning in the MUS10 computer-music language (descended from Music V by Max Mathews). They are Schroeder Reverberators, well tuned for their size. Nowadays, the more expensive freeverb is more commonly used (see the Faust examples directory).

`satrev` was made from a listing of "SATREV", dated May 15, 1971, which was recovered from an old SAIL DART backup tape. John Chowning thinks this might be the one used on his often-heard brass canon sound examples, one of which can be found at * https://ccrma.stanford.edu/~jos/wav/FM-BrassCanon2.wav.

Usage

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

Test

```
re = library("reverbs.lib");
os = library("oscillators.lib");
satrev_test = os.osc(330) : re.satrev;
```

Feedback Delay Network (FDN) Reverberators

(re.)fdnrev0

Pure Feedback Delay Network Reverberator (generalized for easy scaling). `fdnrev0` is a standard Faust function.

Usage

```
<1,2,4,...,N signals> <:
fdnrev0(MAXDELAY,delays,BBSO,freqs,durs,loopgainmax,nonl) :>
<1,2,4,...,N signals>
```

Where:

- `N`: 2, 4, 8, … (power of 2)
- `MAXDELAY`: power of 2 at least as large as longest delay-line length
- `delays`: N delay lines, N a power of 2, lengths preferably 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)

Test

```
re = library("reverbs.lib");
os = library("oscillators.lib");
fdnrev0_test = (os.osc(220), os.osc(330), os.osc(440), os.osc(550))
  <: re.fdnrev0(4096, (149, 211, 263, 293), 1, (800, 4000), (2.5, 2.0, 1.5), 0.8, 0.0);
```

References

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

---

(re.)zita_rev_fdn

Internal 8x8 late-reverberation FDN used in the FOSS Linux reverb `zita-rev1` by Fons Adriaensen fons@linuxaudio.org. This is an FDN reverb with allpass comb filters in each feedback delay in addition to the damping filters.

Usage

```
si.bus(8) : zita_rev_fdn(f1,f2,t60dc,t60m,fsmax) : si.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)

Test

```
re = library("reverbs.lib");
os = library("oscillators.lib");
zita_rev_fdn_test = par(i, 8, os.osc(110 * (i + 1)))
  <: re.zita_rev_fdn(200, 2000, 3.0, 2.0, 48000);
```

References

- http://www.kokkinizita.net/linuxaudio/zita-rev1-doc/quickguide.html
- https://ccrma.stanford.edu/~jos/pasp/Zita_Rev1.html

`(re.)zita_rev1_stereo`

Extend `zita_rev_fdn` to include `zita_rev1` input/output mapping in stereo mode. `zita_rev1_stereo` is a standard Faust function.

Usage

```
_,_ : zita_rev1_stereo(rdel,f1,f2,t60dc,t60m,fsmax) : _,_
```

Where:

`rdel` = delay (in ms) before reverberation begins (e.g., 0 to ~100 ms) (remaining args and refs as for `zita_rev_fdn` above)

Test

```
re = library("reverbs.lib");
os = library("oscillators.lib");
zita_rev1_stereo_test = (os.osc(440), os.osc(550))
  : re.zita_rev1_stereo(20, 200, 2000, 3.0, 2.0, 48000);
```

`(re.)zita_rev1_ambi`

Extend `zita_rev_fdn` to include `zita_rev1` input/output mapping in "ambisonics mode", as provided in the Linux C++ version.

Usage

```
_,_ : zita_rev1_ambi(rgxyz,rdel,f1,f2,t60dc,t60m,fsmax) : _,_,_,_
```

Where:

`rgxyz` = relative gain of lanes 1,4,2 to lane 0 in output (e.g., -9 to 9) (remaining args and references as for zita_rev1_stereo above)

Test

```
re = library("reverbs.lib");
os = library("oscillators.lib");
zita_rev1_ambi_test = (os.osc(330), os.osc(550))
  : re.zita_rev1_ambi(0.0, 25, 200, 2000, 3.0, 2.0, 48000);
```

`(re.)vital_rev`

A port of the reverb from the Vital synthesizer. All input parameters have been normalized to a continuous [0,1] range, making them easy to modulate. The scaling of the parameters happens inside the function.

Usage

```
_,_ : vital_rev(prelow, prehigh, lowcutoff, highcutoff, lowgain, highgain, chorus_amt, choru
```

Where:

- `prelow`: In the pre-filter, this is the cutoff frequency of a high-pass filter (hence a low value)
- `prehigh`: In the pre-filter, this is the cutoff frequency of a low-pass filter (hence a high value)
- `lowcutoff`: In the feedback filter stage, this is the cutoff frequency of a low-shelf filter
- `highcutoff`: In the feedback filter stage, this is the cutoff frequency of a high-shelf filter
- `lowgain`: In the feedback filter stage, this is the gain of a low-shelf filter
- `highgain`: In the feedback filter stage, this is the gain of a high-shelf filter
- `chorus_amt`: The amount of chorus modulation in the main delay lines
- `chorus_freq`: The LFO rate of chorus modulation in the main delay lines
- `predelay`: The amount of pre-delay time
- `time`: The decay time of the reverb
- `size`: The size of the room

- mix: A wetness value to use in a final dry/wet mixer

Test

```
re = library("reverbs.lib");
os = library("oscillators.lib");
vital_rev_test = (os.osc(330), os.osc(440))
  : re.vital_rev(0.2, 0.8, 0.5, 0.7, 0.4, 0.6, 0.3, 0.2, 0.1, 0.7, 0.5, 0.4);
```

Freeverb

(re.)mono_freeverb

A simple Schroeder reverberator primarily developed by "Jezar at Dreampoint" that is extensively used in the free-software world. It uses four Schroeder allpasses in series and eight parallel Schroeder-Moorer filtered-feedback comb-filters for each audio channel, and is said to be especially well tuned.

`mono_freeverb` is a standard Faust function.

Usage

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

Where:

- `fb1`: coefficient of the lowpass comb filters (0-1)
- `fb2`: coefficient of the allpass comb filters (0-1)
- `damp`: damping of the lowpass comb filter (0-1)
- `spread`: spatial spread in number of samples (for stereo)

Test

```
re = library("reverbs.lib");
os = library("oscillators.lib");
mono_freeverb_test = os.osc(440) : re.mono_freeverb(0.7, 0.5, 0.3, 30);
```

License While this version is licensed LGPL (with exception) along with other GRAME library functions, the file freeverb.dsp in the examples directory of older Faust distributions, such as faust-0.9.85, was released under the BSD license, which is less restrictive.

(re.)stereo_freeverb

A simple Schroeder reverberator primarily developed by "Jezar at Dreampoint" that is extensively used in the free-software world. It uses four Schroeder allpasses in series and eight parallel Schroeder-Moorer filtered-feedback comb-filters for each audio channel, and is said to be especially well tuned.

Usage

```
_,_ : stereo_freeverb(fb1, fb2, damp, spread) : _,_
```

Where:

- `fb1`: coefficient of the lowpass comb filters (0-1)
- `fb2`: coefficient of the allpass comb filters (0-1)
- `damp`: damping of the lowpass comb filter (0-1)
- `spread`: spatial spread in number of samples (for stereo)

Test

```
re = library("reverbs.lib");
os = library("oscillators.lib");
stereo_freeverb_test = (os.osc(330), os.osc(550))
  : re.stereo_freeverb(0.7, 0.5, 0.3, 30);
```

Dattorro Reverb

(re.)dattorro_rev

Reverberator based on the Dattorro reverb topology. This implementation does not use modulated delay lengths (excursion).

Usage

```
_,_ : dattorro_rev(pre_delay, bw, i_diff1, i_diff2, decay, d_diff1, d_diff2, damping) : _,_
```

Where:

- `pre_delay`: pre-delay in samples (fixed at compile time)
- `bw`: band-width filter (pre filtering); (0 - 1)
- `i_diff1`: input diffusion factor 1; (0 - 1)
- `i_diff2`: input diffusion factor 2;
- `decay`: decay rate; (0 - 1); infinite decay = 1.0
- `d_diff1`: decay diffusion factor 1; (0 - 1)
- `d_diff2`: decay diffusion factor 2;
- `damping`: high-frequency damping; no damping = 0.0

Test

```
re = library("reverbs.lib");
os = library("oscillators.lib");
dattorro_rev_test = (os.osc(330), os.osc(550))
  : re.dattorro_rev(200, 0.5, 0.7, 0.6, 0.5, 0.7, 0.5, 0.2);
```

References

- https://ccrma.stanford.edu/~dattorro/EffectDesignPart1.pdf

(re.)dattorro_rev_default

Reverberator based on the Dattorro reverb topology with reverb parameters from the original paper. This implementation does not use modulated delay lengths (excursion) and uses zero length pre-delay.

Usage

```
_,_ : dattorro_rev_default : _,_
```

Test

```
re = library("reverbs.lib");
os = library("oscillators.lib");
dattorro_rev_default_test = (os.osc(330), os.osc(550))
  : re.dattorro_rev_default;
```

References

- https://ccrma.stanford.edu/~dattorro/EffectDesignPart1.pdf

JPverb and Greyhole Reverbs

(re.)jpverb

An algorithmic reverb (stereo in/out), inspired by the lush chorused sound of certain vintage Lexicon and Alesis reverberation units. Designed to sound great with synthetic sound sources, rather than sound like a realistic space.

Usage

```
_,_ : jpverb(t60, damp, size, early_diff, mod_depth, mod_freq, low, mid, high, low_cutoff, h
```

Where:

- `t60`: approximate reverberation time in seconds ([0.1..60] sec) (T60 - the time for the reverb to decay by 60db when damp == 0). Does not effect early reflections
- `damp`: controls damping of high-frequencies as the reverb decays. 0 is no damping, 1 is very strong damping. Values should be in the range ([0..1])
- `size`: scales size of delay-lines within the reverberator, producing the impression of a larger or smaller space. Values below 1 can sound metallic. Values should be in the range [0.5..5]
- `early_diff`: controls shape of early reflections. Values of 0.707 or more produce smooth exponential decay. Lower values produce a slower build-up of echoes. Values should be in the range ([0..1])
- `mod_depth`: depth ([0..1]) of delay-line modulation. Use in combination with `mod_freq` to set amount of chorusing within the structure
- `mod_freq`: frequency ([0..10] Hz) of delay-line modulation. Use in combination with `mod_depth` to set amount of chorusing within the structure
- `low`: multiplier ([0..1]) for the reverberation time within the low band
- `mid`: multiplier ([0..1]) for the reverberation time within the mid band
- `high`: multiplier ([0..1]) for the reverberation time within the high band
- `low_cutoff`: frequency (100..6000 Hz) at which the crossover between the low and mid bands of the reverb occurs
- `high_cutoff`: frequency (1000..10000 Hz) at which the crossover between the mid and high bands of the reverb occurs

Test

```
re = library("reverbs.lib");
os = library("oscillators.lib");
jpverb_test = (os.osc(330), os.osc(440))
  : re.jpverb(3.0, 0.2, 1.0, 0.8, 0.3, 0.4, 0.9, 0.8, 0.7, 500, 4000);
```

References

- https://doc.sccode.org/Overviews/DEIND.html

`(re.)greyhole`

A complex echo-like effect (stereo in/out), inspired by the classic Eventide effect of a similar name. The effect consists of a diffuser (like a mini-reverb, structurally similar to the one used in `jpverb`) connected in a feedback system with a long, modulated delay-line. Excels at producing spacey washes of sound.

Usage

```
_,_ : greyhole(dt, damp, size, early_diff, feedback, mod_depth, mod_freq) : _,_
```

Where:

- `dt`: approximate reverberation time in seconds ([0.1..60 sec])
- `damp`: controls damping of high-frequencies as the reverb decays. 0 is no damping, 1 is very strong damping. Values should be between ([0..1])
- `size`: control of relative "room size" roughly in the range ([0.5..3])
- `early_diff`: controls pattern of echoes produced by the diffuser. At very low values, the diffuser acts like a delay-line whose length is controlled by the 'size' parameter. Medium values produce a slow build-up of echoes, giving the sound a reversed-like quality. Values of 0.707 or greater than produce smooth exponentially decaying echoes. Values should be in the range ([0..1])
- `feedback`: amount of feedback through the system. Sets the number of repeating echoes. A setting of 1.0 produces infinite sustain. Values should be in the range ([0..1])
- `mod_depth`: depth ([0..1]) of delay-line modulation. Use in combination with `mod_freq` to produce chorus and pitch-variations in the echoes
- `mod_freq`: frequency ([0..10] Hz) of delay-line modulation. Use in combination with `mod_depth` to produce chorus and pitch-variations in the echoes

Test

```
re = library("reverbs.lib");
os = library("oscillators.lib");
greyhole_test = (os.osc(220), os.osc(440))
  : re.greyhole(2.0, 0.3, 1.0, 0.6, 0.5, 0.4, 0.2);
```

References

- https://doc.sccode.org/Overviews/DEIND.html

Keith Barr Allpass Loop Reverb

`(re.)kb_rom_rev1`

Reverberator based on Keith Barr's all-pass single feedback loop reverb topology. Originally designed for the Spin Semiconductor FV-1 chip, this code is an adaptation of the rom_rev1.spn file, part of the Spin Semiconductor Free DSP Programs available on the Spin Semiconductor website.
It was submitted by Keith Barr himself and written in Spin Semiconductor Assembly, a dedicated assembly language for programming the FV-1 chip.

In this topology, when multiple delays and all-pass filters are placed in a loop, sound injected into the loop will recirculate, increasing the density of any impulse as the signal successively passes through the all-pass filters. The result, after a short period of time, is a wash of sound, completely diffused into a natural reverb tail.

The reverb typically has a mono input (as from a single source) but benefits from a stereo output, providing the listener with a fuller, more immersive reverberant image.

Usage

```
_,_ : kb_rom_rev1(rt, damp) : _,_
```

Where:

- `rt`: coefficent of the decay of the reverb (0-1)
- `damp`: coefficient of the lowpass filters (0-1)

Test

```
re = library("reverbs.lib");
os = library("oscillators.lib");
kb_rom_rev1_test = (os.osc(330), os.osc(660))
  : re.kb_rom_rev1(0.7, 0.3);
```

References

- https://www.spinsemi.com/programs.php#://~://text=Keith%20Barrrom_rev1.spn,-ROM%20reverb%202
- https://www.spinsemi.com/knowledge_base/effects.html#Reverberation
- https://www.spinsemi.com/knowledge_base/inst_syntax.html

routes.lib

Routing library. Its official prefix is `ro`.

This library provides tools for managing and organizing audio and control signal routing in Faust. It includes functions for channel mapping, splitting, merging, and dynamic routing, as well as utilities for building multichannel processing structures.

The Routes library is organized into 1 section:

- Functions Reference

References

- https://github.com/grame-cncm/faustlibraries/blob/master/routes.lib

Functions Reference

(ro.)cross

Cross N signals: `(x1,x2,..,xn) -> (xn,..,x2,x1)`. `cross` is a standard Faust function.

Usage

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

Where:

- `N`: number of signals (int, as a constant numerical expression)

Test

```
ro = library("routes.lib");
os = library("oscillators.lib");
cross_test = (os.osc(200), os.osc(300), os.osc(400)) : ro.cross(3);
```

Note Special case: `cross2`:

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

(ro.)crossnn

Cross two `bus(N)`s.

Usage

```
(si.bus(2*N)) : crossnn(N) : (si.bus(2*N))
```

Where:

- `N`: the number of signals in the `bus` (int, as a constant numerical expression)

Test

```
ro = library("routes.lib");
os = library("oscillators.lib");
crossnn_test = (os.osc(110), os.osc(220), os.osc(330), os.osc(440)) : ro.crossnn(2);
```

(ro.)crossn1

Cross `bus(N)` and `bus(1)`.

Usage

```
(si.bus(N),_) : crossn1(N) : (_,si.bus(N))
```

Where:

- `N`: the number of signals in the first `bus` (int, as a constant numerical expression)

Test

```
ro = library("routes.lib");
os = library("oscillators.lib");
crossn1_test = (os.osc(100), os.osc(200), os.osc(300), os.osc(400)) : ro.crossn1(3);
```

(ro.)cross1n

Cross `bus(1)` and `bus(N)`.

Usage

```
(_,si.bus(N)) : crossn1(N) : (si.bus(N),_)
```

Where:

- `N`: the number of signals in the second `bus` (int, as a constant numerical expression)

Test

```
ro = library("routes.lib");
os = library("oscillators.lib");
cross1n_test = (os.osc(150), os.osc(250), os.osc(350), os.osc(450)) : ro.cross1n(3);
```

(ro.)crossNM

Cross `bus(N)` and `bus(M)`.

Usage

```
(si.bus(N),si.bus(M)) : crossNM(N,M) : (si.bus(M),si.bus(N))
```

Where:

- `N`: the number of signals in the first `bus` (int, as a constant numerical expression)
- `M`: the number of signals in the second `bus` (int, as a constant numerical expression)

Test

