



lib_audio_dsp - DSP Components

Release: 1.0.0

Publication Date: 2024/07/30

Document Number: XM-015034-PC

Table of Contents

1	DSP Stages	3
1.1	Biquad	3
1.2	Cascaded Biquads	6
1.3	Compressor	8
1.4	Compressor Sidechain	9
1.5	Envelope Detector	10
1.5.1	EnvelopeDetectorPeak	10
1.5.2	EnvelopeDetectorRMS	10
1.6	FIR	11
1.7	Limiter	12
1.7.1	LimiterRMS	12
1.7.2	LimiterPeak	13
1.7.3	HardLimiterPeak	13
1.7.4	Clipper	14
1.8	Noise Gate	15
1.9	Noise Suppressor Expander	16
1.10	Reverb	17
1.11	Signal Chain	19
1.11.1	Bypass	19
1.11.2	Fork	20
1.11.3	Mixer	20
1.11.4	Adder	21
1.11.5	Subtractor	21
1.11.6	FixedGain	22
1.11.7	VolumeControl	22
1.11.8	Switch	23
1.11.9	Delay	24
2	DSP Modules	25
2.1	Library Q Format	25
2.2	Biquad Filters	26
2.2.1	Single Biquad	26
2.2.2	Cascaded Biquads	28
2.3	Dynamic Range Control	29
2.3.1	Attack and Release Times	29
2.3.2	Envelope Detectors	29
2.3.2.1	Peak Envelope Detector	30
2.3.2.2	RMS Envelope Detector	31
2.3.3	Clipper	33
2.3.4	Limiters	34
2.3.4.1	Peak Limiter	35
2.3.4.2	Hard Peak Limiter	36
2.3.4.3	RMS Limiter	39
2.3.5	Compressors	40
2.3.5.1	RMS Compressor	41
2.3.5.2	Sidechain RMS Compressor	43
2.3.6	Expanders	45
2.3.6.1	Noise Gate	46
2.3.6.2	Noise Suppressor/Expander	48
2.4	Finite Impulse Response Filters	51
2.4.1	FIR Direct	51

2.5	Reverb	52
2.5.1	Reverb Room	52
2.6	Signal Chain Components	56
2.6.1	Adder	56
2.6.2	Subtractor	57
2.6.3	Fixed Gain	58
2.6.4	Mixer	59
2.6.5	Volume Control	61
2.6.6	Delay	64
2.7	Python module base class	66

Introduction

lib_audio_dsp provides many common signal processing function optimised for xcore. These can be combined together to make complex audio pipelines for many different applications, such as home audio, music production, voice processing and AI feature extraction.

The library is split into 2 levels of API, DSP stages and DSP modules. Both APIs provide similar DSP functionality, but are suited to different use cases.

DSP Stages

The higher-level APIs are called [DSP Stages](#). These stages are designed to work with the Python DSP pipeline tool. This tool allows developers to quickly and easily create, test, and deploy DSP pipelines without needing to write a lot of code. By using DSP stages, the user can build complex audio processing workflows in a short amount of time, making it ideal for rapid prototyping and development.

DSP Modules

The lower-level APIs are called [DSP Modules](#). They are meant to be used as an API directly in cases where the Python DSP pipeline tool is not used. These modules can be useful when integrating DSP function into an existing system, or as a starting point for creating bespoke DSP functions.

Precision

Note: For fixed point Q formats this document uses the format QM.N, where M is the number of bits before the decimal point (excluding the sign bit), and N is the number of bits after the decimal point. For an int32 number, M+N=31.

By default, the signal processing in the audio pipeline is carried out at 32 bit fixed point precision in Q4.27 format. Assuming a 24 bit input signal in Q0.24 format, this gives 4 bits of internal headroom in the audio pipeline, which is equivalent to 24 dB. The output of the audio pipeline will be clipped back to Q0.24 before returning. For more precision, the pipeline can be configured to run with no headroom in Q0.31 format, but this requires manual headroom management. More information on setting the Q format can be found in the [Library Q Format](#) section.

DSP algorithms are implemented either on the XS3 CPU or VPU (vector processing unit).

CPU algorithms are typically implemented as 32-bit x 32-bit operations into 64-bit results and accumulators, before rounding back to 32-bit outputs.

The VPU allows for 8 simultaneous operations, with a small cost in precision. VPU algorithms are typically implemented as 32-bit x 32-bit operations into 34-bit results and 40-bit accumulators, before rounding back to 32-bit outputs.

Latency

The latency of the DSP pipeline is dependent on the number of threads. By default, the DSP pipeline is configured for one sample of latency per thread. All current DSP modules have zero inbuilt latency (except where specified e.g. delays stages). For pipelines that fit on a single thread, this means the total pipeline latency is 1 sample.

The pipeline can also be configured to use a higher frame size. This increases latency, but can reduce compute for simple functions. For a pipeline consisting of just biquads:

- Frame size = 1, latency = 1 sample, compute = 25 biquads per thread @ 48kHz.
- Frame size = 8, latency = 8 samples, compute = 60 biquads per thread @ 48kHz.

1 DSP Stages

DSP stages are high level blocks for use in the Python DSP pipeline tool. Each Stage has a Python and C implementation, allowing pipelines to be rapidly prototyped in Python before being easily deployed to hardware in C. The audio performance of both implementations is equivalent.

Most stages have parameters that can be changed at runtime, and the available parameters are outlined in the documentation.

All the DSP stages can be imported into a Python file using: .. code-block:: console

```
from audio_dsp.stages import *
```

The following DSP stages are available for use in the Python DSP pipeline design.

1.1 Biquad

Biquad Stages can be used for basic audio filters.

class `audio_dsp.stages.biquad.Biquad(**kwargs)`

A second order biquadratic filter, which can be used to make many common second order filters. The filter is initialised in a bypass state, and the `make_*` methods can be used to calculate the coefficients.

This Stage implements a direct form 1 biquad filter: $a_0*y[n] = b_0*x[n] + b_1*x[n-1] + b_2*x[n-2] - a_1*y[n-1] - a_2*y[n-2]$

For efficiency the biquad coefficients are normalised by a_0 and the output a coefficients multiplied by -1.

Attributes

dsp_block

[`audio_dsp.dsp.biquad.biquad`] The DSP block class; see [Single Biquad](#) for implementation details.

make_allpass(*f: float, q: float*) → Biquad

Make this biquad an all pass filter.

Parameters

f

[float] Center frequency of the filter in Hz.

q

[float] Q factor of the filter.

make_bandpass(*f: float, bw: float*) → Biquad

Make this biquad a second order bandpass filter.

Parameters

f

[float] Center frequency of the filter in Hz.

bw

[float] Bandwidth of the filter in octaves.

make_bandstop(*f: float, bw: float*) → Biquad

Make this biquad a second order bandstop filter.

Parameters

f
[float] Center frequency of the filter in Hz.

bw
[float] Bandwidth of the filter in octaves.

make_bypass() → Biquad

Make this biquad a bypass by setting the b0 coefficient to 1.

make_constant_q(*f: float, q: float, boost_db: float*) → Biquad

Make this biquad a peaking filter with constant Q.

Constant Q means that the bandwidth of the filter remains constant as the gain varies. It is commonly used for graphic equalisers.

Parameters

f
[float] Center frequency of the filter in Hz.

q
[float] Q factor of the filter.

boost_db
[float] Gain of the filter in decibels.

make_highpass(*f: float, q: float*) → Biquad

Make this biquad a second order high pass filter.

Parameters

f
[float] Cutoff frequency of the filter in Hz.

q
[float] Q factor of the filter roll-off. 0.707 is equivalent to a Butterworth response.

make_highshelf(*f: float, q: float, boost_db: float*) → Biquad

Make this biquad a second order high shelf filter.

The Q factor is defined in a similar way to standard high pass, i.e. > 0.707 will yield peakiness (where the shelf response does not monotonically change). The level change at f will be boost_db/2.

Parameters

f
[float] Cutoff frequency of the shelf in Hz, where the gain is boost_db/2

q
[float] Q factor of the filter.

boost_db
[float] Gain of the filter in decibels.

make_linkwitz(*f0: float, q0: float, fp: float, qp: float*) → Biquad

Make this biquad a Linkwitz Transform biquad filter.

The Linkwitz Transform changes the low frequency cutoff of a filter, and is commonly used to change the low frequency roll off slope of a loudspeaker. When applied to a loudspeaker, it will change the cutoff frequency from f0 to fp, and the Q factor from q0 to qp.

Parameters

f0
[float] The original cutoff frequency of the filter in Hz.

q0
[float] The original quality factor of the filter at f0.

fp
[float] The target cutoff frequency for the filter in Hz.

qp
[float] The target quality factor for the filter.

make_lowpass(*f: float, q: float*) → Biquad

Make this biquad a second order low pass filter.

Parameters

f
[float] Cutoff frequency of the filter in Hz.

q
[float] Q factor of the filter roll-off. 0.707 is equivalent to a Butterworth response.

make_lowshelf(*f: float, q: float, boost_db: float*) → Biquad

Make this biquad a second order low shelf filter.

The Q factor is defined in a similar way to standard low pass, i.e. > 0.707 will yield peakiness (where the shelf response does not monotonically change). The level change at f will be boost_db/2.

Parameters

f
[float] Cutoff frequency of the shelf in Hz, where the gain is boost_db/2

q
[float] Q factor of the filter.

boost_db
[float] Gain of the filter in decibels.

make_notch(*f: float, q: float*) → Biquad

Make this biquad a notch filter.

Parameters

f
[float] Center frequency of the filter in Hz.

q
[float] Q factor of the filter.

make_peaking(*f: float, q: float, boost_db: float*) → Biquad

Make this biquad a peaking filter.

Parameters

f
[float] Center frequency of the filter in Hz.

q
[float] Q factor of the filter.

boost_db
[float] Gain of the filter in decibels.

Control

The following runtime control parameters are available for the Biquad Stage:

Command ID macro	Payload length	Description
CMD_BIQUAD_LEFT_SHIFT	sizeof(int)	The number of bits to shift the output left by, in order to compensate for any right shift applied to the biquad b coefficients.
CMD_BIQUAD_FILTER_COEFFS	sizeof(int32_t)*[5]	The normalised biquad filter coefficients, in the order $[b0, b1, b2, -a1, -a2]/a0$. The coefficients should be in Q1.30 format. If the maximum b coefficient magnitude is greater than 2.0, the b coefficients should be right shifted to fit in Q1.30 format, and the shift value passed as left_shift to correct the gain after filtering.
CMD_BIQUAD_RESERVED	sizeof(int32_t)*[3]	Reserved memory to ensure the VPU receives 8 DWORD_ALIGNED coefficients.

1.2 Cascaded Biquads

Cascaded biquad Stages consist of several biquad filters connected together in series.

class `audio_dsp.stages.cascaded_biquads.CascadedBiquads(**kwargs)`

8 cascaded biquad filters. This allows up to 8 second order biquad filters to be run in series.

This can be used for either:

- an Nth order filter built out of cascaded second order sections
- a parametric EQ, where several biquad filters are used at once.

For documentation on the individual biquad filters, see `audio_dsp.stages.biquad.Biquad` and `audio_dsp.dsp.biquad.biquad`

Attributes

dsp_block

[`audio_dsp.dsp.cascaded_biquad.cascaded_biquad`] The DSP block class; see [Cascaded Biquads](#) for implementation details.

make_butterworth_highpass(*N: int, fc: float*) → `CascadedBiquads`

Configure this instance as an Nth order Butterworth highpass filter using N/2 cascaded biquads.

For details on the implementation, see `audio_dsp.dsp.cascaded_biquads.make_butterworth_highpass`

Parameters

N

[int] Filter order, must be even

fc

[float] -3 dB frequency in Hz.

make_butterworth_lowpass(*N: int, fc: float*) → `CascadedBiquads`

Configure this instance as an Nth order Butterworth lowpass filter using N/2 cascaded biquads.

For details on the implementation, see `audio_dsp.dsp.cascaded_biquads.make_butterworth_lowpass`

Parameters

N

[int] Filter order, must be even

fc

[float] -3 dB frequency in Hz.

make_parametric_eq(*filter_spec: list[list[Any]]*) → CascadedBiquads

Configure this instance as a Parametric Equaliser.

This allows each of the 8 biquads to be individually designed using the designer methods for the biquad. This expects to receive a list of up to 8 biquad design descriptions where a biquad design description is of the form:

```
["type", args...]
```

where "type" is a string defining how the biquad should be designed e.g. "lowpass", and args... is all the parameters to design that type of filter. All options and arguments are listed below:

```
["biquad_allpass", filter_freq, q_factor, Q_sig]
["biquad_bandpass", filter_freq, bw, Q_sig]
["biquad_bandstop", filter_freq, bw, Q_sig]
["biquad_bypass", Q_sig]
["biquad_constant_q", filter_freq, q_factor, boost_db, Q_sig]
["biquad_gain", gain_db, Q_sig]
["biquad_highpass", filter_freq, q_factor, Q_sig]
["biquad_highshelf", filter_freq, q_factor, boost_db, Q_sig]
["biquad_linkwitz", f0, q0, fp, qp, Q_sig]
["biquad_lowpass", filter_freq, q_factor, Q_sig]
["biquad_lowshelf", filter_freq, q_factor, boost_db, Q_sig]
["biquad_notch", filter_freq, q_factor, Q_sig]
["biquad_peaking", filter_freq, q_factor, boost_db, Q_sig]
```

Control

The following runtime control parameters are available for the CascadedBiquads Stage:

Command ID macro	Payload length	Description
CMD_CASCADED_BIQUADS_LEFT_SHIFT	sizeof(int)*[8]	The coefficient shift applied to the output of each biquad in the cascade. The shifts should be in the same format as specified in the individual biquad.
CMD_CASCADED_BIQUADS_FILTER_COEFFS	sizeof(int32_t)*[40]	The normalised biquad filter coefficients for each biquad in the cascade as an array of [5][8], with 5 coefficients for up to 8 biquads. The coefficients should be in the same format as specified in the individual biquad.

