

Faust Standard Libraries

Contents

Faust Libraries	13
Using the Faust Libraries	14
Contributing	15
New Functions	15
New Libraries	16
General Organization	16
Coding Conventions	17
Documentation	17
Library Import	18
“Demo” Functions	19
“Standard” Functions	19
Copyright / License	19
Standard Functions	19
Analysis Tools	20
Basic Elements	20
Conversion	20
Effects	21
Envelope Generators	21
Filters	21
Oscillators/Sound Generators	22
Synths	22
analyzers.lib	23
Amplitude Tracking	23
amp_follower	23
amp_follower_ud	23
Spectrum-Analyzers	24
mth_octave_analyzer	25
Mth-Octave Spectral Level	25
mth_octave_spectral_level6e	25
[third half]_octave_[analyzer filterbank]	26
Arbitrary-Crossover Filter-Banks and Spectrum Analyzers	26
analyzer	26

amp_follower_ar	27
basics.lib	27
Conversion Tools	27
samp2sec	27
sec2samp	28
db2linear	28
linear2db	28
lin2LogGain	29
log2LinGain	29
tau2pole	29
pole2tau	30
midikey2hz	30
pianokey2hz	30
hz2pianokey	30
Counters and Time/Tempo Tools	31
countdown	31
countup	31
sweep	32
time	32
tempo	32
period	32
pulse	33
pulsen	33
beat	33
pulse_countup	34
pulse_countdown	34
pulse_countup_loop	34
pulse_countdown_loop	35
Array Processing/Pattern Matching	35
count	35
take	35
subseq	36
Selectors (Conditions)	36
if	36
selector	37
selectn	37
select2stereo	37
Other	38
latch	38
sAndH	38
downSample	38
peakhold	39
peakholder	39
impulsify	39
automat	40

bpf	40
listInterp	40
bypass1	41
bypass2	41
toggle	41
on_and_off	42
selectoutn	42
compressors.lib	43
Functions Reference	43
compressor_mono	43
compressor_stereo	43
limiter_1176_R4_mono	44
limiter_1176_R4_stereo	44
delays.lib	45
Basic Delay Functions	45
delay	45
fdelay	46
sdelay	46
Lagrange Interpolation	46
fdelaylti and fdelayltv	46
fdelay[n]	47
Thiran Allpass Interpolation	47
fdelay[n]a	47
demos.lib	48
Analyzers	48
mth_octave_spectral_level_demo	48
Filters	48
parametric_eq_demo	48
spectral_tilt_demo	48
mth_octave_filterbank_demo and filterbank_demo	49
Effects	49
cubicnl_demo	49
gate_demo	49
compressor_demo	50
moog_vcf_demo	50
wah4_demo	50
crybaby_demo	50
flanger_demo	50
phaser2_demo	51
stereo_reverb_tester	51
fdnrev0_demo	51
zita_rev_fdn_demo	52
zita_rev1	52

Generators	52
sawtooth_demo	52
virtual_analog_oscillator_demo	52
oscrs_demo	53
velvet_noise_demo	53
latch_demo	53
envelopes_demo	53
exciter	54
vocoder_demo	54
freeverb_demo	54
envelopes.lib	55
Functions Reference	55
smoothEnvelope	55
ar	55
asr	55
adsr	56
adsre	56
filters.lib	57
Basic Filters	57
zero	57
pole	58
integrator	58
dcblokerat	58
dcblocker	59
Comb Filters	59
ff_comb	59
ff_fcomb	59
ffcombfilter	60
fb_comb	60
fb_fcomb	61
rev1	61
fbcombfilter and ffbcombfilter	61
allpass_comb	62
allpass_fcomb	62
rev2	63
allpass_fcomb5 and allpass_fcomb1a	63
Direct-Form Digital Filter Sections	63
iir	63
fir	64
conv and convN	65
tf1, tf2 and tf3	65
notchw	65
Direct-Form Second-Order Biquad Sections	66
tf21, tf22, tf22t and tf21t	66

Ladder/Lattice Digital Filters	67
av2sv	67
bvav2nuv	67
iir_lat2	68
allpassnt	68
iir_kl	68
allpassnkl	68
iir_lat1	69
allpassn1mt	69
iir_n1	69
allpassnn1t	70
Useful Special Cases	70
tf2np	70
wgr	71
nlf2	71
apn1	72
Ladder/Lattice Allpass Filters	72
allpassn	72
allpassnn	73
allpasskl	73
allpass1m	74
Digital Filter Sections Specified as Analog Filter Sections	74
tf2s and tf2snp	74
tf3slf	75
tf1s	75
tf2sb	76
tf1sb	76
Simple Resonator Filters	77
resonlp	77
resonhp	77
resonbp	78
Butterworth Lowpass/Highpass Filters	78
lowpass	78
highpass	78
lowpass0_highpass1	79
Special Filter-Bank Delay-Equalizing Allpass Filters	79
lowpass_plus minus_highpass	79
Elliptic (Cauer) Lowpass Filters	79
lowpass3e	79
lowpass6e	80
Elliptic Highpass Filters	80
highpass3e	80
highpass6e	81
Butterworth Bandpass/Bandstop Filters	81
bandpass	81
bandstop	82

Elliptic Bandpass Filters	82
bandpass6e	82
bandpass12e	82
Parametric Equalizers (Shelf, Peaking)	83
low_shelf	83
high_shelf	84
peak_eq	84
peak_eq_cq	85
peak_eq_rm	85
spectral_tilt	85
levelfilter	86
levelfilterN	87
Mth-Octave Filter-Banks	87
mth_octave_filterbank[n]	88
Arbitrary-Crossover Filter-Banks and Spectrum Analyzers	89
filterbank	89
filterbanki	89
hoa.lib	90
encoder	90
decoder	90
decoderStereo	91
Optimization Functions	91
optimBasic	91
optimMaxRe	91
optimInPhase	92
Usage	92
wider	92
map	92
rotate	93
maths.lib	93
Functions Reference	93
SR	93
BS	93
PI	94
FTZ	94
neg	94
sub(x,y)	94
inv	95
cbrt	95
hypot	95
ldexp	95
scalb	95
log1p	96
logb	96

ilogb	96
log2	96
expm1	97
acosh	97
asinh	97
atanh	97
sinh	97
cosh	98
tanh	98
erf	98
erfc	98
gamma	99
lgamma	99
J0	99
J1	99
Jn	100
Y0	100
Y1	100
Yn	100
fabs, fmax, fmin	101
np2	101
frac	101
isnan	101
chebychev	102
chebyshevpoly	102
diffn	103
signum	103
misceffects.lib	103
Dynamic	103
cubicnl	103
gate_mono	104
gate_stereo	104
Filtering	105
speakerbp	105
piano_dispersion_filter	106
stereo_width	106
Time Based	107
echo	107
Pitch Shifting	107
transpose	107
Meshes	107
mesh_square	107
noises.lib	109
Functions Reference	109

noise	109
multirandom	109
multinoise	109
noises	110
pink_noise	110
pink_noise_vm	110
lfnoise, lfnoise0 and lfnoiseN	110
sparse_noise_vm	111
velvet_noise_vm	111
gnoise	112
oscillators.lib	112
Wave-Table-Based Oscillators	112
sinwaveform	112
coswaveform	113
phasor	113
oscsin	113
osccos	114
oscp	114
osci	114
LFOs	115
lf_imptrain	115
lf_pulsetrainpos	115
lf_squarewavepos	115
lf_squarewave	116
lf_trianglepos	116
Low Frequency Sawtooths	116
lf_rawsaw	117
lf_sawpos_phase	117
Bandlimited Sawtooth	117
sawNp	119
saw2dpw	119
saw3	119
sawtooth	119
saw2f2	119
saw2f4	119
Bandlimited Pulse, Square, and Impulse Trains	120
pulsetrainN	120
pulsetrain	120
squareN	121
square	121
impulse	121
imptrainN	121
imptrain	121
triangleN	122
triangle	122

Filter-Based Oscillators	122
oscb	123
oscrq	123
oscrs	123
oscrc	124
osc	124
oscs	124
Waveguide-Resonator-Based Oscillators	125
oscw	125
oscws	125
oscwq	126
oscw	126
lf_sawpos	126
lf_saw	127
lf_triangle	127
phaflangers.lib	127
Functions Reference	128
flanger_mono	128
flanger_stereo	128
phaser2_mono	129
phaser2_stereo	129
physmodels.lib	130
Global Variables	131
speedOfSound	131
maxLength	131
Conversion Tools	131
f2l	131
l2f	132
l2s	132
Bidirectional Utilities	132
basicBlock	132
chain	133
inLeftWave	133
inRightWave	133
in	134
outLeftWave	134
outRightWave	134
out	134
terminations	135
lTermination	135
rTermination	136
closeIns	136
closeOuts	136
endChain	136

Basic Elements	137
waveguideN	137
waveguide	137
bridgeFilter	138
modeFilter	138
String Instruments	138
stringSegment	138
openString	139
nylonString	139
steelString	140
openStringPick	140
openStringPickUp	140
openStringPickDown	141
ksReflexionFilter	141
rStringRigidTermination	142
lStringRigidTermination	142
elecGuitarBridge	142
elecGuitarNuts	142
guitarBridge	143
guitarNuts	143
idealString	143
ks	144
ks_ui_MIDI	144
elecGuitarModel	144
elecGuitar	145
elecGuitar_ui_MIDI	145
guitarBody	145
guitarModel	146
guitar	146
guitar_ui_MIDI	146
nylonGuitarModel	147
nylonGuitar	147
nylonGuitar_ui_MIDI	148
Bowed String Instruments	148
bowTable	148
violinBowTable	148
bowInteraction	149
violinBow	149
violinBowedString	149
violinNuts	150
violinBridge	150
violinBody	150
violinModel	150
violinModel_ui	151
violin_ui_MIDI	151
Wind Instruments	151

openTube	151
reedTable	152
fluteJetTable	152
brassLipsTable	152
clarinetReed	153
clarinetMouthPiece	153
brassLips	154
fluteEmbouchure	154
wBell	154
fluteHead	155
fluteFoot	155
clarinetModel	155
clarinetModel_ui	156
clarinet_ui	156
clarinet_ui_MIDI	156
brassModel	156
brassModel_ui	157
brass_ui	157
brass_ui_MIDI	157
fluteModel	158
fluteModel_ui	158
flute_ui	158
flute_ui_MIDI	159
Exciters	159
impulseExcitation	159
strikeModel	159
strike	160
pluckString	160
blower	161
blower_ui	161
Modal Percussions	161
djembeModel	161
djembe	162
djembe_ui_MIDI	162
marimbaBarModel	162
marimbaResTube	163
marimbaModel	163
marimba	164
marimba_ui_MIDI	164
churchBellModel	164
churchBell	165
churchBell_ui	166
englishBellModel	166
englishBell	166
englishBell_ui	167
frenchBellModel	167

frenchBell	168
frenchBell_ui	168
germanBellModel	169
germanBell	169
germanBell_ui	170
russianBellModel	170
russianBell	171
russianBell_ui	171
standardBellModel	172
standardBell	172
standardBell_ui	173
Vocal Synthesis	173
formantFilter	173
SFFormantModel	174
SFFormantModel_ui	174
SFFormantModel_ui_MIDI	174
Misc Functions	175
allpassNL	175
reverbs.lib	175
Schroeder Reverberators	175
jcrev	175
satrev	176
Feedback Delay Network (FDN) Reverberators	176
fdnrev0	176
zita_rev_fdn	177
zita_rev1_stereo	177
zita_rev1_ambi	178
Freeverb	178
mono_freeverb	178
stereo_freeverb	179
routes.lib	179
Functions Reference	179
cross	179
crossnn	180
crossn1	180
interleave	180
butterfly	181
hadamard	181
recursive	181
signals.lib	182
Functions Reference	182
bus	182
block	182

interpolate	183
smoo	183
polySmooth	183
smoothAndH	184
bsmooth	184
dot	184
smooth	184
lag_ud	185
spats.lib	185
panner	185
spat	186
stereoize	186
synths.lib	186
popFilterPerc	187
dubDub	187
sawTrombone	187
combString	188
additiveDrum	188
fm	189
vaeffects.lib	189
Functions Reference	189
moog_vcf	189
moog_vcf_2b[n]	190
wah4	190
autowah	191
crybaby	191
vocoder	191
Licenses	192
STK 4.3 License	192
LGPL License	192

Faust Libraries

NOTE: this documentation was automatically generated.

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

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

WARNING: These libraries replace the “old” Faust libraries. They are still being beta tested so you might encounter bugs while using them. If your codes still use the “old” Faust libraries, you might want to try to use Bart Brouns’ script that automatically makes an old Faust code compatible with the new libraries: <https://github.com/magnetophon/faustCompressors/blob/master/newlib.sh>. If you find a bug, please report it at rmichon_at_ccrma_dot_stanford_dot_edu. Thanks ;)!