```
ro = library("routes.lib");
os = library("oscillators.lib");
crossNM_test = (os.osc(180), os.osc(280), os.osc(380), os.osc(480), os.osc(580)) : ro.crossN
```

`(ro.)interleave`

Interleave R x C cables from column order to row order. That is, transpose the input CxR matrix, the first R inputs is the first row.

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

```
si.bus(R*C) : interleave(R,C) : si.bus(R*C)
```

Where:

- `R`: row length (int, as a constant numerical expression)
- `C`: column length (int, as a constant numerical expression)

Test

```
ro = library("routes.lib");
os = library("oscillators.lib");
interleave_test = (os.osc(200), os.osc(300), os.osc(400), os.osc(500)) : ro.interleave(2,2);
```

`(ro.)butterfly`

Addition (first half) then substraction (second half) of interleaved signals.

Usage

```
si.bus(N) : butterfly(N) : si.bus(N)
```

Where:

- `N`: size of the butterfly (N is int, even and as a constant numerical expression)

Test

```
ro = library("routes.lib");
os = library("oscillators.lib");
butterfly_test = (os.osc(250), os.osc(350), os.osc(450), os.osc(550)) : ro.butterfly(4);
```

(ro.)hadamard

Hadamard matrix function of size `N = 2^k`.

Usage

```
si.bus(N) : hadamard(N) : si.bus(N)
```

Where:

- `N`: `2^k`, size of the matrix (int, as a constant numerical expression)

Test

```
ro = library("routes.lib");
os = library("oscillators.lib");
hadamard_test = (os.osc(220), os.osc(330), os.osc(440), os.osc(550)) : ro.hadamard(4);
```

(ro.)recursivize

Create a recursion from two arbitrary processors `p` and `q`.

Usage

```
_,_ : recursivize(p,q) : _,_
```

Where:

- `p`: the forward arbitrary processor
- `q`: the feedback arbitrary processor

Test

```
ro = library("routes.lib");
os = library("oscillators.lib");
recursivize_test = (os.osc(220), os.osc(330)) : ro.recursivize(*(0.5), *(0.3));
```

(ro.)bubbleSort

Sort a set of N parallel signals in ascending order on-the-fly through the Bubble Sort algorithm.

Mechanism: having a set of N parallel signals indexed from 0 to N - 1, compare the first pair of signals and swap them if sig[0] > sig[1]; repeat the pair comparison for the signals sig[1] and sig[2], then again recursively until reaching the signals sig[N - 2] and sig[N - 1]; by the end, the largest element in the set will be placed last; repeat the process for the remaining N - 1 signals until there is a single pair left.

Note that this implementation will always perform the worst-case computation, O(n^2).

Even though the Bubble Sort algorithm is one of the least efficient ones, it is a useful example of how automatic sorting can be implemented at the signal level.

Usage

```
si.bus(N) : bubbleSort(N) : si.bus(N)
```

Where:

- `N`: the number of signals to be sorted (must be an int >= 0, as a constant numerical expression)

Test

```
ro = library("routes.lib");
bubbleSort_test = (
    hslider("bubbleSort:x0", 0.3, -1, 1, 0.01),
    hslider("bubbleSort:x1", -0.2, -1, 1, 0.01),
    hslider("bubbleSort:x2", 0.8, -1, 1, 0.01),
    hslider("bubbleSort:x3", -0.5, -1, 1, 0.01)
) : ro.bubbleSort(4);
```

References

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

signals.lib

Signals library. Its official prefix is `si`.

This library provides fundamental signal processing operations for Faust, including generators, combinators, selectors, and basic DSP utilities. It defines essential functions used across all Faust libraries for building and manipulating audio and control signals.

The Signals library is organized into 1 section:

- Functions Reference

References

- https://github.com/grame-cncm/faustlibraries/blob/master/signals.lib

Functions Reference

`(si.)bus`

Put N cables in parallel. `bus` is a standard Faust function.

Usage

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

Where:

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

Test

```
si = library("signals.lib");
bus_test = (
    hslider("bus:x0", 0, -1, 1, 0.01),
    hslider("bus:x1", 0, -1, 1, 0.01),
    hslider("bus:x2", 0, -1, 1, 0.01)
) : si.bus(3);
```

`(si.)block`

Block - terminate N signals. `block` is a standard Faust function.

Usage

```
si.bus(N) : block(N)
```

Where:

- `N`: the number of signals to be blocked known at compile time

Test

```
si = library("signals.lib");
block_test = (
    hslider("block:x0", 0, -1, 1, 0.01),
    hslider("block:x1", 0, -1, 1, 0.01)
) : (si.block(1), _);
```

(si.)interpolate

Linear interpolation between two signals.

Usage

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

Where:

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

Test

```
si = library("signals.lib");
os = library("oscillators.lib");
interpolate_test = si.interpolate(
    hslider("interpolate:mix", 0.5, 0, 1, 0.01),
    os.osc(220),
    os.osc(440)
);
```

(si.)repeat

Repeat an effect N time(s) and take the parallel sum of all intermediate buses.

Usage

```
si.bus(inputs(FX)) : repeat(N, FX) : si.bus(outputs(FX))
```

Where:

- `N`: Number of repetitions, minimum of 1, a constant numerical expression
- `FX`: an arbitrary effect (N inputs and N outputs) that will be repeated

Test

```
si = library("signals.lib");
repeat_test = hslider("repeat:input", 0, -1, 1, 0.01) : si.repeat(3, *(0.5));
```

Example 1:

```
process = repeat(2, dm.zita_light) : _*.5,_*.5;
```

Example 2:

```
N = 4;
C = 2;
fx(i) = i+1, par(j, C, @(i*5000));
process = 0, si.bus(C) : repeat(N, fx) : !, par(i, C, _*.2/N);
```

References

- https://github.com/orlarey/presentation-compilateur-faust/blob/master/slides.pdf

---

`(si.)smoo`

Smoothing function based on `smooth` ideal to smooth UI signals (sliders, etc.) down. Approximately, this is a 7 Hz one-pole low-pass considering the coefficient calculation: exp(-2pi*CF/SR).

`smoo` is a standard Faust function.

Usage

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

Test

```
si = library("signals.lib");
smoo_test = hslider("smoo:input", 0, -1, 1, 0.01) : si.smoo;
```

`(si.)polySmooth`

A smoothing function based on `smooth` that doesn't smooth when a trigger signal is given. This is very useful when making polyphonic synthesizer to make sure that the value of the parameter is the right one when the note is started.

Usage

```
hslider(...) : polySmooth(g,s,d) : _
```

Where:

- `g`: the gate/trigger signal used when making polyphonic synths
- `s`: the smoothness (see `smooth`)
- `d`: the number of samples to wait before the signal start being smoothed after `g` switched to 1

Test

```
si = library("signals.lib");
polySmooth_test = hslider("polySmooth:input", 0, -1, 1, 0.01)
  : si.polySmooth(button("polySmooth:gate"), 0.999, 32);
```

(si.)smoothAndH

A smoothing function based on `smooth` that holds its output signal when a trigger is sent to it. This feature is convenient when implementing polyphonic instruments to prevent some smoothed parameter to change when a note-off event is sent.

Usage

```
hslider(...) : smoothAndH(g,s) : _
```

Where:

- `g`: the hold signal (0 for hold, 1 for bypass)
- `s`: the smoothness (see `smooth`)

Test

```
si = library("signals.lib");
smoothAndH_test = hslider("smoothAndH:input", 0, -1, 1, 0.01)
  : si.smoothAndH(button("smoothAndH:hold"), 0.999);
```

(si.)bsmooth

Block smooth linear interpolation during a block of samples (given by the `ma.BS` value).

Usage

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

Test

```
si = library("signals.lib");
bsmooth_test = hslider("bsmooth:input", 0, -1, 1, 0.01) : si.bsmooth;
```

(si.)dot

Dot product for two vectors of size N.

Usage

```
si.bus(N), si.bus(N) : dot(N) : _
```

Where:

- `N`: size of the vectors (int, must be known at compile time)

Test

```
si = library("signals.lib");
os = library("oscillators.lib");
dot_test = (
    os.osc(100), os.osc(200), os.osc(300),
    os.osc(400), os.osc(500), os.osc(600)
) : si.dot(3);
```

(si.)smooth

Exponential smoothing by a unity-dc-gain one-pole lowpass. `smooth` is a standard Faust function.

Usage:

```
_ : si.smooth(ba.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.

Test

```
si = library("signals.lib");
smooth_test = hslider("smooth:input", 0, -1, 1, 0.01) : si.smooth(0.9);
```

References

- https://ccrma.stanford.edu/~jos/mdft/Convolution_Example_2_ADSR.html
- https://ccrma.stanford.edu/~jos/aspf/Appendix_B_Inspecting_Assembly.html

---

`(si.)smoothq`

Smoothing with continuously variable curves from Exponential to Linear, with a constant time.

Usage

```
_ : smoothq(time, q) : _;
```

Where:

- `time`: seconds to reach target
- `q`: curve shape (between 0..1, 0 is Exponential, 1 is Linear)

Test

```
si = library("signals.lib");
smoothq_test = hslider("smoothq:input", 0, -1, 1, 0.01) : si.smoothq(0.25, 0.5);
```

`(si.)cbus`

N parallel cables for complex signals. `cbus` is a standard Faust function.

Usage

```
cbus(N)
cbus(4) : (r0,i0), (r1,i1), (r2,i2), (r3,i3)
```

Where:

- `N`: is an integer known at compile time that indicates the number of parallel cables.
- each complex number is represented by two real signals as (real,imag)

Test

```
si = library("signals.lib");
os = library("oscillators.lib");
cbus_test = (
    os.osc(100), os.osc(150),
    os.osc(200), os.osc(250)
) : si.cbus(2);
```

(si.)cmul

Multiply two complex signals pointwise. `cmul` is a standard Faust function.

Usage

```
(r1,i1) : cmul(r2,i2) : (_,_)
```

Where:

- Each complex number is represented by two real signals as (real,imag), so
- `(r1,i1)` = real and imaginary parts of signal 1
- `(r2,i2)` = real and imaginary parts of signal 2

Test

```
si = library("signals.lib");
os = library("oscillators.lib");
cmul_test = si.cmul(
    os.osc(110), os.osc(220),
   os.osc(330), os.osc(440)
);
```

(si.)cconj

Complex conjugation of a (complex) signal. `cconj` is a standard Faust function.

Usage

```
(r1,i1) : cconj : (_,_)
```

Where:

- Each complex number is represented by two real signals as (real,imag), so
- `(r1,i1)` = real and imaginary parts of the input signal
- `(r1,-i1)` = real and imaginary parts of the output signal

Test

```
si = library("signals.lib");
os = library("oscillators.lib");
cconj_test = (os.osc(210), os.osc(310)) : si.cconj;
```

(si.)onePoleSwitching

One pole filter with independent attack and release times.

Usage

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

Where:

- `att`: the attack tau time constant in second
- `rel`: the release tau time constant in second

Test

```
si = library("signals.lib");
onePoleSwitching_test = hslider("onePoleSwitching:input", 0, -1, 1, 0.01)
  : si.onePoleSwitching(0.05, 0.2);
```

(si.)rev

Reverse the input signal by blocks of n>0 samples. `rev(1)` is the indentity function. `rev(n)` has a latency of `n-1` samples.

Usage

```
_ : rev(n) : _
```

Where:

- `n`: the block size in samples

Test

```
si = library("signals.lib");
os = library("oscillators.lib");
rev_test = os.osc(440) : si.rev(32);
```

(si.)vecOp

This function is a generalisation of Faust's iterators such as `prod` and `sum`, and it allows to perform operations on an arbitrary number of vectors, provided that they all have the same length. Unlike Faust's iterators `prod` and `sum` where the vector size is equal to one and the vector space dimension must be specified by the user, this function will infer the vector space dimension and vector size based on the vectors list that we provide.

The outputs of the function are equal to the vector size, whereas the number of inputs is dependent on whether the elements of the vectors provided expect an incoming signal themselves or not. We will see a clarifying example later; in general, the number of total inputs will be the sum of the inputs in each input vector.

Note that we must provide a list of at least two vectors, each with a size that is greater or equal to one.

Usage

```
si.bus(inputs(vectorsList)) : vecOp((vectorsList), op) : si.bus(outputs(ba.take(1, vect
```

Where

- `vectorsList`: is a list of vectors
- `op`: is a two-input, one-output operator

Test

```
si = library("signals.lib");
vecOp_test = si.vecOp((v0, v1), +)
with {
    v0 = (hslider("vecOp:v0_0", 0.1, -1, 1, 0.01), hslider("vecOp:v0_1", 0.2, -1, 1, 0.01))
    v1 = (hslider("vecOp:v1_0", 0.3, -1, 1, 0.01), hslider("vecOp:v1_1", 0.4, -1, 1, 0.01))
};
```

For example, consider the following vectors lists:

```
v0 = (0 , 1 , 2 , 3);
v1 = (4 , 5 , 6 , 7);
v2 = (8 , 9 , 10 , 11);
v3 = (12 , 13 , 14 , 15);
v4 = (+(16) , _ , 18 , *(19));
vv = (v0 , v1 , v2 , v3);
```

Although Faust has limitations for list processing, these vectors can be combined or processed individually.

If we do:

```
process = vecOp(v0, +);
```

the function will deduce a vector space of dimension equal to four and a vector length equal to one. Note that this is equivalent to writing:

```
process = v0 : sum(i, 4, _);
```

Similarly, we can write:

```
process = vecOp((v0 , v1), *) :> _;
```

and we have a dimension-two space and length-four vectors. This is the dot product between vectors v0 and v1, which is equivalent to writing:

```
process = v0 , v1 : dot(4);
```

The examples above have no inputs, as none of the elements of the vectors expect inputs. On the other hand, we can write:

```
process = vecOp((v4 , v4), +);
```

and the function will have six inputs and four outputs, as each vector has three of the four elements expecting an input, times two, as the two input vectors are identical.

Finally, we can write:

```
process = vecOp(vv, &);
```

to perform the bitwise AND on all the elements at the same position in each vector, having dimension equal to the vector length equal to four.

Or even:

```
process = vecOp((vv , vv), &);
```

which gives us a dimension equal to two, and a vector size equal to sixteen.

For a more practical use-case, this is how we can implement a time-invariant feedback delay network with Hadamard matrix:

```
N = 4;
normalisation = 1.0 / sqrt(N);
coeffVec = par(i, N, .99 * normalisation);
delVec = par(i, N, (i + 1) * 3);
process = vecOp((si.bus(N) , si.bus(N)), +) ~
    vecOp((vecOp((ro.hadamard(N) , coeffVec), *) , delVec), @);
```

`(si.)bpar`

Balanced `par` where the repeated expression doesn't depend on a variable. The built-in `par` is implemented as an unbalanced tree, and also has to substitute the variable into the repeated expression, which is expensive even when the variable doesn't appear. This version is implemented as a balanced tree (which allows node reuse during tree traversal) and also doesn't search for the variable. This can be much faster than `par` to compile.

Usage

```
si.bus(N * inputs(f)) : bpar(N, f) : si.bus(N * outputs(f))
```

Where:

- `N`: number of repetitions, minimum 1, a constant numerical expression
- `f`: an arbitrary expression

Test

```
si = library("signals.lib");
os = library("oscillators.lib");
bpar_test = (os.osc(120), os.osc(240), os.osc(360)) : si.bpar(3, *(0.5));
```

Example:

```
// square each of 4000 inputs
process = si.bpar(4000, (_ <: _, _ : *));
```

(si.)bsum

Balanced `sum`, see `si.bpar`.

Usage

```
si.bus(N * inputs(f)) : bsum(N, f) : _
```

Where:

- `N`: number of repetitions, minimum 1, a constant numerical expression
- `f`: an arbitrary expression with 1 output.

Test

```
si = library("signals.lib");
os = library("oscillators.lib");
bsum_test = (os.osc(100), os.osc(200), os.osc(300))
  : si.bsum(3, *(0.5));
```

Example:

```
// square each of 1000 inputs and add the results
process = si.bsum(1000, (_ <: _, _ : *));
```

`(si.)bprod`

Balanced `prod`, see `si.bpar`.

Usage

```
si.bus(N * inputs(f)) : bprod(N, f) : _
```

Where:

- `N`: number of repetitions, minimum 1, a constant numerical expression
- `f`: an arbitrary expression with 1 output.

Test

```
si = library("signals.lib");
bprod_test = (
    hslider("bprod:x0", 0.5, 0, 2, 0.01),
    hslider("bprod:x1", 0.8, 0, 2, 0.01)
) : si.bprod(2, _);
```

Example:

```
// Add 8000 consecutive inputs (in pairs) and multiply the results
process = si.bprod(4000, +);
```

soundfiles.lib

Soundfiles library. Its official prefix is `so`.

This library provides functions and abstractions to read, write, and manage audio files in Faust. It supports interpolation and looping controls for integration of recorded or pre-rendered audio in synthesis, effects, and compositional contexts.

The Soundfiles library is organized into 1 section:

- Functions Reference

References

- https://github.com/grame-cncm/faustlibraries/blob/master/soundfiles.lib

Functions Reference

(so.)loop

Play a soundfile in a loop taking into account its sampling rate. `loop` is a standard Faust function.