1.3 Compressor

Compressor stages allow for control of the dynamic range of the signal, such as reducing the level of loud sounds.

class `audio_dsp.stages.compressor.CompressorRMS(**kwargs)`

A compressor based on the RMS envelope of the input signal.

When the RMS envelope of the signal exceeds the threshold, the signal amplitude is reduced by the compression ratio.

The threshold sets the value above which compression occurs. The ratio sets how much the signal is compressed. A ratio of 1 results in no compression, while a ratio of infinity results in the same behaviour as a limiter. The attack time sets how fast the compressor starts compressing. The release time sets how long the signal takes to ramp up to its original level after the envelope is below the threshold.

Attributes

dsp_block

[`audio_dsp.dsp.drc.drc.compressor_rms`] The DSB block class; see [RMS Compressor](#) for implementation details.

make_compressor_rms(*ratio, threshold_db, attack_t, release_t, Q_sig=27*)

Update compressor configuration based on new parameters.

Parameters

ratio

[float] Compression gain ratio applied when the signal is above the threshold.

threshold_db

[float] Threshold in decibels above which compression occurs.

attack_t

[float] Attack time of the compressor in seconds.

release_t

[float] Release time of the compressor in seconds.

Control

The following runtime control parameters are available for the CompressorRMS Stage:

Command ID macro	Payload length	Description
CMD_COMPRESSOR_RMS_ATTACK_ALPHA	<code>sizeof(int32_t)</code>	The attack alpha in Q0.31 format.
CMD_COMPRESSOR_RMS_RELEASE_ALPHA	<code>sizeof(int32_t)</code>	The release alpha in Q0.31 format.
CMD_COMPRESSOR_RMS_ENVELOPE	<code>sizeof(int32_t)</code>	The current RMS envelope of the signal in Q_SIG format.
CMD_COMPRESSOR_RMS_THRESHOLD	<code>sizeof(int32_t)</code>	The threshold in Q_SIG format above which compression will occur.
CMD_COMPRESSOR_RMS_GAIN	<code>sizeof(int32_t)</code>	The current gain applied by the compressor in Q0.31 format.
CMD_COMPRESSOR_RMS_SLOPE	<code>sizeof(float)</code>	The compression slope of the compressor. This is calculated as $(1 - 1 / \text{ratio}) / 2.0$.

1.4 Compressor Sidechain

Sidechain compressor Stages use the envelope of one input to control the level of a different input.

class `audio_dsp.stages.compressor_sidechain.CompressorSidechain(**kwargs)`

An sidechain compressor based on the RMS envelope of the detect signal.

This stage is limited to accepting 2 channels. The first is the channel that will be compressed. The second is the detect channel. The level of compression depends on the envelope of the second channel.

When the RMS envelope of the detect signal exceeds the threshold, the processed signal amplitude is reduced by the compression ratio.

The threshold sets the value above which compression occurs. The ratio sets how much the signal is compressed. A ratio of 1 results in no compression, while a ratio of infinity results in the same behaviour as a limiter. The attack time sets how fast the compressor starts compressing. The release time sets how long the signal takes to ramp up to its original level after the envelope is below the threshold.

Attributes

dsp_block

[`audio_dsp.dsp.drc.sidechain.compressor_rms_sidechain_mono`] The DSP block class; see [Sidechain RMS Compressor](#) for implementation details.

make_compressor_sidechain(*ratio, threshold_db, attack_t, release_t, Q_sig=27*)

Update compressor configuration based on new parameters.

Parameters

ratio

[float] Compression gain ratio applied when the signal is above the threshold.

threshold_db

[float] Threshold in decibels above which compression occurs.

attack_t

[float] Attack time of the compressor in seconds.

release_t

[float] Release time of the compressor in seconds.

Control

The following runtime control parameters are available for the CompressorSidechain Stage:

Command ID macro	Payload length	Description
<code>CMD_COMPRESSOR_SIDECHAIN_ATTACK_ALPHA</code>	<code>sizeof(int32_t)</code>	The attack alpha in Q0.31 format.
<code>CMD_COMPRESSOR_SIDECHAIN_RELEASE_ALPHA</code>	<code>sizeof(int32_t)</code>	The release alpha in Q0.31 format.
<code>CMD_COMPRESSOR_SIDECHAIN_ENVELOPE</code>	<code>sizeof(int32_t)</code>	The current RMS envelope of the signal in Q_SIG format.
<code>CMD_COMPRESSOR_SIDECHAIN_THRESHOLD</code>	<code>sizeof(int32_t)</code>	The threshold in Q_SIG format above which compression will occur.
<code>CMD_COMPRESSOR_SIDECHAIN_GAIN</code>	<code>sizeof(int32_t)</code>	The current gain applied by the compressor in Q0.31 format.
<code>CMD_COMPRESSOR_SIDECHAIN_SLOPE</code>	<code>sizeof(float)</code>	The compression slope of the compressor. This is calculated as $(1 - 1 / \text{ratio}) / 2.0$.

1.5 Envelope Detector

Envelope detector Stages measure how the average or peak amplitude of a signal varies over time.

1.5.1 EnvelopeDetectorPeak

class `audio_dsp.stages.envelope_detector.EnvelopeDetectorPeak(**kwargs)`

A stage with no outputs that measures the signal peak envelope.

The current envelope of the signal can be read out using this stage's `envelope` control.

Attributes

dsp_block

[`audio_dsp.dsp.drc.drc.envelope_detector_peak`] The DSP block class; see [Peak Envelope Detector](#) for implementation details.

make_env_det_peak(*attack_t*, *release_t*, *Q_sig=27*)

Update envelope detector configuration based on new parameters.

Parameters

attack_t

[float] Attack time of the envelope detector in seconds.

release_t

[float] Release time of the envelope detector in seconds.

Control

The following runtime control parameters are available for the `EnvelopeDetectorPeak` Stage:

Command ID macro	Payload length	Description
<code>CMD_ENVELOPE_DETECTOR_PEAK_ATTACK_ALPHA</code>	<code>sizeof(int32_t)</code>	The attack alpha in Q0.31 format.
<code>CMD_ENVELOPE_DETECTOR_PEAK_RELEASE_ALPHA</code>	<code>sizeof(int32_t)</code>	The release alpha in Q0.31 format.
<code>CMD_ENVELOPE_DETECTOR_PEAK_ENVELOPE</code>	<code>sizeof(int32_t)</code>	The current peak envelope of the signal in Q_SIG format.

1.5.2 EnvelopeDetectorRMS

class `audio_dsp.stages.envelope_detector.EnvelopeDetectorRMS(**kwargs)`

A stage with no outputs that measures the signal RMS envelope.

The current envelope of the signal can be read out using this stage's `envelope` control.

Attributes

dsp_block

[`audio_dsp.dsp.drc.drc.envelope_detector_rms`] The DSP block class; see [RMS Envelope Detector](#) for implementation details.

make_env_det_rms(*attack_t*, *release_t*, *Q_sig*=27)

Update envelope detector configuration based on new parameters.

Parameters

attack_t

[float] Attack time of the envelope detector in seconds.

release_t

[float] Release time of the envelope detector in seconds.

Control

The following runtime control parameters are available for the EnvelopeDetectorRMS Stage:

Command ID macro	Payload length	Description
CMD_ENVELOPE_DETECTOR_RMS_ATTACK_ALPHA	sizeof(int32_t)	The attack alpha in Q0.31 format.
CMD_ENVELOPE_DETECTOR_RMS_RELEASE_ALPHA	sizeof(int32_t)	The release alpha in Q0.31 format.
CMD_ENVELOPE_DETECTOR_RMS_ENVELOPE	sizeof(int32_t)	The current RMS envelope of the signal in Q_SIG format.

1.6 FIR

Finite impulse response (FIR) filter Stages allow the use of arbitrary filters with a finite number of taps.

class `audio_dsp.stages.fir.FirDirect`(*coeffs_path*, ***kwargs*)

A FIR filter implemented in the time domain. The input signal is convolved with the filter coefficients. The filter coefficients can only be set at compile time.

Parameters

coeffs_path

[Path] Path to a file containing the coefficients, in a format supported by [np.loadtxt](#).

Attributes

dsp_block

[`audio_dsp.dsp.fir.fir_direct`] The DSP block class; see [FIR Direct](#) for implementation details.

make_fir_direct(*coeffs_path*, *Q_sig*=27)

Update FIR configuration based on new parameters.

Parameters

coeffs_path

[Path] Path to a file containing the coefficients, in a format supported by [np.loadtxt](#).

Control

The FirDirect Stage has no runtime controllable parameters.

1.7 Limiter

Limiter Stages allow the amplitude of the signal to be restricted based on its envelope.

1.7.1 LimiterRMS

class `audio_dsp.stages.limiter.LimiterRMS(**kwargs)`

A limiter based on the RMS value of the signal. When the RMS envelope of the signal exceeds the threshold, the signal amplitude is reduced.

The threshold sets the value above which limiting occurs. The attack time sets how fast the limiter starts limiting. The release time sets how long the signal takes to ramp up to its original level after the envelope is below the threshold.

Attributes

dsp_block

[`audio_dsp.dsp.drc.drc.limiter_rms`] The DSP block class; see [RMS Limiter](#) for implementation details.

make_limiter_rms(*threshold_db, attack_t, release_t, Q_sig=27*)

Update limiter configuration based on new parameters.

Parameters

threshold_db

[float] Threshold in decibels above which limiting occurs.

attack_t

[float] Attack time of the limiter in seconds.

release_t

[float] Release time of the limiter in seconds.

Control

The following runtime control parameters are available for the LimiterRMS Stage:

Command ID macro	Payload length	Description
<code>CMD_LIMITER_RMS_ATTACK_ALPHA</code>	<code>sizeof(int32_t)</code>	The attack alpha in Q0.31 format.
<code>CMD_LIMITER_RMS_RELEASE_ALPHA</code>	<code>sizeof(int32_t)</code>	The release alpha in Q0.31 format.
<code>CMD_LIMITER_RMS_ENVELOPE</code>	<code>sizeof(int32_t)</code>	The current RMS envelope of the signal in Q_SIG format.
<code>CMD_LIMITER_RMS_THRESHOLD</code>	<code>sizeof(int32_t)</code>	The threshold in Q_SIG format above which limiting will occur.
<code>CMD_LIMITER_RMS_GAIN</code>	<code>sizeof(int32_t)</code>	The current gain applied by the limiter in Q_GAIN format.

1.7.2 LimiterPeak

class `audio_dsp.stages.limiter.LimiterPeak(**kwargs)`

A limiter based on the peak value of the signal. When the peak envelope of the signal exceeds the threshold, the signal amplitude is reduced.

The threshold sets the value above which limiting occurs. The attack time sets how fast the limiter starts limiting. The release time sets how long the signal takes to ramp up to its original level after the envelope is below the threshold.

Attributes

dsp_block

[`audio_dsp.dsp.drc.drc.limiter_peak`] The DSP block class; see [Peak Limiter](#) for implementation details.

make_limiter_peak(*threshold_db, attack_t, release_t, Q_sig=27*)

Update limiter configuration based on new parameters.

Parameters

threshold_db

[float] Threshold in decibels above which limiting occurs.

attack_t

[float] Attack time of the limiter in seconds.

release_t

[float] Release time of the limiter in seconds.

Control

The following runtime control parameters are available for the LimiterPeak Stage:

Command ID macro	Payload length	Description
CMD_LIMITER_PEAK_ATTACK_ALPHA	<code>sizeof(int32_t)</code>	The attack alpha in Q0.31 format.
CMD_LIMITER_PEAK_RELEASE_ALPHA	<code>sizeof(int32_t)</code>	The release alpha in Q0.31 format.
CMD_LIMITER_PEAK_ENVELOPE	<code>sizeof(int32_t)</code>	The current peak envelope of the signal in Q_SIG format.
CMD_LIMITER_PEAK_THRESHOLD	<code>sizeof(int32_t)</code>	The threshold in Q_SIG format above which limiting will occur.
CMD_LIMITER_PEAK_GAIN	<code>sizeof(int32_t)</code>	The current gain applied by the limiter in Q_GAIN format.

1.7.3 HardLimiterPeak

class `audio_dsp.stages.limiter.HardLimiterPeak(**kwargs)`

A limiter based on the peak value of the signal. The peak envelope of the signal may never exceed the threshold.

When the peak envelope of the signal exceeds the threshold, the signal amplitude is reduced. If the signal still exceeds the threshold, it is clipped.

The threshold sets the value above which limiting/clipping occurs. The attack time sets how fast the limiter starts limiting. The release time sets how long the signal takes to ramp up to its original level after the envelope is below the threshold.

Attributes

dsp_block

[audio_dsp.dsp.drc.drc.hard_limiter_peak] The DSP block class; see [Hard Peak Limiter](#) for implementation details.

make_hard_limiter_peak(*threshold_db, attack_t, release_t, Q_sig=27*)

Update limiter configuration based on new parameters.

Parameters

threshold_db

[float] Threshold in decibels above which limiting occurs.

attack_t

[float] Attack time of the limiter in seconds.

release_t

[float] Release time of the limiter in seconds.

Control

The following runtime control parameters are available for the HardLimiterPeak Stage:

Command ID macro	Payload length	Description
CMD_HARD_LIMITER_PEAK_ATTACK_ALPHA	sizeof(int32_t)	The attack alpha in Q0.31 format.
CMD_HARD_LIMITER_PEAK_RELEASE_ALPHA	sizeof(int32_t)	The release alpha in Q0.31 format.
CMD_HARD_LIMITER_PEAK_ENVELOPE	sizeof(int32_t)	The current peak envelope of the signal in Q_SIG format.
CMD_HARD_LIMITER_PEAK_THRESHOLD	sizeof(int32_t)	The threshold in Q_SIG format above which limiting will occur.
CMD_HARD_LIMITER_PEAK_GAIN	sizeof(int32_t)	The current gain applied by the limiter in Q0.31 format.

1.7.4 Clipper

class audio_dsp.stages.limiter.**Clipper**(**kwargs)

A simple clipper that limits the signal to a specified threshold.

If the signal is greater than the threshold level, it is set to the threshold value.

Attributes

dsp_block

[audio_dsp.dsp.drc.drc.clipper] The DSP block class; see [Clipper](#) for implementation details.

make_clipper(*threshold_db, Q_sig=27*)

Update clipper configuration based on new parameters.

Parameters

threshold_db

[float] Threshold in decibels above which clipping occurs.

Control

The following runtime control parameters are available for the Clipper Stage:

Command ID macro	Payload length	Description
CMD_CLIPPER_THRESHOLD	sizeof(int32_t)	The threshold in Q_SIG format above which clipping will occur.

1.8 Noise Gate

Noise gate Stages remove quiet signals from the audio output.

class `audio_dsp.stages.noise_gate.NoiseGate(**kwargs)`

A noise gate that reduces the level of an audio signal when it falls below a threshold.

When the signal envelope falls below the threshold, the gain applied to the signal is reduced to 0 over the release time. When the envelope returns above the threshold, the gain applied to the signal is increased to 1 over the attack time.

The initial state of the noise gate is with the gate open (no attenuation); this models a full scale signal having been present before $t = 0$.

Attributes

dsp_block

[`audio_dsp.dsp.drc.expander.noise_gate`] The DSP block class; see [Noise Gate](#) for implementation details.

make_noise_gate(*threshold_db, attack_t, release_t, Q_sig=27*)

Update noise gate configuration based on new parameters.

Parameters

threshold_db

[float] The threshold level in decibels below which the audio signal is attenuated.

attack_t

[float] Attack time of the noise gate in seconds.

release_t

[float] Release time of the noise gate in seconds.

Control

The following runtime control parameters are available for the NoiseGate Stage:

Command ID macro	Payload length	Description
CMD_NOISE_GATE_ATTACK_ALPHA	sizeof(int32_t)	The attack alpha in Q0.31 format.
CMD_NOISE_GATE_RELEASE_ALPHA	sizeof(int32_t)	The release alpha in Q0.31 format.
CMD_NOISE_GATE_ENVELOPE	sizeof(int32_t)	The current peak envelope of the signal in Q_SIG format.
CMD_NOISE_GATE_THRESHOLD	sizeof(int32_t)	The threshold in Q_SIG format below which gating will occur.
CMD_NOISE_GATE_GAIN	sizeof(int32_t)	The current gain applied by the noise gate in Q0.31 format.

1.9 Noise Suppressor Expander

Noise suppressor and expander Stages control the behaviour of quiet signals, typically by trying to reduce the audibility of noise in the signal.

class `audio_dsp.stages.noise_suppressor_expander.NoiseSuppressorExpander(**kwargs)`

The Noise Suppressor (Expander) stage. A noise suppressor that reduces the level of an audio signal when it falls below a threshold. This is also known as an expander.

When the signal envelope falls below the threshold, the gain applied to the signal is reduced relative to the expansion ratio over the release time. When the envelope returns above the threshold, the gain applied to the signal is increased to 1 over the attack time.

The initial state of the noise suppressor is with the suppression off; this models a full scale signal having been present before $t = 0$.

Attributes

dsp_block

[`audio_dsp.dsp.drc.expander.noise_suppressor_expander`] The DSP block class; see [Noise Suppressor/Expander](#) for implementation details.

make_noise_suppressor_expander(*ratio, threshold_db, attack_t, release_t, Q_sig=27*)

Update noise suppressor (expander) configuration based on new parameters.

All parameters are passed to the constructor of `audio_dsp.drc.noise_suppressor_expander`.

Parameters

ratio

[float] The expansion ratio applied to the signal when the envelope falls below the threshold.

threshold_db

[float] The threshold level in decibels below which the audio signal is attenuated.

attack_t

[float] Attack time of the noise suppressor in seconds.

release_t

[float] Release time of the noise suppressor in seconds.

Control

The following runtime control parameters are available for the NoiseSuppressorExpander Stage:

Command ID macro	Payload length	Description
CMD_NOISE_SUPPRESSOR_EXPANDER_ATTACK_ALPHA	sizeof(int32_t)	The attack alpha in Q0.31 format.
CMD_NOISE_SUPPRESSOR_EXPANDER_RELEASE_ALPHA	sizeof(int32_t)	The release alpha in Q0.31 format.
CMD_NOISE_SUPPRESSOR_EXPANDER_ENVELOPE	sizeof(int32_t)	The current peak envelope of the signal in Q_SIG format.
CMD_NOISE_SUPPRESSOR_EXPANDER_THRESHOLD	sizeof(int32_t)	The threshold in Q_SIG format below which suppression will occur.
CMD_NOISE_SUPPRESSOR_EXPANDER_GAIN	sizeof(int32_t)	The current gain applied by the noise suppressor in Q0.31 format.
CMD_NOISE_SUPPRESSOR_EXPANDER_SLOPE	sizeof(float)	The expansion slope of the noise suppressor. This is calculated as (1 - ratio).

1.10 Reverb

Reverb Stages emulate the natural reverberance of rooms.

class `audio_dsp.stages.reverb.ReverbRoom(max_room_size=1, **kwargs)`

The room reverb stage. This is based on Freeverb by Jezar at Dreampoint, and consists of 8 parallel comb filters fed into 4 series all-pass filters.

Parameters

max_room_size

Sets the maximum room size for this reverb. The `room_size` parameter sets the fraction of this value actually used at any given time. For optimal memory usage, `max_room_size` should be set so that the longest reverb tail occurs when `room_size=1.0`.

Attributes

dsp_block

[`audio_dsp.dsp.reverb.reverb_room`] The DSP block class; see [Reverb Room](#) for implementation details.

set_damping(damping)

Set the damping of the reverb room stage. This controls how much high frequency attenuation is in the room. Higher values yield shorter reverberation times at high frequencies.

Parameters

damping

[float] How much high frequency attenuation in the room, between 0 and 1.

set_decay(decay)

Set the decay of the reverb room stage. This sets how reverberant the room is. Higher values will give a longer reverberation time for a given room size.

Parameters

decay

[float] How long the reverberation of the room is, between 0 and 1.

set_dry_gain(*gain_db*)

Set the dry gain of the reverb room stage. This sets the level of the unprocessed signal.

Parameters

gain_db

[float] Dry gain in dB, less than 0 dB.

set_pre_gain(*pre_gain*)

Set the pre gain of the reverb room stage. It is not advised to increase this value above the default 0.015, as it can result in saturation inside the reverb delay lines.

Parameters

pre_gain

[float] Pre gain value. Must be less than 1 (default 0.015).

set_room_size(*new_room_size*)

Set the room size, will adjust the delay line lengths.

The room size is proportional to *max_room_size*, and must be between 0 and 1. To increase the *room_size* above 1.0, *max_room_size* must instead be increased. Optimal memory usage occurs when *room_size* is set to 1.0.

Parameters

new_room_size

[float] How big the room is as a proportion of *max_room_size*. This sets delay line lengths and must be between 0 and 1.

set_wet_gain(*gain_db*)

Set the wet gain of the reverb room stage. This sets the level of the reverberated signal.

Parameters

gain_db

[float] Wet gain in dB, less than 0 dB.

Control

The following runtime control parameters are available for the ReverbRoom Stage:

Command ID macro	Payload length	Description
CMD_REVERB_ROOM_SAMPLING_FREQ	sizeof(uint32_t)	The sampling frequency of the reverb, can only be set at initialisation.
CMD_REVERB_ROOM_MAX_ROOM_SIZE	sizeof(float)	Sets the maximum size of the delay buffers, can only be set at initialisation.
CMD_REVERB_ROOM_ROOM_SIZE	sizeof(float)	How big the room is as a proportion of max_room_size. This sets delay line lengths and must be between 0 and 1.
CMD_REVERB_ROOM_FEEDBACK	sizeof(int32_t)	feedback gain in Q0.31 format. Feedback can be calculated from decay as $(0.28 \text{ decay}) + 0.7$.
CMD_REVERB_ROOM_DAMPING	sizeof(int32_t)	High frequency attenuation in Q0.31 format.
CMD_REVERB_ROOM_WET_GAIN	sizeof(int32_t)	Gain applied to the wet signal in Q0.31 format
CMD_REVERB_ROOM_DRY_GAIN	sizeof(int32_t)	Dry signal gain in Q0.31 format.
CMD_REVERB_ROOM_PREGAIN	sizeof(int32_t)	The pregain applied to the signal before reverb. Changing this value is not recommended.

1.11 Signal Chain

Signal chain stages allow for the control of signal flow through the pipeline. This includes stages for combining and splitting signals, basic gain components, and delays.

1.11.1 Bypass

class audio_dsp.stages.signal_chain.Bypass(**kwargs)

Stage which does not modify its inputs. Useful if data needs to flow through a thread which is not being processed on to keep pipeline lengths aligned.

process(in_channels)

Return a copy of the inputs.

Control

The Bypass Stage has no runtime controllable parameters.

1.11.2 Fork

class `audio_dsp.stages.signal_chain.Fork(count=2, **kwargs)`

Fork the signal.

Use if the same data needs to be sent to multiple data paths:

```
a = t.stage(Example, ...)
f = t.stage(Fork, a, count=2) # count optional, default is 2
b = t.stage(Example, f.forks[0])
c = t.stage(Example, f.forks[1])
```

Attributes

forks

[list[list[StageOutput]]] For convenience, each forked output will be available in this list each entry contains a set of outputs which will contain the same data as the input.

class `ForkOutputList(edges: Optional[list[audio_dsp.design.stage.StageOutput | None]] = None)`

Custom StageOutputList that is created by Fork.

This allows convenient access to each fork output.

Attributes

forks: list[StageOutputList]

Fork duplicates its inputs, each entry in the forks list is a single copy of the input edges.

get_frequency_response(*nfft=512*)

Fork has no sensible frequency response, not implemented.

process(*in_channels*)

Duplicate the inputs to the outputs based on this fork's configuration.

Control

The Fork Stage has no runtime controllable parameters.

1.11.3 Mixer

class `audio_dsp.stages.signal_chain.Mixer(**kwargs)`

Mixes the input signals together. The mixer can be used to add signals together, or to attenuate the input signals.

Attributes

dsp_block

[`audio_dsp.dsp.signal_chain.mixer`] The DSP block class; see [Mixer](#) for implementation details

set_gain(*gain_db*)

Set the gain of the mixer in dB.

Parameters

gain_db

[float] The gain of the mixer in dB.

Control

The following runtime control parameters are available for the Mixer Stage:

Command ID macro	Payload length	Description
CMD_MIXER_GAIN	sizeof(int32_t)	The current gain in Q_GAIN format.

1.11.4 Adder

class `audio_dsp.stages.signal_chain.Adder(**kwargs)`

Add the input signals together. The adder can be used to add signals together.

Attributes

dsp_block

[`audio_dsp.dsp.signal_chain.adder`] The DSP block class; see [Adder](#) for implementation details.

Control

The Adder Stage has no runtime controllable parameters.

1.11.5 Subtractor

class `audio_dsp.stages.signal_chain.Subtractor(**kwargs)`

Subtract the second input from the first. The subtractor can be used to subtract signals from each other. It must only have 2 inputs.

Attributes

dsp_block

[`audio_dsp.dsp.signal_chain.subtractor`] The DSP block class; see [Subtractor](#) for implementation details.

Control

The Subtractor Stage has no runtime controllable parameters.

1.11.6 FixedGain

class `audio_dsp.stages.signal_chain.FixedGain(gain_db=0, **kwargs)`

This stage implements a fixed gain. The input signal is multiplied by a gain. If the gain is changed at runtime, pops and clicks may occur.

If the gain needs to be changed at runtime, use a `VolumeControl` stage instead.

Parameters

gain_db

[float, optional] The gain of the mixer in dB.

Attributes

dsp_block

[`audio_dsp.dsp.signal_chain.fixed_gain`] The DSP block class; see [Fixed Gain](#) for implementation details.

set_gain(*gain_db*)

Set the gain of the fixed gain in dB.