Using the Faust Libraries

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

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

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

- `sf: all.lib`
- `an: analyzers.lib`
- `ba: basics.lib`
- `co: compressors.lib`
- `de: delays.lib`
- `dm: demos.lib`
- `en: envelopes.lib`
- `fi: filters.lib`
- `ho: hoa.lib`
- `ma: maths.lib`
- `ef: misceffects.lib`
- `os: oscillators.lib`
- `no: noises.lib`
- `pf: phaflangers.lib`
- `pm: physmodels.lib`
- `re: reverbs.lib`
- `ro: routes.lib`
- `si: signals.lib`
- `sp: spats.lib`
- `sy: synths.lib`
- `ve: vaeffects.lib`

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

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

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

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

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

Alternatively, environments can be created by hand:

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

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

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

Contributing

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

New Functions

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

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

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

New Libraries

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

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

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

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

General Organization

Only the libraries that are considered to be “standard” are documented:

- `analyzers.lib`
- `basics.lib`

- `compressors.lib`
- `delays.lib`
- `demos.lib`
- `envelopes.lib`
- `filters.lib`
- `hoa.lib`
- `maths.lib`
- `misceffects.lib`
- `oscillators.lib`
- `noises.lib`
- `phaflangers.lib`
- `physmodels.lib`
- `reverbs.lib`
- `routes.lib`
- `signals.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`

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

Coding Conventions

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

Documentation

- All the functions that we want to be “public” are documented.

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

Library Import

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

```
an = library("analyzers.lib");
ba = library("basics.lib");
co = library("compressors.lib");
de = library("delays.lib");
dm = library("demos.lib");
en = library("envelopes.lib");
fi = library("filters.lib");
ho = library("hoa.lib");
ma = library("maths.lib");
ef = library("misceffects.lib");
os = library("oscillators.lib");
no = library("noises.lib");
pf = library("phaflangers.lib");
pm = library("physmodels.lib");
re = library("reverbs.lib");
ro = library("routes.lib");
sp = library("spats.lib");
si = library("signals.lib");
sy = library("synths.lib");
ve = library("vaeffects.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.

“Demo” Functions

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

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

“Standard” Functions

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

Copyright / License

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

Note that it is presently possible for a library function released under one license to utilize another library function having some different license. There is presently no indication of this situation in the Faust compiler output, but such notice is planned. For now, library contributors should strive to use only library functions having compatible licenses, and concerned end-users must manually determine the union of licenses applicable to the library functions they are using.

Standard Functions

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

Analysis Tools

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

Basic Elements

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

Conversion

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

Effects

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

Envelope Generators

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

Filters

Function Type	Function Name	Description
Bandpass (Butterworth)	<code>fi.bandpass</code>	Generic butterworth bandpass
Bandpass (Resonant)	<code>fi.resonbp</code>	Virtual analog resonant bandpass
Bandstop (Butterworth)	<code>fi.bandstop</code>	Generic butterworth bandstop
Biquad	<code>fi.tf2</code>	“Standard” biquad filter
Comb (Allpass)	<code>fi.allpass_fcomb</code>	Schroeder allpass comb filter
Comb (Feedback)	<code>fi.fb_fcomb</code>	Feedback comb filter

Function Type	Function Name	Description
Comb (Feedforward)	<code>fi.ff_fcomb</code>	Feed-forward comb filter.
DC Blocker	<code>fi.dcblocker</code>	Default dc blocker
Filterbank	<code>fi.filterbank</code>	Generic filter bank
FIR (Arbitrary Order)	<code>fi.fir</code>	Nth-order FIR filter
High Shelf	<code>fi.high_shelf</code>	High shelf
Highpass (Butterworth)	<code>fi.highpass</code>	Nth-order Butterworth highpass
Highpass (Resonant)	<code>fi.resonhp</code>	Virtual analog resonant highpass
IIR (Arbitrary Order)	<code>fi.iir</code>	Nth-order IIR filter
Level Filter	<code>fi.levelfilter</code>	Dynamic level lowpass
Low Shelf	<code>fi.low_shelf</code>	Low shelf
Lowpass (Butterworth)	<code>fi.lowpass</code>	Nth-order Butterworth lowpass
Lowpass (Resonant)	<code>fi.resonlp</code>	Virtual analog resonant lowpass
Notch Filter	<code>fi.notchw</code>	Simple notch filter
Peak Equalizer	<code>fi.peak_eq</code>	Peaking equalizer section

Oscillators/Sound Generators

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

Synths

Function Type	Function Name	Description
Additive Drum	<code>sy.additiveDrum</code>	Additive synthesis drum
Bandpassed Sawtooth	<code>sy.dubDub</code>	Sawtooth through resonant bandpass
Comb String	<code>sy.combString</code>	String model based on a comb filter

Function Type	Function Name	Description
FM	<code>sy.fm</code>	Frequency modulation synthesizer
Lowpassed Sawtooth	<code>sy.sawTrombone</code>	“Trombone” based on a filtered sawtooth
Popping Filter	<code>sy.popFilterPerc</code>	Popping filter percussion instrument

analyzers.lib

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

Amplitude Tracking

`amp_follower`

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

Usage

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

Where:

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

Reference

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

`amp_follower_ud`

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

Usage

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

Where:

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

Note

We assume $\text{rel} \gg \text{att}$. Otherwise, consider $\text{rel} \sim \max(\text{rel}, \text{att})$. For audio, att is normally faster (smaller) than rel (e.g., 0.001 and 0.01). Use `amp_follower_ar` below to remove this restriction.

Reference

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

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 ($< \text{SR}/2$)

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

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

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

Increasing Channel Isolation

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

References

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

`mth_octave_analyzer`

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

Usage

```
_ : mth_octave_analyzer(0,M,ftop,N) : par(i,N,_); // 0th-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:

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

Mth-Octave Spectral Level

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

`mth_octave_spectral_level6e`

Spectral level display.

Usage:

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

Where:

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

Also for convenience:

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

[third|half]_octave_[analyzer|filterbank]

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

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

Usage

See `mth_octave_spectral_level_demo`.

Arbitrary-Crossover Filter-Banks and Spectrum Analyzers

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

analyzer

Analyzer.

Usage

```
_ : analyzer(0,freqs) : par(i,N,_); // No delay equalizer
```

Where:

- 0: 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)) : _,_,_
```

amp_follower_ar

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

Usage

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

- Author Jonatan Liljedahl, revised by RM
-

basics.lib

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

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

The official prefix of this library is **ba**.

Conversion Tools

samp2sec

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

Usage

`samp2sec(n)` : _

Where:

- `n`: number of samples
-

`sec2samp`

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

Usage

`sec2samp(d)` : _

Where:

- `d`: duration in seconds
-

`db2linear`

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

Usage

`db2linear(l)` : _

Where:

- `l`: loudness in dB
-

`linear2db`

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

Usage

`linear2db(g) : _`

Where:

- `g`: a linear gain
-

`lin2LogGain`

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

Usage

`_ : lin2LogGain : _`

`log2LinGain`

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

Usage

`_ : log2LinGain : _`

`tau2pole`

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

Usage

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

Where:

- `tau`: time-constant in seconds
-

pole2tau

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

Usage

pole2tau(pole) : _

Where:

- pole: the pole
-

midkey2hz

Converts a MIDI key number to a frequency in Hz (MIDI key 69 = A440). **midkey2hz** is a standard Faust function.

Usage

midkey2hz(mk) : _

Where:

- mk: the MIDI key number
-

pianokey2hz

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

Usage

pianokey2hz(pk) : _

Where:

- pk: the piano key number
-

hz2pianokey

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

Usage

`hz2pianokey(f) : _`

Where:

- `f`: frequency in Hz
-

Counters and Time/Tempo Tools

`countdown`

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

Usage

`countdown(n,trig) : _`

Where:

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

`countup`

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

Usage

`countup(n,trig) : _`

Where:

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

sweep

Counts from 0 to **period** samples repeatedly, while **run** is 1. Outputs zero while **run** is 0.

Usage

sweep(period,run) : _

time

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

Usage

time : _

tempo

Converts a tempo in BPM into a number of samples.

Usage

tempo(t) : _

Where:

- **t**: tempo in BPM
-

period

Basic sawtooth wave of period **p**.

Usage

`period(p) : _`

Where:

- `p`: period as a number of samples
-

`pulse`

Pulses (10000) generated at period `p`.

Usage

`pulse(p) : _`

Where:

- `p`: period as a number of samples
-

`pulsen`

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

Usage

`pulsen(n,p) : _`

Where:

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

`beat`

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

Usage

`beat(t) : _`

Where:

- `t`: tempo in BPM
-

`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)
-

`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)
-

`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)
-

pulse_countdown_loop

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

Usage

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

Where:

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

Array Processing/Pattern Matching

count

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

Usage

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

Where:

- **l**: list of elements
-

take

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

Usage

```
take(e,1)
take(3,(10,20,30,40)) -> 30
```

Where:

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

subseq

Extract a part of a list.

Usage

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

Where:

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

Note:

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

Selectors (Conditions)

if

if-then-else implemented with a select2.

Usage

- if(c, t, e) : _

Where:

- c: condition

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

selector

Selects the ith input among n at compile time.

Usage

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

Where:

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

selectn

Selects the ith input among N at run time.

Usage

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

Where:

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

Example test program

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

select2stereo

Select between 2 stereo signals.

Usage

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

Where:

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

Other

latch

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

Usage

```
_ : latch(clocksigsig) : _
```

Where:

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

sAndH

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

Usage

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

Where:

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

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
-

peakhold

Outputs current max value above zero.

Usage

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

Where:

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

peakholder

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

Usage

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

impulsify

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

Usage

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

automat

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

Usage

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

bpf

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

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

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

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

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

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

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

For example the following definition :

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

implements a break-point function `f` such that :

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

`bpf` is a standard Faust function.

listInterp

Linearly interpolates between the elements of a list.

Usage

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

Where:

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

bypass1

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

Usage

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

Where:

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

bypass2

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

Usage

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

Where:

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

toggle

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

Usage

```
_ : toggle : _
```

Examples

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

on_and_off

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

Usage

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

Example

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

selectoutn

Route input to the output among N at run time.

Usage

```
_ : selectoutn(n, s) : _,_,...n
```

Where:

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

Example

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

compressors.lib

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

Functions Reference

`compressor_mono`

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

Usage

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

Where:

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

References

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

`compressor_stereo`

Stereo dynamic range compressors.

Usage

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

Where:

- **ratio**: compression ratio (1 = no compression, >1 means compression)
- **thresh**: dB level threshold above which compression kicks in (0 dB = max level)

- **att**: attack time = time constant (sec) when level & compression going up
- **rel**: release time = time constant (sec) coming out of compression

References

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

limiter_1176_R4_mono

A limiter guards against hard-clipping. It can be implemented as a compressor having a high threshold (near the clipping level), fast attack and release, and high ratio. Since the ratio is so high, some knee smoothing is desirable ("soft limiting"). This example is intended to get you started using `compressor_*` as a limiter, so all parameters are hardwired to nominal values here. Ratios: 4 (moderate compression), 8 (severe compression), 12 (mild limiting), or 20 to 1 (hard limiting) Att: 20-800 MICROseconds (Note: scaled by ratio in the 1176) Rel: 50-1100 ms (Note: scaled by ratio in the 1176) Mike Shipley likes 4:1 (Grammy-winning mixer for Queen, Tom Petty, etc.) Faster attack gives "more bite" (e.g. on vocals) He hears a bright, clear eq effect as well (not implemented here) `limiter_1176_R4_mono` is a standard Faust function.

Usage

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

Reference:

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

limiter_1176_R4_stereo

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

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

Usage

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

Reference:

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

delays.lib

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

Basic Delay Functions

delay

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

Usage

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

Where:

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

fdelay

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

sdelay

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

Usage

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

Where :

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

Lagrange Interpolation

fdelaylti and fdelayltv

Fractional delay line using Lagrange interpolation.

Usage

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

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

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

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

References

- https://ccrma.stanford.edu/~jos/pasp/Lagrange_Interpolation.html
- Timo I. Laakso et al., "Splitting the Unit Delay - Tools for Fractional Delay Filter Design", IEEE Signal Processing Magazine, vol. 13, no. 1, pp. 30-60, Jan 1996.

- Philippe Depalle and Stephan Tassart, “Fractional Delay Lines using Lagrange Interpolators”, ICMC Proceedings, pp. 341-343, 1996.
-

`fdelay[n]`

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

Thiran Allpass Interpolation

Thiran Allpass Interpolation

Reference

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

`fdelay[n]a`

Delay lines interpolated using Thiran allpass interpolation.

Usage

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

(exactly like `fdelay`)

Where:

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

Note

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

Delay arguments too small will produce an UNSTABLE allpass!

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

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

demos.lib

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

Analyzers

math_octave_spectral_level_demo

Demonstrate `math_octave_spectral_level` in a standalone GUI.

Usage

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

Filters

parametric_eq_demo

A parametric equalizer application.

Usage:

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

spectral_tilt_demo

A spectral tilt application.

Usage

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

Where:

- N: filter order (integer)

All other parameters interactive

mth_octave_filterbank_demo and filterbank_demo

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

Usage

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

Where:

- N: number of bands per octave
-

Effects

cubicnl_demo

Distortion demo application.

Usage:

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

gate_demo

Gate demo application.

Usage

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

compressor_demo

Compressor demo application.

Usage

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

moog_vcf_demo

Illustrate and compare all three Moog VCF implementations above.

Usage

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

wah4_demo

Wah pedal application.

Usage

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

crybaby_demo

Crybaby effect application.

Usage

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

flanger_demo

Flanger effect application.

Usage

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

phaser2_demo

Phaser effect demo application.

Usage

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

stereo_reverb_tester

Handy test inputs for reverberator demos below.

Usage

```
_ : stereo_reverb_tester : _
```

fdnrev0_demo

A reverb application using `fdnrev0`.

Usage

```
_,_ : fdnrev0_demo(N,NB,BBS0) : _,_
```

Where:

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

zita_rev_fdn_demo

Reverb demo application based on `zita_rev_fdn`.

Usage

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

zita_rev1

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

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

Usage

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

Reference

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

Generators

sawtooth_demo

An application demonstrating the different sawtooth oscillators of Faust.

Usage

```
sawtooth_demo : _
```

virtual_analog_oscillator_demo

Virtual analog oscillator demo application.

Usage

```
virtual_analog_oscillator_demo : _
```

oscrs_demo

Simple application demoing filter based oscillators.

Usage

```
oscrs_demo : _
```

velvet_noise_demo

Listen to velvet_noise!

Usage

```
velvet_noise_demo : _
```

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);
```

envelopes_demo

Illustrate various envelopes overlaid

Usage

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

exciter

Psychoacoustic harmonic exciter, with GUI.