Usage

```
loop(sf, part) : si.bus(outputs(sf))
```

Where:

- `sf`: the soundfile
- `part`: the part in the soundfile list of sounds

Test

```
so = library("soundfiles.lib");
sf = soundfile("sound[url:{'tests/assets/silence.wav'}]", 1);
loop_test = so.loop(sf, 0);
```

(so.)loop_speed

Play a soundfile in a loop taking into account its sampling rate, with speed control. `loop_speed` is a standard Faust function.

Usage

```
loop_speed(sf, part, speed) : si.bus(outputs(sf))
```

Where:

- `sf`: the soundfile
- `part`: the part in the soundfile list of sounds
- `speed`: the speed between 0 and n

Test

```
so = library("soundfiles.lib");
sf = soundfile("sound[url:{'tests/assets/silence.wav'}]", 1);
loop_speed_test = so.loop_speed(sf, 0, hslider("loop_speed:speed", 1, 0, 2, 0.01));
```

(so.)loop_speed_level

Play a soundfile in a loop taking into account its sampling rate, with speed and level controls. `loop_speed_level` is a standard Faust function.

Usage

```
loop_speed_level(sf, part, speed, level) : si.bus(outputs(sf))
```

Where:

- `sf`: the soundfile
- `part`: the part in the soundfile list of sounds
- `speed`: the speed between 0 and n
- `level`: the volume between 0 and n

Test

```
so = library("soundfiles.lib");
sf = soundfile("sound[url:{'tests/assets/silence.wav'}]", 1);
loop_speed_level_test = so.loop_speed_level(
    sf,
    0,
    hslider("loop_speed_level:speed", 1, 0, 2, 0.01),
    hslider("loop_speed_level:level", 0.5, 0, 1, 0.01)
);
```

spats.lib

Spatialization (Spats) library. Its official prefix is `sp`.

This library provides spatialization in Faust. It includes panning and wfs algorithms.

The Spats library is organized into 1 section:

- Functions Reference

References

- https://github.com/grame-cncm/faustlibraries/blob/master/spats.lib

Functions Reference

`(sp.)panner`

A simple linear stereo panner. `panner` is a standard Faust function.

Usage

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

Where:

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

Test

```
sp = library("spats.lib");
os = library("oscillators.lib");
panner_test = os.osc(220) : sp.panner(hslider("panner:pan", 0.3, 0, 1, 0.01));
```

`(sp.)constantPowerPan`

Apply the constant power pan rule to a stereo signal. The channels are not respatialized. Their gains are simply adjusted. A pan of 0 preserves the left channel and silences the right channel. A pan of 1 has the opposite effect. A pan value of 0.5 applies a gain of 0.5 to both channels.

Usage

```
_,_ : constantPowerPan(p) : _,_
```

Where:

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

Test

```
sp = library("spats.lib");
os = library("oscillators.lib");
constantPowerPan_test = (os.osc(110), os.osc(220))
  : sp.constantPowerPan(hslider("constantPowerPan:pan", 0.4, 0, 1, 0.01));
```

`(sp.)spat`

GMEM SPAT: n-outputs spatializer. `spat` is a standard Faust function.

Usage

```
_ : spat(N,r,d) : si.bus(N)
```

Where:

- `N`: number of outputs (a constant numerical expression)
- `r`: rotation (between 0 et 1)
- `d`: distance of the source (between 0 et 1)

Test

```
sp = library("spats.lib");
os = library("oscillators.lib");
spat_test = os.osc(330)
  : sp.spat(4,
      hslider("spat:rotation", 0.25, 0, 1, 0.01),
      hslider("spat:distance", 0.5, 0, 1, 0.01));
```

(sp.)wfs

Wave Field Synthesis algorithm for multiple sound sources. Implementation generalized starting from Pierre Lecomte version.

Usage

```
wfs(xref, yref, zref, speakersDist, nSources, nSpeakers, inProc, xs, ys, zs) : si.bus(nSpeak
```

Where:

- `xref`: x-coordinate of the reference listening point in meters
- `yref`: y-coordinate of the reference listening point in meters
- `zref`: z-coordinate of the reference listening point in meters
- `speakersDist`: distance between speakers in meters
- `nSources`: number of sound sources
- `nSpeakers`: number of speakers
- `inProc`: per-source processor function, as a function of the source index
- `xs`: x-coordinate of the sound source in meters, as a function of the source index
- `ys`: y-coordinate of the sound source in meters, as a function of the source index
- `zs`: z-coordinate of the sound source in meters, as a function of the source index

Test

```
sp = library("spats.lib");
os = library("oscillators.lib");
wfs_proc(i) = *(0.5); // Simple gain processor
wfs_xs(i) = 0.0;
wfs_ys(i) = 1.0;
wfs_zs(i) = 0.0;
wfs_test = os.osc(440)
  : sp.wfs(0, 1, 0, 0.5, 1, 2, wfs_inGain, wfs_proc, wfs_xs, wfs_ys, wfs_zs);
```

(sp.)wfs_ui

Wave Field Synthesis algorithm for multiple sound sources with a built-in UI.

Usage

```
wfs_ui(xref, yref, zref, speakersDist, nSources, nSpeaker) : si.bus(nSpeakers)
```

Where:

- `xref`: x-coordinate of the reference listening point in meters
- `yref`: y-coordinate of the reference listening point in meters
- `zref`: z-coordinate of the reference listening point in meters
- `speakersDist`: distance between speakers in meters
- `nSources`: number of sound sources
- `nSpeakers`: number of speakers

Test

```
sp = library("spats.lib");
os = library("oscillators.lib");
wfs_ui_test = os.osc(550)
  : sp.wfs_ui(0, 1, 0, 0.5, 1, 2);
```

Example test program

```
// Distance between speakers in meters
speakersDist = 0.0783;

// Reference listening point (central position for WFS)
xref = 0;
yref = 1;
zref = 0;

Spatialize 4 sound sources on 16 speakers
process = wfs_ui(xref,yref,zref,speakersDist,4,16);
```

(sp.)stereoize

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

Usage

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

Where:

- p: the arbitrary processor

Test

```
sp = library("spats.lib");
os = library("oscillators.lib");
stereoize_test = (os.osc(660), os.osc(770))
  : sp.stereoize(+);
```

synths.lib

Synths library. Its official prefix is sy.

This library provides synthesizer and drum building blocks.

The Synths library is organized into 2 sections:

- Synthesizers
- Drum Synthesis

References

- https://github.com/grame-cncm/faustlibraries/blob/master/synths.lib

Synthesizers

(sy.)popFilterDrum

A simple percussion instrument based on a "popped" resonant bandpass filter. popFilterDrum is a standard Faust function.

Usage

```
popFilterDrum(freq,q,gate) : _
```

Where:

- freq: the resonance frequency of the instrument in Hz
- q: the q of the res filter (typically, 5 is a good value)
- gate: the trigger signal (0 or 1)

Test

```
sy = library("synths.lib");
popFilterDrum_test = sy.popFilterDrum(
    hslider("popFilterDrum:freq", 200, 50, 1000, 1),
    hslider("popFilterDrum:q", 5, 1, 20, 0.1),
```

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

(sy.)dubDub

A simple synth based on a sawtooth wave filtered by a resonant lowpass. `dubDub` is a standard Faust function.

Usage

```
dubDub(freq,ctFreq,q,gate) : _
```

Where:

- `freq`: frequency of the sawtooth in Hz
- `ctFreq`: cutoff frequency of the filter
- `q`: Q of the filter
- `gate`: the trigger signal (0 or 1)

Test

```
sy = library("synths.lib");
dubDub_test = sy.dubDub(
    hslider("dubDub:freq", 220, 50, 1000, 1),
    hslider("dubDub:cutoff", 800, 100, 6000, 1),
    hslider("dubDub:q", 2, 0.2, 10, 0.1),
    button("dubDub:gate")
);
```

(sy.)sawTrombone

A simple trombone based on a lowpassed sawtooth wave. `sawTrombone` is a standard Faust function.

Usage

```
sawTrombone(freq,gain,gate) : _
```

Where:

- `freq`: the frequency in Hz
- `gain`: the gain (0-1)
- `gate`: the gate (0 or 1)

Test

```
sy = library("synths.lib");
sawTrombone_test = sy.sawTrombone(
    hslider("sawTrombone:freq", 196, 50, 600, 1),
    hslider("sawTrombone:gain", 0.6, 0, 1, 0.01),
    button("sawTrombone:gate")
);
```

`(sy.)combString`

Simplest string physical model ever based on a comb filter. `combString` is a standard Faust function.

Usage

```
combString(freq,res,gate) : _
```

Where:

- `freq`: the frequency of the string in Hz
- `res`: string T60 (resonance time) in second
- `gate`: trigger signal (0 or 1)

Test

```
sy = library("synths.lib");
combString_test = sy.combString(
    hslider("combString:freq", 220, 55, 880, 1),
    hslider("combString:res", 4, 0.1, 10, 0.01),
    button("combString:gate")
);
```

`(sy.)additiveDrum`

A simple drum using additive synthesis. `additiveDrum` is a standard Faust function.

Usage

```
additiveDrum(freq,freqRatio,gain,harmDec,att,rel,gate) : _
```

Where:

- `freq`: the resonance frequency of the drum in Hz

- `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)

Test

```
sy = library("synths.lib");
additiveDrum_test = sy.additiveDrum(
    hslider("additiveDrum:freq", 180, 60, 600, 1),
    (1, 1.3, 2.4, 3.2),
    (1, 0.8, 0.6, 0.4),
    hslider("additiveDrum:harmDec", 0.4, 0, 1, 0.01),
    0.01,
    0.4,
    button("additiveDrum:gate")
);
```

`(sy.)fm`

An FM synthesizer with an arbitrary number of modulators connected as a sequence. `fm` is a standard Faust function.

Usage

```
freqs = (300,400,...);
indices = (20,...);
fm(freqs,indices) : _
```

Where:

- `freqs`: a list of frequencies where the first one is the frequency of the carrier and the others, the frequency of the modulator(s)
- `indices`: the indices of modulation (Nfreqs-1)

Test

```
sy = library("synths.lib");
fm_test = sy.fm((220, 440, 660), (1.5, 0.8));
```

Drum Synthesis

Drum Synthesis ported in Faust from a version written in Elementary and JavaScript by Nick Thompson.

References

- https://www.nickwritesablog.com/drum-synthesis-in-javascript/

`(sy.)kick`

Kick drum synthesis via a pitched sine sweep.

Usage

```
kick(pitch, click, attack, decay, drive, gate) : _
```

Where:

- `pitch`: the base frequency of the kick drum in Hz
- `click`: the speed of the pitch envelope, tuned for [0.005s, 1s]
- `attack`: attack time in seconds, tuned for [0.005s, 0.4s]
- `decay`: decay time in seconds, tuned for [0.005s, 4.0s]
- `drive`: a gain multiplier going into the saturator. Tuned for [1, 10]
- `gate`: the gate which triggers the amp envelope

Test

```
sy = library("synths.lib");
kick_test = sy.kick(
    hslider("kick:pitch", 60, 30, 120, 0.1),
    hslider("kick:click", 0.2, 0.005, 1, 0.001),
    0.01,
    0.5,
    hslider("kick:drive", 3, 1, 10, 0.1),
    button("kick:gate")
);
```

References

- https://github.com/nick-thompson/drumsynth/blob/master/kick.js

---

`(sy.)clap`

Clap synthesis via filtered white noise.

Usage

```
clap(tone, attack, decay, gate) : _
```

Where:

- `tone`: bandpass filter cutoff frequency, tuned for [400Hz, 3500Hz]
- `attack`: attack time in seconds, tuned for [0s, 0.2s]
- `decay`: decay time in seconds, tuned for [0s, 4.0s]
- `gate`: the gate which triggers the amp envelope

Test

```
sy = library("synths.lib");
clap_test = sy.clap(
    hslider("clap:tone", 1200, 400, 3500, 10),
    0.01,
    0.6,
    button("clap:gate")
);
```

References

- https://github.com/nick-thompson/drumsynth/blob/master/clap.js

---

`(sy.)hat`

Hi hat drum synthesis via phase modulation.

Usage

```
hat(pitch, tone, attack, decay, gate): _
```

Where:

- `pitch`: base frequency in the range [317Hz, 3170Hz]
- `tone`: bandpass filter cutoff frequency, tuned for [800Hz, 18kHz]
- `attack`: attack time in seconds, tuned for [0.005s, 0.2s]
- `decay`: decay time in seconds, tuned for [0.005s, 4.0s]
- `gate`: the gate which triggers the amp envelope

Test

```
sy = library("synths.lib");
hat_test = sy.hat(
    hslider("hat:pitch", 800, 317, 3170, 1),
    hslider("hat:tone", 5000, 800, 18000, 10),
    0.005,
```

```
    0.3,
    button("hat:gate")
);
```

References

- https://github.com/nick-thompson/drumsynth/blob/master/hat.js

vaeffects.lib

Virtual Analog Effects (VAE) library. Its official prefix is `ve`.

This library provides virtual analog (VA) audio effects modeled after classic analog circuitry. It includes nonlinear filters and effects.

The virtual analog filter library is organized into 7 sections:

- Moog Filters
- Korg 35 Filters
- Oberheim Filters
- Sallen Key Filters
- Korg 35 Filters
- Vicanek's matched (decramped) second-order filters
- Effects

References

- https://github.com/grame-cncm/faustlibraries/blob/master/vaeffects.lib

Moog Filters

`(ve.)moog_vcf`

Moog "Voltage Controlled Filter" (VCF) in "analog" form. Moog VCF implemented using the same logical block diagram as the classic analog circuit. As such, it neglects the one-sample delay associated with the feedback path around the four one-poles. This extra delay alters the response, especially at high frequencies (see reference [1] for details). See `moog_vcf_2b` below for a more accurate implementation.

Usage

`_ : moog_vcf(res,fr) : _`

Where:

- `res`: normalized amount of corner-resonance between 0 and 1 (0 is no resonance, 1 is maximum)
- `fr`: corner-resonance frequency in Hz (less than SR/6.3 or so)

Test

```
ve = library("vaeffects.lib");
os = library("oscillators.lib");
moog_vcf_test = os.osc(440)
  : ve.moog_vcf(
      hslider("moog_vcf:res", 0.5, 0, 1, 0.01),
      hslider("moog_vcf:freq", 1000, 50, 4000, 1)
    );
```

References

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

`(ve.)moog_vcf_2b[n]`

Moog "Voltage Controlled Filter" (VCF) as two biquads. Implementation of the ideal Moog VCF transfer function factored into second-order sections. As a result, it is more accurate than `moog_vcf` above, but its coefficient formulas are more complex when one or both parameters are varied. Here, res is the fourth root of that in `moog_vcf`, so, as the sampling rate approaches infinity, `moog_vcf(res,fr)` becomes equivalent to `moog_vcf_2b[n](res^4,fr)` (when res and fr are constant). `moog_vcf_2b` uses two direct-form biquads (`tf2`). `moog_vcf_2bn` uses two protected normalized-ladder biquads (`tf2np`).

Usage

```
_ : moog_vcf_2b(res,fr) : _
_ : moog_vcf_2bn(res,fr) : _
```

Where:

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

Test

```
ve = library("vaeffects.lib");
os = library("oscillators.lib");
moog_vcf_2b_test = os.osc(330)
  : ve.moog_vcf_2b(
```

```
      hslider("moog_vcf_2b:res", 0.4, 0, 1, 0.01),
      hslider("moog_vcf_2b:freq", 1200, 50, 6000, 1)
    );
moog_vcf_2bn_test = os.osc(330)
  : ve.moog_vcf_2bn(
      hslider("moog_vcf_2bn:res", 0.4, 0, 1, 0.01),
      hslider("moog_vcf_2bn:freq", 1200, 50, 6000, 1)
    );
```

`(ve.)moogLadder`

Virtual analog model of the 4th-order Moog Ladder (without any nonlinearities), which is arguably the most well-known ladder filter in analog synthesizers. Several 1st-order filters are cascaded in series. Feedback is then used, in part, to control the cut-off frequency and the resonance.

Usage

```
_ : moogLadder(normFreq,Q) : _
```

Where:

- `normFreq`: normalized frequency (0-1)
- `Q`: quality factor between .707 (0 feedback coefficient) to 25 (feedback = 4, which is the self-oscillating threshold).

Test

```
ve = library("vaeffects.lib");
os = library("oscillators.lib");
moogLadder_test = os.osc(220)
  : ve.moogLadder(
      hslider("moogLadder:normFreq", 0.3, 0, 1, 0.001),
      hslider("moogLadder:Q", 4, 0.7, 20, 0.1)
    );
```

References

- [Zavalishin 2012] (revision 2.1.2, February 2020)
- https://www.native-instruments.com/fileadmin/ni_media/downloads/pdf/VAFilterDesign_2.1.2.pdf
- Lorenzo Della Cioppa's correction to Pirkle's implementation: https://www.kvraudio.com/forum/viewtopic.php?f=33%3Chttps://www.kvraudio.com/forum/viewtopic.php?f=33&t=571909%3Et=571909

(ve.)lowpassLadder4

Topology-preserving transform implementation of a four-pole ladder lowpass. This is essentially the same filter as the moogLadder above except for the parameters, which will be expressed in Hz, for the cutoff, and as a raw feedback coefficient, for the resonance. Also, note that the parameter order has changed.