Parameters

gain_db

[float] The gain of the fixed gain in dB.

Control

The following runtime control parameters are available for the FixedGain Stage:

Command ID macro	Payload length	Description
<code>CMD_FIXED_GAIN_GAIN</code>	<code>sizeof(int32_t)</code>	The gain value in Q_GAIN format.

1.11.7 VolumeControl

class `audio_dsp.stages.signal_chain.VolumeControl(gain_db=0, mute_state=0, **kwargs)`

This stage implements a volume control. The input signal is multiplied by a gain. The gain can be changed at runtime. To avoid pops and clicks during gain changes, a slew is applied to the gain update. The stage can be muted and unmuted at runtime.

Parameters

gain_db

[float, optional] The gain of the mixer in dB.

mute_state

[int, optional] The mute state of the Volume Control: 0: unmuted, 1: muted.

Attributes

dsp_block

[`audio_dsp.dsp.signal_chain.volume_control`] The DSP block class; see [Volume Control](#) for implementation details.

make_volume_control(*gain_db, slew_shift, mute_state, Q_sig=27*)

Update the settings of this volume control.

Parameters

gain_dB

Target gain of this volume control.

slew_shift

The shift value used in the exponential slew.

mute_state

The mute state of the Volume Control: 0: unmuted, 1: muted.

set_gain(*gain_dB*)

Set the gain of the volume control in dB.

Parameters**gain_db**

[float] The gain of the volume control in dB.

set_mute_state(*mute_state*)

Set the mute state of the volume control.

Parameters**mute_state**

[bool] The mute state of the volume control.

Control

The following runtime control parameters are available for the VolumeControl Stage:

Command ID macro	Payload length	Description
CMD_VOLUME_CONTROL_TARGET_GAIN	sizeof(int32_t)	The target gain of the volume control in Q_GAIN format.
CMD_VOLUME_CONTROL_GAIN	sizeof(int32_t)	The current gain of the volume control in Q_GAIN format. This will slew towards the target gain.
CMD_VOLUME_CONTROL_SLEW_SHIFT	sizeof(int32_t)	The shift value used to set the slew rate. See the volume control documentation for conversions between slew_shift and time constant.
CMD_VOLUME_CONTROL_MUTE_STATE	sizeof(uint8_t)	Sets the mute state. 1 is muted and 0 is unmuted.

1.11.8 Switch

class audio_dsp.stages.signal_chain.**Switch**(*index=0, **kwargs*)

Switch the input to one of the outputs. The switch can be used to select between different signals.

move_switch(*position*)

Move the switch to the specified position.

Parameters**position**

[int] The position to which to move the switch. This changes the output signal to the input[position]

Control

The following runtime control parameters are available for the Switch Stage:

Command ID macro	Payload length	Description
CMD_SWITCH_POSITION	sizeof(int32_t)	The current switch position.

1.11.9 Delay

class `audio_dsp.stages.signal_chain.Delay(max_delay, starting_delay, units='samples', **kwargs)`

Delay the input signal by a specified amount.

The maximum delay is set at compile time, and the runtime delay can be set between 0 and `max_delay`.

Parameters

max_delay

[float] The maximum delay in specified units. This can only be set at compile time.

starting_delay

[float] The starting delay in specified units.

units

[str, optional] The units of the delay, can be 'samples', 'ms' or 's'. Default is 'samples'.

Attributes

dsp_block

[`audio_dsp.dsp.signal_chain.delay`] The DSP block class; see [Delay](#) for implementation details.

set_delay(*delay*, *units='samples'*)

Set the length of the delay line, will saturate at `max_delay`.

Parameters

delay

[float] The delay in specified units.

units

[str] The units of the delay, can be 'samples', 'ms' or 's'. Default is 'samples'.

Control

The following runtime control parameters are available for the Delay Stage:

Command ID macro	Payload length	Description
CMD_DELAY_MAX_DELAY	sizeof(uint32_t)	The maximum delay value in samples. This is only configurable at compile time.
CMD_DELAY_DELAY	sizeof(uint32_t)	The current delay value in samples.

2 DSP Modules

In `lib_audio_dsp`, DSP modules are the lower level functions and APIs. These can be used directly without the pipeline building tool. The documentation also includes more implementation details about the DSP algorithms. It includes topics such as Q formats, C and Python APIs, providing more detailed view of the DSP modules.

Each DSP module has been implemented in floating point Python, fixed point int32 Python and fixed point int32 C, with optimisations for xcore. The Python and C fixed point implementations aim to be bit exact with each other, allowing for Python prototyping of DSP pipelines.

2.1 Library Q Format

Note: For fixed point Q formats this document uses the format QM.N, where M is the number of bits before the decimal point (excluding the sign bit), and N is the number of bits after the decimal point. For an int32 number, M+N=31.

By default, the signal processing in the audio pipeline is carried out at 32 bit fixed point precision in Q4.27 format. Assuming a 24 bit input signal in Q0.24 format, this gives 4 bits of internal headroom in the audio pipeline.

Most modules in this library assume that the signal is in a specific global Q format. This format is defined by the `Q_SIG` macro. An additional macro for the signal exponent, `SIG_EXP` is defined, where `SIG_EXP = - Q_SIG`.

Q_SIG

Default Q format

SIG_EXP

Default signal exponent

To ensure optimal headroom and noise floor, the user should ensure that signals are in the correct Q format before processing. Either the input Q format can be converted to `Q_SIG`, or `Q_SIG` can be changed to the desired value.

Note: Not using the DSP pipeline tool means that Q formats will not automatically be managed, and the user should take care to ensure they have the correct values for optimum performance and signal level.

For example, for more precision, the pipeline can be configured to run with no headroom in Q0.31 format, but this would require manual headroom management (e.g. reducing the signal level before a boost to avoid clipping).

To convert between `Q_SIG` and Q0.31 in a safe and optimised way, the APIs below are provided.

`int32_t adsp_from_q31 (int32_t input)`

Convert from Q0.31 to `Q_SIG`.

Parameters

- **input** – Input in Q0.31 format

Returns

int32_t Output in `Q_SIG` format

`int32_t adsp_to_q31(int32_t input)`
Convert from Q_SIG to Q0.31.

Parameters

- **input** – Input in Q_SIG format

Returns

int32_t Output in Q0.31 format

2.2 Biquad Filters

2.2.1 Single Biquad

A second order biquadratic filter, which can be used to implement many common second order filters. The filter had been implemented in the direct form 1, and uses the `xcore.ai` vector unit to calculate the 5 filter taps in a single instruction.

Coefficients are stored in Q1.30 format to benefit from the vector unit, allowing for a filter coefficient range of $[-2, 1.999]$. For some high gain biquads (e.g. high shelf filters), the numerator coefficients may exceed this range. If this is the case, the numerator coefficients only should be right-shifted until they fit within the range (the denominator coefficients cannot become larger than 2.0 without the poles exceeding the unit circle). The shift should be passed into the API, and the output signal from the biquad will then have a left-shift applied. This is equivalent to reducing the overall signal level in the biquad, then returning to unity gain afterwards.

The state should be initialised to 0. The state and coeffs must be word-aligned.

C API

`int32_t adsp_biquad(int32_t new_sample, q2_30 coeffs[5], int32_t state[8], left_shift_t lsh)`

Biquad filter. This function implements a biquad filter. The filter is implemented as a direct form 1.

Parameters

- **new_sample** – New sample to be filtered
- **coeffs** – Filter coefficients
- **state** – Filter state
- **lsh** – Left shift compensation value

Returns

int32_t Filtered sample

Python API

`class audio_dsp.dsp.biquad.biquad(coeffs: list[float], fs: int, n_chans: int = 1, b_shift: int = 0, Q_sig: int = 27)`

A second order biquadratic filter instance.

This implements a direct form 1 biquad filter, using the coefficients provided at initialisation: $a0*y[n] = b0*x[n] + b1*x[n-1] + b2*x[n-2] - a1*y[n-1] - a2*y[n-2]$

For efficiency the biquad coefficients are normalised by $a0$ and the output a coefficients multiplied by -1.

Parameters

coeffs

[list[float]] List of normalised biquad coefficients in the form in the form $[b0, b1, b2, -a1, -a2]/a0$

fs

[int] Sampling frequency in Hz.

n_chans

[int] Number of channels the block runs on.

b_shift

[int] The number of right shift bits applied to the b coefficients. The default coefficient scaling allows for a maximum coefficient value of 2, but high gain shelf and peaking filters can have coefficients above this value. Shifting the b coefficients down allows coefficients greater than 2, with the cost of b_shift bits of precision.

Q_sig: int, optional

Q format of the signal, number of bits after the decimal point. Defaults to Q4.27.

Attributes**fs**

[int] Sampling frequency in Hz.

n_chans

[int] Number of channels the block runs on.

Q_sig: int

Q format of the signal, number of bits after the decimal point.

coeffs

[list[float]] List of normalised float biquad coefficients in the form in the form $[b0, b1, b2, -a1, -a2]/a0$, rounded to int32 precision.

int_coeffs

[list[int]] List of normalised int biquad coefficients in the form in the form $[b0, b1, b2, -a1, -a2]/a0$, scaled and rounded to int32.

process(*sample: float, channel: int = 0*) → float

Filter a single sample using direct form 1 biquad using floating point maths.

Parameters**sample**

[float] The input sample to be processed.

channel

[int, optional] The channel index to process the sample on. Default is 0.

Returns**float**

The processed sample.

update_coeffs(*new_coeffs: list[float]*)

Update the saved coefficients to the input values.

Parameters**new_coeffs**

[list[float]] The new coefficients to be updated.

reset_state()

Reset the biquad saved states to zero.

2.2.2 Cascaded Biquads

The cascaded biquad module is equivalent to 8 individual biquad filters connected in series. It can be used to implement a simple parametric equaliser or high-order Butterworth filters, implemented as cascaded second order sections.

C API

`int32_t adsp_cascaded_biquads_8b(int32_t new_sample, q2_30 coeffs[40], int32_t state[64], left_shift_t lsh[8])`

8-band cascaded biquad filter This function implements an 8-band cascaded biquad filter. The filter is implemented as a direct form 1 filter.

Note: The filter coefficients must be in [5][8]

Parameters

- **new_sample** – New sample to be filtered
- **coeffs** – Filter coefficients
- **state** – Filter state
- **lsh** – Left shift compensation value

Returns

int32_t Filtered sample

Python API

`class audio_dsp.dsp.cascaded_biquads.cascaded_biquads_8(coeffs_list, fs, n_chans, Q_sig=27)`

A class representing a cascaded biquad filter with up to 8 biquads.

This can be used to either implement a parametric equaliser or a higher order filter built out of second order sections.

8 biquad objects are always created, if there are less than 8 biquads in the cascade, the remaining biquads are set to bypass ($b_0 = 1$).

For documentation on individual biquads, see `audio_dsp.dsp.biquad.biquad`.

Parameters

coeffs_list

[list] List of coefficients for each biquad in the cascade.

fs

[int] Sampling frequency in Hz.

n_chans

[int] Number of channels the block runs on.

Q_sig: int, optional

Q format of the signal, number of bits after the decimal point. Defaults to Q4.27.

Attributes

fs

[int] Sampling frequency in Hz.

n_chans
[int] Number of channels the block runs on.

Q_sig: int
Q format of the signal, number of bits after the decimal point.

biquads
[list] List of biquad objects representing each biquad in the cascade.

process(*sample*, *channel=0*)
Process the input sample through the cascaded biquads using floating point maths.

Parameters

sample
[float] The input sample to be processed.

channel
[int] The channel index to process the sample on.

Returns

float
The processed output sample.

reset_state()
Reset the biquad saved states to zero.

2.3 Dynamic Range Control

Dynamic Range Control (DRC) in audio digital signal processing (DSP) refers to the automatic adjustment of an audio signal's amplitude to reduce its dynamic range - the difference between the loudest and quietest parts of the audio. They include compressors, limiters and clippers, as well as the envelope detectors used to detect the signal level.

2.3.1 Attack and Release Times

Nearly all DRC modules feature an attack and release time to control the responsiveness of the module to changes in signal level. Attack and release times converted from seconds to alpha coefficients for use in the exponential moving average calculation. The shorter the attack or release time, the bigger the alpha. Large alpha will result in the envelope becoming more reactive to the input samples. Small alpha values will give more smoothed behaviour. The difference between the input level and the current envelope or gain determines whether the attack or release alpha is used.

2.3.2 Envelope Detectors

Envelope detectors run an exponential moving average (EMA) of the incoming signal. They are used as a part of the most DRC components. They can also be used to implement VU meters and level detectors.

They feature [attack and release times](#) to control the responsiveness of the envelope detector.

The C struct below is used for all the envelope detector implementations.

```
struct env_detector_t  
    Envelope detector state structure.
```

Public Members

q1_31 **attack_alpha**

Attack alpha

q1_31 **release_alpha**

Release alpha

int32_t **envelope**

Current envelope

2.3.2.1 Peak Envelope Detector

A peak-based envelope detector will run its EMA using the absolute value of the input sample.

C API

void **adsp_env_detector_peak**([env_detector_t](#) *env_det, int32_t new_sample)

Update the envelope detector peak with a new sample.

Parameters

- **env_det** – Envelope detector object
- **new_sample** – New sample

Python API

class `audio_dsp.dsp.drc.drc.envelope_detector_peak(fs, n_chans, attack_t, release_t, Q_sig=27)`

Envelope detector that follows the absolute peak value of a signal.

The attack time sets how fast the envelope detector ramps up. The release time sets how fast the envelope detector ramps down.

Parameters

fs

[int] Sampling frequency in Hz.

n_chans

[int] Number of channels the block runs on.

attack_t

[float, optional] Attack time of the envelope detector in seconds.

release_t

[float, optional] Release time of the envelope detector in seconds.

Q_sig: int, optional

Q format of the signal, number of bits after the decimal point. Defaults to Q4.27.

Attributes

fs

[int] Sampling frequency in Hz.

n_chans

[int] Number of channels the block runs on.

Q_sig: int

Q format of the signal, number of bits after the decimal point.

attack_alpha

[float] Attack time parameter used for exponential moving average in floating point processing.

release_alpha

[float] Release time parameter used for exponential moving average in floating point processing.

envelope

[list[float]] Current envelope value for each channel for floating point processing.

attack_alpha_int

[int] attack_alpha in 32-bit int format.

release_alpha_int

[int] release_alpha in 32-bit int format.

envelope_int

[list[int]] current envelope value for each channel in 32-bit int format.

process(*sample*, *channel*=0)

Update the peak envelope for a signal, using floating point maths.

Take one new sample and return the updated envelope. Input should be scaled with 0 dB = 1.0.

Parameters**sample**

[float] The input sample to be processed.

channel

[int, optional] The channel index to process the sample on. Default is 0.

Returns**float**

The processed sample.

reset_state()

Reset the envelope to zero.

2.3.2.2 RMS Envelope Detector

An RMS-based envelope detector will run its EMA using the square of the input sample. It returns the mean² in order to avoid a square root.

C API

void **adsp_env_detector_rms**(*env_detector_t* *env_det, int32_t new_sample)

Update the envelope detector RMS with a new sample.

Parameters

- **env_det** – Envelope detector object
- **new_sample** – New sample

Python API

class `audio_dsp.dsp.drc.drc.envelope_detector_rms(fs, n_chans, attack_t, release_t, Q_sig=27)`

Envelope detector that follows the RMS value of a signal.

Note this returns the mean^2 value, there is no need to do the `sqrt()` as if the output is converted to dB, $10\log_{10}()$ can be taken instead of $20\log_{10}()$.

The attack time sets how fast the envelope detector ramps up. The release time sets how fast the envelope detector ramps down.

Parameters

fs

[int] Sampling frequency in Hz.

n_chans

[int] Number of channels the block runs on.

attack_t

[float, optional] Attack time of the envelope detector in seconds.

release_t

[float, optional] Release time of the envelope detector in seconds.

Q_sig: int, optional

Q format of the signal, number of bits after the decimal point. Defaults to Q4.27.

Attributes

fs

[int] Sampling frequency in Hz.

n_chans

[int] Number of channels the block runs on.

Q_sig: int

Q format of the signal, number of bits after the decimal point.

attack_alpha

[float] Attack time parameter used for exponential moving average in floating point processing.

release_alpha

[float] Release time parameter used for exponential moving average in floating point processing.

envelope

[list[float]] Current envelope value for each channel for floating point processing.

attack_alpha_int

[int] attack_alpha in 32-bit int format.

release_alpha_int

[int] release_alpha in 32-bit int format.

envelope_int

[list[int]] current envelope value for each channel in 32-bit int format.

process(*sample*, *channel*=0)

Update the RMS envelope for a signal, using floating point maths.

Take one new sample and return the updated envelope. Input should be scaled with 0 dB = 1.0.

Note this returns the mean^2 value, there is no need to do the `sqrt()` as if the output is converted to dB, $10\log_{10}()$ can be taken instead of $20\log_{10}()$.

Parameters

sample

[float] The input sample to be processed.

channel

[int, optional] The channel index to process the sample on. Default is 0.

Returns**float**

The processed sample.

reset_state()

Reset the envelope to zero.

2.3.3 Clipper

A clipper limits the signal to a specified threshold. It is applied instantaneously, so has no attack or release times.

typedef int32_t **clipper_t**

Clipper state structure. Should be initialised with the linear threshold.

C API

int32_t **adsp_clipper**([clipper_t](#) clip, int32_t new_samp)

Process a new sample with a clipper.

Parameters

- **clip** – Clipper object
- **new_samp** – New sample

Returns

int32_t Clipped sample

Python API

class audio_dsp.dsp.drc.drc.**clipper**(*fs, n_chans, threshold_db, Q_sig=27*)

A simple clipper that limits the signal to a specified threshold.

Parameters**fs**

[int] Sampling frequency in Hz.

n_chans

[int] Number of channels the block runs on.

threshold_db

[float] Threshold above which clipping occurs in dB.

Q_sig: int, optional

Q format of the signal, number of bits after the decimal point. Defaults to Q4.27.

Attributes**fs**

[int] Sampling frequency in Hz.

n_chans

[int] Number of channels the block runs on.

Q_sig: int

Q format of the signal, number of bits after the decimal point.

threshold

[float] Value above which clipping occurs for floating point processing.

threshold_int

[int] Value above which clipping occurs for int32 fixed point processing.

process(*sample, channel=0*)

Take one new sample and return the clipped sample, using floating point maths. Input should be scaled with 0 dB = 1.0.

Parameters**sample**

[float] The input sample to be processed.

channel

[int, optional] The channel index to process the sample on. Default is 0.

Returns**float**

The processed sample.

2.3.4 Limiters

Limiters will reduce the amplitude of a signal when the signal envelope is greater than the desired threshold. This is similar behaviour to a compressor with an infinite ratio.

A limiter will run an internal envelope detector to get the signal envelope, then compare it to the threshold. If the envelope is greater than the threshold, the applied gain will be reduced. If the envelope is below the threshold, unity gain will be applied. The gain is run through an EMA to avoid abrupt changes. The same [attack and release times](#) are used for the envelope detector and the gain smoothing.

The C struct below is used for all the limiter implementations.

struct **limiter_t**

Limiter state structure.

Public Members

[env_detector_t](#) **env_det**

Envelope detector

int32_t **threshold**

Linear threshold

int32_t **gain**

Linear gain

2.3.4.1 Peak Limiter

A peak limiter uses the [Peak Envelope Detector](#) to get an envelope. When envelope is above the threshold, the new gain is calculated as $\text{threshold} / \text{envelope}$.

C API

`int32_t adsp_limiter_peak(limiter_t *lim, int32_t new_samp)`

Process a new sample with a peak limiter.

Parameters

- **lim** – Limiter object
- **new_samp** – New sample

Returns

int32_t Limited sample

Python API

`class audio_dsp.dsp.drc.drc.limiter_peak(fs, n_chans, threshold_db, attack_t, release_t, Q_sig=27)`

A limiter based on the peak value of the signal. When the peak envelope of the signal exceeds the threshold, the signal amplitude is reduced.

The threshold set the value above which limiting occurs. The attack time sets how fast the limiter starts limiting. The release time sets how long the signal takes to ramp up to its original level after the envelope is below the threshold.

Parameters

fs

[int] Sampling frequency in Hz.

n_chans

[int] number of parallel channels the compressor/limiter runs on. The channels are limited/compressed separately, only the constant parameters are shared.

threshold_db

[float] Threshold in decibels above which limiting occurs.

attack_t

[float] Attack time of the compressor/limiter in seconds.

release_t: float

Release time of the compressor/limiter in seconds.

Q_sig: int, optional

Q format of the signal, number of bits after the decimal point. Defaults to Q4.27.

Attributes

fs

[int] Sampling frequency in Hz.

n_chans

[int] Number of channels the block runs on.

Q_sig: int

Q format of the signal, number of bits after the decimal point.

env_detector

[[envelope_detector_peak](#)] Nested peak envelope detector used to calculate the envelope of the signal.

threshold

[float] Value above which compression/limiting occurs for floating point processing.

gain

[list[float]] Current gain to be applied to the signal for each channel for floating point processing.

attack_alpha

[float] Attack time parameter used for exponential moving average in floating point processing.

release_alpha

[float] Release time parameter used for exponential moving average in floating point processing.

threshold_int

[int] Value above which compression/limiting occurs for int32 fixed point processing.

gain_int

[list[int]] Current gain to be applied to the signal for each channel for int32 fixed point processing.

attack_alpha_int

[int] attack_alpha in 32-bit int format.

release_alpha_int

[int] release_alpha in 32-bit int format.

process(*sample, channel=0*)

Update the envelope for a signal, then calculate and apply the required gain for compression/limiting, using floating point maths.

Take one new sample and return the compressed/limited sample. Input should be scaled with 0 dB = 1.0.

Parameters**sample**

[float] The input sample to be processed.

channel

[int, optional] The channel index to process the sample on. Default is 0.

Returns**float**

The processed sample.

reset_state()

Reset the envelope detector to 0 and the gain to 1.

2.3.4.2 Hard Peak Limiter

A hard peak limiter is similar to a [Peak Limiter](#), but will clip the output if it's still above the threshold after the peak limiter. This can be useful for a final output limiter before truncating any headroom bits.

C API

`int32_t adsp_hard_limiter_peak(limiter_t *lim, int32_t new_samp)`

Process a new sample with a hard limiter peak.

Parameters

- **lim** – Limiter object
- **new_samp** – New sample

Returns

`int32_t` Limited sample

Python API

```
class audio_dsp.dsp.drc.drc.hard_limiter_peak(fs, n_chans, threshold_db, attack_t, release_t,  
                                              Q_sig=27)
```

A limiter based on the peak value of the signal. When the peak envelope of the signal exceeds the threshold, the signal amplitude is reduced. If the signal still exceeds the threshold, it is clipped.

The threshold set the value above which limiting/clipping occurs. The attack time sets how fast the limiter starts limiting. The release time sets how long the signal takes to ramp up to it's original level after the envelope is below the threshold.

Parameters

fs

[int] Sampling frequency in Hz.

n_chans

[int] number of parallel channels the compressor/limiter runs on. The channels are limited/compressed separately, only the constant parameters are shared.

threshold_db

[float] Threshold in decibels above which limiting occurs.

attack_t

[float] Attack time of the compressor/limiter in seconds.

release_t: float

Release time of the compressor/limiter in seconds.

Q_sig: int, optional

Q format of the signal, number of bits after the decimal point. Defaults to Q4.27.

Attributes

fs

[int] Sampling frequency in Hz.

n_chans

[int] Number of channels the block runs on.

Q_sig: int

Q format of the signal, number of bits after the decimal point.

env_detector

[`envelope_detector_peak`] Nested peak envelope detector used to calculate the envelope of the signal.

threshold

[float] Value above which compression/limiting occurs for floating point processing.

gain

[list[float]] Current gain to be applied to the signal for each channel for floating point processing.

attack_alpha

[float] Attack time parameter used for exponential moving average in floating point processing.

release_alpha

[float] Release time parameter used for exponential moving average in floating point processing.

threshold_int

[int] Value above which compression/limiting occurs for int32 fixed point processing.

gain_int

[list[int]] Current gain to be applied to the signal for each channel for int32 fixed point processing.

attack_alpha_int

[int] attack_alpha in 32-bit int format.

release_alpha_int

[int] release_alpha in 32-bit int format.

process(*sample*, *channel*=0)

Update the envelope for a signal, then calculate and apply the required gain for limiting, using floating point maths. If the output signal exceeds the threshold, clip it to the threshold.

Take one new sample and return the limited sample. Input should be scaled with 0 dB = 1.0.

Parameters**sample**

[float] The input sample to be processed.

channel

[int, optional] The channel index to process the sample on. Default is 0.

Returns**float**

The processed sample.

reset_state()

Reset the envelope detector to 0 and the gain to 1.

2.3.4.3 RMS Limiter

A RMS limiter uses the [RMS Envelope Detector](#) to calculate an envelope. When envelope is above the threshold, the new gain is calculated as $\sqrt{\text{threshold} / \text{envelope}}$.

C API

`int32_t adsp_limiter_rms(limiter_t *lim, int32_t new_samp)`

Process a new sample with an RMS limiter.

Parameters

- **lim** – Limiter object
- **new_samp** – New sample

Returns

int32_t Limited sample

Python API

`class audio_dsp.dsp.drc.drc.limiter_rms(fs, n_chans, threshold_db, attack_t, release_t, Q_sig=27)`

A limiter based on the RMS value of the signal. When the RMS envelope of the signal exceeds the threshold, the signal amplitude is reduced.

The threshold set the value above which limiting occurs. The attack time sets how fast the limiter starts limiting. The release time sets how long the signal takes to ramp up to its original level after the envelope is below the threshold.

Parameters

fs

[int] Sampling frequency in Hz.

n_chans

[int] number of parallel channels the compressor/limiter runs on. The channels are limited/compressed separately, only the constant parameters are shared.

threshold_db

[float] Threshold in decibels above which limiting occurs.

attack_t

[float] Attack time of the compressor/limiter in seconds.

release_t: float

Release time of the compressor/limiter in seconds.

Q_sig: int, optional

Q format of the signal, number of bits after the decimal point. Defaults to Q4.27.

Attributes

fs

[int] Sampling frequency in Hz.

n_chans

[int] Number of channels the block runs on.

Q_sig: int

Q format of the signal, number of bits after the decimal point.

env_detector

[[envelope_detector_rms](#)] Nested RMS envelope detector used to calculate the envelope of the signal.

threshold

[float] Value above which limiting occurs for floating point processing. Note the threshold is saved in the power domain, as the RMS envelope detector returns x^2

gain

[list[float]] Current gain to be applied to the signal for each channel for floating point processing.

attack_alpha

[float] Attack time parameter used for exponential moving average in floating point processing.

release_alpha

[float] Release time parameter used for exponential moving average in floating point processing.

threshold_int

[int] Value above which compression/limiting occurs for int32 fixed point processing.

gain_int

[list[int]] Current gain to be applied to the signal for each channel for int32 fixed point processing.

attack_alpha_int

[int] attack_alpha in 32-bit int format.

release_alpha_int

[int] release_alpha in 32-bit int format.

process(*sample*, *channel*=0)

Update the envelope for a signal, then calculate and apply the required gain for compression/limiting, using floating point maths.

Take one new sample and return the compressed/limited sample. Input should be scaled with 0 dB = 1.0.

Parameters**sample**

[float] The input sample to be processed.

channel

[int, optional] The channel index to process the sample on. Default is 0.

Returns**float**

The processed sample.

reset_state()

Reset the envelope detector to 0 and the gain to 1.

2.3.5 Compressors

A compressor will attenuate the signal when the envelope is greater than the threshold. The input/output relationship above the threshold is defined by the compressor *ratio*.

As with a limiter, the compressor runs an internal envelope detector to get the signal envelope, then compares it to the threshold. If the envelope is greater than the threshold, the gain will be proportionally reduced by the *ratio*, such that it is greater than the threshold by a smaller amount. If the envelope is below the threshold, unity gain will be applied. The gain is then run through an EMA to avoid abrupt changes, before being applied.

The *ratio* defines the input/output gradient in the logarithmic domain. For example, a ratio of 2 will reduce the output gain by 0.5 dB for every 1 dB the envelope is over the threshold. A ratio of 1 will apply no compression.

To avoid converting the envelope to the logarithmic domain for the gain calculation, the ratio is converted to the slope as $(1 - 1 / \text{ratio}) / 2$. The gain can then be calculated as an exponential in the linear domain. The C struct below is used for all the compressors implementations.

struct **compressor_t**

Compressor state structure.

Public Members

[*env_detector_t*](#) **env_det**

Envelope detector

int32_t **threshold**

Linear threshold

int32_t **gain**

Linear gain

float **slope**

Slope of the compression curve

2.3.5.1 RMS Compressor

The RMS compressor uses the [*RMS Envelope Detector*](#) to calculate an envelope. When the envelope is above the threshold, the new gain is calculated as $(\text{threshold} / \text{envelope}) ^ \text{slope}$.

C API

int32_t **adsp_compressor_rms**([*compressor_t*](#) *comp, int32_t new_samp)

Process a new sample with an RMS compressor.

Parameters

- **comp** – Compressor object
- **new_samp** – New sample

Returns

int32_t Compressed sample

Python API

```
class audio_dsp.dsp.drc.drc.compressor_rms(fs, n_chans, ratio, threshold_db, attack_t, release_t, Q_sig=27)
```

A compressor based on the RMS value of the signal. When the RMS envelope of the signal exceeds the threshold, the signal amplitude is reduced by the compression ratio.

The threshold sets the value above which compression occurs. The ratio sets how much the signal is compressed. A ratio of 1 results in no compression, while a ratio of infinity results in the same behaviour as a limiter. The attack time sets how fast the compressor starts compressing. The release time sets how long the signal takes to ramp up to it's original level after the envelope is below the threshold.

Parameters

fs

[int] Sampling frequency in Hz.

n_chans

[int] number of parallel channels the compressor/limiter runs on. The channels are limited/compressed separately, only the constant parameters are shared.

ratio

[float] Compression gain ratio applied when the signal is above the threshold

threshold_db

[float] Threshold in decibels above which compression occurs.

attack_t

[float] Attack time of the compressor/limiter in seconds.

release_t: float

Release time of the compressor/limiter in seconds.

Q_sig: int, optional

Q format of the signal, number of bits after the decimal point. Defaults to Q4.27.

Attributes

fs

[int] Sampling frequency in Hz.

n_chans

[int] Number of channels the block runs on.

Q_sig: int

Q format of the signal, number of bits after the decimal point.

env_detector

[envelope_detector_rms] Nested RMS envelope detector used to calculate the envelope of the signal.

threshold

[float] Value above which compression occurs for floating point processing.

gain

[list[float]] Current gain to be applied to the signal for each channel for floating point processing.

attack_alpha

[float] Attack time parameter used for exponential moving average in floating point processing.

release_alpha

[float] Release time parameter used for exponential moving average in floating point processing.

threshold_int

[int] Value above which compression occurs for int32 fixed point processing.

gain_int

[list[int]] Current gain to be applied to the signal for each channel for int32 fixed point processing.

attack_alpha_int

[int] attack_alpha in 32-bit int format.

release_alpha_int

[int] release_alpha in 32-bit int format.

ratio

[float] Compression gain ratio applied when the signal is above the threshold.

slope

[float] The slope factor of the compressor, defined as $slope = (1 - 1/ratio)$.

slope_f32

[float32] The slope factor of the compressor, used for int32 to float32 processing.

process(*sample*, *channel*=0)

Update the envelope for a signal, then calculate and apply the required gain for compression/limiting, using floating point maths.

Take one new sample and return the compressed/limited sample. Input should be scaled with 0 dB = 1.0.

Parameters**sample**

[float] The input sample to be processed.

channel

[int, optional] The channel index to process the sample on. Default is 0.

Returns**float**

The processed sample.

reset_state()

Reset the envelope detector to 0 and the gain to 1.

2.3.5.2 Sidechain RMS Compressor

The sidechain RMS compressor calculates the envelope of one signal and uses it to compress another signal. It takes two signals: *detect* and *input*. The envelope of the *detect* signal is calculated using an internal [RMS Envelope Detector](#). The gain is calculated in the same way as a [RMS Compressor](#), but the gain is then applied to the *input* sample. This can be used to reduce the level of the *input* signal when the *detect* signal gets above the threshold.

C API

int32_t **adsp_compressor_rms_sidechain**([compressor_t](#) *comp, int32_t input_samp, int32_t detect_samp)

Process a new sample with a sidechain RMS compressor.

Parameters

- **comp** – Compressor object
- **input_samp** – Input sample
- **detect_samp** – Sidechain sample

Returns

int32_t Compressed sample

Python API

```
class audio_dsp.dsp.drc.sidechain.compressor_rms_sidechain_mono(fs, ratio, threshold_db, attack_t,  
                                                             release_t, Q_sig=27)
```

A mono sidechain compressor based on the RMS value of the signal. When the RMS envelope of the signal exceeds the threshold, the signal amplitude is reduced by the compression ratio.

The threshold sets the value above which compression occurs. The ratio sets how much the signal is compressed. A ratio of 1 results in no compression, while a ratio of infinity results in the same behaviour as a limiter. The attack time sets how fast the compressor starts compressing. The release time sets how long the signal takes to ramp up to it's original level after the envelope is below the threshold.

Parameters

- fs**
[int] Sampling frequency in Hz.
- ratio**
[float] Compression gain ratio applied when the signal is above the threshold
- threshold_db**
[float] Threshold in decibels above which compression occurs.
- attack_t**
[float] Attack time of the compressor/limiter in seconds.
- release_t: float**
Release time of the compressor/limiter in seconds.
- Q_sig: int, optional**
Q format of the signal, number of bits after the decimal point. Defaults to Q4.27.

Attributes

- fs**
[int] Sampling frequency in Hz.
- n_chans**
[int] Number of channels the block runs on.
- Q_sig: int**
Q format of the signal, number of bits after the decimal point.
- env_detector**
[envelope_detector_rms] Nested RMS envelope detector used to calculate the envelope of the signal.
- threshold**
[float] Value above which compression occurs for floating point processing.
- gain**
[list[float]] Current gain to be applied to the signal for each channel for floating point processing.
- attack_alpha**
[float] Attack time parameter used for exponential moving average in floating point processing.
- release_alpha**
[float] Release time parameter used for exponential moving average in floating point processing.
- threshold_int**
[int] Value above which compression occurs for int32 fixed point processing.

gain_int

[list[int]] Current gain to be applied to the signal for each channel for int32 fixed point processing.

attack_alpha_int

[int] attack_alpha in 32-bit int format.

release_alpha_int

[int] release_alpha in 32-bit int format.

ratio

[float] Compression gain ratio applied when the signal is above the threshold.

slope

[float] The slope factor of the compressor, defined as $slope = (1 - 1/ratio)$.

slope_f32

[float32] The slope factor of the compressor, used for int32 to float32 processing.

process(*input_sample: float, detect_sample: float*)

Update the envelope for the detection signal, then calculate and apply the required gain for compression/limiting, and apply to the input signal using floating point maths.

Take one new sample and return the compressed/limited sample. Input should be scaled with 0 dB = 1.0.

Parameters**input_sample**

[float] The input sample to be compressed.

detect_sample

[float] The sample used by the envelope detector to determine the amount of compression to apply to the input_sample.

Returns**float**

The processed sample.

reset_state()

Reset the envelope detectors to 0 and the gain to 1.

2.3.6 Expanders

An expander attenuates a signal when the envelope is below the threshold. This increases the dynamic range of the signal, and can be used to attenuate quiet signals, such as low level noise.

Like limiters and compressors, an expander will run an internal envelope detector to calculate the envelope and compare it to the threshold. If the envelope is below the threshold, the applied gain will be reduced. If the envelope is greater than the threshold, unity gain will be applied. The gain is run through an EMA to avoid abrupt changes. The same [attack and release times](#) are used for the envelope detector and the gain smoothing. In an expander, the attack time is defined as the speed at which the gain returns to unity after the signal has been below the threshold.

2.3.6.1 Noise Gate

A noise gate uses the [Peak Envelope Detector](#) to calculate the envelope of the input signal. When the envelope is below the threshold, a gain of 0 is applied to the input signal. Otherwise, unity gain is applied.

```
typedef limiter\_t noise_gate_t  
    Noise gate state structure.
```

C API

```
int32_t adsp_noise_gate(noise\_gate\_t *ng, int32_t new_samp)  
    Process a new sample with a noise gate.
```

Parameters

- **ng** – Noise gate object
- **new_samp** – New sample

Returns

int32_t Gated sample

Python API

```
class audio_dsp.dsp.drc.expander.noise_gate(fs, n_chans, threshold_db, attack_t, release_t, Q_sig=27)
```

A noise gate that reduces the level of an audio signal when it falls below a threshold.

When the signal envelope falls below the threshold, the gain applied to the signal is reduced to 0 (based on the release time). When the envelope returns above the threshold, the gain applied to the signal is increased to 1 over the attack time.

The initial state of the noise gate is with the gate open (no attenuation), assuming a full scale signal has been present before $t = 0$.

Parameters

fs

[int] Sampling frequency in Hz.

n_chans

[int] number of parallel channels the expander runs on. The channels are expanded separately, only the constant parameters are shared.

threshold_db

[float] The threshold level in decibels below which the audio signal is attenuated.

attack_t

[float] Attack time of the expander in seconds.

release_t

[float] Release time of the expander in seconds.

Q_sig: int, optional

Q format of the signal, number of bits after the decimal point. Defaults to Q4.27.

Attributes

fs

[int] Sampling frequency in Hz.

n_chans

[int] Number of channels the block runs on.

Q_sig: int

Q format of the signal, number of bits after the decimal point.

env_detector

[envelope_detector_peak] An instance of the envelope_detector_peak class used for envelope detection.

threshold

[float] The threshold below which the signal is gated.

gain

[list[float]] Current gain to be applied to the signal for each channel for floating point processing.

attack_alpha

[float] Attack time parameter used for exponential moving average in floating point processing.

release_alpha

[float] Release time parameter used for exponential moving average in floating point processing.

threshold_int

[int] The threshold level as a 32-bit signed integer.

gain_int

[list[int]] Current gain to be applied to the signal for each channel for int32 fixed point processing.

attack_alpha_int

[int] attack_alpha in 32-bit int format.

release_alpha_int

[int] release_alpha in 32-bit int format.

gain_calc

[function] function pointer to floating point gain calculation function.

gain_calc_int

[function] function pointer to fixed point gain calculation function.

process(sample, channel=0)

Update the envelope for a signal, then calculate and apply the required gain for expanding, using floating point maths.

Take one new sample and return the expanded sample. Input should be scaled with 0 dB = 1.0.

Parameters**sample**

[float] The input sample to be processed.

channel

[int, optional] The channel index to process the sample on. Default is 0.

Returns**float**

The processed sample.

reset_state()

Reset the envelope detector to 1 and the gain to 1, so the gate starts off.

2.3.6.2 Noise Suppressor/Expander

A basic expander can also be used as a noise suppressor. It uses the [Peak Envelope Detector](#) to calculate the envelope of the input signal. When the envelope is below the threshold, the gain of the signal is reduced according to the ratio. Otherwise, unity gain is applied.

Like a compressor, the `ratio` defines the input/output gradient in the logarithmic domain. For example, a ratio of 2 will reduce the output gain by 0.5 dB for every 1 dB the envelope is below the threshold. A ratio of 1 will apply no gain changes. To avoid converting the envelope to the logarithmic domain for the gain calculation, the ratio is converted to the `slope` as $(1 - \text{ratio})$. The gain can then be calculated as an exponential in the linear domain.

For speed, some parameters such as `inv_threshold` are computed at initialisation to simplify run-time computation.

struct **noise_suppressor_expander_t**

Public Members

[env_detector_t](#) **env_det**

Envelope detector

int32_t **threshold**

Linear threshold

int64_t **inv_threshold**

Inverse threshold

int32_t **gain**

Linear gain

float **slope**

Slope of the noise suppression curve

C API

int32_t **adsp_noise_suppressor_expander**([noise_suppressor_expander_t](#) *nse, int32_t new_samp)

Process a new sample with a noise suppressor (expander)

Parameters

- **nse** – Noise suppressor (Expander) object
- **new_samp** – New sample

Returns

int32_t Suppressed sample

Python API

class `audio_dsp.dsp.drc.expander.noise_suppressor_expander` (*fs, n_chans, ratio, threshold_db, attack_t, release_t, Q_sig=27*)

A noise suppressor that reduces the level of an audio signal when it falls below a threshold. This is also known as an expander.

When the signal envelope falls below the threshold, the gain applied to the signal is reduced relative to the expansion ratio over the release time. When the envelope returns above the threshold, the gain applied to the signal is increased to 1 over the attack time.

The initial state of the noise suppressor is with the suppression off, assuming a full scale signal has been present before $t = 0$.

Parameters

fs

[int] Sampling frequency in Hz.

n_chans

[int] number of parallel channels the expander runs on. The channels are expanded separately, only the constant parameters are shared.

ratio

[float] The expansion ratio applied to the signal when the envelope falls below the threshold.

threshold_db

[float] The threshold level in decibels below which the audio signal is attenuated.

attack_t

[float] Attack time of the expander in seconds.

release_t

[float] Release time of the expander in seconds.

Q_sig: int, optional

Q format of the signal, number of bits after the decimal point. Defaults to Q4.27.

Attributes

fs

[int] Sampling frequency in Hz.

n_chans

[int] Number of channels the block runs on.

Q_sig: int

Q format of the signal, number of bits after the decimal point.

env_detector

[envelope_detector_peak] An instance of the `envelope_detector_peak` class used for envelope detection.

threshold

[float] The threshold below which the signal is gated.

gain

[list[float]] Current gain to be applied to the signal for each channel for floating point processing.

attack_alpha

[float] Attack time parameter used for exponential moving average in floating point processing.

release_alpha

[float] Release time parameter used for exponential moving average in floating point processing.

threshold_int

[int] The threshold level as a 32-bit signed integer.

gain_int

[list[int]] Current gain to be applied to the signal for each channel for int32 fixed point processing.

attack_alpha_int

[int] attack_alpha in 32-bit int format.

release_alpha_int

[int] release_alpha in 32-bit int format.

gain_calc

[function] function pointer to floating point gain calculation function.

gain_calc_int

[function] function pointer to fixed point gain calculation function.

process(*sample*, *channel=0*)

Update the envelope for a signal, then calculate and apply the required gain for expanding, using floating point maths.

Take one new sample and return the expanded sample. Input should be scaled with 0 dB = 1.0.

Parameters**sample**

[float] The input sample to be processed.

channel

[int, optional] The channel index to process the sample on. Default is 0.

Returns**float**

The processed sample.

reset_state()

Reset the envelope detector to 1 and the gain to 1, so the gate starts off.

2.4 Finite Impulse Response Filters

Finite impulse response (FIR) filters allow the use of arbitrary filters with a finite number of taps. This library does not provide FIR filter design tools, but allows for coefficients to be imported from other design tools, such as [SciPy](#).

2.4.1 FIR Direct

The direct FIR implements the filter as a convolution in the time domain. This library uses FIR `filter_fir_s32` implementation from `lib_xcore_math` to run on xcore. More information on implementation can be found in `lib_xcore_math` [documentation](#).

class `audio_dsp.dsp.fir.fir_direct`(*fs: float, n_chans: int, coeffs_path: Path, Q_sig: int = 27*)

An FIR filter, implemented in direct form in the time domain.

When the filter coefficients are converted to fixed point, if there will be leading zeros, a left shift is applied to the coefficients in order to use the full dynamic range of the VPU. A subsequent right shift is applied to the accumulator after the convolution to return to the same gain.

Parameters

fs

[int] Sampling frequency in Hz.

n_chans

[int] Number of channels the block runs on.

coeffs_path

[Path] Path to a file containing the coefficients, in a format supported by [np.loadtxt](#).

Q_sig: int, optional

Q format of the signal, number of bits after the decimal point. Defaults to Q4.27.

Attributes

fs

[int] Sampling frequency in Hz.

n_chans

[int] Number of channels the block runs on.

Q_sig: int

Q format of the signal, number of bits after the decimal point.

coeffs

[np.ndarray] Array of the FIR coefficients in floating point format.

coeffs_int

[list] Array of the FIR coefficients in fixed point int32 format.

shift

[int] Right shift to be applied to the fixed point convolution result. This compensates for any left shift applied to the coefficients.

n_taps

[int] Number of taps in the filter.

buffer

[np.ndarray] Buffer of previous inputs for the convolution in floating point format.

buffer_int

[list] Buffer of previous inputs for the convolution in fixed point format.

buffer_idx

[list] List of the floating point buffer head for each channel.

buffer_idx_int

[list] List of the fixed point point buffer head for each channel.

process(*sample: float, channel: int = 0*) → float

Update the buffer with the current sample and convolve with the filter coefficients, using floating point math.

Parameters**sample**

[float] The input sample to be processed.

channel

[int] The channel index to process the sample on.

Returns**float**

The processed output sample.

reset_state() → None

Reset all the delay line values to zero.

check_coeff_scaling()

Check the coefficient scaling is optimal.

If there will be leading zeros, calculate a shift to use the full dynamic range of the VPU

2.5 Reverb

2.5.1 Reverb Room

The room reverb module imitates the reflections of a room. The algorithm is a Schroeder style reverberation, based on [Freeverb by Jezar at Dreampoint](#). It consists of 8 parallel comb filters fed into 4 series all-pass filters, with a wet and dry microphone control to set the effect level.

For more details on the algorithm, see [Physical Audio Signal Processing](#) by Julius Smith.

struct **reverb_room_t**

A room reverb filter structure.

Public Members

uint32_t **total_buffer_length**

Total buffer length

float **room_size**

Room size

int32_t **wet_gain**

Wet linear gain

int32_t **dry_gain**

Dry linear gain

int32_t **pre_gain**

Linear pre-gain

comb_fv_t **combs**[ADSP_RVR_N_COMBS]

Comb filters

allpass_fv_t **allpasses**[ADSP_RVR_N_APS]

Allpass filters

C API

int32_t **adsp_reverb_room**([reverb_room_t](#) *rv, int32_t new_samp)

Process a sample through a reverb room object.

Parameters

- **rv** – Reverb room object
- **new_samp** – New sample to process

Returns

int32_t Processed sample

Python API

```
class audio_dsp.dsp.reverb.reverb_room(fs, n_chans, max_room_size=1, room_size=1, decay=0.5,  
                                         damping=0.4, wet_gain_db=-1, dry_gain_db=-1, pregain=0.015,  
                                         Q_sig=27)
```

Generate a room reverb effect. This is based on Freeverb by Jezar at Dreampoint, and consists of 8 parallel comb filters fed into 4 series all-pass filters.

Parameters

fs

[int] Sampling frequency in Hz.

n_chans

[int] Number of channels the block runs on.

max_room_size

[float, optional] sets the maximum size of the delay buffers, can only be set at initialisation.

room_size

[float, optional] how big the room is as a proportion of max_room_size. This sets delay line lengths and must be between 0 and 1.

decay

[int, optional] how long the reverberation of the room is, between 0 and 1

damping

[float, optional] how much high frequency attenuation in the room, between 0 and 1

wet_gain_db

[int, optional] wet signal gain, less than 0 dB.

dry_gain_db

[int, optional] dry signal gain, less than 0 dB.

pregain

[float, optional] the amount of gain applied to the signal before being passed into the reverb, less than 1.

Q_sig: int, optional

Q format of the signal, number of bits after the decimal point. Defaults to Q4.27.

Attributes**fs**

[int] Sampling frequency in Hz.

n_chans

[int] Number of channels the block runs on.

Q_sig: int

Q format of the signal, number of bits after the decimal point.

pregain

[float] The pregain applied before the reverb as a floating point number.

pregain_int

[int] The pregain applied before the reverb as a fixed point number.

wet

[float] The linear gain applied to the wet signal as a floating point number.

wet_int

[int] The linear gain applied to the wet signal as a fixed point number.

dry

[float] The linear gain applied to the dry signal as a floating point number.

dry_int

[int] The linear gain applied to the dry signal as a fixed point number.

comb_lengths

[np.ndarray] An array of the comb filter delay line lengths, scaled by max_room_size.

ap_length

[np.ndarray] An array of the all pass filter delay line lengths, scaled by max_room_size.

combs

[list] A list of comb_fv objects containing the comb filters for the reverb.

allpasses

[list] A list of allpass_fv objects containing the all pass filters for the reverb.

room_size

[float] The room size as a proportion of the max_room_size.

process(*sample*, *channel=0*)

Add reverberation to a signal, using floating point maths.

Take one new sample and return the sample with reverb. Input should be scaled with 0 dB = 1.0.

Parameters**sample**

[float] The input sample to be processed.

channel

[int, optional] The channel index to process the sample on. Default is 0.

Returns

float

The processed sample.

reset_state()

Reset all the delay line values to zero.

set_pre_gain(*pre_gain*)

Set the pre gain.

Parameters

pre_gain

[float] pre gain value, less than 1.

set_wet_gain(*wet_gain_db*)

Set the wet gain.

Parameters

wet_gain_db

[float] Wet gain in dB, less than 0 dB.

set_dry_gain(*dry_gain_db*)

Set the dry gain.

Parameters

dry_gain_db

[float] Dry gain in dB, less than 0 dB.

set_decay(*decay*)

Set the decay of the reverb.

Parameters

decay

[float] How long the reverberation of the room is, between 0 and 1.

set_damping(*damping*)

Set the damping of the reverb.

Parameters

damping

[float] How much high frequency attenuation in the room, between 0 and 1.

set_room_size(*room_size*)

Set the current room size; will adjust the delay line lengths accordingly.

Parameters

room_size

[float] How big the room is as a proportion of max_room_size. This sets delay line lengths and must be between 0 and 1.

2.6 Signal Chain Components

Signal chain components includes DSP modules for: * combining signals, such as subtracting, adding, and mixing * forks for splitting signals * basic gain components, such as fixed gain, volume control, and mute * basic delay buffers.

2.6.1 Adder

The adder will add samples from N inputs together. It will round and saturate the result to the Q0.31 range.

C API

`int32_t adsp_adder(int32_t *input, unsigned n_ch)`
Saturating addition of an array of samples.

Note: Will work for any q format

Parameters

- **input** – Array of samples
- **n_ch** – Number of channels

Returns

int32_t Sum of samples

Python API

`class audio_dsp.dsp.signal_chain.adder(fs: float, n_chans: int, Q_sig: int = 27)`

A class representing an adder in a signal chain.

This class inherits from the *mixer* class and provides an adder with no attenuation.

Parameters

- fs**
[int] Sampling frequency in Hz.
- n_chans**
[int] Number of channels the block runs on.
- Q_sig: int, optional**
Q format of the signal, number of bits after the decimal point. Defaults to Q4.27.

Attributes

- fs**
[int] Sampling frequency in Hz.
- n_chans**
[int] Number of channels the block runs on.
- Q_sig: int**
Q format of the signal, number of bits after the decimal point.
- gain_db**
[float] Gain in decibels.

gain

[float] Gain as a linear value.

gain_int

[int] Gain as an integer value.

process_channels(*sample_list: list[float]*) → float

Process a single sample. Apply the gain to all the input samples then sum them using floating point maths.

Parameters**sample_list**

[list] List of input samples

Returns**float**

Output sample.

2.6.2 Subtractor

The subtractor will subtract one sample from another, then round and saturate the difference to Q0.31 range.

C API

int32_t **adsp_subtractor**(int32_t x, int32_t y)

Saturating subtraction of two samples, this returns $x - y$.

Note: Will work for any q format

Parameters

- **x** – Minuend
- **y** – Subtrahend

Returns

int32_t Difference

Python API

class audio_dsp.dsp.signal_chain.**subtractor**(*fs: float, Q_sig: int = 27*)

Subtractor class for subtracting two signals.

Parameters**fs**

[int] Sampling frequency in Hz.

Q_sig: int, optional

Q format of the signal, number of bits after the decimal point. Defaults to Q4.27.

Attributes**fs**

[int] Sampling frequency in Hz.

n_chans

[int] Number of channels the block runs on.

Q_sig: int

Q format of the signal, number of bits after the decimal point.

process_channels(*sample_list: list[float]*) → float

Subtract the second input sample from the first using floating point maths.

Parameters

sample_list

[list[float]] List of input samples.

Returns

float

Result of the subtraction.

2.6.3 Fixed Gain

This module applies a fixed gain to a sample, with rounding and saturation to Q0.31 range. The gain must be in Q_GAIN format.

Q_GAIN

Gain format to be used in the gain APIs

C API

int32_t **adsp_fixed_gain**(int32_t input, int32_t gain)

Fixed-point gain.

Note: One of the inputs has to be in Q_GAIN format

Parameters

- **input** – Input sample
- **gain** – Gain

Returns

int32_t Output sample

Python API

class audio_dsp.dsp.signal_chain.**fixed_gain**(*fs: float, n_chans: int, gain_db: float, Q_sig: int = 27*)

Multiply every sample by a fixed gain value.

In the current implementation, the maximum boost is +24 dB.

Parameters

fs

[int] Sampling frequency in Hz.

n_chans

[int] Number of channels the block runs on.

gain_db

[float] The gain in decibels. Maximum fixed gain is +24 dB.

Q_sig: int, optional

Q format of the signal, number of bits after the decimal point. Defaults to Q4.27.

Attributes**fs**

[int] Sampling frequency in Hz.

n_chans

[int] Number of channels the block runs on.

Q_sig: int

Q format of the signal, number of bits after the decimal point.

process(*sample: float, channel: int = 0*) → float

Multiply the input sample by the gain, using floating point maths.

Parameters**sample**

[float] The input sample to be processed.

channel

[int] The channel index to process the sample on, not used by this module.

Returns**float**

The processed output sample.

2.6.4 Mixer

The mixer applies a gain to all N channels of input samples and adds them together. The sum is rounded and saturated to Q0.31 range. The gain must be in Q_GAIN format.

C API

int32_t **adsp_mixer**(int32_t *input, unsigned n_ch, int32_t gain)

Mixer. Will add signals with gain applied to each signal before mixing.

Note: Inputs or gain have to be in Q_GAIN format

Parameters

- **input** – Array of samples
- **n_ch** – Number of channels
- **gain** – Gain

Returns

int32_t Mixed sample

An alternative way to implement a mixer is to multiply-accumulate the input samples into a 64-bit word, then saturate it to a 32-bit word using:

`int32_t adsp_saturate_32b(int64_t acc)`

Saturating 64-bit accumulator. Will saturate to 32-bit, so that the output value is in the range of `int32_t`.

Parameters

- **acc** – Accumulator

Returns

`int32_t` Saturated value

Python API

class `audio_dsp.dsp.signal_chain.mixer`(*fs: float, n_chans: int, gain_db: float = -6, Q_sig: int = 27*)

Mixer class for adding signals with attenuation to maintain headroom.

Parameters

fs

[int] Sampling frequency in Hz.

n_chans

[int] Number of channels the block runs on.

gain_db

[float] Gain in decibels (default is -6 dB).

Q_sig: int, optional

Q format of the signal, number of bits after the decimal point. Defaults to Q4.27.

Attributes

fs

[int] Sampling frequency in Hz.

n_chans

[int] Number of channels the block runs on.

Q_sig: int

Q format of the signal, number of bits after the decimal point.

gain_db

[float] Gain in decibels.

gain

[float] Gain as a linear value.

gain_int

[int] Gain as an integer value.

process_channels(*sample_list: list[float]*) → float

Process a single sample. Apply the gain to all the input samples then sum them using floating point maths.

Parameters

sample_list

[list] List of input samples

Returns

float

Output sample.

2.6.5 Volume Control

The volume control allows safe real-time gain adjustments with minimal artifacts. When the target gain is changed, a slew is used to move from the current gain to the target gain. This allows smooth gain change and no clicks in the output signal.

The mute API allows the user to safely mute the signal by setting the target gain to 0, with the slew ensuring no pops or clicks. Unmuting will restore the pre-mute target gain. The new gain can be set while muted, but will not take effect until unmute is called. There are separate APIs for process, setting the gain, muting and unmute so that volume control can easily be implemented into the control system.

The slew is applied as an exponential of the difference between the target and current gain. For run-time efficiency, instead of an EMA-style alpha, the difference is right shifted by the `slew_shift` parameter. The relation between `slew_shift` and time is further discussed in the Python class documentation.

struct **volume_control_t**

Volume control state structure.

Public Members

int32_t **target_gain**

Target linear gain

int32_t **gain**

Current linear gain

int32_t **slew_shift**

Slew shift

int32_t **saved_gain**

Saved linear gain

uint8_t **mute_state**

Mute state: 0: unmuted, 1 muted

C API

int32_t **adsp_volume_control**([volume_control_t](#) *vol_ctl, int32_t samp)

Process a new sample with a volume control.

Parameters

- **vol_ctl** – Volume control object
- **samp** – New sample

Returns

int32_t Processed sample

void **adsp_volume_control_set_gain**([volume_control_t](#) *vol_ctl, int32_t new_gain)

Set the target gain of a volume control.

Parameters

- **vol_ctl** – Volume control object

- **new_gain** – New target linear gain

void **adsp_volume_control_mute**(*volume_control_t* *vol_ctl)

Mute a volume control. Will save the current target gain and set the target gain to 0.

Parameters

- **vol_ctl** – Volume control object

void **adsp_volume_control_unmute**(*volume_control_t* *vol_ctl)

Unmute a volume control. Will restore the saved target gain.

Parameters

- **vol_ctl** – Volume control object

Python API

class `audio_dsp.dsp.signal_chain.volume_control`(*fs: float, n_chans: int, gain_db: float = -6, slew_shift: int = 7, mute_state: int = 0, Q_sig: int = 27*)

A volume control class that allows setting the gain in decibels. When the gain is updated, an exponential slew is applied to reduce artifacts.

The slew is implemented as a shift operation. The slew rate can be converted to a time constant using the formula: $time_constant = -1/\ln(1 - 2^{-slew_shift}) * (1/fs)$

A table of the first 10 slew shifts is shown below:

slew_shift	Time constant (ms)
1	0.03
2	0.07
3	0.16
4	0.32
5	0.66
6	1.32
7	2.66
8	5.32
9	10.66
10	21.32

Parameters

fs

[int] Sampling frequency in Hz.

n_chans

[int] Number of channels the block runs on.

gain_db

[float, optional] The initial gain in decibels

slew_shift

[int, optional] The shift value used in the exponential slew.

mute_state

[int, optional] The mute state of the Volume Control: 0: unmuted, 1: muted.

Q_sig: int, optional

Q format of the signal, number of bits after the decimal point. Defaults to Q4.27.

Raises

ValueError

If the `gain_db` parameter is greater than 24 dB.

Attributes**fs**

[int] Sampling frequency in Hz.

n_chans

[int] Number of channels the block runs on.

Q_sig: int

Q format of the signal, number of bits after the decimal point.

target_gain_db

[float] The target gain in decibels.

target_gain

[float] The target gain as a linear value.

target_gain_int

[int] The target gain as a fixed-point integer value.

gain_db

[float] The current gain in decibels.

gain

[float] The current gain as a linear value.

gain_int

[int] The current gain as a fixed-point integer value.

slew_shift

[int] The shift value used in the exponential slew.

mute_state

[int] The mute state of the Volume Control: 0: unmuted, 1: muted

process(*sample: float, channel: int = 0*) → float

Update the current gain, then multiply the input sample by it, using floating point maths.

Parameters**sample**

[float] The input sample to be processed.

channel

[int, optional] The channel index to process the sample on. Not used by this module.

Returns**float**

The processed output sample.

set_gain(*gain_db: float*) → None

Set the gain of the volume control.

Parameters**gain_db**

[float] The gain in decibels. Must be less than or equal to 24 dB.

Raises**ValueError**

If the `gain_db` parameter is greater than 24 dB.

mute() → None

Mute the volume control.

unmute() → None

Unmute the volume control.

2.6.6 Delay

The delay module uses a memory buffer to return a sample after a specified time period. The returned samples will be delayed by a specified value. The `max_delay` is set at initialisation, and sets the amount of memory used by the buffers. It cannot be changed at runtime. The current `delay` value can be changed at runtime within the range `[0, max_delay]`

struct **delay_t**

Delay state structure.

Public Members

float **fs**

Sampling frequency

uint32_t **delay**

Current delay in samples

uint32_t **max_delay**

Maximum delay in samples

uint32_t **buffer_idx**

Current buffer index

int32_t ***buffer**

Buffer

C API

int32_t **adsp_delay**([delay_t](#) *delay, int32_t samp)

Process a new sample through a delay object.

Parameters

- **delay** – Delay object
- **samp** – New sample

Returns

int32_t Oldest sample

Python API

```
class audio_dsp.dsp.signal_chain.delay(fs, n_chans, max_delay: float, starting_delay: float, units: str = 'samples')
```

A simple delay line class.

Parameters

fs

[int] Sampling frequency in Hz.

n_chans

[int] Number of channels the block runs on.

max_delay

[float] The maximum delay in specified units.

starting_delay

[float] The starting delay in specified units.

units

[str, optional] The units of the delay, can be 'samples', 'ms' or 's'. Default is 'samples'.

Attributes

fs

[int] Sampling frequency in Hz.

n_chans

[int] Number of channels the block runs on.

Q_sig: int

Q format of the signal, number of bits after the decimal point.

max_delay

[int] The maximum delay in samples.

delay

[int] The current delay in samples.

buffer

[np.ndarray] The delay line buffer.

buffer_idx

[int] The current index of the buffer.

process_channels(*sample: list[float]*) → list[float]

Put the new sample in the buffer and return the oldest sample.

Parameters

sample

[list] List of input samples

Returns

float

List of delayed samples.

reset_state() → None

Reset all the delay line values to zero.

set_delay(*delay: float, units: str = 'samples'*) → None

Set the length of the delay line, will saturate at max_delay.

Parameters

delay

[float] The delay length in specified units.

units

[str, optional] The units of the delay, can be 'samples', 'ms' or 's'. Default is 'samples'.

2.7 Python module base class

All the Python DSP modules are based on a common base class. In order to keep the documentation short, all Python classes in the previous sections only had the `process` method described, and control methods where necessary. This section provides the user with a more in-depth information of the Python API, which may be useful when adding custom DSP modules.

Some classes overload the base class APIs where they require different input data types or dimensions. However, they will all have the attributes and methods described below.

The process methods can be split into 2 groups:

- 1) `process` is a 64b floating point implementation
- 2) `process_xcore` is a 32b fixed-point implementation, with the aim of being bit exact with the C/assembly implementation.

The `process_xcore` methods can be used to simulate the xcore implementation precision and the noise floor. The Python `process_xcore` implementations have very similar accuracy to the xcore C `adsp_*` implementations (subject to the module and implementation). Python simulation methods tend to be slower as Python has a limited support for the fixed point processing. Bit exactness is not always possible for modules that use 32b float operations, as the rounding of these can differ between C libraries.

class `audio_dsp.dsp.generic.dsp_block(fs, n_chans, Q_sig=27)`

Generic DSP block, all blocks should inherit from this class and implement it's methods.

By using the metaclass `NumpyDocstringInheritanceInitMeta`, parameter and attribute documentation can be inherited by the child classes.

Parameters

fs

[int] Sampling frequency in Hz.

n_chans

[int] Number of channels the block runs on.

Q_sig: int, optional

Q format of the signal, number of bits after the decimal point. Defaults to Q4.27.

Attributes

fs

[int] Sampling frequency in Hz.

n_chans

[int] Number of channels the block runs on.

Q_sig: int

Q format of the signal, number of bits after the decimal point.

freq_response(nfft=512)

Calculate the frequency response of the module for a nominal input.

The generic module has a flat frequency response.

Parameters

nfft

[int, optional] The number of points to use for the FFT, by default 512

Returns

tuple

A tuple containing the frequency values and the corresponding complex response.

process(*sample: float, channel=0*)

Take one new sample and give it back. Do no processing for the generic block.

Parameters**sample**

[float] The input sample to be processed.

channel

[int, optional] The channel index to process the sample on. Default is 0.

Returns**float**

The processed sample.

process_frame(*frame: list*)

Take a list frames of samples and return the processed frames.

A frame is defined as a list of 1-D numpy arrays, where the number of arrays is equal to the number of channels, and the length of the arrays is equal to the frame size.

For the generic implementation, just call process for each sample for each channel.

Parameters**frame**

[list] List of frames, where each frame is a 1-D numpy array.

Returns**list**

List of processed frames, with the same structure as the input frame.

process_frame_xcore(*frame: list*)

Take a list frames of samples and return the processed frames, using an xcore-like implementation.

A frame is defined as a list of 1-D numpy arrays, where the number of arrays is equal to the number of channels, and the length of the arrays is equal to the frame size.

For the generic implementation, just call process for each sample for each channel.

Parameters**frame**

[list] List of frames, where each frame is a 1-D numpy array.

Returns**list**

List of processed frames, with the same structure as the input frame.

process_xcore(*sample: float, channel=0*)

Take one new sample and return 1 processed sample.

For the generic implementation, scale and quantize the input, call the xcore-like implementation, then scale back to 1.0 = 0 dB.

Parameters**sample**

[float] The input sample to be processed.

channel

[int, optional] The channel index to process the sample on. Default is 0.

Returns

float

The processed output sample.



Copyright © 2024, XMOS Ltd

XMOS Ltd. is the owner or licensee of this design, code, or Information (collectively, the "Information") and is providing it to you "AS IS" with no warranty of any kind, express or implied and shall have no liability in relation to its use. XMOS Ltd makes no representation that the Information, or any particular implementation thereof, is or will be free from any claims of infringement and again, shall have no liability in relation to any such claims.

XMOS, XCORE, VocalFusion and the XMOS logo are registered trademarks of XMOS Ltd. in the United Kingdom and other countries and may not be used without written permission. Company and product names mentioned in this document are the trademarks or registered trademarks of their respective owners.