Usage

```
_ : exciter : _
```

References

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

vocoder_demo

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

Usage

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

freeverb_demo

Freeverb demo application.

Usage

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

envelopes.lib

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

Functions Reference

smoothEnvelope

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

Usage

smoothEnvelope(ar,t) : _

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

ar

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

Usage

ar(a,r,t) : _

Where:

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

asr

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

Usage

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

Where:

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

adsr

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

Usage

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

Where:

- **a, d, s, r**: attack (sec), decay (sec), sustain level (percentage of max), release (sec)
 - **g**: gate signal (>0 for attack, then release is when g back to 0)
-

adsre

ADSRE (Attack, Decay, Sustain, Release) envelope generator with Exponential segments. **adsre** is a standard Faust function.

Usage

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

Where:

- **a, d, s, r**: attack (sec), decay (sec), sustain level (percentage of max), release (sec)
 - **g**: gate signal (>0 for attack, then release is when g back to 0)
-

filters.lib

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

The Filters library is organized into 18 sections:

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

For more information, see [../documentation/library.pdf](#)

Basic Filters

zero

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

Usage

_ : zero(z) : _

Where:

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

Reference

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

pole

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

Usage

`_ : pole(z) : _`

Where:

- `p`: pole location = feedback coefficient

Reference

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

integrator

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

dcblockerat

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

Usage

`_ : dcblockerat(fb) : _`

Where:

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

Reference

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

dcblocker

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

Usage

_ : dcblocker : _

Comb Filters

ff_comb

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

Usage

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

Where:

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

Reference

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

ff_fcomb

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

Usage

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

Where:

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

Reference

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

ffcombfiler

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

fb_comb

Feed-Back Comb Filter (integer delay).

Usage

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

Where:

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

Reference

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

fb_fcomb

Feed-Back Comb Filter (floating point delay).

Usage

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

Where:

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

Reference

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

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.

fbcombfilter and ffbcombfilter

Other special cases of Feed-Back Comb Filter.

Usage

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

Where:

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

Reference

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

allpass_comb

Schroeder Allpass Comb Filter. Note that

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

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

Usage

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

Where:

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

References

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

allpass_fcomb

Schroeder Allpass Comb Filter. Note that

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

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

allpass_fcomb is a standard Faust library.

Usage

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

Where:

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

References

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

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.

allpass_fcomb5 and allpass_fcomb1a

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

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

Direct-Form Digital Filter Sections

iir

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

`iir` is a standard Faust function.

Usage

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

Where:

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

Reference

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

fir

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

Usage

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

fir is standard Faust function.

Where:

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

Note

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

Example

Smoothing white noise with a five-point moving average:

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

Equivalent (note double parens):

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

conv and convN

Convolution of input signal with given coefficients.

Usage

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

tf1, tf2 and tf3

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

Usage

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

Where:

- a: the poles
- b: the zeros

Reference

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

notchw

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

Usage:

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

Where:

- width: “notch width” in Hz (approximate)
- freq: “notch frequency” in Hz

Reference

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

Direct-Form Second-Order Biquad Sections

Direct-Form Second-Order Biquad Sections

Reference

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

tf21, tf22, tf22t and tf21t

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

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

Usage

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

Where:

- a: the poles
- b: the zeros

Reference

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

Ladder/Lattice Digital Filters

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

av2sv

Compute reflection coefficients **sv** from transfer-function denominator **av**.

Usage

sv = **av2sv**(**av**)

Where:

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

where **ro** = **i**th reflection coefficient, and **ai** = coefficient of z^{-i} in the filter transfer-function denominator **A**(**z**).

Reference

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

bvav2nuv

Compute lattice tap coefficients from transfer-function coefficients.

Usage

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

Where:

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

where **nu****i** is the **i**'th tap coefficient, **bi** is the coefficient of z^{-i} in the filter numerator, **ai** is the coefficient of z^{-i} in the filter denominator

iir_lat2

Two-multiply lattice IIR filter or arbitrary order.

Usage

_ : iir_lat2(bv,av) : _

Where:

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

allpassnt

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

Usage

_ : allpassnt(n,sv) : _

Where:

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

iir_kl

Kelly-Lochbaum ladder IIR filter or arbitrary order.

Usage

_ : iir_kl(bv,av) : _

Where:

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

allpassnklr

Kelly-Lochbaum ladder allpass.

Usage:

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

Where:

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

iir_lat1

One-multiply lattice IIR filter or arbitrary order.

Usage

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

Where:

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

allpassn1mt

One-multiply lattice allpass with tap lines.

Usage

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

Where:

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

iir_n1

Normalized ladder filter of arbitrary order.

Usage

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

Where:

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

References

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

`allpassnlt`

Normalized ladder allpass filter of arbitrary order.

Usage:

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

Where:

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

References

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

Useful Special Cases

`tf2np`

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

Usage

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

Where:

- **a**: the poles
 - **b**: the zeros
-

wgr

Second-order transformer-normalized digital waveguide resonator.

Usage

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

Where:

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

References

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

nlf2

Second order normalized digital waveguide resonator.

Usage

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

Where:

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

Reference

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

apnl

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

Usage

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

Where:

- **a1** and **a2**: allpass coefficients

Reference

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

Ladder/Lattice Allpass Filters

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

References

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

allpassn

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

Usage:

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

Where:

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

References

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

allpassnn

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

Usage:

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

Where:

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

allpasskl

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

Usage:

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

Where:

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

allpass1m

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

Usage:

`_ : allpass1m(n,sv) : _`

Where:

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

Digital Filter Sections Specified as Analog Filter Sections

tf2s and tf2snp

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

Usage

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

Where:

$$H(s) = \frac{b2 s^2 + b1 s + b0}{s^2 + a1 s + a0}$$

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

Example

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

$$H(s) = \frac{1}{s^2 + a1 s + 1}$$

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

Method

Bilinear transform scaled for exact mapping of $w1$.

Reference

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

tf3slf

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

Usage

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

tf1s

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

Usage

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

Where:

$$b1 s + b0$$

$$H(s) = \frac{b1 s + b0}{s + a0}$$

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

Example

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

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

so $b_0 = a_0 = 1$ and $b_1 = 0$. 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 w_1 .

Reference

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

tf2sb

Bandpass mapping of **tf2s**: In addition to a frequency-scaling parameter w_1 (set to HALF the desired passband width in rad/sec), there is a desired center-frequency parameter w_c (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 w_1), (2) bandpass mapping to w_c , then (3) the bilinear transform, with the usual scale parameter $2*SR$. Algebra carried out in maxima and pasted here.

Usage

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

tf1sb

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

Usage

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

Simple Resonator Filters

resonlp

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)
-

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)
-

resonbp

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

Usage

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

Where:

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

Butterworth Lowpass/Highpass Filters

lowpass

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

Usage

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

Where:

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

References

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

highpass

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

Usage

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

Where:

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

References

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

`lowpass0_highpass1`

Special Filter-Bank Delay-Equalizing Allpass Filters

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

`lowpass_plus|minus_highpass`

Elliptic (Cauer) Lowpass Filters

Elliptic (Cauer) Lowpass Filters

References

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

`lowpass3e`

Third-order Elliptic (Cauer) lowpass filter.

Usage

`_ : lowpass3e(fc) : _`

Where:

- `fc`: -3dB frequency in Hz

Design

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

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

`lowpass6e`

Sixth-order Elliptic/Cauer lowpass filter.

Usage

`_ : lowpass6e(fc) : _`

Where:

- `fc`: -3dB frequency in Hz

Design

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

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

Elliptic Highpass Filters

`highpass3e`

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

Usage

`_ : highpass3e(fc) : _`

Where:

- `fc`: -3dB frequency in Hz
-

highpass6e

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

Usage

`_ : highpass6e(fc) : _`

Where:

- `fc`: -3dB frequency in Hz
-

Butterworth Bandpass/Bandstop Filters

bandpass

Order $2*Nh$ Butterworth bandpass filter made using the transformation $s \leftarrow s + wc^2/s$ 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,f1,fu) : _`

Where:

- `Nh`: HALF the desired bandpass order (which is therefore even)
- `f1`: lower -3dB frequency in Hz
- `fu`: upper -3dB frequency in Hz Thus, the passband width is `fu-f1`, and its center frequency is `(f1+fu)/2`.

Reference

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

bandstop

Order $2*Nh$ Butterworth bandstop filter made using the transformation $s \leftarrow s + wc^2/s$ 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,f1,fu) : _`

Where:

- Nh : HALF the desired bandstop order (which is therefore even)
- $f1$: lower -3dB frequency in Hz
- fu : upper -3dB frequency in Hz Thus, the passband (stopband) width is $fu-f1$, and its center frequency is $(f1+fu)/2$.

Reference

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

Elliptic Bandpass Filters

bandpass6e

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

bandpass12e

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

Parametric Equalizers (Shelf, Peaking)

Parametric Equalizers (Shelf, Peaking)

References

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

low_shelf

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

Usage

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

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

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

Shelf Shape

The magnitude frequency response is approximately piecewise-linear on a log-log plot (“BODE PLOT”). The Bode “stick diagram” approximation $L(f)$ 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 $lf2 = 2nd\ corner\ frequency$;
 - $L(lf) = 0$, $lf > lf2$.
- $lf2 = lfx + L0/N = dB\text{-frequency at which level gets back to 0 dB}$.
- $L0 < 0$:
 - $L(lf) = L0$, f between 0 and $fl = 1st\ corner\ frequency$;

- $L(lf) = -N * (lfx - lf)$, f between f_l and $lfx = 2nd$ corner frequency;
- $L(lf) = 0$, $lf > lfx$.
- $lf_l = lfx + L0/N = dB$ -frequency at which level goes up from $L0$.

See `lowshelf_other_freq`.

high_shelf

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

Usage

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

Where:

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

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

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
-

peak_eq_cq

Constant-Q second order peaking equalizer section.

Usage

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

Where:

- **Lfx**: level (dB) at **fx**
 - **fx**: boost or cut frequency (Hz)
 - **Q**: “Quality factor” = fx/B where B = bandwidth of peak in Hz
-

peak_eq_rm

Regalia-Mitra second order peaking equalizer section

Usage

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

Where:

- **Lfx**: level (dB) at **fx**
- **fx**: boost or cut frequency (Hz)
- **tanPiBT**: $\tan(\pi B/SR)$, where B = -3dB bandwidth (Hz) when $10^{(Lfx/20)} = 0 \sim \pi B/SR$ for narrow bandwidths B

Reference

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

spectral_tilt

Spectral tilt filter, providing an arbitrary spectral rolloff factor α 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, α 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 α to $-1/2$.

Usage

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

Where:

- **N**: desired integer filter order (fixed at compile time)
- **f0**: lower frequency limit for desired roll-off band
- **bw**: bandwidth of desired roll-off band
- **alpha**: slope of roll-off desired in nepers per neper ($\ln \text{mag} / \ln \text{radian freq}$)

Examples

See `spectral_tilt_demo`.

Reference

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

levelfilter

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

Usage

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

Where:

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

Reference

https://ccrma.stanford.edu/real/simple/faust_strings/Dynamic_Level_Lowpass_Filter.html

levelfilterN

Dynamic level lowpass filter.