Usage

```
_ : lowpassLadder4(k, CF) : _
```

Where:

- `k`: feedback coefficient between 0 and 4, which is the stability threshold.
- `CF`: the filter's cutoff in Hz.

Test

```
ve = library("vaeffects.lib");
os = library("oscillators.lib");
lowpassLadder4_test = os.osc(110)
  : ve.lowpassLadder4(
      hslider("lowpassLadder4:k", 2.0, 0, 4, 0.1),
      hslider("lowpassLadder4:freq", 800, 50, 5000, 1)
    );
```

Notes:

If you want to express the feedback coefficient as the resonance peak, you can use the formula:

```
k = 4.0 - 1.0 / Q;
```

where Q, between .25 and infinity, corresponds to the peak of the filter at cutoff. I.e., if you feed the filter with a sine whose frequency is the same as the cutoff, the output peak corresponds exactly to that set via the Q-param. #### References

- [Zavalishin 2012] (revision 2.1.2, February 2020)
- https://www.native-instruments.com/fileadmin/ni_media/downloads/pdf/VAFilterDesign_2.1.2.pdf

---

(ve.)moogHalfLadder

Virtual analog model of the 2nd-order Moog Half Ladder (simplified version of `(ve.)moogLadder`). Several 1st-order filters are cascaded in series. Feedback is then used, in part, to control the cut-off frequency and the resonance.

This filter was implemented in Faust by Eric Tarr during the 2019 Embedded DSP With Faust Workshop.

Usage

`_ : moogHalfLadder(normFreq,Q) : _`

Where:

- `normFreq`: normalized frequency (0-1)
- `Q`: q

Test

```
ve = library("vaeffects.lib");
os = library("oscillators.lib");
moogHalfLadder_test = os.osc(220)
  : ve.moogHalfLadder(
      hslider("moogHalfLadder:normFreq", 0.3, 0, 1, 0.001),
      hslider("moogHalfLadder:Q", 4, 0.7, 20, 0.1)
    );
```

References

- https://www.willpirkle.com/app-notes/virtual-analog-moog-half-ladder-filter
- http://www.willpirkle.com/Downloads/AN-8MoogHalfLadderFilter.pdf

---

`(ve.)diodeLadder`

4th order virtual analog diode ladder filter. In addition to the individual states used within each independent 1st-order filter, there are also additional feedback paths found in the block diagram. These feedback paths are labeled as connecting states. Rather than separately storing these connecting states in the Faust implementation, they are simply implicitly calculated by tracing back to the other states (`s1`,`s2`,`s3`,`s4`) each recursive step.

This filter was implemented in Faust by Eric Tarr during the 2019 Embedded DSP With Faust Workshop.

Usage

`_ : diodeLadder(normFreq,Q) : _`

Where:

- `normFreq`: normalized frequency (0-1)
- `Q`: q

Test

```
ve = library("vaeffects.lib");
os = library("oscillators.lib");
diodeLadder_test = os.osc(220)
  : ve.diodeLadder(
      hslider("diodeLadder:normFreq", 0.4, 0, 1, 0.001),
      hslider("diodeLadder:Q", 4, 0.7, 20, 0.1)
    );
```

References

- https://www.willpirkle.com/virtual-analog-diode-ladder-filter/
- http://www.willpirkle.com/Downloads/AN-6DiodeLadderFilter.pdf

Korg 35 Filters

The following filters are virtual analog models of the Korg 35 low-pass filter and high-pass filter found in the MS-10 and MS-20 synthesizers. The virtual analog models for the LPF and HPF are different, making these filters more interesting than simply tapping different states of the same circuit.

These filters were implemented in Faust by Eric Tarr during the 2019 Embedded DSP With Faust Workshop.

Filter history:

- https://secretlifeofsynthesizers.com/the-korg-35-filter/

`(ve.)korg35LPF`

Virtual analog models of the Korg 35 low-pass filter found in the MS-10 and MS-20 synthesizers.

Usage

```
_ : korg35LPF(normFreq,Q) : _
```

Where:

- `normFreq`: normalized frequency (0-1)
- `Q`: q

Test

```
ve = library("vaeffects.lib");
os = library("oscillators.lib");
korg35LPF_test = os.osc(220)
```

```
  : ve.korg35LPF(
      hslider("korg35LPF:normFreq", 0.35, 0, 1, 0.001),
      hslider("korg35LPF:Q", 3.5, 0.7, 10, 0.1)
    );
```

(ve.)korg35HPF

Virtual analog models of the Korg 35 high-pass filter found in the MS-10 and MS-20 synthesizers.

Usage

```
_ : korg35HPF(normFreq,Q) : _
```

Where:

- `normFreq`: normalized frequency (0-1)
- `Q`: q

Test

```
ve = library("vaeffects.lib");
os = library("oscillators.lib");
korg35HPF_test = os.osc(330)
  : ve.korg35HPF(
      hslider("korg35HPF:normFreq", 0.4, 0, 1, 0.001),
      hslider("korg35HPF:Q", 3.5, 0.7, 10, 0.1)
    );
```

Oberheim Filters

The following filter (4 types) is an implementation of the virtual analog model described in Section 7.2 of the Will Pirkle book, "Designing Software Synthesizer Plug-ins in C++". It is based on the block diagram in Figure 7.5.

The Oberheim filter is a state-variable filter with soft-clipping distortion within the circuit.

In many VA filters, distortion is accomplished using the "tanh" function. For this Faust implementation, that distortion function was replaced with the `(ef.)cubicnl` function.

(ve.)oberheim

Generic multi-outputs Oberheim filter that produces the BSF, BPF, HPF and LPF outputs (see description above).

Usage

```
_ : oberheim(normFreq,Q) : _,_,_,_
```

Where:

- `normFreq`: normalized frequency (0-1)
- `Q`: q

Test

```
ve = library("vaeffects.lib");
os = library("oscillators.lib");
oberheim_test = os.osc(220)
  : ve.oberheim(
      hslider("oberheim:normFreq", 0.4, 0, 1, 0.001),
      hslider("oberheim:Q", 1.5, 0.5, 10, 0.1)
    );
```

`(ve.)oberheimBSF`

Band-Stop Oberheim filter (see description above). Specialize the generic implementation: keep the first BSF output, the compiler will only generate the needed code.

Usage

```
_ : oberheimBSF(normFreq,Q) : _
```

Where:

- `normFreq`: normalized frequency (0-1)
- `Q`: q

Test

```
ve = library("vaeffects.lib");
os = library("oscillators.lib");
oberheimBSF_test = os.osc(220)
  : ve.oberheimBSF(
      hslider("oberheimBSF:normFreq", 0.4, 0, 1, 0.001),
      hslider("oberheimBSF:Q", 1.5, 0.5, 10, 0.1)
    );
```

(ve.)oberheimBPF

Band-Pass Oberheim filter (see description above). Specialize the generic implementation: keep the second BPF output, the compiler will only generate the needed code.

Usage

```
_ : oberheimBPF(normFreq,Q) : _
```

Where:

- `normFreq`: normalized frequency (0-1)
- `Q`: q

Test

```
ve = library("vaeffects.lib");
os = library("oscillators.lib");
oberheimBPF_test = os.osc(220)
  : ve.oberheimBPF(
      hslider("oberheimBPF:normFreq", 0.4, 0, 1, 0.001),
      hslider("oberheimBPF:Q", 1.5, 0.5, 10, 0.1)
    );
```

(ve.)oberheimHPF

High-Pass Oberheim filter (see description above). Specialize the generic implementation: keep the third HPF output, the compiler will only generate the needed code.

Usage

```
_ : oberheimHPF(normFreq,Q) : _
```

Where:

- `normFreq`: normalized frequency (0-1)
- `Q`: q

Test

```
ve = library("vaeffects.lib");
os = library("oscillators.lib");
oberheimHPF_test = os.osc(220)
  : ve.oberheimHPF(
      hslider("oberheimHPF:normFreq", 0.4, 0, 1, 0.001),
      hslider("oberheimHPF:Q", 1.5, 0.5, 10, 0.1)
```

```
    );
```

(ve.)oberheimLPF

Low-Pass Oberheim filter (see description above). Specialize the generic implementation: keep the fourth LPF output, the compiler will only generate the needed code.

Usage

```
_ : oberheimLPF(normFreq,Q) : _
```

Where:

- `normFreq`: normalized frequency (0-1)
- `Q`: q

Test

```
ve = library("vaeffects.lib");
os = library("oscillators.lib");
oberheimLPF_test = os.osc(220)
  : ve.oberheimLPF(
      hslider("oberheimLPF:normFreq", 0.4, 0, 1, 0.001),
      hslider("oberheimLPF:Q", 1.5, 0.5, 10, 0.1)
    );
```

Sallen Key Filters

The following filters were implemented based on VA models of synthesizer filters.

The modeling approach is based on a Topology Preserving Transform (TPT) to resolve the delay-free feedback loop in the corresponding analog filters.

The primary processing block used to build other filters (Moog, Korg, etc.) is based on a 1st-order Sallen-Key filter.

The filters included in this script are 1st-order LPF/HPF and 2nd-order state-variable filters capable of LPF, HPF, and BPF.

Resources:

- Vadim Zavalishin (2018) "The Art of VA Filter Design", v2.1.0
- https://www.native-instruments.com/fileadmin/ni_media/downloads/pdf/VAFilterDesign_2.1.0.pdf
- Will Pirkle (2014) "Resolving Delay-Free Loops in Recursive Filters Using
- the Modified Härmä Method", AES 137 http://www.aes.org/e-lib/browse.cfm?elib=17517

- Description and diagrams of 1st- and 2nd-order TPT filters:
- https://www.willpirkle.com/706-2/

(ve.)sallenKeyOnePole

Sallen-Key generic One Pole filter that produces the LPF and HPF outputs (see description above).

For the Faust implementation of this filter, recursion (`letrec`) is used for storing filter "states". The output (e.g. `y`) is calculated by using the input signal and the previous states of the filter.

During the current recursive step, the states of the filter (e.g. `s`) for the next step are also calculated.

Admittedly, this is not an efficient way to implement a filter because it requires independently calculating the output and each state during each recursive step. However, it works as a way to store and use "states" within the constraints of Faust. The simplest example is the 1st-order LPF (shown on the cover of Zavalishin * 2018 and Fig 4.3 of https://www.willpirkle.com/706-2/).

Here, the input signal is split in parallel for the calculation of the output signal, `y`, and the state `s`. The value of the state is only used for feedback to the next step of recursion. It is blocked (!) from also being routed to the output.

A trick used for calculating the state `s` is to observe that the input to the delay block is the sum of two signal: what appears to be a feedforward path and a feedback path. In reality, the signals being summed are identical `(signal*2)` plus the value of the current state.

Usage

```
_ : sallenKeyOnePole(normFreq) : _,_
```

Where:

- `normFreq`: normalized frequency (0-1)

Test

```
ve = library("vaeffects.lib");
os = library("oscillators.lib");
sallenKeyOnePole_test = os.osc(440)
  : ve.sallenKeyOnePole(
      hslider("sallenKeyOnePole:normFreq", 0.25, 0, 1, 0.001)
    );
```

(ve.)sallenKeyOnePoleLPF

Sallen-Key One Pole lowpass filter (see description above). Specialize the generic implementation: keep the first LPF output, the compiler will only generate the needed code.

Usage

```
_ : sallenKeyOnePoleLPF(normFreq) : _
```

Where:

- `normFreq`: normalized frequency (0-1)

Test

```
ve = library("vaeffects.lib");
os = library("oscillators.lib");
sallenKeyOnePoleLPF_test = os.osc(440)
  : ve.sallenKeyOnePoleLPF(
      hslider("sallenKeyOnePoleLPF:normFreq", 0.25, 0, 1, 0.001)
    );
```

(ve.)sallenKeyOnePoleHPF

Sallen-Key One Pole Highpass filter (see description above). The dry input signal is routed in parallel to the output. The LPF'd signal is subtracted from the input so that the HPF remains. Specialize the generic implementation: keep the second HPF output, the compiler will only generate the needed code.

Usage

```
_ : sallenKeyOnePoleHPF(normFreq) : _
```

Where:

- `normFreq`: normalized frequency (0-1)

Test

```
ve = library("vaeffects.lib");
os = library("oscillators.lib");
sallenKeyOnePoleHPF_test = os.osc(440)
  : ve.sallenKeyOnePoleHPF(
      hslider("sallenKeyOnePoleHPF:normFreq", 0.25, 0, 1, 0.001)
    );
```

`(ve.)sallenKey2ndOrder`

Sallen-Key generic 2nd order filter that produces the LPF, BPF and HPF outputs.

This is a 2nd-order Sallen-Key state-variable filter. The idea is that by "tapping" into different points in the circuit, different filters (LPF,BPF,HPF) can be achieved. See Figure 4.6 of * https://www.willpirkle.com/706-2/

This is also a good example of the next step for generalizing the Faust programming approach used for all these VA filters. In this case, there are three things to calculate each recursive step (`y`,`s1`,`s2`). For each thing, the circuit is only calculated up to that point.

Comparing the LPF to BPF, the output signal (`y`) is calculated similarly. Except, the output of the BPF stops earlier in the circuit. Similarly, the states (`s1` and `s2`) only differ in that `s2` includes a couple more terms beyond what is used for `s1`.

Usage

```
_ : sallenKey2ndOrder(normFreq,Q) : _,_,_
```

Where:

- `normFreq`: normalized frequency (0-1)
- `Q`: quality factor controlling the sharpness/resonance of the filter around the center frequency (CF). For bandpass filters, higher Q increases the gain at the center frequency. Must be in the range `[ma.EPSILON, ma.MAX]`

Test

```
ve = library("vaeffects.lib");
os = library("oscillators.lib");
sallenKey2ndOrder_test = os.osc(330)
  : ve.sallenKey2ndOrder(
      hslider("sallenKey2ndOrder:normFreq", 0.3, 0, 1, 0.001),
      hslider("sallenKey2ndOrder:Q", 1.0, 0.1, 10, 0.1)
    );
```

`(ve.)sallenKey2ndOrderLPF`

Sallen-Key 2nd order lowpass filter (see description above). Specialize the generic implementation: keep the first LPF output, the compiler will only generate the needed code.

Usage

```
_ : sallenKey2ndOrderLPF(normFreq,Q) : _
```

Where:

- `normFreq`: normalized frequency (0-1)
- `Q`: quality factor controlling the sharpness/resonance of the filter around the center frequency (CF). For bandpass filters, higher Q increases the gain at the center frequency. Must be in the range `[ma.EPSILON, ma.MAX]`

Test

```
ve = library("vaeffects.lib");
os = library("oscillators.lib");
sallenKey2ndOrderLPF_test = os.osc(330)
  : ve.sallenKey2ndOrderLPF(
      hslider("sallenKey2ndOrderLPF:normFreq", 0.3, 0, 1, 0.001),
      hslider("sallenKey2ndOrderLPF:Q", 0.8, 0.1, 10, 0.1)
    );
```

`(ve.)sallenKey2ndOrderBPF`

Sallen-Key 2nd order bandpass filter (see description above). Specialize the generic implementation: keep the second BPF output, the compiler will only generate the needed code.

Usage

```
_ : sallenKey2ndOrderBPF(normFreq,Q) : _
```

Where:

- `normFreq`: normalized frequency (0-1)
- `Q`: quality factor controlling the sharpness/resonance of the filter around the center frequency (CF). For bandpass filters, higher Q increases the gain at the center frequency. Must be in the range `[ma.EPSILON, ma.MAX]`

Test

```
ve = library("vaeffects.lib");
os = library("oscillators.lib");
sallenKey2ndOrderBPF_test = os.osc(330)
  : ve.sallenKey2ndOrderBPF(
      hslider("sallenKey2ndOrderBPF:normFreq", 0.3, 0, 1, 0.001),
      hslider("sallenKey2ndOrderBPF:Q", 1.5, 0.1, 10, 0.1)
    );
```

`(ve.)sallenKey2ndOrderHPF`

Sallen-Key 2nd order highpass filter (see description above). Specialize the generic implementation: keep the third HPF output, the compiler will only generate the needed code.

Usage

```
_ : sallenKey2ndOrderHPF(normFreq,Q) : _
```

Where:

- `normFreq`: normalized frequency (0-1)
- `Q`: quality factor controlling the sharpness/resonance of the filter around the center frequency (CF). For bandpass filters, higher Q increases the gain at the center frequency. Must be in the range `[ma.EPSILON, ma.MAX]`

Test

```
ve = library("vaeffects.lib");
os = library("oscillators.lib");
sallenKey2ndOrderHPF_test = os.osc(330)
  : ve.sallenKey2ndOrderHPF(
      hslider("sallenKey2ndOrderHPF:normFreq", 0.3, 0, 1, 0.001),
      hslider("sallenKey2ndOrderHPF:Q", 0.8, 0.1, 10, 0.1)
    );
```

Vicanek's Matched (Decramped) Second-Order Filters

Vicanek's Matched (Decramped) Second-Order Filters.

This collection implements high-quality, double-precision second-order filters based on the work of Vicanek, offering improved frequency accuracy and dynamic response over traditional biquads—especially near Nyquist.

Standard digital filter designs (like bilinear-transformed biquads) suffer from frequency warping, which distorts the placement of poles and zeros. Vicanek's method, detailed in his paper *"Matched Second Order Digital Filters"*, proposes a set of matched filter formulas that eliminate such warping, preserving the intended analog-like behavior and frequency response.

The filters provided here include:

- `biquad` — generic difference equation implementation

- `lowpass2Matched` — second-order lowpass with resonance

- `highpass2Matched` — second-order highpass with resonance

- `bandpass2Matched` — second-order bandpass with resonance

- `peaking2Matched` — second-order peaking EQ
- `lowshelf2Matched` – second-order Butterworth lowshelf
- `highshelf2Matched` – second-order Butterworth highshelf

Each filter relies on carefully derived coefficient formulas that guarantee accurate placement of the frequency response peak and preserve Q and gain behavior.

Note: These filters require **double-precision** support.

References

- Vicanek, M. (2016) *Matched Second Order Digital Filters*
- https://www.vicanek.de/articles/BiquadFits.pdf

`(ve.)biquad`

Basic biquad section implementing the difference equation: `y[n] = b0 * x[n] + b1 * x[n-1] + b2 * x[n-2] - a1 * y[n-1] - a2 * y[n-2]`

Usage:

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

Where:

- `b0, b1, b2, a1, a2` are the coefficients of the difference equation above

Test

```
ve = library("vaeffects.lib");
os = library("oscillators.lib");
biquad_test = os.osc(440)
  : ve.biquad(0.5, 0.3, 0.2, -0.3, 0.2);
```

`(ve.)lowpass2Matched`

Vicanek's decramped second-order resonant lowpass filter.

Note: These filters require **double-precision** support.

Usage:

```
_ : lowpass2Matched(CF, Q) : _
```

Where:

- `CF`: cutoff frequency in Hz
- `Q`: resonance linear amplitude

Test

```
ve = library("vaeffects.lib");
os = library("oscillators.lib");
lowpass2Matched_test = os.osc(440)
  : ve.lowpass2Matched(
      hslider("lowpass2Matched:CF", 1000, 50, 5000, 1),
      hslider("lowpass2Matched:Q", 0.707, 0.1, 5, 0.01)
    );
```

`(ve.)highpass2Matched`

Vicanek's decramped second-order resonant highpass filter.

Note: These filters require **double-precision** support.

Usage:

```
_ : highpass2Matched(CF, Q) : _
```

Where:

- `CF`: cutoff frequency in Hz
- `Q`: resonance linear amplitude

Test

```
ve = library("vaeffects.lib");
os = library("oscillators.lib");
highpass2Matched_test = os.osc(440)
  : ve.highpass2Matched(
      hslider("highpass2Matched:CF", 500, 50, 5000, 1),
      hslider("highpass2Matched:Q", 0.707, 0.1, 5, 0.01)
    );
```

`(ve.)bandpass2Matched`

Vicanek's decramped second-order resonant bandpass filter.

Note: These filters require **double-precision** support.

Usage:

```
_ : bandpass2Matched(CF, Q) : _
```

Where:

- `CF`: cutoff frequency in Hz
- `Q`: peak width

Test

```
ve = library("vaeffects.lib");
os = library("oscillators.lib");
bandpass2Matched_test = os.osc(440)
  : ve.bandpass2Matched(
      hslider("bandpass2Matched:CF", 1200, 50, 5000, 1),
      hslider("bandpass2Matched:Q", 2.0, 0.1, 10, 0.01)
    );
```

(ve.)peaking2Matched

Vicanek's decramped second-order resonant bandpass filter.

> **Note:** These filters require **double-precision** support.

Usage:

```
_ : peaking2Matched(G, CF, Q) : _
```

Where:

- `G`: peak linear amplitude
- `CF`: cutoff frequency in Hz
- `Q`: peak width

Test

```
ve = library("vaeffects.lib");
os = library("oscillators.lib");
peaking2Matched_test = os.osc(440)
  : ve.peaking2Matched(
      hslider("peaking2Matched:G", 1.5, 0.1, 4, 0.01),
      hslider("peaking2Matched:CF", 1000, 50, 5000, 1),
      hslider("peaking2Matched:Q", 2.0, 0.1, 10, 0.01)
    );
```

(ve.)lowshelf2Matched

Vicanek's decramped second-order Butterworth lowshelf filter.

Note: These filters require **double-precision** support.

Usage:

```
_ : lowshelf2Matched(G, CF) : _
```

Where:

- `G`: shelf linear amplitude
- `CF`: cutoff frequency in Hz

Test

```
ve = library("vaeffects.lib");
os = library("oscillators.lib");
lowshelf2Matched_test = os.osc(330)
  : ve.lowshelf2Matched(
      hslider("lowshelf2Matched:G", 1.5, 0.5, 4, 0.01),
      hslider("lowshelf2Matched:CF", 500, 50, 5000, 1)
    );
```

(ve.)highshelf2Matched

Vicanek's decramped second-order Butterworth highshelf filter.

Note: These filters require **double-precision** support.

Usage:

```
_ : highshelf2Matched(G, CF) : _
```

Where:

- `G`: shelf linear amplitude
- `CF`: cutoff frequency in Hz

Test

```
ve = library("vaeffects.lib");
os = library("oscillators.lib");
highshelf2Matched_test = os.osc(330)
  : ve.highshelf2Matched(
      hslider("highshelf2Matched:G", 1.5, 0.5, 4, 0.01),
      hslider("highshelf2Matched:CF", 1500, 50, 10000, 1)
    );
```

Effects

(ve.)wah4

Wah effect, 4th order. `wah4` is a standard Faust function.

Usage

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

Where:

- `fr`: resonance frequency in Hz

Test

```
ve = library("vaeffects.lib");
os = library("oscillators.lib");
wah4_test = os.osc(220)
  : ve.wah4(
      hslider("wah4:freq", 800, 200, 2000, 1)
    );
```

References

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

(ve.)autowah

Auto-wah effect. `autowah` is a standard Faust function.

Usage

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

Where:

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

Test

```
ve = library("vaeffects.lib");
os = library("oscillators.lib");
autowah_test = os.osc(220)
  : ve.autowah(
      hslider("autowah:level", 0.7, 0, 1, 0.01)
    );
```

`(ve.)crybaby`

Digitized CryBaby wah pedal. `crybaby` is a standard Faust function.

Usage

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

Where:

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

Test

```
ve = library("vaeffects.lib");
os = library("oscillators.lib");
crybaby_test = os.osc(220)
  : ve.crybaby(
      hslider("crybaby:wah", 0.3, 0, 1, 0.01)
    );
```

References

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

`(ve.)vocoder`

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

Usage

```
_ : vocoder(nBands,att,rel,BWRatio,source,excitation) : _
```

Where:

- `nBands`: Number of vocoder bands
- `att`: Attack time in seconds
- `rel`: Release time in seconds
- `BWRatio`: Coefficient to adjust the bandwidth of each band (0.1 - 2)
- `source`: Modulation signal
- `excitation`: Excitation/Carrier signal

Test

```
ve = library("vaeffects.lib");
no = library("noises.lib");
os = library("oscillators.lib");
vocoder_test = (no.noise, os.osc(220))
  : ve.vocoder(
      8,
      hslider("vocoder:att", 0.01, 0.001, 0.1, 0.001),
      hslider("vocoder:rel", 0.1, 0.01, 0.5, 0.01),
      hslider("vocoder:BWRatio", 1.0, 0.5, 1.5, 0.01)
    );
```

version.lib

Semantic versioning for the Faust libraries. Its official prefix is `vl`.

References

- https://github.com/grame-cncm/faustlibraries/blob/master/version.lib

---

`(vl.)version`

Return the version number of the Faust standard libraries as a MAJOR, MINOR, PATCH versioning triplet.

Usage

```
version : _,_,_
```

wdmodels.lib

A library of basic adaptors and methods to help construct Wave Digital Filter models in Faust. Its official prefix is `wd`.

The WDM library is organized into 8 sections:

- Algebraic One Port Adaptors
- Reactive One Port Adaptors
- Nonlinear One Port Adaptors
- Two Port Adaptors
- Three Port Adaptors
- R-Type Adaptors
- Node Creating Functions
- Model Building Functions

Library ReadMe

This library is intended for use for creating Wave Digital (WD) based models of audio circuitry for real-time audio processing within the Faust programming language. The goal is to provide a framework to create real-time virtual-analog audio effects and synthesizers using WD models without the use of C++. Furthermore, we seek to provide access to the technique of WD modeling to those without extensive knowledge of advanced digital signal processing techniques. Finally, we hope to provide a library which can integrate with all aspects of Faust, thus creating a platform for virtual circuit bending. The library itself is written in Faust to maintain portability.

This library is heavily based on Kurt Werner's Dissertation, "Virtual Analog Modeling of Audio Circuitry Using Wave Digital Filters." I have tried to maintain consistent notation between the adaptors appearing within thesis and my adaptor code. The majority of the adaptors found in chapter 1 and chapter 3 are currently supported.

For inquires about use of this library in a commercial product, please contact dirk [dot] roosenburg [dot] 30 [at] gmail [dot] com. This documentation is taken directly from the readme. Please refer to it for a more updated version.

Many of the more in depth comments within the library include jargon. I plan to create videos detailing the theory of WD models. For now I recommend Kurt Werner's PhD, Virtual analog modeling of Audio circuitry using Wave Digital Filters.
I have tried to maintain consistent syntax and notation to the thesis. This library currently includes the majority of the adaptors covered in chapter 1 and some from chapter 3.

Using this Library

Use of this library expects some level of familiarity with WDF techniques, especially simplification and decomposition of electronic circuits into WDF connection trees. I plan to create video to cover both these techniques and use of the library.

Quick Start

To get a quick overview of the library, start with the `secondOrderFilters.dsp` code found in examples. Note that the `wdmodels.lib` library is now embedded in the online Faust IDE.

A Simple RC Filter Model

Creating a model using this library consists fo three steps. First, declare a set of components. Second, model the relationship between them using a tree. Finally, build the tree using the libraries build functions.

First, a set of components is declared using adaptors from the library. This list of components is created based on analysis of the circuit using WDF techniques, though generally each circuit element (resistor, capacitor, diode, etc.) can be expected to appear within the component set. For example, first order RC lowpass filter would require an unadapted voltage source, a 47k resistor, and a 10nF capacitor which outputs the voltage across itself. These can be declared with:

```
vs1(i) = wd.u_voltage(i, no.noise);
r1(i) = wd.resistor(i, 47*10^3);
c1(i) = wd.capacitor_Vout(i, 10*10^-9);
```

Note that the first argument, i, is left un-parametrized. Components must be declared in this form, as the build algorithm expects to receive adaptors which have exactly one parameter.

Also note that we have chosen to declare a white noise function as the input to our voltage source. We could potentially declare this as a direct input to our model, but to do so is more complicated process which cannot be covered within this tutorial. For information on how to do this see Declaring Model Parameters as Inputs or see various implementations in examples.

Second, the declared components and interconnection/structural adaptors (i.e. series, parallel, etc) are arranged into the connection tree which is produced from performing WD analysis on the modeled circuit. For example, to produce our first order RC lowpass circuit model, the following tree is declared:

```
tree_lowpass = vs1 : wd.series : (r1, c1);
```

For more information on how to represent trees in Faust, see Trees in Faust.

Finally, the tree is built using the the `buildtree` function. To build and compute our first order RC lowpass circuit model, we use:

```
process = wd.buildtree(tree_lowpass);
```

More information about build functions, see Model Building Functions.

Building a Model

After creating a connection tree which consists of WD adaptors, the connection tree must be passed to a build function in order to build the model.

Automatic model building `buildtree(connection_tree)`

The simplest build function for use with basic models. This automatically implements `buildup`, `builddown`, and `buildout` to create a working model. However, it gives minimum control to the user and cannot currently be used on trees which have parameters declared as inputs.

Manual model building Wave Digital Filters are an explicit state-space model, meaning they use a previous system state in order to calculate the current output. This is achieved in Faust by using a single global feedback operator. The models feed-forward terms are generated using `builddown` and the models feedback terms are generated using `buildup`. Thus, the most common model implementation (the method used by `buildtree`) is:

```
builddown(connection_tree)~buildup(connection_tree) : buildout(connection_tree)
```

Since the `~` operator in Faust will leave feedback terms hanging as outputs, `buildout` is a function provided for convenience. It automatically truncates the hanging outputs by identifying leaf components which have an intended output and generating an output matrix.

Building the model manually allows for greater user control and is often very helpful in testing. Also provided for testing are the `getres` and `parres` functions, which can be used to determine the upward-facing port resistance of an element.

Declaring Model Parameters as Inputs

When possible, parameters of components should be declared explicitly, meaning they are dependent on a function with no inputs. This might be something as simple as integer(declaring a static component), a function dependent on a UI input (declaring a component with variable value), or even a time-dependent function like an oscillator (declaring an audio input or circuit bending).

However, it is often necessary to declare parameters as input. To achieve this there are two possible methods. The first and recommended option is to create a separate model function and declare parameters which will later be implemented as inputs. This allows inputs to be explicitly declared as component parameters. For example, one might use:

```
model(in1) = buildtree(tree)
with {
   ...
   vin(i) = wd.u_voltage(i, in1);
   ...
   tree = vin : ...;
};
```

In order to simulate an audio input to the circuit.

Note that the tree and components must be declared inside a `with {...}` statement, or the model's parameters will not be accessible.

The Empty Signal Operator The Empty signal operator, `_` should NEVER be used to declare a parameter as in input in a wave-digital model.

Using it will result on breaking the internal routing of the model and thus breaks the model. Instead, use explicit declaration as shown directly above.

Trees in Faust

Since WD models use connection trees to represent relationships of elements, a comprehensive way to represent trees is critical. As there is no current convention for creating trees in Faust, I've developed a method using the existing series and parallel/list methods in Faust.

The series operator `:` is used to separate parent and child elements. For example the tree:

```
A
|
B
```

is represented by `A : B` in Faust.

To denote a parent element with multiple child elements, simply use a list `(a1, a2, ... an)` of children connected to a single parent. ' For example the tree:

```
  A
 / \
B   C
```

is represented by:

`A : (B, C)`

Finally, for a tree with many levels, simply break the tree into subtrees following the above rules and connect the subtree as if it was an individual node. For example the tree:

```
    A
   / \
  B   C
 /   / \
X   Y   Z
```

can be represented by:

```
B_sub = B : X; //B subtree
C_sub = C : (Y, Z); //C subtree
tree = A : (B_sub, C_sub); //full tree
```

or more simply, using parentheses:

`A : ((B : X), (C : (Y, Z)))` ### How Adaptors are Structured In wave digital filters, adaptors can be described by the form `b = Sa` where `b` is a vector of output waves `b = (b0, b1, b2, ... bn)`, `a` is a vector of input waves`a = (a0, a1, a2, ... an)`, and `S` is an n x n scattering matrix. `S` is dependent on `R`, a list of port resistances `(R0, R1, R2, ... Rn)`.

The output wave vector `b` can be divided into downward-going and upward-going waves (downward-going waves travel down the connection tree, upward-going waves travel up). For adapted adaptors, with the zeroth port being the upward-facing port, the downward-going wave vector is `(b1, b2, ... bn)` and the upward-going wave vector is `(b0)`. For unadapted adaptors, there are no upward-going waves, so the downward-going wave vector is simply `b = (b0, b1, b2, ... bn)`.

In order for adaptors to be interpretable by the compiler, they must be structured in a specific way. Each adaptor is divided into three cases by their first parameter. This parameter, while accessible by the user, should only be set by the compiler/builder.

All other parameters are value declarations (for components), inputs (for voltage or current ins), or parameter controls (for potentiometers/variable capacitors/variable inductors).

First case - downward going waves `(0, params) => downward-going(R1, ... Rn, a0, a1, ... an)` outputs: `(b1, b2, ... bn)` this function takes any number of port resistances, the downward going wave, and any number of upward going waves as inputs. These values/waves are used to calculate the downward going waves coming from this adaptor.

Second case `(1, params) => upward-going(R1, ... Rn, a1, ... an)` outputs : `(b0)` this function takes any number of port resistances and any number of upward going waves as inputs. These values/waves are used to calculate the upward going wave coming from this adaptor.

Third case `(2, params) => port-resistance(R1, ... Rn)` outputs: `(R0)` this function takes any number of port resistances as inputs. These values are used to calculate the upward going port resistance of the element.

Unadapted Adaptors Unadapted adaptor's names will always begin `u_` An unadapted adaptor MUST be used as the root of the WD connection tree. Unadapted adaptors can ONLY be used as a root of the WD connection tree. While unadapted adaptors contain all three cases, the second and third are purely structural. Only the first case should contain computational information.

How the Build Functions Work

Expect this section to be added soon! It's currently in progress.