Usage

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

Where:

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

Reference

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

Mth-Octave Filter-Banks

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

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

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

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

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

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

Increasing Channel Isolation

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

References

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

`meth_octave_filterbank[n]`

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

Usage

```
_ : meth_octave_filterbank(0,M,ftop,N) : par(i,N,_); // 0th-order  
_ : meth_octave_filterbank_alt(0,M,ftop,N) : par(i,N,_); // dc-inverted version
```

Also for convenience:

```
_ : meth_octave_filterbank3(M,ftop,N) : par(i,N,_); // 3d-order Butterworth  
_ : meth_octave_filterbank5(M,ftop,N) : par(i,N,_); // 5th-order Butterworth  
meth_octave_filterbank_default = meth_octave_analyzer6e;
```

Where:

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

Arbitrary-Crossover Filter-Banks and Spectrum Analyzers

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

filterbank

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

Usage

```
_ : filterbank (0,freqs) : par(i,N,_); // Butterworth band-splits
```

Where:

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

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

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

filterbanki

Inverted-dc filter bank.

Usage

```
_ : filterbanki(0,freqs) : par(i,N,_); // Inverted-dc version
```

Where:

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

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

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

hoa.lib

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

encoder

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

Usage

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

Where:

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

decoder

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

Usage

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

Where:

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

Note

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

decoderStereo

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

`_ : decoderStereo(n) : _`

Where:

- **n**: the order
-

Optimization Functions

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

optimBasic

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

Usage

`_ : optimBasic(n) : _`

Where:

- **n**: the order
-

optimMaxRe

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

Usage

`_ : optimMaxRe(n) : _`

Where:

- **n**: the order

optimInPhase

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

Usage

“ optimInPhase(n) : _ “

here:

n: the order

wider

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

Usage

_ : wider(n,w) : _

Where:

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

map

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

Usage

map(n, x, r, a)

Where:

- n: the order
- x: the signal

- **r**: the radius
 - **a**: the angle in radian
-

rotate

Rotates the sound field.

Usage

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

Where:

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

maths.lib

Mathematic library for Faust. 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.

The official prefix of this library is **ma**.

Functions Reference

SR

Current sampling rate (between 1000Hz and 192000Hz). Constant during program execution.

Usage

SR : **_**

BS

Current block-size. Can change during the execution.

Usage

BS : _

PI

Constant PI in double precision

Usage

PI : _

FTZ

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

Usage

_ : ftz : _

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

neg

Invert the sign (-x) of a signal.

Usage

_ : neg : _

sub(x,y)

Subtract x and y.

inv

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

Usage

`_ : inv : _`

cbrt

Computes the cube root of of the input signal.

Usage

`_ : cbrt : _`

hypot

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

Usage

`_,_ : hypot : _`

ldexp

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

Usage

`_,_ : ldexp : _`

scalb

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

Usage

`_,_ : scalb : _`

log1p

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

Usage

`_ : log1p : _`

logb

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

Usage

`_ : logb : _`

ilogb

Return exponent of the input signal as an integer number.

Usage

`_ : ilogb : _`

log2

Returns the base 2 logarithm of x .

Usage

`_ : log2 : _`

expm1

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

Usage

_ : expm1 : _

acosh

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

Usage

_ : acosh : _

asinh

Computes the inverse hyperbolic sine of the input signal.

Usage

_ : asinh : _

atanh

Computes the inverse hyperbolic tangent of the input signal.

Usage

_ : atanh : _

sinh

Computes the hyperbolic sine of the input signal.

Usage

`_ : sinh : _`

`cosh`

Computes the hyperbolic cosine of the input signal.

Usage

`_ : cosh : _`

`tanh`

Computes the hyperbolic tangent of the input signal.

Usage

`_ : tanh : _`

`erf`

Computes the error function of the input signal.

Usage

`_ : erf : _`

`erfc`

Computes the complementary error function of the input signal.

Usage

`_ : erfc : _`

gamma

Computes the gamma function of the input signal.

Usage

_ : gamma : _

lgamma

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

Usage

_ : lgamma : _

J0

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

Usage

_ : J0 : _

J1

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

Usage

_ : J1 : _

Jn

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

Usage

`_,_ : Jn : _`

Y0

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

Usage

`_ : Y0 : _`

Y1

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

Usage

`_ : Y0 : _`

Yn

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

Usage

`_,_ : Yn : _`

fabs, fmax, fmin

Just for compatibility...

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

np2

Gives the next power of 2 of x.

Usage

np2(n) : _

Where:

- n: an integer
-

frac

Gives the fractional part of n.

Usage

frac(n) : _

Where:

- n: a decimal number
-

isnan

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

Usage

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

Where:

- `x`: signal to analyse
-

chebychev

Chebyshev transformation of order `n`.

Usage

`_ : chebychev(n) : _`

Where:

- `n`: the order of the polynomial

Semantics

$T[0](x) = 1,$
 $T[1](x) = x,$
 $T[n](x) = 2x * T[n-1](x) - T[n-2](x)$

Reference

http://en.wikipedia.org/wiki/Chebyshev_polynomial

chebychevpoly

Linear combination of the first Chebyshev polynomials.

Usage

`_ : chebychevpoly((c0,c1,...,cn)) : _`

Where:

- `cn`: the different Chebyshev polynomials such that: $\text{chebychevpoly}((c0,c1,\dots,cn)) = \text{Sum of } \text{chebychev}(i) * c_i$

Reference

<http://www.csounds.com/manual/html/chebyshevpoly.html>

diffn

Negated first-order difference.

Usage

```
_ : diffn : _
```

signum

The signum function `signum(x)` is defined as -1 for $x < 0$, 0 for $x = 0$, and 1 for $x > 0$;

Usage

```
_ : signum : _
```

misceffects.lib

This library contains a collection of audio effects. Its official prefix is **ef**.

Dynamic

cubicnl

Cubic nonlinearity distortion. `cubicnl` is a standard Faust library.

Usage:

```
_ : cubicnl(drive,offset) : _  
_ : cubicnl_nodc(drive,offset) : _
```

Where:

- **drive**: distortion amount, between 0 and 1
- **offset**: constant added before nonlinearity to give even harmonics. Note: offset can introduce a nonzero mean - feed cubicnl output to dcblocker to remove this.

References:

- https://ccrma.stanford.edu/~jos/pasp/Cubic_Soft_Clipper.html
 - https://ccrma.stanford.edu/~jos/pasp/Nonlinear_Distortion.html
-

gate_mono

Mono signal gate. **gate_mono** is a standard Faust function.

Usage

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

Where:

- **thresh**: dB level threshold above which gate opens (e.g., -60 dB)
- **att**: attack time = time constant (sec) for gate to open (e.g., 0.0001 s = 0.1 ms)
- **hold**: hold time = time (sec) gate stays open after signal level < thresh (e.g., 0.1 s)
- **rel**: release time = time constant (sec) for gate to close (e.g., 0.020 s = 20 ms)

References

- http://en.wikipedia.org/wiki/Noise_gate
 - <http://www.soundonsound.com/sos/apr01/articles/advanced.asp>
 - [http://en.wikipedia.org/wiki/Gating_\(sound_engineering\)](http://en.wikipedia.org/wiki/Gating_(sound_engineering))
-

gate_stereo

Stereo signal gates. **gate_stereo** is a standard Faust function.

Usage

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

Where:

- **thresh**: dB level threshold above which gate opens (e.g., -60 dB)
- **att**: attack time = time constant (sec) for gate to open (e.g., 0.0001 s = 0.1 ms)
- **hold**: hold time = time (sec) gate stays open after signal level < thresh (e.g., 0.1 s)
- **rel**: release time = time constant (sec) for gate to close (e.g., 0.020 s = 20 ms)

References

- http://en.wikipedia.org/wiki/Noise_gate
 - <http://www.soundonsound.com/sos/apr01/articles/advanced.asp>
 - [http://en.wikipedia.org/wiki/Gating_\(sound_engineering\)](http://en.wikipedia.org/wiki/Gating_(sound_engineering))
-

Filtering

speakerbp

Dirt-simple speaker simulator (overall bandpass eq with observed roll-offs above and below the passband).

Low-frequency speaker model = +12 dB/octave slope breaking to flat near f1. Implemented using two dc blockers in series.

High-frequency model = -24 dB/octave slope implemented using a fourth-order Butterworth lowpass.

Example based on measured Celestion G12 (12" speaker):

speakerbp is a standard Faust function

Usage

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

piano_dispersion_filter

Piano dispersion allpass filter in closed form.

Usage

```
piano_dispersion_filter(M,B,f0)
_ : piano_dispersion_filter(1,B,f0) : +(totalDelay),_ : fdelay(maxDelay) : _
```

Where:

- M: number of first-order allpass sections (compile-time only) Keep below 20. 8 is typical for medium-sized piano strings.
- B: string inharmonicity coefficient (0.0001 is typical)
- f0: fundamental frequency in Hz

Outputs

- MINUS the estimated delay at f0 of allpass chain in samples, provided in negative form to facilitate subtraction from delay-line length.
- Output signal from allpass chain

stereo_width

Stereo Width effect using the Blumlein Shuffler technique. **stereo_width** is a standard Faust function.

Usage

```
_,_ : stereo_width(w) : _,_
```

Where:

- w: stereo width between 0 and 1

At $w=0$, the output signal is mono $((\text{left}+\text{right})/2)$ in both channels). At $w=1$, there is no effect (original stereo image). Thus, w between 0 and 1 varies stereo width from 0 to “original”.

Reference

- “Applications of Blumlein Shuffling to Stereo Microphone Techniques”
Michael A. Gerzon, JAES vol. 42, no. 6, June 1994

Time Based

echo

A simple echo effect.

echo is a standard Faust function

Usage

```
_ : echo(maxDuration,duration,feedback) : _
```

Where:

- **maxDuration**: the max echo duration in seconds
 - **duration**: the echo duration in seconds
 - **feedback**: the feedback coefficient
-

Pitch Shifting

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)
-

Meshes

mesh_square

Square Rectangular Digital Waveguide Mesh.

Usage

```
bus(4*N) : mesh_square(N) : bus(4*N);
```

Where:

- N: number of nodes along each edge - a power of two (1,2,4,8,...)

Reference

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

Signal Order In and Out

The mesh is constructed recursively using 2x2 embeddings. Thus, the top level of `mesh_square(M)` is a block 2x2 mesh, where each block is a `mesh(M/2)`. Let these blocks be numbered 1,2,3,4 in the geometry NW,NE,SW,SE, i.e., as 1 2 3 4. Each block has four vector inputs and four vector outputs, where the length of each vector is M/2. Label the input vectors as Ni,Ei,Wi,Si, i.e., as the inputs from the North, East South, and West, and similarly for the outputs. Then, for example, the upper left input block of M/2 signals is labeled 1Ni. Most of the connections are internal, such as 1Eo -> 2Wi. The 8*(M/2) input signals are grouped in the order 1Ni 2Ni 3Si 4Si 1Wi 3Wi 2Ei 4Ei and the output signals are 1No 1Wo 2No 2Eo 3So 3Wo 4So 4Eo or

In: 1No 1Wo 2No 2Eo 3So 3Wo 4So 4Eo

Out: 1Ni 2Ni 3Si 4Si 1Wi 3Wi 2Ei 4Ei

Thus, the inputs are grouped by direction N,S,W,E, while the outputs are grouped by block number 1,2,3,4, which can also be interpreted as directions NW, NE, SW, SE. A simple program illustrating these orderings is `process = mesh_square(2);`.

Example

Reflectively terminated mesh impulsed at one corner:

```
mesh_square_test(N,x) = mesh_square(N)~(busi(4*N,x)) // input to corner
with { busi(N,x) = bus(N) : par(i,N,*(-1)) : par(i,N-1,_), +(x); };
process = 1-1' : mesh_square_test(4); // all modes excited forever
```

In this simple example, the mesh edges are connected as follows:

1No -> 1Ni, 1Wo -> 2Ni, 2No -> 3Si, 2Eo -> 4Si,

3So -> 1Wi, 3Wo -> 3Wi, 4So -> 2Ei, 4Eo -> 4Ei

A routing matrix can be used to obtain other connection geometries.

noises.lib

Faust Noise Generator Library. Its official prefix is **no**.

Functions Reference

noise

White noise generator (outputs random number between -1 and 1). **Noise** is a standard Faust function.

Usage

noise : _

multirandom

Generates multiple decorrelated random numbers in parallel.

Usage

multirandom(n) : **si.bus(n)**

Where:

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

multinoise

Generates multiple decorrelated noises in parallel.

Usage

multinoise(n) : **si.bus(n)**

Where:

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

noises

TODO.

pink_noise

Pink noise (1/f noise) generator (third-order approximation) **pink_noise** is a standard Faust function.

Usage

```
pink_noise : _;
```

Reference:

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

pink_noise_vm

Multi pink noise generator.

Usage

```
pink_noise_vm(N) : _;
```

Where:

- N: number of latched white-noise processes to sum, not to exceed sizeof(int) in C++ (typically 32).

References

- <http://www.dsprelated.com/showarticle/908.php>
 - <http://www.firstpr.com.au/dsp/pink-noise/#Voss-McCartney>
-

lfnoise, lfnoise0 and lfnoiseN

Low-frequency noise generators (Butterworth-filtered downsampled white noise)

Usage

```
lfnoise0(rate) : _;    // new random number every int(SR/rate) samples or so
lfnoiseN(N,rate) : _; // same as "lfnoise0(rate) : lowpass(N,rate)" [see filters.lib]
lfnoise(rate) : _;    // same as "lfnoise0(rate) : seq(i,5,lowpass(N,rate))" (no overshoot)
```

Example

(view waveforms in faust2octave):

```
rate = SR/100.0; // new random value every 100 samples (SR from music.lib)
process = lfnoise0(rate), // sampled/held noise (piecewise constant)
         lfnoiseN(3,rate), // lfnoise0 smoothed by 3rd order Butterworth LPF
         lfnoise(rate);    // lfnoise0 smoothed with no overshoot
```

sparse_noise_vm

sparse noise generator.

Usage

```
sparse_noise(f0) : _;
```

Where:

- f0: average frequency of noise impulses per second

Random impulses in the amplitude range -1 to 1 are generated at an average rate of f0 impulses per second.

Reference

- See velvet_noise
-

velvet_noise_vm

velvet noise generator.

Usage

```
velvet_noise(amp,f0) : _;
```

Where:

- **amp**: amplitude of noise impulses (positive and negative)
- **f0**: average frequency of noise impulses per second

Reference

- Matti Karjalainen and Hanna Jarvelainen, “Reverberation Modeling Using Velvet Noise”, in Proc. 30th Int. Conf. Intelligent Audio Environments (AES07), March 2007.
-

gnoise

approximate zero-mean, unit-variance Gaussian white noise generator

Usage

`gnoise(N) : _;`

Where:

- **N**: number of uniform random numbers added to approximate Gaussian white noise

Reference

- See Central Limit Theorem
-

oscillators.lib

This library contains a collection of sound generators. Its official prefix is **os**.

Wave-Table-Based Oscillators

sinwaveform

Sine waveform ready to use with a **rdtable**.

Usage

`sinwaveform) : _`

Where:

- `tablesize`: the table size
-

`coswaveform`

Cosine waveform ready to use with a `rdtable`.

Usage

`coswaveform) : _`

Where:

- `tablesize`: the table size
-

`phasor`

A simple phasor to be used with a `rdtable`. `phasor` is a standard Faust function.

Usage

`phasor,freq) : _`

Where:

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

`oscsin`

Sine wave oscillator. `oscsin` is a standard Faust function.

Usage

`oscsin(freq) : _`

Where:

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

osccos

Cosine wave oscillator.

Usage

`osccos(freq) : _`

Where:

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

oscp

A sine wave generator with controllable phase.

Usage

`oscp(freq,p) : _`

Where:

- `freq`: the frequency of the wave (Hz)
 - `p`: the phase in radian
-

osci

Interpolated phase sine wave oscillator.

Usage

`osci(freq) : _`

Where:

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

LFOs

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

`lf_imptrain`

Unit-amplitude low-frequency impulse train. `lf_imptrain` is a standard Faust function.

Usage

`lf_imptrain(freq) : _`

Where:

- `freq`: frequency in Hz
-

`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
-

`lf_squarewavepos`

Positive LF square wave in $[0,1]$

Usage

`lf_squarewavepos(freq) : _`

Where:

- `freq`: frequency in Hz
-

`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
-

`lf_trianglepos`

Positive unit-amplitude LF positive triangle wave

Usage

`lf_trianglepos(freq) : _`

Where:

- `freq`: frequency in Hz
-

Low Frequency Sawtooths

Sawtooth waveform oscillators for virtual analog synthesis et al. The ‘simple’ versions (`lf_rawsaw`, `lf_sawpos` and `saw1`), are mere samplings of the ideal continuous-time (“analog”) waveforms. While simple, the aliasing due to sampling is quite audible. The differentiated polynomial waveform family (`saw2`, `sawN`, and derived functions) do some extra processing to suppress aliasing (not audible for very low fundamental frequencies). According to Lehtonen et al. (JASA 2012), the aliasing of `saw2` should be inaudible at fundamental frequencies

below 2 kHz or so, for a 44.1 kHz sampling rate and 60 dB SPL presentation level; fundamentals 415 and below required no aliasing suppression (i.e., **saw1** is ok).

lf_rawsaw

Simple sawtooth waveform oscillator between 0 and period in samples.

Usage

```
lf_rawsaw(periodsamps)
```

Where:

- **periodsamps**: number of periods per samples
-

lf_sawpos_phase

Simple sawtooth waveform oscillator between 0 and 1 with phase control.

Usage

```
lf_sawpos_phase(freq,phase)
```

Where:

- **freq**: frequency
 - **phase**: phase
-

Bandlimited Sawtooth

//—————**sawN**————— Bandlimited Sawtooth

```
sawN(N,freq), sawNp, saw2dpw(freq), saw2(freq), saw3(freq), saw4(freq),  
saw5(freq), saw6(freq), sawtooth(freq), saw2f2(freq) saw2f4(freq)
```

Method 1 (saw2)

Polynomial Transition Regions (PTR) (for aliasing suppression)

Reference

- Kleimola, J.; Valimaki, V., “Reducing Aliasing from Synthetic Audio Signals Using Polynomial Transition Regions,” in Signal Processing Letters, IEEE , vol.19, no.2, pp.67-70, Feb. 2012
- <https://aaltodoc.aalto.fi/bitstream/handle/123456789/7747/publication6.pdf?sequence=9>
- <http://research.spa.aalto.fi/publications/papers/spl-ptr/>

Method 2 (**sawN**)

Differentiated Polynomial Waves (DPW) (for aliasing suppression)

Reference

“Alias-Suppressed Oscillators based on Differentiated Polynomial Waveforms”, Vesa Valimaki, Juhan Nam, Julius Smith, and Jonathan Abel, IEEE Tr. Acoustics, Speech, and Language Processing (IEEE-ASLP), Vol. 18, no. 5, May 2010.

Other Cases

Correction-filtered versions of **saw2**: **saw2f2**, **saw2f4** The correction filter compensates “droop” near half the sampling rate. See reference for **sawN**.

Usage

```
sawN(N,freq) : _  
sawNp(N,freq,phase) : _  
saw2dpw(freq) : _  
saw2(freq) : _  
saw3(freq) : _ // based on sawN  
saw4(freq) : _ // based on sawN  
saw5(freq) : _ // based on sawN  
saw6(freq) : _ // based on sawN  
sawtooth(freq) : _ // = saw2  
saw2f2(freq) : _  
saw2f4(freq) : _
```

Where:

- **N**: polynomial order
- **freq**: frequency in Hz
- **phase**: phase

sawNp

TODO: Markdown doc in comments

saw2dpw

TODO: Markdown doc in comments

saw3

TODO: Markdown doc in comments

sawtooth

Alias-free sawtooth wave. 2nd order interpolation (based on **saw2**). **sawtooth** is a standard Faust function.

Usage

sawtooth(freq) : _

Where:

- **freq**: frequency
-

saw2f2

TODO: Markdown doc in comments

saw2f4

TODO: Markdown doc in comments

Bandlimited Pulse, Square, and Impulse Trains

Bandlimited Pulse, Square, and Impulse Trains

`pulsetrainN`, `pulsetrain`, `squareN`, `square`, `imptrain`, `imptrainN`, `triangle`, `triangleN`

All are zero-mean and meant to oscillate in the audio frequency range. Use simpler sample-rounded `lf_*` versions above for LFOs.

Usage

```
pulsetrainN(N,freq,duty) : _  
pulsetrain(freq, duty) : _ // = pulsetrainN(2)  
squareN(N, freq) : _  
square : _ // = squareN(2)  
imptrainN(N,freq) : _  
imptrain : _ // = impttrainN(2)  
triangleN(N,freq) : _  
triangle : _ // = triangleN(2)
```

Where:

- `N`: polynomial order
- `freq`: frequency in Hz

`pulsetrainN`

TODO: Markdown doc in comments

`pulsetrain`

Bandlimited pulse train oscillator. Based on `pulsetrainN(2)`. `pulsetrain` is a standard Faust function.

Usage

```
pulsetrain(freq, duty) : _
```

Where:

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

squareN

TODO: Markdown doc in comments

square

Bandlimited square wave oscillator. Based on `squareN(2)`. `square` is a standard Faust function.

Usage

`square(freq) : _`

Where:

- `freq`: frequency
-

impulse

One-time impulse generated when the Faust process is started. `impulse` is a standard Faust function.

Usage

`impulse : _`

imptrainN

TODO: Markdown doc in comments

imptrain

Bandlimited impulse train generator. Based on `imptrainN(2)`. `imptrain` is a standard Faust function.

Usage

`imptrain(freq) : _`

Where:

- `freq`: frequency
-

`triangleN`

TODO: Markdown doc in comments

`triangle`

Bandlimited triangle wave oscillator. Based on `triangleN(2)`. `triangle` is a standard Faust function.

Usage

`triangle(freq) : _`

Where:

- `freq`: frequency
-

Filter-Based Oscillators

Filter-Based Oscillators

Usage

`osc[b|r|rs|rc|s|w](f)`, where `f` = frequency in Hz.

References

- <http://lac.linuxaudio.org/2012/download/lac12-slides-jos.pdf>
- <https://ccrma.stanford.edu/~jos/pdf/lac12-paper-jos.pdf>

oscb

Sinusoidal oscillator based on the biquad.

Usage

`oscb(freq) : _`

Where:

- `freq`: frequency
-

oscrq

Sinusoidal (sine and cosine) oscillator based on 2D vector rotation, = undamped “coupled-form” resonator = lossless 2nd-order normalized ladder filter.

Usage

`oscrq(freq) : _,_`

Where:

- `freq`: frequency

Reference

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

oscrrs

Sinusoidal (sine) oscillator based on 2D vector rotation, = undamped “coupled-form” resonator = lossless 2nd-order normalized ladder filter.

Usage

`oscrrs(freq) : _`

Where:

- `freq`: frequency

Reference

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

osrcr

Sinusoidal (cosine) oscillator based on 2D vector rotation, = undamped “coupled-form” resonator = lossless 2nd-order normalized ladder filter.

Usage

`osrcr(freq) : _`

Where:

- `freq`: frequency

Reference

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

osc

Default sine wave oscillator (same as `oscrs`). `osc` is a standard Faust function.

Usage

`osc(freq) : _`

Where:

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

oscs

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

Waveguide-Resonator-Based Oscillators

Sinusoidal oscillator based on the waveguide resonator `wgr`.

`oscw`

Sinusoidal oscillator based on the waveguide resonator `wgr`. Unit-amplitude cosine oscillator.

Usage

`oscwc(freq) : _`

Where:

- `freq`: frequency

Reference

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

`oscws`

Sinusoidal oscillator based on the waveguide resonator `wgr`. Unit-amplitude sine oscillator

Usage

`oscws(freq) : _`

Where:

- `freq`: frequency

Reference

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

oscwq

Sinusoidal oscillator based on the waveguide resonator **wgr**. Unit-amplitude cosine and sine (quadrature) oscillator.

Usage

oscwq(freq) : _

Where:

- **freq**: frequency

Reference

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

oscw

Sinusoidal oscillator based on the waveguide resonator **wgr**. Unit-amplitude cosine oscillator (default)

Usage

oscw(freq) : _

Where:

- **freq**: frequency

Reference

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

lf_sawpos

Simple sawtooth waveform oscillator between 0 and 1.

Usage

`lf_sawpos(freq)`

Where:

- `freq`: frequency
-

`lf_saw`

Simple sawtooth waveform. `lf_saw` is a standard Faust function.

Usage

`lf_saw(freq)`

Where:

- `freq`: frequency
-

`lf_triangle`

Positive unit-amplitude LF triangle wave `lf_triangle` is a standard Faust function.

Usage

`lf_triangle(freq) : _`

Where:

- `freq`: frequency in Hz
-

phaflangers.lib

A library of phasor and flanger effects. Its official prefix is `pf`.

Functions Reference

flanger_mono

Mono flanging effect.

Usage:

```
_ : flanger_mono(dmax,curdel,depth,fb,invert) : _;
```

Where:

- **dmax**: maximum delay-line length (power of 2) - 10 ms typical
- **curdel**: current dynamic delay (not to exceed dmax)
- **depth**: effect strength between 0 and 1 (1 typical)
- **fb**: feedback gain between 0 and 1 (0 typical)
- **invert**: 0 for normal, 1 to invert sign of flanging sum

Reference

<https://ccrma.stanford.edu/~jos/pasp/Flanging.html>

flanger_stereo

Stereo flanging effect. **flanger_stereo** is a standard Faust function.

Usage:

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

Where:

- **dmax**: maximum delay-line length (power of 2) - 10 ms typical
- **curdel**: current dynamic delay (not to exceed dmax)
- **depth**: effect strength between 0 and 1 (1 typical)
- **fb**: feedback gain between 0 and 1 (0 typical)
- **invert**: 0 for normal, 1 to invert sign of flanging sum

Reference

<https://ccrma.stanford.edu/~jos/pasp/Flanging.html>

phaser2_mono

Mono phasing effect.