Acknowledgements

Many thanks to Kurt Werner for helping me to understand wave digital filter models. Without his publications and consultations, the library would not exist. Thanks also to my advisors, Rob Owen and Eli Stine whose input was critical to the development of the library. Finally, thanks to Romain Michon,

Stephane Letz, and the Faust Slack for contributing to testing, development, and inspiration when creating the library.

References

- https://github.com/grame-cncm/faustlibraries/blob/master/wdmodels.lib

Algebraic One Port Adaptors

(wd.)resistor

Adapted Resistor.

A basic node implementing a resistor for use within Wave Digital Filter connection trees.

It should be used as a leaf/terminating element of the connection tree.

Usage

```
r1(i) = resistor(i, R);
buildtree( A : r1 );
```

Where:

- `i`: index used by model-building functions. Should never be user declared.
- `R` : Resistance/Impedance of the resistor being modeled in Ohms.

Test

```
wd = library("wdmodels.lib");
os = library("oscillators.lib");

drive = os.osc(220);

vsrc(i) = wd.u_voltage(i, drive);
series_node(i) = wd.series(i);
res_leaf(i) = wd.resistor(i, 1000);
probe(i) = wd.resistor_Vout(i, 1000);

resistor_test = wd.buildtree(vsrc : (series_node : (res_leaf, probe)));
```

Note: the adaptor must be declared as a separate function before integration into the connection tree. Correct implementation is shown above.

References K. Werner, "Virtual Analog Modeling of Audio Circuitry Using Wave Digital Filters", 1.2.1

(wd.)resistor_Vout

Adapted Resistor + voltage Out.

A basic adaptor implementing a resistor for use within Wave Digital Filter connection trees.

It should be used as a leaf/terminating element of the connection tree. The resistor will also pass the voltage across itself as an output of the model.

Usage

```
rout(i) = resistor_Vout(i, R);
buildtree( A : rout ) : _
```

Where:

- `i`: index used by model-building functions. Should never be user declared.
- `R` : Resistance/Impedance of the resistor being modeled in Ohms.

Test

```
wd = library("wdmodels.lib");
os = library("oscillators.lib");

drive = os.osc(220);

vsrc(i) = wd.u_voltage(i, drive);
series_node(i) = wd.series(i);
res_probe(i) = wd.resistor_Vout(i, 820);
res_load(i) = wd.resistor(i, 1800);

resistor_Vout_test = wd.buildtree(vsrc : (series_node : (res_probe, res_load)));
```

Note: the adaptor must be declared as a separate function before integration into the connection tree. Correct implementation is shown above.

References K. Werner, "Virtual Analog Modeling of Audio Circuitry Using Wave Digital Filters", 1.2.1

(wd.)resistor_Iout

Resistor + current Out.

A basic adaptor implementing a resistor for use within Wave Digital Filter connection trees.

It should be used as a leaf/terminating element of the connection tree. The resistor will also pass the current through itself as an output of the model.

Usage

```
rout(i) = resistor_Iout(i, R);
buildtree( A : rout ) : _
```

Where:

- `i`: index used by model-building functions. Should never be user declared.
- `R` : Resistance/Impedance of the resistor being modeled in Ohms.

Test

```
wd = library("wdmodels.lib");
os = library("oscillators.lib");

drive = os.osc(220);

vsrc(i) = wd.u_voltage(i, drive);
series_node(i) = wd.series(i);
current_probe(i) = wd.resistor_Iout(i, 1000);
load(i) = wd.resistor_Vout(i, 1500);

resistor_Iout_test = wd.buildtree(vsrc : (series_node : (current_probe, load)));
```

Note: the adaptor must be declared as a separate function before integration into the connection tree. Correct implementation is shown above.

References K. Werner, "Virtual Analog Modeling of Audio Circuitry Using Wave Digital Filters", 1.2.1

(wd.)u_voltage

Unadapted Ideal Voltage Source.

An adaptor implementing an ideal voltage source within Wave Digital Filter connection trees.

It should be used as the root/top element of the connection tree. Can be used for either DC (constant) or AC (signal) voltage sources.

Usage

```
v1(i) = u_Voltage(i, ein);
buildtree( v1 : B );
```

Where:

- `i`: index used by model-building functions. Should never be user declared.
- `ein` : Voltage/Potential across ideal voltage source in Volts

Test

```
wd = library("wdmodels.lib");
os = library("oscillators.lib");

drive = os.osc(330);

vsrc(i) = wd.u_voltage(i, drive);
series_node(i) = wd.series(i);
branch_a(i) = wd.resistor(i, 1200);
branch_b(i) = wd.resistor_Vout(i, 2200);

u_voltage_test = wd.buildtree(vsrc : (series_node : (branch_a, branch_b)));
```

Note: only usable as the root of a tree. The adaptor must be declared as a separate function before integration into the connection tree. Correct implementation is shown above.

References K. Werner, "Virtual Analog Modeling of Audio Circuitry Using Wave Digital Filters", 1.2.2

`(wd.)u_current`

Unadapted Ideal Current Source.

An unadapted adaptor implementing an ideal current source within Wave Digital Filter connection trees.

It should be used as the root/top element of the connection tree. Can be used for either DC (constant) or AC (signal) current sources.

Usage

```
i1(i) = u_current(i, jin);
buildtree( i1 : B );
```

Where:

- `i`: index used by model-building functions. Should never be user declared.

- `jin` : Current through the ideal current source in Amps

Test

```
wd = library("wdmodels.lib");
os = library("oscillators.lib");

drive = os.osc(110);

isrc(i) = wd.u_current(i, drive);
parallel_node(i) = wd.parallel(i);
branch_a(i) = wd.resistor(i, 560);
branch_b(i) = wd.resistor_Vout(i, 2200);

u_current_test = wd.buildtree(isrc : (parallel_node : (branch_a, branch_b)));
```

Note: only usable as the root of a tree. The adaptor must be declared as a separate function before integration into the connection tree. Correct implementation is shown above.

References K. Werner, "Virtual Analog Modeling of Audio Circuitry Using Wave Digital Filters", 1.2.3

(wd.)resVoltage

Adapted Resistive Voltage Source.

An adaptor implementing a resistive voltage source within Wave Digital Filter connection trees.

It should be used as a leaf/terminating element of the connection tree. It is comprised of an ideal voltage source in series with a resistor. Can be used for either DC (constant) or AC (signal) voltage sources.

Usage

```
v1(i) = resVoltage(i, R, ein);
buildtree( A : v1 );
```

Where:

- `i`: index used by model-building functions. Should never be user declared
- `R` : Resistance/Impedance of the series resistor in Ohms
- `ein` : Voltage/Potential of the ideal voltage source in Volts

Test

```
wd = library("wdmodels.lib");
os = library("oscillators.lib");

drive = os.osc(440);

vsrc(i) = wd.u_voltage(i, drive);
series_node(i) = wd.series(i);
branch_source(i) = wd.resVoltage(i, 1000, 0.5);
probe(i) = wd.resistor_Vout(i, 1800);

resVoltage_test = wd.buildtree(vsrc : (series_node : (branch_source, probe)));
```

Note: the adaptor must be declared as a separate function before integration into the connection tree. Correct implementation is shown above.

References K. Werner, "Virtual Analog Modeling of Audio Circuitry Using Wave Digital Filters", 1.2.4

`(wd.)resVoltage_Vout`

Adapted Resistive Voltage Source + voltage output.

An adaptor implementing an adapted resistive voltage source within Wave Digital Filter connection trees.

It should be used as a leaf/terminating element of the connection tree. It is comprised of an ideal voltage source in series with a resistor. Can be used for either DC (constant) or AC (signal) voltage sources. The resistive voltage source will also pass the voltage across it as an output of the model.

Usage

```
vout(i) = resVoltage_Vout(i, R, ein);
buildtree( A : vout ) : _
```

Where:

- `i`: index used by model-building functions. Should never be user declared
- `R` : Resistance/Impedance of the series resistor in Ohms
- `ein` : Voltage/Potential across ideal voltage source in Volts

Test

```
wd = library("wdmodels.lib");
os = library("oscillators.lib");
```

```
drive = os.osc(330);

vsrc(i) = wd.u_voltage(i, drive);
series_node(i) = wd.series(i);
branch_source(i) = wd.resVoltage_Vout(i, 1500, 0.3);
load(i) = wd.resistor(i, 2200);

resVoltage_Vout_test = wd.buildtree(vsrc : (series_node : (branch_source, load)));
```

Note: the adaptor must be declared as a separate function before integration into the connection tree. Correct implementation is shown above.

References K. Werner, "Virtual Analog Modeling of Audio Circuitry Using Wave Digital Filters", 1.2.4

(wd.)u_resVoltage

Unadapted Resistive Voltage Source.

An unadapted adaptor implementing a resistive voltage source within Wave Digital Filter connection trees.

It should be used as the root/top element of the connection tree. It is comprised of an ideal voltage source in series with a resistor. Can be used for either DC (constant) or AC (signal) voltage sources.

Usage

```
v1(i) = u_resVoltage(i, R, ein);
buildtree( v1 : B );
```

Where:

- `i`: index used by model-building functions. Should never be user declared
- `R` : Resistance/Impedance of the series resistor in Ohms
- `ein` : Voltage/Potential across ideal voltage source in Volts

Test

```
wd = library("wdmodels.lib");
os = library("oscillators.lib");

drive = os.osc(220);

root(i) = wd.u_resVoltage(i, 1800, drive);
series_node(i) = wd.series(i);
branch_a(i) = wd.resistor(i, 1500);
```

```
branch_b(i) = wd.resistor_Vout(i, 2200);

u_resVoltage_test = wd.buildtree(root : (series_node : (branch_a, branch_b)));
```

Note: only usable as the root of a tree. The adaptor must be declared as a separate function before integration into the connection tree. Correct implementation is shown above.

References K. Werner, "Virtual Analog Modeling of Audio Circuitry Using Wave Digital Filters", 1.2.4

`(wd.)resCurrent`

Adapted Resistive Current Source.

An adaptor implementing a resistive current source within Wave Digital Filter connection trees.

It should be used as a leaf/terminating element of the connection tree. It is comprised of an ideal current source in parallel with a resistor. Can be used for either DC (constant) or AC (signal) current sources.

Usage

```
i1(i) = resCurrent(i, R, jin);
buildtree( A : i1 );
```

Where:

- `i`: index used by model-building functions. Should never be user declared.
- `R` : Resistance/Impedance of the parallel resistor in Ohms
- `jin` : Current through the ideal current source in Amps

Test

```
wd = library("wdmodels.lib");
os = library("oscillators.lib");

drive = os.osc(110);

root(i) = wd.u_current(i, drive);
parallel_node(i) = wd.parallel(i);
source_branch(i) = wd.resCurrent(i, 2200, 0.15);
probe(i) = wd.resistor_Vout(i, 1500);

resCurrent_test = wd.buildtree(root : (parallel_node : (source_branch, probe)));
```

Note: the adaptor must be declared as a separate function before integration into the connection tree. Correct implementation is shown above.

References K. Werner, "Virtual Analog Modeling of Audio Circuitry Using Wave Digital Filters", 1.2.5

(wd.)u_resCurrent

Unadapted Resistive Current Source.

An unadapted adaptor implementing a resistive current source within Wave Digital Filter connection trees.

It should be used as the root/top element of the connection tree. It is comprised of an ideal current source in parallel with a resistor. Can be used for either DC (constant) or AC (signal) current sources.

Usage

```
i1(i) = u_resCurrent(i, R, jin);
buildtree( i1 : B );
```

Where:

- `i`: index used by model-building functions. Should never be user declared.
- `R` : Resistance/Impedance of the series resistor in Ohms
- `jin` : Current through the ideal current source in Amps

Test

```
wd = library("wdmodels.lib");
os = library("oscillators.lib");

drive = os.osc(150);

root(i) = wd.u_resCurrent(i, 2000, drive);
parallel_node(i) = wd.parallel(i);
branch_a(i) = wd.resistor(i, 1200);
branch_b(i) = wd.resistor_Vout(i, 1800);

u_resCurrent_test = wd.buildtree(root : (parallel_node : (branch_a, branch_b)));
```

Note: only usable as the root of a tree. The adaptor must be declared as a separate function before integration into the connection tree. Correct implementation is shown above.

References K. Werner, "Virtual Analog Modeling of Audio Circuitry Using Wave Digital Filters", 1.2.5

(wd.)u_switch

Unadapted Ideal Switch.

An unadapted adaptor implementing an ideal switch for Wave Digital Filter connection trees.

It should be used as the root/top element of the connection tree

Usage

```
s1(i) = u_resCurrent(i, lambda);
buildtree( s1 : B );
```

Where:

- `i`: index used by model-building functions. Should never be user declared.
- `lambda` : switch state control. -1 for closed switch, 1 for open switch.

Test

```
wd = library("wdmodels.lib");
os = library("oscillators.lib");

drive = os.osc(330);
lambda = hslider("u_switch:lambda", -1, -1, 1, 0.01);

root(i) = wd.u_switch(i, lambda);
series_node(i) = wd.series(i);
branch_a(i) = wd.resistor(i, 1000);
branch_b(i) = wd.resistor_Vout(i, 2200);

u_switch_test = wd.buildtree(root : (series_node : (branch_a, branch_b)));
```

Note: only usable as the root of a tree. The adaptor must be declared as a separate function before integration into the connection tree. Correct implementation is shown above.

References K. Werner, "Virtual Analog Modeling of Audio Circuitry Using Wave Digital Filters", 1.2.8

Reactive One Port Adaptors

(wd.)capacitor

Adapted Capacitor.

A basic adaptor implementing a capacitor for use within Wave Digital Filter connection trees.

It should be used as a leaf/terminating element of the connection tree. This capacitor model was digitized using the bi-linear transform.

Usage

```
c1(i) = capacitor(i, R);
buildtree( A : c1 ) : _
```

Where:

- `i`: index used by model-building functions. Should never be user declared.
- `R` : Capacitance/Impedance of the capacitor being modeled in Farads.

Test

```
wd = library("wdmodels.lib");
os = library("oscillators.lib");

drive = os.osc(440);

vsrc(i) = wd.u_voltage(i, drive);
series_node(i) = wd.series(i);
cap_branch(i) = wd.capacitor(i, 1e-7);
probe(i) = wd.resistor_Vout(i, 1800);

capacitor_test = wd.buildtree(vsrc : (series_node : (cap_branch, probe)));
```

Note: the adaptor must be declared as a separate function before integration into the connection tree. Correct implementation is shown above.

References K. Werner, "Virtual Analog Modeling of Audio Circuitry Using Wave Digital Filters", 1.3.1

(wd.)capacitor_Vout

Adapted Capacitor + voltage out.

A basic adaptor implementing a capacitor for use within Wave Digital Filter connection trees.

It should be used as a leaf/terminating element of the connection tree. The capacitor will also pass the voltage across itself as an output of the model. This capacitor model was digitized using the bi-linear transform.

Usage

```
cout(i) = capacitor_Vout(i, R);
buildtree( A : cout ) : _
```

Where:

- `i`: index used by model-building functions. Should never be user declared
- `R` : Capacitance/Impedence of the capacitor being modeled in Farads

Test

```
wd = library("wdmodels.lib");
os = library("oscillators.lib");

drive = os.osc(330);

vsrc(i) = wd.u_voltage(i, drive);
series_node(i) = wd.series(i);
cap_branch(i) = wd.capacitor_Vout(i, 2e-7);
load(i) = wd.resistor(i, 1500);

capacitor_Vout_test = wd.buildtree(vsrc : (series_node : (cap_branch, load)));
```

Note: the adaptor must be declared as a seperate function before integration into the connection tree. Correct implementation is shown above.

References K. Werner, "Virtual Analog Modeling of Audio Circuitry Using Wave Digital Filters", 1.3.1

`(wd.)inductor`

Unadapted Inductor.

A basic adaptor implementing an inductor for use within Wave Digital Filter connection trees.

It should be used as a leaf/terminating element of the connection tree. This inductor model was digitized using the bi-linear transform.

Usage

```
l1(i) = inductor(i, R);
buildtree( A : l1 );
```

Where:

- `i`: index used by model-building functions. Should never be user declared
- `R` : Inductance/Impedance of the inductor being modeled in Henries

Test

```
wd = library("wdmodels.lib");
os = library("oscillators.lib");

drive = os.osc(260);

vsrc(i) = wd.u_voltage(i, drive);
series_node(i) = wd.series(i);
inductive_branch(i) = wd.inductor(i, 0.01);
probe(i) = wd.resistor_Vout(i, 2200);

inductor_test = wd.buildtree(vsrc : (series_node : (inductive_branch, probe)));
```

Note: the adaptor must be declared as a separate function before integration into the connection tree. Correct implementation is shown above.

References K. Werner, "Virtual Analog Modeling of Audio Circuitry Using Wave Digital Filters", 1.3.2

`(wd.)inductor_Vout`

Unadapted Inductor + Voltage out.

A basic adaptor implementing an inductor for use within Wave Digital Filter connection trees.

It should be used as a leaf/terminating element of the connection tree. The inductor will also pass the voltage across itself as an output of the model. This inductor model was digitized using the bi-linear transform.

Usage

```
lout(i) = inductor_Vout(i, R);
buildtree( A : lout ) : _
```

Where:

- `i`: index used by model-building functions. Should never be user declared
- `R` : Inductance/Impedance of the inductor being modeled in Henries

Test

```
wd = library("wdmodels.lib");
os = library("oscillators.lib");

drive = os.osc(280);

vsrc(i) = wd.u_voltage(i, drive);
series_node(i) = wd.series(i);
inductive_branch(i) = wd.inductor_Vout(i, 0.02);
load(i) = wd.resistor(i, 1500);

inductor_Vout_test = wd.buildtree(vsrc : (series_node : (inductive_branch, load)));
```

Note: the adaptor must be declared as a separate function before integration into the connection tree. Correct implementation is shown above.

References K. Werner, "Virtual Analog Modeling of Audio Circuitry Using Wave Digital Filters", 1.3.2

Nonlinear One Port Adaptors

`(wd.)u_idealDiode`

Unadapted Ideal Diode.

An unadapted adaptor implementing an ideal diode for Wave Digital Filter connection trees.

It should be used as the root/top element of the connection tree.

Usage

```
buildtree( u_idealDiode : B );
```

Note: only usable as the root of a tree. Correct implementation is shown above.

Test

```
wd = library("wdmodels.lib");
os = library("oscillators.lib");

diode(i) = wd.u_idealDiode(i);
series_node(i) = wd.series(i);
branch_a(i) = wd.resistor(i, 1200);
branch_b(i) = wd.resistor_Vout(i, 1800);
```

```
u_idealDiode_test = wd.buildtree(diode : (series_node : (branch_a, branch_b)));
```

References K. Werner, "Virtual Analog Modeling of Audio Circuitry Using Wave Digital Filters", 3.2.3

(wd.)u_chua

Unadapted Chua Diode.

An adaptor implementing the chua diode / non-linear resistor within Wave Digital Filter connection trees.

It should be used as the root/top element of the connection tree.

Usage

```
chua1(i) = u_chua(i, G1, G2, V0);
buildtree( chua1 : B );
```

Where:

- `i`: index used by model-building functions. Should never be user declared
- `G1` : resistance parameter 1 of the chua diode
- `G2` : resistance parameter 2 of the chua diode
- `V0` : voltage parameter of the chua diode

Test

```
wd = library("wdmodels.lib");
os = library("oscillators.lib");

chua_node(i) = wd.u_chua(i, 1e-3, 5e-4, 0.2);
series_node(i) = wd.series(i);
branch_a(i) = wd.resistor(i, 1500);
branch_b(i) = wd.resistor_Vout(i, 2200);

u_chua_test = wd.buildtree(chua_node : (series_node : (branch_a, branch_b)));
```

Note: only usable as the root of a tree. The adaptor must be declared as a separate function before integration into the connection tree. Correct implementation is shown above.

References Meerkotter and Scholz, "Digital Simulation of Nonlinear Circuits by Wave Digital Filter Principles"

`(wd.)lambert`

An implementation of the lambert function. It uses Halley's method of iteration to approximate the output. Included in the WD library for use in non-linear diode models. Adapted from K M Brigg's c++ lambert function approximation.

Usage

```
lambert(n, itr) : _
```

Where: * `n`: value at which the lambert function will be evaluated * `itr`: number of iterations before output

Test

```
wd = library("wdmodels.lib");
os = library("oscillators.lib");

lambert_gain = wd.lambert(0.5, 6);
lambert_test = os.osc(220) * lambert_gain;
```

`(wd.)u_diodePair`

Unadapted pair of diodes facing in opposite directions.

An unadapted adaptor implementing two antiparallel diodes for Wave Digital Filter connection trees. The behavior is approximated using Schottkey's ideal diode law.

Usage

```
d1(i) = u_diodePair(i, Is, Vt);
buildtree( d1 : B );
```

Where:

- `i`: index used by model-building functions. Should never be user declared
- `Is` : saturation current of the diodes
- `Vt` : thermal resistances of the diodes

Test

```
wd = library("wdmodels.lib");

u_diodePair_test = wd.u_diodePair(2, 1e-12, 0.025);
```

Note: only usable as the root of a tree. Correct implementation is shown above.

References K. Werner et al. "An Improved and Generalized Diode Clipper Model for Wave Digital Filters"

(wd.)u_diodeSingle

Unadapted single diode.

An unadapted adaptor implementing a single diode for Wave Digital Filter connection trees. The behavior is approximated using Schottkey's ideal diode law.

Usage

```
d1(i) = u_diodeSingle(i, Is, Vt);
buildtree( d1 : B );
```

Where:

- `i`: index used by model-building functions. Should never be user declared
- `Is` : saturation current of the diodes
- `Vt` : thermal resistances of the diodes

Test

```
wd = library("wdmodels.lib");

u_diodeSingle_test = wd.u_diodeSingle(2, 8e-13, 0.026);
```

Note: only usable as the root of a tree. Correct implementation is shown above.

References K. Werner et al. "An Improved and Generalized Diode Clipper Model for Wave Digital Filters"

(wd.)u_diodeAntiparallel

Unadapted set of antiparallel diodes with M diodes facing forwards and N diodes facing backwards.

An unadapted adaptor implementing antiparallel diodes for Wave Digital Filter connection trees. The behavior is approximated using Schottkey's ideal diode law.

Usage

```
d1(i) = u_diodeAntiparallel(i, Is, Vt);
buildtree( d1 : B );
```

Where:

- `i`: index used by model-building functions. Should never be user declared
- `Is` : saturation current of the diodes
- `Vt` : thermal resistances of the diodes

Test

```
wd = library("wdmodels.lib");
u_diodeAntiparallel_test = wd.u_diodeAntiparallel(2, 1e-12, 0.025, 2, 2);
```

Note: only usable as the root of a tree. Correct implementation is shown above.

References K. Werner et al. "An Improved and Generalized Diode Clipper Model for Wave Digital Filters"

Two Port Adaptors

`(wd.)u_parallel2Port`

Unadapted 2-port parallel connection.

An unadapted adaptor implementing a 2-port parallel connection between adaptors for Wave Digital Filter connection trees. Elements connected to this adaptor will behave as if connected in parallel in circuit.

Usage

```
buildtree( u_parallel2Port : (A, B) );
```

Note: only usable as the root of a tree. This adaptor has no user-accessible parameters. Correct implementation is shown above.

Test

```
wd = library("wdmodels.lib");
os = library("oscillators.lib");

root(i) = wd.u_parallel2Port(i);
branch_source(i) = wd.resVoltage_Vout(i, 1500, 0.2 * os.osc(220));
branch_load(i) = wd.resistor(i, 1800);

u_parallel2Port_test = wd.buildtree(root : (branch_source, branch_load));
```

References K. Werner, "Virtual Analog Modeling of Audio Circuitry Using Wave Digital Filters", 1.4.1

(wd.)parallel2Port

Adapted 2-port parallel connection.

An adaptor implementing a 2-port parallel connection between adaptors for Wave Digital Filter connection trees. Elements connected to this adaptor will behave as if connected in parallel in circuit.

Usage

```
buildtree( A : parallel2Port : B );
```

Note: this adaptor has no user-accessible parameters. It should be used within the connection tree with one previous and one forward adaptor. Correct implementation is shown above.

Test

```
wd = library("wdmodels.lib");
os = library("oscillators.lib");

vsrc(i) = wd.u_voltage(i, os.osc(260));
connector(i) = wd.parallel2Port(i);
load(i) = wd.resistor_Vout(i, 1800);

parallel2Port_test = wd.buildtree(vsrc : (connector : load));
```

References K. Werner, "Virtual Analog Modeling of Audio Circuitry Using Wave Digital Filters", 1.4.1

(wd.)u_series2Port

Unadapted 2-port series connection.

An unadapted adaptor implementing a 2-port series connection between adaptors for Wave Digital Filter connection trees. Elements connected to this adaptor will behave as if connected in series in circuit.

Usage

```
buildtree( u_series2Port : (A, B) );
```

Note: only usable as the root of a tree. This adaptor has no user-accessible parameters. Correct implementation is shown above.

Test

```
wd = library("wdmodels.lib");
os = library("oscillators.lib");

root(i) = wd.u_series2Port(i);
branch_source(i) = wd.resVoltage_Vout(i, 1200, 0.25 * os.osc(180));
branch_load(i) = wd.resistor(i, 1800);

u_series2Port_test = wd.buildtree(root : (branch_source, branch_load));
```

References K. Werner, “Virtual Analog Modeling of Audio Circuitry Using Wave Digital Filters”, 1.4.1

(wd.)series2Port

Adapted 2-port series connection.

An adaptor implementing a 2-port series connection between adaptors for Wave Digital Filter connection trees. Elements connected to this adaptor will behave as if connected in series in circuit.

Usage

```
buildtree( A : series2Port : B );
```

Note: this adaptor has no user-accessible parameters. It should be used within the connection tree with one previous and one forward adaptor. Correct implementation is shown above.

Test

```
wd = library("wdmodels.lib");
os = library("oscillators.lib");

vsrc(i) = wd.u_voltage(i, os.osc(200));
connector(i) = wd.series2Port(i);
load(i) = wd.resistor_Vout(i, 2200);

series2Port_test = wd.buildtree(vsrc : (connector : load));
```

References K. Werner, “Virtual Analog Modeling of Audio Circuitry Using Wave Digital Filters”, 1.4.1

(wd.)parallelCurrent

Adapted 2-port parallel connection + ideal current source.

An adaptor implementing a 2-port series connection and internal idealized current source between adaptors for Wave Digital Filter connection trees. This adaptor connects the two connected elements and an additional ideal current source in parallel.

Usage

```
i1(i) = parallelCurrent(i, jin);
buildtree(A : i1 : B);
```

Where:

- `i`: index used by model-building functions. Should never be user declared
- `jin` : Current through the ideal current source in Amps

Test

```
wd = library("wdmodels.lib");
os = library("oscillators.lib");

vsrc(i) = wd.u_voltage(i, os.osc(240));
connector(i) = wd.parallelCurrent(i, 0.1);
load(i) = wd.resistor_Vout(i, 1500);

parallelCurrent_test = wd.buildtree(vsrc : (connector : load));
```

Note: the adaptor must be declared as a separate function before integration into the connection tree. It should be used within a connection tree with one previous and one forward adaptor. Correct implementation is shown above.

References K. Werner, "Virtual Analog Modeling of Audio Circuitry Using Wave Digital Filters", 1.4.2

(wd.)seriesVoltage

Adapted 2-port series connection + ideal voltage source.

An adaptor implementing a 2-port series connection and internal ideal voltage source between adaptors for Wave Digital Filter connection trees. This adaptor connects the two connected adaptors and an additional ideal voltage source in series.

Usage

```
v1(i) = seriesVoltage(i, vin)
buildtree( A : v1 : B );
```

Where:

- `i`: index used by model-building functions. Should never be user declared
- `vin` : voltage across the ideal current source in Volts

Test

```
wd = library("wdmodels.lib");
os = library("oscillators.lib");

vsrc(i) = wd.u_voltage(i, os.osc(210));
connector(i) = wd.seriesVoltage(i, 0.3);
load(i) = wd.resistor_Vout(i, 1500);

seriesVoltage_test = wd.buildtree(vsrc : (connector : load));
```

Note: the adaptor must be declared as a separate function before integration into the connection tree. It should be used within the connection tree with one previous and one forward adaptor.

References K. Werner, "Virtual Analog Modeling of Audio Circuitry Using Wave Digital Filters", 1.4.2

`(wd.)u_transformer`

Unadapted ideal transformer.

An adaptor implementing an ideal transformer for Wave Digital Filter connection trees. The first downward-facing port corresponds to the primary winding connections, and the second downward-facing port to the secondary winding connections.

Usage

```
t1(i) = u_transformer(i, tr);
buildtree(t1 : (A , B));
```

Where:

- `i`: index used by model-building functions. Should never be user declared
- `tr` : the turn ratio between the windings on the primary and secondary coils

Test

```
wd = library("wdmodels.lib");
os = library("oscillators.lib");

root(i) = wd.u_transformer(i, 2.0);
primary(i) = wd.resVoltage_Vout(i, 1500, 0.2 * os.osc(220));
secondary(i) = wd.resistor_Vout(i, 2200);

u_transformer_test = wd.buildtree(root : (primary, secondary));
```

Note: the adaptor must be declared as a separate function before integration into the connection tree. It may only be used as the root of the connection tree with two forward nodes.

References K. Werner, "Virtual Analog Modeling of Audio Circuitry Using Wave Digital Filters", 1.4.3

`(wd.)transformer`

Adapted ideal transformer.

An adaptor implementing an ideal transformer for Wave Digital Filter connection trees. The upward-facing port corresponds to the primary winding connections, and the downward-facing port to the secondary winding connections

Usage

```
t1(i) = transformer(i, tr);
buildtree(A : t1 : B);
```

Where:

- `i`: index used by model-building functions. Should never be user declared
- `tr` : the turn ratio between the windings on the primary and secondary coils

Test

```
wd = library("wdmodels.lib");
os = library("oscillators.lib");

vsrc(i) = wd.u_voltage(i, os.osc(180));
xfmr(i) = wd.transformer(i, 2.5);
load(i) = wd.resistor_Vout(i, 2200);

transformer_test = wd.buildtree(vsrc : (xfmr : load));
```

Note: the adaptor must be declared as a separate function before integration into the connection tree. It should be used within the connection tree with one backward and one forward nodes.

References K. Werner, "Virtual Analog Modeling of Audio Circuitry Using Wave Digital Filters", 1.4.3

(wd.)u_transformerActive

Unadapted ideal active transformer.

An adaptor implementing an ideal transformer for Wave Digital Filter connection trees. The first downward-facing port corresponds to the primary winding connections, and the second downward-facing port to the secondary winding connections.

Usage

```
t1(i) = u_transformerActive(i, gamma1, gamma2);
buildtree(t1 : (A , B));
```

Where:

- `i`: index used by model-building functions. Should never be user declared
- `gamma1` : the turn ratio describing the voltage relationship between the primary and secondary coils
- `gamma2` : the turn ratio describing the current relationship between the primary and secondary coils

Test

```
wd = library("wdmodels.lib");
os = library("oscillators.lib");

root(i) = wd.u_transformerActive(i, 0.9, 0.8);
primary(i) = wd.resVoltage_Vout(i, 1200, 0.18 * os.osc(190));
secondary(i) = wd.resistor_Vout(i, 2200);

u_transformerActive_test = wd.buildtree(root : (primary, secondary));
```

Note: the adaptor must be declared as a separate function before integration into the connection tree. It may only be used as the root of the connection tree with two forward nodes.

References K. Werner, "Virtual Analog Modeling of Audio Circuitry Using Wave Digital Filters", 1.4.3

(wd.)transformerActive

Adapted ideal active transformer.

An adaptor implementing an ideal active transformer for Wave Digital Filter connection trees. The upward-facing port corresponds to the primary winding connections, and the downward-facing port to the secondary winding connections

Usage

```
t1(i) = transformerActive(i, gamma1, gamma2);
buildtree(A : t1 : B);
```

Where:

- `i`: index used by model-building functions. Should never be user declared
- `gamma1` : the turn ratio describing the voltage relationship between the primary and secondary coils
- `gamma2` : the turn ratio describing the current relationship between the primary and secondary coils

Test

```
wd = library("wdmodels.lib");
os = library("oscillators.lib");

vsrc(i) = wd.u_voltage(i, os.osc(175));
xfmr(i) = wd.transformerActive(i, 0.9, 0.8);
load(i) = wd.resistor_Vout(i, 2200);

transformerActive_test = wd.buildtree(vsrc : (xfmr : load));
```

Note: the adaptor must be declared as a separate function before integration into the connection tree. It should be used within the connection tree with two forward nodes.

References K. Werner, "Virtual Analog Modeling of Audio Circuitry Using Wave Digital Filters", 1.4.3

Three Port Adaptors

(wd.)parallel

Adapted 3-port parallel connection.

An adaptor implementing a 3-port parallel connection between adaptors for Wave Digital Filter connection trees. This adaptor is used to connect adaptors simulating components connected in parallel in the circuit.

Usage

```
buildtree( A : parallel : (B, C) );
```

Note: this adaptor has no user-accessible parameters. It should be used within the connection tree with one previous and two forward adaptors.

Test

```
wd = library("wdmodels.lib");
os = library("oscillators.lib");

vsrc(i) = wd.u_voltage(i, os.osc(220));
junction(i) = wd.parallel(i);
branch_a(i) = wd.resistor(i, 1200);
branch_b(i) = wd.resistor_Vout(i, 1800);

parallel_test = wd.buildtree(vsrc : (junction : (branch_a, branch_b)));
```

References K. Werner, “Virtual Analog Modeling of Audio Circuitry Using Wave Digital Filters”, 1.5.1

(wd.)series

Adapted 3-port series connection.

An adaptor implementing a 3-port series connection between adaptors for Wave Digital Filter connection trees. This adaptor is used to connect adaptors simulating components connected in series in the circuit.

Usage

```
tree = A : (series : (B, C));
```

Note: this adaptor has no user-accessible parameters. It should be used within the connection tree with one previous and two forward adaptors.

Test

```
wd = library("wdmodels.lib");
os = library("oscillators.lib");

vsrc(i) = wd.u_voltage(i, os.osc(260));
junction(i) = wd.series(i);
branch_a(i) = wd.resistor(i, 1000);
branch_b(i) = wd.resistor_Vout(i, 2200);

series_test = wd.buildtree(vsrc : (junction : (branch_a, branch_b)));
```

References K. Werner, "Virtual Analog Modeling of Audio Circuitry Using Wave Digital Filters", 1.5.2

R-Type Adaptors

(wd.)u_sixportPassive

Unadapted six-port rigid connection.

An adaptor implementing a six-port passive rigid connection between elements. It implements the simplest possible rigid connection found in the Fender Bassman Tonestack circuit.

Usage

```
tree = u_sixportPassive : (A, B, C, D, E, F));
```

Note: this adaptor has no user-accessible parameters. It should be used within the connection tree with six forward adaptors.

Test

```
wd = library("wdmodels.lib");
os = library("oscillators.lib");

u_sixportPassive_test = (1000, 1200, 1400, 1600, 1800, 2000, os.osc(220), 0, 0, 0, 0, 0, 0)
  : wd.u_sixportPassive(0) : _, !, !, !, !;
```

References K. Werner, "Virtual Analog Modeling of Audio Circuitry Using Wave Digital Filters", 2.1.5

Node Creating Functions

(wd.)genericNode

Function for generating an adapted node from another faust function or scattering matrix.

This function generates a node which is suitable for use in the connection tree structure. `genericNode` separates the function that it is passed into upward-going and downward-going waves.

Usage

```
n1(i) = genericNode(i, scatter, upRes);
```

Where:

- `i`: index used by model-building functions. Should never be user declared
- `scatter` : the function which describes the the node's scattering behavior
- `upRes` : the function which describes the node's upward-facing port-resistance

Test

```
wd = library("wdmodels.lib");
os = library("oscillators.lib");

scatter(a) = -a * 0.3;
upRes = 1400;

node_iout(i) = wd.genericNode_Iout(i, scatter, upRes);
vsrc(i) = wd.u_voltage(i, os.osc(230));
branch(i) = wd.series(i);
load(i) = wd.resistor(i, 1800);

genericNode_Iout_test = wd.buildtree(vsrc : (branch : (node_iout, load)));
```

Test

```
wd = library("wdmodels.lib");
os = library("oscillators.lib");

scatter(a) = -a * 0.4;
upRes = 1600;

node_vout(i) = wd.genericNode_Vout(i, scatter, upRes);
vsrc(i) = wd.u_voltage(i, os.osc(200));
```

```
branch(i) = wd.series(i);
load(i) = wd.resistor(i, 1800);

genericNode_Vout_test = wd.buildtree(vsrc : (branch : (node_vout, load)));
```

Test

```
wd = library("wdmodels.lib");
os = library("oscillators.lib");

scatter(a) = -a * 0.5;
upRes = 1200;

node(i) = wd.genericNode(i, scatter, upRes);
vsrc(i) = wd.u_voltage(i, os.osc(220));
branch(i) = wd.series(i);
probe(i) = wd.resistor_Vout(i, 1800);

genericNode_test = wd.buildtree(vsrc : (branch : (node, probe)));
```

Note: `scatter` must be a function with n inputs, n outputs, and n-1 parameter inputs. input/output 1 will be used as the adapted upward-facing port of the node, ports 2 to n will all be downward-facing. The first input/output pair is assumed to already be adapted - i.e. the output 1 is not dependent on input 1. The parameter inputs will receive the port resistances of the downward-facing ports.

`upRes` must be a function with n-1 parameter inputs and 1 output. The parameter inputs will receive the port resistances of the downward-facing ports. The output should give the upward-facing port resistance of the node based on the upward-facing port resistances of the input.

If used on a leaf element (n=1), the model will automatically introduce a one-sample delay. Thus, the output of the node at sample t based on the input, a[t], should be the output one sample ahead, b[t+1]. This may require transformation of the output signal.

`(wd.)genericNode_Vout`

Function for generating a terminating/leaf node which gives the voltage across itself as a model output.

This function generates a node which is suitable for use in the connection tree structure. It also calculates the voltage across the element and gives it as a model output.

Usage

`n1(i) = genericNode_Vout(i, scatter, upRes);`

Where:

- `i`: index used by model-building functions. Should never be user declared
- `scatter` : the function which describes the the node's scattering behavior
- `upRes` : the function which describes the node's upward-facing port-resistance

Note: `scatter` must be a function with 1 input and 1 output. It should give the output from the node based on the incident wave.

The model will automatically introduce a one-sample delay to the output of the function Thus, the output of the node at sample t based on the input, a[t], should be the output one sample ahead, b[t+1]. This may require transformation of the output signal.

`upRes` must be a function with no inputs and 1 output. The output should give the upward-facing port resistance of the node.

`(wd.)genericNode_Iout`

Function for generating a terminating/leaf node which gives the current through itself as a model output.

This function generates a node which is suitable for use in the connection tree structure. It also calculates the current through the element and gives it as a model output.

Usage

`n1(i) = genericNode_Iout(i, scatter, upRes);`

Where:

- `i`: index used by model-building functions. Should never be user declared
- `scatter` : the function which describes the the node's scattering behavior
- `upRes` : the function which describes the node's upward-facing port-resistance

Note: `scatter` must be a function with 1 input and 1 output. It should give the output from the node based on the incident wave.

The model will automatically introduce a one-sample delay to the output of the function. Thus, the output of the node at sample t based on the input, a[t], should be the output one sample ahead, b[t+1]. This may require transformation of the output signal.

upRes must be a function with no inputs and 1 output. The output should give the upward-facing port resistance of the node.

(wd.)u_genericNode

Function for generating an unadapted node from another Faust function or scattering matrix.

This function generates a node which is suitable for use as the root of the connection tree structure.

Usage

```
n1(i) = u_genericNode(i, scatter);
```

Where:

- i: index used by model-building functions. Should never be user declared
- scatter : the function which describes the the node's scattering behavior

Test

```
wd = library("wdmodels.lib");
os = library("oscillators.lib");

scatter(a) = -a * 0.5;

root(i) = wd.u_genericNode(i, scatter);
branch(i) = wd.series(i);
load_a(i) = wd.resistor(i, 1500);
load_b(i) = wd.resistor_Vout(i, 2200);

u_genericNode_test = wd.buildtree(root : (branch : (load_a, load_b)));
```

Note: scatter must be a function with n inputs, n outputs, and n parameter inputs. each input/output pair will be used as a downward-facing port of the node the parameter inputs will receive the port resistances of the downward-facing ports.

Model Building Functions

(wd.)builddown

Function for building the structure for calculating waves traveling down the WD connection tree.

It recursively steps through the given tree, parametrizes the adaptors, and builds an algorithm. It is used in conjunction with the buildup() function to create a model.

Usage

```
builddown(A : B)~buildup(A : B);
```

Where: `(A : B)` : is a connection tree composed of WD adaptors

Test

```
wd = library("wdmodels.lib");
os = library("oscillators.lib");

vsrc(i) = wd.u_voltage(i, os.osc(220));
branch(i) = wd.series(i);
res_leaf(i) = wd.resistor(i, 1200);
probe(i) = wd.resistor_Vout(i, 1800);
tree = vsrc : (branch : (res_leaf, probe));

builddown_test = wd.builddown(tree) ~ wd.buildup(tree) : wd.buildout(tree);
```

`(wd.)buildup`

Function for building the structure for calculating waves traveling up the WD connection tree.

It recursively steps through the given tree, parametrizes the adaptors, and builds an algorithm. It is used in conjunction with the builddown() function to create a full structure.

Usage

```
builddown(A : B)~buildup(A : B);
```

Where: `(A : B)` : is a connection tree composed of WD adaptors

Test

```
wd = library("wdmodels.lib");
os = library("oscillators.lib");

vsrc(i) = wd.u_voltage(i, os.osc(220));
branch(i) = wd.series(i);
res_leaf(i) = wd.resistor(i, 1200);
probe(i) = wd.resistor_Vout(i, 1800);
```

```
tree = vsrc : (branch : (res_leaf, probe));

buildup_test = wd.builddown(tree) ~ wd.buildup(tree) : wd.buildout(tree);
```

(wd.)getres

Function for determining the upward-facing port resistance of a partial WD connection tree.

It recursively steps through the given tree, parametrizes the adaptors, and builds an algorithm. It is used by the buildup and builddown functions but is also helpful in testing.

Usage

```
getres(A : B)~getres(A : B);
```

Where: `(A : B)` : is a partial connection tree composed of WD adaptors

Test

```
wd = library("wdmodels.lib");
os = library("oscillators.lib");

branch(i) = wd.series(i);
res_leaf(i) = wd.resistor(i, 1200);
probe(i) = wd.resistor_Vout(i, 1800);
subtree = branch : (res_leaf, probe);

getres_value = wd.getres(subtree);
getres_test = os.osc(110) * (1.0/(1.0 + getres_value));
```

Note: This function cannot be used on a complete WD tree. When called on an unadapted adaptor (u_ prefix), it will create errors.

(wd.)parres

Function for determining the upward-facing port resistance of a partial WD connection tree.

It recursively steps through the given tree, parametrizes the adaptors, and builds an algorithm. It is used by the buildup and builddown functions but is also helpful in testing. This function is a parallelized version of `getres`.

Usage

```
parres((A , B))~parres((A , B));
```

Where: `(A , B)` : is a partial connection tree composed of WD adaptors

Test

```
wd = library("wdmodels.lib");
os = library("oscillators.lib");

branchLeft(i) = wd.series(i);
res_left(i) = wd.resistor(i, 1200);
probe_left(i) = wd.resistor(i, 1800);
subtree_left = branchLeft : (res_left, probe_left);

branchRight(i) = wd.parallel(i);
res_right(i) = wd.resistor(i, 1500);
probe_right(i) = wd.resistor(i, 2200);
subtree_right = branchRight : (res_right, probe_right);

parres_test = wd.parres((subtree_left, subtree_right)) : _, !;
```

Note: this function cannot be used on a complete WD tree. When called on an unadapted adaptor (u_ prefix), it will create errors.

`(wd.)buildout`

Function for creating the output matrix for a WD model from a WD connection tree.

It recursively steps through the given tree and creates an output matrix passing only outputs.

Usage

```
buildout( A : B );
```

Where: `(A : B)` : is a connection tree composed of WD adaptors

Test

```
wd = library("wdmodels.lib");
os = library("oscillators.lib");

vsrc(i) = wd.u_voltage(i, os.osc(240));
branch(i) = wd.series(i);
res_leaf(i) = wd.resistor(i, 1200);
```

```
probe(i) = wd.resistor_Vout(i, 1800);
tree = vsrc : (branch : (res_leaf, probe));

buildout_matrix = wd.buildout(tree);
buildout_test = wd.builddown(tree) ~ wd.buildup(tree) : buildout_matrix;
```

(wd.)buildtree

Function for building the DSP model from a WD connection tree structure.

It recursively steps through the given tree, parametrizes the adaptors, and builds the algorithm.

Usage

```
buildtree(A : B);
```

Where: `(A : B)` : a connection tree composed of WD adaptors

Test

```
wd = library("wdmodels.lib");
os = library("oscillators.lib");

vsrc(i) = wd.u_voltage(i, os.osc(220));
branch(i) = wd.series(i);
res_leaf(i) = wd.resistor(i, 1200);
probe(i) = wd.resistor_Vout(i, 1800);
tree = vsrc : (branch : (res_leaf, probe));

buildtree_test = wd.buildtree(tree);
```

webaudio.lib

An implementation of the WebAudio API filters (https://www.w3.org/TR/webaudio/). Its official prefix is `wa`.

This library implement WebAudio filters, using their C++ version as a starting point, taken from Mozilla Firefox implementation.

References

- https://github.com/grame-cncm/faustlibraries/blob/master/webaudio.lib

---

(wa.)lowpass2

Standard second-order resonant lowpass filter with 12dB/octave rolloff. Frequencies below the cutoff pass through, frequencies above it are attenuated.

Usage

```
_ : lowpass2(f0, Q, dtune) : _
```

Where:

- `f0`: cutoff frequency in Hz
- `Q`: the quality factor
- `dtune`: detuning of the frequency in cents

Test

```
wa = library("webaudio.lib");
os = library("oscillators.lib");
lowpass2_test = os.osc(440) : wa.lowpass2(1000, 0.707, 0);
```

References

- https://searchfox.org/mozilla-central/source/dom/media/webaudio/blink/Biquad.cpp#98

(wa.)highpass2

Standard second-order resonant highpass filter with 12dB/octave rolloff. Frequencies below the cutoff are attenuated, frequencies above it pass through.

Usage

```
_ : highpass2(f0, Q, dtune) : _
```

Where:

- `f0`: cutoff frequency in Hz
- `Q`: the quality factor
- `dtune`: detuning of the frequency in cents

Test

```
wa = library("webaudio.lib");
os = library("oscillators.lib");
highpass2_test = os.osc(440) : wa.highpass2(1000, 0.707, 0);
```

References

- https://searchfox.org/mozilla-central/source/dom/media/webaudio/blink/Biquad.cpp#127

(wa.)bandpass2

Standard second-order bandpass filter. Frequencies outside the given range of frequencies are attenuated, the frequencies inside it pass through.

Usage

```
_ : bandpass2(f0, Q, dtune) : _
```

Where:

- `f0`: cutoff frequency in Hz
- `Q`: the quality factor
- `dtune`: detuning of the frequency in cents

Test

```
wa = library("webaudio.lib");
os = library("oscillators.lib");
bandpass2_test = os.osc(440) : wa.bandpass2(1000, 1, 0);
```

References

- https://searchfox.org/mozilla-central/source/dom/media/webaudio/blink/Biquad.cpp#334

(wa.)notch2

Standard notch filter, also called a band-stop or band-rejection filter. It is the opposite of a bandpass filter: frequencies outside the give range of frequencies pass through, frequencies inside it are attenuated.

Usage

```
_ : notch2(f0, Q, dtune) : _
```

Where:

- `f0`: cutoff frequency in Hz
- `Q`: the quality factor
- `dtune`: detuning of the frequency in cents

Test

```
wa = library("webaudio.lib");
os = library("oscillators.lib");
notch2_test = os.osc(440) : wa.notch2(1000, 1, 0);
```

References

- https://searchfox.org/mozilla-central/source/dom/media/webaudio/blink/Biquad.cpp#301

(wa.)allpass2

Standard second-order allpass filter. It lets all frequencies through, but changes the phase-relationship between the various frequencies.

Usage

```
_ : allpass2(f0, Q, dtune) : _
```

Where:

- `f0`: cutoff frequency in Hz
- `Q`: the quality factor
- `dtune`: detuning of the frequency in cents

Test

```
wa = library("webaudio.lib");
os = library("oscillators.lib");
allpass2_test = os.osc(440) : wa.allpass2(1000, 1, 0);
```

References

- https://searchfox.org/mozilla-central/source/dom/media/webaudio/blink/Biquad.cpp#268

(wa.)peaking2

Frequencies inside the range get a boost or an attenuation, frequencies outside it are unchanged.

Usage

```
_ : peaking2(f0, gain, Q, dtune) : _
```

Where:

- `f0`: cutoff frequency in Hz
- `gain`: the gain in dB
- `Q`: the quality factor
- `dtune`: detuning of the frequency in cents

Test

```
wa = library("webaudio.lib");
os = library("oscillators.lib");
peaking2_test = os.osc(440) : wa.peaking2(1000, 3, 1, 0);
```

References

- https://searchfox.org/mozilla-central/source/dom/media/webaudio/blink/Biquad.cpp#233

`(wa.)lowshelf2`

Standard second-order lowshelf filter. Frequencies lower than the frequency get a boost, or an attenuation, frequencies over it are unchanged.

```
_ : lowshelf2(f0, gain, dtune) : _
```

Where:

- `f0`: cutoff frequency in Hz
- `gain`: the gain in dB
- `dtune`: detuning of the frequency in cents

Test

```
wa = library("webaudio.lib");
os = library("oscillators.lib");
lowshelf2_test = os.osc(440) : wa.lowshelf2(500, 6, 0);
```

References

- https://searchfox.org/mozilla-central/source/dom/media/webaudio/blink/Biquad.cpp#169

`(wa.)highshelf2`

Standard second-order highshelf filter. Frequencies higher than the frequency get a boost or an attenuation, frequencies lower than it are unchanged.

```
_ : highshelf2(f0, gain, dtune) : _
```

Where:

- `f0`: cutoff frequency in Hz
- `gain`: the gain in dB
- `dtune`: detuning of the frequency in cents

Test

```
wa = library("webaudio.lib");
os = library("oscillators.lib");
highshelf2_test = os.osc(440) : wa.highshelf2(2000, -6, 0);
```

References

- https://searchfox.org/mozilla-central/source/dom/media/webaudio/blink/Biquad.cpp#201