Phaser

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

Where:

- **Notches**: number of spectral notches (MACRO ARGUMENT - not a signal)
- **phase**: phase of the oscillator (0-1)
- **width**: approximate width of spectral notches in Hz
- **frqmin**: approximate minimum frequency of first spectral notch in Hz
- **fratio**: ratio of adjacent notch frequencies
- **frqmax**: approximate maximum frequency of first spectral notch in Hz
- **speed**: LFO frequency in Hz (rate of periodic notch sweep cycles)
- **depth**: effect strength between 0 and 1 (1 typical) (aka “intensity”) when depth=2, “vibrato mode” is obtained (pure allpass chain)
- **fb**: feedback gain between -1 and 1 (0 typical)
- **invert**: 0 for normal, 1 to invert sign of flanging sum

Reference:

- <https://ccrma.stanford.edu/~jos/pasp/Phasing.html>
 - http://www.geofex.com/Article_Folders/phasers/phase.html
 - ‘An Allpass Approach to Digital Phasing and Flanging’, Julius O. Smith III, Proc. Int. Computer Music Conf. (ICMC-84), pp. 103-109, Paris, 1984.
 - CCRMA Tech. Report STAN-M-21: <https://ccrma.stanford.edu/STANM/stanms/stanm21/>
-

phaser2_stereo

Stereo phasing effect. **phaser2_stereo** is a standard Faust function.

Phaser

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

Where:

- **Notches**: number of spectral notches (MACRO ARGUMENT - not a signal)
- **phase**: phase of the oscillator (0-1)

- **width**: approximate width of spectral notches in Hz
- **frqmin**: approximate minimum frequency of first spectral notch in Hz
- **fratio**: ratio of adjacent notch frequencies
- **frqmax**: approximate maximum frequency of first spectral notch in Hz
- **speed**: LFO frequency in Hz (rate of periodic notch sweep cycles)
- **depth**: effect strength between 0 and 1 (1 typical) (aka “intensity”) when depth=2, “vibrato mode” is obtained (pure allpass chain)
- **fb**: feedback gain between -1 and 1 (0 typical)
- **invert**: 0 for normal, 1 to invert sign of flanging sum

Reference:

- <https://ccrma.stanford.edu/~jos/pasp/Phasing.html>
- http://www.geofex.com/Article_Folders/phasers/phase.html
- ‘An Allpass Approach to Digital Phasing and Flanging’, Julius O. Smith III, Proc. Int. Computer Music Conf. (ICMC-84), pp. 103-109, Paris, 1984.
- CCRMA Tech. Report STAN-M-21: <https://ccrma.stanford.edu/STANM/stanms/stanm21/>

physmodels.lib

Faust physical modeling library; Its official prefix is **pm**.

This library provides an environment to facilitate physical modeling of musical instruments. It contains dozens of functions implementing low and high level elements going from a simple waveguide to fully operational models with built-in UI, etc.

It is organized as follows:

- **Global Variables**: Useful pre-defined variables for physical modeling (e.g., speed of sound, etc.).
- **Conversion Tools**: Conversion functions specific to physical modeling (e.g., length to frequency, etc.).
- **Bidirectional Utilities**: Functions to create bidirectional block diagrams for physical modeling.
- **Basic Elements**: waveguides, specific types of filters, etc.
- **String Instruments**: various types of strings (e.g., steel, nylon, etc.), bridges, guitars, etc.
- **Bowed String Instruments**: parts and models specific to bowed string instruments (e.g., bows, bridges, violins, etc.).
- **Wind Instrument**: parts and models specific to wind string instruments (e.g., reeds, mouthpieces, flutes, clarinets, etc.).
- **Exciters**: pluck generators, “blowers”, etc.

- Modal Percussions: percussion instruments based on modal models.
- Vocal Synthesis: functions for various vocal synthesis techniques (e.g., fof, source/filter, etc.) and vocal synthesizers.
- Misc Functions: any other functions that don't fit in the previous category (e.g., nonlinear filters, etc.)

This library is part of the Faust Physical Modeling ToolKit. More information on how to use this library can be found on this page: <https://ccrma.stanford.edu/~rmichon/pmFaust>. Tutorials on how to make physical models of musical instruments using Faust can be found here as well.

Global Variables

Useful pre-defined variables for physical modeling.

speedOfSound

Speed of sound in meters per second (340m/s).

maxLength

The default maximum length (3) in meters of strings and tubes used in this library. This variable should be overridden allow longer strings or tubes.

Conversion Tools

Useful conversion tools for physical modeling.

f2l

Frequency to length in meters.

Usage

f2l(freq) : distanceInMeters

Where:

- **freq**: the frequency
-

12f

Length in meters to frequency.

Usage

`12f(length) : freq`

Where:

- **length**: length/distance in meters
-

12s

Length in meters to number of samples.

Usage

`12s(l) : numberOfSamples`

Where:

- **l**: length in meters
-

Bidirectional Utilities

Set of fundamental functions to create bi-directional block diagrams in Faust. These elements are used as the basis of this library to connect high level elements (e.g., mouthpieces, strings, bridge, instrument body, etc.). Each block has 3 inputs and 3 outputs. The first input/output carry left going waves, the second input/output carry right going waves, and the third input/output is used to carry any potential output signal to the end of the algorithm.

basicBlock

Empty bidirectional block to be used with **chain**: 3 signals ins and 3 signals out.

Usage

`chain(basicBlock : basicBlock : etc.)`

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 = _,'_,-;
};
```

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.

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.

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.

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.

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.

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.

terminations

Creates terminations on both sides of a **chain** without closing the inputs and outputs of the bidirectional signals chain. As for **chain**, this function adds a 1 sample delay to the bidirectional signal, both ways. Of courses, this function can be nested within a **chain**.

Usage

```
terminations(a,b,c)
with{
    a = *(-1); // left termination
    b = chain(D : E : F); // bidirectional chain of blocks (D, E, F, etc.)
    c = *(-1); // right termination
};
```

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.)
};
```

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
};
```

closeIns

Closes the inputs of a bidirectional chain in all directions.

Usage

```
closeIns : chain(...) : _,_,_
```

closeOuts

Closes the outputs of a bidirectional chain in all directions except for the main signal output (3d output).

Usage

```
_,_,_ : chain(...) : _
```

endChain

Closes the inputs and outputs of a bidirectional chain in all directions except for the main signal output (3d output).

Usage

`endChain(chain(...)) : _`

Basic Elements

Basic elements for physical modeling (e.g., waveguides, specific filters, etc.).

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.
-

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.
-

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)
-

modeFilter

Resonant bandpass filter that can be used to implement a single resonance (mode).

Usage

```
_ : modeFilter(freq,t60,gain) : _
```

Where:

- **freq**: mode frequency
 - **t60**: mode resonance duration (in seconds)
 - **gain**: mode gain (0-1)
-

String Instruments

Low and high level string instruments parts. Most of the elements in this section can be used in a bidirectional chain.

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
-

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
-

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
-

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
-

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
-

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 independently 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
-

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 independently 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
-

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)
```

rStringRigidTermination

Bidirectional block implementing a right rigid string termination (no damping, just phase inversion).

Usage

```
chain(rStringRigidTermination : stringSegment : ...)
```

lStringRigidTermination

Bidirectional block implementing a left rigid string termination (no damping, just phase inversion).

Usage

```
chain(... : stringSegment : lStringRigidTermination)
```

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)
```

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 : ...)
```

guitarBridge

Bidirectional block implementing a simple acoustic guitar bridge. This bridge damps more high 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)
```

guitarNuts

Bidirectional block implementing a simple acoustic guitar nuts. This nuts damps more high 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 : ...)
```

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

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
-

ks_ui_MIDI

Ready-to-use, MIDI-enabled Karplus-Strong string with built-in UI.

Usage

`ks_ui_MIDI : _`

elecGuitarModel

A simple electric guitar model (without audio effects, of course) with selectable pluck position. This model implements a single string. Additional strings should be created by making a polyphonic applications out of this function. Pitch is changed by changing the length of the string and not through a finger model.

Usage

`elecGuitarModel(length,pluckPosition,excitation) : _`

Where:

- **length**: the length of the string in meters
 - **pluckPosition**: pluck position (0-1) (1 is on the bridge)
 - **excitation**: excitation signal
-

elecGuitar

A simple electric guitar model with steel strings (based on `elecGuitarModel`) implementing an excitation model. This model implements a single string. Additional strings should be created by making a polyphonic applications out of this function.

Usage

```
elecGuitar(length,pluckPosition,trigger) : _
```

Where:

- **length**: the length of the string in meters
 - **pluckPosition**: pluck position (0-1) (1 is on the bridge)
 - **gain**: gain of the pluck (0-1)
 - **trigger**: trigger signal (1 for on, 0 for off)
-

elecGuitar_ui_MIDI

Ready-to-use MIDI-enabled electric guitar physical model with built-in UI.

Usage

```
elecGuitar_ui_MIDI : _
```

guitarBody

WARNING: not implemented yet! Bidirectional block implementing a simple acoustic guitar body.

Usage

```
chain(... : guitarBody)
```

guitarModel

A simple acoustic guitar model with steel strings and selectable excitation position. This model implements a single string. Additional strings should be created by making a polyphonic applications out of this function. Pitch is changed by changing the length of the string and not through a finger model. WARNING: this function doesn't currently implement a body (just strings and bridge)

Usage

```
guitarModel(length,pluckPosition,excitation) : _
```

Where:

- **length**: the length of the string in meters
 - **pluckPosition**: pluck position (0-1) (1 is on the bridge)
 - **excitation**: excitation signal
-

guitar

A simple acoustic guitar model with steel strings (based on **guitarModel**) implementing an excitation model. This model implements a single string. Additional strings should be created by making a polyphonic applications out of this function.

Usage

```
guitar(length,pluckPosition,trigger) : _
```

Where:

- **length**: the length of the string in meters
 - **pluckPosition**: pluck position (0-1) (1 is on the bridge)
 - **gain**: gain of the excitation
 - **trigger**: trigger signal (1 for on, 0 for off)
-

guitar_ui_MIDI

Ready-to-use MIDI-enabled steel strings acoustic guitar physical model with built-in UI.

Usage

`guitar_ui_MIDI : _`

nylonGuitarModel

A simple acoustic guitar model with nylon strings and selectable excitation position. This model implements a single string. Additional strings should be created by making a polyphonic applications out of this function. Pitch is changed by changing the length of the string and not through a finger model. WARNING: this function doesn't currently implement a body (just strings and bridge)

Usage

`nylonGuitarModel(length,pluckPosition,excitation) : _`

Where:

- **length**: the length of the string in meters
 - **pluckPosition**: pluck position (0-1) (1 is on the bridge)
 - **excitation**: excitation signal
-

nylonGuitar

A simple acoustic guitar model with steel strings (based on `nylonGuitarModel`) implementing an excitation model. This model implements a single string. Additional strings should be created by making a polyphonic applications out of this function.

Usage

`nylonGuitar(length,pluckPosition,trigger) : _`

Where:

- **length**: the length of the string in meters
 - **pluckPosition**: pluck position (0-1) (1 is on the bridge)
 - **gain**: gain of the excitation (0-1)
 - **trigger**: trigger signal (1 for on, 0 for off)
-

nylonGuitar_ui_MIDI

Ready-to-use MIDI-enabled nylon strings acoustic guitar physical model with built-in UI.

Usage

nylonGuitar_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.

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
-

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)
-

bowInteraction

Bidirectional block implementing the interaction of a bow in a **chain**.

Usage

```
chain(... : stringSegment : bowInteraction(bowTable) : stringSegment : ...)
```

Where:

- **bowTable**: the bow table
-

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)
-

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)
-

violinNuts

Bidirectional block implementing simple violin nuts. This function is based on `bridgeFilter`.

Usage

```
chain(violinNuts : stringSegment : ...)
```

violinBridge

Bidirectional block implementing a simple violin bridge. This function is based on `bridgeFilter`.

Usage

```
chain(... : stringSegment : violinBridge
```

violinBody

Bidirectional block implementing a simple violin body (just a simple resonant lowpass filter).

Usage

```
chain(... : stringSegment : violinBridge : violinBody)
```

violinModel

Ready-to-use simple violin physical model. This model implements a single string. Additional strings should be created by making a polyphonic applications out of this function. Pitch is changed by changing the length of the string (and not through a finger model).

Usage

```
violinModel(stringLength,bowPressure,bowVelocity,bridgeReflexion,  
bridgeAbsorption,bowPosition) : _
```

Where:

- **stringLength**: the length of the string in meters
 - **bowVelocity**: velocity of the bow / excitation signal (0-1)
 - **bowPressure**: bow pressure on the string (0-1)
 - **bowPosition**: the position of the bow on the string (0-1)
-

violinModel_ui

Ready-to-use violin physical model with built-in UI.

Usage

```
violinModel_ui : _
```

violin_ui_MIDI

Ready-to-use MIDI-enabled violin physical model with built-in UI.

Usage

```
violin_ui_MIDI : _
```

Wind Instruments

Low and high level basic wind instruments parts. Most of the elements in this section can be used in a bidirectional chain.

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
-

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(offset,slope) : _
```

Where:

- **excitation**: an excitation signal
 - **offset**: table offset
 - **slope**: table slope
-

fluteJetTable

Extremely basic flute jet table.

Usage

```
excitation : fluteJetTable : _
```

Where:

- **excitation**: an excitation signal
-

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)
-

clarinetReed

Clarinet reed based on **reedTable** with controllable stiffness.

Usage

`excitation : clarinetReed(stiffness) : _`

Where:

- **excitation**: an excitation signal
 - **stiffness**: reed stiffness (0-1)
-

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)
-

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.)
-

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.)
-

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)
-

fluteHead

Simple flute head implementing waves reflexion.

Usage

```
chain(fluteHead : tube : ...)
```

fluteFoot

Simple flute foot implementing waves reflexion and dispersion.

Usage

```
chain(... : tube : fluteFoot)
```

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)

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.)
-

clarinet_ui

Ready-to-use clarinet physical model with built-in UI based on `clarinetModel`.

Usage

```
clarinet_ui : _
```

clarinet_ui_MIDI

Ready-to-use MIDI compliant clarinet physical model with built-in UI.

Usage

```
clarinet_ui_MIDI : _
```

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.)
-

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.)
-

brass_ui

Ready-to-use brass instrument physical model with built-in UI based on **brassModel**.

Usage

`brass_ui : _`

brass_ui_MIDI

Ready-to-use MIDI-controllable brass instrument physical model with built-in UI.

Usage

`brass_ui_MIDI : _`

`fluteModel`

A simple generic brass instrument physical model without tone holes (pitch is changed by changing the length of the tube of the instrument).

Usage

`fluteModel(tubeLength,lipsTension,mute,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.)
-

`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.)
-

`flute_ui`

Ready-to-use flute physical model with built-in UI based on `fluteModel`.

Usage

```
flute_ui : _
```

flute_ui_MIDI

Ready-to-use MIDI-controllable flute physical model with built-in UI.

Usage

```
brass_ui_MIDI : _
```

Exciters

Various kind of excitation signal generators.

impulseExcitation

Creates an impulse excitation of one sample.

Usage

```
gate = button('gate');  
impulseExcitation(gate) : chain;
```

Where:

- `gate`: a gate button
-

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)
-

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)
-

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)
-

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 cutoff frequency (Hz) (recommended: 2000)
-

blower_ui

Same as **blower** but with a built-in UI.

Usage

```
blower : somethingToBeBlown
```

Modal Percussions

High and low level functions for modal synthesis of percussion instruments.

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
-

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 (0-5, default: 0.5)
 - **gain**: gain of the strike
 - **trigger**: trigger signal (0: off, 1: on)
-

djembe_ui_MIDI

Simple MIDI controllable djembe physical model with built-in UI.

Usage

```
djembe_ui_MIDI : _
```

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)
-

marimbaResTube

Simple marimba resonance tube.

Usage

```
marimbaResTube(tubeLength,excitation)
```

Where:

- **tubeLength**: the length of the tube in meters
 - **excitation**: the excitation signal (audio in)
-

marimbaModel

Simple marimba physical model implementing a single tone bar connected to tube. This model is scalable and can be adapted to any size of bar/tube (see `marimbaBarModel` to know more about the limitations of this type of system).

Usage

```
excitation : marimbaModel(freq,exPos) : _
```

Where:

- **freq**: the frequency of the bar/tube couple
- **exPos**: excitation position (0-4)

marimba

Simple marimba physical model implementing a single tone bar connected to tube. This model is scalable and can be adapted to any size of bar/tube (see `marimbaBarModel` to know more about the limitations of this type of system).

This function also implement a virtual exciter to drive the model.

Usage

```
excitation : marimba(freq,strikePosition,strikeCutoff,strikeSharpness,gain,trigger) : _
```

Where:

- **excitation**: the excitation signal
- **freq**: the frequency of the bar/tube couple
- **strikePosition**: strike position (0-4)
- **strikeCutoff**: cutoff frequency of the strike genarator (recommended: ~7000Hz)
- **strikeSharpness**: shaarpness of the strike (recommened: ~0.25)
- **gain**: gain of the strike (0-1)
- **trigger** signal (0: off, 1: on)

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 : _
```

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(exPos,t60,t60DecayRatio,t60DecaySlope)
```

Where:

- **excitation**: the excitation signal
 - **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)
-

churchBell

Generic church bell modal model.

Modeled after T. Rossing and R. Perrin, Vibrations of Bells, Applied Acoustics 2, 1987.

Model height is 301 mm.

This model contains 7 excitation positions going linearly from the bottom to the top of the bell. Obviously, a model with more excitation position could be regenerated using `mesh2faust`.

This function also implement a virtual exciter to drive the model.

Usage

```
excitation : churchBell(strikePosition,strikeCutoff,strikeSharpness,gain,trigger) : _
```

Where:

- **excitation**: the excitation signal
 - **strikePosition**: strike position (0-6)
 - **strikeCutoff**: cutoff frequency of the strike genarator (recommended: ~7000Hz)
 - **strikeSharpness**: shaarpness of the strike (recommened: ~0.25)
 - **gain**: gain of the strike (0-1)
 - **trigger** signal (0: off, 1: on)
-

churchBell_ui

Church bell physical model based on `churchBell` with built-in UI.

Usage

`churchBell_ui` : _

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(exPos,t60,t60DecayRatio,t60DecaySlope)`

Where:

- `excitation`: the excitation signal
 - `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)
-

englishBell

English church bell modal model.

Modeled after D. Bartocha and . Baron, Influence of Tin Bronze Melting and Pouring Parameters on Its Properties and Bell' Tone, Archives of Foundry Engineering, 2016.

Model height is 1 m.

This model contains 7 excitation positions going linearly from the bottom to the top of the bell. Obviously, a model with more excitation position could be regenerated using `mesh2faust`.

This function also implement a virtual exciter to drive the model.

Usage

```
excitation : englishBell(strikePosition,strikeCutoff,strikeSharpness,gain,trigger) : _
```

Where:

- **excitation**: the excitation signal
 - **strikePosition**: strike position (0-6)
 - **strikeCutoff**: cutoff frequency of the strike genarator (recommended: ~7000Hz)
 - **strikeSharpness**: shaarpness of the strike (recommened: ~0.25)
 - **gain**: gain of the strike (0-1)
 - **trigger** signal (0: off, 1: on)
-

englishBell_ui

English church bell physical model based on `englishBell` with built-in UI.

Usage

```
englishBell_ui : _
```

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(exPos,t60,t60DecayRatio,t60DecaySlope)`

Where:

- **excitation**: the excitation signal
 - **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)
-

frenchBell

French church bell modal model.

Modeled after D. Bartocha and . Baron, Influence of Tin Bronze Melting and Pouring Parameters on Its Properties and Bell' Tone, Archives of Foundry Engineering, 2016.

Model height is 1 m.

This model contains 7 excitation positions going linearly from the bottom to the top of the bell. Obviously, a model with more excitation position could be regenerated using `mesh2faust`.

This function also implement a virtual exciter to drive the model.

Usage

`excitation : frenchBell(strikePosition,strikeCutoff,strikeSharpness,gain,trigger) : _`

Where:

- **excitation**: the excitation signal
 - **strikePosition**: strike position (0-6)
 - **strikeCutoff**: cutoff frequency of the strike genarator (recommended: ~7000Hz)
 - **strikeSharpness**: shaarpness of the strike (recommened: ~0.25)
 - **gain**: gain of the strike (0-1)
 - **trigger** signal (0: off, 1: on)
-

frenchBell_ui

French church bell physical model based on `frenchBell` with built-in UI.

Usage

frenchBell_ui : _

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(exPos,t60,t60DecayRatio,t60DecaySlope)`

Where:

- `excitation`: the excitation signal
 - `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)
-

germanBell

German church bell modal model.

Modeled after D. Bartocha and . Baron, Influence of Tin Bronze Melting and Pouring Parameters on Its Properties and Bell' Tone, Archives of Foundry Engineering, 2016.

Model height is 1 m.

This model contains 7 excitation positions going linearly from the bottom to the top of the bell. Obviously, a model with more excitation position could be regenerated using `mesh2faust`.

This function also implement a virtual exciter to drive the model.

Usage

`excitation : germanBell(strikePosition,strikeCutoff,strikeSharpness,gain,trigger) : _`

Where:

- **excitation**: the excitation signal
 - **strikePosition**: strike position (0-6)
 - **strikeCutoff**: cutoff frequency of the strike genarator (recommended: ~7000Hz)
 - **strikeSharpness**: shaarpness of the strike (recommened: ~0.25)
 - **gain**: gain of the strike (0-1)
 - **trigger** signal (0: off, 1: on)
-

`germanBell_ui`

German church bell physical model based on `germanBell` with built-in UI.

Usage

`germanBell_ui : _`

`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(exPos,t60,t60DecayRatio,t60DecaySlope)`

Where:

- **excitation**: the excitation signal
- **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)
-

russianBell

Russian church bell modal model.

Modeled after D. Bartocha and . Baron, Influence of Tin Bronze Melting and Pouring Parameters on Its Properties and Bell' Tone, Archives of Foundry Engineering, 2016.

Model height is 2 m.

This model contains 7 excitation positions going linearly from the bottom to the top of the bell. Obviously, a model with more excitation position could be regenerated using **mesh2faust**.

This function also implement a virtual exciter to drive the model.

Usage

```
excitation : russianBell(strikePosition,strikeCutoff,strikeSharpness,gain,trigger) : _
```

Where:

- **excitation**: the excitation signal
 - **strikePosition**: strike position (0-6)
 - **strikeCutoff**: cutoff frequency of the strike genarator (recommended: ~7000Hz)
 - **strikeSharpness**: shaarpness of the strike (recommened: ~0.25)
 - **gain**: gain of the strike (0-1)
 - **trigger** signal (0: off, 1: on)
-

russianBell_ui

Russian church bell physical model based on **russianBell** with built-in UI.

Usage

```
russianBell_ui : _
```

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(exPos,t60,t60DecayRatio,t60DecaySlope)`

Where:

- **excitation**: the excitation signal
 - **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)
-

standardBell

Standard church bell modal model.

Modeled after T. Rossing and R. Perrin, Vibrations of Bells, Applied Acoustics 2, 1987.

Model height is 1.8 m.

This model contains 7 excitation positions going linearly from the bottom to the top of the bell. Obviously, a model with more excitation position could be regenerated using `mesh2faust`.

This function also implement a virtual exciter to drive the model.

Usage

`excitation : standardBell(strikePosition,strikeCutoff,strikeSharpness,gain,trigger) : _`

Where:

- **excitation**: the excitation signal

- **strikePosition**: strike position (0-6)
 - **strikeCutoff**: cutoff frequency of the strike generator (recommended: ~7000Hz)
 - **strikeSharpness**: sharpness of the strike (recommended: ~0.25)
 - **gain**: gain of the strike (0-1)
 - **trigger** signal (0: off, 1: on)
-

standardBell_ui

Standard church bell physical model based on **standardBell** with built-in UI.

Usage

standardBell_ui : _

Vocal Synthesis

Vocal synthesizer functions (source/filter, fof, etc.).

formantFilter

Formant filter based on a bank of resonant bandpass. 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.

The formant data used here come from the CSOUND manual <http://www.csounds.com/manual/html/>.

Usage

_ : **formantFilter**(voiceType,vowel) : _

Where:

- **voiceType**: the voice type (0: alto, 1: bass, 2: countertenor, 3: soprano, 4: tenor)
 - **vowel**: the vowel (0: a, 1: e, 2: i, 3: o, 4: u)
-

SFFormantModel

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. 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

SFFormantModel(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)
-

SFFormantModel_ui

Ready-to-use source-filter vocal synthesizer with built-in user interface.

Usage

SFFormantModel_ui : _

SFFormantModel_ui_MIDI

Ready-to-use MIDI-controllable source-filter vocal synthesizer.

Usage

SFFormantModel_ui_MIDI : _

Misc Functions

Various miscellaneous functions.

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)
-

reverbs.lib

A library of reverb effects. Its official prefix is **re**.

Schroeder Reverberators

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 : _,_,_,_
```

satrev

This artificial reverberator take a mono signal and output stereo (**satrev**) and quad (**jcrev**). They were implemented by John Chowning in the MUS10 computer-music language (descended from Music V by Max Mathews). They are Schroeder Reverberators, well tuned for their size. Nowadays, the more expensive freeverb is more commonly used (see the Faust examples directory).

satrev was made from a listing of “SATREV”, dated May 15, 1971, which was recovered from an old SAIL DART backup tape. John Chowning thinks this might be the one used on his often-heard brass canon sound examples, one of which can be found at https://ccrma.stanford.edu/~jos/wav/FM_BrassCanon2.wav

Usage

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

Feedback Delay Network (FDN) Reverberators

fdnrev0

Pure Feedback Delay Network Reverberator (generalized for easy scaling). **fdnrev0** is a standard Faust function.

Usage

```
<1,2,4,...,N signals> <:  
fdnrev0(MAXDELAY,delay,BBS0,freqs,durs,loopgainmax,nonl) :>  
<1,2,4,...,N signals>
```

Where:

- **N**: 2, 4, 8, ... (power of 2)
- **MAXDELAY**: power of 2 at least as large as longest delay-line length
- **delays**: N delay lines, N a power of 2, lengths perferably coprime
- **BBS0**: 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 “squench” the reverb
- **nonl**: nonlinearity (0 to 0.999..., 0 being linear)

Reference

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

zita_rev_fdn

Internal 8x8 late-reverberation FDN used in the FOSS Linux reverb zita-rev1 by Fons Adriaensen fons@linuxaudio.org. This is an FDN reverb with allpass comb filters in each feedback delay in addition to the damping filters.

Usage

`bus(8) : zita_rev_fdn(f1,f2,t60dc,t60m,fsmx) : 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)
- `fsmx`: maximum sampling rate to be used (Hz)

Reference

- <http://www.kokkinizita.net/linuxaudio/zita-rev1-doc/quickguide.html>
 - https://ccrma.stanford.edu/~jos/pasp/Zita_Rev1.html
-

zita_rev1_stereo

Extend `zita_rev_fdn` to include `zita_rev1` input/output mapping in stereo mode. `zita_rev1_stereo` is a standard Faust function.

Usage

`_,_ : zita_rev1_stereo(rdel,f1,f2,t60dc,t60m,fsmx) : _,_`

Where:

`rdel` = delay (in ms) before reverberation begins (e.g., 0 to ~100 ms) (remaining args and refs as for `zita_rev_fdn` above)

zita_rev1_ambi

Extend `zita_rev_fdn` to include `zita_rev1` input/output mapping in “ambisonics mode”, as provided in the Linux C++ version.

Usage

```
_,_ : zita_rev1_ambi(rgxyz,rdel,f1,f2,t60dc,t60m,fsmx) : _,_,_,_
```

Where:

`rgxyz` = relative gain of lanes 1,4,2 to lane 0 in output (e.g., -9 to 9) (remaining args and references as for `zita_rev1_stereo` above)

Freeverb

mono_freeverb

A simple Schroeder reverberator primarily developed by “Jezar at Dreampoint” that is extensively used in the free-software world. It uses four Schroeder allpasses in series and eight parallel Schroeder-Moorer filtered-feedback comb-filters for each audio channel, and is said to be especially well tuned.

`mono_freeverb` is a standard Faust function.

Usage

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

Where:

- `fb1`: coefficient of the lowpass comb filters (0-1)
- `fb2`: coefficient of the allpass comb filters (0-1)
- `damp`: damping of the lowpass comb filter (0-1)
- `spread`: spatial spread in number of samples (for stereo)

License

While this version is licensed LGPL (with exception) along with other GRAME library functions, the file `freeverb.dsp` in the examples directory of older Faust distributions, such as `faust-0.9.85`, was released under the BSD license, which is less restrictive.

stereo_freeverb

A simple Schroeder reverberator primarily developed by “Jezar at Dreampoint” that is extensively used in the free-software world. It uses four Schroeder allpasses in series and eight parallel Schroeder-Moorer filtered-feedback comb-filters for each audio channel, and is said to be especially well tuned.

Usage

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

Where:

- **fb1**: coefficient of the lowpass comb filters (0-1)
 - **fb2**: coefficient of the allpass comb filters (0-1)
 - **damp**: damping of the lowpass comb filter (0-1)
 - **spread**: spatial spread in number of samples (for stereo)
-

routes.lib

A library of basic elements to handle signal routing in Faust. Its official prefix is **ro**.

Functions Reference

cross

Cross n signals: $(x_1, x_2, \dots, x_n) \rightarrow (x_n, \dots, x_2, x_1)$. **cross** is a standard Faust function.

Usage

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

Where:

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

Note

Special case: `cross2`:

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

`crossnn`

Cross two `bus(n)`s.

Usage

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

Where:

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

`crossn1`

Cross `bus(n)` and `bus(1)`.

Usage

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

Where:

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

`interleave`

Interleave *rowcol* cables from column order to row order. *input* : $x(0)$, $x(1)$, $x(2)$..., $x(\text{rowcol}-1)$ *output*: $x(0+0\text{row})$, $x(0+1\text{row})$, $x(0+2\text{row})$, ..., $x(1+0\text{row})$, $x(1+1\text{row})$, $x(1+2\text{row})$, ...

Usage

```
_,_,_,_,_,_ : interleave(row,column) : _,_,_,_,_,_
```

Where:

- **row**: the number of row (int, known at compile time)
 - **column**: the number of column (int, known at compile time)
-

butterfly

Addition (first half) then subtraction (second half) of interleaved signals.

Usage

`_,_,_,_ : butterfly(n) : _,_,_,_`

Where:

- **n**: size of the butterfly (n is int, even and known at compile time)
-

hadamard

Hadamard matrix function of size $n = 2^k$.

Usage

`_,_,_,_ : hadamard(n) : _,_,_,_`

Where:

- **n**: 2^k , size of the matrix (int, must be known at compile time)

Note:

Implementation contributed by Remy Muller.

recursivize

Create a recursion from two arbitrary processors p and q.

Usage

`_,_ : recursivize(p,q) : _,_`

Where:

- `p`: the forward arbitrary processor
 - `q`: the feedback arbitrary processor
-

signals.lib

A library of basic elements to handle signals in Faust. Its official prefix is `si`.

Functions Reference

bus

`n` parallel cables. `bus` is a standard Faust function.

Usage

`bus(n)`

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

Where:

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

block

Block - terminate `n` signals. `block` is a standard Faust function.

Usage

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

Where:

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

interpolate

Linear interpolation between two signals.

Usage

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

Where:

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

smoo

Smoothing function based on **smooth** ideal to smooth UI signals (sliders, etc.) down. **smoo** is a standard Faust function.

Usage

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

polySmooth

A smoothing function based on **smooth** that doesn't smooth when a trigger signal is given. This is very useful when making polyphonic synthesizer to make sure that the value of the parameter is the right one when the note is started.

Usage

```
hslider(...) : polysmooth(g,s,d) : _
```

Where:

- **g**: the gate/trigger signal used when making polyphonic synths
 - **s**: the smoothness (see **smooth**)
 - **d**: the number of samples to wait before the signal start being smoothed after **g** switched to 1
-

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**)
-

bsmooth

Block smooth linear interpolation during a block of samples.

Usage

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

dot

Dot product for two vectors of size n.

Usage

```
_,_,_,_,_,_ : dot(n) : _
```

Where:

- **n**: size of the vectors (int, must be known at compile time)
-

smooth

Exponential smoothing by a unity-dc-gain one-pole lowpass. **smooth** is a standard Faust function.

Usage:

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

Where:

- **tau**: desired smoothing time constant in seconds, or

```
hslider(...) : smooth(s) : _
```

Where:

- **s**: smoothness between 0 and 1. $s=0$ for no smoothing, $s=0.999$ is “very smooth”, $s>1$ is unstable, and $s=1$ yields the zero signal for all inputs. The exponential time-constant is approximately $1/(1-s)$ samples, when s is close to (but less than) 1.

Reference:

https://ccrma.stanford.edu/~jos/mdft/Convolution_Example_2_ADSR.html

lag_ud

Lag filter with separate times for up and down.

Usage

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

spats.lib

This library contains a collection of tools for sound spatialization. Its official prefix is **sp**.

panner

A simple linear stereo panner. **panner** is a standard Faust function.

Usage

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

Where:

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

spat

GMEM SPAT: n-outputs spatializer. **spat** is a standard Faust function.

Usage

`_ : spat(n,r,d) : _,_,...`

Where:

- `n`: number of outputs
 - `r`: rotation (between 0 et 1)
 - `d`: distance of the source (between 0 et 1)
-

stereoize

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

Usage

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

Where:

- `p`: the arbitrary processor
-

synths.lib

This library contains a collection of envelope generators. Its official prefix is **sy**.

popFilterPerc

A simple percussion instrument based on a “popped” resonant bandpass filter. **popFilterPerc** is a standard Faust function.

Usage

```
popFilterDrum(freq,q,gate) : _;
```

Where:

- **freq**: the resonance frequency of the instrument
 - **q**: the q of the res filter (typically, 5 is a good value)
 - **gate**: the trigger signal (0 or 1)
-

dubDub

A simple synth based on a sawtooth wave filtered by a resonant lowpass. **dubDub** is a standard Faust function.

Usage

```
dubDub(freq,ctFreq,q,gate) : _;
```

Where:

- **freq**: frequency of the sawtooth
 - **ctFreq**: cutoff frequency of the filter
 - **q**: Q of the filter
 - **gate**: the trigger signal (0 or 1)
-

sawTrombone

A simple trombone based on a lowpassed sawtooth wave. **sawTrombone** is a standard Faust function.

Usage

```
sawTrombone(att,freq,gain,gate) : _
```

Where:

- **att**: exponential attack duration in s (typically 0.01)
- **freq**: the frequency

- **gain**: the gain (0-1)
 - **gate**: the gate (0 or 1)
-

combString

Simplest string physical model ever based on a comb filter. **combString** is a standard Faust function.

Usage

```
combString(freq,res,gate) : _;
```

Where:

- **freq**: the frequency of the string
 - **res**: string T60 (resonance time) in second
 - **gate**: trigger signal (0 or 1)
-

additiveDrum

A simple drum using additive synthesis. **additiveDrum** is a standard Faust function.

Usage

```
additiveDrum(freq,freqRatio,gain,harmDec,att,rel,gate) : _
```

Where:

- **freq**: the resonance frequency of the drum
 - **freqRatio**: a list of ratio to choose the frequency of the mode in function of **freq** e.g.(1 1.2 1.5 ...). The first element should always be one (fundamental).
 - **gain**: the gain of each mode as a list (1 0.9 0.8 ...). The first element is the gain of the fundamental.
 - **harmDec**: harmonic decay ratio (0-1): configure the speed at which higher modes decay compare to lower modes.
 - **att**: attack duration in second
 - **rel**: release duration in second
 - **gate**: trigger signal (0 or 1)
-

fm

An FM synthesizer with an arbitrary number of modulators connected as a sequence. **fm** is a standard Faust function.

Usage

```
freqs = (300,400,...);  
indices = (20,...);  
fm(freqs,indices) : _
```

Where:

- **freqs**: a list of frequencies where the first one is the frequency of the carrier and the others, the frequency of the modulator(s)
 - **indices**: the indices of modulation (Nfreqs-1)
-

vaeffects.lib

A library of virtual analog filter effects. Its official prefix is **ve**.

Functions Reference

moog_vcf

Moog “Voltage Controlled Filter” (VCF) in “analog” form. Moog VCF implemented using the same logical block diagram as the classic analog circuit. As such, it neglects the one-sample delay associated with the feedback path around the four one-poles. This extra delay alters the response, especially at high frequencies (see reference [1] for details). See **moog_vcf_2b** below for a more accurate implementation.

Usage

```
moog_vcf(res,fr)
```

Where:

- **fr**: corner-resonance frequency in Hz (less than SR/6.3 or so)
- **res**: Normalized amount of corner-resonance between 0 and 1 (0 is no resonance, 1 is maximum)

References

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

moog_vcf_2b[n]

Moog “Voltage Controlled Filter” (VCF) as two biquads. Implementation of the ideal Moog VCF transfer function factored into second-order sections. As a result, it is more accurate than `moog_vcf` above, but its coefficient formulas are more complex when one or both parameters are varied. Here, `res` is the fourth root of that in `moog_vcf`, so, as the sampling rate approaches infinity, `moog_vcf(res,fr)` becomes equivalent to `moog_vcf_2b[n](res^4,fr)` (when `res` and `fr` are constant). `moog_vcf_2b` uses two direct-form biquads (`tf2`). `moog_vcf_2bn` uses two protected normalized-ladder biquads (`tf2np`).

Usage

```
moog_vcf_2b(res,fr)
moog_vcf_2bn(res,fr)
```

Where:

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

wah4

Wah effect, 4th order. `wah4` is a standard Faust function.

Usage

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

Where:

- `fr`: resonance frequency in Hz

Reference

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

autowah

Auto-wah effect. **autowah** is a standard Faust function.

Usage

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

Where:

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

crybaby

Digitized CryBaby wah pedal. **crybaby** is a standard Faust function.

Usage

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

Where:

- **wah**: “pedal angle” from 0 to 1

Reference

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

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
-

Licenses

STK 4.3 License

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the “Software”), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

Any person wishing to distribute modifications to the Software is asked to send the modifications to the original developer so that they can be incorporated into the canonical version. For software copyrighted by Julius O. Smith III, email your modifications to jos@ccrma.stanford.edu. This is, however, not a binding provision of this license.

THE SOFTWARE IS PROVIDED “AS IS”, WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

LGPL License

This program is free software; you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation; either version 2.1 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY

ITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.

You should have received a copy of the GNU Lesser General Public License along with the GNU C Library; if not, write to the Free Software Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA.