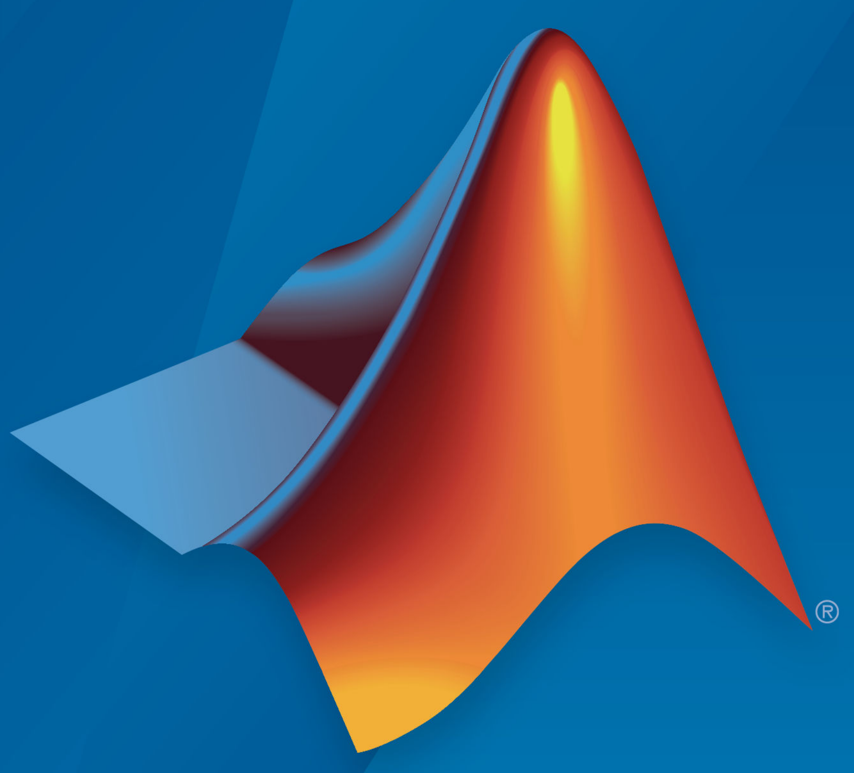
Audio System Toolbox™

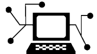
Reference



MATLAB® & SIMULINK®

R2018a

How to Contact MathWorks



Latest news: www.mathworks.com

Sales and services: www.mathworks.com/sales_and_services

User community: www.mathworks.com/matlabcentral

Technical support: www.mathworks.com/support/contact_us

Phone: 508-647-7000

The MathWorks, Inc.
3 Apple Hill Drive
Natick, MA 01760-2098

Audio System Toolbox ™ Reference Guide

© COPYRIGHT 2016 - 2018 by The MathWorks, Inc.

The software described in this document is furnished under a license agreement. The software may be used or copied only under the terms of the license agreement. No part of this manual may be photocopied or reproduced in any form without prior written consent from The MathWorks, Inc.

FEDERAL ACQUISITION: This provision applies to all acquisitions of the Program and Documentation by, for, or through the federal government of the United States. By accepting delivery of the Program or Documentation, the government hereby agrees that this software or documentation qualifies as commercial computer software or commercial computer software documentation as such terms are used or defined in FAR 12.212, DFARS Part 227.72, and DFARS 252.227-7014. Accordingly, the terms and conditions of this Agreement and only those rights specified in this Agreement, shall pertain to and govern the use, modification, reproduction, release, performance, display, and disclosure of the Program and Documentation by the federal government (or other entity acquiring for or through the federal government) and shall supersede any conflicting contractual terms or conditions. If this License fails to meet the government's needs or is inconsistent in any respect with federal procurement law, the government agrees to return the Program and Documentation, unused, to The MathWorks, Inc.

Trademarks

MATLAB and Simulink are registered trademarks of The MathWorks, Inc. See www.mathworks.com/trademarks for a list of additional trademarks. Other product or brand names may be trademarks or registered trademarks of their respective holders.

Patents

MathWorks products are protected by one or more U.S. patents. Please see www.mathworks.com/patents for more information.

Revision History

March 2016	Online only	New for Version 1.0 (Release 2016a)
September 2016	Online only	Revised for Version 1.1 (Release 2016b)
March 2017	Online only	Revised for Version 1.2 (Release 2017a)
September 2017	Online only	Revised for Version 1.3 (Release 2017b)
March 2018	Online only	Revised for Version 1.4 (Release 2018a)

Contents

1 Apps in Audio System Toolbox

2 Functions in Audio System Toolbox

3 System objects in Audio System Toolbox

4 Classes in Audio System Toolbox

5 Blocks in Audio System Toolbox

1

Apps in Audio System Toolbox

Impulse Response Measurer

Measure the impulse response of an audio system

Description

The **Impulse Response Measurer** app enables you to acquire, analyze, and export impulse response and frequency response measurements through a user interface.

Using this app, you can:

- Acquire impulse responses to create filters and generate models for offline simulations.
- Determine whether audio devices (loudspeakers, for example) meet time and frequency specifications.
- Optimize audio systems, such as automotive-acoustic systems, to match goal specifications.
- Acquire accurate impulse response measurements for use in acoustic reporting.

Open the Impulse Response Measurer App

MATLAB® Toolstrip: On the **Apps** tab, under **Signal Processing and Communications**, click the app icon.

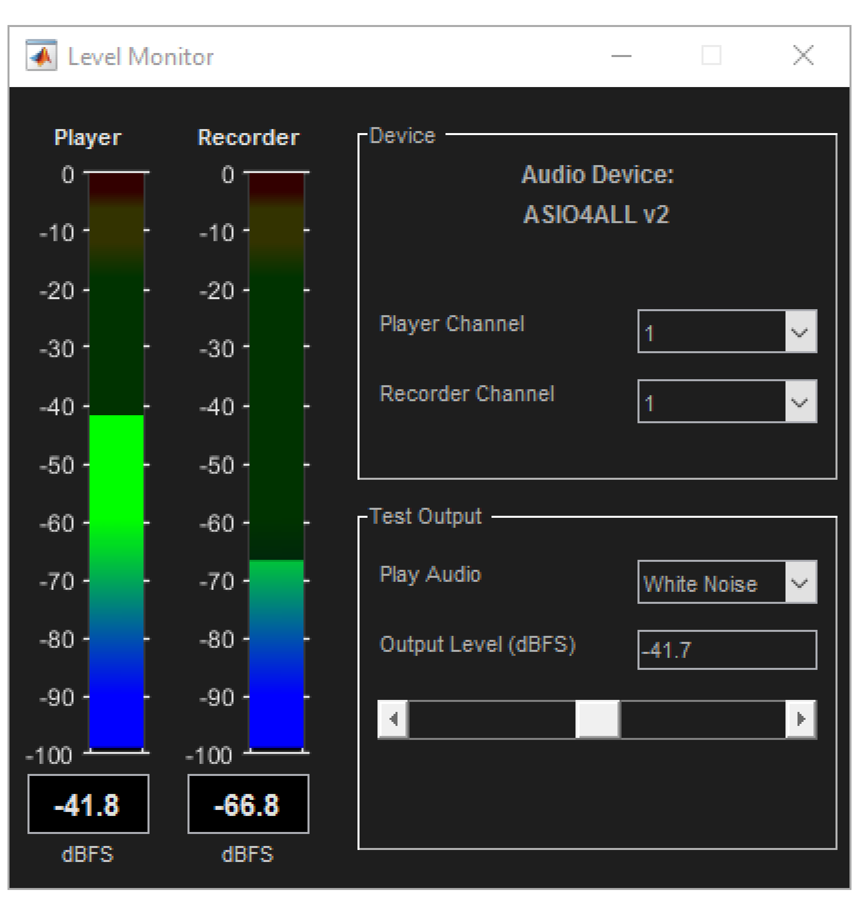
MATLAB Command prompt: Enter `impulseResponseMeasurer`.

Examples

Verify Input/Output Configuration

For large systems with multiple audio devices and multiple input and output channels, tracking how reported devices and channels correspond to physical devices can be difficult. The **Impulse Response Measurer** provides a level monitor so that you can verify your audio I/O configuration.

To open the level monitor, click **Level Monitor**, .



Choose a player and recorder channel, the test signal, and the output level. Verify that the level reported by the recorder reacts appropriately to level changes output by the player. Once you are satisfied that your system is configured correctly, close the level monitor and begin the impulse response capture.

- “Impulse Response Measurer Walkthrough”

Parameters

`Method` — Select excitation signal as MLS or swept sine wave
`MLS` (default) | `Exponential Swept Sine`

Select the excitation signal algorithm used to generate an impulse response measurement:

- `MLS` –– The Maximum Length Sequences (MLS) technique is based on the excitation of the acoustical space by a periodic pseudorandom signal. The impulse response is obtained by circular cross-correlation between the measured output and the test tone (MLS sequence). For more details, see [2].
- `Exponential Swept Sine` –– The swept sine measurement technique uses an exponential time-growing frequency seep as an output signal. The output signal is recorded, and deconvolution is used to recover the impulse response from the swept sine tone. For more details, see [1]. The swept sine technique enables you to modify additional **Advanced Settings** to control the excitation signal. The advanced settings apply per run:

  - **Sweep start frequency**
  - **Sweep stop frequency**
  - **Sweep duration**
  - **End silence duration**

  The value of the **End silence duration** is read-only and depends on the **Sweep duration** and **Duration per Run (s)**: End silence duration = Duration per Run − Sweep duration

References

[1] Farino, Angelo. "Advancements in Impulse Response Measurements by Sine Sweeps." Presented at the *Audio Engineering Society 122nd Convention*, Vienna, Austria, 2007.

[2] Guy-Bart, Stan, Jean-Jacques Embrachts, and Dominique Archambeau. "Comparison of Different Impulse Response Measurement Techniques." *Journal of Audio Engineering Society*. Vol. 50, Issue 4, 2002, pp. 246–262.

[3] Armelloni, Enrico, Christian Giottoli, and Angelo Farina. "Implement of Real-Time Partitioned Convolution on a DSP Board." *Application of Signal Processing to Audio and Acoustics, 2003 IEEE Workshop*, pp. 71–74. IEEE, 2003.

See Also

System Objects
`audioPlayerRecorder` | `reverberator` | `splMeter`

Topics

"Impulse Response Measurer Walkthrough"

Introduced in R2018a

Audio Test Bench

Debug, test, and tune audio plugin

Description

The **Audio Test Bench** provides a graphical interface through which you can develop, debug, test, and tune your audio plugin in real time. You can interact with properties of your audio plugin using associated parameter graphical widgets. See `audioPluginParameter` for more information.

Using the **Audio Test Bench**, you can:

- Debug your audio plugin.
- Simulate your audio plugin as generated in a digital audio workstation (DAW).
- Visualize your processing with time-domain and frequency-domain scopes.
- Interactively synchronize MIDI controls to plugin properties.
- Run validation checks and generate VST plugins.

Develop and Test Features

Button		Description
	Run	Run your audio plugin in an audio stream loop using the specified input and output configuration. You can tune parameters of your audio processing algorithm in real time. The MATLAB command line and objects used by the test bench are locked while the test bench is running.
	Pause (appears while test bench runs)	Pause audio stream loop. The MATLAB command line is released. Objects used by the test bench remain locked.
	Step Forward	Call the processing function of your audio plugin one time in an audio stream loop, with input and output specified by your input and output configuration.

Button		Description
	Stop	Stop the audio stream loop. The MATLAB command line and objects used by the test bench are released.
	Reset	Reset internal states of your audio plugin and set parameters to their initial values.
	View Source Code	Open the source file of your audio plugin.
	Synchronize to MIDI Controls	Start the `configureMIDI` user interface (UI) for your plugin object.
	Time Scope	Open an instance of `dsp.TimeScope`, which provides a time-domain visualization of the output from your audio stream loop.
	Spectrum Analyzer	Open an instance of `dsp.SpectrumAnalyzer`, which provides a frequency-domain visualization of the output from your audio stream loop.
	Generate VST 2 Audio Plugin	Open a UI to validate and generate your plugin object. For Audio System Toolbox System objects, the **Audio Test Bench** creates an `audioPlugin` class using the `createAudioPluginClass` method of the object. The created plugin class is used to generate a plugin object. For more information, see `validateAudioPlugin`, `generateAudioPlugin`, and the `createAudioPluginClass` method of your System object™.
	Generate MATLAB Script	Generate a MATLAB script implementation of your audio test bench.
	Help	Open MATLAB documentation for **Audio Test Bench**.

Button		Description
	Configure Input	Open the input configuration UI. The UI options depend on your choice of input to the audio stream loop. See the corresponding documentation for your input choice: • Audio File Reader -- dsp.AudioFileReader • Audio Device Reader -- audioDeviceReader • Audio Oscillator -- audioOscillator • Wavetable Synthesizer -- wavetableSynthesizer • Chirp Signal -- dsp.Chirp • Colored Noise -- dsp.ColoredNoise
	Configure Output	Open the output configuration UI. The UI options depend on whether you choose Audio File Writer or Audio Device Writer for the output from your audio stream loop. If you choose to output Both, two dialog boxes open: one for the Audio File Writer and one for the Audio Device Writer. For more information, see dsp.AudioFileWriter and audioDeviceWriter.

Open the Audio Test Bench App

MATLAB command prompt: Enter `audioTestBench`.

Examples

- "Audio Test Bench Walkthrough"

Programmatic Use

`audioTestBench pluginClass` opens the **Audio Test Bench** for an instance of `pluginClass`. The input to `audioTestBench` must derive from the `audioPlugin` class, not the `audioPluginSource` class.

audioTestBench(pluginClassInstance) opens the **Audio Test Bench** for pluginClassInstance, where pluginClassInstance is an instance of an audio plugin class. The input to audioTestBench must derive from the audioPlugin class, not the audioPluginSource class.

audioTestBench ASTSystemObject opens the **Audio Test Bench** for an instance of a compatible Audio System Toolbox System object.

audioTestBench(ASTSystemObjectInstance) opens the **Audio Test Bench** for ASTSystemObjectInstance, where ASTSystemObjectInstance is an instance of a compatible Audio System Toolbox System object.

audioTestBench(hostedPlugin) opens the **Audio Test Bench** for hostedPlugin, where hostedPlugin is an object returned by the loadAudioPlugin function.

audioTestBench(pluginPath) opens the **Audio Test Bench** for pluginPath, where pluginPath is the location of an external plugin. Use the full path to specify the audio plugin you want to host. If the plugin is located in the current folder, specify it by its name.

See Also

Functions
audioPluginInterface | audioPluginParameter | generateAudioPlugin | validateAudioPlugin

Classes
audioPlugin | audioPluginSource

Topics
"Audio Test Bench Walkthrough"
"What Are DAWs, Audio Plugins, and MIDI Controllers?"
"Design an Audio Plugin"
"Audio Plugin Example Gallery"

Introduced in R2016a

Functions in Audio System Toolbox

mididevinfo

MIDI device information

Syntax

```
mididevinfo
deviceInformation = mididevinfo
```

Description

`mididevinfo` displays a table containing information about the MIDI devices attached to the system.

`deviceInformation = mididevinfo` returns a structure, `deviceInformation`, containing information about the MIDI devices attached to the system.

Note Before starting MATLAB, connect your MIDI device to your computer and turn on the device. For connection instructions, see the instructions for your MIDI device. If you start MATLAB before connecting your device, MATLAB might not recognize your device when you connect it. To correct the problem, restart MATLAB with the device already connected.

Examples

Display MIDI Device Connections

Call `mididevinfo` to display a table containing information about the MIDI devices attached to your system.

```
mididevinfo
```

```
  MIDI devices available:
  ID  Direction  Interface   Name
```

```
   0   output    MMSystem   'Microsoft MIDI Mapper'
   1    input    MMSystem   'USB MIDI Interface '
   2   output    MMSystem   'Microsoft GS Wavetable Synth'
   3   output    MMSystem   'USB MIDI Interface '
```

Return Structure of MIDI Device Connections

Call `mididevinfo` with an output argument to return a structure containing MIDI device information.

```
deviceInformation = mididevinfo
```

```
deviceInformation = struct with fields:
     input: [0×0 struct]
    output: [1×2 struct]
```

The `deviceInformation` structure has two fields: `input` and `output.` Both `input` and `output` contain arrays of structures. Each member has three fields: `Name`, `Interface`, and `ID`. Get the device information for the output Microsoft GS Wavetable Synth device.

```
deviceInformation.output(2)
```

```
ans = struct with fields:
         Name: 'Microsoft GS Wavetable Synth'
    Interface: 'MMSystem'
           ID: 1
```

Output Arguments

`deviceInformation` — Description of available devices
struct

Description of available devices, returned as nested structures. The outer structure has two fields: `input` and `output`. The input and output values are arrays of structures, and each member has three fields: `Name`, `Interface`, and `ID`.

Data Types: `struct`

See Also

`mididevice` | `midimsg` | `midireceive` | `midisend`

Topics

"MIDI Device Interface"

External Websites

MIDI Manufacturers Association

Introduced in R2018a

pitch

Estimate fundamental frequency of audio signal

Syntax

```
f0 = pitch(audioIn,fs)
f0 = pitch(audioIn,fs,Name,Value)
[f0,loc] = pitch( ___ )
```

Description

`f0 = pitch(audioIn,fs)` returns estimates of the fundamental frequency over time for the audio input, `audioIn`, with sample rate `fs`. Columns of the input are treated as individual channels.

`f0 = pitch(audioIn,fs,Name,Value)` specifies options using one or more `Name,Value` pair arguments.

`[f0,loc] = pitch( ___ )` returns the locations, `loc`, associated with fundamental frequency estimates.

Examples

Estimate Pitch of Speech Signal Using Default Parameters

Read in an audio file and then call the `pitch` function with default parameters.

```
[audioIn,fs] = audioread('Counting-16-44p1-mono-15secs.wav');
[f0,idx] = pitch(audioIn,fs);
```

Plot the audio signal and pitch contour.

```
subplot(2,1,1)
plot(audioIn)
```

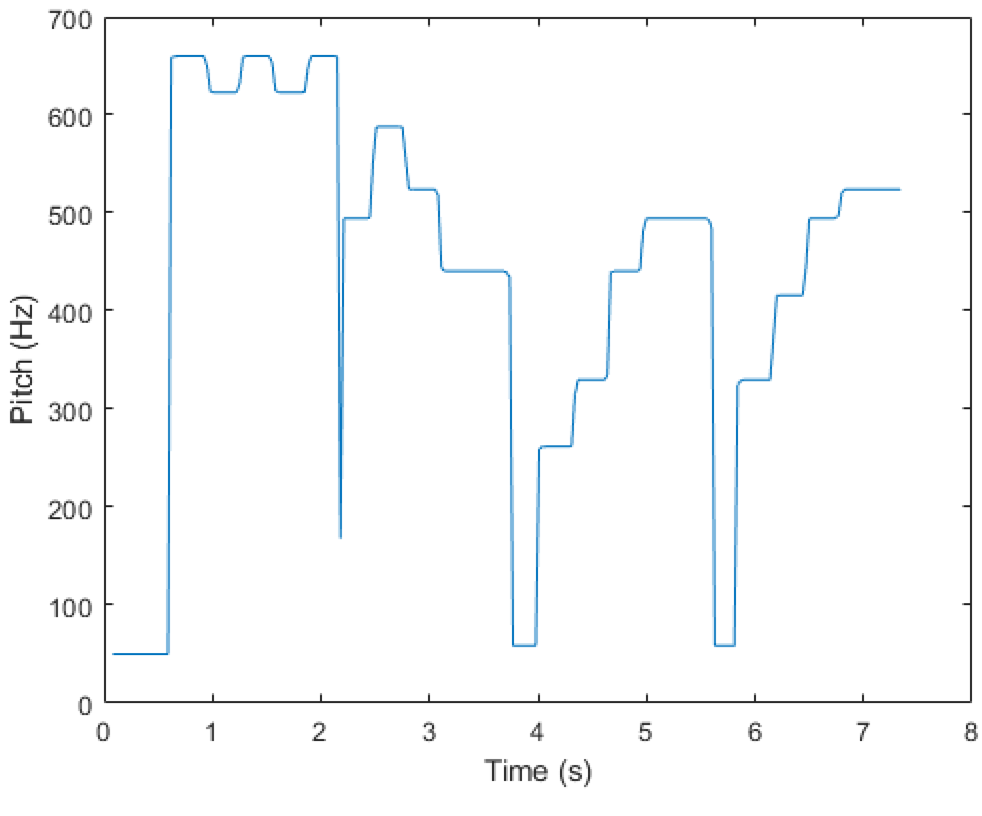
```
ylabel('Amplitude')

subplot(2,1,2)
plot(idx,f0)
ylabel('Pitch (Hz)')
xlabel('Sample Number')
```

The `pitch` function estimates the fundamental frequency of the input signal at locations determined by the `WindowLength` and `OverlapLength` name-value pairs.

Estimate Pitch of Musical Signal Using Nondefault Parameters

Load an audio file of the introduction to Für Elise and the sample rate of the audio. Call the `pitch` function using the pitch estimate filter (`PEF`), a search range from 50 Hz to 800 Hz, a window length of 80 ms, and an overlap of 50 ms. Plot the results and listen to the song to verify the fundamental frequency estimates returned by the `pitch` function.

```
load FurElise.mat song fs

[f0,loc] = pitch(song,fs, ...
    'Method','PEF', ...
    'Range',[50 800], ...
    'WindowLength',round(fs*0.08), ...
    'OverlapLength',round(fs*0.05));

t = loc/fs;
plot(t,f0)
ylabel('Pitch (Hz)')
xlabel('Time (s)')
```




```
sound(song,fs)
```

Compare Pitch Detection Algorithms

The different methods of estimating pitch provide trade-offs in terms of noise robustness, accuracy, optimal lag, and computation expense. In this example, you compare the performance of different pitch detection algorithms in terms of gross pitch error (GPE) and computation time under different noise conditions.

Prepare Test Signals

Load an audio file and determine the number of samples it has. Also load the true pitch corresponding to the audio file. The true pitch was determined as an average of several third-party algorithms on the clean speech file.

```
[audioIn,fs] = audioread('Counting-16-44p1-mono-15secs.wav');
numSamples = size(audioIn,1);
load TruePitch.mat truePitch
```

Create test signals by adding noise to the audio signal at given SNRs. The `mixSNR` function is a convenience function local to this example, which takes a signal, noise, and requested SNR and returns a noisy signal at the request SNR.

```
testSignals = zeros(numSamples,4);

turbine = audioread('Turbine-16-44p1-mono-22secs.wav');
testSignals(:,1) = mixSNR(audioIn,turbine,20);
testSignals(:,2) = mixSNR(audioIn,turbine,0);

whiteNoiseMaker = dsp.ColoredNoise('Color','white','SamplesPerFrame',size(audioIn,1));
testSignals(:,3) = mixSNR(audioIn,whiteNoiseMaker(),20);
testSignals(:,4) = mixSNR(audioIn,whiteNoiseMaker(),0);
```

Save the noise conditions and algorithm names as cell arrays for labeling and indexing.

```
noiseConditions = {'Turbine (20 dB)','Turbine (0 dB)','WhiteNoise (20 dB)','WhiteNoise
algorithms = {'NCF','PEF','CEP','LHS','SRH'};
```

Run Pitch Detection Algorithms

Preallocate arrays to hold pitch decisions for each algorithm and noise condition pair, and the timing information. In a loop, call the `pitch` function on each combination of algorithm and noise condition. Each algorithm has an optimal window length associated with it. In this example, for simplicity, you use the default window length for all algorithms. Use a 3-element median filter to smooth the pitch decisions.

```
f0 = zeros(numel(truePitch),numel(algorithms),numel(noiseConditions));
algorithmTimer = zeros(numel(noiseConditions),numel(algorithms));

for k = 1:numel(noiseConditions)
    x = testSignals(:,k);
    for i = 1:numel(algorithms)
        tic
```

```
        f0temp = pitch(x,fs, ...
            'Range',[50 300], ...
            'Method',algorithms{i}, ...
            'MedianFilterLength',3);
        algorithmTimer(k,i) = toc;
        f0(1:max(numel(f0temp),numel(truePitch)),i,k) = f0temp;
    end
end
```

Compare Gross Pitch Error

Gross pitch error (GPE) is a popular metric when comparing pitch detection algorithms. GPE is defined as the proportion of pitch decisions for which the relative error is higher than a given threshold, traditionally 20% in speech studies. Calculate the GPE and print it to the Command Window.

```
idxToCompare = ~isnan(truePitch);
truePitch = truePitch(idxToCompare);
f0 = f0(idxToCompare,:,:);

p = 0.20;
GPE = mean( abs(f0(1:numel(truePitch),:,:) - truePitch) > truePitch.*p).*100;

for ik = 1:numel(noiseConditions)
    fprintf('\nGPE (p = %0.2f), Noise = %s.\n',p,noiseConditions{ik});
    for i = 1:size(GPE,2)
        fprintf('- %s : %0.1f %%\n',algorithms{i},GPE(1,i,ik))
    end
end
```

```
GPE (p = 0.20), Noise = Turbine (20 dB).
- NCF : 4.0 %
- PEF : 0.9 %
- CEP : 18.3 %
- LHS : 15.5 %
- SRH : 8.0 %

GPE (p = 0.20), Noise = Turbine (0 dB).
- NCF : 13.9 %
- PEF : 13.0 %
- CEP : 27.6 %
- LHS : 12.4 %
- SRH : 53.3 %

GPE (p = 0.20), Noise = WhiteNoise (20 dB).
```

```
- NCF : 3.1 %
- PEF : 0.6 %
- CEP : 19.2 %
- LHS : 17.6 %
- SRH : 4.0 %

GPE (p = 0.20), Noise = WhiteNoise (0 dB).
- NCF : 3.4 %
- PEF : 1.5 %
- CEP : 37.5 %
- LHS : 10.8 %
- SRH : 3.4 %
```

Calculate the average time it takes to process one second of data for each of the algorithms and print the results.

```
aT = sum(algorithmTimer)./((numSamples/fs)*numel(noiseConditions));
for ik = 1:numel(algorithms)
    fprintf('- %s : %0.3f (s)\n',algorithms{ik},aT(ik))
end
```

```
- NCF : 0.105 (s)
- PEF : 0.349 (s)
- CEP : 0.087 (s)
- LHS : 0.171 (s)
- SRH : 0.373 (s)
```

Determine Pitch Contour using pitch and voiceActivityDetector

Read in an entire speech file and determine the fundamental frequency of the audio using the `pitch` function. Then use the `voiceActivityDetector` to remove irrelevant pitch information that does not correspond to the speaker.

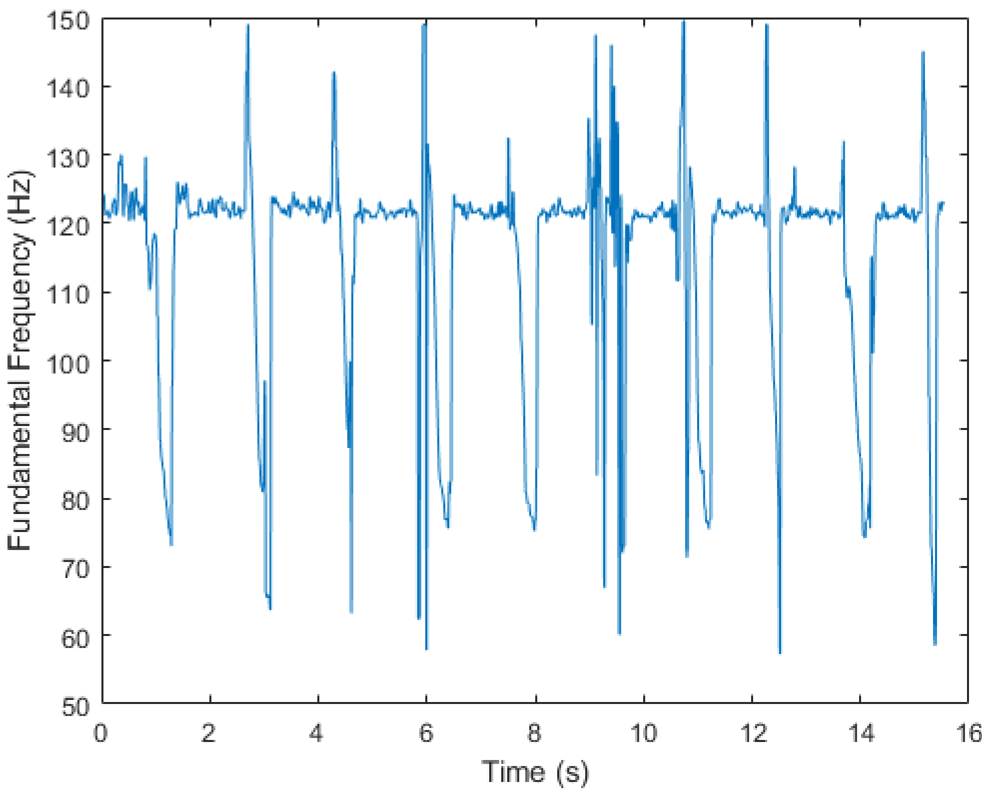
Read in the audio file and associated sample rate.

```
[audio,fs] = audioread('Counting-16-44p1-mono-15secs.wav');
```

Specify pitch detection using a 50 ms window length and 40 ms overlap (10 ms hop). Specify that the `pitch` function searches for the fundamental frequency over the range 50-150 Hz and postprocesses the results with a median filter. Plot the results.

```
windowLength = round(0.05*fs);
overlapLength = round(0.04*fs);
```

```
hopLength = windowLength - overlapLength;

[f0,loc] = pitch(audio,fs, ...
    'WindowLength',windowLength, ...
    'OverlapLength',overlapLength, ...
    'Range',[50 150], ...
    'MedianFilterLength',3);

plot(loc/fs,f0)
ylabel('Fundamental Frequency (Hz)')
xlabel('Time (s)')
```



Create a `dsp.AsyncBuffer` System object™ to chunk the audio signal into overlapped frames. Also create a `voiceActivityDetector` System object™ to determine if the frames contain speech.

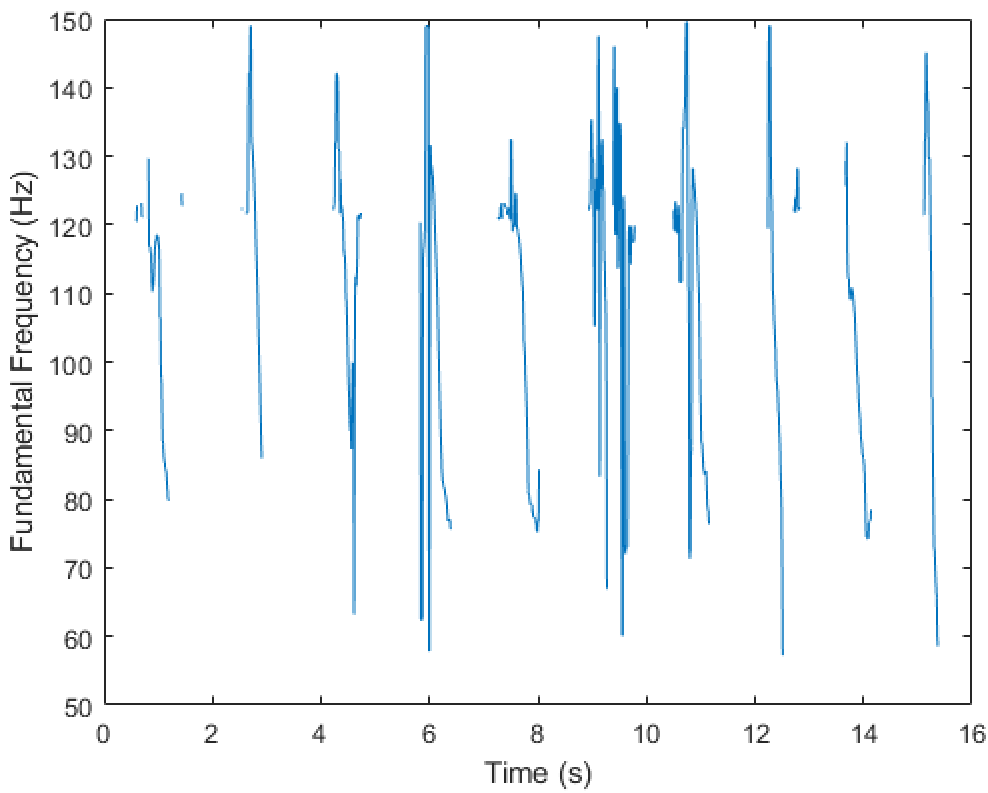
```
buffer = dsp.AsyncBuffer(numel(audio));
write(buffer,audio);
VAD = voiceActivityDetector;
```

While there are enough samples to hop, read from the buffer and determine the probability that the frame contains speech. To mimic the decision spacing in time of the `pitch` function, the first frame read from the buffer has no overlap.

```
n = 1;
probabilityVector = zeros(numel(loc),1);
while buffer.NumUnreadSamples >= hopLength
    if n==1
        x = read(buffer,windowLength);
    else
        x = read(buffer,windowLength,overlapLength);
    end
    probabilityVector(n) = VAD(x);
    n = n+1;
end
```

Use the probability vector determined by the `voiceActivityDetector` to plot a pitch contour for the speech file that corresponds to regions of speech.

```
validIdx = probabilityVector>0.99;
loc(~validIdx) = nan;
f0(~validIdx) = nan;
plot(loc/fs,f0)
ylabel('Fundamental Frequency (Hz)')
xlabel('Time (s)')
```



- “Speaker Identification Using Pitch and MFCC”

Input Arguments

audioIn — Audio input signal
vector | matrix

Audio input signal, specified as a vector or matrix. The columns of the matrix are treated as individual audio channels.

Data Types: `single` | `double`

fs — Sample rate (Hz)
positive scalar

Sample rate of the input signal in Hz, specified as a positive scalar.

The sample rate must be greater than or equal to twice the upper bound of the search range. Specify the search range using the Range name-value pair.

Data Types: single | double

Name-Value Pair Arguments

Specify optional comma-separated pairs of Name,Value arguments. Name is the argument name and Value is the corresponding value. Name must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as Name1,Value1,...,NameN,ValueN.

Example: pitch(audioIn,fs,'Range',[50,150],'Method','PEF')

Range — Search range for pitch estimates
[50,400] (default) | two-element row vector with increasing positive integer values

Search range for pitch estimates, specified as the comma-separated pair consisting of 'Range' and a two-element row vector with increasing positive integer values. The function searches for a best estimate of the fundamental frequency within the upper and lower band edges specified by the vector, according to the algorithm specified by Method. The range is inclusive and units are in Hz.

Valid values for the search range depend on the sample rate, fs, and on the values of WindowLength and Method:

Method	Minimum Range	Maximum Range
'NCF'	fs/WindowLength < Range(1)	Range(2) < fs/2
'PEF'	10 < Range(1)	Range(2) < min(4000,fs/2)
'CEP'	fs/(2^nextpow2(2*WindowLength-1)) < Range(1)	Range(2) < fs/2
'LHS'	1 < Range(1)	Range(2) < fs/5 - 1

Method	Minimum Range	Maximum Range
'SRH'	`1 < Range(1)`	`Range(2) < fs/5 - 1`

Data Types: `single` | `double`

`WindowLength` — Number of samples in analysis window
`round(fs*0.052)` (default) | integer

Number of samples in the analysis window, specified as the comma-separated pair consisting of '`WindowLength`' and an integer in the range [1, min(size(`audioIn`,1), 192000)]. Typical analysis windows are in the range 20–100 ms. The default window length is 52 ms.

Data Types: `single` | `double`

`OverlapLength` — Number of samples of overlap between adjacent analysis windows
`round(fs*0.042)` (default) | integer

Number of samples of overlap between adjacent analysis windows, specified as the comma-separated pair consisting of '`OverlapLength`' and an integer in the range (`-inf`,`WindowLength`). A negative overlap length indicates non-overlapping analysis windows.

Data Types: `single` | `double`

`Method` — Method used to estimate pitch
`'NCF'` (default) | `'PEF'` | `'CEP'` | `'LHS'` | `'SRH'`

Method used to estimate pitch, specified as the comma-separated pair consisting of '`Method`' and `'NCF'`, `'PEF'`,`'CEP'`, `'LHS'`, or `'SRH'`. The different methods of calculating pitch provide trade-offs in terms of noise robustness, accuracy, and computation expense. The algorithms used to calculate pitch are based on the following papers:

- `'NCF'` –– Normalized Correlation Function [1]
- `'PEF'` –– Pitch Estimation Filter [2]. The function does not use the amplitude compression described by the paper.
- `'CEP'` –– Cepstrum Pitch Determination [3]
- `'LHS'` –– Log-Harmonic Summation [4]

- 'SRH' –– Summation of Residual Harmonics [5]

Data Types: char | string

MedianFilterLength — Median filter length used to smooth pitch estimates over time
1 (default) | positive integer

Median filter length used to smooth pitch estimates over time, specified as the comma-separated pair consisting of 'MedianFilterLength' and a positive integer. The default, 1, corresponds to no median filtering. Median filtering is a postprocessing technique used to remove outliers while estimating pitch. The function uses movmedian after estimating the pitch using the specified Method.

Data Types: single | double

Output Arguments

f0 — Estimated fundamental frequency (Hz)
scalar | vector | matrix

Estimated fundamental frequency, in Hz, returned as a scalar, vector, or matrix. The number of rows returned depends on the values of the WindowLength and OverlapLength name-value pairs, and on the input signal size. The number of columns (channels) returned depends on the number of columns of the input signal size.

Data Types: single | double

loc — Locations associated with fundamental frequency estimations
scalar | vector | matrix

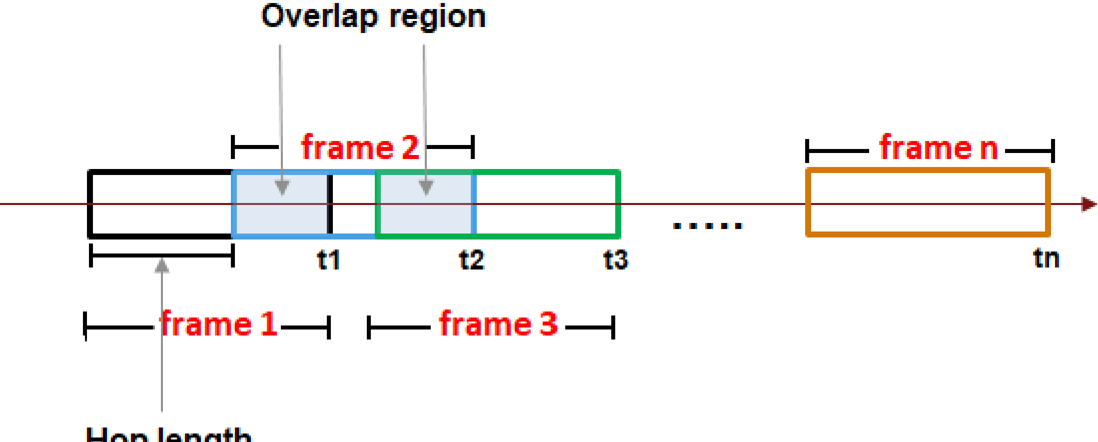
Locations associated with fundamental frequency estimations, returned as a scalar, vector, or matrix the same size as f0.

Fundamental frequency is estimated locally over a region of WindowLength samples. The values of loc correspond to the most recent sample (largest sample number) used to estimate fundamental frequency.

Data Types: single | double

Algorithms

The `pitch` function segments the audio input according to the `WindowLength` and `OverlapLength` arguments. The fundamental frequency is estimated for each frame. The locations output, `loc` contains the most recent samples (largest sample numbers) of the corresponding frame.



For a description of the algorithms used to estimate the fundamental frequency, consult the corresponding references:

- `'NCF'` -- Normalized Correlation Function [1]
- `'PEF'` -- Pitch Estimation Filter [2]. The function does not use the amplitude compression described by the paper.
- `'CEP'` -- Cepstrum Pitch Determination [3]
- `'LHS'` -- Log-Harmonic Summation [4]
- `'SRH'` -- Summation of Residual Harmonics [5]

References

[1] Atal, B.S. "Automatic Speaker Recognition Based on Pitch Contours." *The Journal of the Acoustical Society of America*. Vol. 52, No. 6B, 1972, pp. 1687–1697.

[2] Gonzalez, Sira, and Mike Brookes. "A Pitch Estimation Filter robust to high levels of noise (PEFAC)." 19th European Signal Processing Conference. Barcelona, 2011, pp. 451–455.

[3] Noll, Michael A. "Cepstrum Pitch Determination." *The Journal of the Acoustical Society of America*. Vol. 31, No. 2, 1967, pp. 293–309.

[4] Hermes, Dik J. "Measurement of Pitch by Subharmonic Summation." *The Journal of the Acoustical Society of America*. Vol. 83, No. 1, 1988, pp. 257–264.

[5] Drugman, Thomas, and Abeer Alwan. "Joint Robust Voicing Detection and Pitch Estimation Based on Residual Harmonics." Proceedings of the Annual Conference of the International Speech Communication Association, INTERSPEECH. 2011, pp. 1973–1976.

See Also

Functions
`mfcc`

System Objects
`cepstralFeatureExtractor` | `voiceActivityDetector`

Topics

"Speaker Identification Using Pitch and MFCC"

Introduced in R2018a

mfcc

Extract mfcc, log energy, delta, and delta-delta of audio signal

Syntax

```
coeffs = mfcc(audioIn,fs)
coeffs = mfcc( ___ ,Name,Value)
[coeffs,delta,deltaDelta,loc] = mfcc( ___ )
```

Description

`coeffs = mfcc(audioIn,fs)` returns the mel frequency cepstral coefficients (MFCCs) for the audio input, sampled at a frequency of `fs` Hz.

`coeffs = mfcc( ___ ,Name,Value)` sets each property `Name` to the specified `Value`. Unspecified properties have default values..

Example: `[coeffs] = mfcc(audioIn,fs,'LogEnergy','Replace')` returns mel frequency cepstral coefficients for the audio input signal sampled at `fs` Hz. The first coefficient in the `coeffs` vector is replaced with the log energy value.

`[coeffs,delta,deltaDelta,loc] = mfcc( ___ )` returns the delta, delta-delta, and location of samples corresponding to each window of data.

Examples

Compute Mel Frequency Cepstral Coefficients

Compute the mel frequency cepstral coefficients of a speech signal using the `mfcc` function. The function returns `delta`, the change in coefficients, and `deltaDelta`, the change in delta values. The log energy value that the function computes can prepend the coefficients vector or replace the first element of the coefficients vector. This is done based on whether you set the `'LogEnergy'` argument to `'Append'` or `'Replace'`.

Read an audio signal from the `'Counting-16-44p1-mono-15secs.wav'` file using the `audioread` function. The `mfcc` function processes the entire speech data in a batch. The default `DeltaWindowLength` is 2. Therefore, `delta` is computed as the difference between the current coefficients and the previous coefficients. `deltaDelta` is computed as the difference between the current and the previous delta values. Based on the number of input rows, the window length, and the hop length, `mfcc` partitions the speech into 1551 frames and computes the cepstral features for each frame. Each row in the `coeffs` matrix corresponds to the log-energy value followed by the 13 mel-frequency cepstral coefficients for the corresponding frame of the speech file. The function also computes `loc`, the location of the last sample in each input frame.

```
[audioIn,fs] = audioread('Counting-16-44p1-mono-15secs.wav');
[coeffs,delta,deltaDelta,loc] = mfcc(audioIn,fs);
```

Input Arguments

`audioIn` — Input signal
vector | matrix

Input signal, specified as a vector or matrix. If specified as a matrix, `mfcc` treats the columns of the matrix as individual audio channels.

Data Types: `single` | `double`

`fs` — Sample rate in Hz
positive scalar

Sample rate of the input signal in Hz, specified as a positive scalar.

Data Types: `single` | `double`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `[coeffs,delta,deltaDelta,loc] = mfcc(audioIn,fs,'LogEnergy','Replace','DeltaWindowLength',5)` returns

mel frequency cepstral coefficients for the audio input signal sampled at `fs` Hz. The first coefficient in the `coeffs` vector is replaced with the log energy value. A set of 5 cepstral coefficients is used to compute the delta and the delta-delta values.

`WindowLength` — Number of samples in analysis window
`round(fs*0.03)` (default) | positive scalar integer

Number of samples in analysis window used to calculate the coefficients, specified as an integer greater than or equal to `2`. If unspecified, the `'WindowLength'` value defaults to `round(fs*0.03)`. Window length must be in the range `[2,size(audioIn,1)]`.

Data Types: `single` | `double`

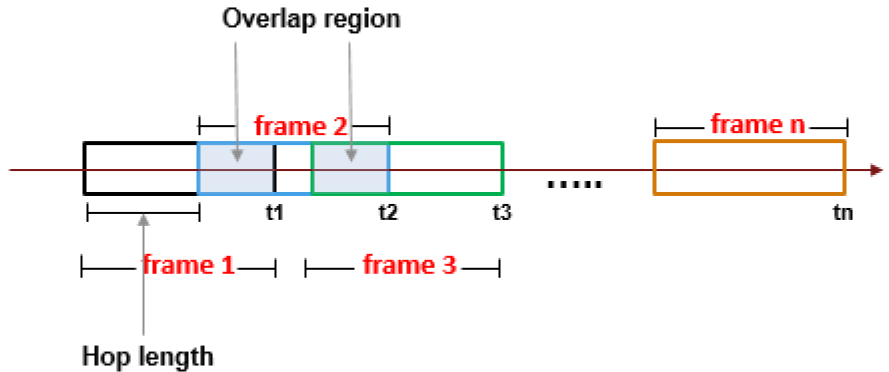
`OverlapLength` — Number of overlapping samples between adjacent windows
`round(fs*0.02)` (default) | integer

Number of samples which overlap or underlap between the adjacent windows. An `'OverlapLength'` value that is:

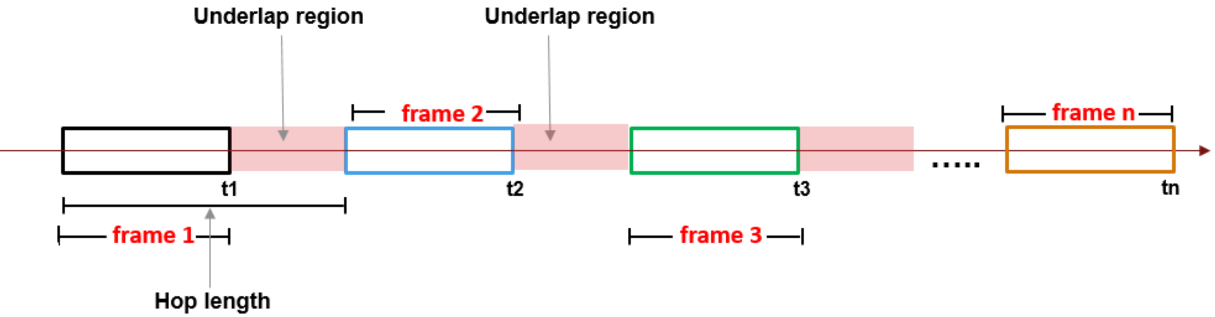
- Positive indicates an overlap between adjacent windows.
- Negative indicates an underlap between adjacent windows.
- Zero indicates no overlap between adjacent windows.

The `'OverlapLength'` value must be set to less than the `'WindowLength'`.

Here is how the overlapping frames look:



Here is how the underlapping frames look:



Data Types: `single` | `double`

`NumCoeffs` — Number of coefficients returned

13 (default) | positive scalar integer

Number of coefficients returned for each window of data, specified as an integer in the range [2 *v*], where *v* is the number of valid passbands.

The number of valid passbands is defined as `sum(BandEdges <= floor(fs/2))-2`. A passband is valid if its edges fall below `fs/2`, where *fs* is the sample rate of the input audio signal, specified as the second argument, `fs`.

The `mfcc` function uses a filter bank of 40 half-overlapped triangles, with band edges defined by the table:

Filters	Passband Edges (Hz)
Filter 1	[133 267]
Filter 2	[200 333]
Filter 3	[267 400]
Filter 4	[333 467]
Filter 5	[400 533]
Filter 6	[467 600]
Filter 7	[533 667]
Filter 8	[600 733]
Filter 9	[667 800]

Filters	Passband Edges (Hz)
Filter 10	[733 867]
Filter 11	[800 933]
Filter 12	[867 999]
Filter 13	[933 1071]
Filter 14	[999 1147]
Filter 15	[1071 1229]
Filter 16	[1147 1316]
Filter 17	[1229 1410]
Filter 18	[1316 1510]
Filter 19	[1410 1618]
Filter 20	[1510 1733]
Filter 21	[1618 1856]
Filter 22	[1733 1988]
Filter 23	[1856 2130]
Filter 24	[1988 2281]
Filter 25	[2130 2444]
Filter 26	[2281 2618]
Filter 27	[2444 2804]
Filter 28	[2618 3004]
Filter 29	[2804 3217]
Filter 30	[3004 3446]
Filter 31	[3217 3692]
Filter 32	[3446 3954]
Filter 33	[3692 4236]
Filter 34	[3954 4537]
Filter 35	[4236 4860]
Filter 36	[4537 5206]
Filter 37	[4860 5577

Filters	Passband Edges (Hz)
Filter 38	[5206 5973]
Filter 39	[5577 6399]
Filter 40	[5973 6854]

The passband edges in the table are rounded for readability. For exact edges, see the default settings of the `cepstralFeatureExtractor` System object.

Data Types: `single` | `double`

`FFTLength` — Number of bins for calculating DFT
`WindowLength` (default) | positive scalar integer

Number of bins used to calculate the DFT of windowed input samples. The FFT length value must be greater than or equal to the `'WindowLength'` value. The `'WindowLength'` argument specifies the number of rows in the windowed input. By default, the FFT length value is set to the `'WindowLength'`.

Data Types: `single` | `double`

`DeltaWindowLength` — Number of coefficients for calculating delta and delta-delta
`2` (default) | odd integer greater than 2

Number of coefficients used to calculate the delta and the delta-delta values, specfied as 2 or an odd integer greater than 2.

If `'DeltaWindowLength'` is set to `2`, the `delta` is given by the difference between the current coefficients and the previous coefficients, $delta = currCoeffs - prevCoeffs$

If `'DeltaWindowLength'` is set to an odd integer greater than `2`, the delta values are given by the following equation:

$$delta = \frac{\sum_{k=-M}^{M} k \cdot coeffs(k,:)}{\sum_{k=-M}^{M} k^2}$$

The function uses a least-square approximation to the local slope over a region around the current time sample. The delta cepstral values are computed by fitting the cepstral

coefficients of neighboring frames (*M* frames before the current frame and *M* frames after the current frame) by a straight line. For details, see [1].

Data Types: `single` | `double`

LogEnergy — Specify how the log energy is shown
`'Append'` (default) | `'Replace'` | `'Ignore'`

Specify how the log energy is shown in the coefficients vector output, specified as:

- `'Append'` -– The function prepends the log energy to the coefficients vector. The length of the coefficients vector is 1 + `NumCoeffs`.
- `'Replace'` -– The function replaces the first coefficient with the log energy of the signal. The length of the coefficients vector is `NumCoeffs`.
- `'Ignore'` -– The object does not calculate or return the log energy.

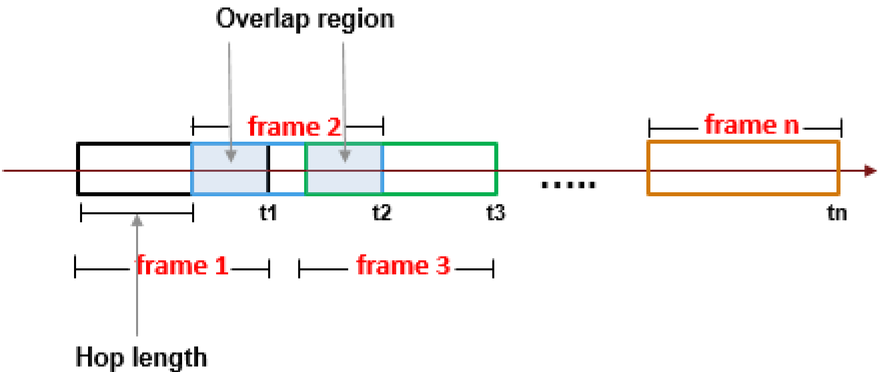
Data Types: `char` | `string`

Output Arguments

`coeffs` — Mel frequency cepstral coefficients (MFCCs)
matrix | array

Mel frequency cepstral coefficients, returned as an *L*-by-*M* matrix or an *L*-by-*M*-by-*N* array, where,

- *L* -– Number of frames the audio signal is partitioned into. The `'WindowLength'` and `'OverlapLength'` properties control this dimension.



The number of audio frames, L, is computed using the following equation:

$$L = \text{floor}\left(\left(nRows - winLen\right)/hopLen\right) + 1$$

- $nRows$ -- Number of input rows.
- $winLen$ -- Number of samples in the analysis window, specified by the `'WindowLength'` argument. If not specified, the window length is `round(fs*0.03)`.
- $hopLen$ -- Number of samples in the current frame before the start of the next frame. Hop length is given by $hopLen = WindowLength - OverlapLength$.

- M -- Number of coefficients returned per frame. This value is determined by the `NumCoeffs` and `LogEnergy` properties.

  When the `LogEnergy` property is set to:

  - `'Append'` -- The object prepends the log energy value to the coefficients vector. The length of the coefficients vector is 1 + `NumCoeffs`.
  - `'Replace'` -- The object replaces the first coefficient with the log energy of the signal. The length of the coefficients vector is `NumCoeffs`.
  - `'Ignore'` -- The object does not calculate or return the log energy.
- N -- Number of input channels (columns).

Data Types: `single` | `double`

delta — Change in coefficients
matrix | array

Change in coefficients from one frame of data to another, returned as an *L*-by-*M* matrix or an *L*-by-*M*-by-*N* array. The `delta` array is of the same size and data type as the `coeffs` array.

If `'DeltaWindowLength'` is set to `2`, the `delta` is given by the difference between the current coefficients and the previous coefficients, $delta = currCoeffs - prevCoeffs$

Consider the example below which computes the mel frequency coefficients for the entire speech file. The `'DeltaWindowLength'` value is `2`. The `mfcc` function partitions the speech into 1551 frames. Each row in the `coeffs` matrix corresponds to the log energy value followed by the 13 mel frequency cepstral coefficients for the corresponding segment of the speech file.

```
[audioIn,fs] = audioread('Counting-16-44p1-mono-15secs.wav');
[coeffs,delta,deltaDelta,loc] = mfcc(audioIn,fs);
```

The first row of the delta matrix, `delta(1,:)` is zeros. The second row, `delta(2,:)` equals the difference in coefficients for the current frame, `coeffs(2,:)` and the previous frame, `coeffs(1,:)`.

If `'DeltaWindowLength'` is set to an odd integer greater than `2`, the delta values are given by the following equation:

$$delta = \frac{\sum_{k=-M}^{M} k \cdot coeffs(k,:)}{\sum_{k=-M}^{M} k^2}$$

The function uses a least-square approximation to the local slope over a region around the current time sample. For details, see [1].

Data Types: `single` | `double`

deltaDelta — Change in delta values
matrix | array

Change in `delta` values frome one frame of data to another, returned as an *L*-by-*M* matrix or an *L*-by-*M*-by-*N* array. The `deltaDelta` array is the same size and data type as the `coeffs` and `delta` arrays.

If `'DeltaWindowLength'` is set to `2`, the `deltaDelta` is given by the difference between the current delta values and the previous delta values,

$$deltaDelta = currdelta - prevdelta$$

Consider the example below which computes the mel frequency coefficients for the entire speech file. The `'DeltaWindowLength'` value is `2`.

```
[audioIn,fs] = audioread('Counting-16-44p1-mono-15secs.wav');
[coeffs,delta,deltaDelta,loc] = mfcc(audioIn,fs);
```

The first row of the `deltaDelta` matrix, `deltaDelta(1,:)` is zeros. The second row, `deltaDelta(2,:)` equals the difference in delta values for the current frame, `delta(2,:)` and the previous frame, `delta(1,:)`.

If `'DeltaWindowLength'` is set to an odd integer greater than `2`, the `deltaDelta` values are given by the following equation:

$$deltaDelta = \frac{\sum_{k=-M}^{M} k \cdot delta(k,:)}{\sum_{k=-M}^{M} k^2}$$

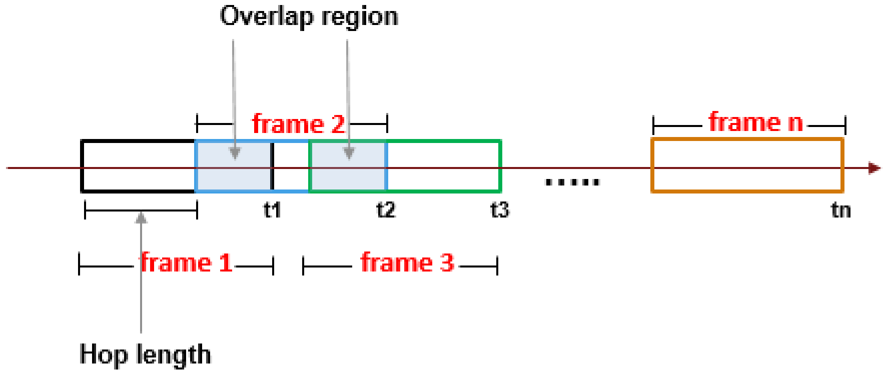
The function a the least-square approximation to the local slope over a region around the current time sample. For details, see [1].

Data Types: `single` | `double`

`loc` — Location of the last sample in each input frame

vector

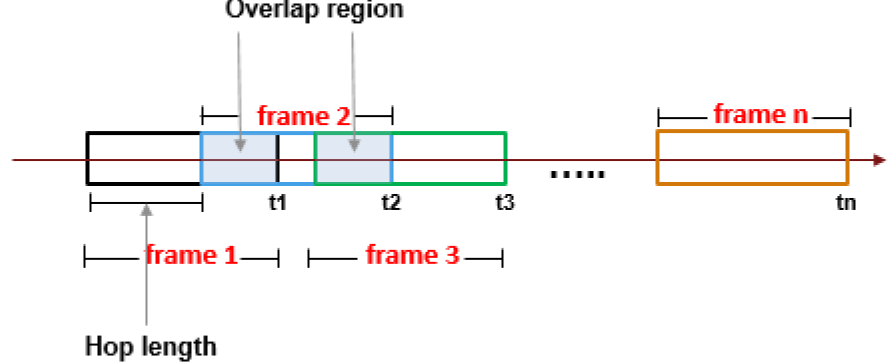
Location of last sample in each input frame, returned as a vector. The `loc` vector is given by the [t_1, t_2, t_3,...,t_n] elements in the following diagram, where *n* corresponds to the number of frames the input is paritioned into, and t_n is the last sample of the last frame.



Data Types: `single` | `double`

Algorithms

The `mfcc` function splits the entire data into overlapping segments. The length of each segment is determined by the `'WindowLength'` argument. The length of overlap between segments is determined by the `'OverlapLength'` argument.



The function computes the mel frequency cepstral coefficients, log energy values, cepstral delta, and the cepstral delta-delta values for each segment as per the algorithm described in `cepstralFeatureExtractor` System object.

References

[1] Rabiner, Lawrence R., and Ronald W. Schafer. *Theory and Applications of Digital Speech Processing*. Upper Saddle River, NJ: Pearson, 2010.

See Also

Functions
`pitch`

System Objects
`cepstralFeatureExtractor` | `voiceActivityDetector`

Blocks
Cepstral Feature Extractor | Voice Activity Detector

Introduced in R2018a

asiosettings

Open settings panel for ASIO driver

Syntax

```
asiosettings
asiosettings(deviceName)
```

Description

`asiosettings` opens the settings panel for the ASIO driver associated with the default audio device.

`asiosettings(deviceName)` opens the settings panel for the ASIO driver associated with the audio device, `deviceName`.

Examples

Open ASIO Settings Panel for Specified Device

Create an audio I/O object, `audioPlayerRecorder`. Call `asiosettings` with the device associated with `audioPlayerRecorder` as the argument.

```
playRec = audioPlayerRecorder;
asiosettings(playRec.Device)
```

Open ASIO Settings Panel for Default Device

Call the `asiosettings` function with no arguments.

```
asiosettings()
```

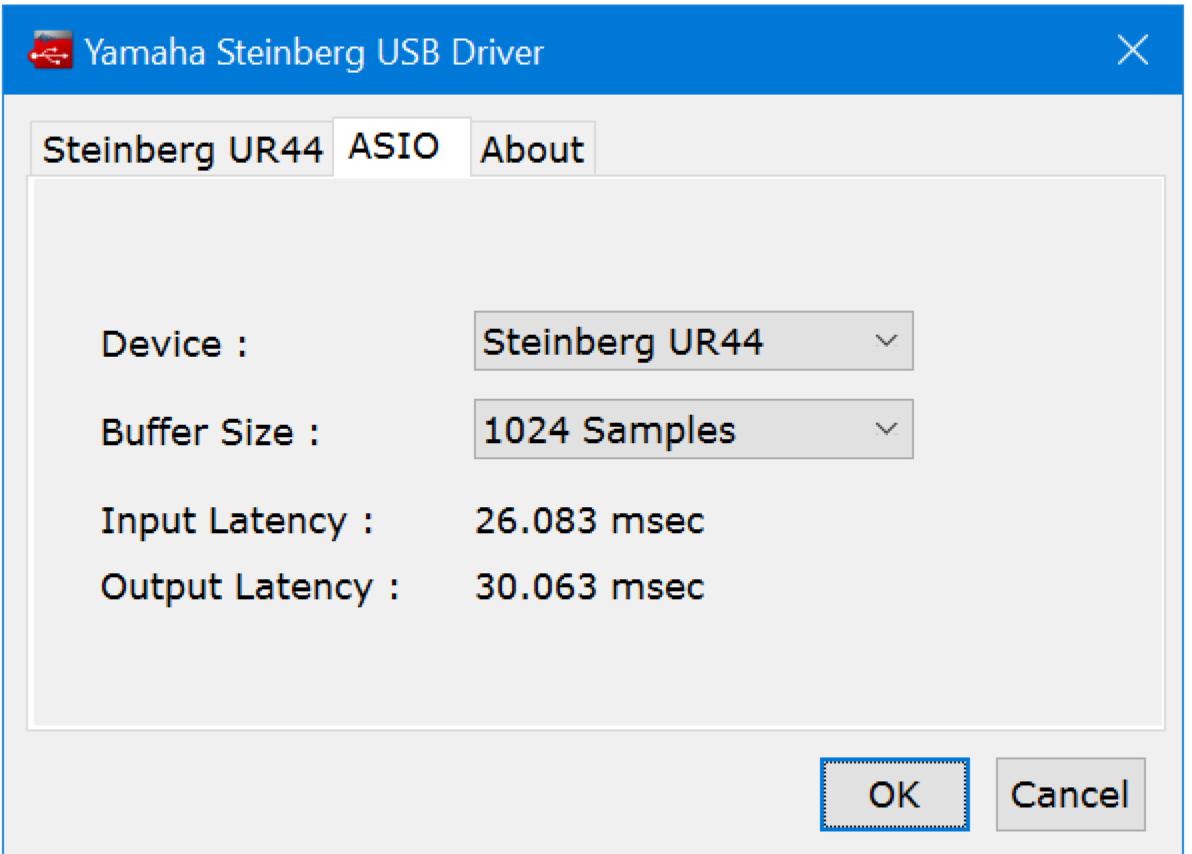
Optimize Latency

To optimize latency when using an ASIO driver, set the buffer size of the ASIO driver to the buffer size of your audio I/O object. In this example, assume the input to your audio device writer is 64 samples per frame. This example requires a Windows machine and an ASIO driver.

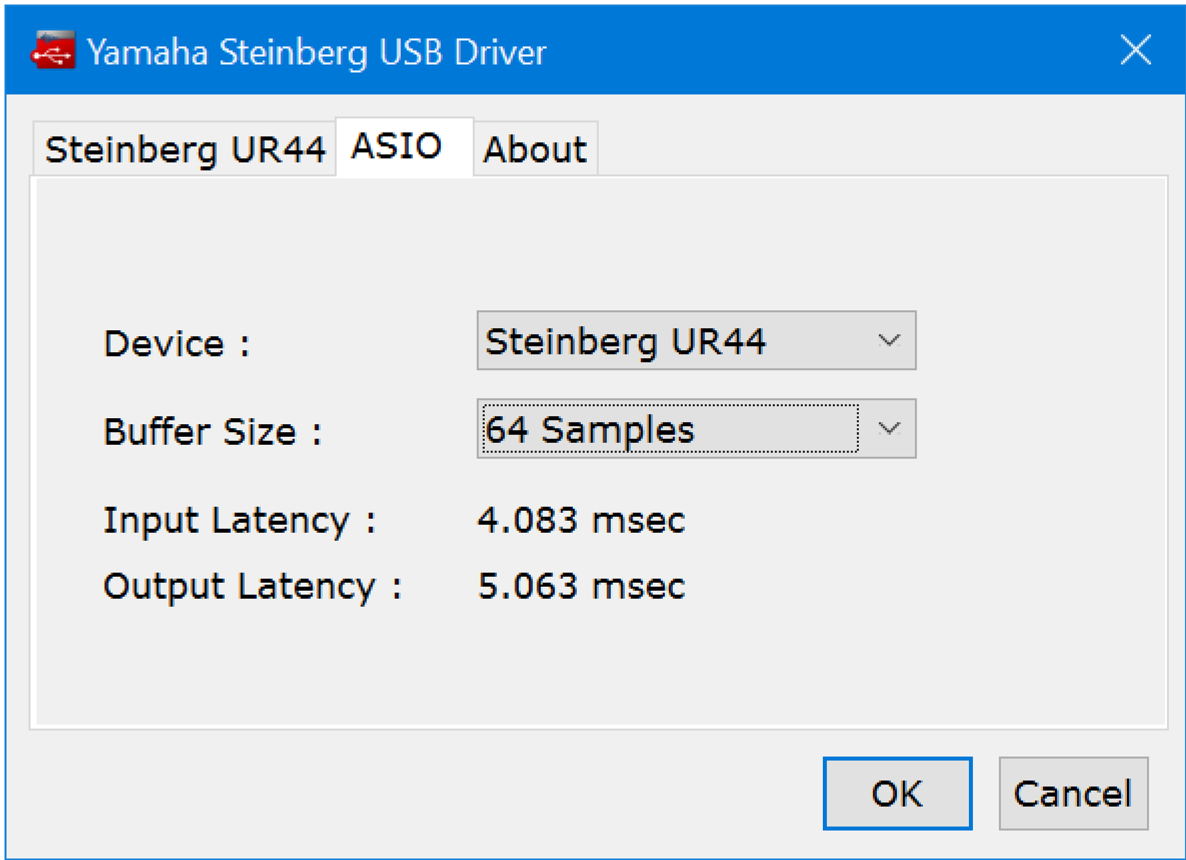
Create an `audioDeviceWriter` System object™. Open the ASIO settings panel for an ASIO-compatible device associated with your device writer.

```
deviceWriter = audioDeviceWriter('Driver','ASIO');
asiosettings(deviceWriter.Device)
```

On the machine in this example, the following dialog opens:



The dialog that opens is specific to your ASIO driver. Set the ASIO buffer size to the desired size, 64.



The latency is now minimized for the frame size of 64 samples. If you want to measure the reduction in latency specific to your system, follow the steps in the Measure Audio Latency example.

Input Arguments

deviceName — Name of ASIO-compatible device

default ASIO-compatible device (default) | character vector | string

Name of ASIO-compatible device, specified as a character vector or string. If `deviceName` is not specified, the default ASIO-compatible device is used.

To view a list of valid ASIO device names on your machine, use `getAudioDevices` on an `audioPlayerRecorder`, `audioDeviceReader('Driver','ASIO')`, or `audioDeviceWriter('Driver','ASIO')` object.

Data Types: `char` | `string`

Tips

- `asiosettings` is compatible only on Windows machines with ASIO drivers. ASIO drivers do not come pre-installed with Windows.
- `asiosettings` returns an error if called with a locked audio device. For example:

```
aDR = audioDeviceReader('Driver','ASIO');
aDR();
asiosettings(aDR.Device)
```

```
Error using audio_asiosettings
PortAudio Error: Device unavailable


Error in asiosettings (line 77)
    audio_asiosettings(ID);
```

See Also

System Objects
`audioDeviceReader` | `audioDeviceWriter` | `audioPlayerRecorder`

Topics
"Audio I/O: Buffering, Latency, and Throughput"

Introduced in R2017b

getAudioDevices

List available audio devices

Syntax

`devices = getAudioDevices(obj)`

Description

`devices = getAudioDevices(obj)` returns a list of audio devices that are available and compatible with your audio I/O object, `obj`.

Examples

List Audio Devices Available to audioDeviceReader

Create an `audioDeviceReader` System object™, and then call `getAudioDevices` on your object.

```
deviceReader = audioDeviceReader;
devices = getAudioDevices(deviceReader)
```

List Audio Devices Available to audioDeviceWriter

Create an `audioDeviceWriter` System object™, and then call `getAudioDevices` on your object.

```
deviceWriter = audioDeviceWriter;
devices = getAudioDevices(deviceWriter)
```

List Audio Devices Available to audioPlayerRecorder

Create an `audioPlayerRecorder` System object™, and then call `getAudioDevices` on your object.

```
playRec = audioPlayerRecorder;
devices = getAudioDevices(playRec)
```

Input Arguments

obj — Audio I/O object
object of `audioDeviceReader` | object of `audioDeviceWriter` | object of `audioPlayerRecorder`

Audio I/O object, specified as an object of `audioDeviceReader`, `audioDeviceWriter`, or `audioPlayerRecorder`.

Data Types: `object`

Output Arguments

devices — List of available and compatible devices
array

List of available and compatible devices.

For `audioDeviceReader` and `audioDeviceWriter`, the list of audio devices depends on the specified `Driver` property of your object.

For `audioPlayerRecorder`, the audio devices listed support full-duplex mode and have a platform-appropriate driver:

- Windows® –– ASIO™
- Mac –– CoreAudio
- Linux® –– ALSA

Data Types: `cell`

See Also

System Objects
`audioDeviceReader` | `audioDeviceWriter` | `audioPlayerRecorder`

Topics
"Audio I/O: Buffering, Latency, and Throughput"

Introduced in R2016a

audioPluginInterface

Specify audio plugin interface

Syntax

```
PluginInterface = audioPluginInterface
PluginInterface = audioPluginInterface(pluginParameters)
PluginInterface = audioPluginInterface(Name,Value)
```

Description

`PluginInterface = audioPluginInterface` returns an object, `PluginInterface`, that specifies the interface of an audio plugin in a digital audio workstation (DAW) environment. It also specifies interface attributes, such as naming for identification.

`PluginInterface = audioPluginInterface(pluginParameters)` specifies audio plugin parameters, which are user-facing variables associated with audio plugin properties. See `audioPluginParameter` for more details.

`PluginInterface = audioPluginInterface(Name,Value)` specifies `audioPluginInterface` properties using one or more `Name,Value` pair arguments.

Examples

Specify Default Audio Plugin Interface

Create a basic audio plugin class definition file.

```
classdef myAudioPlugin < audioPlugin
    methods
        function out = process(~,in)
            out = in;
        end
    end
end
```

Add a constant property, `PluginInterface`, which is specified as an `audioPluginInterface` object.

```
classdef myAudioPlugin < audioPlugin
    properties (Constant)
        PluginInterface = audioPluginInterface;
    end
    methods
        function out = process(~,in)
            out = in;
        end
    end
end
```

Associate Property with Parameter

Create a basic audio plugin class definition file. Specify a property, `Gain`, and a processing function that multiplies input by `Gain`.

```
classdef myAudioPlugin < audioPlugin
    properties
        Gain = 1;
    end
    methods
        function out = process(plugin,in)
            out = in*plugin.Gain;
        end
    end
end
```

Add a constant property, `PluginInterface`, which is specified as an `audioPluginInterface` object.

```
classdef myAudioPlugin < audioPlugin
    properties
        Gain = 1;
    end
    properties (Constant)
        PluginInterface = audioPluginInterface;
    end
    methods
        function out = process(plugin,in)
            out = in*plugin.Gain;
```

```
        end
    end
end
```

Pass `audioPluginParameter` to `audioPluginInterface`. To associate the plugin property, `Gain`, to a plugin parameter, specify the first argument of `audioPluginParameter` as the property name, `'Gain'`.

```
classdef myAudioPlugin < audioPlugin
    properties
        Gain = 1;
    end
    properties (Constant)
        PluginInterface = audioPluginInterface(...
            audioPluginParameter('Gain'));
    end
    methods
        function out = process(plugin,in)
            out = in*plugin.Gain;
        end
    end
end
```

If you generate and deploy `myAudioPlugin` to a digital audio workstation (DAW) environment, the plugin property, `Gain`, synchronizes with a user-facing plugin parameter.

Specify Interface Properties

Create a basic audio plugin class definition file. Specify the plugin name, vendor name, vendor version, unique identification, number of input channels, and number of output channels.

```
classdef monoGain < audioPlugin
    properties
        Gain = 1;
    end
    properties (Constant)
        PluginInterface = audioPluginInterface(...
            audioPluginParameter('Gain'),...
            'PluginName','Simple Gain',...
            'VendorName','Cool Company',...
```

```
            'VendorVersion','1.0.0',...
            'UniqueId','1a1Z',...
            'InputChannels',1,...
            'OutputChannels',1);
    end
    methods
        function out = process(plugin,in)
            out = in*plugin.Gain;
        end
    end
end
```

Input Arguments

pluginParameters — Audio plugin parameters

none (default) | one or more `audioPluginParameter` objects

Audio plugin parameters, specified as one or more `audioPluginParameter` objects.

To create an audio plugin parameter, use the `audioPluginParameter` function. In a digital audio workstation (DAW) environment, they synchronize plugin class properties with user-facing parameters.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'PluginName','cool effect','VendorVersion','1.0.2'` specifies the name of the generated audio plugin as `'cool effect'` and the vendor version as `'1.0.2'`.

PluginName — Name of generated plugin

name of plugin class (default) | character vector

Name of your generated plugin, as seen by a host audio application, specified as a comma-separated pair consisting of `'PluginName'` and a string of up to 127 characters. If `'PluginName'` is not specified, the generated plugin is given the name of the audio plugin class it is generated from.

`VendorName` — Vendor name of the plugin creator
' ' (default) | character vector

Vendor name of the plugin creator, specified as the comma-separated pair `'VendorName'` and a character vector of up to 127 characters.

`VendorVersion` — Vendor version
'1.0.0' (default) | dot-separated character vector

Vendor version used to track plugin releases, specified as a comma-separated pair consisting of `'VendorVersion'` and a dot-separated string of 1–3 integers in the range 0 to 9.

Example: `'1'`

Example: `'1.4'`

Example: `'1.3.5'`

`UniqueId` — Unique identifier of plugin
'MWap' (default) | four-element character vector

Unique identifier for your plugin, specified as a comma-separated pair consisting of `'UniqueID'` and a four-character string, used for recognition in certain digital audio workstation (DAW) environments.

`InputChannels` — Input channels
2 (default) | integer | vector of integers

Input channels, specified as a comma-separated pair consisting of `'InputChannels'` and an integer or vector of integers. The input channels are the number of input data arguments and associated channels (columns) passed to the processing function of your audio plugin.

Example: `'InputChannels',3` calls the processing function with one data argument containing 3 channels.

Example: `'InputChannels',[2,4,1,5]` calls the processing function with 4 data arguments. The first argument contains 2 channels, the second contains 4 channels, the third contains 1 channel, and the fourth contains 5 channels.

Note This property is not applicable for audio source plugins, and must be omitted.

`OutputChannels` — Output channels
2 (default) | integer | vector of integers

Output channels, specified a comma-separated pair consisting of `'OutputChannels'` and an integer or vector of integers. The output channels are the number of input data arguments and associated channels (columns) passed from the processing function of your audio plugin.

Example: `'OutputChannels',3` specifies the processing function to output one data argument containing 3 channels.

Example: `'OutputChannels',[2,4,1,5]` specifies the processing function to output 4 data arguments. The first argument contains 2 channels, the second contains 4 channels, the third contains 1 channel, and the fourth contains 5 channels.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

Classes
`audioPlugin` | `audioPluginSource`

Functions
`audioPluginParameter` | `generateAudioPlugin` | `validateAudioPlugin`

Topics

"Design an Audio Plugin"

Introduced in R2016a

audioPluginParameter

Specify audio plugin parameters

Syntax

```
pluginParameter = audioPluginParameter(propertyName)
pluginParameter = audioPluginParameter(propertyName,Name,Value)
```

Description

`pluginParameter = audioPluginParameter(propertyName)` returns an object, `pluginParameter`, that associates an audio plugin parameter to the audio plugin property specified by `propertyName`. Use the plugin parameter object, `pluginParameter`, as an argument to an `audioPluginInterface` function in your plugin class definition.

In a digital audio workstation (DAW) environment, or when using **Audio Test Bench** in the MATLAB environment, plugin parameters are tunable, user-facing variables with defined ranges mapped to controls. When you modify a parameter value using a control, the associated plugin property is also modified. If the audio processing algorithm of the plugin depends on properties, the algorithm is also modified.

To visualize the relationship between plugin properties, parameters, and the environment in which a plugin is run, see “Implementation of Audio Plugin Parameters” on page 2-61.

`pluginParameter = audioPluginParameter(propertyName,Name,Value)` specifies `audioPluginParameter` properties using one or more `Name,Value` pair arguments.

Examples

Associate Property with Parameter

Create a basic audio plugin class definition file. Specify a property, `Gain`, and a processing function that multiplies input by `Gain`.

```
classdef myAudioPlugin < audioPlugin
    properties
        Gain = 1;
    end
    methods
        function out = process(plugin,in)
            out = in*plugin.Gain;
        end
    end
end
```

Add a constant property, `PluginInterface`, which is specified as an `audioPluginInterface` object.

```
classdef myAudioPlugin < audioPlugin
    properties
        Gain = 1;
    end
    properties (Constant)
        PluginInterface = audioPluginInterface;
    end
    methods
        function out = process(plugin,in)
            out = in*plugin.Gain;
        end
    end
end
```

Pass `audioPluginParameter` to `audioPluginInterface`. To associate the plugin property, `Gain`, to a plugin parameter, specify the first argument of `audioPluginParameter` as the property name, `'Gain'`.

```
classdef myAudioPlugin < audioPlugin
    properties
        Gain = 1;
    end
    properties (Constant)
        PluginInterface = audioPluginInterface(...
            audioPluginParameter('Gain'));
    end
```

```
    methods
        function out = process(plugin,in)
            out = in*plugin.Gain;
        end
    end
end
```

Specify Parameter Information

Create a basic plugin class definition file. Specify `'DisplayName'` as `'Awesome Gain'`, `'Label'` as `'linear'`, and `'Mapping'` as `{'lin',0,20}`.

```
classdef myAudioPlugin < audioPlugin
    properties
        Gain = 1;
    end
    properties (Constant)
        PluginInterface = audioPluginInterface(...
            audioPluginParameter('Gain',...
            'DisplayName', 'Awesome Gain',...
            'Label', 'linear',...
            'Mapping', {'lin',0,20}));
    end
    methods
        function out = process(plugin,in)
            out = in*plugin.Gain;
        end
    end
end
```

Integer Parameter Mapping

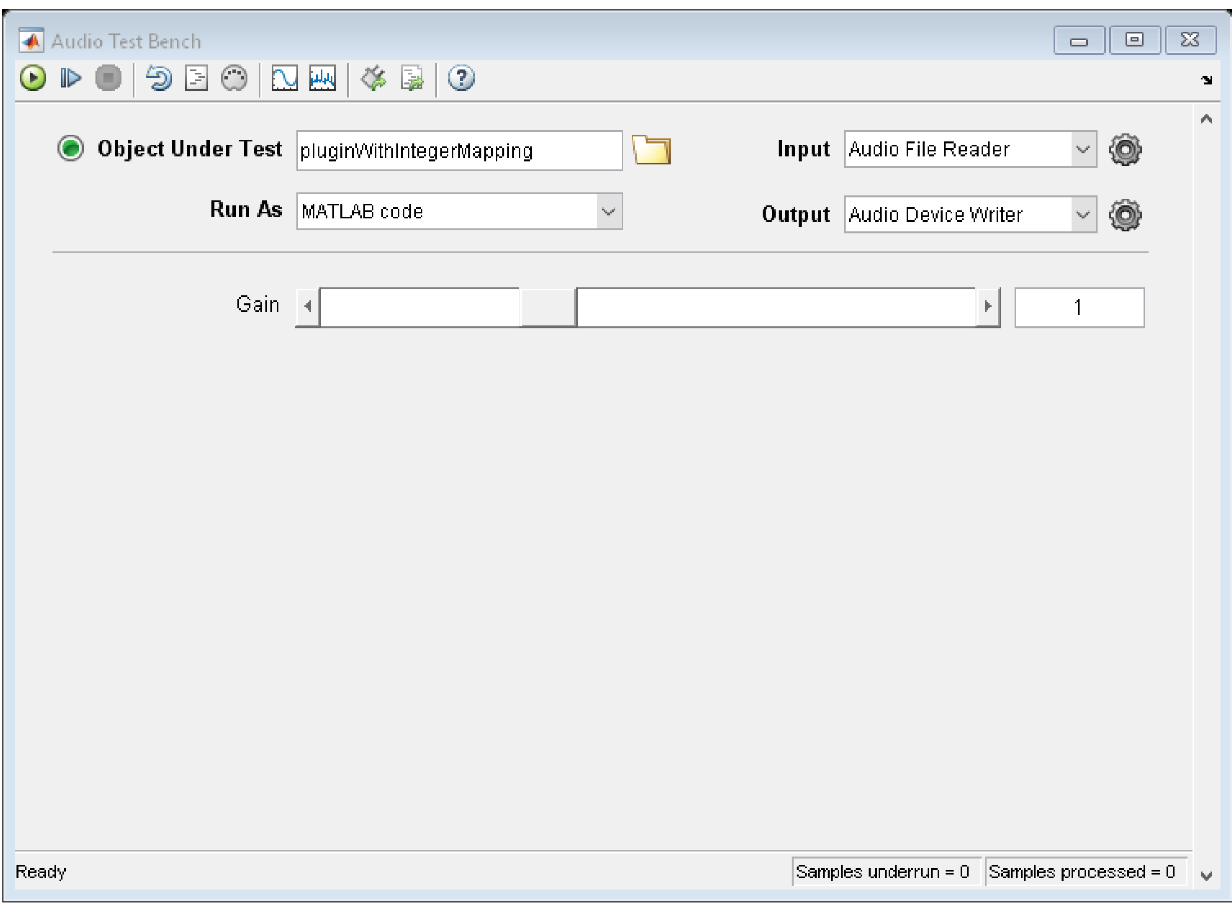
The following class definition uses integer parameter mapping to define the relationship between a property and a parameter. You can use the plugin created from this class to tune the linear gain of an audio signal in integer steps from 0 to 3.

```
classdef pluginWithIntegerMapping < audioPlugin
    properties
        Gain = 1;
    end
```

```
    properties (Constant)
        PluginInterface = audioPluginInterface( ...
            audioPluginParameter('Gain', ...
            'Mapping', {'int',0,3}));
    end
    methods
        function out = process(plugin,in)
            out = in*plugin.Gain;
        end
    end
end
```

To run the plugin, save the class definition to a local folder and then call the Audio Test Bench.

```
audioTestBench(pluginWithIntegerMapping)
```



Power Parameter Mapping

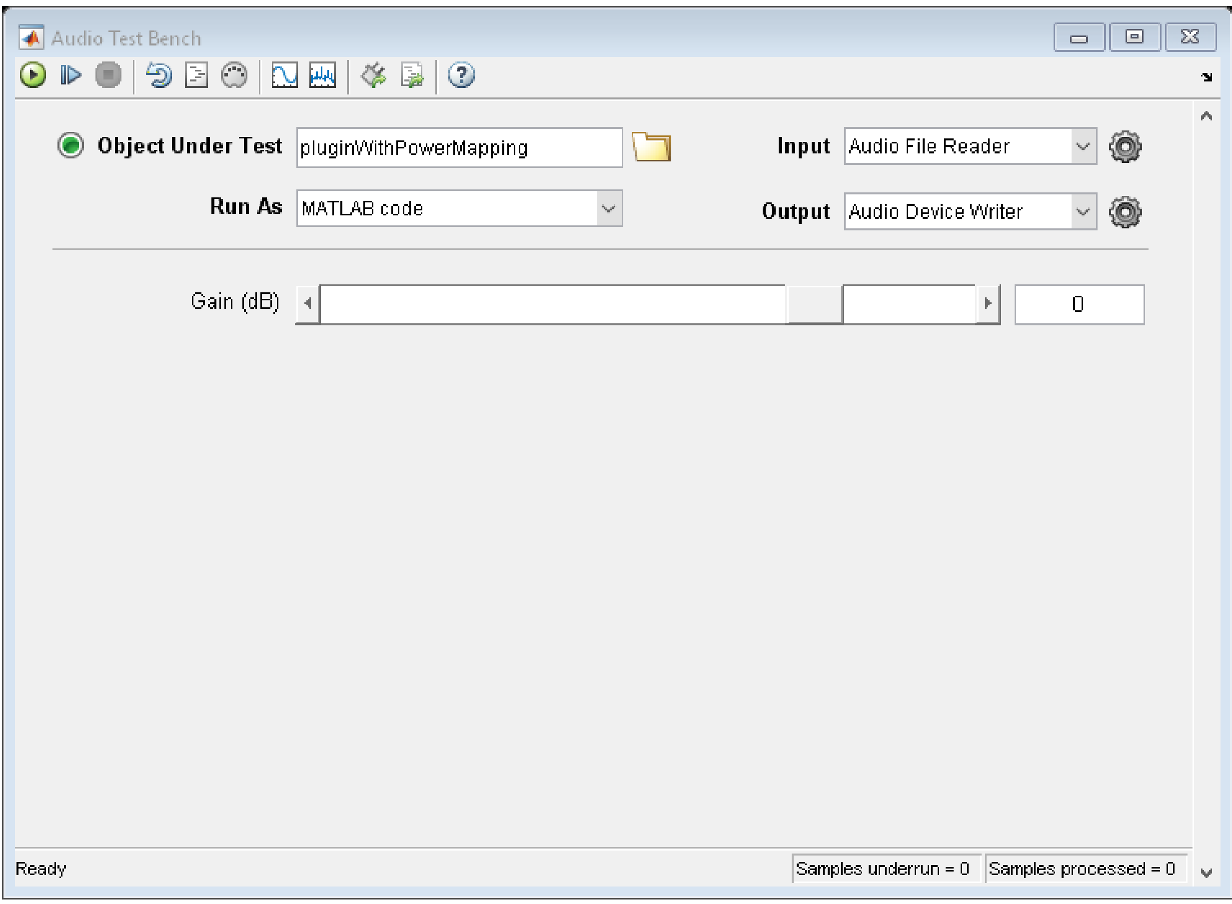
The following class definition uses power parameter mapping to define the relationship between a property and a parameter. You can use the plugin created from this class to tune the gain of an audio signal in dB.

```
classdef pluginWithPowerMapping < audioPlugin
    properties
        Gain = 0;
```

```
    end
    properties (Constant)
        PluginInterface = audioPluginInterface( ...
            audioPluginParameter('Gain', ...
            'Label', 'dB', ...
            'Mapping', {'pow', 1/3, -140, 12}));
    end
    methods
        function out = process(plugin,in)
            dBGain = 10^(plugin.Gain/20);
            out = in*dBGain;
        end
    end
end
```

To run the plugin, save the class definition to a local folder and then call the Audio Test Bench.

```
audioTestBench(pluginWithPowerMapping)
```



Logarithmic Parameter Mapping

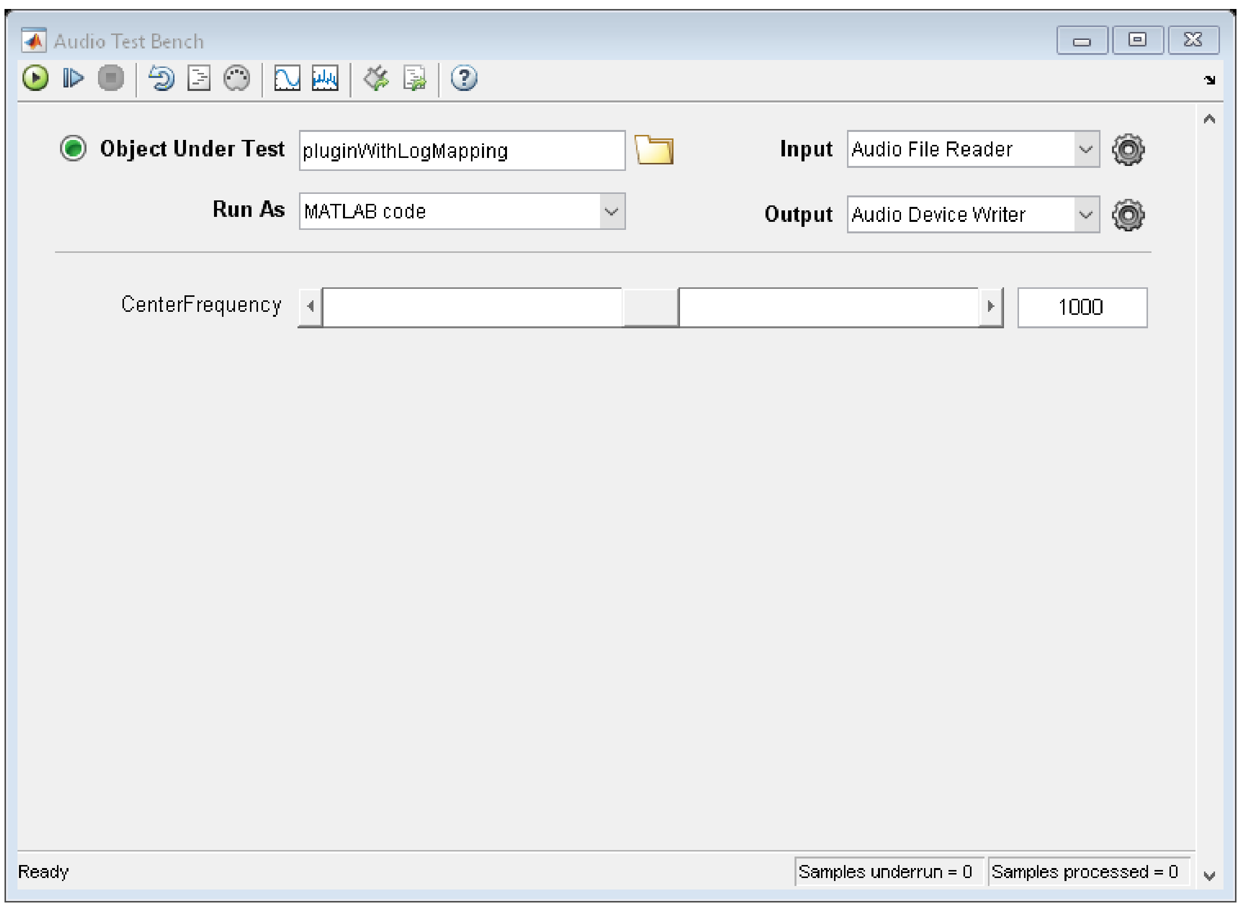
The following class definition uses logarithmic parameter mapping to define the relationship between a property and a parameter. You can use the plugin created from this class to tune the center frequency of a single-band EQ filter from 100 to 10000.

```
classdef pluginWithLogMapping < audioPlugin
    properties
        EQ
```

```
        CenterFrequency = 1000;
    end
    properties (Constant)
        PluginInterface = audioPluginInterface( ...
            audioPluginParameter('CenterFrequency', ...
            'Mapping', {'log',100,10000}));
    end
    methods
        function plugin = pluginWithLogMapping
            plugin.EQ = multibandParametricEQ('NumEQBands',1, ...
                'PeakGains',20, ...
                'Frequencies',plugin.CenterFrequency);
        end
        function out = process(plugin,in)
            out = plugin.EQ(in);
        end
        function set.CenterFrequency(plugin,val)
            plugin.CenterFrequency = val;
            plugin.EQ.Frequencies = val;
        end
        function reset(plugin)
            plugin.EQ.SampleRate = getSampleRate(plugin);
        end
    end
end
```

To run the plugin, save the class definition to a local folder and then call the Audio Test Bench.

```
audioTestBench(pluginWithLogMapping)
```



Enumeration for Logical Properties Parameter Mapping

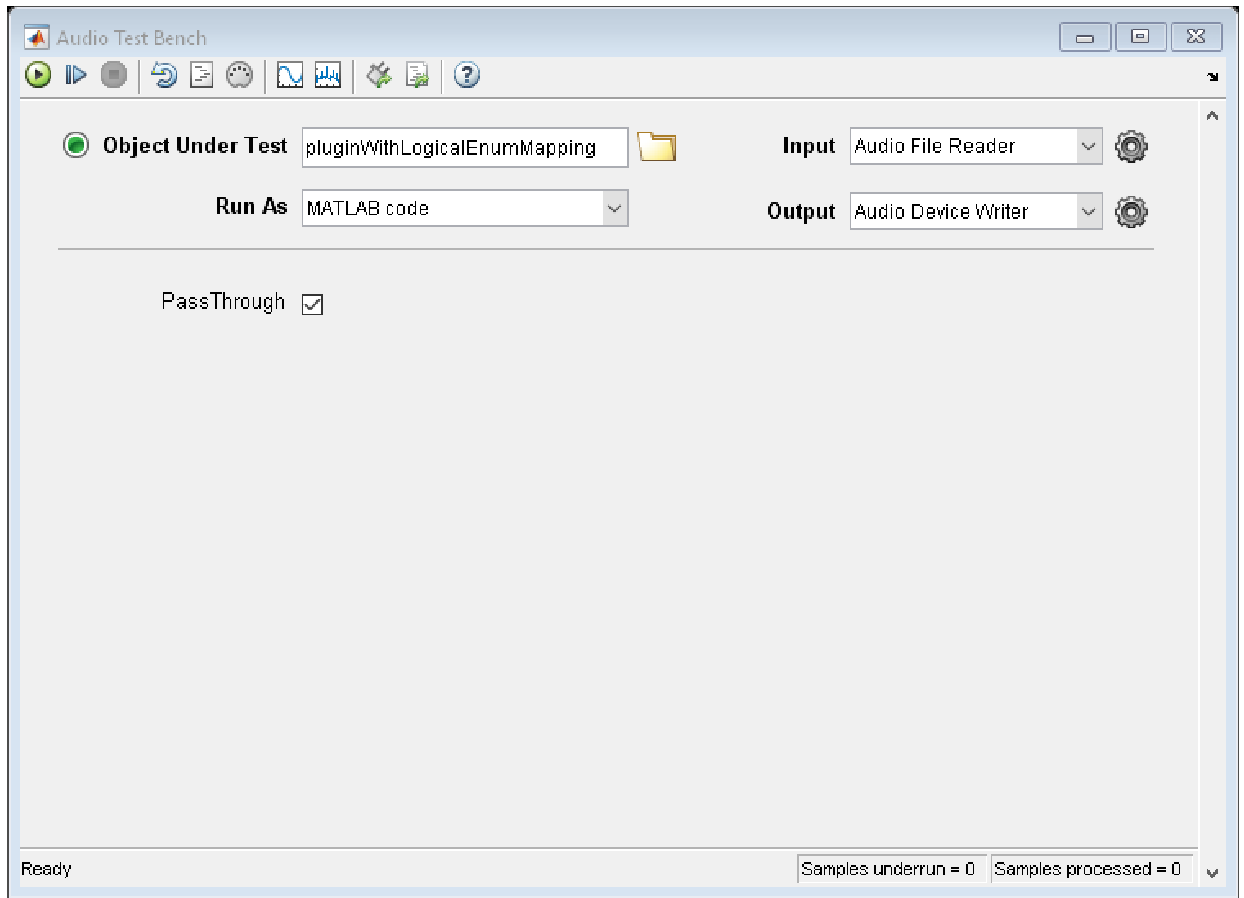
The following class definition uses enumeration parameter mapping to define the relationship between a property and a parameter. You can use the plugin created from this class to block or pass through the audio signal by tuning the `PassThrough` parameter.

```
classdef pluginWithLogicalEnumMapping < audioPlugin
    properties
```

```
            PassThrough = true;
        end
        properties (Constant)
            PluginInterface = audioPluginInterface( ...
                audioPluginParameter('PassThrough', ...
                'Mapping', {'enum','Block signal','Pass through'}));
        end
        methods
            function out = process(plugin,in)
                if plugin.PassThrough
                    out = in;
                else
                    out = zeros(size(in));
                end
            end
        end
    end
```

To run the plugin, save the class definition to a local folder and then call the Audio Test Bench.

```
audioTestBench(pluginWithLogicalEnumMapping)
```



'enum' for Enumeration Class Parameter Mapping

The following class definitions comprise a simple example of enumeration parameter mapping for properties defined by an enumeration class. You can specify the operating mode of the plugin created from this class by tuning the `Mode` parameter.

Plugin Class Definition

```
classdef pluginWithEnumMapping < audioPlugin
```

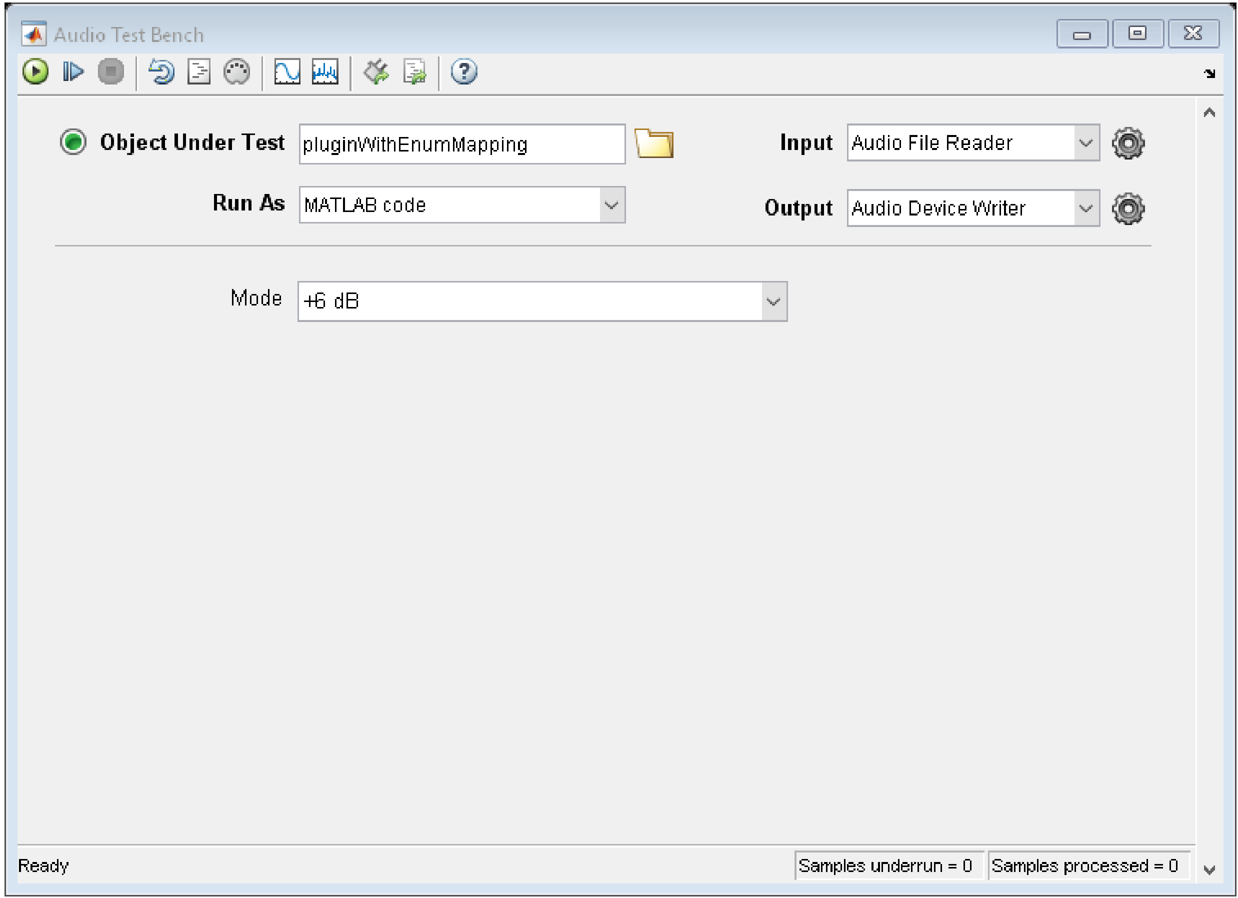
```
    properties
        Mode = OperatingMode.boost;
    end
    properties (Constant)
        PluginInterface = audioPluginInterface(...
            audioPluginParameter('Mode',...
                'Mapping',{'enum','+6 dB','-6 dB','silence','white noise'}));
    end
    methods
        function out = process(plugin,in)
            switch (plugin.Mode)
                case OperatingMode.boost
                    out = in * 2;
                case OperatingMode.cut
                    out = in / 2;
                case OperatingMode.mute
                    out = zeros(size(in));
                case OperatingMode.noise
                    out = rand(size(in)) - 0.5;
                otherwise
                    out = in;
            end
        end
    end
end
```

Enumeration Class Definition

```
classdef OperatingMode < int8
    enumeration
        boost (0)
        cut   (1)
        mute  (2)
        noise (3)
    end
end
```

To run the plugin, save the plugin and enumeration class definition files to a local folder. Then call the Audio Test Bench on the plugin class.

```
audioTestBench(pluginWithEnumMapping)
```



Input Arguments

`propertyName` — Name of audio plugin property
character vector

Name of the audio plugin property that you want to associate with a parameter, specified as a character vector. Enter the property name exactly as it is defined in the property section of your audio plugin class.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'DisplayName','Gain','Label','dB'` specifies the display name of your parameter as `'Gain'` and the display label for parameter value units as `'dB'`.

`DisplayName` — Display name of parameter

associated property name (default) | character vector

Display name of your parameter, specified as a comma-separated pair consisting of `'DisplayName'` and a character vector. If `'DisplayName'` is not specified, the name of the associated property is used.

The display name of your parameter is used in a digital audio workstation (DAW) environment, and when using **Audio Test Bench** in the MATLAB environment.

`Label` — Display label for parameter value units

`''` (default) | character vector

Display label for parameter value units, specified as a comma-separated pair consisting of `'Label'` and a character vector.

The display label for parameter value units is used in a digital audio workstation (DAW) environment, and when using **Audio Test Bench** in the MATLAB environment.

The `'Label'` name-value pair is ignored for nonnumeric parameters.

`Mapping` — Mapping between property and parameter range

cell array

Mapping between property and parameter range, specified as the comma-separated pair consisting of `'Mapping'` and a cell array.

Parameter range mapping specifies a mapping between a property and the associated parameter range.

The first element of the cell array is a character vector specifying the kind of mapping. The valid values are `'lin'`, `'log'`, `'pow'`, `'int'`, and `'enum'`. The subsequent

elements of the cell array depend on the kind of mapping. The valid mappings depend on the property data type.

Property Data Type	Valid Mappings	Default
double	`'lin'`, `'log'`, `'pow'`, `'int'`	`{'lin', 0, 1}`
logical	`'enum'`	`{'enum', 'off', 'on'}`
enumeration class	`'enum'`	enumeration names

Mapping	Description	Example
`'lin'`	Specifies a linear relationship with given minimum and maximum values. $(property\ value) = \min + (\max - \min) \times (parameter\ value)$	`{'lin', 0, 24}` specifies a linear relationship with a minimum of 0 and maximum of 24. **Simple Example:** "Specify Parameter Information" on page 2-47
`'log'`	Specifies a logarithmic relationship with given minimum and maximum values, where the control position maps to the logarithm of the property value. The minimum value must be greater than `0`. $(property\ value) = \min \times (\max/\min)^{(parameter\ value)}$	`{'log', 1, 22050}` specifies a logarithmic relationship with a minimum of 1 and a maximum of 22050. **Simple Example:** "Logarithmic Parameter Mapping" on page 2-51
`'pow'`	Specifies a power law relationship with given exponent, minimum, and maximum values. The property value is related to the control position raised to the exponent: $(property\ value) = \min + (\max - \min) \times (parameter\ value)^{\exp}$	`{'pow', 1/3, -140, 12}` specifies a power law relationship with an exponent of 1/3, a minimum of –140, and a maximum of 12. **Simple Example:** "Power Parameter Mapping" on page 2-49

Mapping	Description	Example
`'int'`	Quantizes the control position and maps it to the range of consecutive integers with given minimum and maximum values. $(\mathit{property\ value}) = \mathit{floor}\left(0.5 + \min + (\max - \min) \times (\mathit{parameter\ value})\right)$	`{'int', 0, 3}` specifies a linear, quantized relationship with a minimum of 0 and maximum of 3. The property value is mapped as an integer in the range 0 to 3. **Simple Example:** "Integer Parameter Mapping" on page 2-47
`'enum'` (logical)	Optionally provides character vectors for display on the plugin dialog box.	`{'enum','Block signal','Passthrough'}` specifies the character vector `'Block signal'` if the parameter value is `false` and `'Passthrough'` if the parameter value is `true`. **Simple Example:** "Enumeration for Logical Properties Parameter Mapping" on page 2-53
`'enum'` (enumeration class)	Optionally provides character vectors for the members of the enumeration class.	`{'enum', '+6 dB', '-6 dB', 'silence', 'white noise'}` specifies the character vectors `'+6 dB'`, `'-6 dB'`, `'silence'`, and `'white noise'`. **Simple Example:** "'enum' for Enumeration Class Parameter Mapping" on page 2-55

For nontrivial examples of audio plugin parameter mapping, see "Audio Plugin Example Gallery".

Definitions

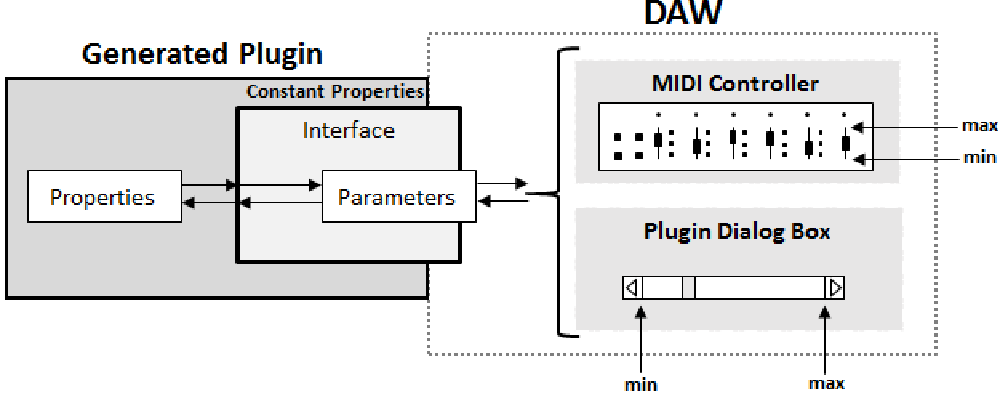
Implementation of Audio Plugin Parameters

Audio plugin parameters are visible and tunable in both the MATLAB and digital audio workstation (DAW) environments.

MATLAB Environment. Use **Audio Test Bench** to interact with plugin parameters in the MATLAB environment.

MATLAB
Plugin
Constant Properties
Interface
Properties
Parameters
Slider for property data type double
Check box for property data type logical
Drop-down menu for property data type enumeration class
Display name of parameter
min
max
Display label for parameter value units
Parameter value

DAW Environment. Use `generateAudioPlugin` to deploy your audio plugin to a DAW environment. The DAW environment determines the exact layout of plugin parameters as seen by the plugin user.



Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

Classes
audioPlugin | audioPluginSource

Functions
audioPluginInterface | generateAudioPlugin | validateAudioPlugin

Topics

"Design an Audio Plugin"

Introduced in R2016a

configureMIDI

Configure MIDI connections between audio object and MIDI controller

Syntax

```
configureMIDI(audioObject)
configureMIDI(audioObject,propertyName)
configureMIDI(audioObject,propertyName,controlNumber)
configureMIDI(audioObject,propertyName,controlNumber,'DeviceName',
deviceNameValue)
```

Description

`configureMIDI(audioObject)` opens a MIDI configuration user interface (UI). Use the UI to synchronize parameters of the plugin, `audioObject`, to MIDI controls on your default MIDI device. You can also generate MATLAB code corresponding to the MIDI configuration developed using the `configureMIDI` UI.

To set your default device, type this syntax in the command line:

```
setpref midi DefaultDevice deviceNameValue
```

`deviceNameValue` is the MIDI device name, assigned by the device manufacturer or host operating system. Use `midiid` to get the device name corresponding to your MIDI device.

`configureMIDI(audioObject,propertyName)` makes the property, `propertyName`, respond to any control on the default MIDI device.

`configureMIDI(audioObject,propertyName,controlNumber)` makes the property respond to the MIDI control specified by `controlNumber`.

`configureMIDI(audioObject,propertyName,controlNumber,'DeviceName',
deviceNameValue)` makes the property respond to the MIDI control specified by `controlNumber` on the device specified by `deviceNameValue`.

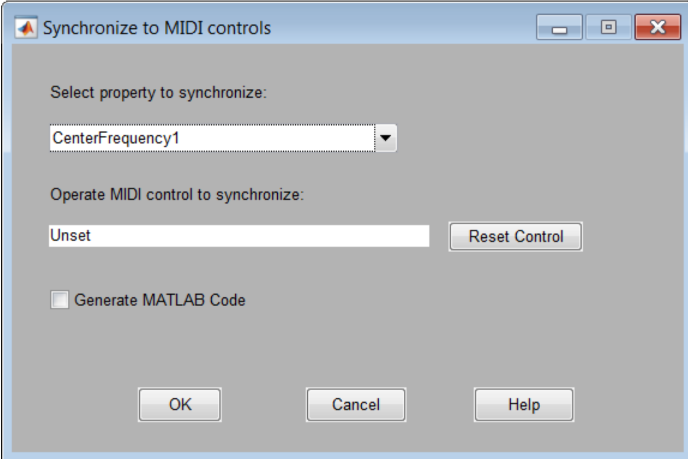
Examples

Synchronize Plugin Parameters to MIDI Controls

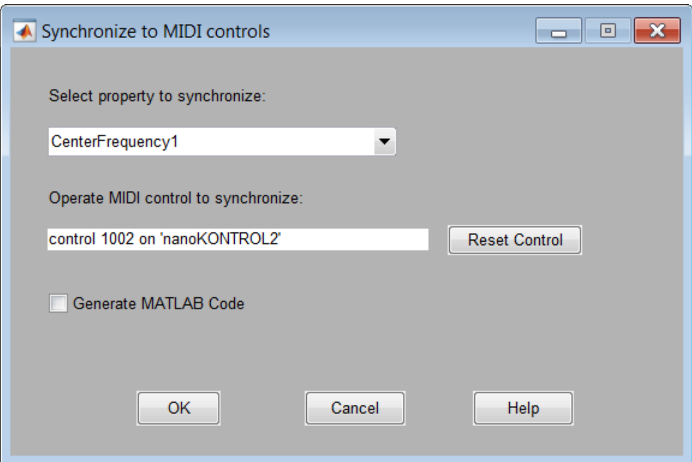
1 Open the MIDI configuration UI for a parametric equalizer plugin object.

```
parametricEQPlugin = audiopluginexample.ParametricEqualizer;
configureMIDI(parametricEQPlugin);
```

2 In the UI, select a property to synchronize with your default MIDI device.



3 On your MIDI device, operate the control that you want to synchronize to the selected plugin property. The control appears in the **Operate MIDI control to synchronize** box.



4. Repeat steps 2 and 3 as needed to synchronize multiple properties to multiple MIDI controls.

   To disconnect the property and control currently displayed on your `configureMIDI` UI, click **Reset Control** at any time.
5. Click **OK**.

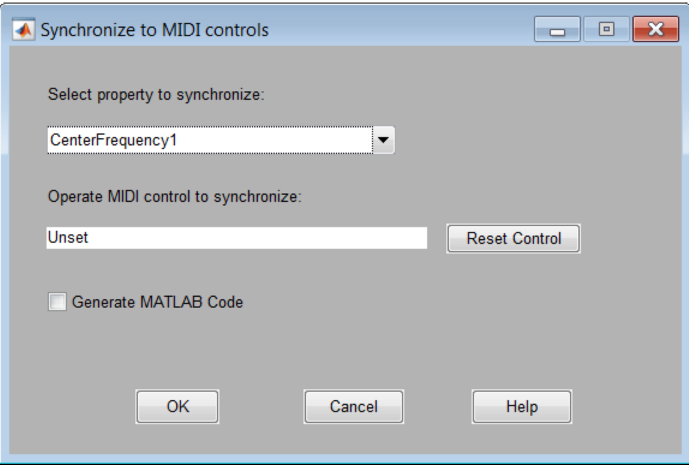
   The specified MIDI controls and properties and now synchronized.

Generate MATLAB Code from `configureMIDI` UI

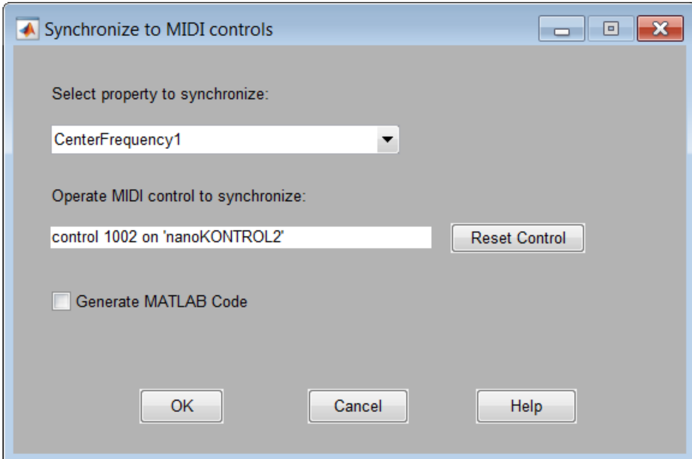
Generate MATLAB code corresponding to the MIDI configuration developed using the `configureMIDI` UI. You can embed the MATLAB code in your simulation so that you do not need to reopen the UI to restore your chosen MIDI connections.

1. Open the MIDI configuration UI for a parametric equalizer plugin object.

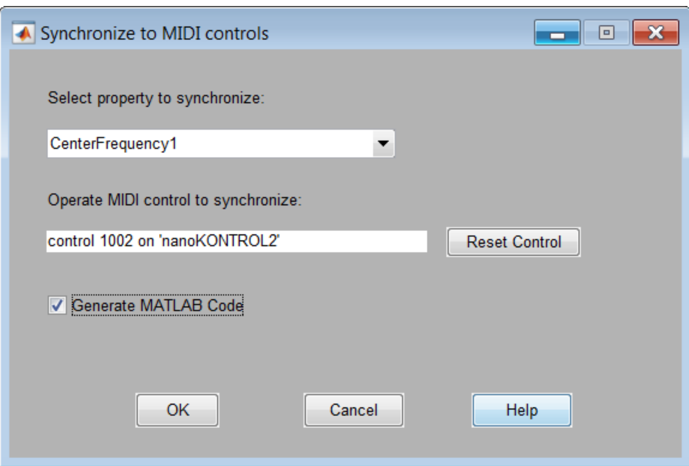
   ```
   parametricEQPlugin = audiopluginexample.ParametricEqualizer;
   configureMIDI(parametricEQPlugin);
   ```
2. In the UI, select a property to synchronize with your default MIDI device.



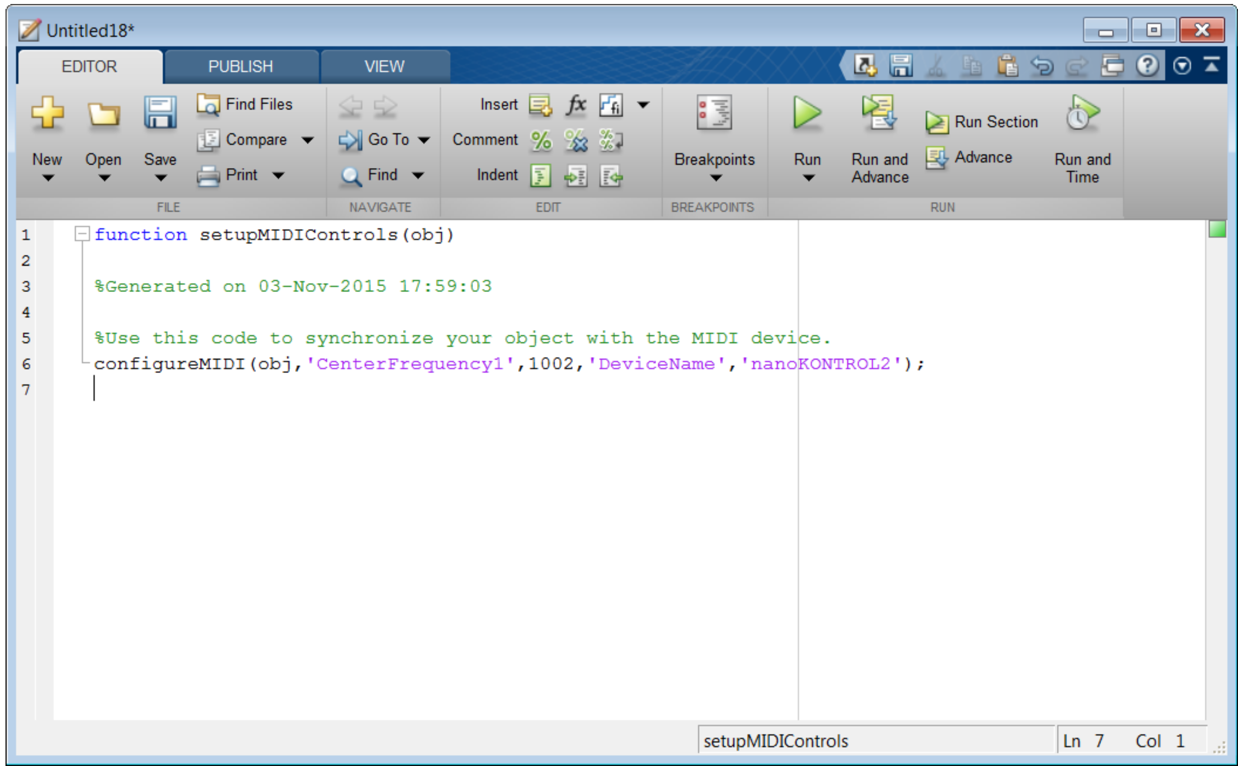
3 On your MIDI device, operate the control that you want to synchronize to the selected plugin property. The control appears in the **Operate MIDI control to synchronize** box.



4 Select the **Generate MATLAB Code** check box.



5. Click **OK**. The generated MATLAB code corresponds to the MIDI configuration that you developed.



Make Plugin Property Respond to Any MIDI Control

Make a plugin property respond to any control on your default MIDI device.

```
parametricEQPlugin = audiopluginexample.ParametricEqualizer;
configureMIDI(parametricEQPlugin,'CenterFrequency1');
```

Make Plugin Property Respond to Specific MIDI Control on Default MIDI Device

Make a plugin property respond to a specific MIDI control on your default MIDI device.

Create an object of the audio plugin example `audiopluginexample.ParametricEqualizer`.

```
parametricEQPlugin = audiopluginexample.ParametricEqualizer;
```

Use `midiid` to identify a MIDI control to synchronize with your property.

```
[controlNumber,device] = midiid
```

```
Move the control you wish to identify; type ^C to abort.
Waiting for control message... done

controlNumber =

        1003


device =

nanoKONTROL2
```

Use `configureMIDI` to synchronize your chosen MIDI control, specified by `controlNumber`, with a property.

```
configureMIDI(parametricEQPlugin,'CenterFrequency1',controlNumber);
```

Make Plugin Property Respond to Specific MIDI Control on a Specific MIDI Device

Make a plugin property respond to any control on your default MIDI device.

Create an object of the audio plugin example, `audiopluginexample.ParametricEqualizer`.

```
parametricEQPlugin = audiopluginexample.ParametricEqualizer;
```

Use `midiid` to identify a specific MIDI control on a specific MIDI device.

```
[controlNumber,device] = midiid
```

```
Move the control you wish to identify; type ^C to abort.
Waiting for control message... done

controlNumber =

        1003
```

```
device =

nanoKONTROL2
```

Use `configureMIDI` to synchronize a property with your chosen MIDI control, specified by `controlNumber`, on your chosen MIDI device, specified by `device`.

```
configureMIDI(parametricEQPlugin,'CenterFrequency1',controlNumber,'DeviceName',device)
```

Input Arguments

`audioObject` — Audio object

object

Audio plugin or compatible System object, specified as an object that inherits from the `audioPlugin` class or an object of a compatible Audio System Toolbox System object.

`propertyName` — Name of object property

character vector

Name of the object property, specified as a character vector. Enter the property name exactly as it is defined in the property section of your audio plugin or Audio System Toolbox System object.

`controlNumber` — MIDI device control number

integer values

MIDI device control number, specified as an integer. The value is assigned to the control by the device manufacturer. It is used for identification purposes.

`deviceNameValue` — MIDI device name

character vector

MIDI device name, assigned by the device manufacturer or host operating system, specified as a character vector. If you do not specify a MIDI device name, the default MIDI device is used.

Limitations

For MIDI connections established by `configureMIDI`, moving a MIDI control sends a callback to update the associated property values. To synchronize your MIDI device in an

audio stream loop, you might need to use the `drawnow` command for the callback to process immediately. For efficiency, use the `drawnow limitrate` syntax.

For example, to synchronize your MIDI device and audio object, uncomment the `drawnow limitrate` command from this code:

```
fileReader = dsp.AudioFileReader(...
    'Filename','RockDrums-44p1-stereo-11secs.mp3');
deviceWriter = audioDeviceWriter;
dRC = compressor;

configureMIDI(compressor,'Threshold');

while ~isDone(fileReader)
    input = fileReader();
    output = dRC(input);
    deviceWriter(output);
%     drawnow limitrate;
end

release(fileReader);
release(deviceWriter);
```

If your audio stream loop includes visualizing data on a scope, such as `dsp.SpectrumAnalyzer`, `dsp.TimeScope`, or `dsp.ArrayPlot`, the `drawnow` command is not required.

See Also

Classes
`audioPlugin` | `audioPluginSource`

Functions
`disconnectMIDI` | `getMIDIConnections` | `midicallback` | `midicontrols` | `midiid` | `midiread` | `midisync`

Topics
"MIDI Control for Audio Plugins"
"MIDI Control Surface Interface"

Introduced in R2016a

designParamEQ

Design parametric equalizer

Syntax

```
[B,A] = designParamEQ(N,gain,centerFreq,bandwidth)
[B,A] = designParamEQ(N,gain,centerFreq,bandwidth,mode)
```

Description

`[B,A] = designParamEQ(N,gain,centerFreq,bandwidth)` designs an `N`th-order parametric equalizer with specified gain, center frequency, and bandwidth. `B` and `A` are matrices of numerator and denominator coefficients, with columns corresponding to cascaded second-order section (SOS) filters.

`[B,A] = designParamEQ(N,gain,centerFreq,bandwidth,mode)` specifies whether the parametric equalizer is implemented with second-order sections or fourth-order sections (FOS).

Examples

Design Two-Band Parametric Equalizer

Specify the filter order, peak gain in dB, normalized center frequencies, and normalized bandwidth of the bands of your parametric equalizer.

```
N = [2,4];
gain = [6,-4];
centerFreq = [0.25,0.75];
bandwidth = [0.12,0.10];
```

Generate the filter coefficients using the specified parameters.

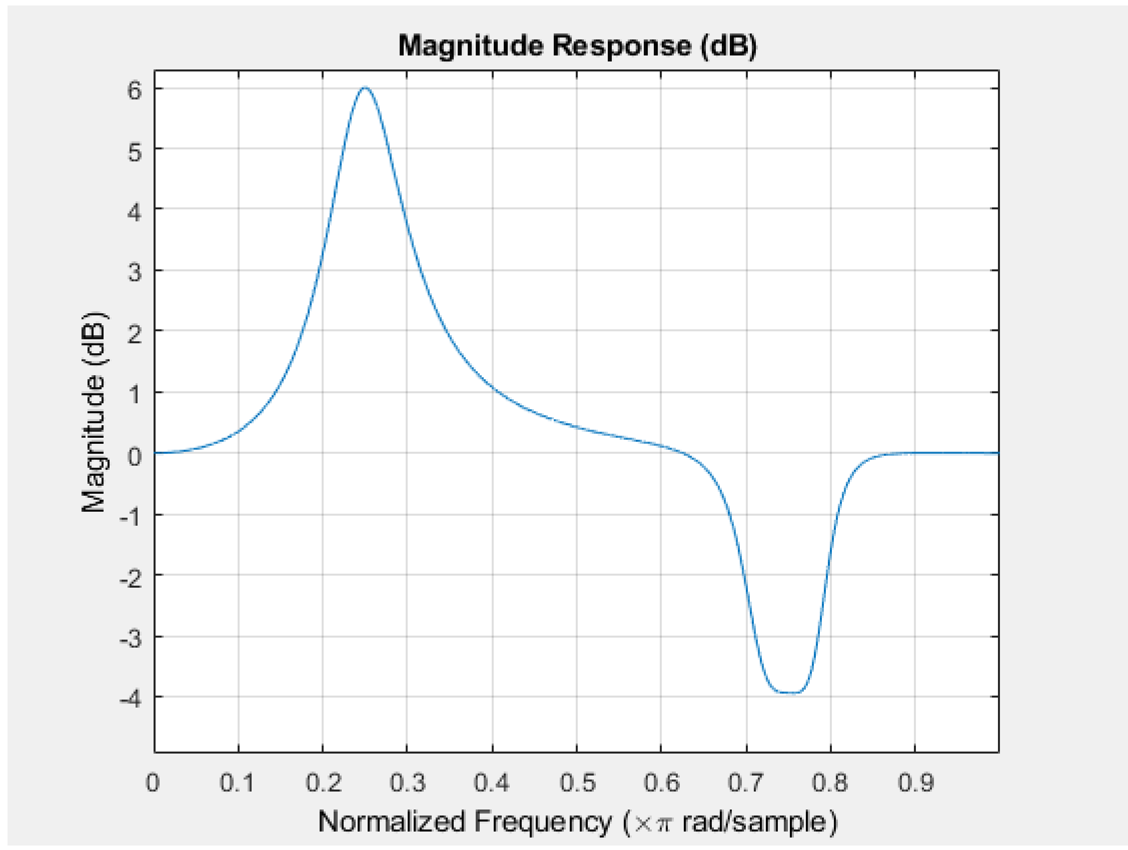
```
[B,A] = designParamEQ(N,gain,centerFreq,bandwidth);
```

Create a filter matrix compatible with `fvtool`.

```
SOS = [B',[ones(sum(N)/2,1),A']];
```

Visualize your filter design.

```
fvtool(SOS)
```



Filter Audio Using SOS Parametric Equalizer

Design a second-order sections (SOS) parametric equalizer using `designParamEQ`, and filter an audio stream.

Construct audio file reader and audio device writer System objects. Use the sample rate of the reader as the sample rate of the writer. Call `setup` to reduce the computational load of initialization in an audio stream loop.

```
frameSize = 256;

fileReader = dsp.AudioFileReader(...
    'RockGuitar-16-44p1-stereo-72secs.wav',...
    'SamplesPerFrame',frameSize);

sampleRate = fileReader.SampleRate;

deviceWriter = audioDeviceWriter(...
    'SampleRate',sampleRate);

setup(fileReader);
setup(deviceWriter,ones(frameSize,2));
```

Play the audio signal through your device.

```
count = 0;
while count < 2500
    audio = fileReader();
    deviceWriter(audio);
    count = count+1;
end
reset(fileReader);
```

Design a SOS parametric equalizer.

```
N = [4,4];
gain = [-25,35];
centerFreq = [0.01,0.5];
bandwidth = [0.35,0.5];
[B,A] = designParamEQ(N,gain,centerFreq,bandwidth);
```

Visualize your filter design.

```
SOS = [B',[ones(4,1),A']];
fvtool(SOS,...
    'Fs',fileReader.SampleRate,...
    'FrequencyScale','Log');
```

Construct a biquad filter System object.

```
myFilter = dsp.BiquadFilter(...
    'SOSMatrixSource','Input port',...
    'ScaleValuesInputPort',false);
```

Construct a spectrum analyzer to visualize the original audio signal and the audio signal passed through your parametric equalizer.

```
scope = dsp.SpectrumAnalyzer(...
    'SampleRate',sampleRate,...
    'PlotAsTwoSidedSpectrum',false,...
    'FrequencyScale','Log',...
    'FrequencyResolutionMethod','WindowLength',...
    'WindowLength',frameSize,...
    'Title','Original and Equalized Signals',...
    'ShowLegend',true,...
    'ChannelNames',{'Original Signal','Equalized Signal'});
```

Play the filtered audio signal and visualize the original and filtered spectrums.

```
setup(scope,ones(frameSize,2));
count = 0;
while count < 2500
    originalSignal = fileReader();
    equalizedSignal = myFilter(originalSignal,B,A);
    scope([originalSignal(:,1),equalizedSignal(:,1)]);
    deviceWriter(equalizedSignal);
    count = count+1;
end

release(scope)
release(deviceWriter)
release(fileReader)
```

Filter Audio Using FOS Parametric Equalizer

Design a fourth-order sections (FOS) parametric equalizer using `designParamEQ`, and filter an audio stream.

Construct audio file reader and audio device writer System objects. Use the sample rate of the reader as the sample rate of the writer. Call `setup` to reduce the computational load of initialization in an audio stream loop.

```
frameSize = 256;

fileReader = dsp.AudioFileReader(...
    'RockGuitar-16-44p1-stereo-72secs.wav',...
    'SamplesPerFrame',frameSize);

sampleRate = fileReader.SampleRate;

deviceWriter = audioDeviceWriter(...
    'SampleRate',sampleRate);

setup(fileReader);
setup(deviceWriter,ones(frameSize,2));
```

Play the audio signal through your device.

```
count = 0;
while count < 2500
    x = fileReader();
    deviceWriter(x);
    count = count+1;
end
reset(fileReader);
```

Design FOS parametric equalizer coefficients.

```
N = [2,4];
gain = [5,10];
centerFreq = [0.025,0.65];
bandwidth = [0.025,0.35];
mode = 'fos';

[B,A] = designParamEQ(N,gain,centerFreq,bandwidth,mode);
```

Construct FOS IIR filters.

```
section1 = dsp.IIRFilter('Numerator',B(:,1)','Denominator',[1,A(:,1)']);
section2 = dsp.IIRFilter('Numerator',B(:,2)','Denominator',[1,A(:,2)']);
```

Visualize the frequency response of your parametric equalizer.

```
[H1,w] = freqz(section1,8192,sampleRate);
H2 = freqz(section2,8192,sampleRate);

H = 20.*log10(abs(H1.*H2));
```

```
semilogx(w,H);
title('Magnitude Response (dB)')
xlabel('Frequency (Hz)')
ylabel('Magnitude (dB)')
grid on
```

Construct a spectrum analyzer to visualize the original audio signal and the audio signal passed through your parametric equalizer.

```
scope = dsp.SpectrumAnalyzer(...
    'SampleRate',sampleRate,...
    'PlotAsTwoSidedSpectrum',false,...
    'FrequencyScale','Log',...
    'FrequencyResolutionMethod','WindowLength',...
    'WindowLength',frameSize,...
    'Title','Original and Equalized Signals',...
    'ShowLegend',true,...
    'ChannelNames',{'Original Signal','Equalized Signal'});
```

Play the filtered audio signal, and visualize the original and filtered spectrums.

```
setup(scope,ones(frameSize,2));
count = 0;
while count < 2500
    x = fileReader();
    y = section1(x);
    z = section2(y);

    scope([x(:,1),z(:,1)]);

    deviceWriter(z);

    count = count + 1;
end

release(fileReader)
release(deviceWriter)
release(scope)
```

Input Arguments

N — Filter order

scalar | row vector

Filter order, specified as a scalar or row vector the same length as centerFreq. Elements of the vector must be even integers.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

gain — Peak gain (dB)

scalar | row vector

Peak gain in dB, specified as a scalar or row vector the same length as centerFreq. Elements of the vector must be real-valued.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

centerFreq — Normalized center frequency of equalizer bands

scalar | row vector

Normalized center frequency of equalizer bands, specified as a scalar or row vector of real values in the range 0 to 1, where 1 corresponds to the Nyquist frequency (π rad/sample). If centerFreq is specified as a row vector, separate equalizers are designed for each element of centerFreq.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

bandwidth — Normalized bandwidth

scalar | row vector

Normalized bandwidth, specified as a scalar or row vector the same length as centerFreq. Elements of the vector are specified as real values in the range 0 to 1, where 1 corresponds to the Nyquist frequency (π rad/sample).

Normalized bandwidth is measured at gain/2 dB. If gain is set to -Inf (notch filter), normalized bandwidth is measured at the 3 dB attenuation point: $10\times\log_{10}(0.5)$.

To convert octave bandwidth to normalized bandwidth, calculate the associated *Q*-factor as

$$Q = \frac{\sqrt{2^{(octave\ bandwidth)}}}{2^{(octave\ bandwidth)} - 1}.$$

Then convert to bandwidth

$$bandwidth = \frac{centerFreq}{Q}.$$

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`mode` — Design mode
`'sos'` (default) | `'fos'`

Design mode, specified as `'sos'` or `'fos'`.

- `'sos'` -– Implements your equalizer as cascaded second-order filters.
- `'fos'` -– Implements your equalizer as cascaded fourth-order filters. Because fourth-order sections do not require the computation of roots, they are generally more computationally efficient.

Data Types: `char` | `string`

Output Arguments

`B` — Numerator filter coefficients
matrix

Numerator filter coefficients, returned as a matrix. Each column of `B` corresponds to the numerator coefficients of a different second-order or fourth-order section of your cascaded equalizer.

`A` — Denominator filter coefficients
matrix

Denominator filter coefficients, returned as a matrix. Each column of `A` corresponds to the denominator coefficients of a different second-order or fourth-order section of your cascaded equalizer.

`A` does not include the leading unity coefficient for each section.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

Functions
`designShelvingEQ` | `designVarSlopeFilter`

System Objects
`dsp.BiquadFilter` | `multibandParametricEQ`

Topics

"Parametric Equalizer Design"
"Equalization"

Introduced in R2016a

designShelvingEQ

Design shelving equalizer

Syntax

```
[B,A] = designShelvingEQ(gain,slope,Fc)
[B,A] = designShelvingEQ(gain,slope,Fc,type)
```

Description

`[B,A] = designShelvingEQ(gain,slope,Fc)` designs a low-shelf equalizer with the specified gain, slope, and cutoff frequency, `Fc`. The equalizer is returned as cascaded second-order section (SOS) IIR filters.

`[B,A] = designShelvingEQ(gain,slope,Fc,type)` specifies the design type as a low-shelving or high-shelving equalizer.

Examples

Design Low-Shelf Equalizer

Design three second-order IIR low-shelf equalizers using `designShelvingEQ`. The three shelving equalizers use three separate slope specifications.

Specify sampling frequency, peak gain, slope coefficient, and normalized cutoff frequency for three shelving equalizers. The sampling frequency is in Hz. The peak gain is in dB.

```
Fs = 44.1e3;

gain = 5;

slope1 = 0.5;
slope2 = 0.75;
slope3 = 1;
```

```
Fc = 1000/(Fs/2);
```

Design the filter coefficients using the specified parameters.

```
[B1,A1] = designShelvingEQ(gain,slope1,Fc);
[B2,A2] = designShelvingEQ(gain,slope2,Fc);
[B3,A3] = designShelvingEQ(gain,slope3,Fc);
```

Create filter matrices compatible with `fvtool`.

```
SOS1 = [B1',[1,A1']];
SOS2 = [B2',[1,A2']];
SOS3 = [B3',[1,A3']];
```

Visualize your filter design.

```
fvtool(...
    dsp.BiquadFilter('SOSMatrix',SOS1),...
    dsp.BiquadFilter('SOSMatrix',SOS2),...
    dsp.BiquadFilter('SOSMatrix',SOS3),...
    'Fs',Fs,...
    'FrequencyScale','Log');

legend('slope = 0.1',...
    'slope = 0.5',...
    'slope = 1');
```

Filter Audio Using Low-Shelf Equalizer

Design a low-shelf equalizer, and then use it to filter an audio signal.

Construct audio file reader and audio device writer objects. Use the sample rate of the reader as the sample rate of the writer. Call `setup` to reduce the computational load of initialization in an audio stream loop.

```
frameSize = 256;

fileReader = dsp.AudioFileReader(...
    'RockGuitar-16-44p1-stereo-72secs.wav',...
    'SamplesPerFrame',frameSize);
```

```
sampleRate = fileReader.SampleRate;

deviceWriter = audioDeviceWriter(...
    'SampleRate',sampleRate);

setup(fileReader);
setup(deviceWriter,ones(frameSize,2));
```

Play the audio signal through your device.

```
count = 0;
while count < 2500
    audio = step(fileReader);
    play(deviceWriter,audio);
    count = count+1;
end
reset(fileReader)
```

Design a second-order sections (SOS) low-shelf equalizer.

```
gain = 10;
slope = 3;
Fc = 0.025;

[B,A] = designShelvingEQ(gain,slope,Fc);
```

Visualize your equalizer design.

```
SOS = [B',[1,A']];
fvtool(dsp.BiquadFilter('SOSMatrix',SOS),...
    'Fs',fileReader.SampleRate,...
    'FrequencyScale','Log');
```

Construct a biquad filter object.

```
myFilter = dsp.BiquadFilter(...
    'SOSMatrixSource','Input port',...
    'ScaleValuesInputPort',false);
```

Construct a spectrum analyzer object to visualize the original audio signal and the audio signal passed through your low-shelf equalizer.

```
scope = dsp.SpectrumAnalyzer(...
    'SampleRate',sampleRate,...
    'PlotAsTwoSidedSpectrum',false,...
```

```
    'FrequencyScale','Log',...
    'FrequencyResolutionMethod','WindowLength',...
    'WindowLength',frameSize,...
    'Title','Original and Equalized Signal',...
    'ShowLegend',true,...
    'ChannelNames',{'Original Signal','Equalized Signal'});
```

Play the equalized audio signal and visualize the original and equalized spectrums.

```
setup(scope,ones(frameSize,2));
count = 0;
while count < 2500
    originalSignal = fileReader();
    equalizedSignal = myFilter(originalSignal,B,A);
    scope([originalSignal(:,1),equalizedSignal(:,1)]);
    deviceWriter(equalizedSignal);
    count = count+1;
end

release(fileReader)
release(scope)
release(deviceWriter)
```

Design High-Shelf Equalizer

Design three second-order IIR high shelf equalizers using `designShelvingEQ`. The three shelving equalizers use three separate gain specifications.

Specify sampling frequency, peak gain, slope coefficient, and normalized cutoff frequency for three shelving equalizers. The sampling frequency is in Hz. The peak gain is in dB.

```
Fs = 44.1e3;

gain1 = -6;
gain2 = 6;
gain3 = 12;

slope = 0.8;

Fc = 18000/(Fs/2);
```

Design the filter coefficients using the specified parameters.

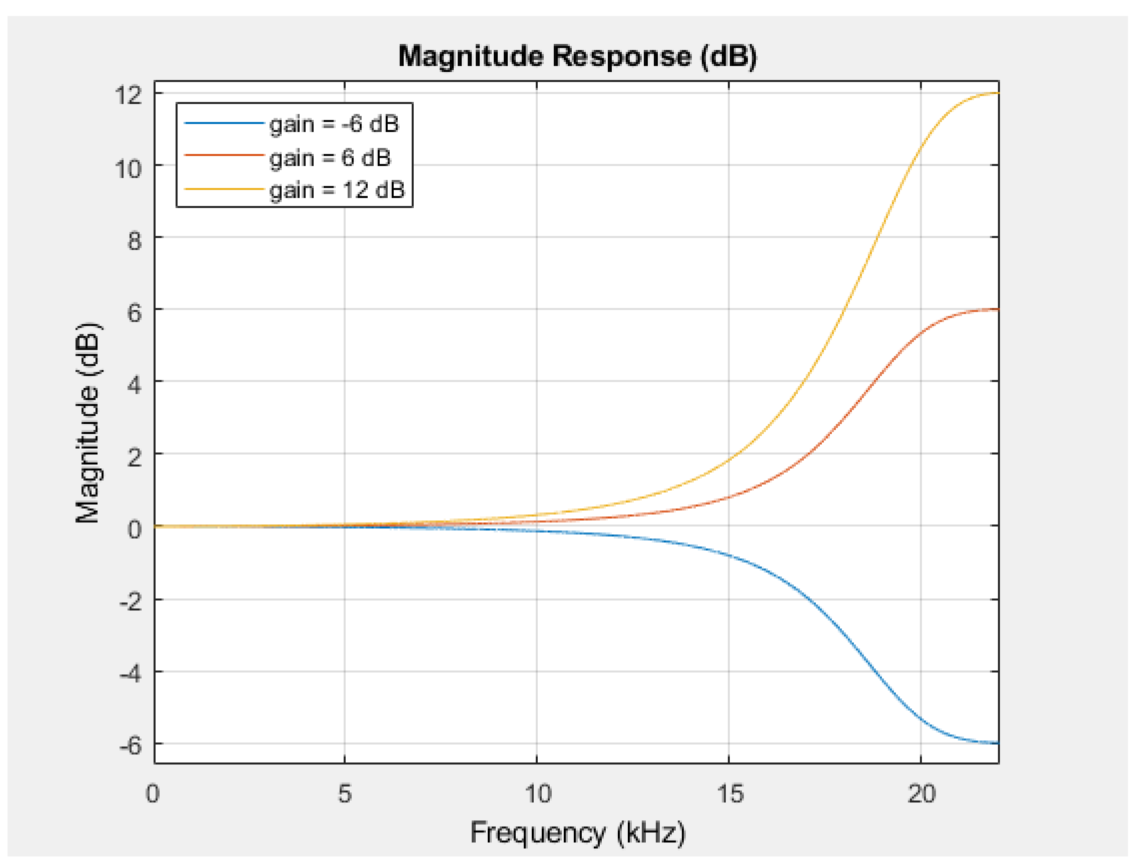
```
[B1,A1] = designShelvingEQ(gain1,slope,Fc,'hi');
[B2,A2] = designShelvingEQ(gain2,slope,Fc,'hi');
[B3,A3] = designShelvingEQ(gain3,slope,Fc,'hi');
```

Create filter matrices compatible with `fvtool`.

```
SOS1 = [B1',[1,A1']];
SOS2 = [B2',[1,A2']];
SOS3 = [B3',[1,A3']];
```

Visualize your filter design.

```
fvtool(dsp.BiquadFilter('SOSMatrix',SOS1),...
    dsp.BiquadFilter('SOSMatrix',SOS2),...
    dsp.BiquadFilter('SOSMatrix',SOS3),...
    'Fs',Fs);
legend('gain = -6 dB',...
    'gain = 6 dB',...
    'gain = 12 dB',...
    'Location','NorthWest')
```



Input Arguments

gain — Peak gain (dB)
real scalar in the range –12 to 12

Peak gain in dB, specified as a real scalar in the range –12 to 12.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

slope — Slope coefficient
real scalar in the range 0 to 5

Slope coefficient, specified as a real scalar in the range 0 to 5.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`Fc` — Normalized cutoff frequency
real scalar in the range 0 to 1

Normalized cutoff frequency, specified as a real scalar in the range 0 to 1, where 1 corresponds to the Nyquist frequency (π rad/sample).

Normalized cutoff frequency is implemented as half the shelving filter gain, or `gain`/2 dB.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`type` — Filter type
`'lo'` (default) | `'hi'`

Filter type, specified as `'lo'` or `'hi'`.

- `'lo'`-– Low shelving equalizer
- `'hi'`-– High shelving equalizer

Data Types: `char` | `string`

Output Arguments

`B` — Numerator filter coefficients
three-element column vector

Numerator filter coefficients of the designed second-order IIR filter, retuned as a three-element column vector.

`A` — Denominator filter coefficients
two-element column vector.

Denominator filter coefficients of the designed second-order IIR filter, returned as a two-element column vector. `A` does not include the leading unity coefficient.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

Functions

`designParamEQ` | `designVarSlopeFilter`

System Objects

`multibandParametricEQ`

Topics

"Parametric Equalizer Design"
"Equalization"

Introduced in R2016a

designVarSlopeFilter

Design variable slope lowpass or highpass IIR filter

Syntax

```
[B,A] = designVarSlopeFilter(slope,Fc)
[B,A] = designVarSlopeFilter(slope,Fc,type)
```

Description

`[B,A] = designVarSlopeFilter(slope,Fc)` designs a lowpass filter with the specified slope and cutoff frequency. `B` and `A` are matrices of numerator and denominator coefficients, with columns corresponding to cascaded second-order sections (SOS).

`[B,A] = designVarSlopeFilter(slope,Fc,type)` specifies the design type as a lowpass or highpass filter.

Examples

Design Lowpass IIR Filter

Design two second-order section (SOS) lowpass IIR filters using `designVarSlopeFilter`.

Specify the sampling frequency, slope, and normalized cutoff frequency for two lowpass IIR filters. The sampling frequency is in Hz. The slope is in dB/octave.

```
Fs = 48e3;

slope = 18;

Fc1 = 10000/(Fs/2);
Fc2 = 16000/(Fs/2);
```

Design the filter coefficients using the specified parameters.

```
[B1,A1] = designVarSlopeFilter(slope,Fc1);
[B2,A2] = designVarSlopeFilter(slope,Fc2);
```

Create filter matrices compatible with `fvtool`.

```
SOS1 = [B1',[ones(4,1),A1']];
SOS2 = [B2',[ones(4,1),A2']];
```

Visualize your filter design.

```
fvtool(SOS1,SOS2,'Fs',Fs);

legend('Fc = 10000 Hz',...
    'Fc = 16000 Hz',...
    'Location','SouthWest');
```

Process Audio Using Lowpass Filter

Design a second-order section (SOS) lowpass IIR filter using `designVarSlopeFilter`. Use your lowpass filter to process an audio signal.

Construct audio file reader and audio device writer System objects. Use the sample rate of the reader as the sample rate of the writer. Call `setup` to reduce the computational load of initialization in an audio stream loop.

```
frameSize = 256;

fileReader = dsp.AudioFileReader( ...
    'RockGuitar-16-44p1-stereo-72secs.wav', ...
    'SamplesPerFrame',frameSize);

sampleRate = fileReader.SampleRate;

deviceWriter = audioDeviceWriter( ...
    'SampleRate',sampleRate);

setup(fileReader);
setup(deviceWriter,ones(frameSize,2));
```

Play the audio signal through your device.

```
count = 0;
while count < 2500
```

```
    audio = fileReader();
    deviceWriter(audio);
    count = count+1;
end
reset(fileReader);
```

Design a lowpass filter with a 12 dB/octave slope and a 0.15 normalized frequency cutoff.

```
[B,A] = designVarSlopeFilter(12,0.15);
```

Visualize your filter design.

```
SOS = [B',[ones(4,1),A']];
fvtool(SOS, ...
    'Fs',sampleRate);
```

Construct a biquad filter System object.

```
myFilter = dsp.BiquadFilter( ...
    'SOSMatrixSource','Input port', ...
    'ScaleValuesInputPort',false);
```

Construct a spectrum analyzer System object to visualize the original audio signal and the audio signal passed through your lowpass filter.

```
scope = dsp.SpectrumAnalyzer( ...
    'SampleRate',sampleRate, ...
    'PlotAsTwoSidedSpectrum',false, ...
    'FrequencyScale','Log', ...
    'FrequencyResolutionMethod','WindowLength', ...
    'WindowLength',frameSize, ...
    'Title','Original and Equalized Signal', ...
    'ShowLegend',true, ...
    'ChannelNames',{'Original Signal','Filtered Signal'});
```

Play the filtered audio signal and visualize the original and filtered spectrums.

```
setup(scope,ones(frameSize,2));
count = 0;
while count < 2500
    originalSignal = fileReader();
    filteredSignal = myFilter(originalSignal,B,A);
    scope([originalSignal(:,1),filteredSignal(:,1)]);
    deviceWriter(filteredSignal);
    count = count+1;
end
```

Design Highpass IIR Filter

Design two second-order section (SOS) highpass IIR filters using `designVarSlopeFilter`.

Specify the sampling frequency in Hz, the slope in dB/octave, and the normalized cutoff frequency.

```
Fs = 48e3;
slope1 = 18;
slope2 = 36;
Fc = 4000/(Fs/2);
```

Design the filter coefficients using the specified parameters.

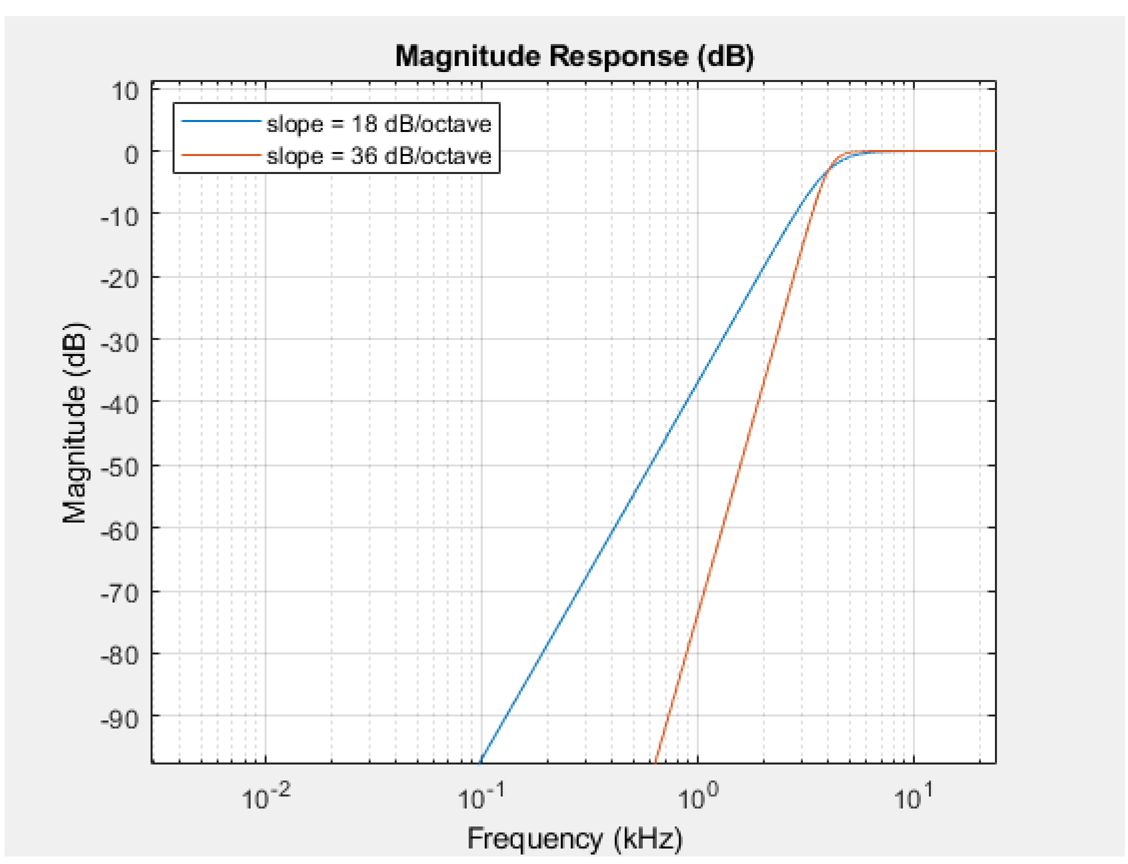
```
[B1,A1] = designVarSlopeFilter(slope1,Fc,'hi');
[B2,A2] = designVarSlopeFilter(slope2,Fc,'hi');
```

Create filter matrices compatible with `fvtool`.

```
SOS1 = [B1',[ones(4,1),A1']];
SOS2 = [B2',[ones(4,1),A2']];
```

Visualize your filter design.

```
fvtool(SOS1,SOS2,...
    'Fs',Fs,...
    'FrequencyScale','Log');
legend('slope = 18 dB/octave',...
    'slope = 36 dB/octave',...
    'Location','NorthWest')
```



Diminish Plosives from Speech Signal

Plosives are consonant sounds resulting from a sudden release of airflow. They are most pronounced in words beginning with *p*, *d*, and *g* sounds. Plosives can be emphasized by the recording process and are often displeasurable to hear. In this example, you minimize the plosives of a speech signal by applying highpass filtering and low-band compression.

Create a `dsp.AudioFileReader` System object™ and a `audioDeviceWriter` System object™ to read an audio signal from a file and write an audio signal to a device. Play the unprocessed signal. Then release the file reader and device writer.

```
fileReader   = dsp.AudioFileReader( ...
    fullfile(matlabroot,'examples','audio','Plosives.wav'));
deviceWriter = audioDeviceWriter;

while ~isDone(fileReader)
    audioIn = fileReader();
    deviceWriter(audioIn);
end
release(deviceWriter)
release(fileReader)
```

Design a highpass filter with a steep rolloff of all frequencies below 120 Hz. Use a `dsp.BiquadFilter` System object to implement the highpass filter design. Create a crossover filter with one crossover at 250 Hz. The crossover filter enables you to separate the band of interest for processing. Create a dynamic range compressor to compress the dynamic range of plosive sounds. To apply no make-up gain, set the `MakeUpGainMode` to `'Property'` and use the default 0 dB `MakeUpGain` property value. Create a time scope to visualize the processed and unprocessed audio signal.

```
[B,A] = designVarSlopeFilter(48,120/(44100/2),'hi');
biquadFilter = dsp.BiquadFilter( ...
    'SOSMatrixSource','Input port', ...
    'ScaleValuesInputPort',false);

crossFilt = crossoverFilter( ...
    'NumCrossovers',1, ...
    'CrossoverFrequencies',250, ...
    'CrossoverSlopes',48);

dRCompressor = compressor( ...
    'Threshold',-35, ...
    'Ratio',10, ...
    'KneeWidth',20, ...
    'AttackTime',1e-4, ...
    'ReleaseTime',3e-1, ...
    'MakeUpGainMode','Property', ...
    'SampleRate',fileReader.SampleRate);

scope = dsp.TimeScope( ...
    'SampleRate',fileReader.SampleRate, ...
    'TimeSpan',3, ...
    'BufferLength',fileReader.SampleRate*3*2, ...
    'YLimits',[-1 1], ...
    'ShowGrid',true, ...
```

```
    'ShowLegend',true, ...
    'ChannelNames',{'Original','Processed'});
```

In an audio stream loop:

1. Read in a frame of the audio file.
2. Apply highpass filtering using your biquad filter.
3. Split the audio signal into two bands.
4. Apply dynamic range compression to the lower band.
5. Remix the channels.
6. Write the processed audio signal to your audio device for listening.
7. Visualize the processed and unprocessed signals on a time scope.

As a best practice, release your objects once done.

```
while ~isDone(fileReader)
    audioIn = fileReader();

    audioIn = biquadFilter(audioIn,B,A);

    [band1,band2] = crossFilt(audioIn);

    band1compressed = dRCompressor(band1);

    audioOut  = band1compressed + band2;

    deviceWriter(audioOut);

    scope([audioIn audioOut]);
end

release(deviceWriter)
release(fileReader)
release(scope)
release(crossFilt)
release(dRCompressor)
```

Input Arguments

`slope` — Filter slope (dB/octave)

real scalar in the range [0:6:48]

Filter slope in dB/octave, specified as a real scalar in the range [0:6:48]. Values that are not multiples of 6 are rounded.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`Fc` — Normalized cutoff frequency

real scalar in the range 0 to 1

Normalized cutoff frequency, specified as a real scalar in the range 0 to 1, where 1 corresponds to the Nyquist frequency (π rad/sample).

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`type` — Filter type

`'lo'` (default) | `'hi'`

Filter type, specified as `'lo'` or `'hi'`.

- `'lo'`-– Lowpass filter
- `'hi'`-– Highpass filter

Data Types: `char` | `string`

Output Arguments

`B` — Numerator filter coefficients

3-by-4 matrix

Numerator filter coefficients, returned as a 3-by-4 matrix. Each column of `B` corresponds to the numerator coefficients of a different second-order section of your cascaded IIR filter.

`A` — Denominator filter coefficients

2-by-4 matrix

Denominator filter coefficients, returned as a 2-by-4 matrix. Each column of `A` corresponds to the denominator coefficients of a different second-order section of your cascaded IIR filter.

`A` does not include the leading unity coefficient for each section.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

Functions
`designParamEQ` | `designShelvingEQ`

System Objects
`multibandParametricEQ`

Topics

"Parametric Equalizer Design"
"Equalization"

Introduced in R2016a

disconnectMIDI

Disconnect MIDI controls from audio object

Syntax

```
disconnectMIDI(audioObject)
```

Description

`disconnectMIDI(audioObject)` disconnects MIDI controls from your audio object, `audioObject`. Only those MIDI connections established using `configureMIDI` are disconnected.

Examples

Disconnect MIDI Controls from Audio Plugin

Create an object of the audio plugin example `audiopluginexample.Echo`.

```
echoPlugin = audiopluginexample.Echo;
```

Get the MIDI connections of `echoPlugin` and verify that it has no MIDI connections.

```
myMIDIConnections = getMIDIConnections(echoPlugin);
isempty(myMIDIConnections)
```

```
ans =

     1
```

Add MIDI connections using `configureMIDI`.

```
configureMIDI(echoPlugin,'Delay1');
```

Get the MIDI connections of `echoPlugin` using `getMIDIConnections`. The MIDI connections you configured are saved as a structure. View details of the MIDI connections using dot notation.

```
myMIDIConnections = getMIDIConnections(echoPlugin);
myMIDIConnections.Delay1
```

```
ans =

            Law: 'lin'
            Min: 0
            Max: 1
    MIDIControl: 'any control on 'BCF2000''
```

Use `disconnectMIDI` to remove MIDI connections between your `echoPlugin` object and your MIDI device.

```
disconnectMIDI(echoPlugin);
```

Get MIDI connections of `echoPlugin` and verify that you have successfully disconnected MIDI controls from your plugin.

```
myMIDIConnections = getMIDIConnections(echoPlugin);
isempty(myMIDIConnections)
```

```
ans =

     1
```

Input Arguments

`audioObject` — Audio object
object

Audio plugin or compatible System object, specified as an object that inherits from the `audioPlugin` class or an object of a compatible Audio System Toolbox System object.

See Also

Classes
`audioPlugin` | `audioPluginSource`

Functions
`configureMIDI` | `getMIDIConnections` | `midicallback` | `midicontrols` | `midiid` | `midiread` | `midisync`

Topics
"MIDI Control for Audio Plugins"
"MIDI Control Surface Interface"

Introduced in R2016a

fdesign.parameq

Parametric equalizer filter specification

Syntax

```
d = fdesign.parameq(spec, specvalue1, specvalue2, ...)
d = fdesign.parameq(... fs)
```

Description

`d = fdesign.parameq(spec, specvalue1, specvalue2, ...)` constructs a parametric equalizer filter design object, where `spec` is a non-case sensitive character vector. The choices for `spec` are as follows:

- `'F0, BW, BWp, Gref, G0, GBW, Gp'` (minimum order default)
- `'F0, BW, BWst, Gref, G0, GBW, Gst'`
- `'F0, BW, BWp, Gref, G0, GBW, Gp, Gst'`
- `'N, F0, BW, Gref, G0, GBW'`
- `'N, F0, BW, Gref, G0, GBW, Gp'`
- `'N, F0, Fc, Qa, G0'`
- `'N, F0, Fc, S, G0'`
- `'N, F0 ,BW, Gref, G0, GBW, Gst'`
- `'N, F0, BW, Gref, G0, GBW, Gp, Gst'`
- `'N, Flow, Fhigh, Gref, G0, GBW'`
- `'N, Flow, Fhigh, Gref, G0, GBW, Gp'`
- `'N, Flow, Fhigh, Gref, G0, GBW, Gst'`
- `'N, Flow, Fhigh, Gref, G0, GBW, Gp, Gst'`

where the parameters are defined as follows:

Parameter	Definition	Unit
`BW`	Bandwidth	
`BWp`	Passband Bandwidth	
`BWst`	Stopband Bandwidth	
`Gref`	Reference Gain	decibels
`G0`	Center Frequency Gain	decibels
`GBW`	Gain at which Bandwidth (`BW`) is measured	decibels
`Gp`	Passband Gain	decibels
`Gst`	Stopband Gain	decibels
`N`	Filter Order	
`F0`	Center Frequency	
`Fc`	Cutoff Frequency	
`Fhigh`	Higher Frequency at Gain `GBW`	
`Flow`	Lower Frequency at Gain `GBW`	
`Qa`	Quality Factor	
`S`	Slope Parameter for Shelving Filters	

Regardless of the specification chosen, there are some conditions that apply to the specification parameters. These are as follows:

- Specifications for parametric equalizers must be given in decibels
- To boost the input signal, set `G0 > Gref`; to cut, set `Gref > G0`
- For boost: `G0 > Gp > GBW > Gst > Gref`; For cut: `G0 < Gp < GBW < Gst < Gref`
- Bandwidth must satisfy: `BWst > BW > BWp`

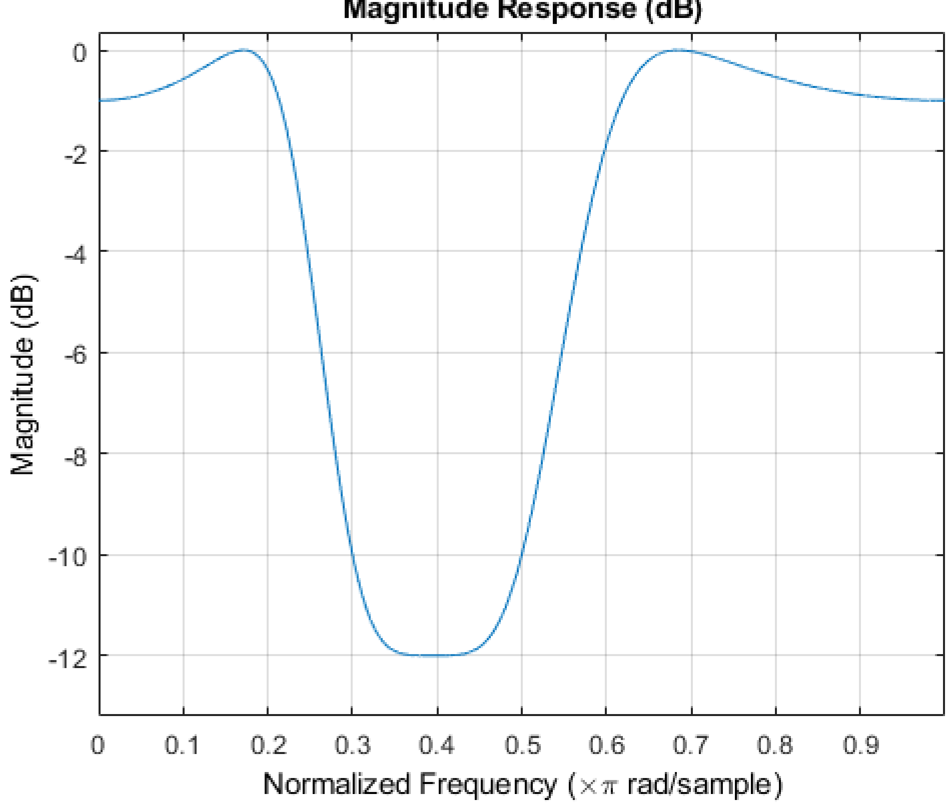
`d = fdesign.parameq(... fs)` adds the input sampling frequency. `fs` must be specified as a scalar trailing the other numerical values provided, and is assumed to be in Hz.

Examples

Design Parametric Equalizers

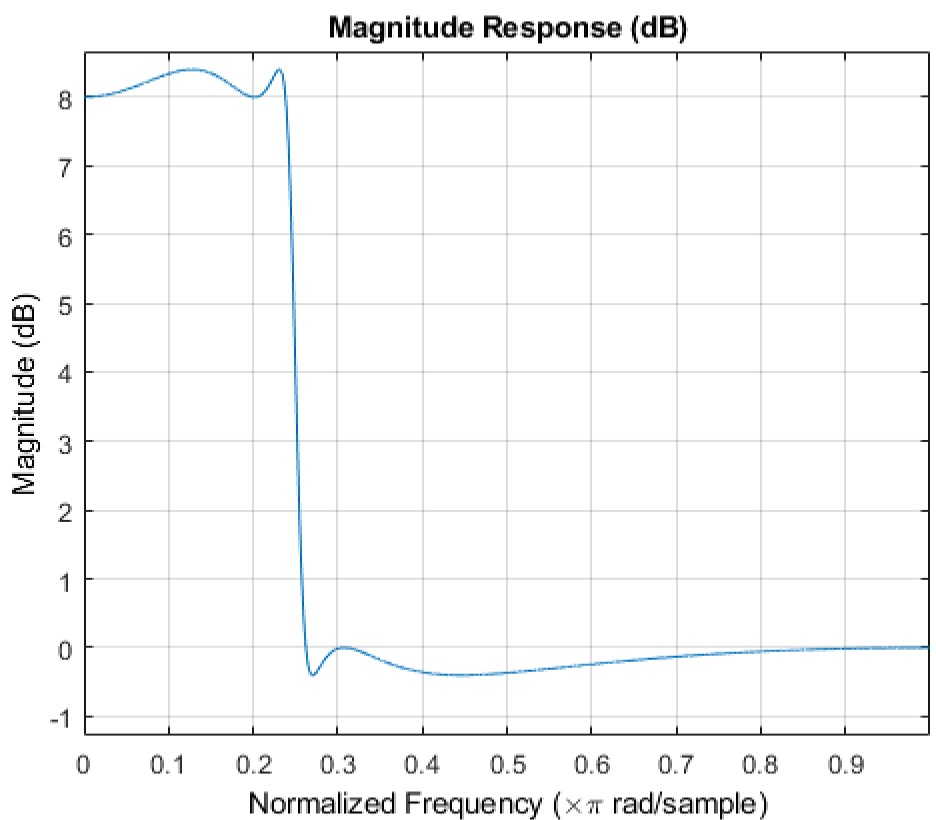
Design a Chebyshev Type II parametric equalizer filter that cuts by 12 dB.

```
parametricEQ = fdesign.parameq('N,Flow,Fhigh,Gref,G0,GBW,Gst', ...
    4,0.3,0.5,0,-12,-10,-1);

parametricEQBiquad = design(parametricEQ,'cheby2','SystemObject',true);
fvtool(parametricEQBiquad)
```



Design a 4th-order lowpass shelving filter with a normalized cutoff frequency of 0.25, a quality factor of 10, and an 8 dB boost gain.

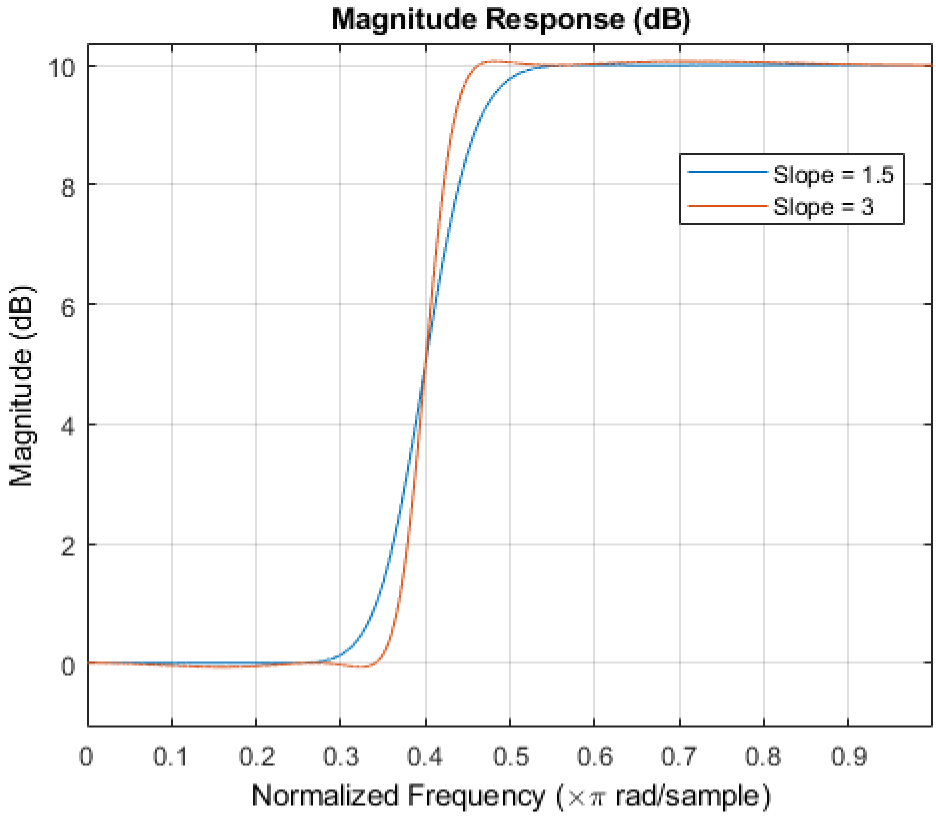
```
parametricEQ  = fdesign.parameq('N,F0,Fc,Qa,G0',4,0,0.25,10,8);
parametricEQBiquad = design(parametricEQ,'SystemObject',true);
fvtool(parametricEQBiquad)
```



Design 4th-order highpass shelving filters with slopes of 1.5 and 3.

```
N  = 4;   % Filter order
F0 = 1;   % Center Frequency (normalized)
Fc = 0.4; % Cutoff Frequency (normalized)
G0 = 10;  % Center Frequency Gain (dB)
```

```
S1 = 1.5; % Slope for filter design 1
S2 = 3;   % Slope for filter design 2

filter = fdesign.parameq('N,F0,Fc,S,G0',N,F0,Fc,S1,G0);
filterDesignS1 = design(filter,'SystemObject',true);

filter.S = S2;
filterDesignS2 = design(filter,'SystemObject',true);

filterVisualization = fvtool(filterDesignS1,filterDesignS2);
legend(filterVisualization,'Slope = 1.5','Slope = 3');
```



See Also

`design` | `designParamEQ` | `designShelvingEQ` | `designVarSlopeFilter` | `fdesign` | `multibandParametricEQ`

Topics

"Parametric Equalizer Design"
"Equalization"

generateAudioPlugin

Generate audio plugin from MATLAB class

Syntax

```
generateAudioPlugin className
generateAudioPlugin options className
```

Description

`generateAudioPlugin className` generates a VST 2 audio plugin from a MATLAB class specified by `className`. See Supported Compilers for a list of compilers supported by `generateAudioPlugin`.

`generateAudioPlugin options className` specifies nondefault output folder, plugin name, or file type. Options can be specified in any grouping, and in any order.

Examples

Generate Audio Plugin

```
generateAudioPlugin audiopluginexample.Echo
```

A VST 2 plugin named `Echo` is saved to your current folder. The extension of your plugin depends on your operating system.

Specify Output Folder for Generated Plugin

```
generateAudioPlugin -outdir myPluginFolder audiopluginexample.Echo
```

A VST 2 plugin named `Echo` is saved to your specified folder. The extension of your plugin depends on your operating system.

Specify File Name of Generated Plugin

```
generateAudioPlugin -output awesomeEffect audiopluginexample.Echo
```

A VST 2 plugin named `awesomeEffect` is saved to your current folder. The extension of your plugin depends on your operating system.

Specify Output Folder and File Name of Generated Plugin

```
generateAudioPlugin -output coolEffect -outdir myPluginFolder audiopluginexample.Echo
```

A VST 2 plugin named `coolEffect` is saved to your specified folder. The extension of your plugin depends on your operating system.

Generate win32 Plugin from win64 System

```
generateAudioPlugin -win32 audiopluginexample.Echo
```

A 32-bit VST 2 plugin named `Echo.dll` is saved to your current folder.

Input Arguments

`options` — Options to specify output folder, plugin name, and file type
`-outdir` *`folder`* | `-output` *`pluginName`* | `-win32`

Options can be specified in any grouping, and in any order.

Option	Description
`-outdir` *`folder`*	Generates a plugin to a specific folder. By default, the generated plugin is placed in the current folder. If *`folder`* is not in the current directory, specify the exact path.

Option	Description
`-output` *`pluginName`*	Specifies the file name of the generated plugin. The appropriate extension is appended to the *`pluginName`* based on the platform on which the plugin is generated. By default, the plugin is named after the class.
`-win32`	Creates a 32-bit audio plugin. Valid only on win64.

`className` — Name of the plugin class to generate
plugin class

Name of the plugin class to generate. The plugin class must be on the MATLAB path. It must derive from either the `audioPlugin` class or the `audioPluginSource` class.

You can specify the plugin class to generate by specifying its class name or file name. For example, the following syntaxes perform equivalent operations:

- `generateAudioPlugin myPlugin`
- `generateAudioPlugin myPlugin.m`

If you want to specify the plugin class by file name, and your plugin class is inside a package, you must specify the package as a file path. For example, the following syntaxes perform equivalent operations:

- `generateAudioPlugin myPluginPackage.myPlugin`
- `generateAudioPlugin +myPluginPackage/myPlugin.m`

Limitations

Build problems can occur when using folder names with spaces. For more information, see “Enable Build Process for Folder Names with Spaces” (Simulink Coder) and Why is the build process failing for a shipped model in Simulink or for a model run in Accelerator mode?.

Definitions

Generated Plugin File Extension

The extension of your generated plugin depends on your operating system.

Operating System	File Extension
Windows	.dll
OSX	.vst

See Also

Apps
Audio Test Bench

Functions
loadAudioPlugin | validateAudioPlugin

Classes
audioPlugin | audioPluginSource

Topics
"Design an Audio Plugin"
"Export a MATLAB Plugin to a DAW"

Introduced in R2016a

integratedLoudness

Measure integrated loudness and loudness range

Syntax

```
loudness = integratedLoudness(audioIn,Fs)
loudness = integratedLoudness(audioIn,Fs,channelWeights)
[loudness,loudnessRange] = integratedLoudness( ___ )
```

Description

`loudness = integratedLoudness(audioIn,Fs)` returns the integrated loudness of an audio signal, `audioIn`, with sample rate `Fs`. The ITU-R BS.1770-4 and EBU R 128 standards define the algorithms to calculate integrated loudness.

`loudness = integratedLoudness(audioIn,Fs,channelWeights)` specifies the channel weights used to compute the integrated loudness. `channelWeights` must be a row vector with the same number of elements as the number of channels in `audioIn`.

`[loudness,loudnessRange] = integratedLoudness( ___ )` returns the loudness range of the audio signal using either of the previous syntaxes. The EBU R 128 Tech 3342 standard defines the loudness range computation.

Examples

Determine Integrated Loudness

Determine the integrated loudness of an audio signal.

Create a two-second sine wave with a 0 dB amplitude, a 1 kHz frequency, and a 48 kHz sample rate.

```
sampleRate = 48e3;
increment  = sampleRate*2;
```

```
amplitude  = 10^(0/20);
frequency  = 1e3;

sineGenerator = audioOscillator(...
    'SampleRate',sampleRate,...
    'SamplesPerFrame',increment,...
    'Amplitude',amplitude,...
    'Frequency', frequency);

signal = sineGenerator();
```

Calculate the integrated loudness of the audio signal at the specified sample rate.

```
loudness = integratedLoudness(signal,sampleRate)
```

Specify Nondefault Channel Weights

Read in a four-channel audio signal. Specify a nondefault weighting vector with four elements.

```
[signal,fs] = audioread('AudioArray-16-16-4channels-20secs.wav');
weightingVector = [1,0.8,0.8,1.2];
```

Calculate the integrated loudness with the default channel weighting and the nondefault channel weighting vector.

```
standardLoudness    = integratedLoudness(signal,fs,weightingVector)
nonStandardLoudness = integratedLoudness(signal,fs)
```

Determine Loudness Range

Read in an audio signal. Clip 3 five-second intervals out of the signal.

```
[x,fs] = audioread('FunkyDrums-44p1-stereo-25secs.mp3');
x1 = x(1:fs*5,:);
x2 = x(5e5:5e5+5*fs,:);
x3 = x(end-5*fs:end,:);
```

Calculate the loudness and loudness range of the total signal and of each interval.

```
[L,LRA] = integratedLoudness(x,fs);
[L1,LRA1] = integratedLoudness(x1,fs);
[L2,LRA2] = integratedLoudness(x2,fs);
[L3,LRA3] = integratedLoudness(x3,fs);

fprintf(['Loudness: %0.2f\n',...
    'Loudness range: %0.2f\n\n',...
    'Beginning loudness: %0.2f\n',...
    'Beginning loudness range: %0.2f\n\n',...
    'Middle loudness: %0.2f\n',...
    'Middle loudness range: %0.2f\n\n',...
    'End loudness: %0.2f\n',...
    'End loudness range: %0.2f\n'],...
    L,LRA,L1,LRA1,L2,LRA2,L3,LRA3);
```

Input Arguments

audioIn — Input signal
matrix

Input signal, specified as a matrix. The columns of the matrix are treated as audio channels.

The maximum number of columns of the input signal depends on your `channelWeights` specification:

- If you use the default `channelWeights`, the input signal has a maximum of five channels. Specify the channels in this order: [Left, Right, Center, Left surround, Right surround].
- If you specify nondefault `channelWeights`, the input signal must have the same number of columns as the number of elements in the `channelWeights` vector.

Data Types: `single` | `double`

Fs — Sample rate (Hz)
positive scalar

Sample rate of the input signal in Hz, specified as a positive scalar.

Data Types: `single` | `double`

`channelWeights` — Linear weighting applied to each input channel
`[1.0, 1,0, 1.0, 1.41, 1.41]` (default) | nonnegative row vector

Linear weighting applied to each input channel, specified as a row vector of nonnegative values. The number of elements in the row vector must be equal to or greater than the number of input channels. Excess values in the vector are ignored.

The default channel weights follow the ITU-R BS.1170-4 standard. To use the default channel weights, specify the channels of the `audioIn` matrix in this order: [Left, Right, Center, Left surround, Right surround].

It is a best practice to specify the `channelWeights` vector in order: [Left, Right, Center, Left surround, Right surround].

Data Types: `single` | `double`

Output Arguments

`loudness` — Integrated loudness (LUFS)
scalar

Integrated loudness in loudness units relative to full scale (LUFS), returned as a scalar.

The ITU-R BS.1770-4 and EBU R 128 standards define the integrated loudness. The algorithm computes the loudness by breaking down the audio signal into 0.4-second segments with 75% overlap. If the input signal is less than 0.4 seconds, `loudness` is returned empty.

Data Types: `single` | `double`

`loudnessRange` — Loudness range (LU)
scalar

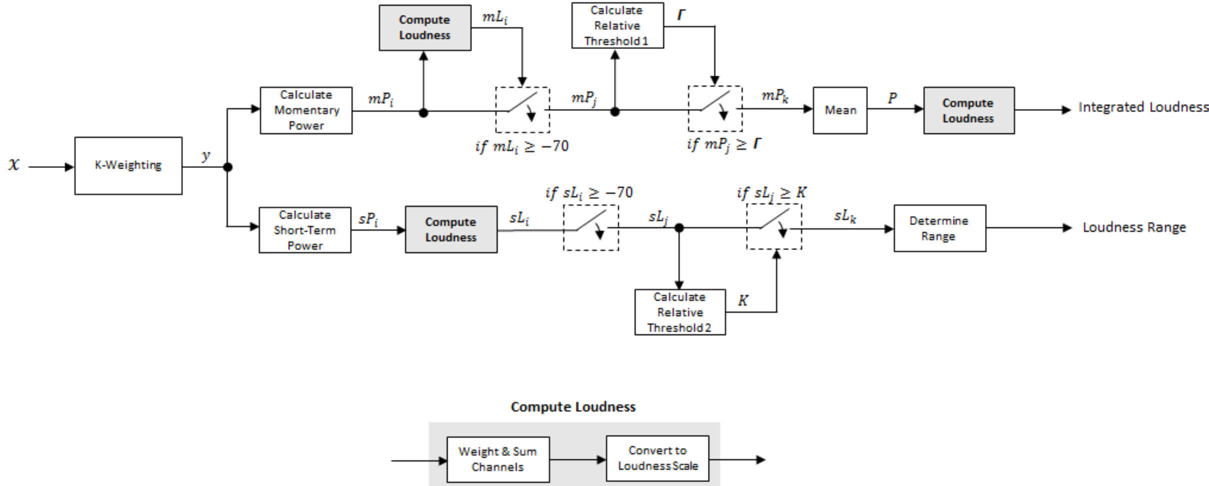
Loudness range in loudness units (LU), returned as a scalar.

The EBU R 128 Tech 3342 standard defines the loudness range. The algorithm computes the loudness range by breaking down the audio into 3-second segments with 2.9-second overlap. If the input signal is less than three seconds, `loudnessRange` is returned empty.

Data Types: `single` | `double`

Algorithms

The `integratedLoudness` function returns the integrated loudness and loudness range (LRA) of an audio signal. You can specify any number of channels and nondefault channel weights used for loudness measurements. The `integratedLoudness` algorithm is described for the general case of *n* channels.



Integrated Loudness and Loudness Range

The input channels, *x*, pass through a K-weighted `weightingFilter`. The K-weighted filter shapes the frequency spectrum to reflect perceived loudness.

1 The K-weighted channels, *y*, are divided into 0.4-second segments with 0.3-second overlap. The power (mean square) of each segment of the K-weighted channels is calculated:

$$mP_i = \frac{1}{w}\sum_{k=1}^{w} y_i^2[k]$$

- mP_i is the momentary power of the *i*th segment of a channel.

- w is the segment length in samples.

2 The momentary loudness, mL, is computed for each segment:

$$mL_i = -0.691 + 10\ \log_{10}\left(\sum_{c=1}^{n} G_c \times mP_{(i,c)}\right) \quad LUFS$$

- G_c is the weighting for channel c.

3 The momentary power is gated using the momentary loudness calculation:

$$mP_i \rightarrow mP_j$$

$$j = \left\{\ i \mid mL_i \geq -70\ \right\}$$

4 The relative threshold, Γ, is computed:

$$\Gamma = -0.691 + 10\log_{10}\left(\sum_{c=1}^{n} G_c \times l_c\right) - 10$$

l_c is the mean momentary power of channel c:

$$l_c = \frac{1}{|j|}\sum_{j} mP_{(j,c)}$$

5 The momentary power subset, mP_j, is gated using the relative threshold:

$$mP_j \rightarrow mP_k$$

$$k = \left\{\ j \mid mP_j \geq \Gamma\ \right\}$$

6 The momentary power segments are averaged:

$$P = \frac{1}{|k|}\sum_{k} mP_k$$

7 The integrated loudness is computed by passing the mean momentary power subset, P, through the Compute Loudness system:

$$\text{Integrated Loudness} = -0.691 + 10\log_{10}\left(\sum_{c=1}^{n} G_c \times P_c\right) \quad LUFS$$

1. The K-weighted channels, y, are divided into 3-second segments with 2.9-second overlap. The power (mean square) of each segment of the K-weighted channels is calculated:

   $$sP_i = \frac{1}{w}\sum_{k=1}^{w} y_i^2[k]$$

   - sP_i is the short-term power of the ith segment of a channel.
   - w is the segment length in samples.
2. The short-term loudness, sL, is computed for each segment:

   $$sL_i = -0.691 + 10\ \log_{10}\left(\sum_{c=1}^{n} G_c \times sP_{(i,c)}\right)$$

   - G_c is the weighting for channel c.
3. The short-term loudness is gated using an absolute threshold:

   $$sL_i \rightarrow sL_j$$

   $$j = \left\{\ i \mid sL_i \geq -70\ \right\}$$
4. The gated short-term loudness is converted back to linear, and then the mean is taken:

   $$sP_j = \frac{1}{|j|}\sum_{j} 10^{\left(sL_j/10\right)}$$

   The relative threshold, K, is computed:

   $$K = -20 + 10\log_{10}\left(sP_j\right)$$

5 The short-term loudness subset, sL_j, is gated using the relative threshold:

$$sL_j \rightarrow sL_k$$

$$k = \{\ j \mid sL_j \geq K\ \}$$

6 The short-term loudness subset, sL_k, is sorted. The loudness range is calculated as between the 10th and 95th percentiles of the distribution, and is returned in loudness units (LU).

References

[1] International Telecommunication Union; Radiocommunication Sector. *Algorithms to Measure Audio Programme Loudness and True-Peak Audio Level*. ITU-R BS. 1770-4. 2015.

[2] European Broadcasting Union. *Loudness Normalisation and Permitted Maximum Level of Audio Signals*. EBU R 128. 2014.

[3] European Broadcasting Union. *Loudness Metering: 'EBU Mode' Metering to Supplement EBU R 128 Loudness Normalization*. EBU R 128 Tech 3341. 2014.

[4] European Broadcasting Union. *Loudness Range: A Measure to Supplement EBU R 128 Loudness Normalization*. EBU R 128 Tech 3342. 2016.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

System Objects

`loudnessMeter` | `weightingFilter`

Blocks
Loudness Meter

Introduced in R2016b

getMIDIConnections

Get MIDI connections of audio object

Syntax

```
connectionInfo = getMIDIConnections(audioObject)
```

Description

`connectionInfo = getMIDIConnections(audioObject)` returns a structure, `connectionInfo`, containing information about the MIDI connections for your audio object, `audioObject`. Only those MIDI connections established using `configureMIDI` are returned.

The `connectionInfo` structure contains a substructure for each tunable property of `audioObject` that has established MIDI connections. Each substructure contains the control number, the device name of the corresponding MIDI control, and the property mapping information (mapping rule, minimum value, and maximum value).

Examples

Get MIDI Connections of Plugin

Create an object of the audio plugin example `audiopluginexample.Echo`.

```
echoEffect = audiopluginexample.Echo;
```

Use `configureMIDI` to synchronize `echoEffect` properties with specific MIDI controls on the default MIDI device.

```
configureMIDI(echoEffect,'Delay1',1001);
configureMIDI(echoEffect,'Gain1' ,1002);
configureMIDI(echoEffect,'Delay2',1003);
configureMIDI(echoEffect,'Gain2' ,1004);
```

Use `getMIDIConnections` to view the MIDI connections you established.

```
connectionInfo = getMIDIConnections(echoEffect)
```

```
connectionInfo = 

    Delay1: [1x1 struct]
     Gain1: [1x1 struct]
    Delay2: [1x1 struct]
     Gain2: [1x1 struct]
```

View details of the `Delay1` MIDI connection using dot notation.

```
connectionInfo.Delay1
```

```
ans = 

            Law: 'lin'
            Min: 0
            Max: 1
    MIDIControl: 'control 1001 on 'nanoKONTROL2''
```

Input Arguments

`audioObject` — Audio object
object

Audio plugin or compatible System object, specified as an object that inherits from the `audioPlugin` class or an object of a compatible Audio System Toolbox System object.

Output Arguments

`connectionInfo` — Information about MIDI connection
structure

Information about MIDI connection between the specified audio plugin object and MIDI devices, returned as a structure. Only those MIDI connections established using `configureMIDI` are returned. The `connectionInfo` structure contains a substructure for each established MIDI connection. Each substructure contains the control number, the device name of the corresponding MIDI control, and the property mapping information (mapping rule, minimum value, and maximum value).

See Also

Classes
`audioPlugin` | `audioPluginSource`

Functions
`configureMIDI` | `disconnectMIDI` | `midicallback` | `midicontrols` | `midiid` | `midiread` | `midisync`

Topics
"MIDI Control for Audio Plugins"
"MIDI Control Surface Interface"

Introduced in R2016a

loadAudioPlugin

Load VST, VST3, and AU plugins into MATLAB environment

Syntax

```
hostedPlugin = loadAudioPlugin(pluginpath)
```

Description

`hostedPlugin = loadAudioPlugin(pluginpath)` loads the 64-bit VST, VST3, or AU audio plugin specified by `pluginpath`.

Your hosted plugin has two display modes: `Parameters` and `Properties`. The default display mode is `Properties`.

- `Parameters` –– Interact with normalized parameter values of the hosted plugin using set and get functions.
- `Properties` –– Interact with heuristically interpreted parameters with real-world values. You can use standard dot notation to set and get the values while using this mode.

You can specify the display mode of the hosted plugin using standard dot notation, for example:

```
hostedPlugin.DisplayMode = 'Parameters';
```

See “Host External Audio Plugins” for a discussion of display modes and a walkthrough of both modes of interaction.

You can interact with and exercise the hosted plugin using the following functions.

Process Audio

- `audioOut = process(hostedPlugin,audioIn)`

  Returns an audio signal processed according to the algorithm and parameters of the hosted plugin. For source plugins, call `process` without an audio input.

Set and Get Normalized Parameter Values

- `value = getParameter(hostedPlugin,parameter)`

  Returns the normalized value of the specified hosted plugin parameter. Normalized values are in the range [0,1]. You can specify a parameter by its name or by its index. To specify the name, use a character vector.
- `setParameter(hostedPlugin,parameter,newValue)`

  Sets the normalized value of the specified hosted plugin parameter to `newValue`. Normalized values are in the range [0,1].

Get High-Level Information About the Hosted Plugin

- `dispParameter(hostedPlugin)`

  Displays all parameters and associated indices, values, displayed values, and display labels of the hosted plugin.
- `pluginInfo = info(hostedPlugin)`

  Returns a structure containing information about the hosted plugin.

Set the Environment in Which the Plugin Is Run

- `frameSize = getSamplesPerFrame(hostedPlugin)`

  Returns the frame size that the hosted plugin returns in subsequent calls to its processing function (source plugins only).
- `setSamplesPerFrame(hostedPlugin,frameSize)`

  Sets the frame size that the hosted plugin must return in subsequent calls to its processing function (source plugins only).
- `setSampleRate(hostedPlugin,sampleRate)`

  Sets the sample rate of the hosted plugin.
- `sampleRate = getSampleRate(hostedPlugin)`

  Returns the sample rate in Hz at which the plugin is being run.

Examples

Host External Plugins in MATLAB

Use loadAudioPlugin to host a VST external plugin and a VST external source plugin in MATLAB®.

Use the fullfile command to determine the full path to the oscillator VST plugin and parametric equalizer VST plugin included with Audio System Toolbox™. If you are using a Mac, replace the .dll file extension with .vst.

```
oscPluginPath = ...
    fullfile(matlabroot,'toolbox/audio/samples/oscillator.dll');
EQPluginPath = ...
    fullfile(matlabroot,'toolbox/audio/samples/ParametricEqualizer.dll');
```

Create external plugin objects by calling loadAudioPlugin for each of the plugin paths.

```
hostedSourcePlugin = loadAudioPlugin(oscPluginPath);
hostedPlugin = loadAudioPlugin(EQPluginPath);
```

Hosted plugins derive from either the externalAudioPlugin or externalAudioSourcePlugin class. Because oscillator.dll is a source audio plugin, the hosted object derives from externalAudioSourcePlugin. Use class() to verify the classes of the hosted plugins.

```
class(hostedPlugin)
```

```
ans =

    'externalAudioPlugin'
```

```
class(hostedSourcePlugin)
```

```
ans =

    'externalAudioPluginSource'
```

Call the hosted plugins to display basic information about them. This information includes the format, the plugin name, the number of channels in and out, and the tunable properties of the plugin. Source plugins also display the frame size of the plugin.

```
hostedSourcePlugin
hostedPlugin
```

```
hostedSourcePlugin = 

  VST plugin 'oscillator'  source, 1 out, 256 samples

    Frequency: 100 Hz
    Amplitude: 1 AU
     DCOffset: 0 AU

hostedPlugin = 

  VST plugin 'ParametricEQ'  2 in, 2 out

              LowPeakGain: 0 dB
       LowCenterFrequency: 100 Hz
               LowQFactor: 2
           MediumPeakGain: 0 dB
    MediumCenterFrequency: 1000 Hz
            MediumQFactor: 2
             HighPeakGain: 0 dB
      HighCenterFrequency: 10000 Hz
              HighQFactor: 2
```

Run External Plugin in MATLAB

Load a VST audio plugin into MATLAB™ by specifying its full path. If you are using a Mac, replace the `.dll` file extension with `.vst`.

```
pluginPath = ...
    fullfile(matlabroot,'toolbox/audio/samples/ParametricEqualizer.dll');
hostedPlugin = loadAudioPlugin(pluginPath);
```

Create input and output objects for an audio stream loop that reads from a file and writes to your audio device. Set the sample rate of the hosted plugin to the sample rate of the input to the plugin.

```
fileReader   = dsp.AudioFileReader('FunkyDrums-44p1-stereo-25secs.mp3');
deviceWriter = audioDeviceWriter('SampleRate',fileReader.SampleRate);
setSampleRate(hostedPlugin,fileReader.SampleRate);
```

Set the `MediumPeakGain` property to -20 dB.

```
hostedPlugin.MediumPeakGain = -20;
```

Use the hosted plugin to process the audio file in an audio stream loop. Sweep the medium peak gain upward in the loop to hear the effect.

```
while hostedPlugin.MediumPeakGain < 19
    hostedPlugin.MediumPeakGain = hostedPlugin.MediumPeakGain + 0.04;
    x = fileReader();
    y = process(hostedPlugin,x);
    deviceWriter(y);
end

release(fileReader)
release(deviceWriter)
```

Run External Source Plugin in MATLAB

Load a VST audio source plugin into MATLAB™ by specifying its full path. If you are using a Mac, replace the `.dll` file extension with `.vst`.

```
pluginPath = fullfile(matlabroot,'toolbox','audio','samples','oscillator.dll');
hostedSourcePlugin = loadAudioPlugin(pluginPath);
```

Set the `Amplitude` property to 0.5. Set the `Frequency` property to 16 kHz.

```
hostedSourcePlugin.Amplitude = 0.5;
hostedSourcePlugin.Frequency = 16000;
```

Set the sample rate at which to run the plugin. Create an output object to write to your audio device.

```
setSampleRate(hostedSourcePlugin,44100);
deviceWriter = audioDeviceWriter('SampleRate',44100);
```

Use the hosted source plugin to output an audio stream. The processing in the audio stream loop ramps the frequency parameter down and then up.

```
k = 1;
for i = 1:1000
    hostedSourcePlugin.Frequency = hostedSourcePlugin.Frequency - 30*k;
    y = process(hostedSourcePlugin);
    deviceWriter(y);
    if (hostedSourcePlugin.Frequency - 30 <= 0.1) || ...
            (hostedSourcePlugin.Frequency + 30 >= 20e3)
        k = -1*k;
    end
end

release(deviceWriter)
```

Input Arguments

pluginpath — Location of external plugin
character vector | string

Location of the external plugin, specified as a character vector. Use the full path to specify the audio plugin you want to host in MATLAB. If the plugin is located in the current folder, specify it by its name.

Example: `loadAudioPlugin('coolPlugin.dll')`

Example: `loadAudioPlugin('C:\Program Files\VSTPlugins\coolPlugin.dll')`

Plugin Path for Mac

For macOS, the plugin locations are predetermined depending on if the plugin was saved system wide or for a particular user.

This table shows the system-wide paths.

Plugin Type	Path
VST2	/Library/Audio/Plug-Ins/VST/coolPlugin.vst
VST3	/Library/Audio/Plug-Ins/VST3/coolPlugin.vst3
AU	/Library/Audio/Plug-Ins/Components/coolPlugin.component

This table shows the user-specific paths.

Plugin Type	Path
VST2	`~/Library/Audio/Plug-Ins/VST/coolPlugin.vst`
VST3	`~/Library/Audio/Plug-Ins/VST3/coolPlugin.vst3`
AU	`~/Library/Audio/Plug-Ins/Components/coolPlugin.component`

Output Arguments

`hostedPlugin` — Object of external plugin
`externalAudioPlugin` | `externalAudioSourcePlugin`

Object of an external plugin, derived from the `externalAudioPlugin` or `externalAudioSourcePlugin` class. You can interact with the hosted plugin as a DAW would, with the additional functionality of the MATLAB environment.

Limitations

The `loadAudioPlugin` function supports 64-bit plugins only. You cannot load 32-bit plugins using the `loadAudioPlugin` function.

See Also

Classes
`audioPlugin` | `audioPluginSource` | `externalAudioPlugin` | `externalAudioPluginSource`

Topics
"Host External Audio Plugins"

Introduced in R2016b

midicallback

Call function handle when MIDI controls change value

Syntax

```
oldFunctionHandle = midicallback(midicontrolsObject,functionHandle)
oldFunctionHandle = midicallback(midicontrolsObject,[])
currentFunctionHandle = midicallback(midicontrolsObject)
```

Description

`oldFunctionHandle = midicallback(midicontrolsObject,functionHandle)` sets `functionHandle` as the function handle called when `midicontrolsObject` changes value, and returns the previous function handle, `oldFunctionHandle`.

`oldFunctionHandle = midicallback(midicontrolsObject,[])` clears the function handle.

`currentFunctionHandle = midicallback(midicontrolsObject)` returns the current function handle.

Examples

Interactively Read MIDI Controls

Create a default MIDI controls object. Use `midicallback` to associate an anonymous function with your MIDI controls object, `mc`.

```
mc = midicontrols;
midicallback(mc,@(x)disp(midiread(x)));
```

Move any control on your default MIDI device to display its current normalized value on the command line.

```
0.5079

0.5000

0.4921

0.4841

0.4762

0.4683

0.4603

0.4683
```

Use midicallback to Update Plot

Use `midiid` to identify the name of your MIDI device and a specified control. Move the MIDI control you want to identify.

```
[controlNumber,deviceName] = midiid;
```

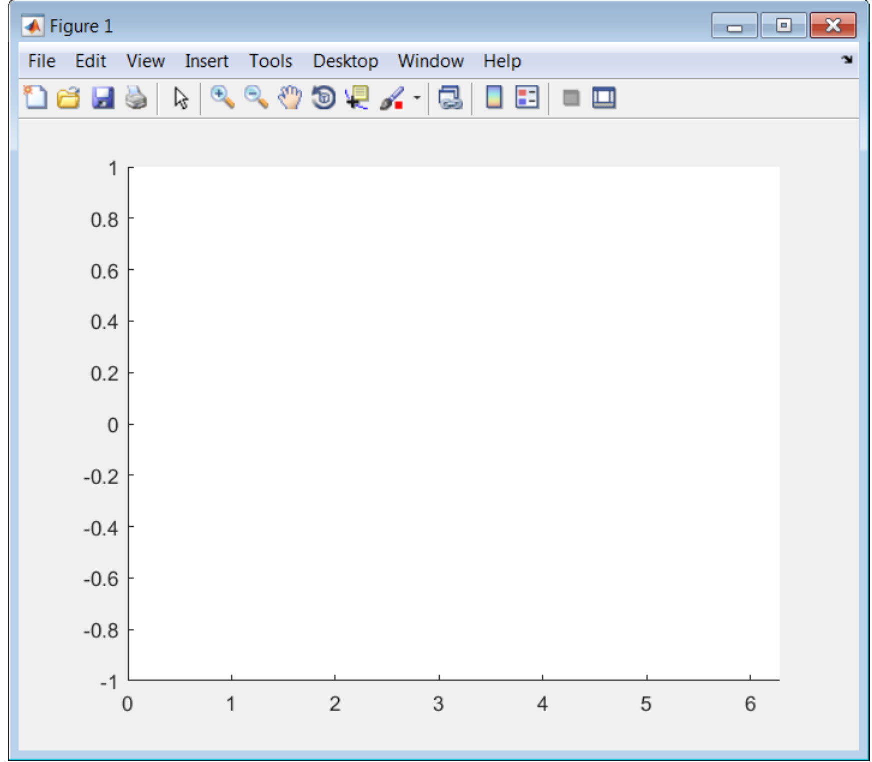
```
Move the control you wish to identify; type ^C to abort.
Waiting for control message...
```

Create an object that responds to the control you specified.

```
midicontrolsObject = midicontrols(controlNumber);
```

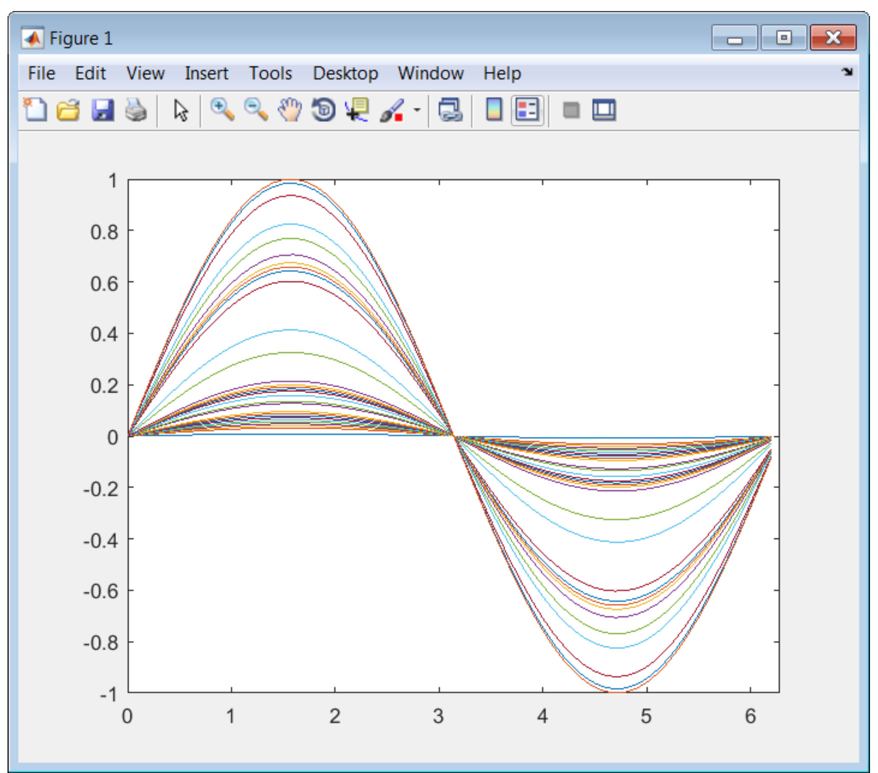
Define a function that plots a sinusoid with the amplitude set by your MIDI control. Make the axis constant.

```
axis([0,2*pi,-1,1]);
axis manual
hold on
sinePlotter = @(obj) plot(0:0.1:2*pi,midiread(obj).*sin(0:0.1:2*pi));
```



Use the `midicallback` function to associate your `sinePlotter` function with the control specified by your `midicontrolsObject`. Move your specified MIDI control. The plot updates automatically with the sinusoid amplitude specified by your MIDI control.

```
midicallback(midicontrolsObject,sinePlotter)
```



Change Function Handle Associated with MIDI Control

Create an object that responds to any control on the default MIDI device.

```
midicontrolsObject = midicontrols;
```

Define an anonymous function to display the current value of the MIDI control. Use `midicallback` to associate your MIDI control object with the function you created. Verify that your object is associated with your function.

```
displayControlValue = @(object) disp(midiread(object));
midicallback(midicontrolsObject,displayControlValue);
currentFunctionHandle = midicallback(midicontrolsObject)
```

```
currentFunctionHandle =

    @(object)disp(midiread(object))
```

Move any control on your default MIDI device to display its current normalized value on the command line.

```
    0.3095

    0.4603

    0.6746

    0.7381

    0.8175

    0.8571

    0.9048
```

Define an anonymous function to print the current value of the MIDI control rounded to two significant digits. Use `midicallback` to associate your MIDI controls object with the function you created. Return the old function handle.

```
displayRoundedControlValue = @(object) fprintf('%.2f\n',midiread(object));
oldFunctionHandle = midicallback(midicontrolsObject,displayRoundedControlValue)
```

```
oldFunctionHandle =

    @(object)disp(midiread(object))
```

Move a control to display its current normalized value rounded to two significant digits.

```
0.91
0.83
0.67
0.49
0.29
0.18
0.05
```

Remove the association between the object and the function. Return the old function handle.

```
oldFunctionHandle = midicallback(midicontrolsObject,[])
```

```
oldFunctionHandle =

    @(object)fprintf('%.2f\n',midiread(object))
```

Verify that no function is associated with your MIDI controls object.

```
currentFunctionHandle = midicallback(midicontrolsObject)
```

```
currentFunctionHandle =

     []
```

Associate a Function with MIDI Controls

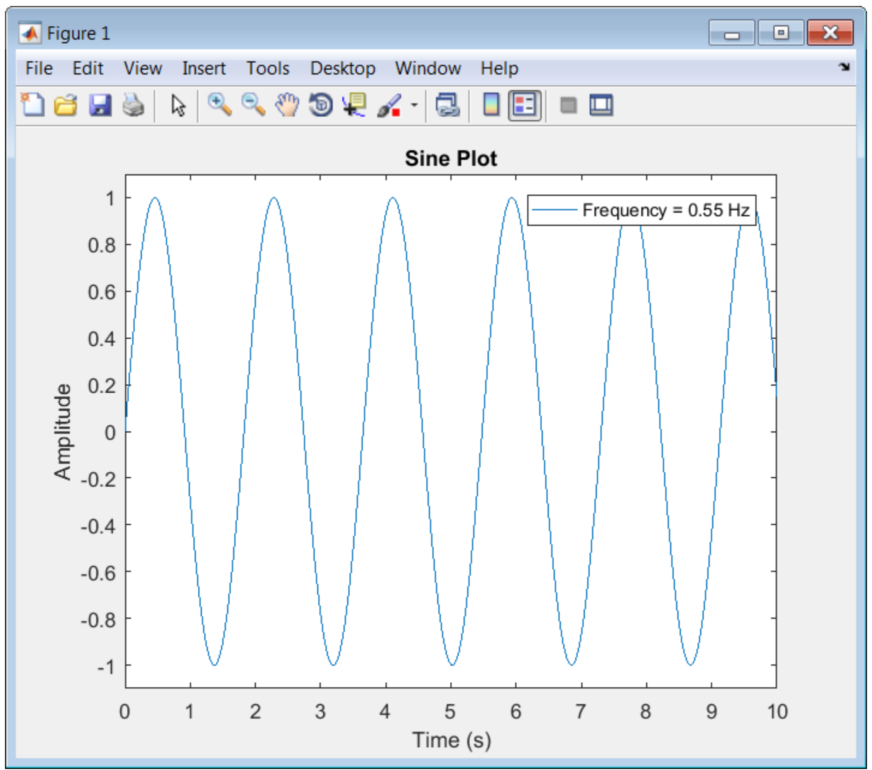
Define this function and save it to your current folder.

```
function plotSine(midicontrolsObject)

frequency = midiread(midicontrolsObject);

x = 0:0.01:10;

sinusoid = sin(2*pi*frequency.*x);

plot(x,sinusoid)
axis([0,10,-1.1,1.1]);
ylabel('Amplitude');
xlabel('Time (s)');
title('Sine Plot')
legend(sprintf('Frequency = %0.2f Hz',frequency));

end
```

Create a `midicontrols` object. Create a function handle for your `plotSine` function. Use `midicallback` to associate your `midicontrolsObject` with `plotSineHandle`.

Move any controller on your MIDI device to plot a sinusoid. The sinusoid frequency updates when you move MIDI controls.

```
midicontrolsObject = midicontrols;
plotSineHandle = @plotSine;
midicallback(midicontrolsObject,plotSineHandle);
```



Input Arguments

`midicontrolsObject` — Object that listens to the controls on a MIDI device
object

Object that listens to the controls on a MIDI device, specified as an object created by `midicontrols`.

`functionHandle` — New function handle
function handle

New function handle, specified as a function handle that contains one input argument. The new function handle is called when `midicontrolsObject` changes value. For information on what function handles are, see “Function Handles” (MATLAB).

Output Arguments

`oldFunctionHandle` — Old function handle
function handle

Old function handle set by the previous call to `midicallback`, returned as a function handle.

`currentFunctionHandle` — Current function handle
function handle

The function handle set by the most recent call to `midicallback`, returned as a function handle.

See Also

Functions
`configureMIDI` | `disconnectMIDI` | `getMIDIConnections` | `midicontrols` | `midiid` | `midiread` | `midisync` | `setpref`

Topics
"MIDI Control Surface Interface"
"MIDI Control for Audio Plugins"

midicontrols

Open group of MIDI controls for reading

Syntax

```
midicontrolsObject = midicontrols
midicontrolsObject = midicontrols(controlNumbers)
midicontrolsObject = midicontrols(controlNumbers,initialValues)
midicontrolsObject = midicontrols( ___ ,'MIDIDevice',deviceName)
midicontrolsObject = midicontrols( ___ ,'OutputMode',mode)
```

Description

`midicontrolsObject = midicontrols` returns an object that listens to all controls on your default MIDI device.

Call `midiread` with the object to return the values of controls on your MIDI device. If you call `midiread` before a control is moved, `midiread` returns the initial value of your `midicontrols` object.

`midicontrolsObject = midicontrols(controlNumbers)` listens to controls specified by `controlNumbers` on your default MIDI device.

`midicontrolsObject = midicontrols(controlNumbers,initialValues)` specifies `initialValues` associated with `controlNumbers`.

`midicontrolsObject = midicontrols( ___ ,'MIDIDevice',deviceName)` specifies the MIDI device your `midicontrols` object listens to, using any of the previous syntaxes.

`midicontrolsObject = midicontrols( ___ ,'OutputMode',mode)` specifies the range of values returned by `midiread` and accepted as `initialValues` for `midicontrols` and as `controlValues` for `midisync`.

Examples

Listen to Any Control on Default Device

Create a `midicontrols` object and read the default control value.

```
midicontrolsObject = midicontrols
midiread(midicontrolsObject)
```

```
midicontrolsObject =

midicontrols object: any control on 'BCF2000'

ans =

     0
```

Move any control on your MIDI device. Use `midiread` to return the most recent value of the last control moved.

```
midiread(midicontrolsObject)
```

```
ans =

    0.3810
```

Listen to Specific Control

Use `midiid` to identify the name of your MIDI device and a specified control. Move the MIDI control you want to identify.

```
[controlNumber,deviceName] = midiid;
```

```
Move the control you wish to identify; type ^C to abort.
Waiting for control message...
```

Create an object that responds to the control you specified.

```
midicontrolsObject = midicontrols(controlNumber);
```

Move your selected MIDI control, and then use `midiread` to return its most recent value.

```
midicontrolsObject = midiread(midicontrolsObject);

ans =

    0.4048
```

Specify Control Numbers and Initial Value

Determine the control numbers of four different controls on your MIDI device.

```
[controlNumber1,~] = midiid;
[controlNumber2,~] = midiid;
[controlNumber3,~] = midiid;
[controlNumber4,~] = midiid;

controlNumbers = [controlNumber1,controlNumber3;...
                  controlNumber2,controlNumber4]

Move the control you wish to identify; type ^C to abort.
Waiting for control message... done
Move the control you wish to identify; type ^C to abort.
Waiting for control message... done
Move the control you wish to identify; type ^C to abort.
Waiting for control message... done
Move the control you wish to identify; type ^C to abort.
Waiting for control message... done

controlNumbers =

        1081        1085
        1082        1087
```

Create a `midicontrols` object that listens to your specified controls. Specify an initial value for all controls.

```
initialValue = 0.5;
midicontrolsObject = midicontrols(controlNumbers,initialValue);
```

Move one of your specified controls, and then read the latest value of all your specified controls.

```
midiread(midicontrolsObject)
```

```
ans =

    0.0873    0.5000
    0.5000    0.5000
```

Specify Controls Numbers, Initial Value, and Output Mode

Determine the control numbers of two different controls on your MIDI device.

```
[controlNumber1,~] = midiid;
[controlNumber2,~] = midiid;

controlNumbers = [controlNumber1,controlNumber2];
```

```
Move the control you wish to identify; type ^C to abort.
Waiting for control message... done
Move the control you wish to identify; type ^C to abort.
Waiting for control message... done
```

Create a `midicontrols` object that listens to your specified controls. Specify an initial value for all controls.

```
initialValue = 12;
midicontrolsObject = midicontrols(controlNumbers,initialValue,'OutputMode','rawmidi');
```

Move one of your specified controls, and then read the latest value of all your specified controls.

```
midiread(midicontrolsObject)
```

```
ans =

    63    12
```

Set the Default MIDI Device

Assume that your MIDI device is a Behringer BCF2000. Enter this syntax at the MATLAB command line:

```
setpref midi DefaultDevice BCF2000
```

This preference persists across MATLAB sessions. You do not need to set it again unless you want to change your default device.

Specify Control Numbers and MIDI Device Name

Assume that your MIDI device is a Behringer BCF2000 and has a control with identification number 1001. Create a `midicontrols` object, which listens to control number 1001 on your Behringer BCF2000 device.

```
midicontrolsObject = midicontrols(1001,'MIDIDevice','BCF2000');
```

Input Arguments

`controlNumbers` — MIDI device control numbers
integer | array of integers

MIDI device control numbers, specified as an integer or array of integers. Use `midiid` to interactively identify the control numbers of your device. See "MIDI Device Control Numbers" on page 2-145 for an advanced explanation of how `controlNumbers` are determined.

If you specify `controlNumbers` as an empty vector, `[ ]`, then the `midicontrols` object responds to any control on your MIDI device.

Example: `1081`

Data Types: `double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`initialValues` — Initial values of MIDI controls
`0` (default) | scalar | array the same size as `controlNumbers`

Initial values of MIDI controls, specified as a scalar or an array the same size as `controlNumbers`. If you specify `initialValues` as a scalar, all controls specified by `controlNumbers` are assigned that value.

The value associated with your MIDI controls cannot be determined until you move a MIDI control. If you specify an initial value associated with your MIDI control, the initial value is returned by the `midiread` function until the MIDI control is moved.

- If `OutputMode` is specified as `'normalized'`, then initial values must be in the range [0,1]. Actual initial values are quantized and can be slightly different from initial values specified when your `midicontrols` object is created.
- If `OutputMode` is specified as `'rawmidi'`, then initial values must be integers in the range [0,127]

Example: `0.3`

Example: `[0,0.3,0.6]`

Example: `5`

Example: `[5;15;20]`

Data Types: `double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

deviceName — MIDI device name
character vector | string

MIDI device name, assigned by the device manufacturer or host operating system, specified as a string. The specified `deviceName` can be a substring of the exact name of your device. If you do not specify `deviceName`, the default MIDI device is used. See "Set the Default MIDI Device" on page 2-142 for an example of specifying a default MIDI device.

If you do not set a default MIDI device, the host operating system chooses the default device in an unspecified way. As a best practice, use `midiid` to identify the name of the device you want.

Example: `'MIDIDevice','BCF2000 MIDI 1'`

Data Types: `char` | `string`

mode — Output mode for MIDI control value
`'normalized'` (default) | `'rawmidi'`

Output mode for MIDI control value, specified as `'normalized'` or `'rawmidi'`.

- `'normalized'` — Values of your MIDI control are normalized. If your `midicontrols` object is called by `midiread`, then values in the range [0,1] are returned.
- `'rawmidi'` — Values of your MIDI control are not normalized. If your `midicontrols` object is called by `midiread`, then integer values in the range [0,127] are returned.

Example: `'OutputMode','normalized'`

Example: `'OutputMode','rawmidi'`

Data Types: `char` | `string`

Output Arguments

`midicontrolsObject` — Object that listens to the controls on a MIDI device
object

Object that listens to the controls on a MIDI device.

Definitions

MIDI Device Control Numbers

MATLAB defines MIDI device control numbers as (*MIDI Channel Number*) × 1000 + (*MIDI Controller Number*).

- MIDI Channel Number is the transmission channel that your device uses to send messages. This value is in the range 1–16.
- MIDI Controller Number is a number assigned to an individual control on your MIDI device. This value is in the range 1–127.

Your MIDI device determines the values of *MIDI Channel Number* and *MIDI Controller Number*.

See Also

Functions
`configureMIDI` | `disconnectMIDI` | `getMIDIConnections` | `midicallback` | `midiid` | `midiread` | `midisync` | `setpref`

Topics
"MIDI Control Surface Interface"
"MIDI Control for Audio Plugins"

midiid

Interactively identify MIDI control

Syntax

```
[controlNumber,deviceName] = midiid
```

Description

`[controlNumber,deviceName] = midiid` returns the control number and device name of the MIDI control you move. Call the function and then move the control you want to identify. The function detects which control you move and returns the control number and device name that specify that control.

Examples

Identify Control Number and Device Name

Call `midiid` and then move the control you want to identify on the MIDI device you want to identify.

```
[ctl,dev] = midiid;
Move the control you wish to identify; type ^C to abort.
Waiting for control message...

ctl =
1002
```

```
dev =
nanoKONTROL
```

Output Arguments

`controlNumber` — MIDI device control number
integer

MIDI device control number, specified as an integer. The device manufacturer assigns the value to the control for identification purposes.

`deviceName` — MIDI device name
string

MIDI device name assigned by the device manufacturer or host operating system, specified as a string.

See Also

Functions
`configureMIDI` | `disconnectMIDI` | `getMIDIConnections` | `midicallback` | `midiread` | `midisync` | `setpref`

Topics
"MIDI Control Surface Interface"
"MIDI Control for Audio Plugins"

midiread

Return most recent value of MIDI controls

Syntax

```
controlValues = midiread(midicontrolsObject)
```

Description

`controlValues = midiread(midicontrolsObject)` returns the most recent value of the MIDI controls associated with the specified `midicontrolsObject`. To create this object, use the `midicontrols` function.

Examples

Read Control Values of MIDI Device

```
midicontrolsObject = midicontrols;
controlValue = midiread(midicontrolsObject);
```

Read Multiple Control Values of MIDI Device

Identify two MIDI controls on your MIDI device.

```
[controlOne,~] = midiid
[controlTwo,~] = midiid
```

```
Move the control you wish to identify; type ^C to abort.
Waiting for control message... done

controlOne =

        1081
```

```
Move the control you wish to identify; type ^C to abort.
Waiting for control message... done

controlTwo =

        1082
```

Create a MIDI controls object that listens to both controls you identified.

```
controlNumbers = [controlOne,controlTwo];
midicontrolsObject = midicontrols(controlNumbers);
```

Move your specified MIDI controls and return their values. The values are returned as a vector that corresponds to your control numbers vector, `controlNumbers`.

```
tic
while toc < 5
    controlValues = midiread(midicontrolsObject)
end

controlValues =

    0.0397    0.0556
```

Read Control Values in an Audio Stream Loop

Use `midiid` to identify the name of your MIDI device and a specified control. Move the MIDI control you want to identify.

```
[controlNumber, deviceName] = midiid;

Move the control you wish to identify; type ^C to abort.
Waiting for control message... done
```

Create a MIDI controls object. The value associated with your MIDI controls object cannot be determined until you move the MIDI control. Specify an initial value associated with your MIDI control. The `midiread` function returns the initial value until the MIDI control is moved.

```
initialControlValue = 1;
midicontrolsObject = midicontrols(controlNumber,initialControlValue);
```

Create a `dsp.AudioFileReader` System object with default settings. Create an `audioDeviceWriter` System object and specify the sample rate.

```
fileReader = dsp.AudioFileReader('RockDrums-44p1-stereo-11secs.mp3');
deviceWriter = audioDeviceWriter(...
    'SampleRate',fileReader.SampleRate);
```

In an audio stream loop, read an audio signal frame from the file, apply gain specified by the control on your MIDI device, and then write the frame to your audio output device. By default, the control value returned by `midiread` is normalized.

```
while ~isDone(fileReader)
    audioData = step(fileReader);

    controlValue = midiread(midicontrolsObject);

    gain = controlValue*2;
    audioDataWithGain = audioData*gain;

    play(deviceWriter,audioDataWithGain);
end
```

Close the input file and release your output device.

```
release(fileReader);
release(deviceWriter);
```

Input Arguments

`midicontrolsObject` — Object that listens to the controls on a MIDI device
object

Object that listens to the controls on a MIDI device, specified as an object created by `midicontrols`.

Output Arguments

`controlValues` — Most recent values of MIDI controls
`[0,1]` (default) | integer values in the range `[0,127]`

Most recent values of MIDI controls, returned as normalized values in the range `[0,1]`, or as integer values in the range `[0,127]`. The output values depend on the `OutputMode` specified when your `midicontrols` object is created.

- If `OutputMode` was specified as `'normalized'`, then `midiread` returns values in the range `[0,1]`. The default `OutputMode` is `'normalized'`.
- If `OutputMode` was specified as `'rawmidi'`, then `midiread` returns integer values in the range `[0,127]`, and no quantization is required.

See Also

Functions
`configureMIDI` | `disconnectMIDI` | `getMIDIConnections` | `midicallback` | `midicontrols` | `midiid` | `midisync` | `setpref`

Topics
"MIDI Control Surface Interface"
"MIDI Control for Audio Plugins"

midisync

Send values to MIDI controls for synchronization

Syntax

```
midisync(midicontrolsObject)
midisync(midicontrolsObject,controlValues)
```

Description

`midisync(midicontrolsObject)` sends the initial values of controls to your MIDI device, as specified by your MIDI controls object. To create this object, use the `midicontrols` function. If your MIDI device can receive and respond to messages, it adjusts its controls as specified.

Note Many MIDI devices are not bidirectional. Calling `midisync` with a unidirectional device has no effect. `midisync` cannot tell whether a value is successfully sent to a device or even whether the device is bidirectional. If sending a value fails, no errors or warnings are generated.

`midisync(midicontrolsObject,controlValues)` sends `controlValues` to the MIDI controls associated with the specified `midicontrolsObject`.

Examples

Synchronize MIDI Control to Initial Value

Use `midiid` to identify a control on your default MIDI device.

```
[controlNumber,~] = midiid;
```

```
Move the control you wish to identify; type ^C to abort.
Waiting for control message... done
```

Create a MIDI controls object. Specify an initial value for your control. Call `midisync` to set the specified control on your device to the initial value.

```
initialValue = 0.5;
midicontrolsObject = midicontrols(controlNumber,initialValue);
midisync(midicontrolsObject);
```

Synchronize MIDI Control to Specified Value

Use `midiid` to identify three controls on your default MIDI device.

```
[controlNumber1,~] = midiid;
[controlNumber2,~] = midiid;
[controlNumber3,~] = midiid;
controlNumbers = [controlNumber1,controlNumber2,controlNumber3];
```

```
Move the control you wish to identify; type ^C to abort.
Waiting for control message... done
Move the control you wish to identify; type ^C to abort.
Waiting for control message... done
Move the control you wish to identify; type ^C to abort.
Waiting for control message... done
```

Create a MIDI controls object. Specify initial values for your controls. Call `midisync` to set the specified control on your device to the initial value.

```
controlValues = [0,0,1];
midicontrolsObject = midicontrols(controlNumbers,controlValues);
midisync(midicontrolsObject);
```

Create a loop that updates your control values and synchronizes those values to the physical controls on your device.

```
for i = 1:100
    controlValues = controlValues + [0.006,0.008,-0.008];
    midisync(midicontrolsObject,controlValues);
    pause(0.1)
end
```

Create UI Slider and Synchronize with MIDI Control

Define this function and save it to your current folder.

```
function trivialmidigui(controlNumber,deviceName)

    slider = uicontrol('Style','slider');
    mc = midicontrols(controlNumber,'MIDIDevice',deviceName);
    midisync(mc);
    set(slider,'Callback',@slidercb);
    midicallback(mc, @mccb);

    function slidercb(slider,~)
        val = get(slider,'Value');
        midisync(mc, val);
        disp(val);
    end

    function mccb(mc)
        val = midiread(mc);
        set(slider,'Value',val);
        disp(val);
    end

end
```

Use `midiid` to identify a control number and device name. Call the function you created, specifying the control number and device name as inputs.

```
[controlNumber,deviceName] = midiid;
trivialmidigui(controlNumber,deviceName)
```

The slider on the user interface is synchronized with the specified control on your device. Move one to see the other respond.

Input Arguments

`midicontrolsObject` — Object that listens to the controls on a MIDI device
object

Object that listens to the controls on a MIDI device, specified as an object created by `midicontrols`.

`controlValues` — Values sent to MIDI device
initial values specified by `midicontrolsObject` (default) | scalar | array

Values sent to MIDI device, specified as a scalar or an array the same size as `controlNumbers` of the associated `midicontrols` object. If you do not specify `controlValues`, the default value is the `initialValues` of the associated `midicontrols` object.

The possible range for `controlValues` depends on the `OutputMode` of the associated `midicontrols` object.

- If `OutputMode` is specified as `'normalized'`, then `controlValues` must consist of values in the range `[0,1]`. The default `OutputMode` is `'normalized'`.
- If `OutputMode` is specified as `'rawmidi'`, then `controlValues` must consist of integer values in the range `[0,127]`.

Example: `0.3`

Example: `[0,0.3,0.6]`

Example: `5`

Example: `[5;15;20]`

Data Types: `double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

See Also

Functions

`configureMIDI` | `disconnectMIDI` | `getMIDIConnections` | `midicallback` | `midicontrols` | `midiid` | `midiread` | `setpref`

Topics

"MIDI Control Surface Interface"
"MIDI Control for Audio Plugins"

validateAudioPlugin

Test MATLAB source code for audio plugin

Syntax

```
validateAudioPlugin classname
validateAudioPlugin options classname
```

Description

`validateAudioPlugin classname` generates and runs a "Test Bench Procedure" on page 2-159 that exercises your audio plugin class.

`validateAudioPlugin options classname` specifies options to modify the default "Test Bench Procedure" on page 2-159.

Examples

Validate Audio Plugin

```
validateAudioPlugin audiopluginexample.Echo
```

```
Checking plug-in class 'audiopluginexample.Echo'... passed.
Generating testbench file 'testbench_Echo.m'... done.
Running testbench... passed.
Generating mex file 'testbench_Echo_mex.mexw64'... done.
Running mex testbench... passed.
Deleting testbench.
Ready to generate audio plug-in.
```

Skip MEX Version of Test Bench

```
validateAudioPlugin -nomex audiopluginexample.Echo
```

```
Checking plug-in class 'audiopluginexample.Echo'... passed.
Generating testbench file 'testbench_Echo.m'... done.
Running testbench... passed.
Skipping mex.
Deleting testbench.
```

Keep Test Benches After Validation

```
validateAudioPlugin -keeptestbench audiopluginexample.Echo
```

```
Checking plug-in class 'audiopluginexample.Echo'... passed.
Generating testbench file 'testbench_Echo.m'... done.
Running testbench... passed.
Generating mex file 'testbench_Echo_mex.mexw64'... done.
Running mex testbench... passed.
Keeping testbench.
Ready to generate audio plug-in.
```

Two test benches are saved to your current folder:

- `testbench_Echo.m`
- `testbench_Echo_mex.mexw64`

Skip MEX Version and Keep Test Bench

```
validateAudioPlugin -keeptestbench -nomex audiopluginexample.Echo
```

```
Checking plug-in class 'audiopluginexample.Echo'... passed.
Generating testbench file 'testbench_Echo.m'... done.
Running testbench... passed.
Skipping mex.
Keeping testbench.
```

One test bench is saved to your current folder:

- `testbench_Echo.m`

Input Arguments

`options` — Options to modify test bench procedure
`-nomex` | `-keeptestbench`

Options to modify test bench procedure, specified as `-nomex` or `-keeptestbench`. Options can be specified together or separately, and in any order.

- `-nomex` –– `validateAudioPlugin` does not generate and run a MEX version of the test bench file. This option significantly reduces run time of the test bench procedure.
- `-keeptestbench` –– `validateAudioPlugin` saves the generated test benches to the current folder.

`classname` — Name of the plugin class to validate
plugin class

Name of the plugin class to validate. The plugin class must derive from either the `audioPlugin` class or the `audioPluginSource` class. The `validateAudioPlugin` function exercises an instance of the specified plugin class.

You can specify the plugin class to validate by specifying its class name or file name. For example, the following syntaxes perform equivalent operations:

- `validateAudioPlugin myPlugin`
- `validateAudioPlugin myPlugin.m`

If you want to specify the plugin class by file name, and your plugin class is inside a package, you must specify the package as a file path. For example, the following syntaxes perform equivalent operations:

- `validateAudioPlugin myPluginPackage.myPlugin`
- `validateAudioPlugin +myPluginPackage/myPlugin.m`

Limitations

The `valdiateAudioPlugin` function is compatible with Windows and Mac operating systems. It is not compatible with Linux.

Definitions

Test Bench Procedure

The `valudateAudioPlugin` function uses dynamic testing to find common audio plugin programming mistakes not found by the static checks performed by `generateAudioPlugin`. The function:

1. Runs a subset of error checks performed by `generateAudioPlugin`.
2. Generates and runs a MATLAB test bench to exercise the class.
3. Generates and runs a MEX version of the test bench.
4. Removes the generated test benches.

If the plugin class fails testing, step 4 is automatically omitted. To debug your plugin, step through the saved generated test bench.

If you use the `-keeptestbench` option, or if an error occurs during validation, the test bench files are saved to your current folder.

See Also

Functions
`generateAudioPlugin`

Classes
`audioPlugin` | `audioPluginSource`

Topics

"Design an Audio Plugin"

Introduced in R2016a

3

System objects in Audio System Toolbox

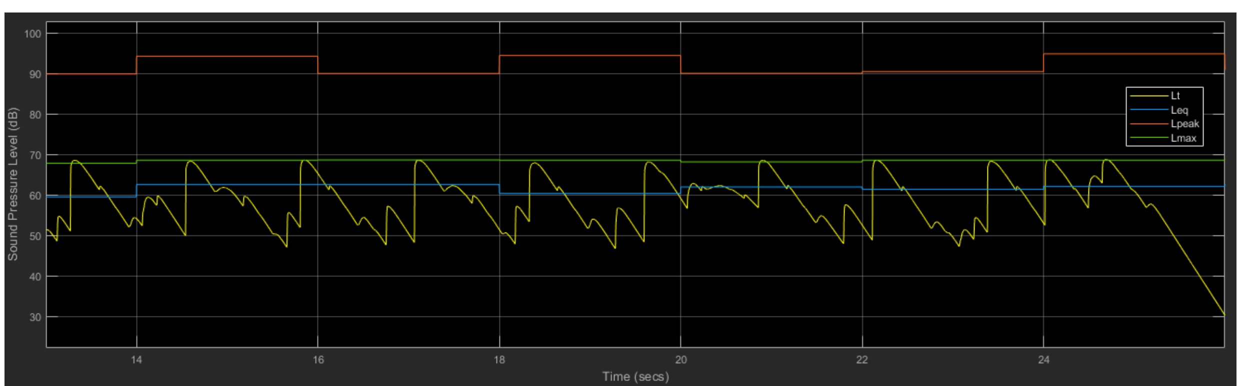
splMeter System object

Measure sound pressure level of audio signal

Description

The `splMeter` System object computes sound pressure level measurements. The object returns measurements for:

- frequency-weighted sound levels
- fast or slow time-weighted sound levels
- equivalent-continuous sound levels
- peak sound levels
- maximum sound levels



To implement SPL metering:

1. Create the splMeter object and set its properties.
2. Call the object with arguments, as if it were a function.

To learn more about how System objects work, see What Are System Objects? (MATLAB).

Creation

Syntax

```
SPL = splMeter
SPL = splMeter(Name,Value)
```

Description

`SPL = splMeter` creates a System object, `SPL`, that performs SPL metering.

`SPL = splMeter(Name,Value)` sets each property `Name` to the specified `Value`. Unspecified properties have default values.

Example: `SPL = splMeter('FrequencyWeighting','C-weighting','SampleRate',12000)` creates a System object, `SPL`, that performs C-weighting and operates at 12 kHz.

Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see System Design in MATLAB Using System Objects (MATLAB).

`FrequencyWeighting` — Frequency weighting applied to input
`'A-weighting'` (default) | `'C-weighting'` | `'Z-weighting'` (no weighting)

Frequency weighting applied to input, specified as `'A-weighting'`, `'C-weighting'`, or `'Z-weighting'`, where Z-weighting corresponds to no weighting. The frequency weighting is designed and implemented using the `weightingFilter` System object.

Tunable: No

Data Types: `char` | `string`

`TimeWeighting` — Time weighting (s)
'Fast' (default) | 'Slow'

Time weighting, in seconds, for calculation of time-weighted sound level and maximum time-weighted sound level, specified as `'Fast'` or `'Slow'`. The `TimeWeighting` property is used to specify the coefficient of a lowpass filter.

- `'Fast'` – 1/8
- `'Slow'` – 1

See “Time-Weighted Sound Level” on page 3-12 for more details.

Tunable: Yes

Data Types: `char` | `string`

`PressureReference` — Reference pressure for dB calculations (Pa)
2e-5 (default) | positive scalar

Reference pressure for dB calculations in Pa, specified as a positive scalar.

Tunable: Yes

Data Types: `single` | `double`

`TimeInterval` — Time interval for reporting level measurements (s)
1 (default) | positive scalar

Time interval, in seconds, to report equivalent-continuous, peak, and maximum time-weighted sound levels, specified as a positive scalar integer.

Tunable: No

Data Types: `single` | `double`

`CalibrationFactor` — Scalar calibration factor multiplied by input
1 (default) | positive finite scalar

Scalar calibration factor multiplied by input.

To set the calibration factor using a reference tone, use `calibrate`.

Tunable: No

Data Types: `single` | `double`

SampleRate — Input sample rate (Hz)
44100 (default) | positive scalar

Input sample rate in Hz, specified as a positive scalar.

Tunable: No

Data Types: single | double

Usage

Syntax

```
[Lt,Leq,Lpeak,Lmax] = SPL(audioIn)
```

Description

`[Lt,Leq,Lpeak,Lmax] = SPL(audioIn)` returns measurement values for the time-weighted (`Lt`) sound level of the current input frame, `audioIn`. The object also returns the equivalent-continuous (`Leq`), peak (`Lpeak`), and maximum time-weighted (`Lmax`) sound levels of the input to your SPL meter.

Input Arguments

audioIn — Audio input to SPL meter
column vector

Audio input to the SPL meter, specified as a column vector.

Data Types: single | double

Output Arguments

Lt — Time-weighted sound level (dB)
column vector

Time-weighted sound level in dB, returned as a column vector the same size and type as `audioIn`.

Data Types: `single` | `double`

Leq — Equivalent-continuous sound level (dB)
column vector

Equivalent-continuous sound level in dB, returned as a column vector the same size and type as `audioIn`.

Data Types: `single` | `double`

Lpeak — Peak sound level (dB)
column vector

Peak sound level in dB, returned as a column vector the same size and type as `audioIn`.

Data Types: `single` | `double`

Lmax — Maximum time-weighted sound level (dB)
column vector

Maximum time-weighted sound level in dB, returned as a column vector the same size and type as `audioIn`.

Data Types: `single` | `double`

Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

Specific to splMeter

calibrate	Calibrate meter using calibration tone with known level

Common to All System Objects

clone	Create duplicate System object
isLocked	Determine if System object is in use
release	Release resources and allow changes to System object property values and input characteristics

reset	Reset internal states of System object
step	Run System object algorithm

Examples

Measure SPL of Audio Signal

Use the `splMeter` System object™ to measure the A-weighted sound pressure level of a streaming audio signal. Specify a two second time-interval for reporting and a fast time-weighting. Visualize the SPL measurements using the `dsp.TimeScope` System object.

Create a `dsp.AudioFileReader` object to read in an audio file frame by frame. Create an `audioDeviceWriter` object to listen to the audio signal. Create a `dsp.TimeScope` object to visualize SPL measurements. Create an `splMeter` to measure the sound pressure level of the audio file. Use the default calibration factor of 1.

```
source = dsp.AudioFileReader('Ambiance-16-44p1-mono-12secs.wav');
fs = source.SampleRate;

player = audioDeviceWriter('SampleRate',fs);

scope  = dsp.TimeScope('SampleRate',fs, ...
    'TimeSpanOverrunAction','Scroll', ...
    'TimeSpan',3,'ShowGrid',true, ...
    'YLimits',[20 110],'AxesScaling','Auto', ...
    'ShowLegend',true,'BufferLength',4*3*fs, ...
    'ChannelNames', ...
    {'Lt_AF','Leq_A','Lpeak_A','Lmax_AF'}, ...
    'Name','Sound Pressure Level Meter');

SPL = splMeter('TimeWeighting','Fast', ...
    'FrequencyWeighting','A-weighting', ...
    'SampleRate',fs, ...
    'TimeInterval',2);
```

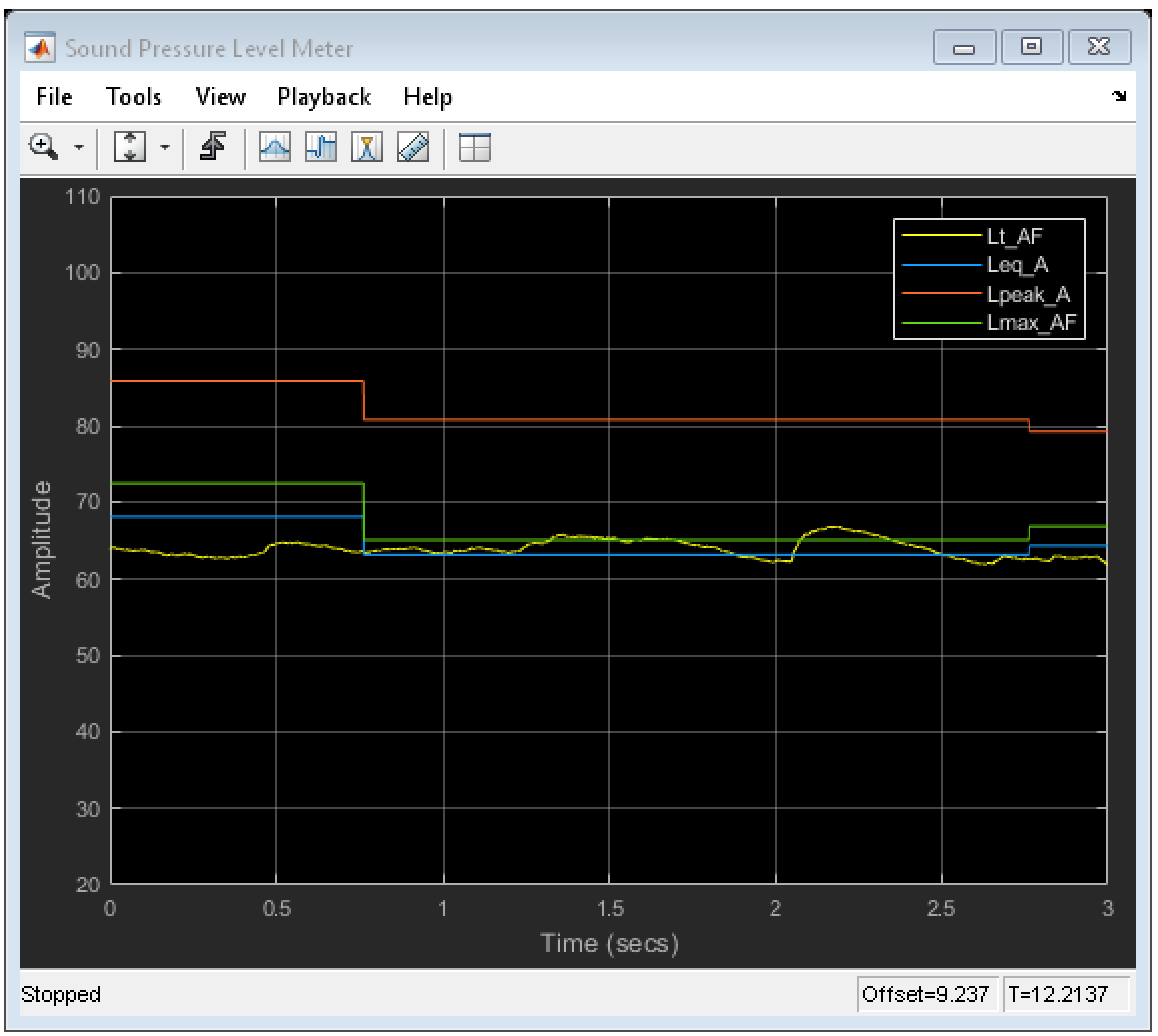
In an audio stream loop:

1. Read in the audio signal frame.
2. Play the audio signal to your output device.
3. Call the SPL meter to return the time-weighted, equivalent-continuous, peak, and maximum time-weighted sound levels in dB.

4 Display the sound levels using the scope.

As a best practice, release your objects once complete.

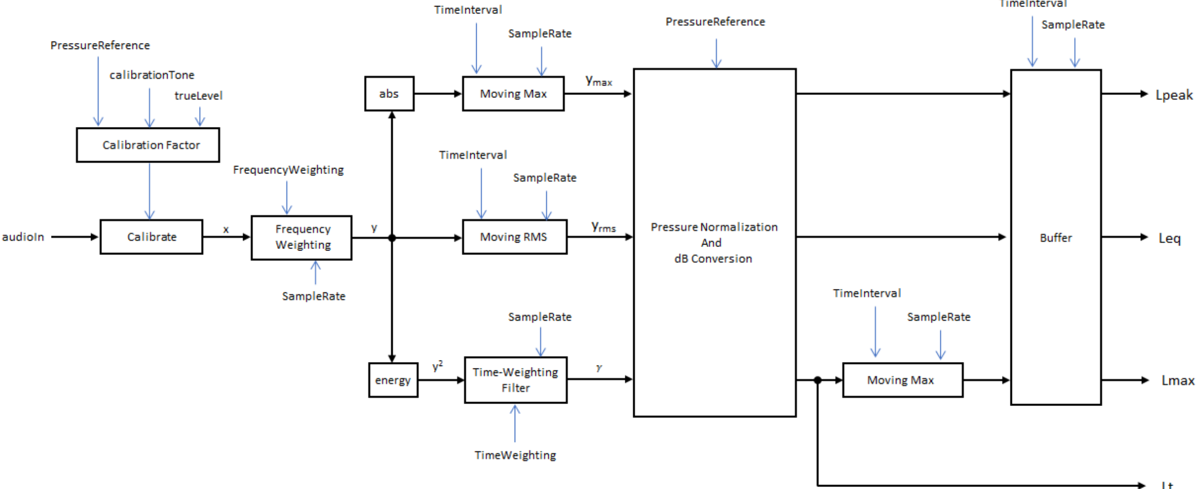
```
while ~isDone(source)
    x = source();
    player(x);
    [Lt,Leq,Lpeak,Lmax] = SPL(x);
    scope([Lt,Leq,Lpeak,Lmax])
end

release(source)
release(player)
release(SPL)
release(scope)
```



- “Sound Pressure Measurement Using Weighting Filters”

Algorithms

Sound pressure level calculations follow the algorithms described in [1]. You can specify property values to conform to standards [2] and [3].



Calibration

To account for environmental and input device effects in SPL measurements, the audio input is multiplied by a calibration factor:

$$x = \mathtt{audioIn} \times \mathtt{CalibrationFactor}$$

The `CalibrationFactor` property can be set directly, or by using the `calibrate` function, which compares a known level with acquired data. The known level is determined using a physical calibrator.

Frequency Weighting

A-, C-, or Z-frequency weighting is applied. The frequency weighting is implemented using the `weightingFilter` System object.

Peak Sound Level

Peak sound level is defined as 20 times the logarithm to the base 10 of the ratio of peak sound pressure to the reference sound pressure. That is,

$$Lpeak = 20\log_{10}\left(\max\left(|y|\right) / p_o\right)$$

where

- y is the output of the frequency-weighting filter.
- p_o is the reference sound pressure, specified by the `PressureReference` property.

The moving maximum is determined using `dsp.MovingMaximum` with window length defined by the TimeInterval and SampleRate properties: `WindowLength = round(TimeInterval*SampleRate)`.

A buffer enables you to specify variable input frame sizes while maintaining a measurement update rate specified by the `TimeInterval` and `SampleRate` properties.

Equivalent-Continuous Sound Level

Equivalent-continuous sound level, also called time-average sound level, is defined as 20 times the logarithm to the base 10 of the ratio of root-mean-square sound pressure to the reference sound pressure. That is,

$$\begin{aligned} Leq &= 10\log_{10}\left\{\frac{\left(1/T\right)\int_{t_1}^{t_2} y^2\,dt}{p_o^2}\right\} \\ &= 20\log_{10}\left(\mathrm{rms}\left(y\right) / p_o\right) \end{aligned}$$

where

- y is the output of the frequency-weighting filter.
- p_o is the reference sound pressure, specified by the PressureReference property.

The moving root-mean-square is determined using `dsp.MovingRMS` with window length defined by the `TimeInterval` and `SampleRate` properties: `WindowLength = round(TimeInterval*SampleRate)`.

A buffer enables you to specify variable input frame sizes while maintaining a measurement update rate specified by the `TimeInterval` and `SampleRate` properties.

Time-Weighted Sound Level

Time-weighted sound level is defined as 20 times the logarithm to the base 10 of the ratio of the time-weighted root-mean-square sound pressure to the reference sound pressure.

$$Lt = 10\log_{10}\left\{\frac{\left(1/\tau\right)\int_{t_s}^{t} y(\xi)^2 e^{-(t-\xi)/\tau} d\xi}{p_o^2}\right\}$$

$$= 10\log_{10}\left\{\frac{h(y^2)}{p_o^2}\right\}$$

$h(y^2)$ can be interpreted as the convolution of y^2 with a filter with impulse response $\left(1/\tau\right) e^{-t/\tau}$. y is the output of the frequency-weighting filter. The impulse response corresponds to a lowpass filter of the form $H(s) = \frac{1/\tau}{s + 1/\tau}$. Using impulse invariance, the discrete filter can be interpreted as,

$$H(z) = \frac{1/(\tau \times fs)}{1 - e^{-1/(\tau \times fs)} z^{-1}} .$$

The time-weighting filter is implemented using `dsp.BiquadFilter`, with coefficients as specified by the discrete transfer function, $H(z)$.

- τ is specified by the time-weighting coefficient as 0.125 (if `TimeWeighting` is set to `'Fast'`) or 1 (if `TimeWeighting` is set to `'Slow'`).

- *fs* is the sample rate specified by the `SampleRate` property.

Max Time-Weighted Sound Level

Maximum time-weighted sound level is defined as the greatest time-weighted sound level within a stated time interval.

The moving maximum is determined using `dsp.MovingMaximum` with window length defined by the `TimeInterval` and `SampleRate` properties: `WindowLength = round(TimeInterval*SampleRate)`.

A buffer enables you to specify variable input frame sizes while maintaining a measurement update rate specified by the `TimeInterval` and `SampleRate` properties.

References

[1] Harris, Cyril M. *Handbook of Acoustical Measurements and Noise Control*. 3rd ed. American Institute of Physics, 1998.

[2] International Electrotechnical Commission. Electroacoustics - Sound level meters - Part 1: Specifications. IEC 61672-1:2013.

[3] American National Standards Institute. ANSI S1.4: Specification for Sound Level Meters. 1983.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

“System Objects in MATLAB Code Generation” (MATLAB Coder)

See Also

System Objects
`loudnessMeter`

Blocks
Loudness Meter

Functions
`integratedLoudness`

Topics
"Sound Pressure Measurement Using Weighting Filters"

Introduced in R2018a

calibrate

Calibrate meter using calibration tone with known level

Syntax

`calibrate(SPL,calibrationTone,trueLevel)`

Description

`calibrate(SPL,calibrationTone,trueLevel)` sets the `CalibrationFactor` property based on the computed sound pressure level of `calibrationTone` and the known `trueLevel`. `trueLevel` refers to the physical calibrator level used to generate the `calibrationTone`.

Input Arguments

`SPL` — `splMeter` System object
object

`splMeter` System object to be calibrated.

`calibrationTone` — Audio signal used to calibrate SPL meter
column vector

Audio signal used to calibrate the SPL meter, specified as a column vector.

Data Types: `single` | `double`

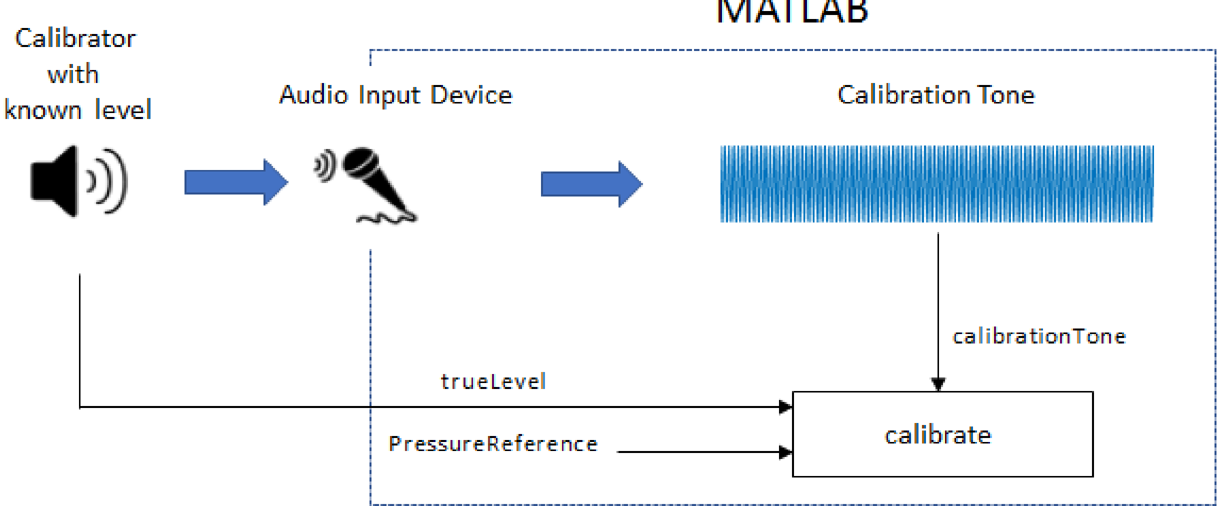
`trueLevel` — True level of calibration tone (dB)
scalar

True level of calibration tone in dB, specified as a scalar. The true level is the known level of output by a physical calibrator.

Data Types: `single` | `double`

Algorithms

To set the `CalibrationFactor` property on an `splMeter` object, the `calibrate` function uses a calibration tone, the known level output by the calibrator to produce the calibration tone, and the `PressureReference` property.



The `CalibrationFactor` property is set according to the equation:

$$\texttt{CalibrationFactor} = \frac{10^{((\texttt{trueLevel} - k)/20)}}{\texttt{rms}(\texttt{calibrationTone})}$$

where k is 1 pascal relative to the reference pressure calculated in dB:

$$k = 20\log_{10}\left(\frac{1}{\texttt{PressureReference}}\right).$$

See Also

System Objects
`splMeter`

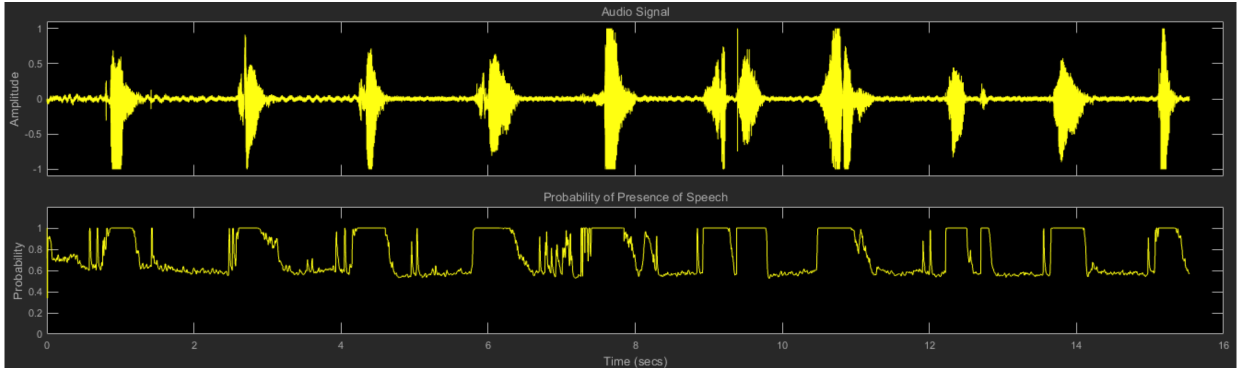
Introduced in R2018a

voiceActivityDetector System object

Detect presence of speech in audio signal

Description

The `voiceActivityDetector` System object detects the presence of speech in an audio segment. You can also use the `voiceActivityDetector` System object to output an estimate of the noise variance per frequency bin.



To detect the presence of speech:

1. Create the voiceActivityDetector object and set its properties.
2. Call the object with arguments, as if it were a function.

To learn more about how System objects work, see What Are System Objects? (MATLAB).

Creation

Syntax

```
VAD = voiceActivityDetector
```

```
VAD = voiceActivityDetector(Name,Value)
```

Description

`VAD = voiceActivityDetector` creates a System object, `VAD`, that detects the presence of speech independently across each input channel.

`VAD = voiceActivityDetector(Name,Value)` sets each property `Name` to the specified `Value`. Unspecified properties have default values.

Example: `VAD = voiceActivityDetector('InputDomain','Frequency')` creates a System object, `VAD`, that accepts frequency-domain input.

Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see System Design in MATLAB Using System Objects (MATLAB).

`InputDomain` — Domain of input signal

`'Time'` (default) | `'Frequency'`

Domain of the input signal, specified as `'Time'` or `'Frequency'`.

Tunable: No

Data Types: `char` | `string`

`FFTLength` — FFT length

`[]` (default) | positive scalar

FFT length, specified as a positive scalar. The default is `[]`, which means that the `FFTLength` is equal to the number of rows of the input.

Tunable: No

Dependencies

To enable this property, set InputDomain to `'Time'`.

Data Types: `single` | `double`

`Window` — Window function for FFT

`'Hann'` (default) | `'Chebyshev'` | `'Flat Top'` | `'Hamming'` | `'Kaiser'` | `'Rectangular'`

Time-domain window function applied before calculating the discrete-time Fourier transform (DTFT), specified as `'Hann'`, `'Rectangular'`, `'Flat Top'`, `'Hamming'`, `'Chebyshev'`, or `'Kaiser'`.

The window function is designed using the algorithms of the following functions:

- Hann -- `hann`
- Chebyshev -- `chebwin`
- Flat Top -- `flattopwin`
- Hamming -- `hamming`
- Kaiser -- `kaiser`

Tunable: No

Dependencies

To enable this property, set InputDomain to `'Time'`.

Data Types: `char` | `string`

`SidelobeAttenuation` — Sidelobe attenuation of window (dB)

`60` (default) | real positive scalar

Sidelobe attenuation of the window in dB, specified as a real positive scalar.

Tunable: No

Dependencies

To enable this property, set InputDomain to `'Time'` and Window to `'Chebyshev'` or `'Kaiser'`.

Data Types: `single` | `double`

`SilenceToSpeechProbability` — Probability of transition from a frame of silence to a frame of speech
`0.2` (default) | scalar in the range [0,1]

Probability of transition from a frame of silence to a frame of speech, specified as a scalar in the range [0,1].

Tunable: Yes

Data Types: `single` | `double`

`SpeechToSilenceProbability` — Probability of transition from a frame of speech to a frame of silence
`0.1` (default) | scalar in the range [0,1]

Probability of transition from a frame of speech to a frame of silence, specified as a scalar in the range [0,1].

Tunable: Yes

Data Types: `single` | `double`

Usage

Syntax

```
[probability,noiseEstimate] = VAD(audioIn)
```

Description

`[probability,noiseEstimate] = VAD(audioIn)` applies a voice activity detector on the input, `audioIn`, and returns the probability that speech is present. It also returns the estimated noise variance per frequency bin.

Input Arguments

`audioIn` — Audio input to voice activity detector
scalar | vector | matrix

Audio input to the voice activity detector, specified as a scalar, vector, or matrix. If `audioIn` is a matrix, the columns are treated as independent audio channels.

The size of the audio input is locked after the first call to the `voiceActivityDetector` object. To change the size of `audioIn`, call `release` on the object.

If InputDomain is set to `'Time'`, `audioIn` must be real-valued. If `InputDomain` is set to `'Frequency'`, `audioIn` can be real-valued or complex-valued.

Data Types: `single` | `double`
Complex Number Support: Yes

Output Arguments

`probability` — Probability that speech is present
scalar | row vector

Probability that speech is present, returned as a scalar or row vector with the same number of columns as `audioIn`.

Data Types: `single` | `double`

`noiseEstimate` — Estimate of noise variance per frequency bin
column vector | matrix

Estimate of the noise variance per frequency bin, returned as a column vector or matrix with the same number of columns as `audioIn`.

Data Types: `single` | `double`

Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

Common to All System Objects

clone	Create duplicate System object
isLocked	Determine if System object is in use

release	Release resources and allow changes to System object property values and input characteristics
reset	Reset internal states of System object
step	Run System object algorithm

Examples

Detect Voice Activity

Use the default `voiceActivityDetector` System object™ to detect the presence of speech in a streaming audio signal.

Create an audio file reader to stream an audio file for processing. Define parameters to chunk the audio signal into 10 ms non-overlapping frames.

```
fileReader = dsp.AudioFileReader('Counting-16-44p1-mono-15secs.wav');
fs = fileReader.SampleRate;
fileReader.SamplesPerFrame = ceil(10e-3*fs);
```

Create a default `voiceActivityDetector` System object to detect the presence of speech in the audio file.

```
VAD = voiceActivityDetector;
```

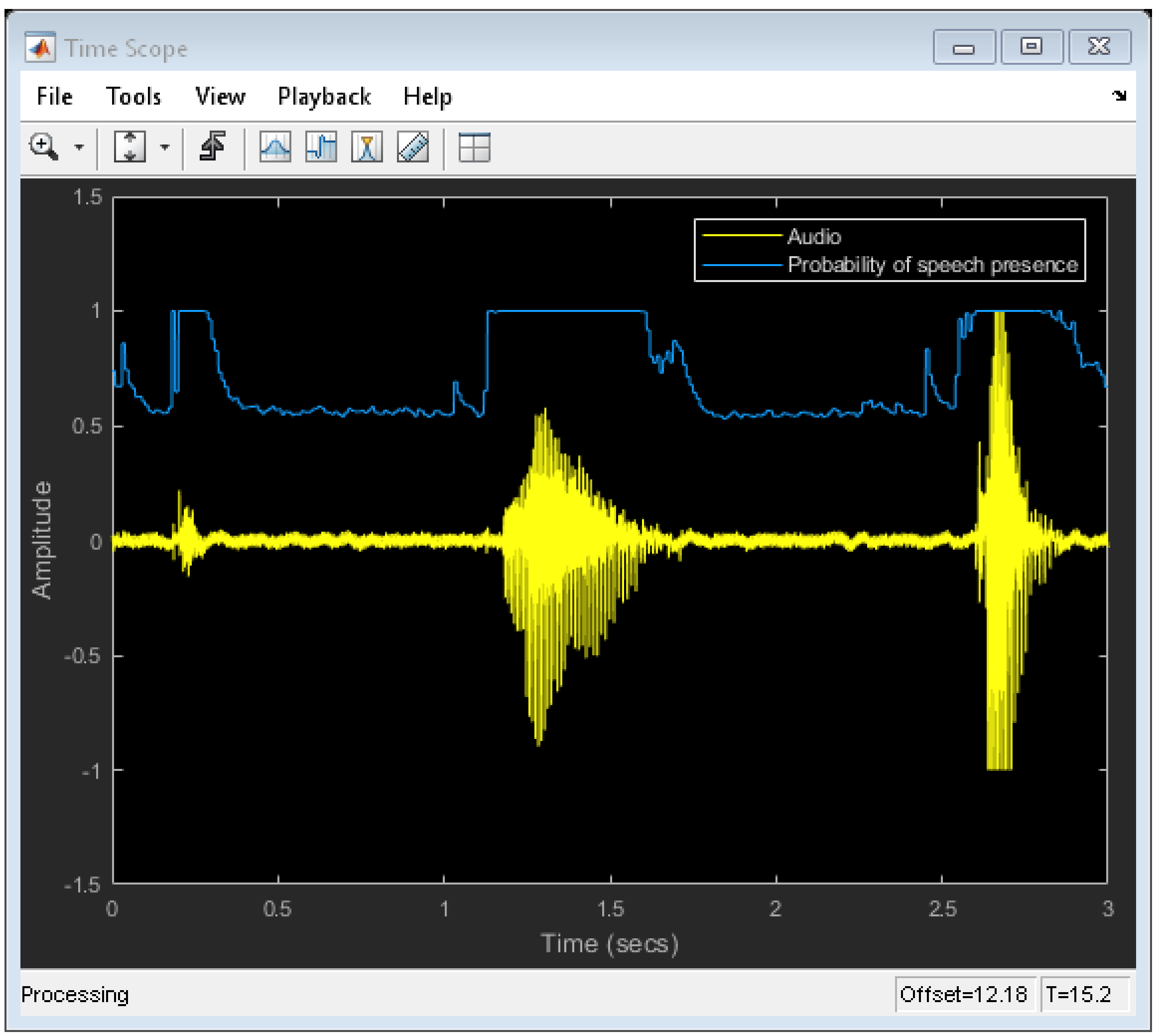
Create a scope to plot the audio signal and corresponding probability of speech presence as detected by the voice activity detector. Create an audio device writer to play the audio through your sound card.

```
scope = dsp.TimeScope( ...
    'NumInputPorts',2, ...
    'SampleRate',fs, ...
    'TimeSpan',3, ...
    'BufferLength',3*fs, ...
    'YLimits',[-1.5 1.5], ...
    'TimeSpanOverrunAction','Scroll', ...
    'ShowLegend',true, ...
    'ChannelNames',{'Audio','Probability of speech presence'});
deviceWriter = audioDeviceWriter('SampleRate',fs);
```

In an audio stream loop:

1. Read from the audio file.
2. Calculate the probability of speech presence.
3. Visualize the audio signal and speech presence probability.
4. Play the audio signal through your sound card.

```
while ~isDone(fileReader)
    audioIn = fileReader();
    probability = VAD(audioIn);
    scope(audioIn,probability*ones(fileReader.SamplesPerFrame,1))
    deviceWriter(audioIn);
end
```



Detect Voice Activity Using Overlapped Frames

Use a voice activity detector to detect the presence of speech in an audio signal. Plot the probability of speech presence along with the audio samples.

Create a `dsp.AudioFileReader` System object™ to read a speech file.

```
afr = dsp.AudioFileReader('Counting-16-44p1-mono-15secs.wav');
fs = afr.SampleRate;
```

Chunk the audio into 20 ms frames with 75% overlap between successive frames. Convert the frame time in seconds to samples. Determine the hop size (the increment of new samples). In the audio file reader, set the samples per frame to the hop size. Create a default `dsp.AsyncBuffer` object to manage overlapping between audio frames.

```
frameSize = ceil(20e-3*fs);
overlapSize = ceil(0.75*frameSize);
hopSize = frameSize - overlapSize;
afr.SamplesPerFrame = hopSize;

inputBuffer = dsp.AsyncBuffer('Capacity',frameSize);
```

Create a `voiceActivityDetector` System object. Specify an FFT length of 1024.

```
VAD = voiceActivityDetector('FFTLength',1024);
```

Create a scope to plot the audio signal and corresponding probability of speech presence as detected by the voice activity detector. Create an `audioDeviceWriter` System object to play audio through your sound card.

```
scope = dsp.TimeScope('NumInputPorts',2, ...
    'SampleRate',fs, ...
    'TimeSpan',3, ...
    'BufferLength',3*fs, ...
    'YLimits',[-1.5,1.5], ...
    'TimeSpanOverrunAction','Scroll', ...
    'ShowLegend',true, ...
    'ChannelNames',{'Audio','Probability of speech presence'});

player = audioDeviceWriter('SampleRate',fs);
```

Initialize a vector to hold the probability values.

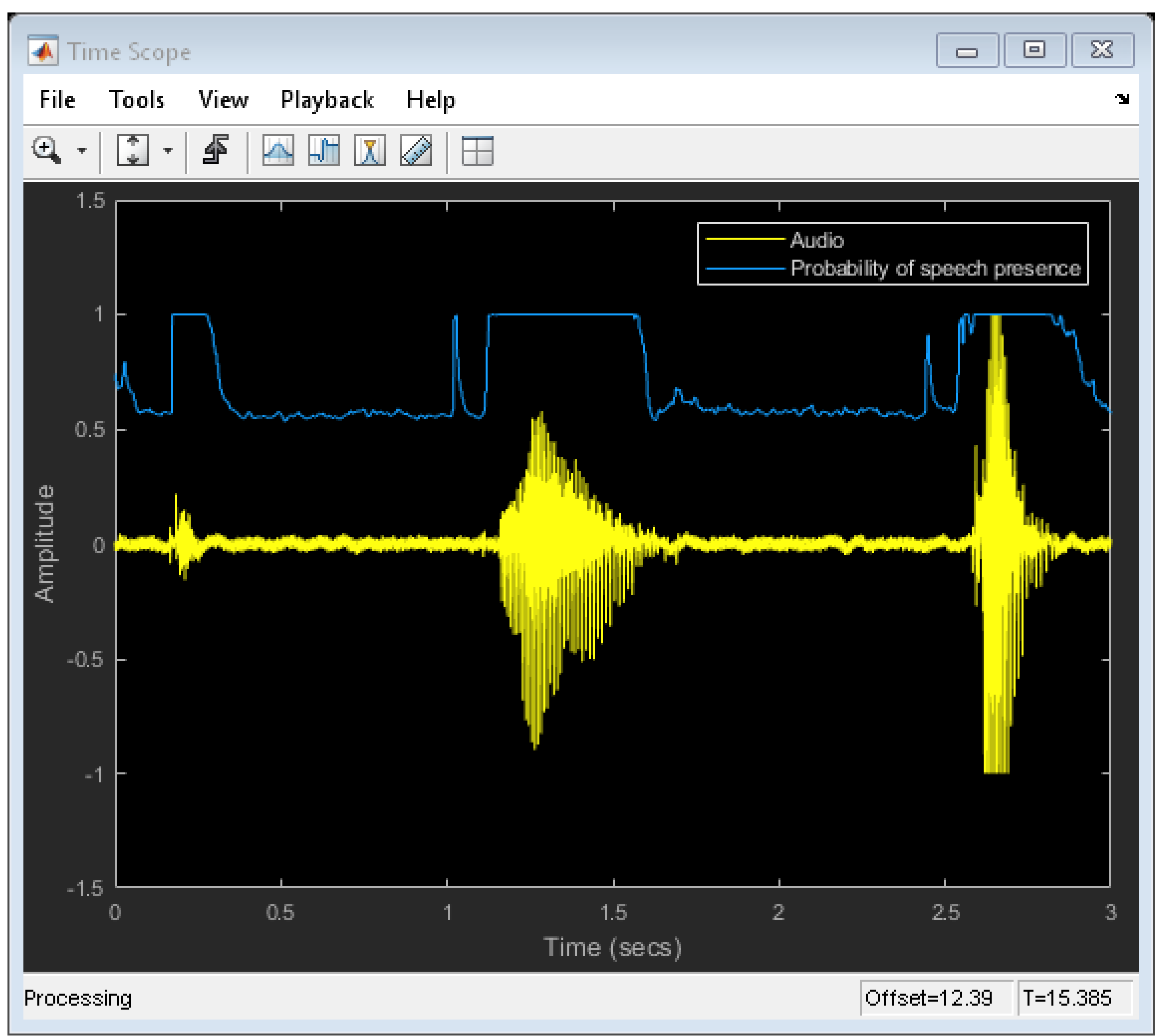
```
pHold = ones(hopSize,1);
```

In an audio stream loop:

1. Read a hop worth of samples from the audio file and save the samples into the buffer.
2. Read a frame from the buffer with specified overlap from the previous frame.

3. Call the voice activity detector to get the probability of speech for the frame under analysis.
4. Set the last element of the probability vector to the new probability decision. Visualize the audio and speech presence probability using the time scope.
5. Play the audio through your sound card.
6. Set the probability vector to the most recent result for plotting in the next loop.

```
while ~isDone(afr)
    x = afr();
    n = write(inputBuffer,x);

    overlappedInput = read(inputBuffer,frameSize,overlapSize);

    p = VAD(overlappedInput);

    pHold(end) = p;
    scope(x,pHold)

    player(x);

    pHold(:) = p;
end
```



Release the `player` once the audio finishes playing.

```
release(player)
```

Frequency-Domain Voice Activity Detection and Cepstral Feature Extraction

Many feature extraction techniques operate on the frequency domain. Converting an audio signal to the frequency domain only once is efficient. In this example, you convert a streaming audio signal to the frequency domain and feed that signal into a voice activity detector. If speech is present, mel-frequency cepstral coefficients (MFCC) features are extracted from the frequency-domain signal using the `cepstralFeatureExtractor System object™`.

Create a `dsp.AudioFileReader` System object to read from an audio file.

```
fileReader = dsp.AudioFileReader('Counting-16-44p1-mono-15secs.wav');
fs = fileReader.SampleRate;
```

Process the audio in 30 ms frames with a 10 ms hop. Create a default `dsp.AsyncBuffer` object to manage overlap between audio frames.

```
samplesPerFrame = ceil(0.03*fs);
samplesPerHop = ceil(0.01*fs);
samplesPerOverlap = samplesPerFrame - samplesPerHop;

fileReader.SamplesPerFrame = samplesPerHop;
buffer = dsp.AsyncBuffer;
```

Create a `voiceActivityDetector` System object and a `cepstralFeatureExtractor` System object. Specify that they operate in the frequency domain. Create a `dsp.SignalSink` to log the extracted cepstral features.

```
VAD = voiceActivityDetector('InputDomain','Frequency');
cepFeatures = cepstralFeatureExtractor('InputDomain','Frequency','SampleRate',fs,'LogE
sink = dsp.SignalSink;
```

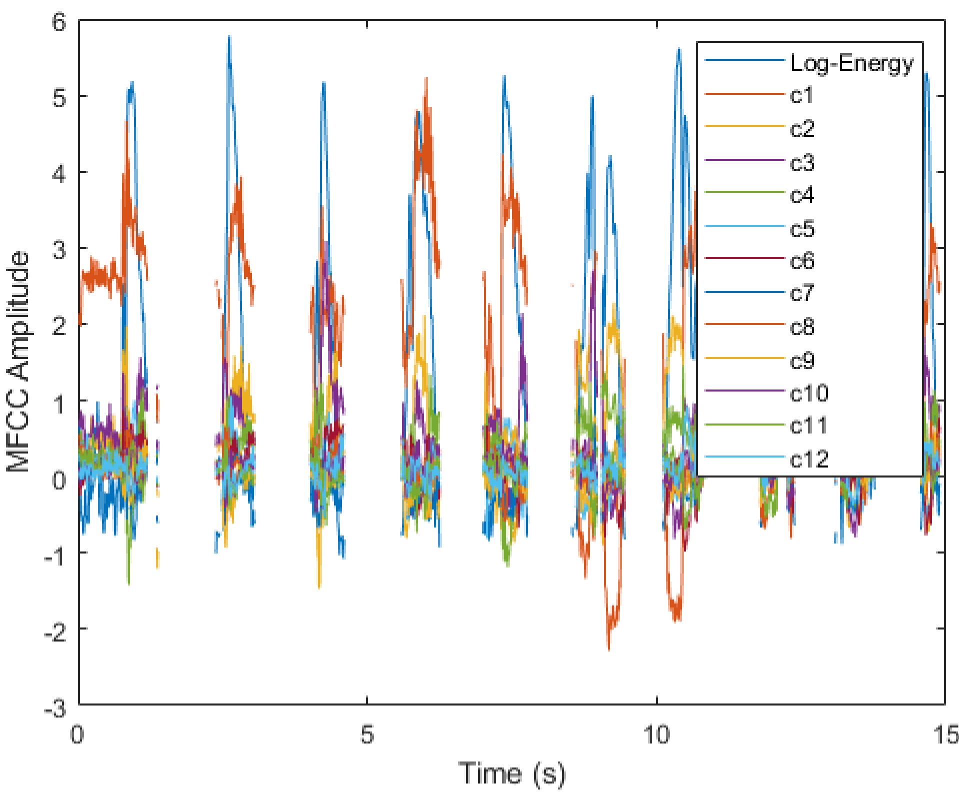
In an audio stream loop:

1. Read one hop's of samples from the audio file and save the samples into the buffer.
2. Read a frame from the `buffer` with specified overlap from the previous frame.
3. Call the voice activity detector to get the probability of speech for the frame under analysis.
4. If the frame under analysis has a probability of speech greater than 0.75, extract cepstral features and log the features using the signal sink. If the frame under analysis has a probability of speech less than 0.75, write a vector of NaNs to the sink.

```
threshold = 0.75;
nanVector = nan(1,13);
while ~isDone(fileReader)
    audioIn = fileReader();
    write(buffer,audioIn);

    overlappedAudio = read(buffer,samplesPerFrame,samplesPerOverlap);
    X = fft(overlappedAudio,2048);

    probabilityOfSpeech = VAD(X);
    if probabilityOfSpeech > threshold
        xFeatures = cepFeatures(X);
        sink(xFeatures')
    else
        sink(nanVector)
    end
end
```

Visualize the cepstral coefficients over time.

```
timeVector = linspace(0,15,size(sink.Buffer,1));
plot(timeVector,sink.Buffer)
xlabel('Time (s)')
ylabel('MFCC Amplitude')
legend('Log-Energy','c1','c2','c3','c4','c5','c6','c7','c8','c9','c10','c11','c12')
```



Determine Pitch Contour using `pitch` and `voiceActivityDetector`

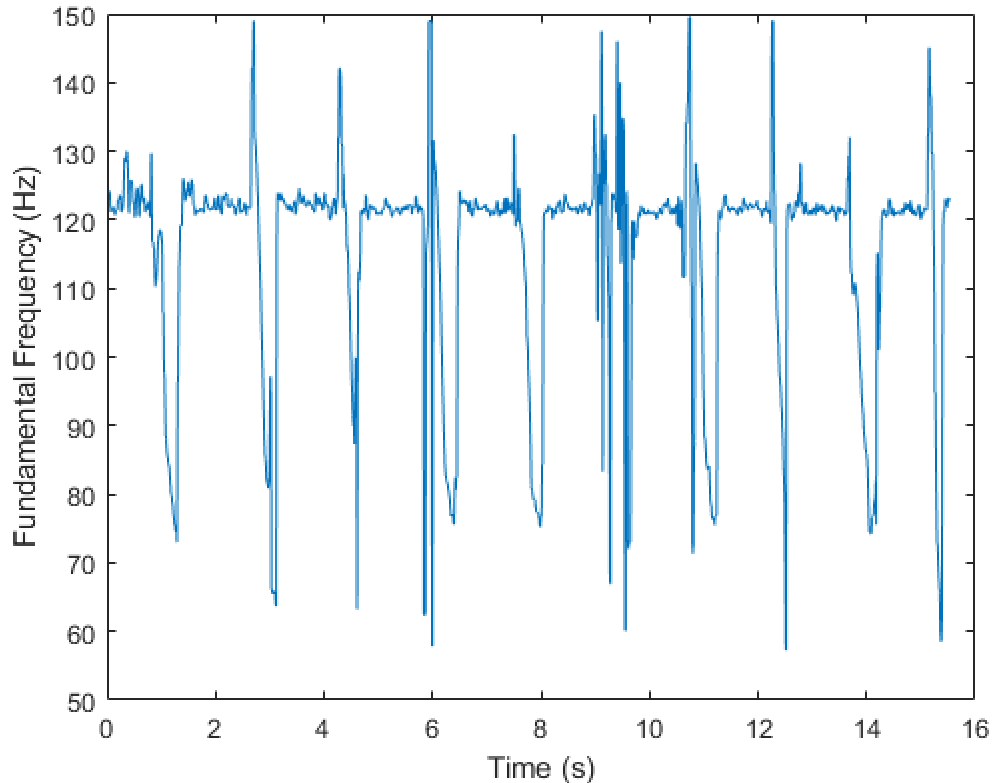
Read in an entire speech file and determine the fundamental frequency of the audio using the `pitch` function. Then use the `voiceActivityDetector` to remove irrelevant pitch information that does not correspond to the speaker.

Read in the audio file and associated sample rate.

```
[audio,fs] = audioread('Counting-16-44p1-mono-15secs.wav');
```

Specify pitch detection using a 50 ms window length and 40 ms overlap (10 ms hop). Specify that the `pitch` function searches for the fundamental frequency over the range 50-150 Hz and postprocesses the results with a median filter. Plot the results.

```
windowLength = round(0.05*fs);
overlapLength = round(0.04*fs);
hopLength = windowLength - overlapLength;

[f0,loc] = pitch(audio,fs, ...
    'WindowLength',windowLength, ...
    'OverlapLength',overlapLength, ...
    'Range',[50 150], ...
    'MedianFilterLength',3);

plot(loc/fs,f0)
ylabel('Fundamental Frequency (Hz)')
xlabel('Time (s)')
```



Create a `dsp.AsyncBuffer` System object™ to chunk the audio signal into overlapped frames. Also create a `voiceActivityDetector` System object™ to determine if the frames contain speech.

```
buffer = dsp.AsyncBuffer(numel(audio));
write(buffer,audio);
VAD = voiceActivityDetector;
```

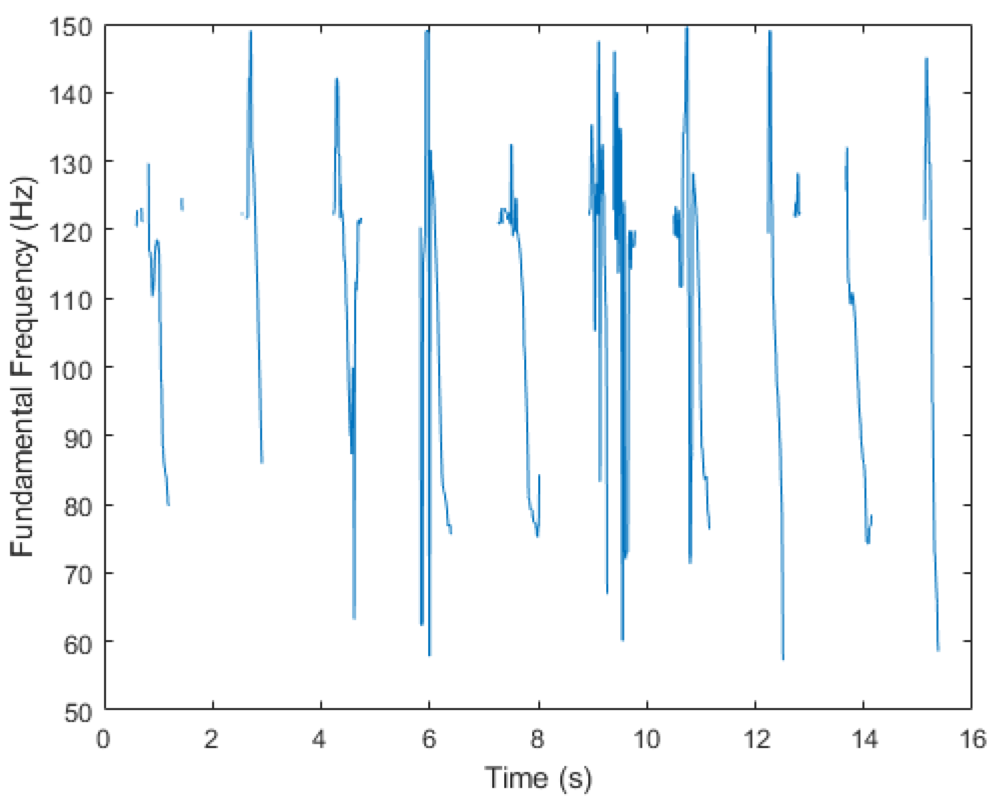
While there are enough samples to hop, read from the buffer and determine the probability that the frame contains speech. To mimic the decision spacing in time of the `pitch` function, the first frame read from the buffer has no overlap.

```
n = 1;
probabilityVector = zeros(numel(loc),1);
```

```
while buffer.NumUnreadSamples >= hopLength
    if n==1
        x = read(buffer,windowLength);
    else
        x = read(buffer,windowLength,overlapLength);
    end
    probabilityVector(n) = VAD(x);
    n = n+1;
end
```

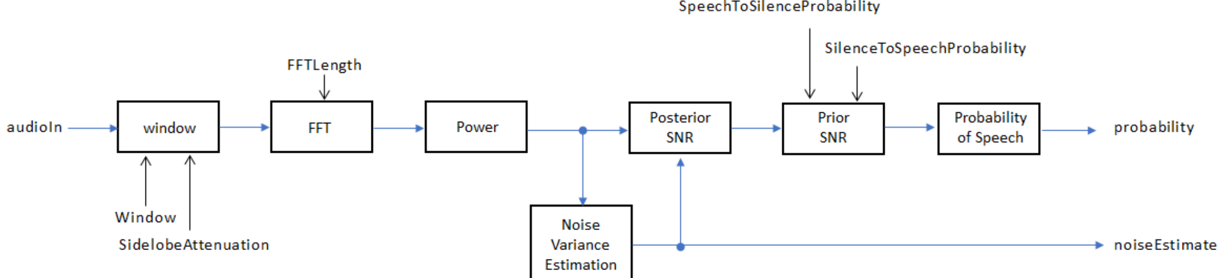
Use the probability vector determined by the `voiceActivityDetector` to plot a pitch contour for the speech file that corresponds to regions of speech.

```
validIdx = probabilityVector>0.99;
loc(~validIdx) = nan;
f0(~validIdx) = nan;
plot(loc/fs,f0)
ylabel('Fundamental Frequency (Hz)')
xlabel('Time (s)')
```



Algorithms

The `voiceActivityDetector` implements the algorithm described in [1].



If `InputDomain` is specified as `'Time'`, the input signal is windowed and then converted to the frequency domain according to the `Window`, `SidelobeAttenuation`, and `FFTLength` properties. If `InputDomain` is specified as frequency, the input is assumed to be a windowed discrete time Fourier transform (DTFT) of an audio signal. The signal is then converted to the power domain. Noise variance is estimated according to [2]. The posterior and prior SNR are estimated according to the Minimum Mean-Square Error (MMSE) formula described in [3]. A log likelihood ratio test and Hidden Markov Model (HMM)-based hang-over scheme determine the probability that the current frame contains speech, according to [1].

References

[1] Sohn, Jongseo., Nam Soo Kim, and Wonyong Sung. "A Statistical Model-Based Voice Activity Detection." *Signal Processing Letters IEEE*. Vol. 6, No. 1, 1999.

[2] Martin, R. "Noise Power Spectral Density Estimation Based on Optimal Smoothing and Minimum Statistics." *IEEE Transactions on Speech and Audio Processing*. Vol. 9, No. 5, 2001, pp. 504–512.

[3] Ephraim, Y., and D. Malah. "Speech Enhancement Using a Minimum Mean-Square Error Short-Time Spectral Amplitude Estimator." *IEEE Transactions on Acoustics, Speech, and Signal Processing*. Vol. 32, No. 6, 1984, pp. 1109–1121.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

"System Objects in MATLAB Code Generation" (MATLAB Coder)

See Also

System Objects
`cepstralFeatureExtractor`

Blocks
Voice Activity Detector

Functions
`mfcc` | `pitch`

Introduced in R2018a

cepstralFeatureExtractor System object

Extract cepstral features from audio segment

Description

The `cepstralFeatureExtractor` System object extracts cepstral features from an audio segment. Cepstral features are commonly used to characterize speech and music signals.

To extract cepstral features:

1. Create the cepstralFeatureExtractor object and set its properties.
2. Call the object with arguments, as if it were a function.

To learn more about how System objects work, see What Are System Objects? (MATLAB).

Creation

Syntax

```
cepFeatures = cepstralFeatureExtractor
cepFeatures = cepstralFeatureExtractor(Name,Value)
```

Description

`cepFeatures = cepstralFeatureExtractor` creates a System object, `cepFeatures`, that calculates cepstral features independently across each input channel. Columns of the input are treated as individual channels.

`cepFeatures = cepstralFeatureExtractor(Name,Value)` sets each property `Name` to the specified `Value`. Unspecified properties have default values.

Example: `cepFeatures = cepstralFeatureExtractor('InputDomain','Frequency','SampleRate',fs,'`

LogEnergy','Replace') accepts a signal in the frequency domain, sampled at fs Hz. The first element of the coefficients vector is replaced by the log energy value.

Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see System Design in MATLAB Using System Objects (MATLAB).

InputDomain — Domain of input signal

'Time' (default) | 'Frequency'

Domain of the input signal, specified as either `'Time'` or `'Frequency'`.

Data Types: char | string

NumCoeffs — Number of coefficients to return

13 (default) | positive integer

Number of coefficients to return, specified as an integer in the range [2, *v*], where *v* is the number of valid passbands.

The number of valid passbands is defined as `sum(BandEdges <= floor(fs/2))-2`. A passband is valid if its edges fall below `fs/2`.

- *BandEdges* –– Vector containing the band edges of the filter bank, specified through the `BandEdges` property.
- *fs* –– Sample rate of the input audio signal, specified through the `SampleRate` property.

Data Types: single | double

FFTLength — FFT length

[] (default) | positive integer

FFT length, specified as a positive integer. The default, `[]`, means that the FFT length is equal to the number of rows in the input signal.

Dependencies

To enable this property, set `InputDomain` to `'Time'`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`LogEnergy` — Specify how the log energy is shown

`'Append'` (default) | `'Replace'` | `'Ignore'`

Specify how the log energy is shown in the coefficients vector output, specified as:

- `'Append'` -– The object prepends the log energy to the coefficients vector. The length of the coefficients vector is 1 + `NumCoeffs`.
- `'Replace'` -– The object replaces the first coefficient with the log energy of the signal. The length of the coefficients vector is `NumCoeffs`.
- `'Ignore'` -– The object does not calculate or return the log energy.

Data Types: `char` | `string`

`SampleRate` — Input sample rate (Hz)

`16000` (default) | positive scalar

Input sample rate in Hz, specified as a real positive scalar.

Tunable: Yes

Data Types: `single` | `double`

Advanced properties

BandEdges — Band edges of auditory filter bank (Hz)

row vector

Band edges of the filter bank in Hz, specified as a nonnegative monotonically increasing row vector in the range [0, ∞). The maximum bandedge frequency can be any finite number. The number of bandedges must be in the range [4, 80].

The default band edges are spaced linearly for the first ten and then logarithmically after. The default band edges are set as recommended by [1].

Data Types: `single` | `double`

FilterBankDesignDomain — Domain for filter bank design
'Hz' (default) | 'Bin'

Domain for filter bank design, specified as either 'Hz' or 'Bin'. The filterbank is designed as overlapped triangles with band edges specified by the BandEdges property.

The BandEdges property is specified in Hz. When you set the design domain to:

- 'Hz' -– Filter bank triangles are drawn in Hz and are mapped onto bins.

  Here is an example that plots the filter bank in bins when the FilterBankDesignDomain is set to 'Hz':

```
[audioFile, fs] = audioread('NoisySpeech-16-22p5-mono-5secs.wav');
duration = round(0.02*fs); % 20 ms audio segment
audioSegment = audioFile(5500:5500+duration-1);
cepFeatures = cepstralFeatureExtractor('SampleRate',fs)
```

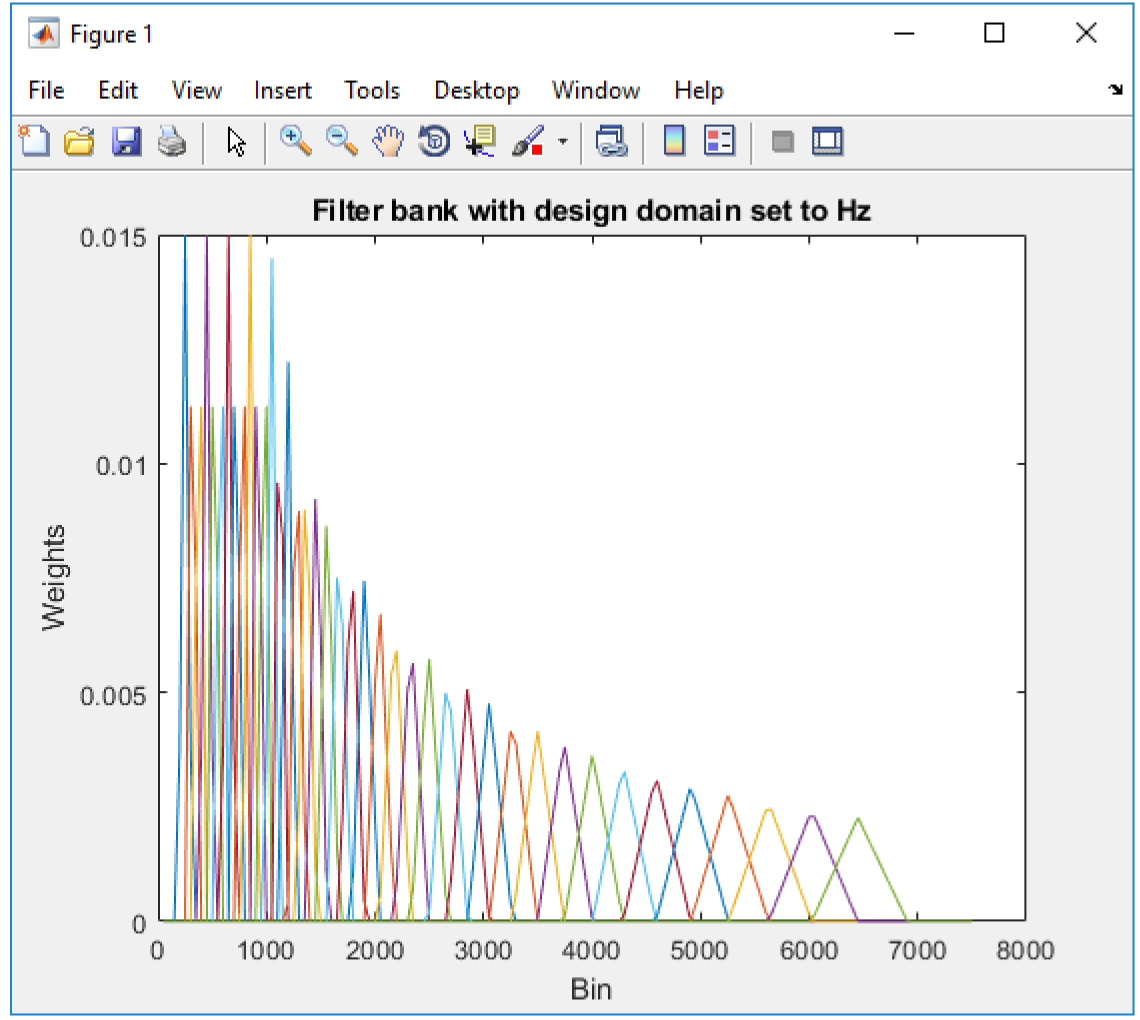
```
cepFeatures =
 cepstralFeatureExtractor with properties:

   Properties
                InputDomain: 'Time'
                  NumCoeffs: 13
                  FFTLength: []
                  LogEnergy: 'Append'
                 SampleRate: 22500

   Advanced Properties
                  BandEdges: [1×42 double]
     FilterBankDesignDomain: 'Hz'
    FilterBankNormalization: 'Bandwidth'
```

  Pass the audio segment as an input to the cepstral feature extractor algorithm to lock the object.

```
[coeffs,delta,deltaDelta] = cepFeatures(audioSegment);
```

  Using the getFilters function, get the filter bank. Plot the filter bank.

```
[filterbank, freq] = getFilters(cepFeatures);
plot(freq(1:150),filterbank(1:150,:))
```



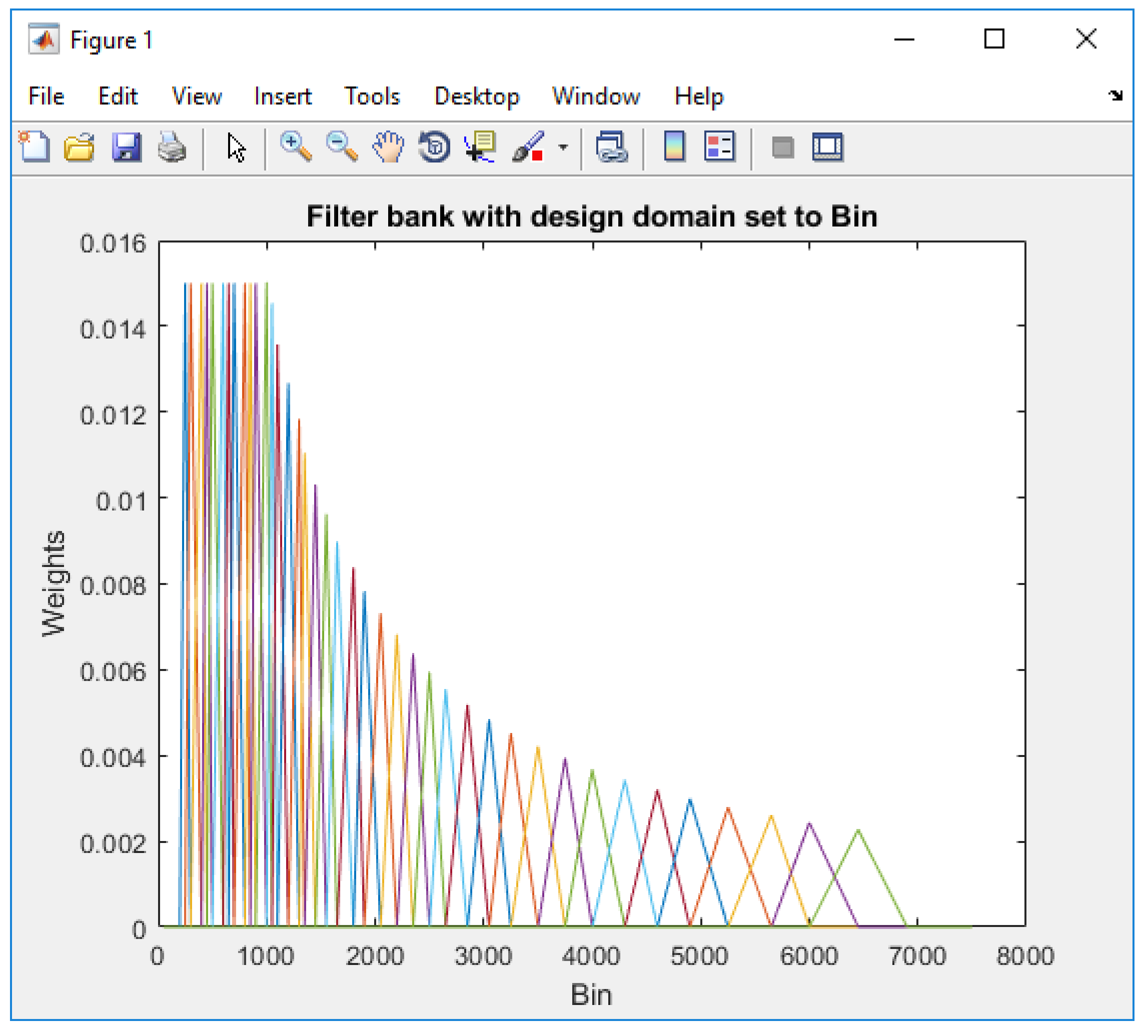
For details, see [1].

- `'Bin'` -- The bandedge frequencies in `'Hz'` are converted to bins. The filter bank triangles are drawn symmetrically in bins.

  Change the `FilterBankDesignDomain` property to `'Bin'`:

```
release(cepFeatures);
cepFeatures.FilterBankDesignDomain = 'Bin';
```

```
[coeffs,delta,deltaDelta] = cepFeatures(audioSegment);
[filterbank, freq] = getFilters(cepFeatures);
plot(freq(1:150),filterbank(1:150,:))
```



For details, see [2].

Data Types: `char` | `string`

FilterBankNormalization — Normalize filter bank
'Bandwidth' (default) | 'Area' | 'None'

Normalization technique used on the weights of the filter bank, specified as:

- 'Bandwidth' –– The weights of each bandpass filter are normalized by the corresponding bandwidth of the filter.
- 'Area' –– The weights of each bandpass filter are normalized by the corresponding area of the bandpass filter.
- 'None' –– The weights of the filter are not normalized.

Data Types: char | string

Usage

Syntax

```
[coeffs,delta,deltaDelta] = cepFeatures(audioIn)
```

Description

[coeffs,delta,deltaDelta] = cepFeatures(audioIn) returns the cepstral coefficients, the log energy, the delta, and the delta-delta.

The log energy value prepends the coefficient vector or replaces the first element of the coefficients vector based on whether you set the LogEnergy property to 'Append' or 'Replace'. For details, see “coeffs” on page 3-0.

Input Arguments

audioIn — Audio input to cepstral feature extractor
column vector | matrix

Audio input to the cepstral feature extractor, specified as a column vector or a matrix. If specified as a matrix, the columns are treated as independent audio channels.

Data Types: single | double

Output Arguments

coeffs — Cepstral coefficients
column vector | matrix

Cepstral coefficients, returned as a column vector or a matrix. If the coefficients matrix is an *N*-by-*M* matrix, *N* is determined by the values you specify in `NumCoeffs` and `LogEnergy` properties. *M* equals the number of input audio channels.

When the `LogEnergy` property is set to:

- `'Append'` –– The object prepends the log energy value to the coefficients vector. The length of the coefficients vector is 1 + `NumCoeffs`.
- `'Replace'` –– The object replaces the first coefficient with the log energy of the signal. The length of the coefficients vector is `NumCoeffs`.
- `'Ignore'` –– The object does not calculate or return the log energy.

Data Types: `single` | `double`

delta — Change in coefficients
column vector | matrix

Change in coefficients over consecutive calls to the algorithm, returned as a vector or a matrix. The `delta` array is of the same size and data type as the `coeffs` array.

In this example, `cepFeatures` is the cepstral feature extractor that accepts audio input signal sampled at 12 kHz. Stream in three segments of audio signal on three consecutive calls to the object algorithm. Return the cepstral coefficients of the filter bank and the corresponding `delta` values.

```
cepFeatures = cepstralFeatureExtractor('SampleRate',12000);
[coeff1,delta1] = cepFeatures(audioIn);
[coeff2,delta2] = cepFeatures(audioIn);
[coeff3,delta3] = cepFeatures(audioIn);
```

`delta2` is computed as `coeff2-coeff1`, while `delta3` is computed as `coeff3-coeff2`. The initial array, `delta1`, is an array of zeros.

Data Types: `single` | `double`

deltaDelta — Change in delta values
column vector | matrix

Change in `delta` values over consecutive calls to the algorithm, returned as a vector or a matrix. The `deltaDelta` array is the same size and data type as the `coeffs` and `delta` arrays.

In this example, consecutive calls to the cepstral feature extractor algorithm return the `deltaDelta` values in addition to the coefficients and the `delta` values.

```
cepFeatures = cepstralFeatureExtractor('SampleRate',12000);
[coeff1,delta1,deltaDelta1] = cepFeatures(audioIn);
[coeff2,delta2,deltaDelta2] = cepFeatures(audioIn);
[coeff3,delta3,deltaDelta3] = cepFeatures(audioIn);
```

`deltaDelta2` is computed as `delta2-delta1`, while `deltaDelta3` is computed as `delta3-delta2`. The initial array, `deltaDelta1`, is an array of zeros.

Data Types: `single` | `double`

Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

Specific to cepstralFeatureExtractor

getFilters	Get auditory filter bank

Common to All System Objects

clone	Create duplicate System object
isLocked	Determine if System object is in use
release	Release resources and allow changes to System object property values and input characteristics
reset	Reset internal states of System object
step	Run System object algorithm

Examples

Get MFCC Data for Speech Segment

Extract the mel frequency cepstral coefficients and the log energy values of segments in a speech file. Return `delta`, the difference between current and the previous cepstral coefficients, and `deltaDelta`, the difference between the current and the previous `delta` values. The log energy value the object computes can prepend the coefficients vector or replace the first element of the coefficients vector. This is done based on whether you set the `LogEnergy` property of the `cepstralFeatureExtractor` object to `'Replace'` or `'Append'`.

Read an audio signal from `'SpeechDFT-16-8-mono-5secs.wav'` file. Extract a 40 ms segment from the audio data. Create a `cepstralFeatureExtractor` object. The cepstral coefficients computed by the default object are the mel frequency coefficients. In addition, the object computes the log energy, delta, and delta-delta values of the audio segment.

```
[audioFile, fs] = audioread('SpeechDFT-16-8-mono-5secs.wav');
duration = round(0.04*fs); % 40 ms
audioSegment = audioFile(5500:5500+duration-1);
cepFeatures = cepstralFeatureExtractor('SampleRate',fs)
```

```
cepFeatures = 
  cepstralFeatureExtractor with properties:

   Properties
    InputDomain: 'Time'
      NumCoeffs: 13
      FFTLength: []
      LogEnergy: 'Append'
     SampleRate: 8000

  Show all properties
```

The `LogEnergy` property is set to `'Append'`. The first element in the coefficients vector is the log energy value and the remaining elements are the 13 cepstral coefficients computed by the object. The number of cepstral coefficients is determined by the value you specify in the `NumCoeffs` property.

```
[coeffs,delta,deltaDelta] = cepFeatures(audioSegment)
```

```
coeffs = 14×1

    3.8281
```

```
  -10.7136
    2.5113
    0.8357
    2.0019
    1.0714
   -0.7524
    0.6335
   -0.3084
   -0.2283
       ⋮

delta = 14×1

     0
     0
     0
     0
     0
     0
     0
     0
     0
     0
       ⋮

deltaDelta = 14×1

     0
     0
     0
     0
     0
     0
     0
     0
     0
     0
       ⋮
```

The initial values for the `delta` and `deltaDelta` arrays are always zero. Consider another 40 ms audio segment in the file and extract the cepstral features from this segment.

```
audioSegmentTwo = audioFile(5820:5820+duration-1);
[coeffsTwo,deltaTwo,deltaDeltaTwo] = cepFeatures(audioSegmentTwo)
```

```
coeffsTwo = 14×1

    3.0899
  -11.8236
    1.3105
    2.3195
    1.6894
   -0.0264
    0.6509
    0.8009
   -0.5502
    0.3022
      ⋮


deltaTwo = 14×1

   -0.7382
   -1.1100
   -1.2008
    1.4838
   -0.3125
   -1.0978
    1.4033
    0.1674
   -0.2418
    0.5306
      ⋮


deltaDeltaTwo = 14×1

   -0.7382
   -1.1100
   -1.2008
    1.4838
   -0.3125
   -1.0978
    1.4033
    0.1674
   -0.2418
    0.5306
```

⋮

Verify that the difference between `coeffsTwo` and `coeffs` vectors equals `deltaTwo`.

```
isequal(coeffsTwo-coeffs,deltaTwo)
```

```
ans = logical
   1
```

Verify that the difference between `deltaTwo` and `delta` vectors equals `deltaDeltaTwo`.

```
isequal(deltaTwo-delta,deltaDeltaTwo)
```

```
ans = logical
   1
```

Frequency-Domain Voice Activity Detection and Cepstral Feature Extraction

Many feature extraction techniques operate on the frequency domain. Converting an audio signal to the frequency domain only once is efficient. In this example, you convert a streaming audio signal to the frequency domain and feed that signal into a voice activity detector. If speech is present, mel-frequency cepstral coefficients (MFCC) features are extracted from the frequency-domain signal using the `cepstralFeatureExtractor System object™`.

Create a `dsp.AudioFileReader` System object to read from an audio file.

```
fileReader = dsp.AudioFileReader('Counting-16-44p1-mono-15secs.wav');
fs = fileReader.SampleRate;
```

Process the audio in 30 ms frames with a 10 ms hop. Create a default `dsp.AsyncBuffer` object to manage overlap between audio frames.

```
samplesPerFrame = ceil(0.03*fs);
samplesPerHop = ceil(0.01*fs);
samplesPerOverlap = samplesPerFrame - samplesPerHop;

fileReader.SamplesPerFrame = samplesPerHop;
buffer = dsp.AsyncBuffer;
```

Create a `voiceActivityDetector` System object and a `cepstralFeatureExtractor` System object. Specify that they operate in the frequency domain. Create a `dsp.SignalSink` to log the extracted cepstral features.

```
VAD = voiceActivityDetector('InputDomain','Frequency');
cepFeatures = cepstralFeatureExtractor('InputDomain','Frequency','SampleRate',fs,'LogEr
sink = dsp.SignalSink;
```

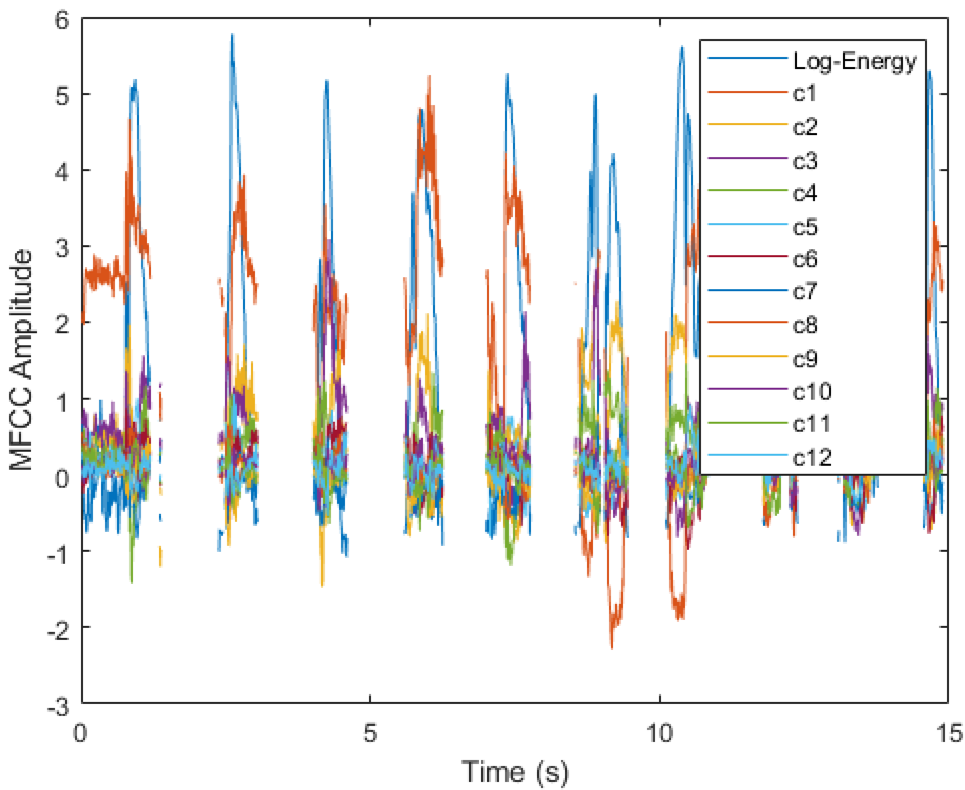
In an audio stream loop:

1. Read one hop's of samples from the audio file and save the samples into the buffer.
2. Read a frame from the `buffer` with specified overlap from the previous frame.
3. Call the voice activity detector to get the probability of speech for the frame under analysis.
4. If the frame under analysis has a probability of speech greater than 0.75, extract cepstral features and log the features using the signal sink. If the frame under analysis has a probability of speech less than 0.75, write a vector of NaNs to the sink.

```
threshold = 0.75;
nanVector = nan(1,13);
while ~isDone(fileReader)
    audioIn = fileReader();
    write(buffer,audioIn);

    overlappedAudio = read(buffer,samplesPerFrame,samplesPerOverlap);
    X = fft(overlappedAudio,2048);

    probabilityOfSpeech = VAD(X);
    if probabilityOfSpeech > threshold
        xFeatures = cepFeatures(X);
        sink(xFeatures')
    else
        sink(nanVector)
    end
end
```

Visualize the cepstral coefficients over time.

```
timeVector = linspace(0,15,size(sink.Buffer,1));
plot(timeVector,sink.Buffer)
xlabel('Time (s)')
ylabel('MFCC Amplitude')
legend('Log-Energy','c1','c2','c3','c4','c5','c6','c7','c8','c9','c10','c11','c12')
```



- “Speaker Identification Using Pitch and MFCC”

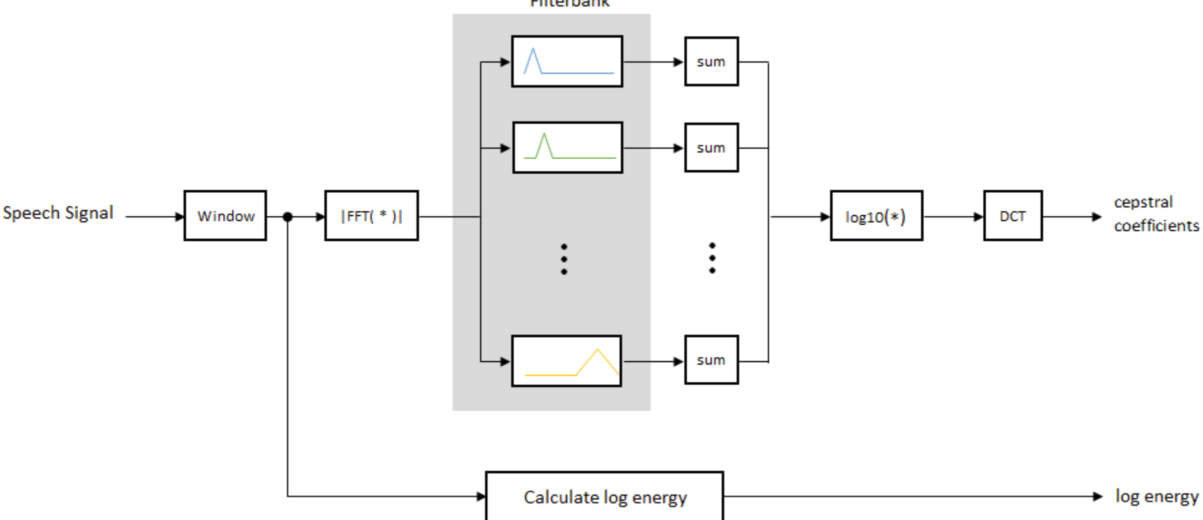
Algorithms

Auditory Cepstrum Coefficients

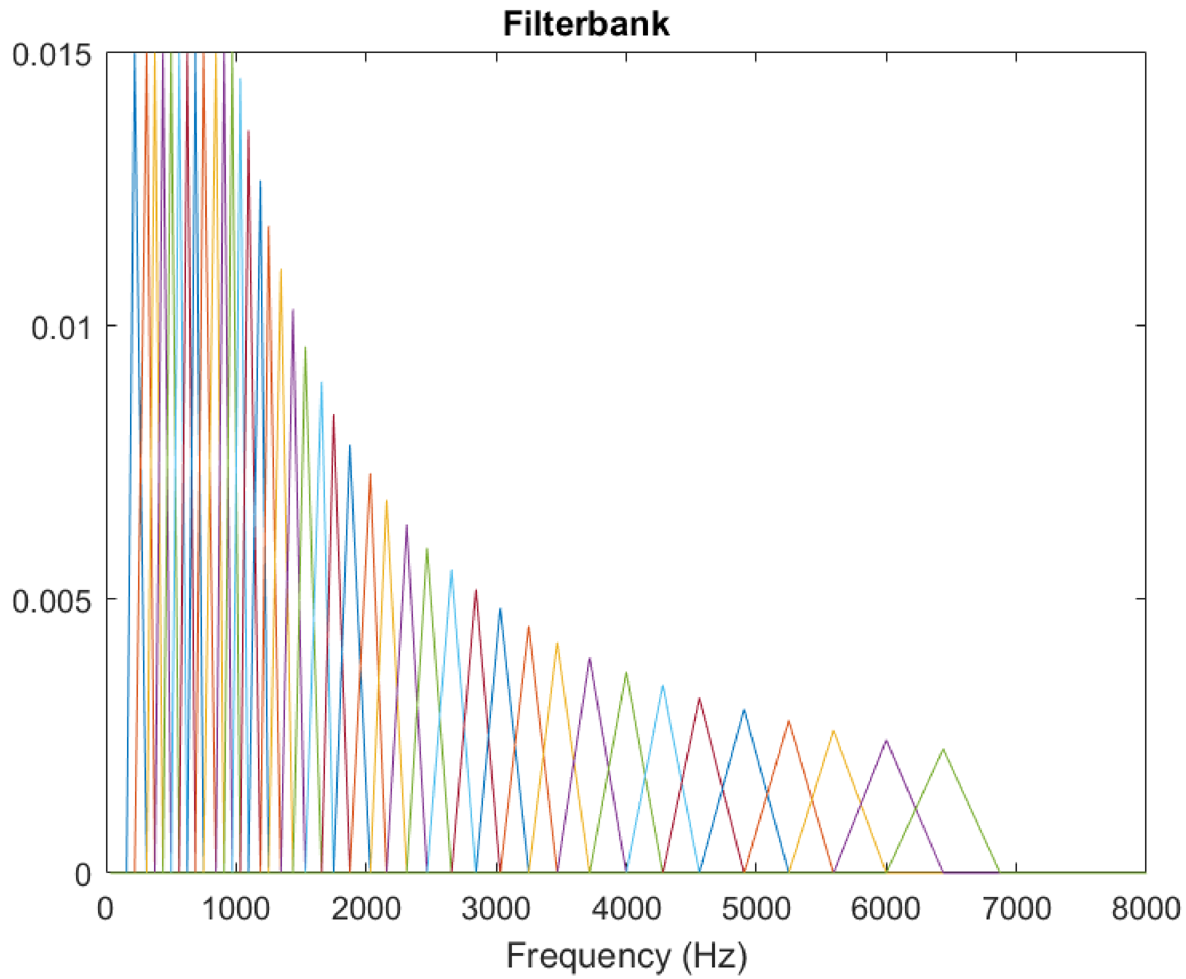
Auditory cepstrum coefficients are popular features extracted from speech signals for use in recognition tasks. In the source-filter model of speech, cepstral coefficients are understood to represent the filter (vocal tract). The vocal tract frequency response is relatively smooth, whereas the source of voiced speech can be modeled as an impulse

train. As a result, the vocal tract can be estimated by the spectral envelope of a speech segment.

The motivating idea of cepstral coefficients is to compress information about the vocal tract (smoothed spectrum) into a small number of coefficients based on an understanding of the cochlea. Although there is no hard standard for calculating the coefficients, the basic steps are outlined by the diagram.



The windowing is done by a Hamming function. The default filter bank linearly spaces the first 10 triangular filters and logarithmically spaces the remaining filters.



Log Energy

If the input (x) is a time-domain signal, the log energy is computed using the following equation:

$$\log E = \log(\mathrm{sum}(x^2))$$

If the input (x) is a frequency-domain signal, the log energy is computed using the following equation:

$$\log E = \log\left(\text{sum}\left(|x|^2\right) \middle/ FFTLength\right)$$

References

[1] Auditory Toolbox. https://engineering.purdue.edu/~malcolm/interval/1998-010/AuditoryToolboxTechReport.pdf

[2] ETSI ES 201 108 V1.1.3 (2003-09). http://www.etsi.org/deliver/etsi_es/201100_201199/201108/01.01.03_60/es_201108v010103p.pdf

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

“System Objects in MATLAB Code Generation” (MATLAB Coder)

See Also

Functions
`mfcc` | `pitch`

System Objects
`voiceActivityDetector`

Blocks
Cepstral Feature Extractor | Voice Activity Detector

Topics

"Speaker Identification Using Pitch and MFCC"

Introduced in R2018a

getFilters

Get auditory filter bank

Syntax

```
[filterbank,freq] = getFilters(cepFeatures)
```

Description

`[filterbank,freq] = getFilters(cepFeatures)` returns the filter bank and the corresponding frequency bins in Hz. Each column of the filter bank corresponds to a single bandpass filter. The filterbank is undefined until the object is locked.

Examples

Get Auditory Filter Bank

The auditory filter bank contains a set of bandpass filters that are used to extract the cepstral features from an audio signal. The cepstral features include cepstral coefficients (`coeffs`), the difference between the current and the previous cepstral coefficients (`delta`), and the difference between the current and the previous delta values, `deltaDelta`. The `getFilters` function returns the auditory filter bank and the corresponding frequency bins.

Read an audio signal from `'SpeechDFT-16-8-mono-5secs.wav'` file. Extract a 40 ms segment from the audio data. Create a `cepstralFeatureExtractor` System object™ that accepts a time-domain audio input signal sampled at 8 kHz.

```
[audioFile, fs] = audioread('SpeechDFT-16-8-mono-5secs.wav');
duration = round(0.04*fs); % 40 ms
audioSegment = audioFile(5500:5500+duration-1);
cepFeatures = cepstralFeatureExtractor('SampleRate',fs)
```

```
cepFeatures = 
  cepstralFeatureExtractor with properties:
```

```
  Properties
   InputDomain: 'Time'
     NumCoeffs: 13
     FFTLength: []
     LogEnergy: 'Append'
    SampleRate: 8000

 Show all properties
```

Pass the 40 ms audio segment as an input to the `cepstralFeatureExtractor` algorithm. The algorithm computes the mel frequency coefficients, log energy, delta, and delta-delta values of the audio segment.

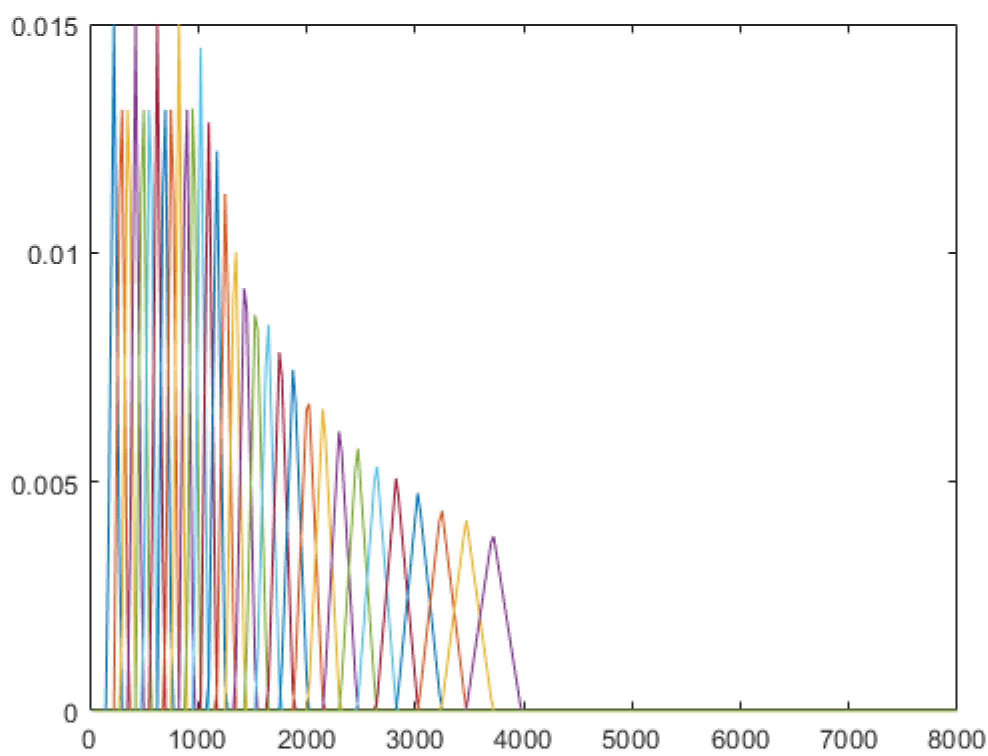
```
[coeffs,delta,deltaDelta] = cepFeatures(audioSegment);
```

Using the `getFilters` function, get the filter bank that computes the cepstral features. Each column in the filter bank contains a bandpass filter. The frequency bins corresponding to the bandpass filters are displayed in `Bins`. Note that the `getFilters` function requires the `cepstralFeatureExtractor` object to be locked.

```
[filterbank, freq] = getFilters(cepFeatures);
```

Plot the filter bank, and you can see that the filter bank columns 33 to 40 have zero elements. These columns correspond to filters whose band edges (one or both) are above the Nyquist frequency, $(fs/2)$. In this example, the Nyquist frequency is 4 kHz.

```
plot(freq,filterbank)
```




```
cepFeatures.BandEdges(33:40)
```

```
ans = 1×8
10³ ×

    3.6915    3.9543    4.2357    4.5371    4.8601    5.2059    5.5765    5.9733 ⋯
```

```
nnz(filterbank(:,33:40))
```

```
ans = 0
```

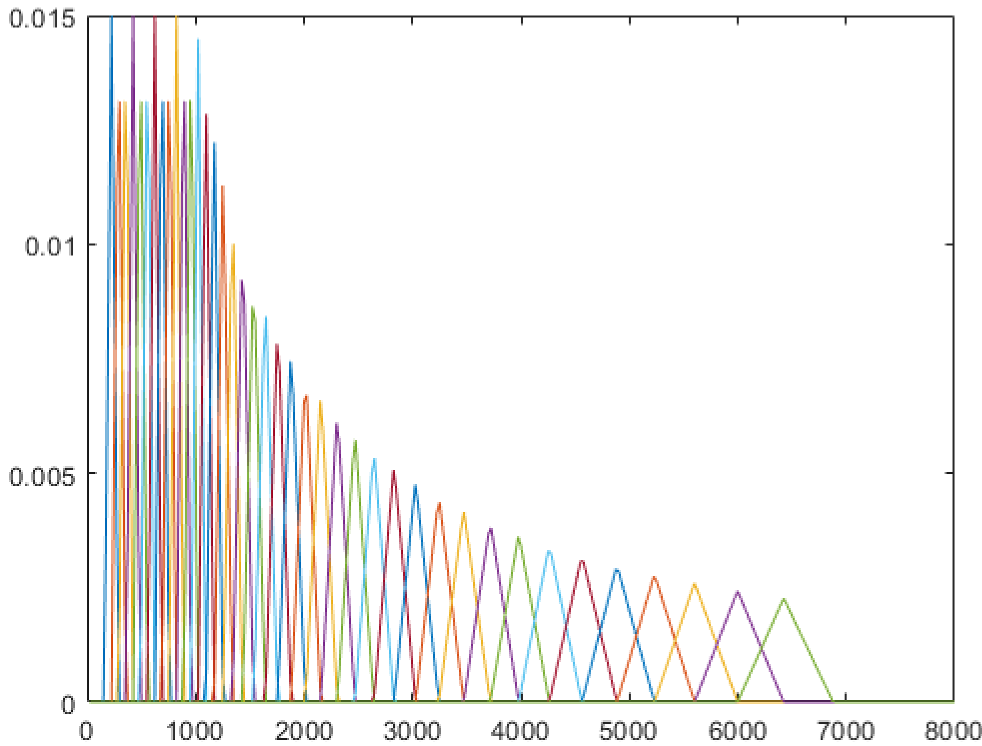
Release the cepstral feature extractor object and pass a different audio signal sampled at 22.5 kHz.

```
release(cepFeatures)
[audioFileTwo, fsTwo] = audioread('NoisySpeech-16-22p5-mono-5secs.wav');
duration = round(0.04*fsTwo); % 40 ms
audioSegmentTwo = audioFileTwo(5500:5500+duration-1);
cepFeatures = cepstralFeatureExtractor('SampleRate',fsTwo)
```

```
cepFeatures =
  cepstralFeatureExtractor with properties:

   Properties
    InputDomain: 'Time'
      NumCoeffs: 13
      FFTLength: []
      LogEnergy: 'Append'
     SampleRate: 22500

  Show all properties
```

The Nyquist frequency is 22,500/2, which is 11,250 Hz. Extract the cepstral features of the second audio segment. Plot the filter bank, which is used to compute the cepstral features. Zoom in on the axis for comparison.

```
[coeffsTwo,deltaTwo,deltaDeltaTwo] = cepFeatures(audioSegmentTwo);
[filterbankTwo, freqTwo] = getFilters(cepFeatures);
plot(freqTwo,filterbankTwo)
axis([0 8000 0 0.015])
```



All the band edges are below the Nyquist frequency, and the bandpass filters in the filter bank have nonzero coefficients.

Input Arguments

`cepFeatures` — Input cepstral feature extractor System object
`cepstralFeatureExtractor System object`

Input cepstral feature extractor, specified as a `cepstralFeatureExtractor` System object. To use the `getFilters` function, the object must be locked. The filter bank is

defined only when the object is locked. The object is locked when you call the object algorithm.

Output Arguments

`filterbank` — Auditory filter bank
matrix

Filter bank used to calculate cepstral features, returned as a matrix. Each column of the matrix corresponds to a single bandpass filter in the filter bank. The number of columns in the matrix is given by $m - 2$, where m is the length of the vector you specify in the `BandEdges` property of the System object. The number of rows in the matrix corresponds to the FFT length. By default, the FFT length equals the number of rows in the input signal. You can also specify the FFT length through the `FFTLength` property of the System object.

If the Nyquist frequency, $fs/2$, is less than the band edge frequencies you specify in the `BandEdges` property, the coefficients of the bandpass filters that fall outside the Nyquist range are set to zero. fs is the sample rate you specify in the `SampleRate` property of the System object.

Data Types: `single` | `double`

`freq` — Frequency bins corresponding to filter bank (Hz)
row vector

Frequency bins corresponding to the filter bank in Hz, returned as a row vector. The length of the vector equals the FFT length.

Data Types: `single` | `double`

See Also

System Objects
`cepstralFeatureExtractor`

Introduced in R2018a

visualize

Visualize static characteristic of dynamic range controller

Syntax

```
visualize(dynamicRangeController)
visualize(dynamicRangeController,inputRange)
outputLevel = visualize( ___ )
```

Description

`visualize(dynamicRangeController)` plots the static characteristic of the dynamic range control object. The plot is updated automatically when properties of the object change.

`visualize(dynamicRangeController,inputRange)` enables you to specify the input range.

`outputLevel = visualize( ___ )` returns the dB output level corresponding to the input range. You can use any of the input arguments from previous syntaxes.

Note This syntax is only available for the `compressor`, `limiter`, and `expander` System objects. It is not available for the `noiseGate` System object.

Examples

Plot Static Characteristic

Create an object of the `compressor` System object™, and then plot the static characteristic.

```
dynamicRangeCompressor = compressor;
visualize(dynamicRangeCompressor)
```

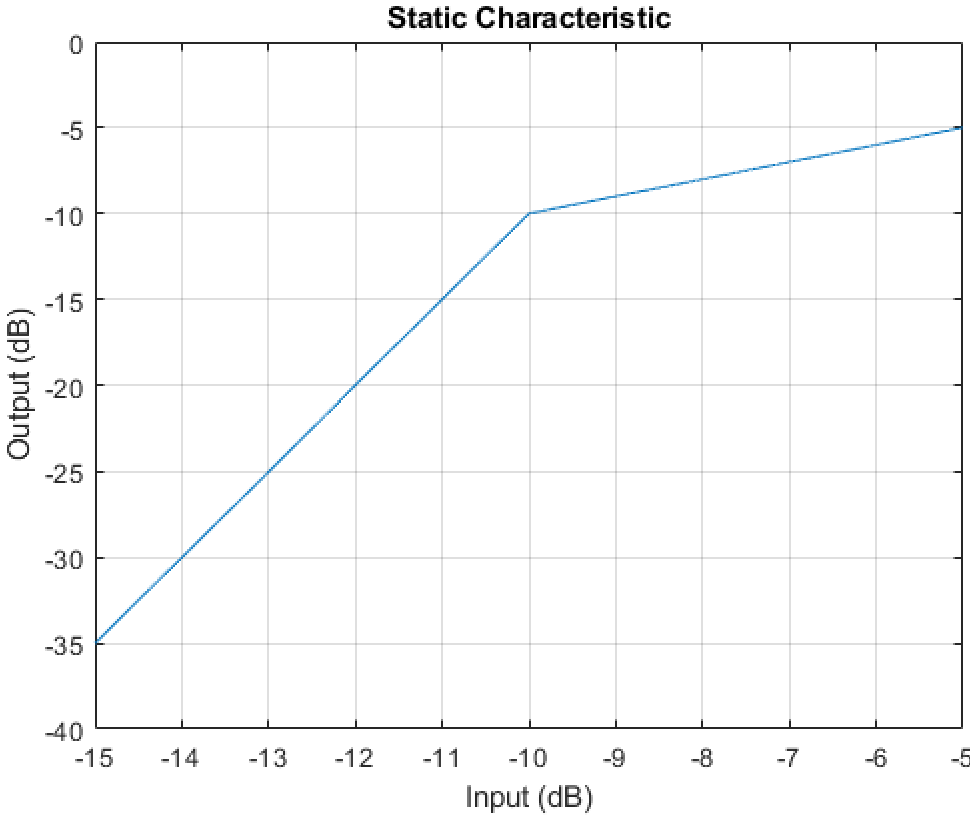
The static characteristic plot updates automatically if you modify a property of the object.

```
dynamicRangeCompressor.Threshold = -30;
```

Specify Range of Static Characteristic Plot

Create an object of the `expander` System object™. Plot the static characteristic over the range -15 to -5, in 0.001 dB increments.

```
dynamicRangeExpander = expander;
visualize(dynamicRangeExpander,-15:0.001:-5)
```



Get Output Level From Static Characteristic

Create an object of the `limiter` System object™. Get the output level of the static characteristic over a specified range.

```
dynamicRangeLimiter = limiter;
inputLevel = -15:1:-5
outputLevel = visualize(dynamicRangeLimiter,inputLevel)
```

Input Arguments

`dynamicRangeController` — Dynamic range control object
object

Dynamic range control object, specified as an object of `compressor` , `expander` , `limiter` or `noiseGate`.

`inputRange` — Range to calculate static characteristic output
vector of monotonically increasing values

Range over which to calculate the output of the static characteristic.

The default input range depends on the dynamic range control object:

- `compressor` –– `[-50:0.01:0]` dB
- `limiter` –– `[-50:0.01:0]` dB
- `expander` –– `[-50:0.01:0]` dB
- `noiseGate` –– `[0:0.001:1]` linear

Output Arguments

`outputLevel` — Output level (dB)
vector

Output level in dB, returned as a vector the same size as `inputRange`.

This output is only available for the `compressor`, `limiter`, and `expander` System objects. It is not available for the `noiseGate` System object.

See Also

System Objects
`compressor` | `expander` | `limiter` | `noiseGate`

Topics
"Dynamic Range Control"

Introduced in R2016a

createAudioPluginClass

Create audio plugin class that implements functionality of System object

Syntax

```
createAudioPluginClass(obj)
createAudioPluginClass(obj,pluginName)
```

Description

`createAudioPluginClass(obj)` creates a System object plugin that implements the functionality of the Audio System Toolbox System object, `obj`. The name of the created class is the System object variable name, `obj`, followed by `'Plugin'`, for example, `objPlugin`.

If the object is locked, the number of input and output channels of the plugin is equal to the number of channels of the object. Otherwise, the number of channels is equal to 2.

`createAudioPluginClass(obj,pluginName)` specifies the name of your created System object plugin class.

Example: `createAudioPluginClass(obj,'coolEffect')` creates a System object plugin with class name `'coolEffect'`.

Examples

Create an Audio Plugin Class From a System Object

Create a `compressor` object. Call `createAudioPluginClass` to create a System object™ plugin class that implements the functionality of the compressor object.

```
cmpr = compressor;
createAudioPluginClass(cmpr)
```

Specify Name of Created Plugin Class

Create an object of the `reverberator` System object™. Call `createAudioPluginClass` to create a System object™ plugin class that implements the functionality of the reverberator object, specifying the plugin class name as the second argument.

```
reverb = reverberator;
createAudioPluginClass(reverb,'Garage')
```

Input Arguments

`obj` — System object to create plugin class from
Audio System Toolbox System object

System object from which to create a plugin class.

`pluginName` — Name of created plugin class
character vector

Name of created plugin class, specified as a character vector with fewer than 64 elements.

Data Types: `char`

See Also

System Objects
`audioOscillator` | `compressor` | `crossoverFilter` | `expander` | `graphicEQ` | `limiter` | `multibandParametricEQ` | `noiseGate` | `octaveFilter` | `reverberator` | `wavetableSynthesizer` | `weightingFilter`

Topics
"Design an Audio Plugin"
"Export a MATLAB Plugin to a DAW"

Introduced in R2016a

getFilter

Return biquad filter object with design parameters set

Syntax

```
biquad = getFilter(obj)
```

Description

biquad = getFilter(obj)returns a dsp.BiquadFilter object, biquad. The SOSMatrix and ScaleValues properties of the biquad filter object are set as specified by the obj System object.

Use getFilter for the design capabilities of the obj System object and the processing capabilities of the dsp.BiquadFilter System object.

Examples

Get Biquad Filter for Octave Filter Design

Create an octaveFilter System object™. Call getFilter on your object to return a dsp.BiquadFilter object with design parameters specified by your octaveFilter System object.

```
octFilt = octaveFilter;
biquad = getFilter(octFilt)
```

Get Biquad Filter for Weighting Filter Design

Create a weightingFilter System object™ and visualize the frequency response.

```
weightFilt = weightingFilter;
visualize(weightFilt)
```

Call `getFilter` on your object to return a `dsp.BiquadFilter` object with design parameters specified by your `weightingFilter` System object. Use `fvtool` to visualize the biquad filter.

```
biquad = getFilter(weightFilt)
fvtool(biquad,'FrequencyScale','log')
```

Input Arguments

obj — System object to get filter from
System object

System object that you want to get a biquad filter object from.

Output Arguments

biquad — Object of `dsp.BiquadFilter`
object

Object of the `dsp.BiquadFilter` System object.

See Also

System Objects
`dsp.BiquadFilter` | `octaveFilter` | `weightingFilter`

Topics
"Audio Weighting Filters"
"Sound Pressure Measurement Using Weighting Filters"

Introduced in R2016b

info

Get audio device information

Syntax

`infoStruct = info(obj)`

Description

`infoStruct = info(obj)` returns a structure, `infoStruct`, containing information about the System object, `obj`.

Examples

Get Input Audio Device Information

Create an object of the `audioDeviceReader` System object™ and then call `info` to return a structure containing information about the selected driver, device name, and the maximum number of input channels.

```
deviceReader = audioDeviceReader;
info(deviceReader)
```

Get Output Audio Device Information

Create an object of the `audioDeviceWriter` System object™ and then call `info` to return a structure containing information about the selected driver, device name, and the maximum number of output channels.

```
deviceWriter = audioDeviceWriter;
info(deviceWriter)
```

Get Audio I/O Device Information

Create an object of the `audioPlayerRecorder` System object™ and then call `info` to return a structure containing information about the selected driver, device name, and the maximum number of input and output channels.

```
playRec = audioPlayerRecorder;
info(playRec)
```

Input Arguments

`obj` — System object to get information from
System object

System object to get information from.

Output Arguments

`infoStruct` — Struct containing object information
struct

Struct containing information about the System object, obj. Fields of the struct depend on the System object.

See Also

System Objects
`audioDeviceReader` | `audioDeviceWriter` | `audioPlayerRecorder`

Introduced in R2016a

cost

Estimate implementation cost of audio System objects

Syntax

```
implementationCost = cost(audioObj)
```

Description

`implementationCost = cost(audioObj)` returns a structure, `implementationCost`, whose fields contain information about the computation cost of implementing the audio System object, `audioObj`.

Examples

Estimate Implementation Cost of Crossover Filter

Create a crossover filter with 2 crossovers with 48 dB/octave slopes. Call `cost` to get an estimate of the implementation cost.

```
crossFilt = crossoverFilter('NumCrossovers',2,'CrossoverSlopes',48);
cost1 = cost(crossFilt)
```

Reduce the crossover slopes for both crossovers to 12 dB/octave. Call `cost` to get an estimate of the new implementation cost.

```
crossFilt.CrossoverSlopes = 12;
cost2 = cost(crossFilt)
```

Input Arguments

audioObj — Audio System object
crossoverFilter object

Specify the input as a supported audio System object.

Data Types: object

Output Arguments

implementationCost — Estimate of implementation cost
struct

Estimate of the implementation cost of a filter, returned as struct:

Structure Field	Description
NumCoefficients	Number of filter coefficients (excluding coefficients with values 0, 1 or -1)
NumStates	Number of states
MultiplicationsPerInputSample	Number of multiplication per input sample
AdditionsPerInputSample	Number of additions per input sample

See Also

crossoverFilter,

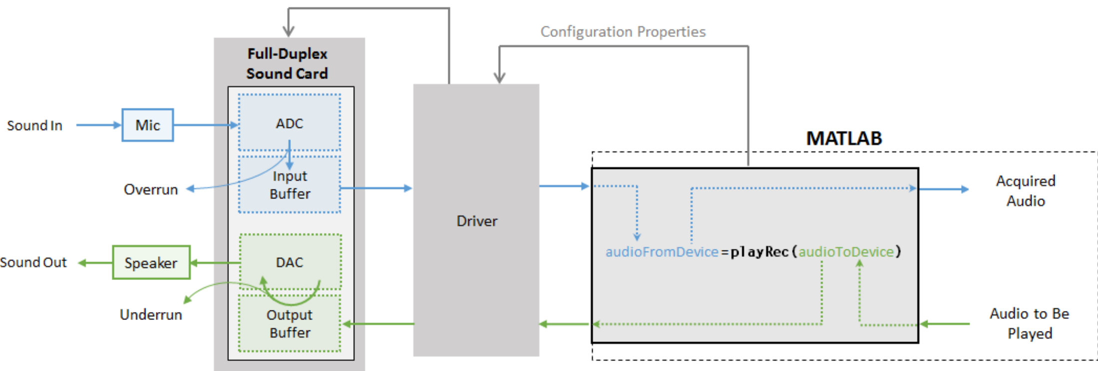
Introduced in R2016a

audioPlayerRecorder System object

Simultaneously play and record using an audio device

Description

The `audioPlayerRecorder` System object reads and writes audio samples using your computer’s audio device. To use `audioPlayerRecorder`, you must have an audio device and driver capable of simultaneous playback and record.



See “Audio I/O: Buffering, Latency, and Throughput” for a detailed explanation of the data flow.

To simultaneously play and record:

1. Create the audioPlayerRecorder object and set its properties.
2. Call the object with arguments, as if it were a function.

To learn more about how System objects work, see What Are System Objects? (MATLAB).

Creation

Syntax

```
playRec = audioPlayerRecorder
playRec = audioPlayerRecorder(sampleRateValue)
playRec = audioPlayerRecorder( ___ ,Name,Value)
```

Description

`playRec = audioPlayerRecorder` returns a System object, `playRec`, that plays audio samples to an audio device and records samples from the same audio device, in real time.

`playRec = audioPlayerRecorder(sampleRateValue)` sets the SampleRate property to `sampleRateValue`.

`playRec = audioPlayerRecorder( ___ ,Name,Value)` sets each property `Name` to the specified `Value`. Unspecified properties have default values.

Example: `playRec = audioPlayerRecorder(48000,'BitDepth','8-bit integer')` creates a System object, `playRec`, that operates at a 48 kHz sample rate and an 8-bit integer bit depth.

Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see System Design in MATLAB Using System Objects (MATLAB).

Device — Device used to play and record audio data
default audio device (default) | character vector | string

Device used to play and record audio data, specified as a character vector or string. The object supports only devices enabled for simultaneous playback and recording (full-duplex mode). Use `getAudioDevices` to list available devices.

Supported drivers for `audioPlayerRecorder` are platform-specific:

- Windows –– ASIO
- Mac –– CoreAudio
- Linux –– ALSA

Note The default audio device is the default device of your machine only if it supports full-duplex mode. If your machine's default audio device does not support full-duplex mode, `audioPlayerRecorder` specifies as the default device the first available device it detects that is capable of full-duplex mode. Use the `info` method to get the device name associated with your `audioPlayerRecorder` object.

Data Types: `char` | `string`

`SampleRate` — Sample rate used by device to record and play audio data (Hz)
`44100` (default) | positive integer

Sample rate used by device to record and play audio data, in Hz, specified as a positive integer. The range of `SampleRate` depends on your audio hardware.

Data Types: `single` | `double`

`BitDepth` — Data type used by device
`'16-bit integer'` (default) | `'8-bit integer'` | `'32-bit float'` | `'24-bit integer'`

Data type used by device, specified as a character vector or string.

Data Types: `char` | `string`

`SupportVariableSize` — Support variable frame size
`false` (default) | `true`

Option to support variable frame size, specified as `false` or `true`.

- `false` –– If the `audioPlayerRecorder` object is locked, the input must have the same frame size at each call. The buffer size of your audio device is the same as the

input frame size. If you are using the object on Windows, open the ASIO UI to set the sound card buffer to the frame size value.

- `true` –- If the `audioPlayerRecorder` object is locked, the input frame size can change at each call. The buffer size of your audio device is specified through the BufferSize property.

To minimize latency, set `SupportVariableSize` to `false`. If variable-size input is required by your audio system, set `SupportVariableSize` to `true`.

Data Types: `logical`

`BufferSize` — Buffer size of audio device
`1024` (default) | positive integer

Buffer size of audio device, specified as a positive integer.

Note If you are using the object on a Windows machine, use `asiosettings` to set the sound card buffer size to the `BufferSize` value of your `audioPlayerRecorder` System object.

Dependencies

To enable this property, set SupportVariableSize to `true`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`PlayerChannelMapping` — Mapping between columns of played data and channels of device
`[]` (default) | scalar | vector

Mapping between columns of played data and channels of output device, specified as a scalar or as a vector of valid channel indices. The default value of this property is `[]`, which means that the default channel mapping is used.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`RecorderChannelMapping` — Mapping between channels of device and columns of recorded data
`1` (default) | scalar | vector

Mapping between channels of your audio device and columns of recorded data, specified as a scalar or as a vector of valid channel indices. The default value is `1`, which means that the first recording channel on the device is used to acquire data and is mapped to a single-column matrix.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Usage

Syntax

```
audioFromDevice = playRec(audioToDevice)
[audioFromDevice,numUnderrun] = playRec(audioToDevice)
[audioFromDevice,numUnderrun,numOverrun] = playRec(audioToDevice)
```

Description

`audioFromDevice = playRec(audioToDevice)` writes one frame of audio samples, `audioToDevice`, to the selected audio device, and returns one frame of audio, `audioFromDevice`.

`[audioFromDevice,numUnderrun] = playRec(audioToDevice)` returns the number of samples overrun since the last call to `playRec`.

`[audioFromDevice,numUnderrun,numOverrun] = playRec(audioToDevice)` returns the number of samples underrun since the last call to `playRec`.

Note: When you call the `audioPlayerRecorder` System object, the audio device specified by the `Device` property is locked. An audio device can be locked by only one `audioPlayerRecorder` at a time. To release the audio device, call `release` on the `audioPlayerRecorder` System object.

Input Arguments

`audioToDevice` — Audio to device
matrix

Audio signal to write to device, specified as a matrix. The columns of the matrix are treated as independent audio channels.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8`

Output Arguments

`audioFromDevice` — Audio from device
matrix

Audio signal read from device, returned as a matrix the same size and data type as `audioToDevice`.

Data Types: `single` | `double` | `int16` | `int32` | `uint8`

`numUnderrun` — Number of samples underrun
scalar

Number of samples by which the player queue was underrun since the last call to `playRec`.

Data Types: `uint32`

`numOverrun` — Number of samples overrun
scalar

Number of samples by which the recorder queue was overrun since the last call to `playRec`.

Data Types: `uint32`

Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

Specific to audioPlayerRecorder

getAudioDevices	List available audio devices
info	Get audio device information

Common to All System Objects

clone	Create duplicate System object
isLocked	Determine if System object is in use
release	Release resources and allow changes to System object property values and input characteristics
reset	Reset internal states of System object
step	Run System object algorithm

Examples

Synchronize Playback and Recording

Synchronize playback and recording using a single audio device. If synchronization is lost, print information about samples dropped.

Create objects to read from and write to an audio file. Create an `audioPlayerRecorder` object to play an audio signal to your device and simultaneously record audio from your device.

```
fileReader = dsp.AudioFileReader('Counting-16-44p1-mono-15secs.wav', ...
    'SamplesPerFrame',256);
fs = fileReader.SampleRate;

fileWriter = dsp.AudioFileWriter('Counting-PlaybackRecorded.wav', ...
    'SampleRate',fs);

aPR = audioPlayerRecorder('SampleRate',fs);
```

In a frame-based loop:

1. Read an audio signal from your file.
2. Play the audio signal to your device and simultaneously record audio from your device. Use the optional `nUnderruns` and `nOverruns` output arguments to track any loss of synchronization.
3. Write your recorded audio to a file.

Once the loop is completed, release the objects to free devices and resources.

```
while ~isDone(fileReader)
    audioToPlay = fileReader();
```

```
    [audioRecorded,nUnderruns,nOverruns] = aPR(audioToPlay);

    fileWriter(audioRecorded)

    if nUnderruns > 0
        fprintf('Audio player queue was underrun by %d samples.\n',nUnderruns);
    end
    if nOverruns > 0
        fprintf('Audio recorder queue was overrun by %d samples.\n',nOverruns);
    end
end

release(fileReader);
release(fileWriter);
release(aPR);
```

```
Audio player queue was underrun by 2304 samples.
```

Specify Nondefault Channel Mapping

The `audioPlayerRecorder` System object™ enables you to specify a nondefault mapping between the channels of your audio device and the data sent to and received from your audio device. To run this example, your audio device must have at least two channels and be capable of full-duplex mode.

Using Default Settings

Create an `audioPlayerRecorder` object with default settings. The `audioPlayerRecorder` is automatically configured to a compatible device and driver.

```
aPR = audioPlayerRecorder;
```

The `audioPlayerRecorder` combines reading from your device and writing to your device in a single call: `audioFromDevice = aPR(audioToDevice)`. Calling the `audioPlayerRecorder` with default settings:

- Maps columns of `audioToDevice` to output channels of your device
- Maps input channels of your device to columns of `audioFromDevice`

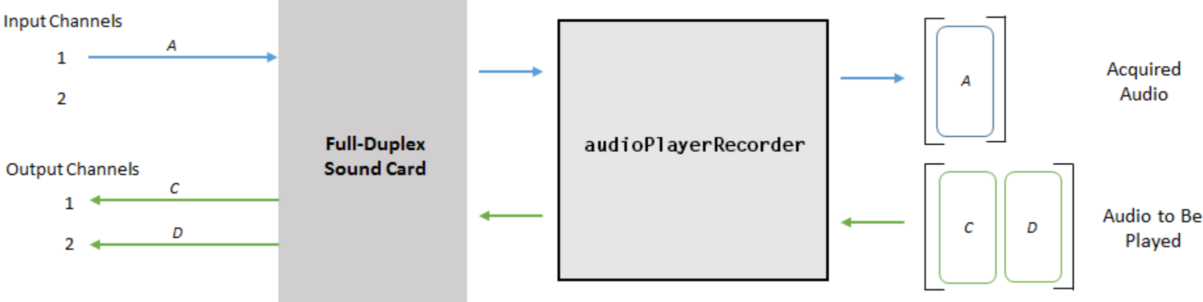
By default, `audioFromDevice` is a one-column matrix corresponding to channel 1 of your audio device. To view the maximum number of input and output channels of your device, use the `info` method.

```
aPRInfo = info(aPR);
```

`aPRInfo` is returned as a structure with fields containing information about your selected driver, audio device, and the maximum number of input and output channels in your configuration.

Call the `audioPlayerRecorder` with a two-column matrix. By default, column 1 is mapped to output channel 1, and column 2 is mapped to output channel 2. The `audioPlayerRecorder` returns a one-column matrix with the same number of rows as the `audioToDevice` matrix.

```
highToneGenerator = audioOscillator('Frequency',600,'SamplesPerFrame',256);
lowToneGenerator = audioOscillator('Frequency',200,'SamplesPerFrame',256);

for i = 1:250
    C = highToneGenerator();
    D = lowToneGenerator();
    audioToDevice = [C,D];
    audioFromDevice = aPR(audioToDevice);
end
```

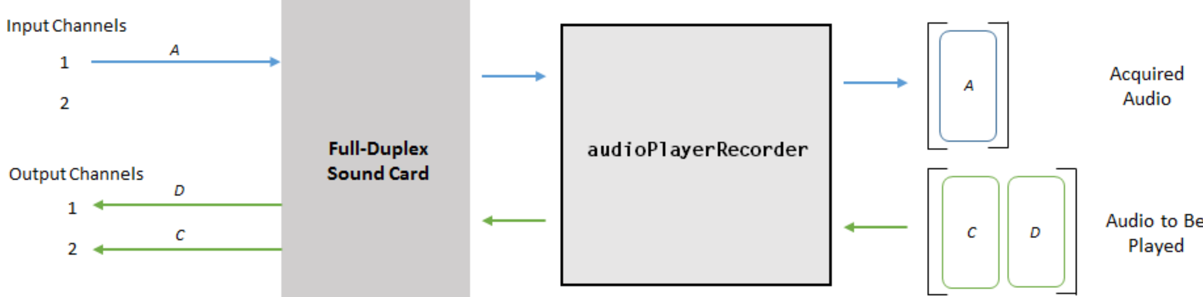


Nondefault Channel Mapping for Audio Output

Specify a nondefault channel mapping for your audio output. Specify that column 1 of `audioToDevice` maps to channel 2, and that column 2 of `audioToDevice` maps to channel 1. To modify the channel mapping, the `audioPlayerRecorder` object must be unlocked.

Run the `audioPlayerRecorder` object. If you are using headphones or stereo speakers, notice that the high frequency and low frequency tones have switched speakers.

```
release(aPR)
aPR.PlayerChannelMapping = [2,1];

for i = 1:250
    C = highToneGenerator();
    D = lowToneGenerator();
    audioToDevice = [C,D];
    audioFromDevice = aPR(audioToDevice);
end
```



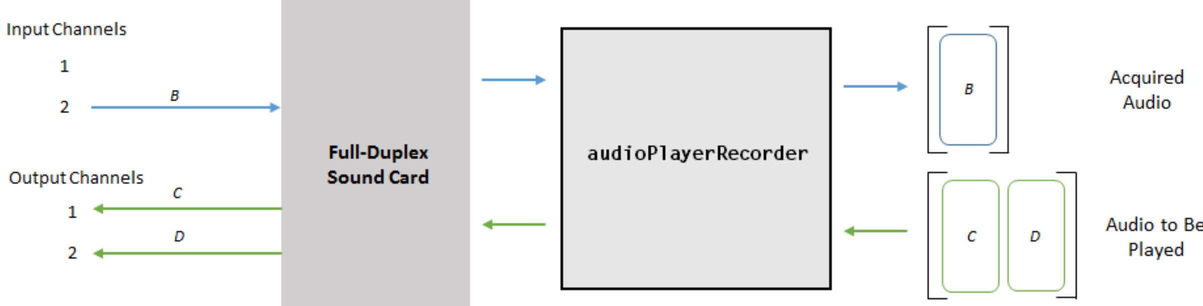
Nondefault Channel Mapping for Audio Input

Specify a nondefault channel mapping for your audio input. Record data from only channel two of your device. In this case, channel 2 is mapped to a one-column matrix. Use `size` to verify that `audioFromDevice` is a 256-by-1 matrix.

```
release(aPR)
aPR.RecorderChannelMapping = 2;

audioFromDevice = aPR(audioToDevice);

[rows,col] = size(audioFromDevice)
```

```
rows =

   256
```

```
col =

     1
```



As a best practice, release your audio device once complete.

```
release(aPR)
```

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- “System Objects in MATLAB Code Generation” (MATLAB Coder)
- The executable generated from this System object relies on prebuilt dynamic library files (`.dll` files) included with MATLAB. Use the `packNGo` function to package the code generated from this object and all the relevant files in a compressed zip file. Using this zip file, you can relocate, unpack, and rebuild your project in another development environment where MATLAB is not installed. For more details, see “Run Audio I/O Features Outside MATLAB and Simulink”.

See Also

Functions
asiosettings | getAudioDevices

Blocks
Audio Device Reader | Audio Device Writer

System Objects
audioDeviceReader | audioDeviceWriter | dsp.AudioFileReader

Topics
"Audio I/O: Buffering, Latency, and Throughput"
"Run Audio I/O Features Outside MATLAB and Simulink"
"Real-Time Audio in MATLAB"

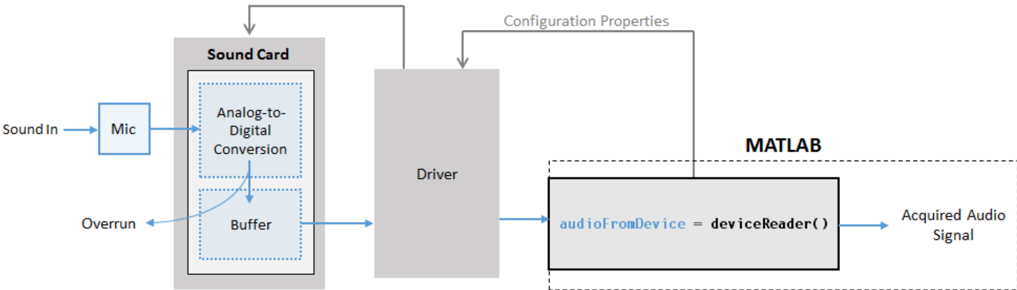
Introduced in R2017a

audioDeviceReader System object

Record from sound card

Description

The `audioDeviceReader` System object reads audio samples using your computer’s audio input device.



See “Audio I/O: Buffering, Latency, and Throughput” for a detailed explanation of the audio device reader data flow.

The audio device reader specifies the driver, the device and its attributes, and the data type and size output from your System object.

To stream data from an audio device:

1. Create the audioDeviceReader object and set its properties.
2. Call the object with arguments, as if it were a function.

To learn more about how System objects work, see What Are System Objects? (MATLAB).

Creation

Syntax

```
deviceReader = audioDeviceReader
deviceReader = audioDeviceReader(sampleRateValue)
deviceReader = audioDeviceReader(sampleRateValue,sampPerFrameValue)
deviceReader = audioDeviceReader( ___ ,Name,Value)
```

Description

`deviceReader = audioDeviceReader` returns a System object, `deviceReader`, that reads audio samples using an audio input device in real time.

`deviceReader = audioDeviceReader(sampleRateValue)` sets the SampleRate property to `sampleRateValue`.

`deviceReader = audioDeviceReader(sampleRateValue,sampPerFrameValue)` sets the SamplesPerFrame property to `sampPerFrameValue`.

`deviceReader = audioDeviceReader( ___ ,Name,Value)` sets each property `Name` to the specified `Value`. Unspecified properties have default values.

Example: `deviceReader = audioDeviceReader(16000,'BitDepth','8-bit integer')` creates a System object, `deviceReader`, that operates at a 16 kHz sample rate and an 8-bit integer bit depth.

Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see System Design in MATLAB Using System Objects (MATLAB).

`Driver` — Driver used to access audio device (Windows only)
`'DirectSound'` (default) | `'ASIO'` | `'WASAPI'`

Driver used to access your audio device, specified as `'DirectSound'`, `'ASIO'`, or `'WASAPI'`.

- ASIO drivers do not come pre-installed on Windows machines. To use the `'ASIO'` driver option, install an ASIO driver outside of MATLAB.

  Note If `Driver` is specified as `'ASIO'`, use `asiosettings` to set the sound card buffer size to the `SamplesPerFrame` value of your `audioDeviceReader` System object.

- WASAPI drivers are supported for exclusive-mode only.

ASIO and WASAPI drivers do not provide sample rate conversion. For ASIO and WASAPI drivers, set `SampleRate` to a sample rate supported by your audio device.

This property applies only on Windows machines. Linux machines always use the ALSA driver. Mac machines always use the CoreAudio driver.

Data Types: `char` | `string`

`Device` — Device used to acquire audio samples
default audio device (default) | character vector | string

Device used to acquire audio samples, specified as a character vector or string. Use `getAudioDevices` to list available devices for the selected driver.

Data Types: `char` | `string`

`NumChannels` — Number of input channels acquired by audio device
`1` (default) | integer

Number of input channels acquired by audio device, specified as an integer. The range of `NumChannels` depends on your audio hardware.

Dependencies

To enable this property, set ChannelMappingSource to `'Auto'`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`SamplesPerFrame` — Frame size read from audio device
`1024` (default) | integer

Frame size read from audio device, specified as a positive integer. `SamplesPerFrame` is also the size of your device buffer and the number of columns of the output matrix returned by your `audioDeviceReader` object.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`SampleRate` — Sample rate used by device to acquire audio data (Hz)
`44100` (default) | positive integer

Sample rate used by device to acquire audio data, in Hz, specified as a positive integer. The range of `SampleRate` depends on your audio hardware.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`BitDepth` — Data type used by device to acquire audio data
`'16-bit integer'` (default) | `'8-bit integer'` | `'32-bit float'` | `'24-bit integer'`

Data type used by device to acquire audio data, specified as a character vector or string.

Data Types: `char` | `string`

`ChannelMappingSource` — Source of mapping between device channels and output matrix
`'Auto'` (default) | `'Property'`

Source of mapping between the channels of your audio input device and columns of the output matrix, specified as `'Auto'` or `'Property'`.

- `'Auto'` –– The default settings determine the mapping between device channels and output matrix. For example, suppose that your audio device has six channels available, and you set `NumChannels` to 6. The output from a call to your audio device reader is a six-column matrix. Column 1 corresponds to channel 1, column 2 corresponds to channel 2, and so on.
- `'Property'` –– The ChannelMapping property determines the mapping between channels of your audio device and columns of the output matrix.

Data Types: `char` | `string`

ChannelMapping — Nondefault mapping between device channels and output matrix
[1:MaximumInputChannels] (default) | scalar | vector

Nondefault mapping between channels of your audio input device and columns of the output matrix, specified as a vector of valid channel indices. See “Specify Channel Mapping for audioDeviceReader” on page 3-98 for more information.

Dependencies

To enable this property, set ChannelMappingSource to `'Property'`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

OutputDataType — Data type of the output
`'double'` (default) | `'single'` | `'int32'` | `'int16'` | `'uint8'`

Data type of the output, specified as a character vector or string.

Note If `OutputDataType` is specified as `'double'` or `'single'`, the audio device reader outputs data in the range [–1, 1]. For other data types, the range is [min, max] of the specified data type.

Data Types: `char` | `string`

Usage

Note For versions earlier than R2016b, use the `step` function to run the System object™ algorithm. The arguments to `step` are the object you created, followed by the arguments shown in this section.

For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Syntax

```
audioFromDevice = deviceReader()
```

```
[audioFromDevice,numOverrun] = deviceReader()
```

Description

`audioFromDevice = deviceReader()` returns one frame of audio samples from the selected audio input device.

`[audioFromDevice,numOverrun] = deviceReader()` returns the number of samples by which the audio reader's queue was overrun since the last call to `deviceReader`.

Note: When you call the `audioDeviceReader` System object, the audio device specified by the Device property is locked. An audio device can be locked by only one `audioDeviceReader` at a time. To release the audio device, call `release` on your `audioDeviceReader` object.

Output Arguments

`audioFromDevice` — Audio from device

matrix

Audio signal read from device, returned as a matrix. The specified number of channels and the SamplesPerFrame property determine the matrix size. The data type of the matrix depends on the OutputDataType property.

Data Types: `single` | `double` | `int16` | `int32` | `uint8`

`numOverrun` — Number of samples overrun

scalar

Number of samples by which the audio reader's queue was overrun since the last call to `deviceReader`.

Data Types: `uint32`

Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

Specific to audioDeviceReader

getAudioDevices	List available audio devices
info	Get audio device information

Common to All System Objects

clone	Create duplicate System object
isLocked	Determine if System object is in use
release	Release resources and allow changes to System object property values and input characteristics
reset	Reset internal states of System object
step	Run System object algorithm

Examples

Read from Microphone and Write to Audio File

Record 10 seconds of speech with a microphone and send the output to a `.wav` file.

Create an `audioDeviceReader` System object™ with default settings. Call `setup` to reduce the computational load of initialization in an audio stream loop.

```
deviceReader = audioDeviceReader;
setup(deviceReader);
```

Create a `dsp.AudioFileWriter` System object. Specify the file name and type to write.

```
fileWriter = dsp.AudioFileWriter(...
    'mySpeech.wav',...
    'FileFormat','WAV');
```

Record 10 seconds of speech. In an audio stream loop, read an audio signal frame from the device, and write the audio signal frame to a specified file. The file saves to your current folder.

```
disp('Speak into microphone now.')
tic;
while toc < 10
    acquiredAudio = deviceReader();
    fileWriter(acquiredAudio);
```

```
end
disp('Recording complete.')
```

Release the audio device and close the output file.

```
release(deviceReader);
release(fileWriter);
```

Reduce Latency Due to Input Device Buffer

Latency due to the input device buffer is the time delay of acquiring one frame of data. In this example, you modify default properties of your `audioDeviceReader` System object™ to reduce latency.

Create an `audioDeviceReader` System object with default settings.

```
deviceReader = audioDeviceReader
```

Calculate the latency due to your device buffer.

```
fprintf('Latency due to device buffer: %f seconds.\n',...
    deviceReader.SamplesPerFrame/deviceReader.SampleRate)
```

Set the `SamplesPerFrame` property of your `audioDeviceReader` System object to 64. Calculate the latency.

```
deviceReader.SamplesPerFrame = 64;
fprintf('Latency due to device buffer: %f seconds.\n',...
    deviceReader.SamplesPerFrame/deviceReader.SampleRate)
```

Set the `SampleRate` property of your `audioDeviceReader` System object to 96,000. Calculate the latency.

```
deviceReader.SampleRate = 96000;
fprintf('Latency due to device buffer: %f seconds.\n',...
    deviceReader.SamplesPerFrame/deviceReader.SampleRate)
```

Determine and Decrease Overrun

Overrun refers to input signal drops, which occur when the audio stream loop does not keep pace with the device. Determine overrun of an audio stream loop, add an artificial

computational load to the audio stream loop, and then modify properties of your `audioDeviceReader` System object™ to decrease overrun. Your results depend on your computer.

Create an `audioDeviceReader` System object with `SamplesPerFrame` set to 256 and `SampleRate` set to 44,100. Call `setup` to reduce the computational load of initialization in an audio stream loop.

```
deviceReader = audioDeviceReader(...
    'SamplesPerFrame',256,...
    'SampleRate',44100);
setup(deviceReader);
```

Create a `dsp.AudioFileWriter` System object. Specify the file name and data type to write.

```
fileWriter = dsp.AudioFileWriter(...
    'mySpeech.wav',...
    'FileFormat','WAV');
```

Record 5 seconds of speech. In an audio stream loop, read an audio signal frame from your device, and write the audio signal frame to a specified file.

```
totalOverrun = 0;
disp('Speak into microphone now.')
tic;
while toc < 5
    [input,numOverrun] = deviceReader();
    totalOverrun = totalOverrun + numOverrun;
    fileWriter(input);
end
fprintf('Recording complete.\n')
fprintf('Total number of samples overrun: %d.\n',...
    totalOverrun);
fprintf('Total seconds overrun: %d.\n',...
    double(totalOverrun)/double(deviceReader.SampleRate));
```

```
Speak into microphone now.
Recording complete.
Total number of samples overrun: 1280.
Total seconds overrun: 2.902494e-02.
```

Release your `audioDeviceReader` and `dsp.AudioDeviceWriter` System objects and zero your counter variable.

```
release(fileWriter);
release(deviceReader);
totalOverrun = 0;
```

Use `pause` to add an artificial computational load to your audio stream loop. The computational load causes the audio stream loop to go slower than the device, which causes acquired samples to be dropped.

```
disp('Speak into microphone now.')
tic;
while toc < 5
    [input,numOverrun] = deviceReader();
    totalOverrun = totalOverrun + numOverrun;
    fileWriter(input);
    pause(0.01)
end
fprintf('Recording complete.\n')
fprintf('Total number of samples overrun: %d.\n',...
    totalOverrun);
fprintf('Total seconds overrun: %d.\n',...
    double(totalOverrun)/double(deviceReader.SampleRate));
```

```
Speak into microphone now.
Recording complete.
Total number of samples overrun: 96256.
Total seconds overrun: 2.182676e+00.
```

Release your `audioDeviceReader` and `dsp.AudioFileWriter` System objects, and set the `SamplePerFrame` property to 512. The device buffer size increases so that the device now takes longer to acquire a frame of data. Set your counter variable to zero.

```
release(fileWriter);
release(deviceReader);
deviceReader.SamplesPerFrame = 512;
totalOverrun = 0;
```

Calculate the total overrun of the audio stream loop using your modified `SamplesPerFrame` property.

```
disp('Speak into microphone now.')
tic;
while toc < 5
    [input,numOverrun] = deviceReader();
    totalOverrun = totalOverrun + numOverrun;
    fileWriter(input);
```

```
    pause(0.01)
end
fprintf('Recording complete.\n')
fprintf('Total number of samples overrun: %d.\n',...
    totalOverrun);
fprintf('Total seconds overrun: %f.\n',...
    totalOverrun/deviceReader.SampleRate);
```

```
Speak into microphone now.
Recording complete.
Total number of samples overrun: 2048.
Total seconds overrun: 0.000000.
```

Specify Channel Mapping for audioDeviceReader

Specify non-default channel mapping for an `audioDeviceReader` System object™. This example is hardware specific. It assumes that your computer has a default audio input device with two available channels.

Create an `audioDeviceReader` System object with default settings.

```
deviceReader = audioDeviceReader;
```

The default number of channels is 1. Call your `audioDeviceReader` System object like a function with no arguments to read one frame of data from your audio device. Verify that the output data matrix has one column.

```
x = deviceReader();
[frameLength,numChannels] = size(x)
```

```
frameLength =

        1024


numChannels =

     1
```

Use `info` to determine the maximum number of input channels available with your specified `Driver` and `Device` configuration.

```
info(deviceReader)
```

```
ans =

  struct with fields:

                  Driver: 'DirectSound'
              DeviceName: 'Primary Sound Capture Driver'
    MaximumInputChannels: 2
```

Set `ChannelMappingSource` to `'Property'`. The `audioDeviceReader` System object must be unlocked to change this property.

```
release(deviceReader);
deviceReader.ChannelMappingSource = 'Property'
```

```
deviceReader =

  audioDeviceReader with properties:

             Driver: 'DirectSound'
             Device: 'Default'
    SamplesPerFrame: 1024
         SampleRate: 44100

  Use get to show all properties
```

By default, if `ChannelMappingSource` is set to `'Property'`, all available channels are mapped to the output. Call your `audioDeviceReader` System object to read one frame of data from your audio device. Verify that the output data matrix has two columns.

```
x = deviceReader();
[frameLength,numChannels] = size(x)
```

```
frameLength =

        1024


numChannels =
```

2

Use the `ChannelMapping` property to specify an alternative mapping between channels of your device and columns of the output matrix. Indicate the input channel number at an index corresponding to the output column. To change this property, first unlock the `audioDeviceReader` System object.

```
release(deviceReader);
deviceReader.ChannelMapping = [2,1];
```

Now when you call your `audioDeviceReader`:

- Input channel 1 of your device maps to the second column of your output matrix.
- Input channel 2 of your device maps to the first column of your output matrix.

Acquire a specific channel from your input device.

```
deviceReader.ChannelMapping = 2;
```

If you call your `audioDeviceReader`, input channel 2 of your device maps to an output vector.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- “System Objects in MATLAB Code Generation” (MATLAB Coder)
- The executable generated from this System object relies on prebuilt dynamic library files (`.dll` files) included with MATLAB. Use the `packNGo` function to package the code generated from this object and all the relevant files in a compressed zip file. Using this zip file, you can relocate, unpack, and rebuild your project in another development environment where MATLAB is not installed. For more details, see “Run Audio I/O Features Outside MATLAB and Simulink”.

See Also

Functions
`asiosettings` | `getAudioDevices`

Blocks
Audio Device Reader

System Objects
`audioDeviceWriter` | `audioPlayerRecorder` | `dsp.AudioFileReader`

Topics
"Audio I/O: Buffering, Latency, and Throughput"
"Run Audio I/O Features Outside MATLAB and Simulink"
"Real-Time Audio in MATLAB"

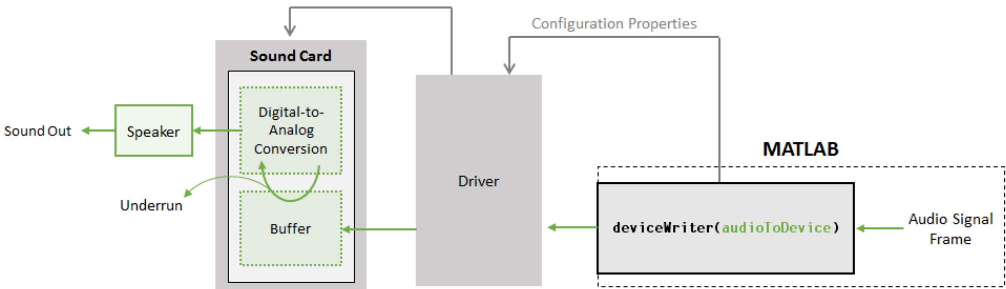
Introduced in R2016a

audioDeviceWriter System object

Play to sound card

Description

The `audioDeviceWriter` System object writes audio samples to an audio output device. Properties of the audio device writer specify the driver, the device, and device attributes such as sample rate, bit depth, and buffer size.



See “Audio I/O: Buffering, Latency, and Throughput” for a detailed explanation of the audio device writer data flow.

To stream data to an audio device:

1. Create the audioDeviceWriter object and set its properties.
2. Call the object with arguments, as if it were a function.

To learn more about how System objects work, see What Are System Objects? (MATLAB).

Creation

Syntax

```
deviceWriter = audioDeviceWriter
deviceWriter = audioDeviceWriter(sampleRateValue)
deviceWriter = audioDeviceWriter( ___ ,Name,Value)
```

Description

`deviceWriter = audioDeviceWriter` returns a System object, `deviceWriter`, that writes audio samples to an audio output device in real time.

`deviceWriter = audioDeviceWriter(sampleRateValue)` sets the SampleRate property to `sampleRateValue`.

`deviceWriter = audioDeviceWriter( ___ ,Name,Value)` sets each property `Name` to the specified `Value`. Unspecified properties have default values.

Example: `deviceWriter = audioDeviceWriter(48000,'BitDepth','8-bit integer')` creates a System object, `deviceWriter`, that operates at a 48 kHz sample rate and an 8-bit integer bit depth.

Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see System Design in MATLAB Using System Objects (MATLAB).

`Driver` — Driver used to access audio device (Windows only)
`'DirectSound'` (default) | `'ASIO'` | `'WASAPI'`

Driver used to access your audio device, specified as `'DirectSound'`, `'ASIO'`, or `'WASAPI'`.

- ASIO drivers do not come pre-installed on Windows machines. To use the `'ASIO'` driver option, install an ASIO driver outside of MATLAB.

  Note If `Driver` is specified as `'ASIO'`, use `asiosettings` to set the sound card buffer size to the buffer size of your `audioDeviceWriter` System object.

- WASAPI drivers are supported for exclusive-mode only.

ASIO and WASAPI drivers do not provide sample rate conversion. For ASIO and WASAPI drivers, set `SampleRate` to a sample rate supported by your audio device.

This property applies only on Windows machines. Linux machines always use the ALSA driver. Mac machines always use the CoreAudio driver.

To specify nondefault `Driver` values, you must have an Audio System Toolbox licence. If the toolbox is not installed, specifying nondefault `Driver` values returns an error.

Data Types: `char` | `string`

Device — Device used to play audio samples
default audio device (default) | character vector | string

Device used to play audio samples, specified as a character vector or string. Use `getAudioDevices` to list available devices for the selected driver.

Data Types: `char` | `string`

SampleRate — Sample rate of signal sent to audio device (Hz)
`44100` (default) | positive integer

Sample rate of signal sent to audio device, in Hz, specified as a positive integer. The range of `SampleRate` depends on your audio hardware.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

BitDepth — Data type used by the device
`'16-bit integer'` (default) | `'8-bit integer'` | `'24-bit integer'` | `'32-bit float'`

Data type used by the device, specified as a character vector or string. Before performing digital-to-analog conversion, the input data is cast to a data type specified by `BitDepth`.

To specify a nondefault `BitDepth`, you must have an Audio System Toolbox licence. If the toolbox is not installed, specifying a nondefault `BitDepth` returns an error.

Data Types: `char` | `string`

`SupportVariableSizeInput` — Support variable frame size
`false` (default) | `true`

Option to support variable frame size, specified as `true` or `false`.

- `false` –– If the `audioDeviceWriter` object is locked, the input must have the same frame size at each call. The buffer size of your audio device is the same as the input frame size.
- `true` –– If the `audioDeviceWriter` object is locked, the input frame size can change at each call. The buffer size of your audio device is specified through the BufferSize property.

Data Types: `char`

`BufferSize` — Buffer size of audio device
`4096` (default) | positive integer

Buffer size of audio device, specified as a positive integer.

Note If `Driver` is specified as `'ASIO'`, open the ASIO UI to set the sound card buffer size to the `BufferSize` value of your `audioDeviceWriter` System object.

Dependencies

To enable this property, set SupportVariableSizeInput to `true`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`ChannelMappingSource` — Source of mapping between input matrix and device channels
`'Auto'` (default) | `'Property'`

Source of mapping between columns of input matrix and channels of audio output device, specified as `'Auto'` or `'Property'`.

- 'Auto' –– Default settings determine the mapping between columns of input matrix and channels of audio output device. For example, suppose that your input is a matrix with four columns, and your audio device has four channels available. Column 1 of your input data writes to channel 1 of your device, column 2 of your input data writes to channel 2 of your device, and so on.
- 'Property' –– The ChannelMapping property determines the mapping between columns of input matrix and channels of audio output device.

Data Types: char | string

ChannelMapping — Nondefault mapping between input matrix and device channels

[1:MaximumOutputChannels] (default) | scalar | vector

Nondefault mapping between columns of input matrix and channels of output device, specified as a scalar or vector of valid channel indices. See the “Specify Channel Mapping for audioDeviceWriter” on page 3-112 example for more information.

To selectively map between columns of the input matrix and your sound card's output channels, you must have an Audio System Toolbox licence. If the toolbox is not installed, specifying a nondefault ChannelMapping returns an error.

Dependencies

To enable this property, set ChannelMappingSource to 'Property'.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

Usage

Note For versions earlier than R2016b, use the step function to run the System object™ algorithm. The arguments to step are the object you created, followed by the arguments shown in this section.

For example, y = step(obj,x) and y = obj(x) perform equivalent operations.

Syntax

`numUnderrun = deviceWriter(audioToDevice)`

Description

`numUnderrun = deviceWriter(audioToDevice)` writes one frame of audio samples, `audioToDevice`, to the selected audio device and returns the number of audio samples underrun since the last call to `deviceWriter`.

Note: When you call the `audioDeviceWriter` System object, the audio device specified by the `Device` property is locked. An audio device can be locked by only one `audioDeviceWriter` at a time. To release the audio device, call `release` on your `audioDeviceWriter` System object.

Input Arguments

`audioToDevice` — Audio to device
matrix

Audio signal to write to device, specified as a matrix. The columns of the matrix are treated as independent audio channels.

If `audioToDevice` is of data type `'double'` or `'single'`, the audio device writer clips values outside the range [–1, 1]. For other data types, the allowed input range is [min, max] of the specified data type.

Data Types: `single` | `double` | `int16` | `int32` | `uint8`

Output Arguments

`numUnderrun` — Number of samples underrun
scalar

Number of samples by which the audio device writer queue was underrun since the last call to `deviceWriter`.

Data Types: `uint32`

Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

Specific to audioDeviceWriter

getAudioDevices	List available audio devices
info	Get audio device information

Common to All System Objects

clone	Create duplicate System object
isLocked	Determine if System object is in use
release	Release resources and allow changes to System object property values and input characteristics
reset	Reset internal states of System object
step	Run System object algorithm

Examples

Read from File and Write to Audio Device

Read an MP3 audio file and play it through your default audio output device.

Create a `dsp.AudioFileReader` System object™ with default settings. Use the `audioinfo` function to return a structure containing information about the audio file.

```
fileReader = dsp.AudioFileReader('speech_dft.mp3');
fileInfo = audioinfo('speech_dft.mp3');
```

Create an `audioDeviceWriter` System object and specify the sample rate. Call `setup` to reduce the computational load of initialization in an audio stream loop.

```
deviceWriter = audioDeviceWriter(...
    'SampleRate',fileInfo.SampleRate);
setup(deviceWriter,...
    zeros(fileReader.SamplesPerFrame,fileInfo.NumChannels));
```

In an audio stream loop, read an audio signal frame from the file, and write the frame to your device.

```
while ~isDone(fileReader)
    audioData = fileReader();
    deviceWriter(audioData);
end
```

Close the input file and release the device.

```
release(fileReader);
release(deviceWriter);
```

Reduce Latency due to Output Device Buffer

Latency due to the output device buffer is the time delay of writing one frame of data. Modify default properties of your `audioDeviceWriter` System object™ to reduce latency due to device buffer size.

Create a `dsp.AudioFileReader` System object to read an audio file with default settings.

```
fileReader = dsp.AudioFileReader('speech_dft.mp3');
```

Create an `audioDeviceWriter` System object and specify the sample rate to match that of the audio file reader.

```
deviceWriter = audioDeviceWriter(...
    'SampleRate',fileReader.SampleRate);
```

Calculate the latency due to your device buffer, in seconds.

```
bufferLatency = fileReader.SamplesPerFrame/deviceWriter.SampleRate %#ok
```

```
bufferLatency = 0.0464
```

Set the `SamplesPerFrame` property of your `dsp.AudioFileReader` System object to 256. Calculate the buffer latency in seconds.

```
fileReader.SamplesPerFrame = 256;
bufferLatency = fileReader.SamplesPerFrame/deviceWriter.SampleRate
```

```
bufferLatency = 0.0116
```

Determine and Decrease Underrun

Underrun refers to output signal silence, which occurs when the audio stream loop does not keep pace with the output device. Determine the underrun of an audio stream loop, add artificial computational load to the audio stream loop, and then modify properties of your `audioDeviceWriter` System object™ to decrease underrun. Your results depend on your computer.

Create a `dsp.AudioFileReader` System object, and specify the file to read. Use the `audioinfo` function to return a structure containing information about the audio file.

```
fileReader = dsp.AudioFileReader('speech_dft.mp3');
fileInfo = audioinfo('speech_dft.mp3');
```

Create an `audioDeviceWriter` System object. Use the `SampleRate` of the file reader as the `SampleRate` of the device writer. Call `setup` to reduce the computational load of initialization in an audio stream loop.

```
deviceWriter = audioDeviceWriter( ...
    'SampleRate',fileReader.SampleRate);
setup(deviceWriter, ...
    zeros(fileReader.SamplesPerFrame,fileInfo.NumChannels));
```

Run your audio stream loop with input from file and output to device. Print the total samples underrun and the underrun in seconds.

```
totalUnderrun = 0;
while ~isDone(fileReader)
    input = fileReader();
    numUnderrun = deviceWriter(input);
    totalUnderrun = totalUnderrun + numUnderrun;
end
fprintf('Total samples underrun: %d.\n', ...
    totalUnderrun);
fprintf('Total seconds underrun: %d.\n', ...
    double(totalUnderrun)/double(deviceWriter.SampleRate));
```

```
Total samples underrun: 0.
Total seconds underrun: 0.
```

Release your `dsp.AudioFileReader` and `audioDeviceWriter` System objects and set your counter variable to zero.

```
release(fileReader);
release(deviceWriter);
totalUnderrun = 0;
```

Use `pause` to mimic an algorithm that takes 0.075 seconds to process. The pause causes the audio stream loop to go slower than the device, which results in periods of silence in the output audio signal.

```
while ~isDone(fileReader)
    input = fileReader();
    numUnderrun = deviceWriter(input);
    totalUnderrun = totalUnderrun + numUnderrun;
    pause(0.075)
end
fprintf('Total samples underrun: %d.\n', ...
    totalUnderrun);
fprintf('Total seconds underrun: %d.\n', ...
    double(totalUnderrun)/double(deviceWriter.SampleRate));
```

```
Total samples underrun: 69632.
Total seconds underrun: 3.157914e+00.
```

Release your `audioDeviceReader` and `dsp.AudioFileWriter` and set the counter variable to zero.

```
release(fileReader);
release(deviceWriter);
totalUnderrun = 0;
```

Set the frame size of your audio stream loop to 2048. Because the `SupportVariableSizeInput` property of your `audioDeviceWriter` System object is set to `false`, the buffer size of your audio device is the same size as the input frame size. Increasing your device buffer size decreases underrun.

```
fileReader = dsp.AudioFileReader('speech_dft.mp3');
fileReader.SamplesPerFrame = 2048;
fileInfo = audioinfo('speech_dft.mp3');

deviceWriter = audioDeviceWriter( ...
    'SampleRate',fileReader.SampleRate);
setup(deviceWriter, ...
    zeros(fileReader.SamplesPerFrame,fileInfo.NumChannels));
```

Calculate the total underrun.

```
while ~isDone(fileReader)
    input = fileReader();
    numUnderrun = deviceWriter(input);
    totalUnderrun = totalUnderrun + numUnderrun;
    pause(0.075)
end
fprintf('Total samples underrun: %d.\n', ...
    totalUnderrun);
fprintf('Total seconds underrun: %d.\n', ...
    double(totalUnderrun)/double(deviceWriter.SampleRate));
```

```
Total samples underrun: 0.
Total seconds underrun: 0.
```

The increased frame size reduces the total underrun of your audio stream loop. However, increasing the frame size also increases latency. Other approaches to reduce underrun include:

- Increasing the buffer size independent of input frame size. To increase buffer size independent of input frame size, you must first set `SupportVariableSizeInput` to `true`. This approach also increases latency.
- Decreasing the sample rate. Decreasing the sample rate reduces both latency and underrun at the cost of signal resolution.
- Choosing an optimal driver and device for your system.

Specify Channel Mapping for audioDeviceWriter

Specify nondefault channel mapping for an `audioDeviceWriter` System object™. This example is hardware specific. It assumes that your computer has a default audio output device with two available channels.

Create an `audioDeviceWriter` System object™ with default settings.

```
deviceWriter = audioDeviceWriter;
```

By default, the `audioDeviceWriter` System object writes the maximum number of channels available, corresponding to the columns of the input matrix. Use `info` to get the maximum number of channels of your device.

```
info(deviceWriter)
```

```
ans = 

  struct with fields:

                   Driver: 'DirectSound'
               DeviceName: 'Primary Sound Driver'
    MaximumOutputChannels: 2
```

If `deviceWriter` is called with one column of data, two channels are written to your audio output device. Both channels correspond to the one column of data.

Use the `audioOscillator` System object to output a tone to your `audioDeviceWriter` System object. Your object, `sineGenerator`, returns a vector when called.

```
sineGenerator = audioOscillator;
```

Write the sine tone to your audio device. If you are using headphones, you can hear the tone from both channels.

```
count = 0;
while count < 500
    sine = sineGenerator();
    deviceWriter(sine);
    count = count + 1;
end
```

If your `audioDeviceWriter` System object is called with two columns of data, two channels are written to your audio output device. The first column corresponds to channel 1 of your audio output device, and the second column corresponds to channel 2 of your audio output device.

Write a two-column matrix to your audio output device. Column 1 corresponds to the sine tone, and column 2 corresponds to a static signal. If you are using headphones, you can hear the tone from one speaker and the static from the other speaker.

```
count = 0;
while count < 500
    sine = sineGenerator();
    static = randn(length(sine),1);
    deviceWriter([sine,static]);
    count = count + 1;
end
```

Specify alternative mappings between channels of your device and columns of the output matrix by indicating the output channel number at an index corresponding to the input column. Set `ChannelMappingSource` to `'Property'`. Indicate that the first column of your input data writes to channel 2 of your output device, and that the second column of your input data writes to channel 1 of your output device. To modify the channel mapping, you must first unlock the `audioDeviceReader` System object.

```
release(deviceWriter);
deviceWriter.ChannelMappingSource = 'Property';
deviceWriter.ChannelMapping = [2,1];
```

Play your audio signals with reversed mapping. If you are using headphones, notice that the tone and static have switched speakers.

```
count = 0;
while count < 500
    sine = sineGenerator();
    static = randn(length(sine),1);
    deviceWriter([sine,static]);
    count = count + 1;
end
```

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- “System Objects in MATLAB Code Generation” (MATLAB Coder)
- The executable generated from this System object relies on prebuilt dynamic library files (`.dll` files) included with MATLAB. Use the `packNGo` function to package the code generated from this object and all the relevant files in a compressed zip file. Using this zip file, you can relocate, unpack, and rebuild your project in another development environment where MATLAB is not installed. For more details, see “Run Audio I/O Features Outside MATLAB and Simulink”.

See Also

Functions
`asiosettings` | `getAudioDevices`

Blocks
Audio Device Writer

System Objects
`audioDeviceReader` | `audioPlayerRecorder` | `dsp.AudioFileReader` | `dsp.AudioFileWriter`

Topics
"Run Audio I/O Features Outside MATLAB and Simulink"
"Audio I/O: Buffering, Latency, and Throughput"
Measure Audio Latency
"Real-Time Audio in MATLAB"

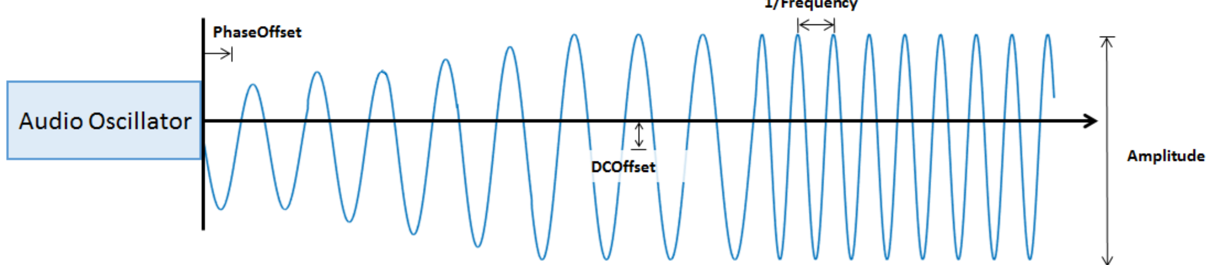
Introduced in R2016a

audioOscillator System object

Generate sine, square, and sawtooth waveforms

Description

The `audioOscillator` System object generates tunable waveforms. Typical uses include the generation of test signals for test benches, and the generation of control signals for audio effects. Properties of the `audioOscillator` System object specify the type of waveform generated.



To generate tunable waveforms:

1. Create the audioOscillator object and set its properties.
2. Call the object with arguments, as if it were a function.

To learn more about how System objects work, see What Are System Objects? (MATLAB).

Creation

Syntax

```
osc = audioOscillator
osc = audioOscillator(signalTypeValue)
```

```
osc = audioOscillator(signalTypeValue,frequencyValue)
osc = audioOscillator( ___ ,Name,Value)
```

Description

`osc = audioOscillator` creates an audio oscillator System object, `osc`, with default property values.

`osc = audioOscillator(signalTypeValue)` sets the SignalType property to `signalTypeValue`.

`osc = audioOscillator(signalTypeValue,frequencyValue)` sets the Frequency property to `frequencyValue`.

`osc = audioOscillator( ___ ,Name,Value)` sets each property `Name` to the specified `Value`. Unspecified properties have default values.

Example: `osc = audioOscillator('SignalType','sine','Frequency', 8000,'DCOffset',1)` creates a System object, `osc`, which generates 8 kHz sinusoids with a DC offset of one.

Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see System Design in MATLAB Using System Objects (MATLAB).

`SignalType` — Type of generated waveform

`'sine'` (default) | `'square'` | `'sawtooth'`

Type of waveform generated by your `audioOscillator` object, specified as `'sine'`, `'square'`, or `'sawtooth'`.

The waveforms are generated using the algorithms specified by the `sin`, `square`, and `sawtooth` functions.

Tunable: No

Data Types: `char` | `string`

`Frequency` — Frequency of generated waveform (Hz)
`100` (default) | real scalar | vector of real scalars

Frequency of generated waveform in Hz, specified as a real scalar or vector of real scalars greater than or equal to 0.

- For sine waveforms, specify `Frequency` as a scalar or as a vector of length NumTones.
- For square waveforms, specify `Frequency` as a scalar.
- For sawtooth waveforms, specify `Frequency` as a scalar.

Tunable: Yes

Data Types: `single` | `double`

`Amplitude` — Amplitude of generated waveform
`1` (default) | real scalar | vector of real scalars

Amplitude of generated waveform, specified as a real scalar or vector of real scalars greater than or equal to 0.

- For sine waveforms, specify `Amplitude` as a vector of length NumTones.
- For square waveforms, specify `Amplitude` as a scalar.
- For sawtooth waveforms, specify `Amplitude` as a scalar.

The generated waveform is multiplied by the value specified by `Amplitude` at the output, before DCOffset is applied.

Tunable: Yes

Data Types: `single` | `double`

`PhaseOffset` — Normalized phase offset of generated waveform
`0` (default) | real scalar | vector of real scalars

Normalized phase offset of generated waveform, specified as a real scalar or vector of real scalars with values in the range [0, 1]. The range is a normalized 2π-radian interval.

- For sine waveforms, specify `PhaseOffset` as a vector of length NumTones.

- For square waveforms, specify `PhaseOffset` as a scalar.
- For sawtooth waveforms, specify `PhaseOffset` as a scalar.

Tunable: No

Data Types: `single` | `double`

`DCOffset` — Value added to each element of generated waveform

`0` (default) | real scalar | vector of real scalars

Value added to each element of generated waveform, specified as a real scalar or vector of real scalars.

- For sine waveforms, specify `DCOffset` as a vector of length NumTones.
- For square waveforms, specify `DCOffset` as a scalar.
- For sawtooth waveforms, specify `DCOffset` as a scalar.

Tunable: Yes

Data Types: `single` | `double`

`NumTones` — Number of pure sine waveform tones

`1` (default) | positive integer

Number of pure sine waveform tones summed and then generated by the audio oscillator.

Individual tones are generated based on values specified by `Frequency`, `Amplitude`, `PhaseOffset`, and `DCOffset`.

Tunable: No

Dependencies

To enable this property, set SignalType to `'sine'`.

Data Types: `single` | `double`

`DutyCycle` — Square waveform duty cycle

`0.5` (default) | scalar in the range [0, 1]

Square waveform duty cycle, specified as a scalar in the range [0, 1].

Square waveform duty cycle is the percentage of one period in which the waveform is above the median amplitude. A `DutyCycle` of 1 or 0 is equivalent to a DC offset.

Tunable: Yes

Dependencies

To enable this property, set SignalType to `'square'`.

Data Types: `single` | `double`

`Width` — Sawtooth width

`1` (default) | scalar in the range [0, 1]

Sawtooth width, specified as a scalar in the range [0, 1].

Sawtooth width determines the point in a sawtooth waveform period at which the maximum occurs.

Tunable: Yes

Dependencies

To enable this property, set SignalType to `'sawtooth'`.

Data Types: `single` | `double`

`SamplesPerFrame` — Number of samples per frame

`512` (default) | positive integer

Number of samples per frame, specified as a positive integer in the range [1, 192000].

This property determines the vector length that your `audioOscillator` object outputs.

Tunable: Yes

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`SampleRate` — Sample rate of generated waveform (Hz)

`44100` (default) | positive scalar

Sample rate of generated waveform in Hz, specified as a positive scalar greater than twice the value specified by Frequency.

Tunable: Yes

Data Types: `single` | `double`

`OutputDataType` — Data type of generated waveform
`'double'` (default) | `'single'`

Data type of generated waveform, specified as `'double'` or `'single'`.

Tunable: Yes

Data Types: `char` | `string`

Usage

Note For versions earlier than R2016b, use the `step` function to run the System object™ algorithm. The arguments to `step` are the object you created, followed by the arguments shown in this section.

For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Syntax

`waveform = osc()`

Description

`waveform = osc()` generates a waveform output, `waveform`. The type of waveform is specified by the algorithm and properties of the System object, `osc`.

Output Arguments

`waveform` — Waveform output from oscillator
column vector

Waveform output from the audio oscillator, returned as a column vector with length specified by the `SamplesPerFrame` property.

Data Types: `single` | `double`

Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

Specific to audioOscillator

createAudioPluginClass	Create audio plugin class that implements functionality of System object

MIDI

configureMIDI	Configure MIDI connections between audio object and MIDI controller
disconnectMIDI	Disconnect MIDI controls from audio object
getMIDIConnections	Get MIDI connections of audio object

Common to All System Objects

clone	Create duplicate System object
isLocked	Determine if System object is in use
release	Release resources and allow changes to System object property values and input characteristics
reset	Reset internal states of System object
step	Run System object algorithm

The `createAudioPluginClass` and `configureMIDI` functions map tunable properties of the `audioOscillator` System object to user-facing parameters:

Property	Range	Mapping	Units
`Frequency`	[0.1, 20000]	log	Hz
`Amplitude`	[0, 10]	linear	no units
`DCOffset`	[–10, 10]	linear	no units
`DutyCycle` (available when you set `SignalType` to `'square'`)	[0, 1]	linear	no units

Property	Range	Mapping	Units
Width (available when you set SignalType to 'sawtooth')	[0, 1]	linear	no units

Examples

Generate Variable-Frequency Sine Wave

Use the audioOscillator System object™ to generate a variable-frequency sine wave.

Create an audio oscillator to generate a sine wave. Use the default settings.

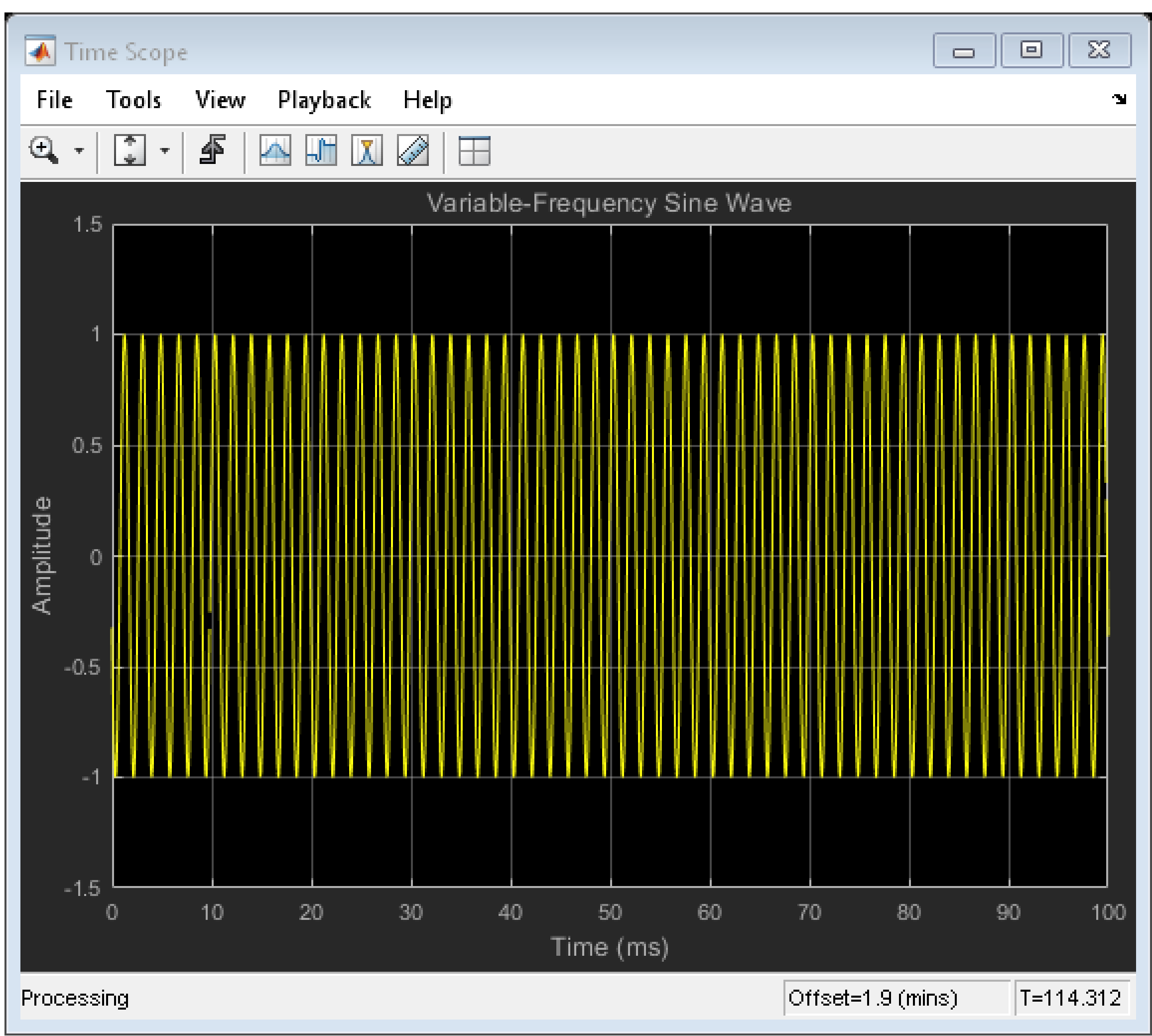
```
osc = audioOscillator;
```

Create a time scope to visualize the variable-frequency sine wave generated by the audio oscillator.

```
scope = dsp.TimeScope( ...
    'SampleRate',osc.SampleRate, ...
    'TimeSpan',0.1, ...
    'YLimits',[-1.5,1.5], ...
    'TimeSpanOverrunAction', 'Scroll', ...
    'ShowGrid',true, ...
    'Title','Variable-Frequency Sine Wave');
```

Place the audio oscillator in an audio stream loop. Increase the frequency of your sine wave in 50-Hz increments.

```
counter = 0;
while (counter < 1e4)
    counter = counter + 1;
    sineWave = osc();
    scope(sineWave);
    if mod(counter,1000)==0
        osc.Frequency = osc.Frequency + 50;
    end
end
```



Create a Melody by Tuning Oscillation Frequency

Tune the frequency of an audio oscillator at regularly spaced intervals to create a melody. Play the melody to your audio output device.

Create a structure to hold the frequency values of notes in a melody.

```
notes = struct('C4',261.63,'E4',329.63,'G4sharp',415.30,'A4',440,'B4',493.88, ...
    'C5',523.25,'D5',587.25,'D5sharp',622.25,'E5',659.25,'Silence',0);
```

Create `audioOscillator` and `audioDeviceWriter` System objects™. Use the default settings.

```
osc = audioOscillator;
aDW = audioDeviceWriter;
```

Create a vector with the initial melody of Fur Elise.

```
melody = [notes.Silence notes.Silence,...
    notes.E5 notes.D5sharp notes.E5 notes.D5sharp notes.E5 notes.B4 ...
        notes.D5 notes.C5 notes.A4 notes.A4 notes.Silence ...
    notes.C4 notes.E4 notes.A4 notes.B4 notes.B4 notes.Silence ...
    notes.E4 notes.G4sharp notes.B4 notes.C5 notes.C5 notes.Silence];
```

Specify the note duration in seconds. In an audio stream loop, call your audio oscillator and write the sound to your audio device. Update the frequency of the audio oscillator in `noteDuration` time steps to follow the melody. As a best practice, release your objects once complete.

```
noteDuration = 0.3;

i = 1;
tic
while i < numel(melody)
    tone = osc();
    aDW(tone);
    if toc >= noteDuration
        i = i + 1;
        osc.Frequency = melody(i);
        tic
    end
end

release(osc);
release(aDW);
```

Control Cutoff Frequency of Lowpass Filter

Create a low-frequency oscillator (LFO) lowpass filter, using the `audioOscillator` as a control signal.

Create `dsp.AudioFileReader` and `audioDeviceWriter` System objects to read from an audio file and write to your audio device. Create a biquad filter object to apply lowpass filtering to your audio signal.

```
fileReader = dsp.AudioFileReader('Filename','Engine-16-44p1-stereo-20sec.wav');
deviceWriter = audioDeviceWriter('SampleRate',fileReader.SampleRate);
lowpassFilter = dsp.BiquadFilter( ...
    'SOSMatrixSource','Input port', ...
    'ScaleValuesInputPort',false);
```

Create an audio oscillator object. Your audio oscillator controls the cutoff frequency of the lowpass filter in an audio stream loop.

```
osc = audioOscillator('SignalType','sawtooth', ...
    'DCOffset',0.05, ...
    'Amplitude',0.03, ...
    'SamplesPerFrame',fileReader.SamplesPerFrame, ...
    'SampleRate',fileReader.SampleRate, ...
    'Frequency',5);
```

In a loop, filter the audio signal through the lowpass filter. Write the output signal to your audio device.

```
while ~isDone(fileReader)
    audioIn    = fileReader();
    ctrlSignal = osc();
    [B,A]      = designVarSlopeFilter(48,ctrlSignal(end));
    audioOut   = lowpassFilter(audioIn,B,A);
    deviceWriter(audioOut);
end
```

As a best practice, release objects once complete.

```
release(osc)
release(fileReader)
release(deviceWriter)
```

For a more complete implementation of an LFO Filter, see audiopluginexample.LFOFilter in the “Audio Plugin Example Gallery”.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

“System Objects in MATLAB Code Generation” (MATLAB Coder)

See Also

System Objects
`wavetableSynthesizer`

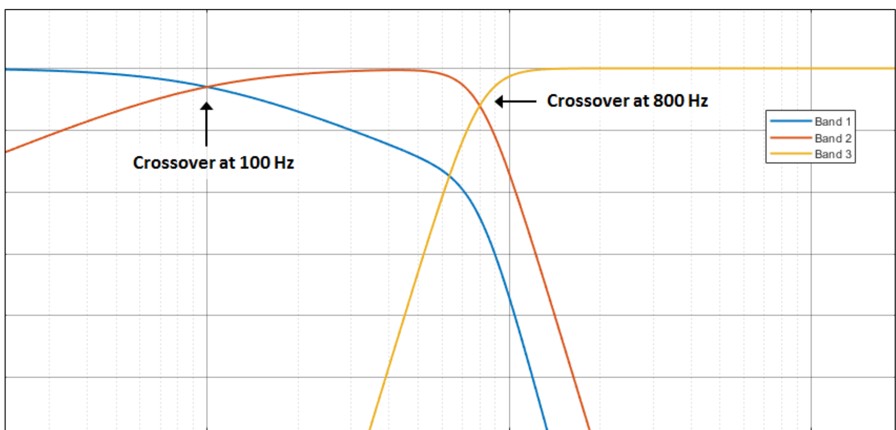
Introduced in R2016a

crossoverFilter System object

Audio crossover filter

Description

The `crossoverFilter` System object implements an audio crossover filter, which is used to split an audio signal into two or more frequency bands. Crossover filters are multiband filters whose overall magnitude frequency response is flat.



To implement an audio crossover filter:

1. Create the crossoverFilter object and set its properties.
2. Call the object with arguments, as if it were a function.

To learn more about how System objects work, see What Are System Objects? (MATLAB).

Creation

Syntax

```
crossFilt = crossoverFilter
crossFilt = crossoverFilter(nCrossovers)
crossFilt = crossoverFilter(nCrossovers,xFrequencies)
crossFilt = crossoverFilter(nCrossovers,xFrequencies,xSlopes)
crossFilt = crossoverFilter(nCrossovers,xFrequencies,xSlopes,Fs)
crossFilt = crossoverFilter( ___ ,Name,Value)
```

Description

`crossFilt = crossoverFilter` creates a System object, `crossFilt`, that implements an audio crossover filter.

`crossFilt = crossoverFilter(nCrossovers)` sets the NumCrossovers property to `nCrossovers`.

`crossFilt = crossoverFilter(nCrossovers,xFrequencies)` sets the CrossoverFrequencies property to `xFrequencies`.

`crossFilt = crossoverFilter(nCrossovers,xFrequencies,xSlopes)` sets the CrossoverSlopes property to `xSlopes`.

`crossFilt = crossoverFilter(nCrossovers,xFrequencies,xSlopes,Fs)` sets the SampleRate property to `Fs`.

`crossFilt = crossoverFilter( ___ ,Name,Value)` sets each property `Name` to the specified `Value`. Unspecified properties have default values.

Example: `crossFilt = crossoverFilter(2,'CrossoverFrequencies',[100,800],'CrossoverSlopes',[6,48])` creates a System object, `crossFilt`, with two crossovers located at 100 Hz and 800 Hz, and crossover slopes of 6 dB/octave and 48 dB/octave, respectively.

Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see System Design in MATLAB Using System Objects (MATLAB).

`NumCrossovers` — Number of magnitude response band crossings
1 (default) | 2 | 3 | 4

Number of magnitude response band crossings, specified as a scalar integer in the range 1 to 4.

The number of bands output when implementing crossover filtering is one more than the `NumCrossovers` value.

Number of magnitude response band crossings	Number of bands output
1	two-band
2	three-band
3	four-band
4	five-band

Tunable: No

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`CrossoverFrequencies` — Crossover frequencies (Hz)
`100` (default) | scalar | vector

Crossover frequencies in Hz, specified as a scalar or vector of real values of length `NumCrossovers`.

Crossover frequencies are the intersections of magnitude response bands of the individual two-band crossover filters used in the multiband crossover filter.

Tunable: Yes

Data Types: `single` | `double`

`CrossoverSlopes` — Crossover slopes (dB/octave)

`12` (default) | scalar | vector

Crossover slopes in dB/octave, specified as a scalar or vector of real values in the range [6:6:48]. If a specified crossover slope is not inside the range, the slope is rounded to the nearest allowed value.

- If `CrossoverSlopes` is a scalar, all two-band component crossover slopes take that value.
- If `CrossoverSlopes` is a vector of length NumCrossovers, the respective two-band component crossover slopes take those values.

Crossover slopes are the slopes of individual bands at the associated crossover frequency, as specified in the two-band component crossover.

Tunable: Yes

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`SampleRate` — Input sample rate (Hz)

`44100` (default) | positive scalar

Input sample rate in Hz, specified as a positive scalar.

Tunable: Yes

Data Types: `single` | `double`

Usage

Note For versions earlier than R2016b, use the `step` function to run the System object™ algorithm. The arguments to `step` are the object you created, followed by the arguments shown in this section.

For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Syntax

```
[band1,...,bandN] = crossFilt(audioIn)
```

Description

`[band1,...,bandN] = crossFilt(audioIn)` applies a crossover filter on the input, `audioIn`, and returns the filtered output bands, `[band1,...,bandN]`, where `N = NumCrossovers + 1`.

Input Arguments

`audioIn` — Audio input to crossover filter
matrix

Audio input to the crossover filter, specified as a matrix. The columns of the matrix are treated as independent audio channels.

Data Types: `single` | `double`

Output Arguments

`[band1,...,bandN]` — Audio bands output from crossover filter
set of matrices

Audio bands output from the crossover filter, returned as a set of `N` bands. The NumCrossovers property determines the number of return arguments: `N = NumCrossovers + 1`. The size of each output argument is the same size as `audioIn`.

Data Types: `single` | `double`

Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

Specific to crossoverFilter

visualize	Visualize magnitude response of crossover filter
cost	Estimate implementation cost of audio System objects
createAudioPluginClass	Create audio plugin class that implements functionality of System object

MIDI

configureMIDI	Configure MIDI connections between audio object and MIDI controller
disconnectMIDI	Disconnect MIDI controls from audio object
getMIDIConnections	Get MIDI connections of audio object

Common to All System Objects

clone	Create duplicate System object
isLocked	Determine if System object is in use
release	Release resources and allow changes to System object property values and input characteristics
reset	Reset internal states of System object
step	Run System object algorithm

The `createAudioPluginClass` and `configureMIDI` functions map tunable properties of the `crossoverFilter` System object to user-facing parameters:

Property	Range	Mapping	Unit
`CrossoverFrequencies`	[20, 20000]	linear	Hz
`CrossoverSlopes`	[6, 48]	linear	dB/octave

Examples

Pass Noise Signal Through Crossover Filter

Use the `crossoverFilter` System object™ to split Gaussian noise into three separate frequency bands.

Create a 5 second noise signal that assumes a 12,000 Hz sample rate.

```
noise = randn(12000*5,1);
```

Create a `crossoverFilter` System object with 2 crossovers (3 bands), crossover frequencies at 4 kHz and 8 kHz, a slope of 48 dB/octave, and a sample rate of 24 kHz.

```
crossFilt = crossoverFilter(...
    'NumCrossovers',2,...
    'CrossoverFrequencies',[4000,8000],...
    'CrossoverSlopes',48,...
    'SampleRate',24000);
```

Visualize the magnitude response of your crossover filter object.

```
visualize(crossFilt)
```

Call your crossover filter like a function with the noise signal as the argument.

```
[y1,y2,y3] = crossFilt(noise);
```

Visualize the results using a spectrogram.

```
figure('Position',[100,100,800,700]);
subplot(4,1,1)
    spectrogram(noise,120,100,6000,24000,'yaxis')
        title('Noise')
subplot(4,1,2)
    spectrogram(y1,120,100,6000,24000,'yaxis')
        title('y1')
subplot(4,1,3)
    spectrogram(y2,120,100,6000,24000,'yaxis')
        title('y2')
subplot(4,1,4)
    spectrogram(y3,120,100,6000,24000,'yaxis')
        title('y3')
```

Split Audio Signal into Three Bands

Use the `crossoverFilter` System object™ to split an audio signal into three frequency bands.

Create the `dsp.AudioFileReader` and `audioDeviceWriter` System objects. Use the sample rate of the reader as the sample rate of the writer. Call `setup` to reduce the computation load of initialization in an audio stream loop.

```
samplesPerFrame = 256;

fileReader = dsp.AudioFileReader( ...
    'RockGuitar-16-44p1-stereo-72secs.wav', ...
    'SamplesPerFrame',samplesPerFrame);
deviceWriter = audioDeviceWriter( ...
    'SampleRate',fileReader.SampleRate);

setup(fileReader)
setup(deviceWriter,ones(samplesPerFrame,2))
```

Create a `crossoverFilter` System object with 2 crossovers (3 bands), crossover frequencies at 500 Hz and 1 kHz, and a slope of 18 dB/octave. Use the sample rate of the reader as the sample rate of the crossover filter.

```
crossFilt = crossoverFilter( ...
    'NumCrossovers',2, ...
    'CrossoverFrequencies',[500,1000], ...
    'CrossoverSlopes',18, ...
    'SampleRate',fileReader.SampleRate);

setup(crossFilt,ones(samplesPerFrame,2))
```

Visualize the bands of the crossover filter.

```
visualize(crossFilt)
```

Get the cost of the crossover filter.

```
cost(crossFilt)
```

Create a spectrum analyzer to visualize the effect of the crossover filter.

```
scope = dsp.SpectrumAnalyzer( ...
    'SampleRate',fileReader.SampleRate, ...
    'PlotAsTwoSidedSpectrum',false, ...
    'FrequencyScale','Log', ...
    'FrequencyResolutionMethod','WindowLength', ...
    'WindowLength',samplesPerFrame, ...
    'Title', ...
    'Crossover Bands and Reconstructed Signal', ...
```

```
    'ShowLegend',true, ...
    'ChannelNames',{'Original Signal','Band 1', ...
    'Band 2','Band 3','Sum'});
```

Play 10 seconds of the audio signal. Visualize the spectrum of the original audio, the crossover bands, and the reconstructed signal (sum of bands).

```
setup(scope,ones(samplesPerFrame,5))
count = 0;
while count < (fileReader.SampleRate/samplesPerFrame)*10
    originalSignal = fileReader();
    [band1,band2,band3] = crossFilt(originalSignal);
    sumOfBands = band1 + band2 + band3;
    scope([originalSignal(:,1), ...
           band1(:,1), ...
           band2(:,1), ...
           band3(:,1), ...
           sumOfBands(:,1)])
    deviceWriter(sumOfBands);
    count = count+1;
end

release(fileReader)
release(crossFilt)
release(scope)
release(deviceWriter)
```

Apply Split-Band De-Essing

De-essing is the process of diminishing sibilant sounds in an audio signal. Sibilance refers to the *s*, *z*, and *sh* sounds in speech, which can be disproportionately emphasized during recording. *es* sounds fall under the category of unvoiced speech with all consonants and have a higher frequency than voiced speech. In this example, you apply split-band de-essing to a speech signal by separating the signal into high and low frequencies, applying an expander to diminish the sibilant frequencies, and then remixing the channels.

Create a `dsp.AudioFileReader` System object™ and an `audioDeviceWriter` System object to read from a sound file and write to an audio device. Listen to the unprocessed signal. Then release the file reader and device writer.

```
fileReader = dsp.AudioFileReader( ...
    fullfile(matlabroot,'examples','audio','Sibilance.wav'));
```

```
deviceWriter = audioDeviceWriter;

while ~isDone(fileReader)
    audioIn = fileReader();
    deviceWriter(audioIn);
end

release(deviceWriter)
release(fileReader)
```

Create an `expander` System object to de-ess the audio signal. Set the sample rate of the expander to the sample rate of the audio file. Create a two-band crossover filter with a crossover of 3000 Hz. Sibilance is usually found in this range. Set the crossover slope to 12. Plot the frequency response of the crossover filter to confirm your design visually.

```
dRExpander = expander( ...
    'Threshold',-50, ...
    'AttackTime', 0.05, ...
    'ReleaseTime',0.05, ...
    'HoldTime',0.005, ...
    'SampleRate',fileReader.SampleRate);

crossFilt = crossoverFilter( ...
    'NumCrossovers',1, ...
    'CrossoverFrequencies',3000, ...
    'CrossoverSlopes',12);
visualize(crossFilt)
```

Create a `dsp.TimeScope` System object to visualize the original and processed audio signals.

```
scope = dsp.TimeScope( ...
    'SampleRate',fileReader.SampleRate, ...
    'TimeSpanOverrunAction','Scroll', ...
    'TimeSpan',4, ...
    'BufferLength',44100*8, ...
    'YLimits',[-1 1], ...
    'ShowGrid',true, ...
    'ShowLegend',true, ...
    'ChannelNames',{'Original','Processed'});
```

In an audio stream loop:

1 Read in a frame of the audio file.

2. Split the audio signal into two bands.
3. Apply dynamic range expansion to the upper band.
4. Remix the channels.
5. Write the processed audio signal to your audio device for listening.
6. Visualize the processed and unprocessed signals on a time scope.

As a best practice, release your objects once done.

```
while ~isDone(fileReader)
    audioIn = fileReader();

    [band1,band2] = crossFilt(audioIn);

    band2processed = dRExpander(band2);

    procAudio  = band1 + band2processed;

    deviceWriter(procAudio);

    scope([audioIn procAudio]);
end

release(deviceWriter)
release(fileReader)
release(scope)
release(crossFilt)
release(dRExpander)
```

Diminish Plosives from Speech Signal

Plosives are consonant sounds resulting from a sudden release of airflow. They are most pronounced in words beginning with *p*, *d*, and *g* sounds. Plosives can be emphasized by the recording process and are often displeasurable to hear. In this example, you minimize the plosives of a speech signal by applying highpass filtering and low-band compression.

Create a `dsp.AudioFileReader` System object™ and a `audioDeviceWriter` System object™ to read an audio signal from a file and write an audio signal to a device. Play the unprocessed signal. Then release the file reader and device writer.

```
fileReader   = dsp.AudioFileReader( ...
    fullfile(matlabroot,'examples','audio','Plosives.wav'));
deviceWriter = audioDeviceWriter;

while ~isDone(fileReader)
    audioIn = fileReader();
    deviceWriter(audioIn);
end
release(deviceWriter)
release(fileReader)
```

Design a highpass filter with a steep rolloff of all frequencies below 120 Hz. Use a `dsp.BiquadFilter` System object to implement the highpass filter design. Create a crossover filter with one crossover at 250 Hz. The crossover filter enables you to separate the band of interest for processing. Create a dynamic range compressor to compress the dynamic range of plosive sounds. To apply no make-up gain, set the `MakeUpGainMode` to `'Property'` and use the default 0 dB `MakeUpGain` property value. Create a time scope to visualize the processed and unprocessed audio signal.

```
[B,A] = designVarSlopeFilter(48,120/(44100/2),'hi');
biquadFilter = dsp.BiquadFilter( ...
    'SOSMatrixSource','Input port', ...
    'ScaleValuesInputPort',false);

crossFilt = crossoverFilter( ...
    'NumCrossovers',1, ...
    'CrossoverFrequencies',250, ...
    'CrossoverSlopes',48);

dRCompressor = compressor( ...
    'Threshold',-35, ...
    'Ratio',10, ...
    'KneeWidth',20, ...
    'AttackTime',1e-4, ...
    'ReleaseTime',3e-1, ...
    'MakeUpGainMode','Property', ...
    'SampleRate',fileReader.SampleRate);

scope = dsp.TimeScope( ...
    'SampleRate',fileReader.SampleRate, ...
    'TimeSpan',3, ...
    'BufferLength',fileReader.SampleRate*3*2, ...
    'YLimits',[-1 1], ...
    'ShowGrid',true, ...
```

```
    'ShowLegend',true, ...
    'ChannelNames',{'Original','Processed'});
```

In an audio stream loop:

1. Read in a frame of the audio file.
2. Apply highpass filtering using your biquad filter.
3. Split the audio signal into two bands.
4. Apply dynamic range compression to the lower band.
5. Remix the channels.
6. Write the processed audio signal to your audio device for listening.
7. Visualize the processed and unprocessed signals on a time scope.

As a best practice, release your objects once done.

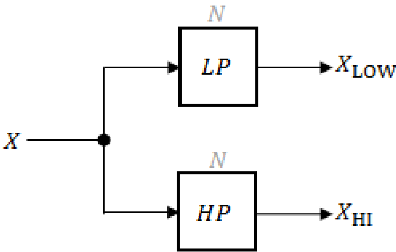
```
while ~isDone(fileReader)
    audioIn = fileReader();

    audioIn = biquadFilter(audioIn,B,A);

    [band1,band2] = crossFilt(audioIn);

    band1compressed = dRCompressor(band1);

    audioOut  = band1compressed + band2;

    deviceWriter(audioOut);

    scope([audioIn audioOut]);
end

release(deviceWriter)
release(fileReader)
release(scope)
release(crossFilt)
release(dRCompressor)
```

Algorithms

The crossover System object is implemented as a binary tree of crossover pairs with additional phase-compensating sections [1]. Odd-order crossovers are implemented with Butterworth filters, while even-order crossovers are implemented with cascaded Butterworth filters (Linkwitz-Riley filters).

Odd-Order Crossover Pair

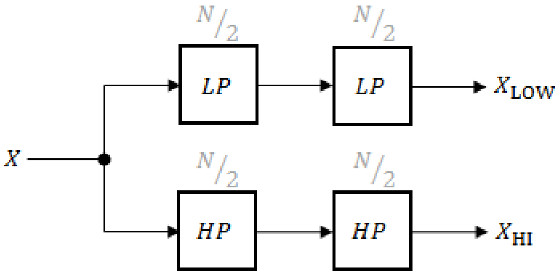
Odd-order two-band (one crossover) filters are implemented as parallel complementary highpass and lowpass filters.



LP and *HP* are Butterworth filters of order *N*, implemented as direct-form II transposed second-order sections. The shared cutoff frequency used in their design corresponds to the crossover of the resulting bands.

Even-Order Crossover Pair

Even-order two-band (one crossover) filters are implemented as parallel complementary highpass and lowpass filters.

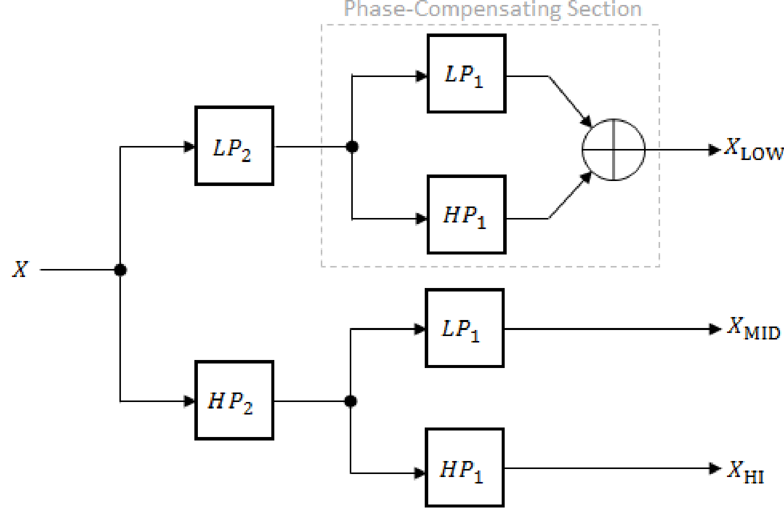


LP and *HP* are Butterworth filters of order *N*/2, where *N* is the order of the overall filter. The filters are implemented as direct-form II transposed second-order sections.

For overall filters of orders 2 and 6, X_{HI} is multiplied by –1 internally so that the branches of your crossover pair are in-phase.

Even-Order Three-Band Filter

Even-order three-band (two crossovers) filters are implemented as parallel complementary highpass and lowpass filters organized in a tree structure.



The phase-compensating section is equivalent to an allpass filter.

The design of four-band and five-band filters (three and four crossovers) are extensions of the pattern developed for even-order and odd-order crossovers and the tree structure specified for three-band (two crossover) filters.

References

[1] D'Appolito, Joseph A. "Active Realization of Multiway All-Pass Crossover Systems." *Journal of Audio Engineering Society*. Vol. 35, Issue 4, 1987, pp. 239–245.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

"System Objects in MATLAB Code Generation" (MATLAB Coder)

See Also

Blocks
Crossover Filter

System Objects
`multibandParametricEQ`

Introduced in R2016a

visualize

Visualize magnitude response of crossover filter

Syntax

```
visualize(crossFilt)
visualize(crossFilter,NFFT)
```

Description

`visualize(crossFilt)` plots the magnitude response of the `crossoverFilter`. The plot is updated automatically when properties of the object change.

`visualize(crossFilter,NFFT)` specifies an N-point FFT used to calculate the magnitude response.

Examples

Visualize Magnitude Response of Crossover Filter

Create an object of the `crossoverFilter` System object™, and then call `visualize` to plot the magnitude response of the filter.

```
crossFilt = crossoverFilter;
visualize(crossFilt)
```

Modify the crossover frequency and observe that the plot is updated automatically.

```
crossFilt.CrossoverFrequencies = 500;
```

Input Arguments

`crossFilt` — Crossover filter to visualize
object of `crossoverFilter` System object

Crossover filter whose magnitude response you want to plot.

`NFFT` — N-point FFT
2048 (default) | positive scalar

Number of bins used to calculate the DFT, specified as a positive scalar.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

See Also

System Objects
`crossoverFilter`

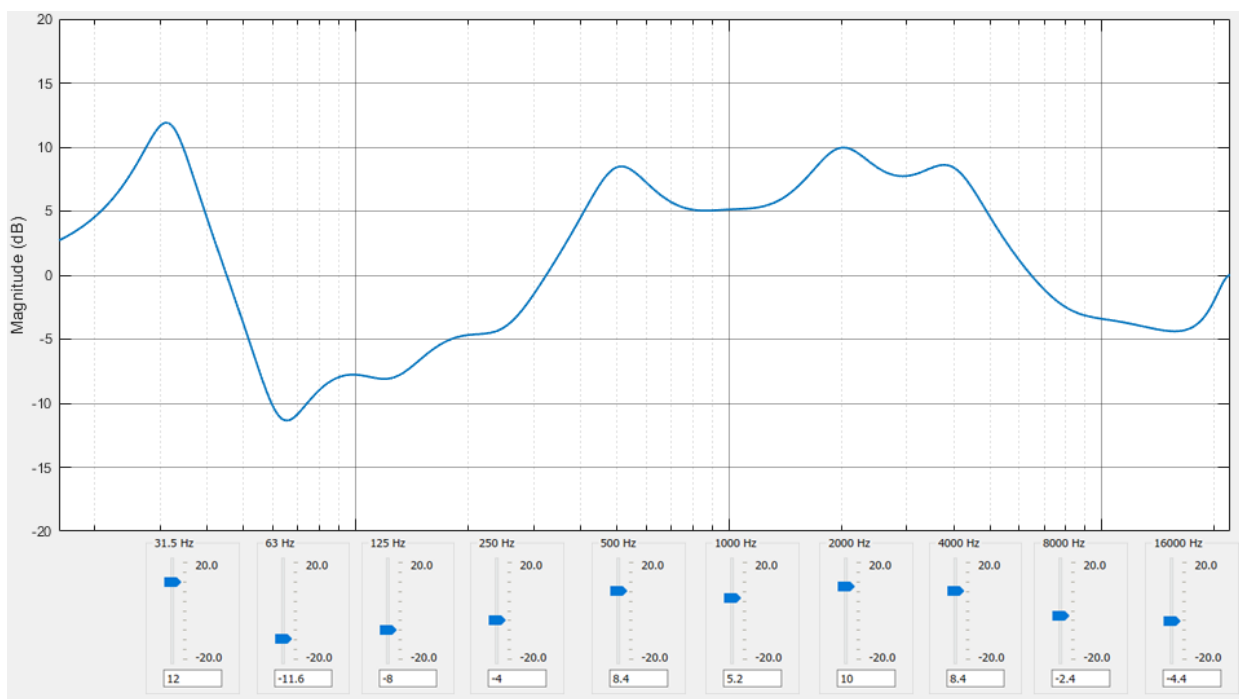
Introduced in R2016a

graphicEQ System object

Standards-based graphic equalizer

Description

The `graphicEQ` System object implements a graphic equalizer that can tune the gain on individual octave or fractional octave bands. The object filters the data independently across each input channel over time using the filter specifications. Center and edge frequencies of the bands are based on the ANSI S1.11-2004 standard.



To equalize an audio signal:

1. Create the graphicEQ object and set its properties.

2 Call the object with arguments, as if it were a function.

To learn more about how System objects work, see What Are System Objects? (MATLAB).

Creation

Syntax

```
equalizer = graphicEQ
equalizer = graphicEQ(Name,Value)
```

Description

`equalizer = graphicEQ` creates a graphic equalizer with default values.

`equalizer = graphicEQ(Name,Value)` sets each property `Name` to the specified `Value`. Unspecified properties have default values.

Example: `equalizer = graphicEQ('Structure','Parallel','EQOrder','1/3 octave')` creates a System object, `equalizer`, which implements filtering using a parallel structure and one-third octave filter bandwidth.

Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see System Design in MATLAB Using System Objects (MATLAB).

`Gains` — Gain of each octave or fractional octave band (dB)
`[0 0 0 0 0 0 0 0 0 0]` (default) | 10-, 15-, or 30-element row vector

Gain of each octave of fractional octave band in dB, specified as a row vector with a length determined by the Bandwidth property:

- '1 octave' –– Specify gains as a 10-element row vector.
- '2/3 octave' –– Specify gains as a 15-element row vector.
- '1/3 octave' –– Specify gains as a 30-element row vector.

Example: `equalizer = graphicEQ('Bandwidth','2/3 octave','Gains',[5,5,5,5,5,0,0,0,0,0,-5,-5,-5,-5,-5])` creates a two-third octave graphic equalizer with specified gains.

You can tune the gains of your graphic equalizer when the object is locked. However, you cannot tune the length of the gains when the object is locked.

Tunable: Yes

Data Types: `single` | `double`

`EQOrder` — Order of individual equalizer bands

2 (default) | positive even integer

Order of individual equalizer bands, specified as a positive even integer. All equalizer bands have the same order.

Tunable: No

Data Types: `single` | `double`

`Bandwidth` — Filter bandwidth (octaves)

`'1 octave'` (default) | `'2/3 octave'` | `'1/3 octave'`

Filter bandwidth in octaves, specified as `'1 octave'`, `'2/3 octave'`, or `'1/3 octave'`.

The ANSI S1.11-2004 standard defines the center and edge frequencies of your equalizer. The ISO 266:1997(E) standard specifies corresponding preferred frequencies for labeling purposes.

1-Octave Bandwidth

Center frequencies	32 63 126 251 501 1000 1995 3981 7943 15849
Edge frequencies	22 45 89 178 355 708 1413 2818 5623 1122 22387

Preferred frequencies	31.5 63 125 250 500 1000 2000 4000 8000 16000

2/3-Octave Bandwidth

Center frequencies	25 40 63 100 158 251 398 631 1000 1585 2512 3981 6310 10000 15849
Edge frequencies	20 32 50 79 126 200 316 501 794 1259 1995 3162 5012 7943 12589 19953
Preferred frequencies	25 40 63 100 160 250 400 630 1000 1600 2500 4000 6300 10000 16000

1/3-Octave Bandwidth

Center frequencies	25 32 40 50 63 79 100 126 158 200 251 316 398 501 631 794 1000 1259 1585 1995 2512 3162 3981 5012 6310 7943 10000 12589 15849 19953
Edge frequencies	22 28 35 45 56 71 89 112 141 178 224 282 355 447 562 708 891 1122 1413 1778 2239 2818 3548 4467 5623 7079 8913 11220 14125 17783 22387
Preferred frequencies	25 31.5 40 50 63 80 100 125 160 200 250 315 400 500 630 800 1000 1250 1600 2000 2500 3150 4000 5000 6300 8000 10000 12500 16000 20000

Tunable: No

Data Types: `char` | `string`

`Structure` — Type of implementation

`'Cascade'` (default) | `'Parallel'`

Type of implementation, specified as `'Cascade'` or `'Parallel'`. See “Algorithms” on page 3-153 and “Graphic Equalization” for information about these implementation structures.

Tunable: No

Data Types: `char` | `string`

`SampleRate` — Input sample rate (Hz)
`44100` (default) | positive scalar

Input sample rate in Hz, specified as a positive scalar.

Tunable: Yes

Data Types: `single` | `double`

Usage

Syntax

```
audioOut = equalizer(audioIn)
```

Description

`audioOut = equalizer(audioIn)` performs graphic equalization on the input signal, `audioIn`, and returns the equalized signal, `audioOut`. The type of equalization is specified by the algorithm and properties of the `graphicEQ` System object, `equalizer`.

Input Arguments

`audioIn` — Audio input to graphic equalizer
matrix

Audio input to the graphic equalizer, specified as a matrix. The columns of the matrix are treated as independent audio channels.

Data Types: `single` | `double`

Output Arguments

`audioOut` — Audio output from graphic equalizer
matrix

Audio output from the graphic equalizer, returned as a matrix the same size as `audioIn`.

Data Types: `single` | `double`

Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

Specific to graphicEQ

createAudioPluginClass	Create audio plugin class that implements functionality of System object
coeffs	Get graphic equalizer coefficients
info	Get standards-based frequencies for graphic equalizer
visualize	Visualize magnitude response of graphic equalizer

MIDI

configureMIDI	Configure MIDI connections between audio object and MIDI controller
disconnectMIDI	Disconnect MIDI controls from audio object
getMIDIConnections	Get MIDI connections of audio object

Common to All System Objects

clone	Create duplicate System object
isLocked	Determine if System object is in use
release	Release resources and allow changes to System object property values and input characteristics
reset	Reset internal states of System object
step	Run System object algorithm

The `createAudioPluginClass` and `configureMIDI` functions map tunable properties of the `graphicEQ` System object to user-facing parameters:

Property	Range	Mapping	Unit
`Gains`	[–20, 20]	linear	dB

Examples

Perform Graphic Equalization

Design and create an object for graphic equalization and then perform equalization on an audio signal.

Create objects to read from an audio file and write to your audio device. Use the sample rate of the reader as the sample rate of the writer.

```
frameLength = 512;
reader = dsp.AudioFileReader( ...
    'Filename','RockDrums-48-stereo-11secs.mp3', ...
    'SamplesPerFrame',frameLength);
player = audioDeviceWriter( ...
    'SampleRate',reader.SampleRate);
```

In an audio stream loop, read audio from a file and play the audio through your audio device.

```
while ~isDone(reader)
    x = reader();
    player(x);
end
release(reader)
release(player)
```

Create a one-octave graphic equalizer implemented with a cascade structure. Use the sample rate of the reader as the sample rate of the equalizer.

```
equalizer = graphicEQ( ...
    'Bandwidth','1 octave', ...
    'Structure','Cascade', ...
    'SampleRate',reader.SampleRate);
```

Specify to increase the gain on low frequencies and then visualize the equalizer.

```
equalizer.Gains = [5 5 5 5 0 0 0 0 0 0];
visualize(equalizer)
```

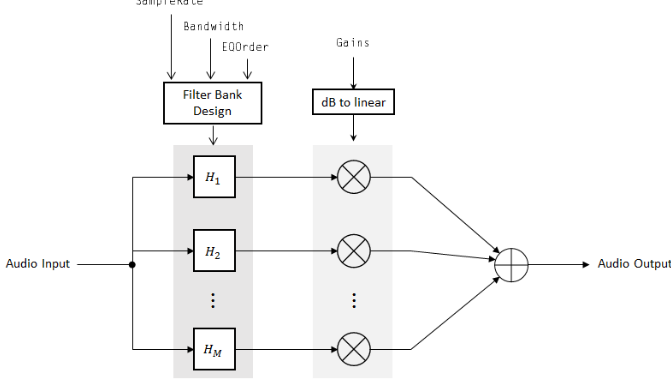
In an audio stream loop, read audio from a file, apply equalization, and then play the equalized audio through your audio device.

```
while ~isDone(reader)
    x = reader();
    y = equalizer(x);
    player(y);
end
release(reader)
release(player)
```

Algorithms

The implementation of your graphic equalizer depends on the `Structure` property. See "Graphic Equalization" for a discussion of the pros and cons of the parallel and cascade implementations. Refer to the following sections to understand how these algorithms are implemented in Audio System Toolbox.

Parallel Structure



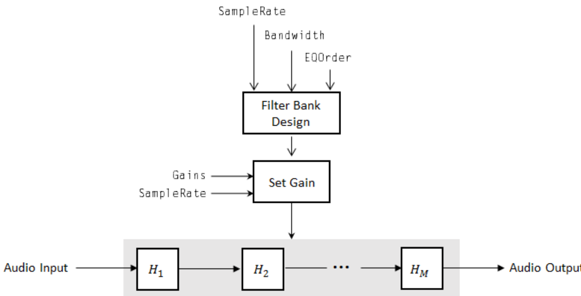
The parallel implementation designs the individual equalizers using the `octaveFilter design method` and spaces them on the spectrum according to the ANSI S1.11-2004 standard.

If you set the SampleRate property so that the Nyquist frequency (`SampleRate/2`) is less than the final bandpass edge defined by the ANSI S1.11-2004 standard, then:

- The final bandpass filter is the one whose upper bandpass edge is less than the Nyquist frequency.
- The final filter is implemented as a highpass filter designed by the `designParamEQ` function.

1. The input signal is fed into a filterbank of *M* filters, where *M* depends on the specified `Bandwidth` and `SampleRate` properties.
2. Each branch of the filterbank is multiplied by the linear form of the corresponding element of the `Gains` property.
3. The branches are summed and the output signal is returned.

Cascade Structure



The cascade implementation designs the graphic equalizer filter bank using the `multibandParametricEQ` System object.

If the EQOrder property is set to 2, then a gain correction is calculated according to [1]. The gain correction is independent of the requested gains. The gain correction is recomputed during the real-time processing only if the `SampleRate` property is modified.

If the `EQOrder` property is not set to 2, no gain correction is applied, and the requested gains are passed on to the `multibandParametricEQ` object.

The input signal is fed into a cascade of *M* biquad filters, where *M* depends on the specified `Bandwidth` and `SampleRate` properties.

References

[1] Oliver, Richard J., and Jean-Marc Jot. "Efficient Multi-Band Digital Audio Graphic Equalizer with Accurate Frequency Response Control." Presented at the 139th Convention of the AES, New York, October 2015.

[2] Acoustical Society of America. *American National Standard Specification for Octave-Band and Fractional-Octave-Band Analog and Digital Filters*. ANSI S1.11-2004. Melville, NY: Acoustical Society of America, 2009.

[3] International Organization for Standardization. *Acoustics –– Preferred frequencies*. ISO 266:1997(E). Second Edition. 1997.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

“System Objects in MATLAB Code Generation” (MATLAB Coder)

See Also

System Objects
`multibandParametricEQ`

Blocks
Graphic EQ | Parametric EQ Filter

Functions
`designParamEQ` | `designShelvingEQ` | `designVarSlopeFilter`

Topics

“Graphic Equalization”
“Equalization”

Introduced in R2017b

coeffs

Get graphic equalizer coefficients

Syntax

```
[B,A] = coeffs(equalizer)
```

Description

`[B,A] = coeffs(equalizer)` returns the coefficients of the filters used in the graphic equalizer System object, `equalizer`. If `Structure` is set to `'Cascade'`, the coefficients are returned as second-order sections. If `Structure` is set to `'Parallel'`, the coefficients are returned as fourth-order sections.

Examples

Get Coefficients for Cascade Structure

Create a `graphicEQ` System object™ using the default `Cascade` structure and 1-octave bandwidth. A 1-octave bandwidth graphic equalizer consists of 10 bandpass filters. Specify the gains of each filter using the `Gains` property.

```
equalizer = graphicEQ('Gains',randn(1,10))
```

```
equalizer =
  graphicEQ with properties:

       EQOrder: 2
     Bandwidth: '1 octave'
     Structure: 'Cascade'
         Gains: [1x10 double]
    SampleRate: 44100
```

Call `coeffs` to return the coefficients of the filter designed by the graphic equalizer. Because the structure of the object is `Cascade`, the coefficients are returned as 2-D matrix. Columns correspond to the cascaded stages of the filter.

```
[B,A] = coeffs(equalizer)
```

```
B = 3×10

    1.0000    1.0009    0.9978    1.0019    1.0012    0.9923    0.9968    0.9897    1.0
   -1.9968   -1.9944   -1.9848   -1.9757   -1.9471   -1.8775   -1.7416   -1.3954   -0.6
    0.9968    0.9936    0.9873    0.9750    0.9509    0.9044    0.8176    0.6648    0.3


A = 2×10

   -1.9968   -1.9944   -1.9848   -1.9757   -1.9471   -1.8775   -1.7416   -1.3954   -0.6
    0.9969    0.9945    0.9851    0.9769    0.9521    0.8967    0.8145    0.6545    0.4
```

Get Coefficients for Parallel Structure

Create a `graphicEQ` System object™ with a `Parallel` structure and 1-octave bandwidth. A 1-octave bandwidth graphic equalizer consists of 10 bandpass filters. Specify the gains of each filter using the `Gains` property.

```
equalizer = graphicEQ('Structure','Parallel','Gains',randn(1,10))
```

```
equalizer = 
  graphicEQ with properties:

       EQOrder: 2
     Bandwidth: '1 octave'
     Structure: 'Parallel'
         Gains: [1x10 double]
    SampleRate: 44100
```

Call `coeffs` to return the coefficients of the filter designed by the graphic equalizer. Because the structure of the object is `Parallel`, the coefficients are returned as 3-D matrix. Pages correspond to the parallel branches of the filter.

```
[B,A] = coeffs(equalizer)
```

```
B =
B(:,:,1) =

    0.0017
         0
   -0.0017
         0
         0


B(:,:,2) =

    0.0039
         0
   -0.0039
         0
         0


B(:,:,3) =

    0.0048
         0
   -0.0048
         0
         0


B(:,:,4) =

    0.0138
         0
   -0.0138
         0
         0


B(:,:,5) =

    0.0255
         0
   -0.0255
         0
         0
```

```
B(:,:,6) =

    0.0411
         0
   -0.0411
         0
         0


B(:,:,7) =

    0.0869
         0
   -0.0869
         0
         0


B(:,:,8) =

    0.1752
         0
   -0.1752
         0
         0


B(:,:,9) =

    0.4475
         0
   -0.4475
         0
         0


B(:,:,10) =

    0.6782
   -0.6782
         0
         0
```

```
         0


A =
A(:,:,1) =

    1.0000
   -1.9968
    0.9968
         0
         0


A(:,:,2) =

    1.0000
   -1.9936
    0.9937
         0
         0


A(:,:,3) =

    1.0000
   -1.9871
    0.9874
         0
         0


A(:,:,4) =

    1.0000
   -1.9738
    0.9751
         0
         0


A(:,:,5) =

    1.0000
   -1.9459
    0.9509
```

```
          0
          0


A(:,:,6) =

     1.0000
    -1.8850
     0.9043
          0
          0


A(:,:,7) =

     1.0000
    -1.7442
     0.8174
          0
          0


A(:,:,8) =

     1.0000
    -1.3992
     0.6632
          0
          0


A(:,:,9) =

     1.0000
    -0.5534
     0.4072
          0
          0


A(:,:,10) =

     1.0000
     0.0139
```

```
0
0
0
```

Input Arguments

`equalizer` — Object of `graphicEQ`
object

Object of the `graphicEQ` System object.

Output Arguments

B — Numerator filter coefficients
matrix | array

Numerator filter coefficients, returned as a 2-D matrix or 3-D array, depending on the implementation structure.

- `'Cascade'` –– 3-by-(number of bandpass equalizers × `EQOrder/2`) matrix
- `'Parallel'` –– 5-by-(`ceil(EQOrder/4)`)-by-(number of bandpass equalizers) matrix

The number of bandpass equalizers depends on the Bandwidth property of your equalizer.

- 1-octave –– 10 filters
- 2/3-octave –– 15 filters
- 1/3-octave –– 30 filters

Data Types: `single` | `double`

A — Denominator filter coefficients
matrix | array

Denominator filter coefficients, returned as a 2-D matrix or 3-D array, depending on the implementation structure.

- `'Cascade'` –– 2-by-(number of bandpass equalizers × `EQOrder/2`) matrix. The leading unity coefficient is not returned.

- 'Parallel'-– 5-by-(ceil(EQOrder/4))-by-(number of bandpass equalizers) matrix

The number of bandpass equalizers depends on the Bandwidth property of your equalizer.

- 1-octave -– 10 filters
- 2/3-octave -– 15 filters
- 1/3-octave -– 30 filters

Data Types: single | double

See Also

System Objects
graphicEQ

Topics
"Equalization"

Introduced in R2017b

info

Get standards-based frequencies for graphic equalizer

Syntax

```
eqInfo = info(equalizer)
```

Description

`eqInfo = info(equalizer)` returns a structure, `eqInfo`, containing center, edge, and preferred frequencies based on the ANSI S1.11-2004 and ISO 266:1997(E) standards.

Examples

Get Graphic Equalizer Standards-Based Frequencies

Create an object of the `graphicEQ` System object™ and then call `info` to return a structure containing standards-based center, edge, and preferred frequencies.

```
equalizer = graphicEQ;
info(equalizer)
```

```
ans = struct with fields:
       CenterFrequencies: [1x10 double]
         EdgeFrequencies: [1x11 double]
    PreferredFrequencies: [31.5000 63 125 250 500 1000 2000 4000 8000 16000]
```

Input Arguments

`equalizer` — System object to get information from
System object

System object to get information from.

Output Arguments

`eqInfo` — Struct containing object information
struct

Struct containing information about the System object, `equalizer`.

See Also

System Objects
`graphicEQ`

Introduced in R2017b

visualize

Visualize magnitude response of graphic equalizer

Syntax

```
visualize(equalizer)
visualize(equalizer,NFFT)
```

Description

`visualize(equalizer)` plots the magnitude response of the `graphicEQ` object, `equalizer`. The plot is updated automatically when properties of the object change.

`visualize(equalizer,NFFT)` specifies an N-point FFT used to calculate the magnitude response.

Examples

Visualize Magnitude Response of Graphic Equalizer

Create a default object of the graphicEQ System object™ and then call `visualize`.

```
equalizer = graphicEQ;
visualize(equalizer)
```

Set the gains of the graphic equalizer to new values. The visualization of the magnitude response updates automatically.

```
equalizer.Gains = [-1,1,2,3,3,2,-10,5,5,-10];
```

Input Arguments

equalizer — Graphic equalizer to visualize
object of graphicEQ System object

Graphic equalizer whose magnitude response you want to plot.

NFFT — N-point FFT
2048 (default) | positive scalar

Number of bins used to calculate the DFT, specified as a positive scalar.

Data Types: single | double

See Also

System Objects
graphicEQ

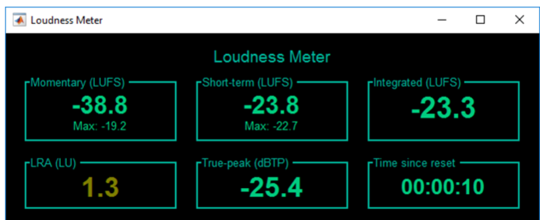
Introduced in R2017b

loudnessMeter System object

Standard-compliant loudness measurements

Description

The `loudnessMeter` System object computes the loudness, loudness range, and true-peak of an audio signal in accordance with EBU R 128 and ITU-R BS.1770-4 standards.



To implement loudness metering:

1. Create the loudnessMeter object and set its properties.
2. Call the object with arguments, as if it were a function.

To learn more about how System objects work, see What Are System Objects? (MATLAB).

Creation

Syntax

```
loudMtr = loudnessMeter
loudMtr = loudnessMeter(Name,Value)
```

Description

`loudMtr = loudnessMeter` creates a System object, `loudMtr`, that performs loudness metering independently across each input channel.

`loudMtr = loudnessMeter(Name,Value)` sets each property `Name` to the specified `Value`. Unspecified properties have default values.

Example: `loudMtr = loudnessMeter('ChannelWeights',[1.2, 0.8],'SampleRate',12000)` creates a System object, `loudMtr`, with channel weights of 1.2 and 0.8, and a sample rate of 12 kHz.

Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see System Design in MATLAB Using System Objects (MATLAB).

`ChannelWeights` — Linear weighting applied to each input channel

`[1, 1, 1, 1.41, 1.41]` (default) | nonnegative row vector

Linear weighting applied to each input channel, specified as a row vector of nonnegative values. The number of elements in the row vector must be equal to or greater than the number of input channels. Excess values in the vector are ignored.

The default channel weights follow the ITU-R BS.1170-4 standard. To use the default channel weights, specify the input signal channels as a matrix in this order: [Left, Right, Center, Left surround, Right surround].

As a best practice, specify the `ChannelWeights` property in order: [Left, Right, Center, Left surround, Right surround].

Tunable: Yes

Data Types: `single` | `double`

`UseRelativeScale` — Use relative scale for loudness measurements

`false` (default) | `true`

Use relative scale for loudness measurements, specified as a logical scalar.

- `false` –– The loudness measurements are absolute and returned in loudness units full scale (LUFS).
- `true` –– The loudness measurements are relative to the `TargetLoudness` value and returned in loudness units (LU).

Tunable: No

Data Types: `logical`

`TargetLoudness` — Target loudness level for relative scale (LUFS)

`-23` (default) | real scalar

Target loudness level for relative scale in LUFS, specified as a real scalar.

For example, if the `TargetLoudness` is –23 LUFS, then a loudness value of –23 LUFS is reported as 0 LU.

Tunable: Yes

Dependencies

To enable this property, set `UseRelativeScale` to `true`.

Data Types: `single` | `double`

`SampleRate` — Input sample rate (Hz)

`44100` (default) | positive scalar

Input sample rate in Hz, specified as a positive scalar.

Tunable: Yes

Data Types: `single` | `double`

Usage

Syntax

```
[momentary,shortTerm,
,range,peak] = loudMtr(audioIn)
```

Description

`[momentary,shortTerm,`integrated`,range,peak] = loudMtr(audioIn)` returns measurement values for momentary and short-term loudness of the input to your loudness meter, and the true-peak value of the current input frame, `audioIn`. It also returns the integrated loudness and loudness range of the input to your loudness meter since the last time `reset` was called.

Input Arguments

`audioIn` — Audio input to loudness meter
matrix

Audio input to the loudness meter, specified as a matrix. The columns of the matrix are treated as independent audio channels.

Note If you use the default `ChannelWeights` of the `loudnessMeter`, as a best practice, specify the input channels in this order: [Left, Right, Center, Left surround, Right surround].

Data Types: `single` | `double`

Output Arguments

`momentary` — Momentary loudness (LUFS)
column vector

Momentary loudness in loudness units relative to full scale (LUFS), returned as a column vector with the same number of rows as `audioIn`.

By default, loudness measurements are returned in LUFS. If you set the `UseRelativeScale` property to `true`, loudness measurements are returned in loudness units (LU).

Data Types: `single` | `double`

`shortTerm` — Short-term loudness (LUFS)
column vector

Short-term loudness in loudness units relative to full scale (LUFS), returned as a column vector with the same number of rows as `audioIn`.

By default, loudness measurements are returned in LUFS. If you set the `UseRelativeScale` property to `true`, loudness measurements are returned in loudness units (LU).

Data Types: `single` | `double`

`integrated` — Integrated loudness (LUFS)
column vector

Integrated loudness in loudness units relative to full scale (LUFS), returned as a column vector with the same number of rows as `audioIn`.

By default, loudness measurements are returned in LUFS. If you set the `UseRelativeScale` property to `true`, loudness measurements are returned in loudness units (LU).

Data Types: `single` | `double`

`range` — Loudness rage (LU)
column vector

Loudness range in loudness units (LU), returned as a column vector with the same number of rows as `audioIn`.

Data Types: `single` | `double`

`peak` — True-peak loudness (dB-TP)
scalar

True-peak loudness in dB-TP, returned as a column vector with the same number of rows as `audioIn`.

Data Types: `single` | `double`

Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

Specific to `loudnessMeter`

visualize	Open 'EBU Mode' meter display

Common to All System Objects

clone	Create duplicate System object
isLocked	Determine if System object is in use
release	Release resources and allow changes to System object property values and input characteristics
reset	Reset internal states of System object
step	Run System object algorithm

Examples

Loudness of Audio Signal

Create a `dsp.AudioFileReader` System object™ to read in an audio file. Create a `loudnesMeter` System object. Use the sample rate of the audio file as the sample rate of the `loudnessMeter`.

```
fileReader = dsp.AudioFileReader('RockDrums-44p1-stereo-11secs.mp3');
loudMtr = loudnessMeter('SampleRate',fileReader.SampleRate);
```

Read in the audio file in an audio stream loop. Use the loudness meter to determine the momentary, short-term, and integrated loudness of the audio signal. Cache the loudness measurements for analysis.

```
momentary = [];
shortTerm = [];
integrated = [];

while ~isDone(fileReader)
```

```
    x = fileReader();
    [m,s,i] = loudMtr(x);
    momentary = [momentary;m];
    shortTerm = [shortTerm;s];
    integrated = [integrated;i];
end

release(fileReader)
```

Plot the momentary, short-term, and integrated loudness of the audio signal.

```
t = linspace(0,11,length(momentary));
plot(t,[momentary,shortTerm,integrated])
title('Loudness Measurements')
legend('Momentary','Short-term','Integrated')
xlabel('Time (seconds)')
ylabel('LUFS')
```

Plot Momentary Loudness and Loudness Range of Audio Stream

Create an audio file reader and an audio device writer.

```
fileReader = dsp.AudioFileReader('FunkyDrums-44p1-stereo-25secs.mp3', ...
    'SamplesPerFrame',1024);
fs = fileReader.SampleRate;
deviceWriter = audioDeviceWriter('SampleRate',fs);
```

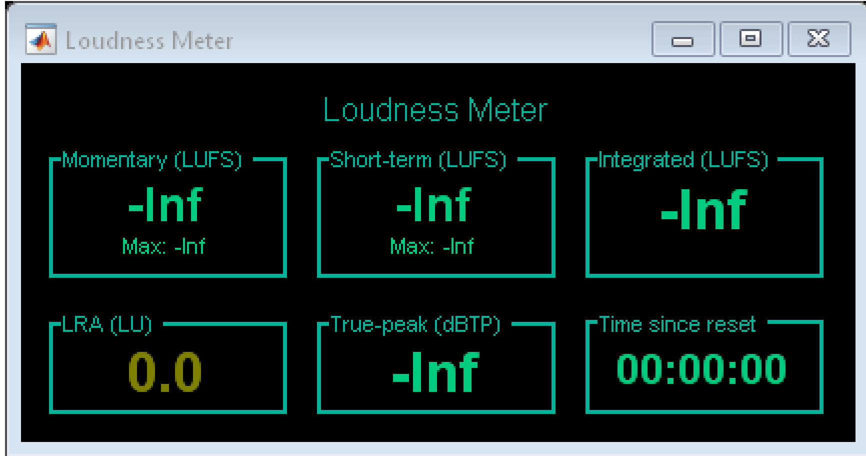
Create a time scope to visualize your audio stream loop.

```
timeScope = dsp.TimeScope('NumInputPorts',2, ...
    'SampleRate',fs, ...
    'TimeSpanOverrunAction','Scroll', ...
    'LayoutDimensions',[2,1], ...
    'TimeSpan',5, ...
    'BufferLength',5*fs);

% Top subplot of scope
timeScope.Title = 'Momentary Loudness';
timeScope.YLabel = 'LUFS';
timeScope.YLimits = [-40, 0];

% Bottom subplot of scope
```

```
timeScope.ActiveDisplay = 2;
timeScope.Title = 'Loudness Range';
timeScope.YLabel = 'LU';
timeScope.YLimits = [-1, 2];
```

Create a loudness meter. Use the sample rate of your input file as the sample rate of your loudness meter. Call `visualize` to open an 'EBU-mode' visualization for your loudness meter.

```
loudMtr = loudnessMeter('SampleRate',fs);
visualize(loudMtr)
```



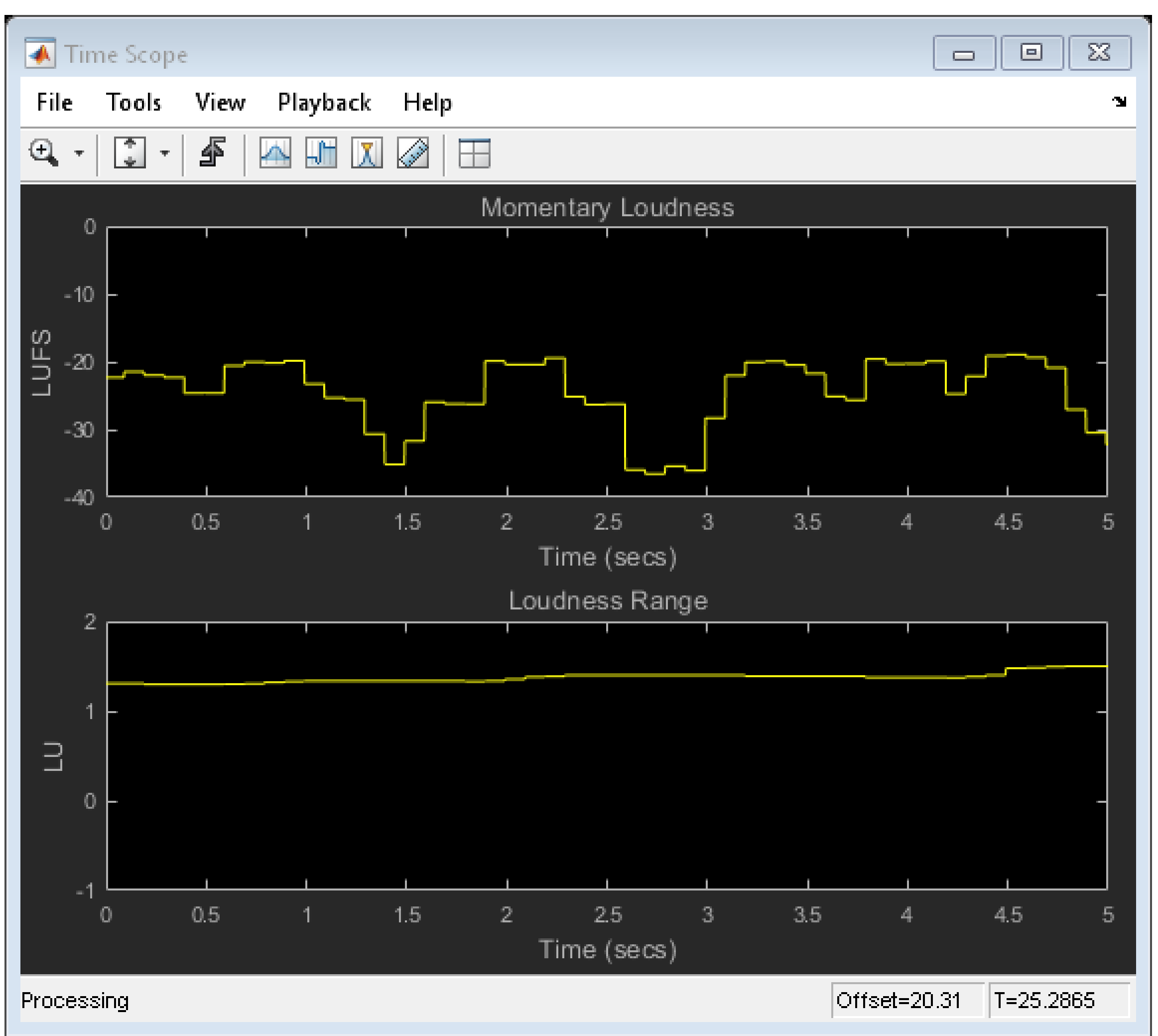
In an audio stream loop:

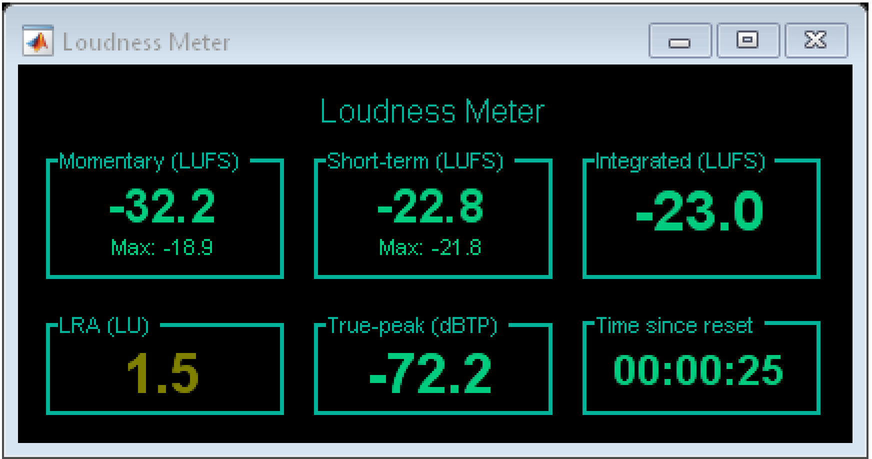
- Read in your audio file.
- Compute the momentary loudness and loudness range.
- Visualize the momentary loudness and loudness range on your time scope.
- Play the audio signal.

The 'EBU-mode' loudness meter visualization updates automatically while it is open. As a best practice, release your file reader and device writer once the loop is completed.

```
while ~isDone(fileReader)
    audioIn = fileReader();
    [momentaryLoudness,~,~,LRA] = loudMtr(audioIn);
    timeScope(momentaryLoudness,LRA);
```

```
    deviceWriter(audioIn);
end

release(fileReader)
release(deviceWriter)
```





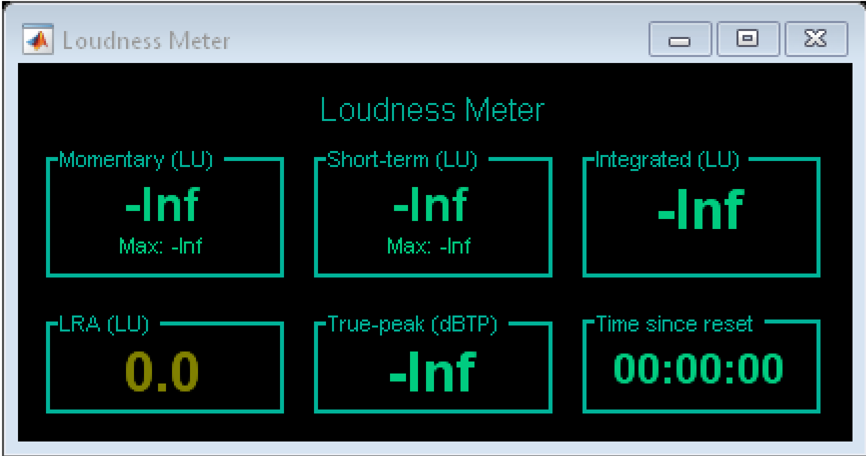
Relative Scale for Loudness Measurements

Create an audio file reader to read in an audio file. Create an audio device writer to write the audio file to your audio device. Use the sample rate of your file reader as the sample rate of your device writer.

```
fileReader = dsp.AudioFileReader('Counting-16-44p1-mono-15secs.wav',...
    'SamplesPerFrame',1024);
fs = fileReader.SampleRate;
deviceWriter = audioDeviceWriter('SampleRate',fs);
```

Create a loudness meter with the target loudness set to the default `-23` LUFS. Open the 'EBU-mode' loudness meter visualization.

```
loudMtr = loudnessMeter('UseRelativeScale',true);
visualize(loudMtr)
```

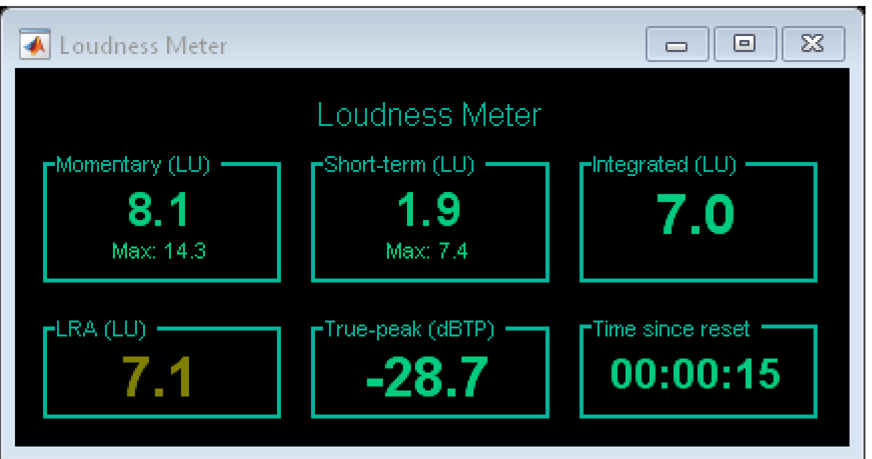


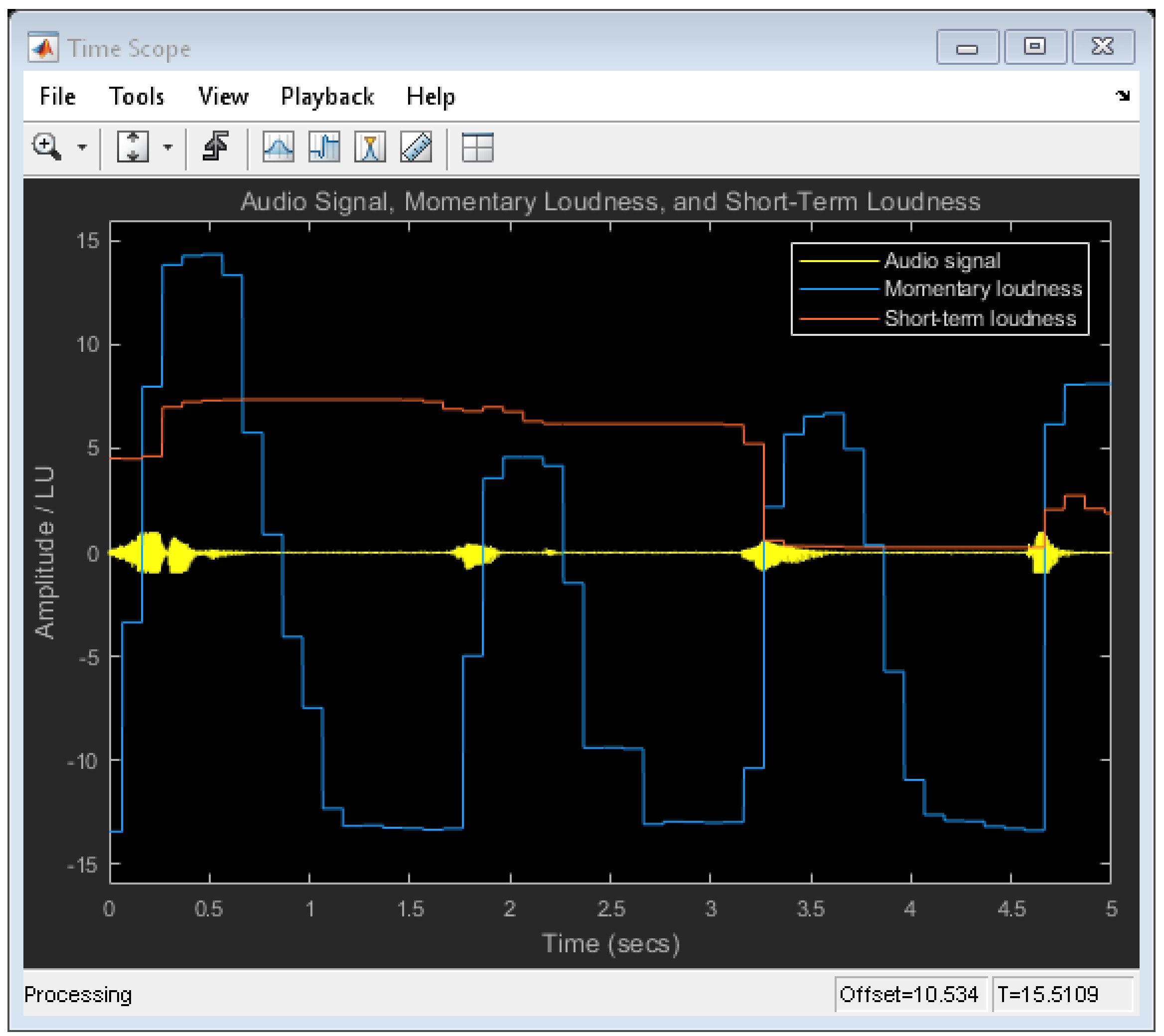
Create a time scope to visualize your audio signal and its measured relative momentary and short-term loudness.

```
scope = dsp.TimeScope( ...
    'NumInputPorts',3, ...
    'SampleRate',fs, ...
    'TimeSpanOverrunAction','Scroll', ...
    'TimeSpan',5, ...
    'BufferLength',5*fs, ...
    'Title','Audio Signal, Momentary Loudness, and Short-Term Loudness', ...
    'ChannelNames',{'Audio signal','Momentary loudness','Short-term loudness'}, ...
    'YLimits',[-16,16], ...
    'YLabel','Amplitude / LU', ...
    'ShowLegend',true);
```

In an audio stream loop, listen to and visualize the audio signal.

```
while ~isDone(fileReader)
    x = fileReader();
    [momentary,shortTerm] = loudMtr(x);
    scope(x,momentary,shortTerm)
    deviceWriter(x);
end

release(deviceWriter)
release(fileReader)
```

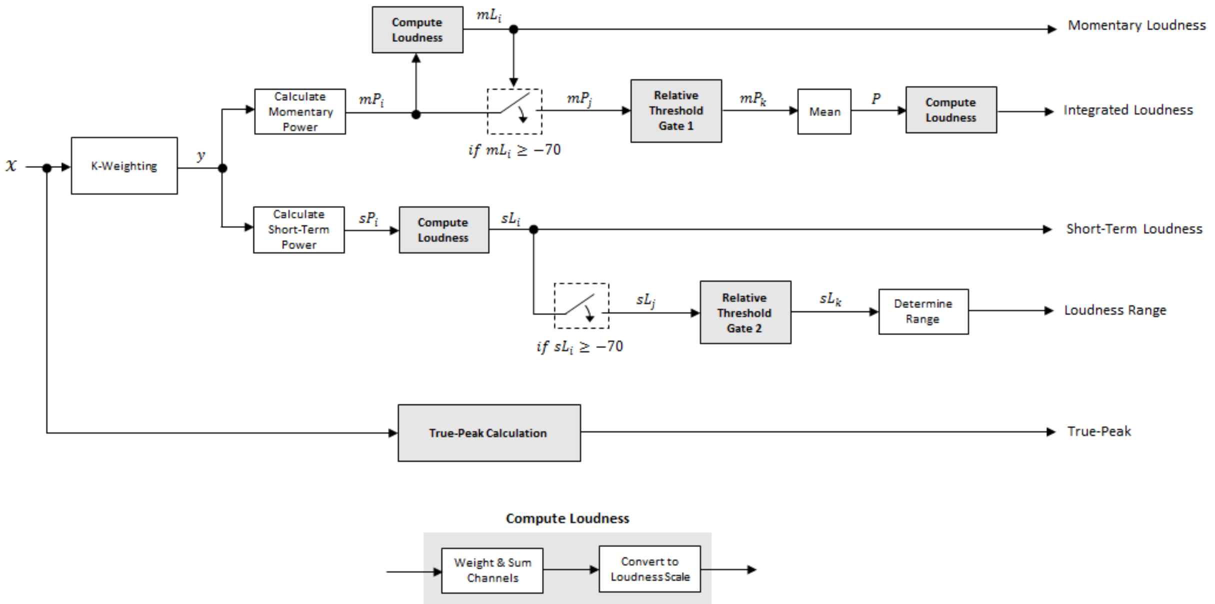

Loudness Meter
Loudness Meter
Momentary (LU)
8.1
Max: 14.3
Short-term (LU)
1.9
Max: 7.4
Integrated (LU)
7.0
LRA (LU)
7.1
True-peak (dBTP)
-28.7
Time since reset
00:00:15



Algorithms

The `loudnessMeter` System object calculates the momentary loudness, short-term loudness, integrated loudness, loudness range (LRA), and true-peak value of an audio signal. You can specify any number of channels and nondefault channel weights used for

loudness measurements. The `loudnessMeter` algorithm is described for the general case of n channels with default channel weights.



Loudness Measurements

The input channels, x, pass through a K-weighted `weightingFilter`. The K-weighted filter shapes the frequency spectrum to reflect perceived loudness.

1 The K-weighted channels, y, are divided into 0.4-second segments with 0.3-second overlap. If the required number of samples have not been collected yet, the `loudnessMeter` System object returns the last computed values for momentary and integrated loudness. If enough samples have been collected, then the power (mean square) of each segment of the K-weighted channels is calculated:

$$mP_i = \frac{1}{w} \sum_{k=1}^{w} y_i^2[k]$$

- mP_i is the momentary power of the ith segment.

- w is the segment length in samples.

2 The momentary loudness, mL, is computed in LUFS for each segment:

$$mL_i = -0.691 + 10\log_{10}\left(\sum_{c=1}^{n} G_c \times mP_{(i,c)}\right)$$

- G_c is the weighting for channel c.

mL is the momentary loudness returned by your `loudnessMeter` System object. It is also used internally to calculate the integrated loudness (steps 3–6).

3 The *integrated loudness* measurement considers the audio signal since the last reset of your loudness meter. To calculate integrated loudness, the momentary power is passed through a gating system. The gate system pauses the measurement during periods of low sound, such as stretches of silence in a movie.

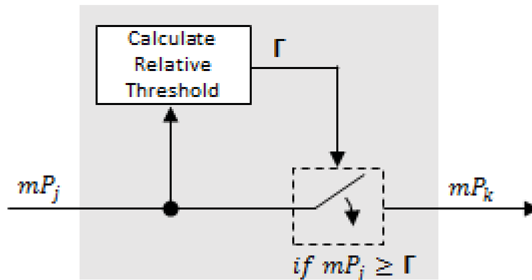
The momentary power segment is gated using the corresponding momentary loudness segment calculation:

$$mP_i \rightarrow mP_j$$

$$j = \left\{\, i \mid mL_i \geq -70 \,\right\}$$

mP_j is cached until your `loudnessMeter` is reset.

4 The momentary power subset, mP_j, passes through a relative threshold gate.



a The relative threshold, Γ, is computed:

$$\Gamma = -0.691 + 10\log_{10}\left(\sum_{c=1}^{n} G_c \times l_c\right) - 10$$

l_c is the mean momentary power of channel c:

$$l_c = \frac{1}{|j|}\sum_{j} mP_{(j,c)}$$

b The momentary power subset, mP_j, is gated using relative threshold Γ:

$$mP_j \rightarrow mP_k$$

$$k = \{\ j \mid mP_j \geq \Gamma\ \}$$

The relative threshold is recomputed during each call to your `loudnessMeter` object. The cached values of mP_j are gated again depending on the updated value of Γ.

5 The momentary power segments are averaged:

$$P = \frac{1}{|k|}\sum_{k} mP_k$$

6 The integrated loudness is computed in LUFS by passing the mean momentary power, P, through the Compute Loudness system:

$$\text{Integrated Loudness} = -0.691 + 10\log_{10}\left(\sum_{c=1}^{n} G_c \times P_c\right)$$

1 The K-weighted channels, y, are divided into 3-second segments with 2.9-second overlap. If the required number of samples have not been collected yet, the `loudnessMeter` System object returns the last computed values for short-term loudness and loudness range. If enough samples have been collected, then the power (mean square) of each K-weighted channel is calculated:

$$sP_i = \frac{1}{w}\sum_{k=1}^{w} y_i^2[k]$$

- sP_i is the short-term power of the ith segment of a channel.
- w is the segment length in samples.

2 The short-term loudness, sL, is computed in LUFS for each segment:

$$sL_i = -0.691 + 10\ \log_{10}\left(\sum_{c=1}^{n} G_c \times sP_{(i,c)}\right)$$

- G_c is the weighting for channel c.

sL is the short-term loudness returned by your `loudnessMeter` System object. It is also used internally to calculate the loudness range (steps 3–5).

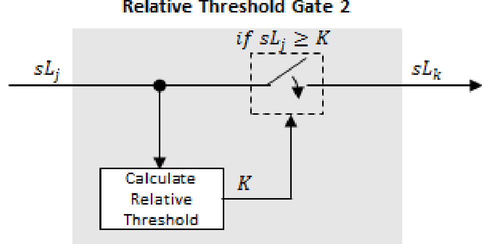
3 The short-term loudness is gated using an absolute threshold:

$$sL_i \rightarrow sL_j$$

$$j = \{\ i\ |\ sL_i \geq -70\ \}$$

sL_j is cached until your `loudnessMeter` is reset.

4 The short-term loudness subset, sL_j passes through a relative threshold gate.



a The gated short-term loudness is converted back to linear and then the mean is taken:

$$sP_j = \frac{1}{|j|} \sum_j 10^{\left(sL_j / 10 \right)}$$

The relative threshold, *K*, is computed:

$$K = -20 + 10 \log_{10} \left(sP_j \right)$$

b The short-term loudness subset, sL_j, is gated using the relative threshold:

$$sL_j \rightarrow sL_k$$

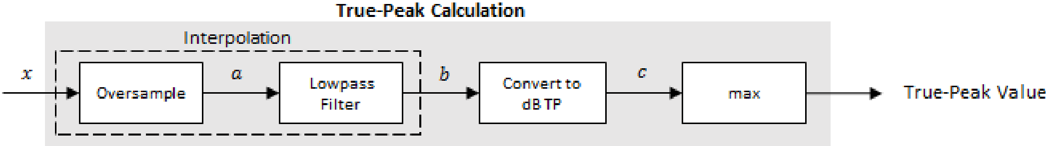
$$k = \left\{ j \mid sL_j \geq K \right\}$$

The relative threshold, *K*, is recomputed during each call to your `loudnessMeter` object. The cached values of sL_j are gated again depending on the updated value of *K*.

5 The short-term loudness subset, sL_k, is sorted. The loudness range is calculated as between the 10th and 95th percentiles of the distribution and is returned in loudness units (LU).

True-Peak

The *true-peak* measurement considers only the current input frame of a call to your loudness meter.



1 The signal is oversampled to at least 192 kHz. To optimize processing, the input sample rate determines the exact oversampling. An input sample rate below 750 Hz is not considered.

Input Sample Rate (kHz)	Upsample Factor
[0.75, 1.5)	256

Input Sample Rate (kHz)	Upsample Factor
[1.5, 3)	128
[3, 6)	64
[6,12)	32
[12, 24)	16
[24, 48)	8
[48, 96)	4
[96,192)	2
[192, ∞)	Not required

2. The oversampled signal, *a*, passes through a lowpass filter with a half-polyphase length of 12 and stopband attenuation of 80 dB. The filter design uses `designMultirateFIR`.
3. The filtered signal, *b*, is rectified and converted to the dB TP scale:

$$c = 20 \times \log_{10}(|b|)$$

4. The true-peak is determined as the maximum of the converted signal, *c*.

References

[1] International Telecommunication Union; Radiocommunication Sector. *Algorithms to Measure Audio Programme Loudness and True-Peak Audio Level*. ITU-R BS. 1770-4. 2015.

[2] European Broadcasting Union. *Loudness Normalisation and Permitted Maximum Level of Audio Signals*. EBU R 128. 2014.

[3] European Broadcasting Union. *Loudness Metering: 'EBU Mode' Metering to Supplement EBU R 128 Loudness Normalization*. EBU R 128 Tech 3341. 2014.

[4] European Broadcasting Union. *Loudness Range: A Measure to Supplement EBU R 128 Loudness Normalization*. EBU R 128 Tech 3342. 2016.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

"System Objects in MATLAB Code Generation" (MATLAB Coder)

Supports MATLAB Function block: No

Dynamic Memory Allocation must not be turned off.

See Also

System Objects
`octaveFilter` | `weightingFilter`

Blocks
Loudness Meter

Functions
`integratedLoudness`

Introduced in R2016b

visualize

Open 'EBU Mode' meter display

Syntax

```
visualize(loudMtr)
```

Description

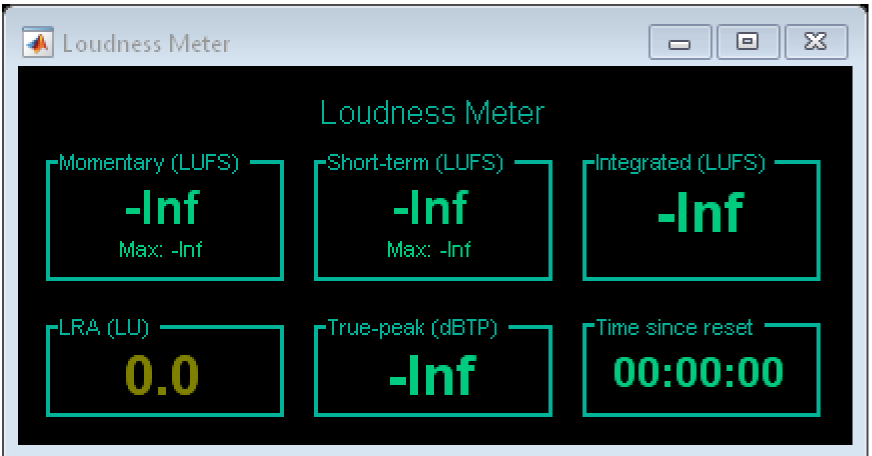
`visualize(loudMtr)` opens an 'EBU Mode' loudness meter display. The values of momentary loudness, short-term loudness, integrated loudness, loudness range, and true-peak are updated as the simulation progresses. The display also shows the maximum value of momentary and short-term loudness, and the time since the last call to `reset`.

Examples

Open an 'EBU Mode' Loudness Meter Display

Create an object of the `loudnessMeter` System object™, and then call `visualize` to open an 'EBU Mode' loudness meter display.

```
loudMtr = loudnessMeter;
visualize(loudMtr)
```

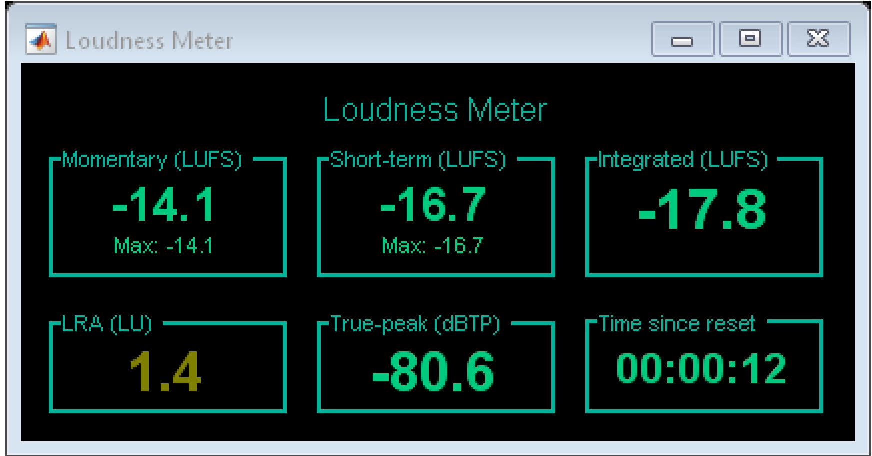


Create an audio file reader System object and specify the audio file to analyze. Create an audio device writer System object to play the audio to your output device.

```
fileReader = dsp.AudioFileReader('RockDrums-48-stereo-11secs.mp3');
deviceWriter = audioDeviceWriter('SampleRate',fileReader.SampleRate);
```

In an audio stream loop, read the audio from the file and play it to your device. The loudness meter visualization updates at each call.

```
while ~isDone(fileReader)
    audioIn = fileReader();
    loudMtr(audioIn);
    deviceWriter(audioIn);
end
```



Input Arguments

`loudMtr` — Object of `loudnessMeter`
object

Object of the `loudnessMeter` System object.

See Also

Blocks
Loudness Meter

Functions
`integratedLoudness`

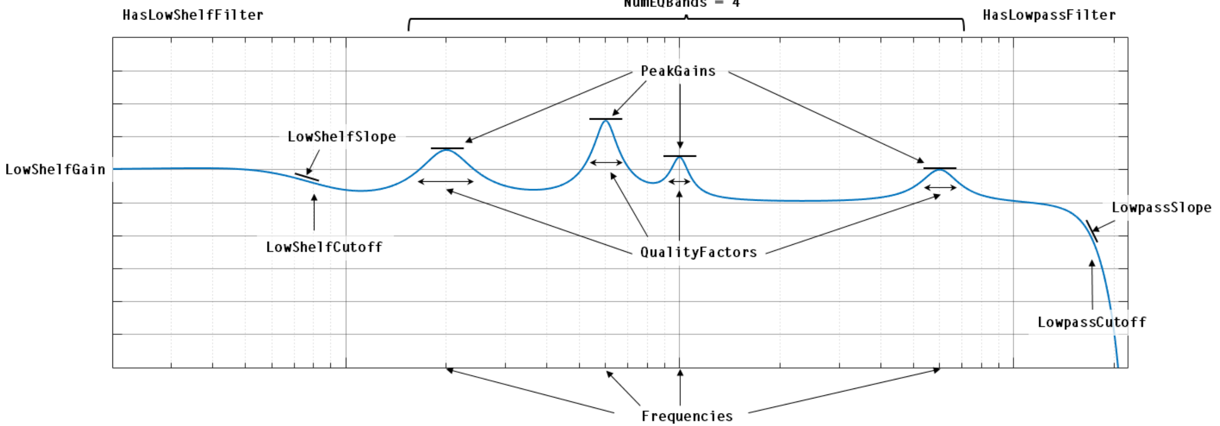
Introduced in R2016b

multibandParametricEQ System object

Multiband parametric equalizer

Description

The `multibandParametricEQ` System object performs multiband parametric equalization independently across each channel of input using specified center frequencies, gains, and quality factors. You can configure the System object with up to 10 bands. You can add low-shelf and high-shelf filters, as well as highpass (low-cut) and lowpass (high-cut) filters.



To implement a multiband parametric equalizer:

1. Create the multibandParametricEQ object and set its properties.
2. Call the object with arguments, as if it were a function.

To learn more about how System objects work, see What Are System Objects? (MATLAB).

Creation

Syntax

```
mPEQ = multibandParametricEQ
mPEQ = multibandParametricEQ(Name,Value)
```

Description

`mPEQ = multibandParametricEQ` creates a System object, `mPEQ`, that performs multiband parametric equalization.

`mPEQ = multibandParametricEQ(Name,Value)` sets each construction argument or property `Name` to the specified `Value`. Unspecified properties and creation arguments have default values.

Example: `mPEQ = multibandParametricEQ('NumEQBands',3,'Frequencies',[300,1200,5000])` creates a multiband parametric equalizer System object, `mPEQ`, with `NumEQBands` set to `3` and the `Frequencies` property set to `[300,1200,5000]`.

Note The value specified by NumEQBands must be the length of the row vectors specified by Frequencies, QualityFactors, and PeakGains. During creation of the System object, the first property you specify locks the value.

Creation Arguments

Creation arguments are properties which are set during creation of the System object and cannot be modified later. If you do not explicitly set a creation argument value, the property takes a default value.

NumEQBands — Number of equalizer bands
3 (default) | integer in the range [1, 10]

Number of equalizer bands, specified as an integer in the range [1, 10]. The number of equalizer bands does not include shelving filters, highpass filters, or lowpass filters.

`NumEQBands` is set during creation of the System object and cannot be modified later. If you do not explicitly set its value, the property takes the default value.

Example: `mPEQ = multibandParametricEQ('NumEQBands',5)` creates a multiband parametric equalizer with 5 bands.

Data Types: `single` | `double`

`EQOrder` — Order of individual equalizer bands
`2` (default) | even integer

Order of individual equalizer bands, specified as an even integer. All equalizer bands have the same order.

`EQOrder` is set during creation of the System object and cannot be modified later. If you do not explicitly set its value, the property takes the default value.

Example: `mPEQ = multibandParametricEQ('EQOrder',6)` creates a multiband parametric equalizer with the default 3 bands, all of order 6.

Data Types: `single` | `double`

`HasLowShelfFilter` — Low-shelf filter toggle
`false` (default) | `true`

Low-shelf filter toggle, specified as `false` or `true`.

- `false` –– Do not enable low-shelf filtering in multiband parametric equalizer implementation.
- `true` –– Enable low-shelf filtering in multiband parametric equalizer implementation.

`HasLowShelfFilter` is set during creation of the System object and cannot be modified later. If you do not explicitly set its value, the property takes the default value.

Example: `mPEQ = multibandParametricEQ('HasLowShelfFilter',true)` creates a default multiband parametric equalizer with low-shelf filtering enabled.

Data Types: `logical`

`HasHighShelfFilter` — High-shelf filter toggle
`false` (default) | `true`

High-shelf filter toggle, specified as `false` or `true`.

- `false` –– Do not enable high-shelf filtering in multiband parametric equalizer implementation.

- `true` –– Enable high-shelf filtering in multiband parametric equalizer implementation.

`HasHighShelfFilter` is set during creation of the System object and cannot be modified later. If you do not explicitly set its value, the property takes the default value.

Example: `mPEQ = multibandParametricEQ('HasHighShelfFilter',true)` creates a default multiband parametric equalizer with high-shelf filtering enabled.

Data Types: `logical`

`HasLowpassFilter` — Lowpass filter toggle

`false` (default) | `true`

Lowpass filter toggle, specified as `false` or `true`.

- `false` –– Do not enable lowpass filtering in multiband parametric equalizer implementation.
- `true` –– Enable lowpass filtering in multiband parametric equalizer implementation.

`HasLowpassFilter` is set during creation of the System object and cannot be modified later. If you do not explicitly set its value, the property takes the default value.

Example: `mPEQ = multibandParametricEQ('HasLowpassFilter',true)` creates a default multiband parametric equalizer with lowpass filtering enabled.

Data Types: `logical`

`HasHighpassFilter` — Highpass filter toggle

`false` (default) | `true`

Highpass filter toggle, specified as `false` or `true`.

- `false` –– Do not enable highpass filtering in multiband parametric equalizer implementation.
- `true` –– Enable highpass filtering in multiband parametric equalizer implementation.

`HasHighpassFilter` is set during creation of the System object and cannot be modified later. If you do not explicitly set its value, the property takes the default value.

Example: `mPEQ = multibandParametricEQ('HasHighpassFilter',true)` creates a default multiband parametric equalizer with highpass filtering enabled.

Data Types: `logical`

Oversample — Oversample toggle
false (default) | true

Oversample toggle, specified as false or true.

- false -- Runs the multiband parametric equalizer at the input sample rate.
- true -- Runs the multiband parametric equalizer at two times the input sample rate. Oversampling minimizes the frequency-warping effects introduced by the bilinear transformation.

  A halfband interpolator implements oversampling before equalization. A halfband decimator reduces the sample rate back to the input sampling rate after equalization.

Oversample is set during creation of the System object and cannot be modified later. If you do not explicitly set its value, the property takes the default value.

Example: mPEQ = multibandParametricEQ('Oversample',true) creates a default multiband parametric equalizer with oversampling enabled.

Data Types: logical

Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the release function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see System Design in MATLAB Using System Objects (MATLAB).

Multiband Equalizer

Frequencies — Center frequencies of equalizer bands (Hz)
[100, 181, 325] (default) | row vector of length NumEQBands

Center frequencies of equalizer bands in Hz, specified as a row vector of length NumEQBands. The vector consists of real scalars in the range 0 to SampleRate/2.

Tunable: Yes

Data Types: `single` | `double`

`QualityFactors` — Quality factors of equalizer bands

`[1.6,1.6,1.6]` (default) | row vector of length `NumEQBands`

Quality factors of equalizer bands, specified as a row vector of length NumEQBands. The vector consists of real scalars in the range [0.2, 700]. Any values outside the range are saturated.

Tunable: Yes

Data Types: `single` | `double`

`PeakGains` — Peak or dip filter gains (dB)

`[0,0,0]` (default) | row vector of length `NumEQBands`

Peak or dip filter gains in dB, specified as a row vector of length NumEQBands. The vector consists of real scalars in the range [–inf, 20]. Values above 20 are saturated.

Tunable: Yes

Data Types: `single` | `double`

Low-Shelf Filter

`LowShelfCutoff` — Low-shelf filter cutoff (Hz)

`200` (default) | scalar

Low-shelf filter cutoff in Hz, specified as a scalar greater than or equal to 0.

Tunable: Yes

Dependencies

To enable this property, set HasLowShelfFilter to `true` during creation.

Data Types: `single` | `double`

`LowShelfSlope` — Low-shelf filter slope coefficient

`1.5` (default) | real scalar in the range [0.1, 5]

Low-shelf filter slope coefficient, specified as a real scalar in the range [0.1, 5]. Values outside the range are saturated.

Tunable: Yes

Dependencies

To enable this property, set HasLowShelfFilter to `true` during creation.

Data Types: `single` | `double`

`LowShelfGain` — Low-shelf filter gain (dB)

`0` (default) | real scalar in the range [–12, 12]

Low-shelf filter gain in dB, specified as a real scalar in the range [–12, 12]. Values outside the range are saturated.

Tunable: Yes

Dependencies

To enable this property, set HasLowShelfFilter to `true` during creation.

Data Types: `single` | `double`

High-Shelf Filter

`HighShelfCutoff` — High-shelf filter cutoff (Hz)

`15000` (default) | nonnegative real scalar

High-shelf filter cutoff in Hz, specified as a real scalar greater than or equal to 0.

Tunable: Yes

Dependencies

To enable this property, set HasHighShelfFilter to `true` during creation.

Data Types: `single` | `double`

`HighShelfSlope` — High-shelf slope coefficient

`1.5` (default) | real scalar in the range [0.1, 5]

High-shelf filter slope coefficient, specified as a real scalar in the range [0.1, 5]. Values outside the range are saturated.

Tunable: Yes

Dependencies

To enable this property, set HasHighShelfFilter to `true` during creation.

Data Types: `single` | `double`

`HighShelfGain` — High-shelf filter gain (dB)

`0` (default) | real scalar in the range [–12, 12]

High-shelf filter gain in dB, specified as a real scalar in the range [–12, 12]. Values outside the range are saturated.

Tunable: Yes

Dependencies

To enable this property, set HasHighShelfFilter to `true` during creation.

Data Types: `single` | `double`

Lowpass Filter

`LowpassCutoff` — Lowpass filter cutoff frequency (Hz)

`18000` (default) | nonnegative real scalar

Lowpass filter cutoff frequency in Hz, specified as a real scalar greater than or equal to 0.

Tunable: Yes

Dependencies

To enable this property, set HasLowpassFilter to `true` during creation.

Data Types: `single` | `double`

`LowpassSlope` — Lowpass filter slope (dB/octave)

`12` (default) | real scalar in the range `[0:6:48]`

Lowpass filter slope in dB/octave, specified as a real scalar in the range `[0:6:48]`. Values that are not multiples of 6 are rounded to the nearest multiple of 6.

Tunable: Yes

Dependencies

To enable this property, set HasLowpassFilter to `true` during creation.

Data Types: `single` | `double`

Highpass Filter

`HighpassCutoff` — Highpass filter cutoff frequency (Hz)
`20` (default) | nonnegative real scalar

Highpass filter cutoff in Hz, specified as a real scalar greater than or equal to 0.

Tunable: Yes

Dependencies

To enable this property, set HasHighpassFilter to `true` during creation.

Data Types: `single` | `double`

`HighpassSlope` — Highpass filter slope (dB/octave)
`30` (default) | real scalar in the range `[0:6:48]`

Highpass filter slope in dB/octave, specified as a real scalar in the range `[0:6:48]`. Values that are not multiples of 6 are rounded to the nearest multiple of 6.

Tunable: Yes

Dependencies

To enable this property, set HasHighpassFilter to `true` during creation.

Data Types: `single` | `double`

Sampling

`SampleRate` — Input sample rate (Hz)
`44100` (default) | positive scalar

Input sample rate in Hz, specified as a positive scalar.

Tunable: Yes

Data Types: `single` | `double`

Usage

Note For versions earlier than R2016b, use the `step` function to run the System object™ algorithm. The arguments to `step` are the object you created, followed by the arguments shown in this section.

For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Syntax

`audioOut = mPEQ(audioIn)`

Description

`audioOut = mPEQ(audioIn)` performs multiband parametric equalization on the input signal, `audioIn`, and returns the filtered signal, `audioOut`. The type of equalization is specified by the algorithm and properties of the `multibandParametricEQ` System object, `mPEQ`.

Input Arguments

`audioIn` — Audio input to equalizer
matrix

Audio input to the equalizer, specified as a matrix. The columns of the matrix are treated as independent audio channels.

Data Types: `single` | `double`

Output Arguments

`audioOut` — Audio output from equalizer
matrix

Audio output from the equalizer, returned as a matrix the same size as `audioIn`.

Data Types: `single` | `double`

Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

Specific to `multibandParametricEQ`

createAudioPluginClass	Create audio plugin class that implements functionality of System object
visualize	Visualize magnitude response of multiband parametric equalizer

MIDI

configureMIDI	Configure MIDI connections between audio object and MIDI controller
disconnectMIDI	Disconnect MIDI controls from audio object
getMIDIConnections	Get MIDI connections of audio object

Common to All System Objects

clone	Create duplicate System object
isLocked	Determine if System object is in use
release	Release resources and allow changes to System object property values and input characteristics
reset	Reset internal states of System object
step	Run System object algorithm

The `createAudioPluginClass` and `configureMIDI` functions map tunable properties of the `multibandParametricEQ` System object to user-facing parameters:

Property	Range	Mapping	Unit
`Frequencies`	[20, 20000]	log	Hz
`QualityFactors`	[0.2, 700]	linear	none
`PeakGains`	[–50, 20]	linear	dB
`LowShelfCutoff`	[20, 20000]	log	Hz
`LowShelfSlope`	[0.1, 5]	linear	none

Property	Range	Mapping	Unit
LowShelfGain	[–12, 12]	linear	dB
HighShelfCutoff	[20, 20000]	log	Hz
HighShelfSlope	[0.1, 5]	linear	none
HighShelfGain	[–12, 12]	linear	dB
LowpassCutoff	[20, 20000]	log	Hz
LowpassSlope	[0, 48]	linear	dB/octave
HighpassCutoff	[20, 20000]	log	Hz
HighpassSlope	[0, 48]	linear	dB/octave

Examples

Multiband Parametric Equalization

Create `dsp.AudioFileReader` and `audioDeviceWriter` System objects™. Use the sample rate of the reader as the sample rate of the writer. Call `setup` to reduce the computational load of initialization in an audio stream loop.

```
frameLength = 512;

fileReader = dsp.AudioFileReader( ...
    'Filename','RockDrums-48-stereo-11secs.mp3', ...
    'SamplesPerFrame',frameLength);
deviceWriter = audioDeviceWriter( ...
    'SampleRate',fileReader.SampleRate);

setup(deviceWriter,ones(frameLength,2));
```

Construct a three-band parametric equalizer with a high-shelf filter.

```
mPEQ = multibandParametricEQ( ...
    'NumEQBands',3, ...
    'Frequencies',[300,1200,5000], ...
    'QualityFactors',[1,1,1], ...
    'PeakGains',[8,-10,7], ...
    'HasHighShelfFilter',true, ...
    'HighShelfCutoff',14000, ...
```

```
    'HighShelfSlope',0.3, ...
    'HighShelfGain',-5, ...
    'SampleRate',fileReader.SampleRate);
```

Visualize the magnitude frequency response of your multiband parametric equalizer.

```
visualize(mPEQ)
```

Play the equalized audio signal. Update the peak gains of your equalizer band to hear the effect of the equalizer and visualize the changing magnitude response.

```
count = 0;
while ~isDone(fileReader)
    originalSignal = fileReader();
    equalizedSignal = mPEQ(originalSignal);
    deviceWriter(equalizedSignal);
    if mod(count,100) == 0
        mPEQ.PeakGains(1) = mPEQ.PeakGains(1) - 1.5;
        mPEQ.PeakGains(2) = mPEQ.PeakGains(2) + 1.5;
        mPEQ.PeakGains(3) = mPEQ.PeakGains(3) - 1.5;
    end
    count = count + 1;
end

release(fileReader)
release(mPEQ)
release(deviceWriter)
```

Oversample Audio Signal

Reduce warping by specifying your `multibandParametricEQ` System object™ to perform oversampling before equalization.

Create a one-band equalizer. Visualize the equalizer band as its center frequency approaches the Nyquist rate.

```
mPEQ = multibandParametricEQ(...
    'NumEQBands',1,...
    'Frequencies',9.5e3,...
    'PeakGains',10);
visualize(mPEQ)
for i = 1:1000
```

```
    mPEQ.Frequencies = mPEQ.Frequencies + 8;
end
```

The equalizer band is warped.

Create a one-band equalizer with `Oversample` set to `true`. Visualize the equalizer band as its center frequency approaches the Nyquist rate.

```
mPEQOversampled = multibandParametricEQ(...
    'NumEQBands',1,...
    'Frequencies',9.5e3,...
    'PeakGains',10,...
    'Oversample',true);
visualize(mPEQOversampled)
for i = 1:1000
    mPEQOversampled.Frequencies = mPEQOversampled.Frequencies + 8;
end
```

Warping is reduced.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

“System Objects in MATLAB Code Generation” (MATLAB Coder)

See Also

Blocks

Parametric EQ Filter

Functions

`designParamEQ` | `designShelvingEQ` | `designVarSlopeFilter`

Topics

“Parametric Equalizer Design”
“Equalization”

Introduced in R2016a

visualize

Visualize magnitude response of multiband parametric equalizer

Syntax

```
visualize(mPEQ)
visualize(obj,NFFT)
```

Description

`visualize(mPEQ)` plots the magnitude response of the `multibandParametricEQ` object, `mPEQ`. The plot is updated automatically when properties of the object change.

`visualize(obj,NFFT)` specifies an N-point FFT used to calculate the magnitude response.

Examples

Specify a Nondefault Number of FFT Points

Create an object of the `multibandParametricEQ` System object™, and then call `visualize` to plot the magnitude response using a 5096-point FFT.

```
mPEQ = multibandParametricEQ('PeakGains',[-inf,5,5]);
visualize(mPEQ,5096)
```

Input Arguments

obj — Multiband parametric equalizer to visualize
object of `multibandParametricEQ` System object

Multiband parametric equalizer whose magnitude response you want to plot.

NFFT — N-point FFT
2048 (default) | positive scalar

Number of bins used to calculate the DFT, specified as a positive scalar.

Data Types: `single` | `double`

See Also

System Objects
`multibandParametricEQ`

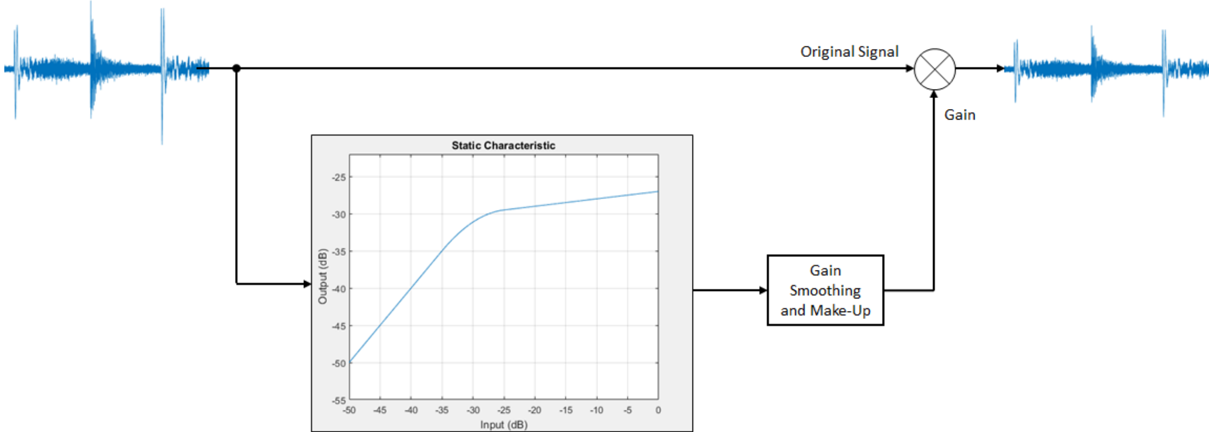
Introduced in R2016a

compressor System object

Dynamic range compressor

Description

The `compressor` System object performs dynamic range compression independently across each input channel. Dynamic range compression attenuates the volume of loud sounds that cross a given threshold. It uses specified attack and release times to achieve a smooth applied gain curve. Properties of the `compressor` System object specify the type of dynamic range compression.



To perform dynamic range compression:

1. Create the compressor object and set its properties.
2. Call the object with arguments, as if it were a function.

To learn more about how System objects work, see What Are System Objects? (MATLAB).

Creation

Syntax

```
dRC = compressor
dRC = compressor(thresholdValue)
dRC = compressor(thresholdValue,ratioValue)
dRC = compressor(___,Name,Value)
```

Description

`dRC = compressor` creates a System object, `dRC`, that performs dynamic range compression independently across each input channel over time.

`dRC = compressor(thresholdValue)` sets the Threshold property to `thresholdValue`.

`dRC = compressor(thresholdValue,ratioValue)` sets the Ratio property to `ratioValue`.

`dRC = compressor(___,Name,Value)` sets each property `Name` to the specified `Value`. Unspecified properties have default values.

Example: `dRC = compressor('AttackTime',0.01,'SampleRate',16000)` creates a System object, `dRC`, with a 10 ms attack time operating at a 16 kHz sample rate.

Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see System Design in MATLAB Using System Objects (MATLAB).

`Threshold` — Operation threshold (dB)
`–10` (default) | real scalar

Operation threshold in dB, specified as a real scalar.

Operation threshold is the level above which gain is applied to the input signal.

Tunable: Yes

Data Types: `single` | `double`

`Ratio` — Compression ratio
`5` (default) | real scalar

Compression ratio, specified as a real scalar greater than or equal to 1.

Compression ratio is the input/output ratio for signals that overshoot the operation threshold.

Assuming a hard knee characteristic and a steady-state input such that $x[n]$ dB > `Threshold`, the compression ratio is defined as $R = \frac{(x[n] - T)}{(y[n] - T)}$.

- R is the compression ratio.
- $x[n]$ is the input signal in dB.
- $y[n]$ is the output signal in dB.
- T is the threshold in dB.

Tunable: Yes

Data Types: `single` | `double`

`KneeWidth` — Knee width (dB)
`0` (default) | real scalar

Knee width in dB, specified as a real scalar greater than or equal to 0.

Knee width is the transition area in the compression characteristic.

For soft knee characteristics, the transition area is defined by the relation

$$y = x + \frac{\left(\frac{1}{R} - 1\right) \times \left(x - T + \frac{W}{2}\right)^2}{(2 \times W)}$$

for the range $\left(2\times\left|x-T\right|\right)\leq W$.

- *y* is the output level in dB.
- *x* is the input level in dB.
- *R* is the compression ratio.
- *T* is the threshold in dB.
- *W* is the knee width in dB.

Tunable: Yes

Data Types: `single` | `double`

`AttackTime` — Attack time (s)
`0.05` (default) | real scalar

Attack time in seconds, specified as a real scalar greater than or equal to 0.

Attack time is the time it takes the compressor gain to rise from 10% to 90% of its final value when the input goes above the threshold.

Tunable: Yes

Data Types: `single` | `double`

`ReleaseTime` — Release time (s)
`0.2` (default) | real scalar

Release time in seconds, specified as a real scalar greater than or equal to 0.

Release time is the time it takes the compressor gain to drop from 90% to 10% of its final value when the input goes below the threshold.

Tunable: Yes

Data Types: `single` | `double`

`MakeUpGainMode` — Make-up gain mode
`'Property'` (default) | `'Auto'`

Make-up gain mode, specified as `'Auto'` or `'Property'`.

- `'Auto'` –– Make-up gain is applied at the output of the dynamic range compressor such that a steady-state 0 dB input has a 0 dB output.
- `'Property'` –– Make-up gain is set to the value specified in the MakeUpGain property.

Tunable: No

Data Types: `char` | `string`

`MakeUpGain` — Make-up gain (dB)
`0` (default) | real scalar

Make-up gain in dB, specified as a real scalar.

Make-up gain compensates for gain lost during compression. It is applied at the output of the dynamic range compressor.

Tunable: Yes

Dependencies

To enable this property, set MakeUpGainMode to `'Property'`.

Data Types: `single` | `double`

`SampleRate` — Input sample rate (Hz)
`44100` (default) | positive scalar

Input sample rate in Hz, specified as a positive scalar.

Tunable: Yes

Data Types: `single` | `double`

Usage

Note For versions earlier than R2016b, use the `step` function to run the System object™ algorithm. The arguments to `step` are the object you created, followed by the arguments shown in this section.

For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Syntax

```
audioOut = dRC(audioIn)
[audioOut,gain] = dRC(audioIn)
```

Description

`audioOut = dRC(audioIn)` performs dynamic range compression on the input signal, `audioIn`, and returns the compressed signal, `audioOut`. The type of dynamic range compression is specified by the algorithm and properties of the `compressor` System object, `dRC`.

`[audioOut,gain] = dRC(audioIn)` also returns the applied gain, in dB, at each input sample.

Input Arguments

audioIn — Audio input to compressor
matrix

Audio input to the compressor, specified as a matrix. The columns of the matrix are treated as independent audio channels.

Data Types: `single` | `double`

Output Arguments

audioOut — Audio output from compressor
matrix

Audio output from the compressor, returned as a matrix the same size as `audioIn`.

Data Types: `single` | `double`

gain — Gain applied by compressor (dB)
matrix

Gain applied by compressor, returned as a matrix the same size as `audioIn`.

Data Types: `single` | `double`

Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

Specific to compressor

visualize	Visualize static characteristic of dynamic range controller
createAudioPluginClass	Create audio plugin class that implements functionality of System object

MIDI

configureMIDI	Configure MIDI connections between audio object and MIDI controller
disconnectMIDI	Disconnect MIDI controls from audio object
getMIDIConnections	Get MIDI connections of audio object

Common to All System Objects

clone	Create duplicate System object
isLocked	Determine if System object is in use
release	Release resources and allow changes to System object property values and input characteristics
reset	Reset internal states of System object
step	Run System object algorithm

The `createAudioPluginClass` and `configureMIDI` functions map tunable properties of the `compressor` System object to user-facing parameters:

Property	Range	Mapping	Unit
`Threshold`	[–50, 0]	linear	dB
`Ratio`	[1, 50]	linear	none
`KneeWidth`	[0, 20]	linear	dB
`AttackTime`	[0, 4]	linear	seconds
`ReleaseTime`	[0, 4]	linear	seconds

Property	Range	Mapping	Unit
MakeUpGain (available when you set MakeUpGainMode to 'Property')	[–10, 24]	linear	dB

Examples

Compress Audio Signal

Use dynamic range compression to attenuate the volume of loud sounds.

Set up the dsp.AudioFileReader and audioDeviceWriter System objects™.

```
frameLength = 1024;
fileReader = dsp.AudioFileReader( ...
    'Filename','RockDrums-44p1-stereo-11secs.mp3', ...
    'SamplesPerFrame',frameLength);
deviceWriter = audioDeviceWriter( ...
    'SampleRate',fileReader.SampleRate);
```

Set up the compressor to have a threshold of -15 dB, a ratio of 7, and a knee width of 5 dB. Use the sample rate of your audio file reader.

```
dRC = compressor(-15,7,...
    'KneeWidth',5,...
    'SampleRate',fileReader.SampleRate);
```

Visualize the compression static characteristic.

```
visualize(dRC)
```

Set up the scope to visualize the original audio signal, the compressed audio signal, and the applied compressor gain.

```
scope = dsp.TimeScope( ...
    'SampleRate',fileReader.SampleRate, ...
    'TimeSpan',1, ...
    'BufferLength',44100*4, ...
    'YLimits',[-1,1], ...
```

```
    'TimeSpanOverrunAction','Scroll', ...
    'ShowGrid',true, ...
    'LayoutDimensions',[2,1], ...
    'NumInputPorts',2, ...
    'Title', ...
    ['Original vs. Compressed Audio (top)' ...
    ' and Compressor Gain in dB (bottom)']);
scope.ActiveDisplay = 2;
scope.YLimits = [-4,0];
scope.YLabel = 'Gain (dB)';
```

Play the processed audio and visualize it on the scope.

```
while ~isDone(fileReader)
    x = fileReader();
    [y,g] = dRC(x);
    deviceWriter(y);
    scope([x(:,1),y(:,1)],g(:,1))
end

release(dRC)
release(deviceWriter)
release(scope)
```

Compare Dynamic Range Limiter and Compressor

A dynamic range limiter is a special type of dynamic range compressor. In limiters, the level above an operational threshold is hard limited. In the simplest implementation of a limiter, the effect is equivalent to audio clipping. In compressors, the level above an operational threshold is lowered using a specified compression ratio. Using a compression ratio results in a smoother processed signal.

Compare Limiter and Compressor Applied to Sinusoid

Create a `limiter` System object™ and a `compressor` System object. Set the `AttackTime` and `ReleaseTime` properties of both objects to zero. Create an `audioOscillator` System object to generate a sinusoid with `Frequency` set to `5` and `Amplitude` set to `0.1`.

```
dRL = limiter('AttackTime',0,'ReleaseTime',0);
dRC = compressor('AttackTime',0,'ReleaseTime',0);
```

```
osc = audioOscillator('Frequency',5,'Amplitude',0.1);
```

Create a time scope to visualize the generated sinusoid and the processed sinusoid.

```
scope = dsp.TimeScope( ...
    'SampleRate',osc.SampleRate, ...
    'TimeSpan',2, ...
    'BufferLength',osc.SampleRate*4, ...
    'YLimits',[-1 1], ...
    'TimeSpanOverrunAction','Scroll', ...
    'ShowGrid',true, ...
    'LayoutDimensions',[2 1], ...
    'NumInputPorts',2, ...
    'Title', ...
    'Original Signal vs. Limited Signal (top) and Compressed Signal (bottom)');
```

In an audio stream loop, visualize the original sinusoid and the sinusoid processed by a limiter and a compressor. Increment the amplitude of the original sinusoid to illustrate the effect.

```
while osc.Amplitude < 0.75
    x = osc();

    xLimited    = dRL(x);
    xCompressed = dRC(x);

    scope([x xLimited],[x xCompressed]);

    osc.Amplitude = osc.Amplitude + 0.0002;
end
release(scope)
release(dRL)
release(dRC)
release(osc)
```

Compare Limiter and Compressor Applied to Audio Signal

Compare the effect of dynamic range limiters and compressors on a drum track. Create a `dsp.AudioFileReader` System object and a `audioDeviceWriter` System object to read audio from a file and write to your audio output device. To emphasize the effect of dynamic range control, set the operational threshold of the limiter and compressor to -20 dB.

```
dRL.Threshold = -20;
dRC.Threshold = -20;

fileReader = dsp.AudioFileReader('FunkyDrums-44p1-stereo-25secs.mp3');
deviceWriter = audioDeviceWriter('SampleRate',fileReader.SampleRate);
```

Read successive frames from an audio file in a loop. Listen to and compare the effect of dynamic range limiting and dynamic range compression on an audio signal.

```
numFrames = 300;

fprintf('Now playing original signal...\n')
for i = 1:numFrames
    x = fileReader();
    deviceWriter(x);
end
reset(fileReader);

fprintf('Now playing limited signal...\n')
for i = 1:numFrames
    x = fileReader();
    xLimited = dRL(x);
    deviceWriter(xLimited);
end
reset(fileReader);

fprintf('Now playing compressed signal...\n')
for i = 1:numFrames
    x = fileReader();
    xCompressed = dRC(x);
    deviceWriter(xCompressed);
end

release(fileReader)
release(deviceWriter)
release(dRC)
release(dRL)
```

Diminish Plosives from Speech Signal

Plosives are consonant sounds resulting from a sudden release of airflow. They are most pronounced in words beginning with *p*, *d*, and *g* sounds. Plosives can be emphasized by

the recording process and are often displeasurable to hear. In this example, you minimize the plosives of a speech signal by applying highpass filtering and low-band compression.

Create a `dsp.AudioFileReader` System object™ and a `audioDeviceWriter` System object™ to read an audio signal from a file and write an audio signal to a device. Play the unprocessed signal. Then release the file reader and device writer.

```
fileReader   = dsp.AudioFileReader( ...
    fullfile(matlabroot,'examples','audio','Plosives.wav'));
deviceWriter = audioDeviceWriter;

while ~isDone(fileReader)
    audioIn = fileReader();
    deviceWriter(audioIn);
end
release(deviceWriter)
release(fileReader)
```

Design a highpass filter with a steep rolloff of all frequencies below 120 Hz. Use a `dsp.BiquadFilter` System object to implement the highpass filter design. Create a crossover filter with one crossover at 250 Hz. The crossover filter enables you to separate the band of interest for processing. Create a dynamic range compressor to compress the dynamic range of plosive sounds. To apply no make-up gain, set the `MakeUpGainMode` to `'Property'` and use the default 0 dB `MakeUpGain` property value. Create a time scope to visualize the processed and unprocessed audio signal.

```
[B,A] = designVarSlopeFilter(48,120/(44100/2),'hi');
biquadFilter = dsp.BiquadFilter( ...
    'SOSMatrixSource','Input port', ...
    'ScaleValuesInputPort',false);

crossFilt = crossoverFilter( ...
    'NumCrossovers',1, ...
    'CrossoverFrequencies',250, ...
    'CrossoverSlopes',48);

dRCompressor = compressor( ...
    'Threshold',-35, ...
    'Ratio',10, ...
    'KneeWidth',20, ...
    'AttackTime',1e-4, ...
    'ReleaseTime',3e-1, ...
    'MakeUpGainMode','Property', ...
    'SampleRate',fileReader.SampleRate);
```

```
scope = dsp.TimeScope( ...
    'SampleRate',fileReader.SampleRate, ...
    'TimeSpan',3, ...
    'BufferLength',fileReader.SampleRate*3*2, ...
    'YLimits',[-1 1], ...
    'ShowGrid',true, ...
    'ShowLegend',true, ...
    'ChannelNames',{'Original','Processed'});
```

In an audio stream loop:

1. Read in a frame of the audio file.
2. Apply highpass filtering using your biquad filter.
3. Split the audio signal into two bands.
4. Apply dynamic range compression to the lower band.
5. Remix the channels.
6. Write the processed audio signal to your audio device for listening.
7. Visualize the processed and unprocessed signals on a time scope.

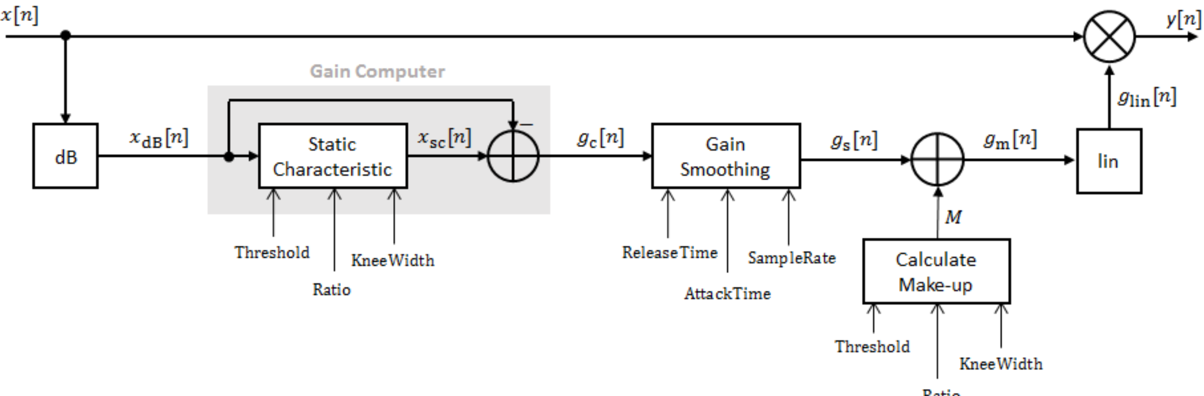
As a best practice, release your objects once done.

```
while ~isDone(fileReader)
    audioIn = fileReader();

    audioIn = biquadFilter(audioIn,B,A);

    [band1,band2] = crossFilt(audioIn);

    band1compressed = dRCompressor(band1);

    audioOut  = band1compressed + band2;

    deviceWriter(audioOut);

    scope([audioIn audioOut]);
end

release(deviceWriter)
release(fileReader)
release(scope)
```

```
release(crossFilt)
release(dRCompressor)
```

Algorithms

The `compressor` System object processes a signal frame by frame and element by element.



Convert Input Signal to dB

The N-point signal, $x[n]$, is converted to decibels:

$$x_{\mathrm{dB}}[n] = 20 \times \log_{10} |x[n]|$$

Gain Computer

$x_{\mathrm{dB}}[n]$ passes through the gain computer. The gain computer uses the static characteristic properties of the dynamic range compressor to attenuate gain that is above the threshold.

If you specified a soft knee, the gain computer has the following static characteristic:

$$x_{sc}(x_{dB}) = \begin{cases} x_{dB} & x_{dB} < \left(T - \frac{W}{2}\right) \\ x_{dB} + \frac{\left(\frac{1}{R} - 1\right)\left(x_{dB} - T + \frac{W}{2}\right)^2}{2W} & \left(T - \frac{W}{2}\right) \le x_{dB} \le \left(T + \frac{W}{2}\right) \\ T + \frac{(x_{dB} - T)}{R} & x_{dB} > \left(T + \frac{W}{2}\right) \end{cases},$$

where T is the threshold, R is the ratio, and W is the knee width.

If you specified a hard knee, the gain computer has the following static characteristic:

$$x_{sc}(x_{dB}) = \begin{cases} x_{dB} & x_{dB} < T \\ T + \frac{(x_{dB} - T)}{R} & x_{dB} \ge T \end{cases}$$

The computed gain, $g_c[n]$, is calculated as

$$g_c[n] = x_{sc}[n] - x_{dB}[n].$$

Gain Smoothing

$g_c[n]$ is smoothed using specified attack and release time properties:

$$g_s[n] = \begin{cases} \alpha_A g_s[n-1] + (1-\alpha_A) g_c[n], & g_c[n] > g_s[n-1] \\ \alpha_R g_s[n-1] + (1-\alpha_R) g_c[n], & g_c[n] \le g_s[n-1] \end{cases}$$

The attack time coefficient, α_A , is calculated as

$$\alpha_A = \exp\left(\frac{-\log(9)}{Fs \times T_A}\right).$$

The release time coefficient, α_R , is calculated as

$$\alpha_R = \exp\left(\frac{-\log(9)}{Fs \times T_R}\right).$$

T_A is the attack time period, specified by the `AttackTime` property. T_R is the release time period, specified by the `ReleaseTime` property. *Fs* is the input sampling rate, specified by the `SampleRate` property.

Calculate and Apply Make-up Gain

If `MakeUpGainMode` is set to the default `'Auto'`, the make-up gain is calculated as the negative of the computed gain for a 0 dB input,

$$M = -x_{sc}\big|_{x_{dB}=0}.$$

Given a steady-state input of 0 dB, this configuration achieves a steady-state output of 0 dB. The make-up gain is determined by the `Threshold`, `Ratio`, and `KneeWidth` properties. It does not depend on the input signal.

The make-up gain, M, is added to the smoothed gain, $g_s[n]$:

$$g_m[n] = g_s[n] + M$$

Calculate and Apply Linear Gain

The calculated gain in dB, $g_m[n]$, is translated to a linear domain:

$$g_{lin}[n] = 10^{\left(\frac{g_m[n]}{20}\right)}$$

The output of the dynamic range compressor is given as

$$y[n] = x[n] \times g_{lin}[n].$$

References

[1] Giannoulis, Dimitrios, Michael Massberg, and Joshua D. Reiss. "Digital Dynamic Range Compressor Design –– A Tutorial and Analysis." *Journal of Audio Engineering Society*. Vol. 60, Issue 6, 2012, pp. 399–408.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

“System Objects in MATLAB Code Generation” (MATLAB Coder)

See Also

Blocks
Compressor

System Objects
`expander` | `limiter` | `noiseGate`

Topics

“Dynamic Range Control”

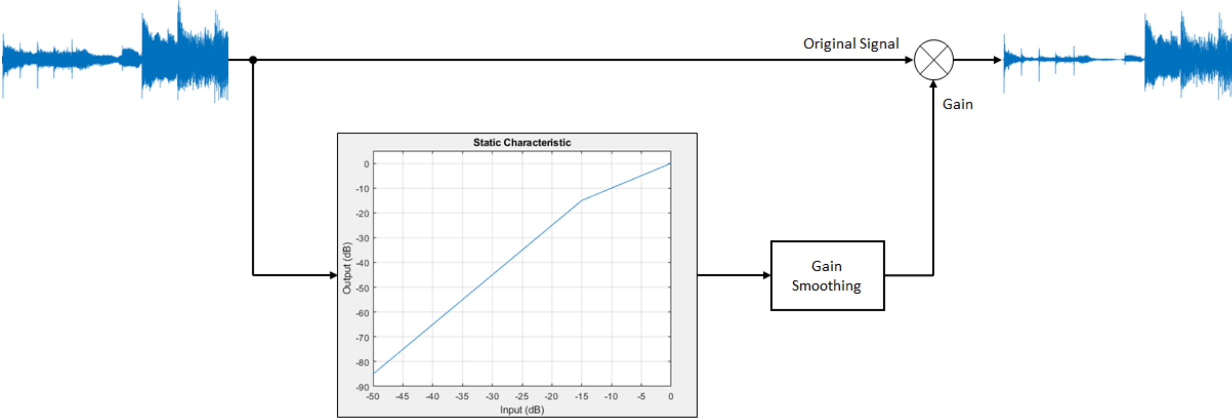
Introduced in R2016a

expander System object

Dynamic range expander

Description

The `expander` System object performs dynamic range expansion independently across each input channel. Dynamic range expansion attenuates the volume of quiet sounds below a given threshold. It uses specified attack, release, and hold times to achieve a smooth applied gain curve. Properties of the `expander` System object specify the type of dynamic range expansion.



To perform dynamic range expansion:

1. Create the expander object and set its properties.
2. Call the object with arguments, as if it were a function.

To learn more about how System objects work, see What Are System Objects? (MATLAB).

Creation

Syntax

```
dRE = expander
dRE = expander(thresholdValue)
dRE = expander(thresholdValue,ratioValue)
dRE = expander( ___ ,Name,Value)
```

Description

`dRE = expander` creates a System object, `dRE`, that performs dynamic range expansion independently across each input channel.

`dRE = expander(thresholdValue)` sets the Threshold property to `thresholdValue`.

`dRE = expander(thresholdValue,ratioValue)` sets the Ratio property to `ratioValue`.

`dRE = expander( ___ ,Name,Value)` sets each property `Name` to the specified `Value`. Unspecified properties have default values.

Example: `dRE = expander('AttackTime',0.01,'SampleRate',16000)` creates a System object, `dRE`, with a 0.01 second attack time and a 16 kHz sample rate.

Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see System Design in MATLAB Using System Objects (MATLAB).

`Threshold` — Operation threshold (dB)
–`10` (default) | real scalar

Operation threshold in dB, specified as a real scalar.

Operation threshold is the level below which gain is applied to the input signal.

Tunable: Yes

Data Types: `single` | `double`

`Ratio` — Expansion ratio

`5` (default) | real scalar

Expansion ratio, specified as a real scalar greater than or equal to 1.

Expansion ratio is the input/output ratio for signals that undershoot the operation threshold.

Assuming a hard knee characteristic and a steady-state input such that $x[n]$ dB < `thresholdValue`, the expansion ratio is defined as $R = \frac{(y[n]-T)}{(x[n]-T)}$.

- R is the expansion ratio.
- $y[n]$ is the output signal in dB.
- $x[n]$ is the input signal in dB.
- T is the threshold in dB.

Tunable: Yes

Data Types: `single` | `double`

`KneeWidth` — Knee width (dB)

`0` (default) | real scalar

Knee width in dB, specified as a real scalar greater than or equal to 0.

Knee width is the transition area in the expansion characteristic.

For soft knee characteristics, the transition area is defined by the relation

$$y = x + \frac{(1-R)\times\left(x - T - \frac{W}{2}\right)^2}{(2\times W)}$$

for the range $\left(2\times|x-T|\right)\leq W$.

- y is the output level in dB.
- x is the input level in dB.
- R is the expansion ratio.
- T is the threshold in dB.
- W is the knee width in dB.

Tunable: Yes

Data Types: `single` | `double`

`AttackTime` — Attack time (s)
`0.05` (default) | real scalar

Attack time in seconds, specified as a real scalar greater than or equal to 0.

Attack time is the time it takes the expander gain to rise from 10% to 90% of its final value when the input goes below the threshold.

Tunable: Yes

Data Types: `single` | `double`

`ReleaseTime` — Release time (s)
`0.2` (default) | real scalar

Release time in seconds, specified as a real scalar greater than or equal to 0.

Release time is the time it takes the expander gain to drop from 90% to 10% of its final value when the input goes above the threshold.

Tunable: Yes

Data Types: `single` | `double`

`HoldTime` — Hold time (s)
`0.05` (default) | real scalar

Hold time in seconds, specified as a real scalar greater than or equal to 0.

Hold time is the period in which the applied gain is held constant before it starts moving toward its steady-state value. Hold time begins when the input level crosses the operation threshold.

Tunable: Yes

Data Types: `single` | `double`

`SampleRate` — Input sample rate (Hz)
`44100` (default) | positive scalar

Input sample rate in Hz, specified as a positive scalar.

Tunable: Yes

Data Types: `single` | `double`

Usage

Note For versions earlier than R2016b, use the `step` function to run the System object™ algorithm. The arguments to `step` are the object you created, followed by the arguments shown in this section.

For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Syntax

```
audioOut = dRE(audioIn)
[audioOut,gain] = dRE(audioIn)
```

Description

`audioOut = dRE(audioIn)` performs dynamic range expansion on the input signal, `audioIn`, and returns the expanded signal, `audioOut`. The type of dynamic range expansion is specified by the algorithm and properties of the `expander` System object, `dRE`.

`[audioOut,gain] = dRE(audioIn)` also returns the applied gain, in dB, at each input sample.

Input Arguments

`audioIn` — Audio input to expander
matrix

Audio input to the expander, specified as a matrix. The columns of the matrix are treated as independent audio channels.

Data Types: `single` | `double`

Output Arguments

`audioOut` — Audio output from expander
matrix

Audio output from the expander, returned as a matrix the same size as `audioIn`.

Data Types: `single` | `double`

`gain` — Gain applied by expander (dB)
matrix

Gain applied by expander, returned as a matrix the same size as `audioIn`.

Data Types: `single` | `double`

Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

Specific to expander

visualize	Visualize static characteristic of dynamic range controller
createAudioPluginClass	Create audio plugin class that implements functionality of System object

MIDI

configureMIDI	Configure MIDI connections between audio object and MIDI controller
disconnectMIDI	Disconnect MIDI controls from audio object
getMIDIConnections	Get MIDI connections of audio object

Common to All System Objects

clone	Create duplicate System object
isLocked	Determine if System object is in use
release	Release resources and allow changes to System object property values and input characteristics
reset	Reset internal states of System object
step	Run System object algorithm

The `createAudioPluginClass` and `configureMIDI` functions map tunable properties of the `expander` System object to user-facing parameters:

Property	Range	Mapping	Unit
`Threshold`	[–140, 0]	linear	dB
`Ratio`	[1, 50]	linear	none
`KneeWidth`	[0, 20]	linear	dB
`AttackTime`	[0, 4]	linear	seconds
`ReleaseTime`	[0, 4]	linear	seconds
`HoldTime`	[0, 4]	linear	seconds

Examples

Expand Audio Signal

Use dynamic range expansion to attenuate background noise from an audio signal.

Set up the `dsp.AudioFileReader` and `audioDeviceWriter` System objects.

```
frameLength = 1024;
fileReader = dsp.AudioFileReader( ...
    'Filename','Counting-16-44p1-mono-15secs.wav', ...
```

```
    'SamplesPerFrame',frameLength);
deviceWriter = audioDeviceWriter( ...
    'SampleRate',fileReader.SampleRate);
```

Corrupt the audio signal with Gaussian noise. Play the audio.

```
while ~isDone(fileReader)
    x = fileReader();
    xCorrupted = x + (1e-2/4)*randn(frameLength,1);
    deviceWriter(xCorrupted);
end

release(fileReader)
```

Set up the expander with a threshold of -40 dB, a ratio of 10, an attack time of 0.01 seconds, a release time of 0.02 seconds, and a hold time of 0 seconds. Use the sample rate of your audio file reader.

```
dRE = expander(-40,10, ...
    'AttackTime',0.01, ...
    'ReleaseTime',0.02, ...
    'HoldTime',0, ...
    'SampleRate',fileReader.SampleRate);
```

Visualize the expansion static characteristic.

```
visualize(dRE)
```

Set up the scope to visualize the signal before and after dynamic range expansion.

```
scope = dsp.TimeScope( ...
    'SampleRate',fileReader.SampleRate, ...
    'TimeSpanOverrunAction','Scroll', ...
    'TimeSpan',16, ...
    'BufferLength',1.5e6, ...
    'YLimits',[-1 1], ...
    'ShowGrid',true, ...
    'ShowLegend',true, ...
    'Title','Corrupted vs. Expanded Audio');
```

Play the processed audio and visualize it on the scope.

```
while ~isDone(fileReader)
    x = fileReader();
    xCorrupted = x + (1e-2/4)*randn(frameLength,1);
```

```
    y = dRE(xCorrupted);
    deviceWriter(y);
    scope([xCorrupted,y])
end

release(fileReader)
release(dRE)
release(deviceWriter)
release(scope)
```

Apply Split-Band De-Essing

De-essing is the process of diminishing sibilant sounds in an audio signal. Sibilance refers to the *s*, *z*, and *sh* sounds in speech, which can be disproportionately emphasized during recording. *es* sounds fall under the category of unvoiced speech with all consonants and have a higher frequency than voiced speech. In this example, you apply split-band de-essing to a speech signal by separating the signal into high and low frequencies, applying an expander to diminish the sibilant frequencies, and then remixing the channels.

Create a `dsp.AudioFileReader` System object™ and an `audioDeviceWriter` System object to read from a sound file and write to an audio device. Listen to the unprocessed signal. Then release the file reader and device writer.

```
fileReader = dsp.AudioFileReader( ...
    fullfile(matlabroot,'examples','audio','Sibilance.wav'));
deviceWriter = audioDeviceWriter;

while ~isDone(fileReader)
    audioIn = fileReader();
    deviceWriter(audioIn);
end

release(deviceWriter)
release(fileReader)
```

Create an `expander` System object to de-ess the audio signal. Set the sample rate of the expander to the sample rate of the audio file. Create a two-band crossover filter with a crossover of 3000 Hz. Sibilance is usually found in this range. Set the crossover slope to 12. Plot the frequency response of the crossover filter to confirm your design visually.

```
dRExpander = expander( ...
    'Threshold',-50, ...
```

```
    'AttackTime', 0.05, ...
    'ReleaseTime',0.05, ...
    'HoldTime',0.005, ...
    'SampleRate',fileReader.SampleRate);

crossFilt = crossoverFilter( ...
    'NumCrossovers',1, ...
    'CrossoverFrequencies',3000, ...
    'CrossoverSlopes',12);
visualize(crossFilt)
```

Create a `dsp.TimeScope` System object to visualize the original and processed audio signals.

```
scope = dsp.TimeScope( ...
    'SampleRate',fileReader.SampleRate, ...
    'TimeSpanOverrunAction','Scroll', ...
    'TimeSpan',4, ...
    'BufferLength',44100*8, ...
    'YLimits',[-1 1], ...
    'ShowGrid',true, ...
    'ShowLegend',true, ...
    'ChannelNames',{'Original','Processed'});
```

In an audio stream loop:

1. Read in a frame of the audio file.
2. Split the audio signal into two bands.
3. Apply dynamic range expansion to the upper band.
4. Remix the channels.
5. Write the processed audio signal to your audio device for listening.
6. Visualize the processed and unprocessed signals on a time scope.

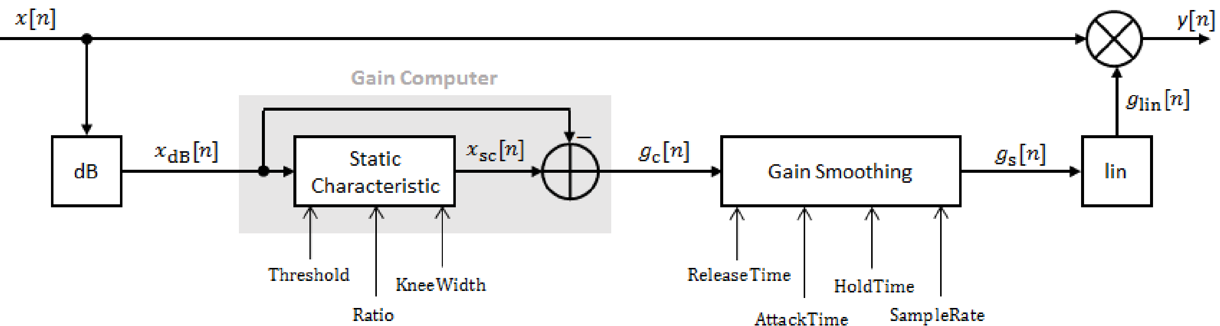
As a best practice, release your objects once done.

```
while ~isDone(fileReader)
    audioIn = fileReader();

    [band1,band2] = crossFilt(audioIn);

    band2processed = dRExpander(band2);
```

```
    procAudio  = band1 + band2processed;

    deviceWriter(procAudio);

    scope([audioIn procAudio]);
end

release(deviceWriter)
release(fileReader)
release(scope)
release(crossFilt)
release(dRExpander)
```

Algorithms

The `expander` System object processes a signal frame by frame and element by element.



Convert Input Signal to dB

The *N*-point signal, *x*[*n*], is converted to decibels:

$$x_{dB}[n] = 20 \times \log_{10}|x[n]|$$

Gain Computer

$x_{dB}[n]$ passes through the gain computer. The gain computer uses the static characteristic properties of the dynamic range expander to attenuate gain that is below the threshold.

If you specified a soft knee, the gain computer has the following static characteristic:

$$x_{sc}(x_{dB}) = \begin{cases} T + (x_{dB} - T) \times R & x_{dB} < \left(T - \frac{W}{2}\right) \\ x_{dB} + \frac{(1-R)\left(x_{dB} - T - \frac{W}{2}\right)^2}{2W} & \left(T - \frac{W}{2}\right) \le x_{dB} \le \left(T + \frac{W}{2}\right) \\ x_{dB} & x_{dB} > \left(T + \frac{W}{2}\right) \end{cases},$$

where T is the threshold, R is the ratio, and W is the knee width.

If you specified a hard knee, the gain computer has the following static characteristic:

$$x_{sc}(x_{dB}) = \begin{cases} T + (x_{dB} - T) \times R & x_{dB} < T \\ x_{dB} & x_{dB} \ge T \end{cases}$$

The computed gain, $g_c[n]$, is calculated as

$$g_c[n] = x_{sc}[n] - x_{dB}[n].$$

Gain Smoothing

$g_c[n]$ is smoothed using specified attack, release, and hold time properties:

$$g_s[n] = \begin{cases} \alpha_A g_s[n-1] + (1-\alpha_A) g_c[n] & (C_A > T_H) \ \& \ (g_c[n] > g_s[n-1]) \\ g_s[n-1] & C_A \le T_H \\ \alpha_R g_s[n-1] + (1-\alpha_R) g_c[n] & (C_R > T_H) \ \& \ (g_c[n] \le g_s[n-1]) \\ g_s[n-1] & C_R \le T_H \end{cases}$$

The attack time coefficient, α_A, is calculated as

$$\alpha_A = \exp\left(\frac{-\log(9)}{Fs \times T_A}\right).$$

The release time coefficient, α_R , is calculated as

$$\alpha_R = \exp\left(\frac{-\log(9)}{Fs \times T_R}\right).$$

T_A is the attack time period, specified by the `AttackTime` property. T_R is the release time period, specified by the `ReleaseTime` property. *Fs* is the input sampling rate, specified by the `SampleRate` property.

C_A and C_R are hold counters for attack and release, respectively. The limit, T_H , is determined by the `HoldTime` property.

Calculate and Apply Linear Gain

The smoothed gain in dB, $g_s[n]$, is translated to a linear domain:

$$g_{lin}[n] = 10^{\left(\frac{g_s[n]}{20}\right)}$$

The output of the dynamic range expander is given as

$$y[n] = x[n] \times g_{lin}[n].$$

References

[1] Giannoulis, Dimitrios, Michael Massberg, and Joshua D. Reiss. "Digital Dynamic Range Compressor Design –– A Tutorial and Analysis." *Journal of Audio Engineering Society*. Vol. 60, Issue 6, 2012, pp. 399–408.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

"System Objects in MATLAB Code Generation" (MATLAB Coder)

See Also

Blocks
Expander

System Objects
`compressor` | `limiter` | `noiseGate`

Topics
"Dynamic Range Control"

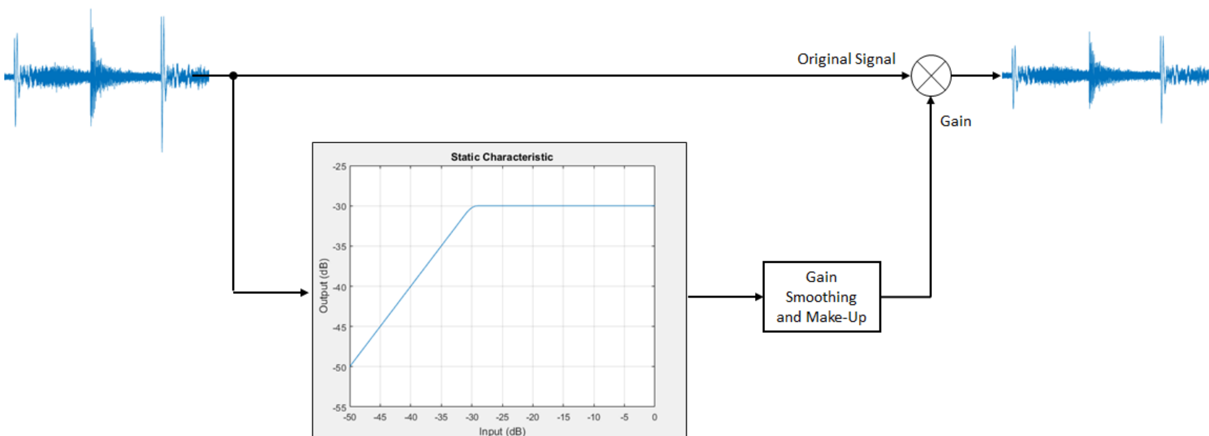
Introduced in R2016a

limiter System object

Dynamic range limiter

Description

The `limiter` System object performs brick-wall dynamic range limiting independently across each input channel. Dynamic range limiting suppresses the volume of loud sounds that cross a given threshold. It uses specified attack and release times to achieve a smooth applied gain curve. Properties of the `limiter` System object specify the type of dynamic range limiting.



To perform dynamic range limiting:

1 Create the limiter object and set its properties.
2 Call the object with arguments, as if it were a function.

To learn more about how System objects work, see What Are System Objects? (MATLAB).

Creation

Syntax

```
dRL = limiter
dRL = limiter(thresholdValue)
dRL = limiter( ___ ,Name,Value)
```

Description

`dRL = limiter` creates a System object, `dRL`, that performs brick-wall dynamic range limiting independently across each input channel.

`dRL = limiter(thresholdValue)` sets the Threshold property to `thresholdValue`.

`dRL = limiter( ___ ,Name,Value)` sets each property `Name` to the specified `Value`. Unspecified properties have default values.

Example: `dRL = limiter('AttackTime',0.01,'SampleRate',16000)` creates a System object, `dRL`, with a 10 ms attack time and a sample rate of 16 kHz.

Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see System Design in MATLAB Using System Objects (MATLAB).

`Threshold` — Operation threshold (dB)
–`10` (default) | real scalar

Operation threshold in dB, specified as a real scalar.

Operation threshold is the level above which gain is applied to the input signal.

Tunable: Yes

Data Types: `single` | `double`

`KneeWidth` — Knee width (dB)
`0` (default) | real scalar

Knee width in dB, specified as a real scalar greater than or equal to 0.

Knee width is the transition area in the limiter characteristic.

For soft knee characteristics, the transition area is defined by the relation

$$y = x - \frac{\left(x - T + \frac{W}{2}\right)^2}{(2 \times W)}$$

for the range $(2 \times |x - T|) \leq W$.

- y is the output level in dB.
- x is the input level in dB.
- T is the threshold in dB.
- W is the knee width in dB.

Tunable: Yes

Data Types: `single` | `double`

`AttackTime` — Attack time (s)
`0` (default) | real scalar

Attack time in seconds, specified as a real scalar greater than or equal to 0.

Attack time is the time it takes the limiter gain to rise from 10% to 90% of its final value when the input goes above the threshold.

Tunable: Yes

Data Types: `single` | `double`

ReleaseTime — Release time (s)
0.2 (default) | real scalar

Release time in seconds, specified as a real scalar greater than or equal to 0.

Release time is the time it takes the limiter gain to drop from 90% to 10% of its final value when the input goes below the threshold.

Tunable: Yes

Data Types: single | double

MakeUpGainMode — Make-up gain mode
'Property' (default) | 'Auto'

Make-up gain mode, specified as 'Auto' or 'Property'.

- 'Auto' –– Make-up gain is applied at the output of the dynamic range limiter such that a steady-state 0 dB input has a 0 dB output.
- 'Property' –– Make-up gain is set to the value specified in the MakeUpGain property.

Tunable: No

Data Types: char | string

MakeUpGain — Make-up gain (dB)
0 (default) | real scalar

Make-up gain in dB, specified as a real scalar.

Make-up gain compensates for gain lost during limiting. It is applied at the output of the dynamic range limiter.

Tunable: Yes

Dependencies

To enable this property, set MakeUpGainMode to 'Property'.

Data Types: single | double

SampleRate — Input sample rate (Hz)
44100 (default) | positive scalar

Input sample rate in Hz, specified as a positive scalar.

Tunable: Yes

Data Types: `single` | `double`

Usage

Note For versions earlier than R2016b, use the `step` function to run the System object™ algorithm. The arguments to `step` are the object you created, followed by the arguments shown in this section.

For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Syntax

```
audioOut = dRL(audioIn)
[audioOut,gain] = dRL(audioIn)
```

Description

`audioOut = dRL(audioIn)`performs dynamic range limiting on the input signal, `audioIn`, and returns the limited signal, `audioOut`. The type of dynamic range limiting is specified by the algorithm and properties of the `limiter` System object, `dRL`.

`[audioOut,gain] = dRL(audioIn)`also returns the applied gain, in dB, at each input sample.

Input Arguments

`audioIn` — Audio input to limiter
matrix

Audio input to the limiter, specified as a matrix. The columns of the matrix are treated as independent audio channels.

Data Types: `single` | `double`

Output Arguments

`audioOut` — Audio output from limiter
matrix

Audio output from the limiter, returned as a matrix the same size as `audioIn`.

Data Types: `single` | `double`

`gain` — Gain applied by limiter (dB)
matrix

Gain applied by the limiter, returned as a matrix the same size as `audioIn`.

Data Types: `single` | `double`

Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

Specific to `limiter`

visualize	Visualize static characteristic of dynamic range controller
createAudioPluginClass	Create audio plugin class that implements functionality of System object

MIDI

configureMIDI	Configure MIDI connections between audio object and MIDI controller
disconnectMIDI	Disconnect MIDI controls from audio object
getMIDIConnections	Get MIDI connections of audio object

Common to All System Objects

clone	Create duplicate System object
isLocked	Determine if System object is in use

release	Release resources and allow changes to System object property values and input characteristics
reset	Reset internal states of System object
step	Run System object algorithm

The `createAudioPluginClass` and `configureMIDI` functions map tunable properties of the `limiter` System object to user-facing parameters:

Property	Range	Mapping	Unit
`Threshold`	[–50, 0]	linear	dB
`KneeWidth`	[0, 20]	linear	dB
`AttackTime`	[0, 4]	linear	seconds
`ReleaseTime`	[0, 4]	linear	seconds
`MakeUpGain` (available when you set `MakeUpGainMode` to `'Property'`)	[–10, 24]	linear	dB

Examples

Limit Audio Signal

Use dynamic range limiting to suppress the volume of loud sounds.

Set up the `dsp.AudioFileReader` and `audioDeviceWriter` System objects™.

```
frameLength = 1024;
fileReader = dsp.AudioFileReader( ...
    'Filename','RockDrums-44p1-stereo-11secs.mp3', ...
    'SamplesPerFrame',frameLength);
deviceWriter = audioDeviceWriter( ...
    'SampleRate',fileReader.SampleRate);
```

Set up the `limiter` to have a threshold of -15 dB, an attack time of 0.005 seconds, and a release time of 0.1 seconds. Set make-up gain to 0 dB (default). To specify this value, set the make-up gain mode to `'Property'` but do not specify the `MakeUpGain` property. Use the sample rate of your audio file reader.

```
dRL = limiter(-15, ...
    'AttackTime',0.005, ...
    'ReleaseTime',0.1, ...
    'MakeUpGainMode','Property', ...
    'SampleRate',fileReader.SampleRate);
```

Visualize the static characteristic of the `limiter`.

```
visualize(dRL)
```

Set up a time scope to visualize the original signal and the limited signal.

```
scope = dsp.TimeScope( ...
    'SampleRate',fileReader.SampleRate, ...
    'TimeSpanOverrunAction','Scroll', ...
    'TimeSpan',1, ...
    'BufferLength',44100*4, ...
    'YLimits',[-1 1], ...
    'ShowGrid',true, ...
    'LayoutDimensions',[2,1], ...
    'NumInputPorts',2, ...
    'ShowLegend',true, ...
    'Title',['Original vs. Limited Audio (top)' ...
    ' and Limiter Gain in dB (bottom)']);
```

Play the processed audio and visualize it on the scope.

```
while ~isDone(fileReader)
    x = fileReader();
    [y,g] = dRL(x);
    deviceWriter(y);
    x1 = x(:,1);
    y1 = y(:,1);
    g1 = g(:,1);
    scope([x1,y1],g1);
end

release(fileReader)
release(dRL)
release(deviceWriter)
release(scope)
```

Compare Dynamic Range Limiter and Compressor

A dynamic range limiter is a special type of dynamic range compressor. In limiters, the level above an operational threshold is hard limited. In the simplest implementation of a limiter, the effect is equivalent to audio clipping. In compressors, the level above an operational threshold is lowered using a specified compression ratio. Using a compression ratio results in a smoother processed signal.

Compare Limiter and Compressor Applied to Sinusoid

Create a `limiter` System object™ and a `compressor` System object. Set the `AttackTime` and `ReleaseTime` properties of both objects to zero. Create an `audioOscillator` System object to generate a sinusoid with `Frequency` set to `5` and `Amplitude` set to `0.1`.

```
dRL = limiter('AttackTime',0,'ReleaseTime',0);
dRC = compressor('AttackTime',0,'ReleaseTime',0);

osc = audioOscillator('Frequency',5,'Amplitude',0.1);
```

Create a time scope to visualize the generated sinusoid and the processed sinusoid.

```
scope = dsp.TimeScope( ...
    'SampleRate',osc.SampleRate, ...
    'TimeSpan',2, ...
    'BufferLength',osc.SampleRate*4, ...
    'YLimits',[-1 1], ...
    'TimeSpanOverrunAction','Scroll', ...
    'ShowGrid',true, ...
    'LayoutDimensions',[2 1], ...
    'NumInputPorts',2, ...
    'Title', ...
    'Original Signal vs. Limited Signal (top) and Compressed Signal (bottom)');
```

In an audio stream loop, visualize the original sinusoid and the sinusoid processed by a limiter and a compressor. Increment the amplitude of the original sinusoid to illustrate the effect.

```
while osc.Amplitude < 0.75
    x = osc();

    xLimited    = dRL(x);
    xCompressed = dRC(x);
```

```
    scope([x xLimited],[x xCompressed]);

    osc.Amplitude = osc.Amplitude + 0.0002;
end
release(scope)
release(dRL)
release(dRC)
release(osc)
```

Compare Limiter and Compressor Applied to Audio Signal

Compare the effect of dynamic range limiters and compressors on a drum track. Create a `dsp.AudioFileReader` System object and a `audioDeviceWriter` System object to read audio from a file and write to your audio output device. To emphasize the effect of dynamic range control, set the operational threshold of the limiter and compressor to -20 dB.

```
dRL.Threshold = -20;
dRC.Threshold = -20;

fileReader = dsp.AudioFileReader('FunkyDrums-44p1-stereo-25secs.mp3');
deviceWriter = audioDeviceWriter('SampleRate',fileReader.SampleRate);
```

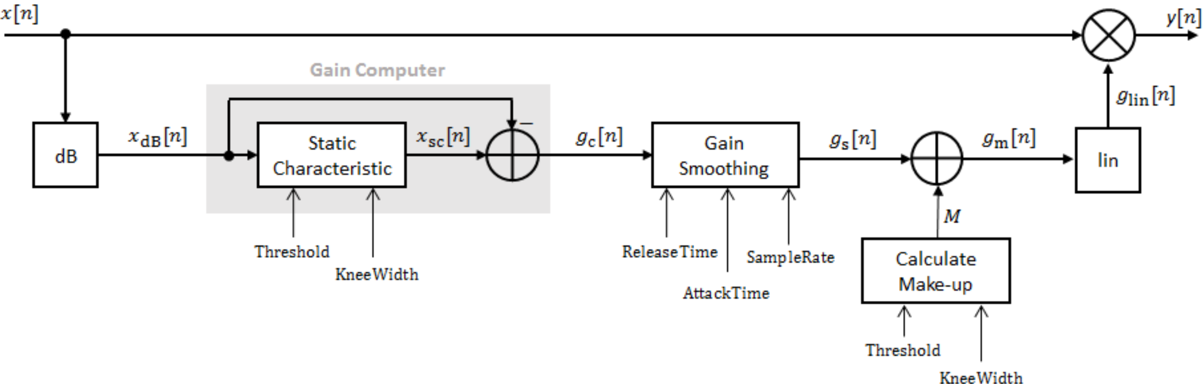
Read successive frames from an audio file in a loop. Listen to and compare the effect of dynamic range limiting and dynamic range compression on an audio signal.

```
numFrames = 300;

fprintf('Now playing original signal...\n')
for i = 1:numFrames
    x = fileReader();
    deviceWriter(x);
end
reset(fileReader);

fprintf('Now playing limited signal...\n')
for i = 1:numFrames
    x = fileReader();
    xLimited = dRL(x);
    deviceWriter(xLimited);
end
reset(fileReader);

fprintf('Now playing compressed signal...\n')
```

```
for i = 1:numFrames
    x = fileReader();
    xCompressed = dRC(x);
    deviceWriter(xCompressed);
end

release(fileReader)
release(deviceWriter)
release(dRC)
release(dRL)
```

Algorithms

The `limiter` System object processes a signal frame by frame and element by element.



Convert Input Signal to dB

The N-point signal, $x[n]$, is converted to decibels:

$$x_{\mathrm{dB}}[n] = 20 \times \log_{10} |x[n]|$$

Gain Computer

$x_{\mathrm{dB}}[n]$ passes through the gain computer. The gain computer uses the static characteristic properties of the dynamic range limiter to brick-wall gain that is above the threshold.

If you specified a soft knee, the gain computer has the following static characteristic:

$$x_{sc}(x_{dB}) = \begin{cases} x_{dB} & x_{dB} < \left(T - \frac{W}{2}\right) \\ x_{dB} - \frac{\left(x_{dB} - T + \frac{W}{2}\right)^2}{2W} & \left(T - \frac{W}{2}\right) \leq x_{dB} \leq \left(T + \frac{W}{2}\right) \\ T & x_{dB} > \left(T + \frac{W}{2}\right) \end{cases},$$

where T is the threshold and W is the knee width.

If you specified a hard knee, the gain computer has the following static characteristic:

$$x_{sc}(x_{dB}) = \begin{cases} x_{dB} & x_{dB} < T \\ T & x_{dB} \geq T \end{cases}$$

The computed gain, $g_c[n]$, is calculated as

$$g_c[n] = x_{sc}[n] - x_{dB}[n].$$

Gain Smoothing

$g_c[n]$ is smoothed using specified attack and release time:

$$g_s[n] = \begin{cases} \alpha_A g_s[n-1] + (1-\alpha_A) g_c[n], & g_c[n] > g_s[n-1] \\ \alpha_R g_s[n-1] + (1-\alpha_R) g_c[n], & g_c[n] \leq g_s[n-1] \end{cases}$$

The attack time coefficient, α_A , is calculated as

$$\alpha_A = \exp\left(\frac{-\log(9)}{Fs \times T_A}\right).$$

The release time coefficient, α_R , is calculated as

$$\alpha_R = \exp\left(\frac{-\log(9)}{Fs \times T_R}\right).$$

T_A is the attack time period, specified by the `AttackTime` property. T_R is the release time period, specified by the `ReleaseTime` property. Fs is the input sampling rate, specified by the `SampleRate` property.

Calculate and Apply Make-up Gain

If `MakeUpGainMode` is set to the default `'Auto'`, the make-up gain is calculated as the negative of the computed gain for a 0 dB input:

$$M = -x_{sc}\big|_{x_{dB}=0}$$

Given a steady-state input of 0 dB, this configuration achieves a steady-state output of 0 dB. The make-up gain is determined by the `Threshold` and `KneeWidth` properties. It does not depend on the input signal.

The make-up gain, M, is added to the smoothed gain, $g_s[n]$:

$$g_m[n] = g_s[n] + M$$

Calculate and Apply Linear Gain

The calculated gain in dB, $g_m[n]$, is translated to a linear domain:

$$g_{lin}[n] = 10^{\left(\frac{g_m[n]}{20}\right)}.$$

The output of the dynamic range limiter is given as

$$y[n] = x[n] \times g_{lin}[n].$$

References

[1] Giannoulis, Dimitrios, Michael Massberg, and Joshua D. Reiss. "Digital Dynamic Range Compressor Design –– A Tutorial and Analysis." *Journal of Audio Engineering Society*. Vol. 60, Issue 6, 2012, pp. 399–408.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

"System Objects in MATLAB Code Generation" (MATLAB Coder)

See Also

Blocks
Limiter

System Objects
`compressor` | `expander` | `noiseGate`

Topics

"Dynamic Range Control"

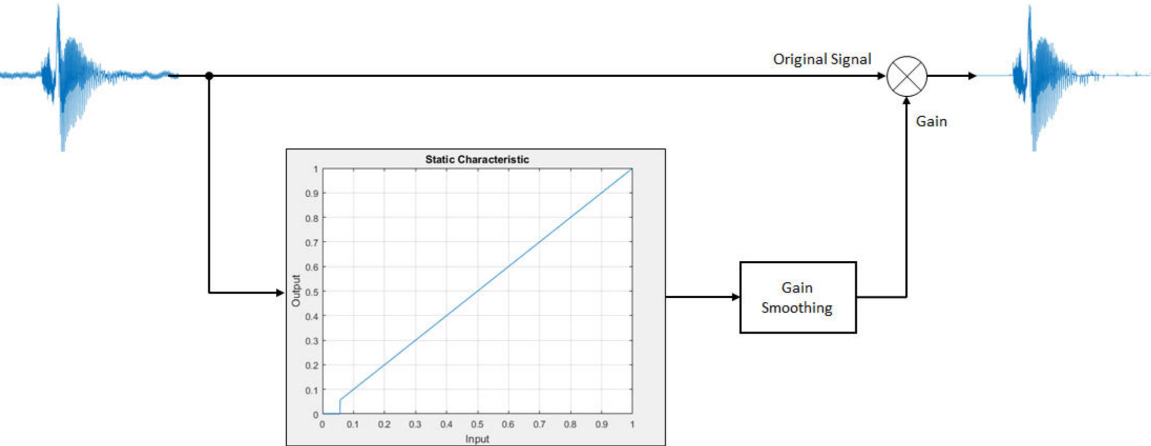
Introduced in R2016a

noiseGate System object

Dynamic range gate

Description

The `noiseGate` System object performs dynamic range gating independently across each input channel. Dynamic range gating suppresses signals below a given threshold. It uses specified attack, release, and hold times to achieve a smooth applied gain curve. Properties of the `noiseGate` System object specify the type of dynamic range gating.



To perform dynamic range gating:

1. Create the noiseGate object and set its properties.
2. Call the object with arguments, as if it were a function.

To learn more about how System objects work, see What Are System Objects? (MATLAB).

Creation

Syntax

```
dRG = noiseGate
dRG = noiseGate(thresholdValue)
dRG = noiseGate( ___ ,Name,Value)
```

Description

`dRG = noiseGate` creates a System object, `dRG`, that performs dynamic range gating independently across each input channel.

`dRG = noiseGate(thresholdValue)` sets the Threshold property to `thresholdValue`.

`dRG = noiseGate( ___ ,Name,Value)` sets each property `Name` to the specified `Value`. Unspecified properties have default values.

Example: `dRG = noiseGate('AttackTime',0.01,'SampleRate',16000)` creates a System object, `dRG`, with a 10 ms attack time and a 16 kHz sample rate.

Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see System Design in MATLAB Using System Objects (MATLAB).

`Threshold` — Operation threshold (dB)
–`10` (default) | real scalar

Operation threshold in dB, specified as a real scalar.

Operation threshold is the level below which gain is applied to the input signal.

Tunable: Yes

Data Types: `single` | `double`

`AttackTime` — Attack time (s)
`0.05` (default) | real scalar

Attack time in seconds, specified as a real scalar greater than or equal to 0.

Attack time is the time it takes the applied gain to rise from 10% to 90% of its final value when the input goes below the threshold.

Tunable: Yes

Data Types: `single` | `double`

`ReleaseTime` — Release time (s)
`0.02` (default) | real scalar

Release time in seconds, specified as a real scalar greater than or equal to 0.

Release time is the time it takes the applied gain to drop from 90% to 10% of its final value when the input goes above the threshold.

Tunable: Yes

Data Types: `single` | `double`

`HoldTime` — Hold time (s)
`0.05` (default) | real finite scalar

Hold time in seconds, specified as a real scalar greater than or equal to 0.

Hold time is the period in which the applied gain is held constant before it starts moving toward its steady-state value. Hold time begins when the input level crosses the operation threshold.

Tunable: Yes

Data Types: `single` | `double`

`SampleRate` — Input sample rate (Hz)
`44100` (default) | positive scalar

Input sample rate in Hz, specified as a positive scalar.

Tunable: Yes

Data Types: `single` | `double`

Usage

Note For versions earlier than R2016b, use the `step` function to run the System object™ algorithm. The arguments to `step` are the object you created, followed by the arguments shown in this section.

For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Syntax

```
audioOut = dRG(audioIn)
[audioOut,gain] = dRG(audioIn)
```

Description

`audioOut = dRG(audioIn)` performs dynamic range gating on the input signal, `audioIn`, and returns the gated signal, `audioOut`. The type of dynamic range gating is specified by the algorithm and properties of the `noiseGate` System object, `dRG`.

`[audioOut,gain] = dRG(audioIn)` also returns the applied gain, in dB, at each input sample.

Input Arguments

`audioIn` — Audio input to noise gate
matrix

Audio input to the noise gate, specified as a matrix. The columns of the matrix are treated as independent audio channels.

Data Types: `single` | `double`

Output Arguments

`audioOut` — Audio output from noise gate
matrix

Audio output from the noise gate, returned as a matrix the same size as `audioIn`.

Data Types: `single` | `double`

`gain` — Gain applied by noise gate (dB)
matrix

Gain applied by noise gate, returned as a matrix the same size as `audioIn`.

Data Types: `single` | `double`

Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

Specific to noiseGate

visualize	Visualize static characteristic of dynamic range controller
createAudioPluginClass	Create audio plugin class that implements functionality of System object

MIDI

configureMIDI	Configure MIDI connections between audio object and MIDI controller
disconnectMIDI	Disconnect MIDI controls from audio object
getMIDIConnections	Get MIDI connections of audio object

Common to All System Objects

clone	Create duplicate System object
isLocked	Determine if System object is in use

release	Release resources and allow changes to System object property values and input characteristics
reset	Reset internal states of System object
step	Run System object algorithm

The `createAudioPluginClass` and `configureMIDI` functions map tunable properties of the `noiseGate` System object to user-facing parameters:

Property	Range	Mapping	Unit
`Threshold`	[–140, 0]	linear	dB
`AttackTime`	[0, 4]	linear	seconds
`ReleaseTime`	[0, 4]	linear	seconds
`HoldTime`	[0, 4]	linear	seconds

Examples

Gate Audio Signal

Use dynamic range gating to attenuate background noise from an audio signal.

Set up the `dsp.AudioFileReader` and `audioDeviceWriter` System objects™.

```
frameLength = 1024;
fileReader = dsp.AudioFileReader( ...
    'Filename','Counting-16-44p1-mono-15secs.wav', ...
    'SamplesPerFrame',frameLength);
deviceWriter = audioDeviceWriter( ...
    'SampleRate',fileReader.SampleRate);
```

Corrupt the audio signal with Gaussian noise. Play the audio.

```
while ~isDone(fileReader)
    x = fileReader();
    xCorrupted = x + (1e-2/4)*randn(frameLength,1);
    deviceWriter(xCorrupted);
end

release(fileReader)
```

Set up a dynamic range gate with a threshold of -25 dB, an attack time of 0.01 seconds, a release time of 0.02 seconds, and a hold time of 0 seconds. Use the sample rate of your audio file reader.

```
gate = noiseGate(-25, ...
    'AttackTime',0.01, ...
    'ReleaseTime',0.02, ...
    'HoldTime',0, ...
    'SampleRate',fileReader.SampleRate);
```

Visualize the static characteristic of the gate.

```
visualize(gate)
```

Set up a time scope to visualize the signal before and after dynamic range gating.

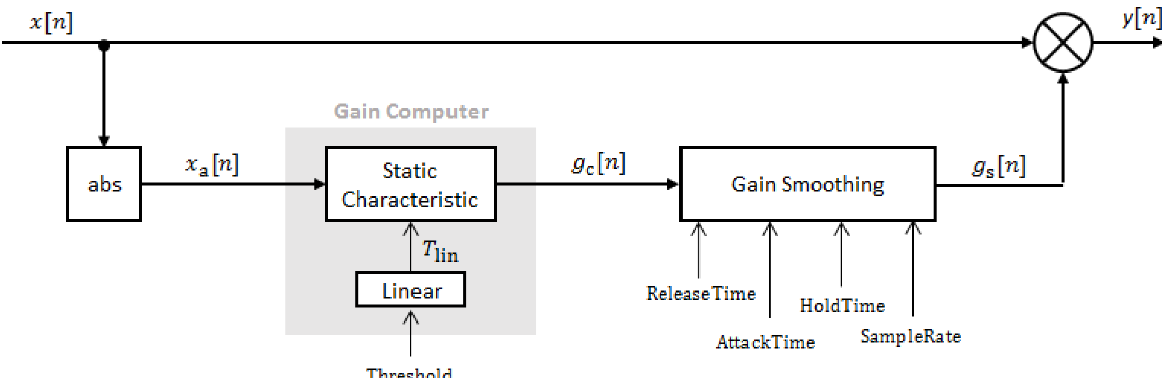
```
scope = dsp.TimeScope( ...
    'SampleRate',fileReader.SampleRate, ...
    'TimeSpanOverrunAction','Scroll', ...
    'TimeSpan',16, ...
    'BufferLength',1.5e6, ...
    'YLimits',[-1 1], ...
    'ShowGrid',true, ...
    'ShowLegend',true, ...
    'Title','Corrupted vs. Gated Audio');
```

Play the processed audio and visualize it on scope.

```
while ~isDone(fileReader)
    x = fileReader();
    xCorrupted = x + (1e-2/4)*randn(frameLength,1);
    y = gate(xCorrupted);
    deviceWriter(y);
    scope([xCorrupted,y]);
end

release(fileReader)
release(gate)
release(deviceWriter)
release(scope)
```

Algorithms

The `noiseGate` System object processes a signal frame by frame and element by element.



Convert Input Signal to Magnitude

The N-point signal, $x[n]$, is converted to magnitude:

$$x_a[n] = |x[n]|.$$

Gain Computer

$x_a[n]$ passes through the gain computer. The gain computer uses the static characteristic properties of the dynamic range gate to determine a brick-wall gain for signal below the threshold:

$$g_c(x_a) = \begin{cases} 0 & x_a < T_{lin} \\ 1 & x_a \geq T_{lin} \end{cases}.$$

T_{lin} is the threshold property converted to a linear domain:

$$T_{lin} = 10^{\left(T_{dB}/20\right)}.$$

Gain Smoothing

The computed gain, $g_c[n]$, is smoothed using specified attack, release, and hold time properties:

$$g_s[n]=\begin{cases}\alpha_A g_s[n-1]+(1-\alpha_A)g_c[n] & (C_A>T_H)\ \&\ (g_c[n]>g_s[n-1])\\ g_s[n-1] & C_A\le T_H\\ \alpha_R g_s[n-1]+(1-\alpha_R)g_c[n] & (C_R>T_H)\ \&\ (g_c[n]\le g_s[n-1])\\ g_s[n-1] & C_R\le T_H\end{cases}$$

The attack time coefficient, α_A , is calculated as

$$\alpha_A=\exp\left(\frac{-\log(9)}{Fs\times T_A}\right).$$

The release time coefficient, α_R , is calculated as

$$\alpha_R=\exp\left(\frac{-\log(9)}{Fs\times T_R}\right).$$

T_A is the attack time period, specified by the `AttackTime` property. T_R is the release time period, specified by the `ReleaseTime` property. *Fs* is the input sampling rate, specified by the `SampleRate` property.

C_A and C_R are hold counters for attack and release, respectively. The limit, T_H , is determined by the `HoldTime` property.

Apply Gain

The output of the dynamic range gate is given as

$$y[n]=x[n]\times g_s[n].$$

References

[1] Giannoulis, Dimitrios, Michael Massberg, and Joshua D. Reiss. "Digital Dynamic Range Compressor Design –– A Tutorial and Analysis." *Journal of Audio Engineering Society*. Vol. 60, Issue 6, 2012, pp. 399–408.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

"System Objects in MATLAB Code Generation" (MATLAB Coder)

See Also

Blocks

Noise Gate

System Objects

`compressor` | `expander` | `limiter`

Topics

"Dynamic Range Control"

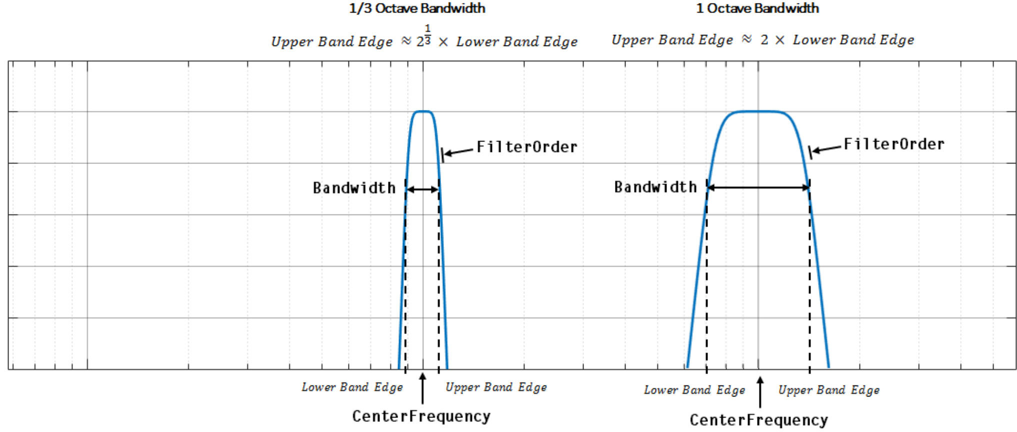
Introduced in R2016a

octaveFilter System object

Octave-band and fractional octave-band filter

Description

The `octaveFilter` System object performs octave-band or fractional octave-band filtering independently across each input channel. An octave-band is a frequency band where the highest frequency is twice the lowest frequency. Octave-band and fractional octave-band filters are commonly used to mimic how humans perceive loudness. Octave filters are best understood when viewed on a logarithmic scale, which models how the human ear weights the spectrum.



To perform octave-band or fractional octave-band filtering on your input:

1. Create the octaveFilter object and set its properties.
2. Call the object with arguments, as if it were a function.

To learn more about how System objects work, see What Are System Objects? (MATLAB).

Creation

Syntax

```
octFilt = octaveFilter
octFilt = octaveFilter(centerFreq)
octFilt = octaveFilter(centerFreq,bw)
octFilt = octaveFilter( ___ ,Name,Value)
```

Description

`octFilt = octaveFilter` creates a System object, `octFilt`, that performs octave-band filtering independently across each input channel.

`octFilt = octaveFilter(centerFreq)` sets the CenterFrequency property to `centerFreq`.

`octFilt = octaveFilter(centerFreq,bw)` sets the Bandwidth property to `bw`.

`octFilt = octaveFilter( ___ ,Name,Value)` sets each property `Name` to the specified `Value`. Unspecified properties have default values.

Example: `octFilt = octaveFilter(1000,'1/3 octave','SampleRate',96000)` creates a System object, `octFilt`, with a center frequency of 1000 Hz, a 1/3 octave filter bandwidth, and a sample rate of 96,000 Hz.

Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see System Design in MATLAB Using System Objects (MATLAB).

`FilterOrder` — Order of octave filter
`6` (default) | even integer

Order of the octave filter, specified as an even integer.

Tunable: No

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`CenterFrequency` — Center frequency of octave filter (Hz)
`1000` (default) | positive scalar

Center frequency of the octave filter in Hz, specified as a positive scalar.

- The maximum center frequency is the value that causes the upper band edge to be equal to the Nyquist frequency, *Fs*/2. Frequencies above this value are saturated.
- The minimum center frequency is the value that causes the lower band edge to be equal to 1 Hz. Frequencies below this value are quantized to the value that corresponds to lower band edge equal to 1 Hz.

Tunable: Yes

Data Types: `single` | `double`

`Bandwidth` — Filter bandwidth (octaves)
`'1 octave'` (default) | `'2/3 octave'` | `'1/2 octave'` | `'1/3 octave'` | `'1/6 octave'` | `'1/12 octave'` | `'1/24 octave'` | `'1/48 octave'`

Filter bandwidth in octaves, specified as `'1 octave'`, `'2/3 octave'`, `'1/2 octave'`, `'1/3 octave'`, `'1/6 octave'`, `'1/12 octave'`, `'1/24 octave'`, or `'1/48 octave'`.

Tunable: Yes

Data Types: `char` | `string`

`Oversample` — Oversample toggle
`false` (default) | `true`

Oversample toggle, specified as `false` or `true`.

- `false` -- The octave filter runs at the input sample rate.
- `true` -- The octave filter runs at two times the input sample rate. Oversampling minimizes the frequency warping effects introduced by the bilinear transformation. An FIR halfband interpolator implements oversampling before octave filtering. A halfband

decimator reduces the sample rate back to the input sampling rate after octave filtering.

Tunable: No

Data Types: `logical`

`SampleRate` — Input sample rate (Hz)
`44100` (default) | positive scalar

Input sample rate in Hz, specified as a positive scalar.

Tunable: Yes

Data Types: `single` | `double`

Usage

Syntax

```
audioOut = octFilt(audioIn)
```

Description

`audioOut = octFilt(audioIn)` applies octave-band filtering to the input signal, `audioIn`, and returns the filtered signal, `audioOut`. The type of filtering is specified by the algorithm and properties of the `octaveFilter` System object, `octFilt`.

Input Arguments

`audioIn` — Audio input to octave filter
matrix

Audio input to the octave filter, specified as a matrix. The columns of the matrix are treated as independent audio channels.

Data Types: `single` | `double`

Output Arguments

audioOut — Audio output from octave filter
matrix

Audio output from the octave filter, returned as a matrix the same size as `audioIn`.

Data Types: `single` | `double`

Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

Specific to octaveFilter

createAudioPluginClass	Create audio plugin class that implements functionality of System object
visualize	Visualize and validate filter response
isStandardCompliant	Verify octave filter design is ANSI S1.11-2004 compliant
getFilter	Return biquad filter object with design parameters set
getANSICenterFrequencies	Get the list of valid ANSI S1.11-2004 center frequencies

MIDI

configureMIDI	Configure MIDI connections between audio object and MIDI controller
disconnectMIDI	Disconnect MIDI controls from audio object
getMIDIConnections	Get MIDI connections of audio object

Common to All System Objects

clone	Create duplicate System object
isLocked	Determine if System object is in use
release	Release resources and allow changes to System object property values and input characteristics
reset	Reset internal states of System object
step	Run System object algorithm

The `createAudioPluginClass` and `configureMIDI` functions map tunable properties of the `octaveFilter` System object to user-facing parameters:

Property	Range	Mapping	Units
`CenterFrequency`	[3, 22000]	log	Hz
`Bandwidth`	`'1 octave'`, `'2/3 octave'`, `'1/2 octave'`, `'1/3 octave'`, `'1/6 octave'`, `'1/12 octave'`, `'1/24 octave'`, or `'1/48 octave'`	Your MIDI controller range is discretized into seven levels, corresponding to the seven `Bandwidth` choices.	--

Examples

Perform Fractional Octave-Band Filtering

Use the `octaveFilter` System object™ to design a 1/3 octave-band filter centered at 1000 Hz. Process an audio signal using your octave filter design.

Create a `dsp.AudioFileReader` System object.

```
samplesPerFrame = 1024;
reader = dsp.AudioFileReader('Filename', ...
    'RockGuitar-16-44p1-stereo-72secs.wav', ...
    'SamplesPerFrame',samplesPerFrame, ...
    'PlayCount',Inf);
```

Create an `octaveFilter` System object. Use the sample rate of the reader as the sample rate of the octave filter.

```
centerFreq = 1000;
bw = '1/3 octave';
Fs = reader.SampleRate;

octFilt = octaveFilter(centerFreq,bw,'SampleRate',Fs);
```

Visualize the filter response and verify that it fits within the class 0 mask of the ANSI S1.11-2004 standard.

```
visualize(octFilt,'class 0')
```

Create a spectrum analyzer to visualize the original audio signal and the audio signal after octave-band filtering.

```
scope = dsp.SpectrumAnalyzer( ...
    'SampleRate',Fs, ...
    'PlotAsTwoSidedSpectrum',false, ...
    'FrequencyScale','Log', ...
    'FrequencyResolutionMethod','WindowLength', ...
    'WindowLength',samplesPerFrame, ...
    'Title','Octave-Band Filtering', ...
    'ShowLegend',true, ...
    'ChannelNames',{'Original signal','Filtered signal'});
```

Process the audio signal in an audio stream loop. Visualize the filtered audio and the original audio. As a best practice, release the System objects when complete.

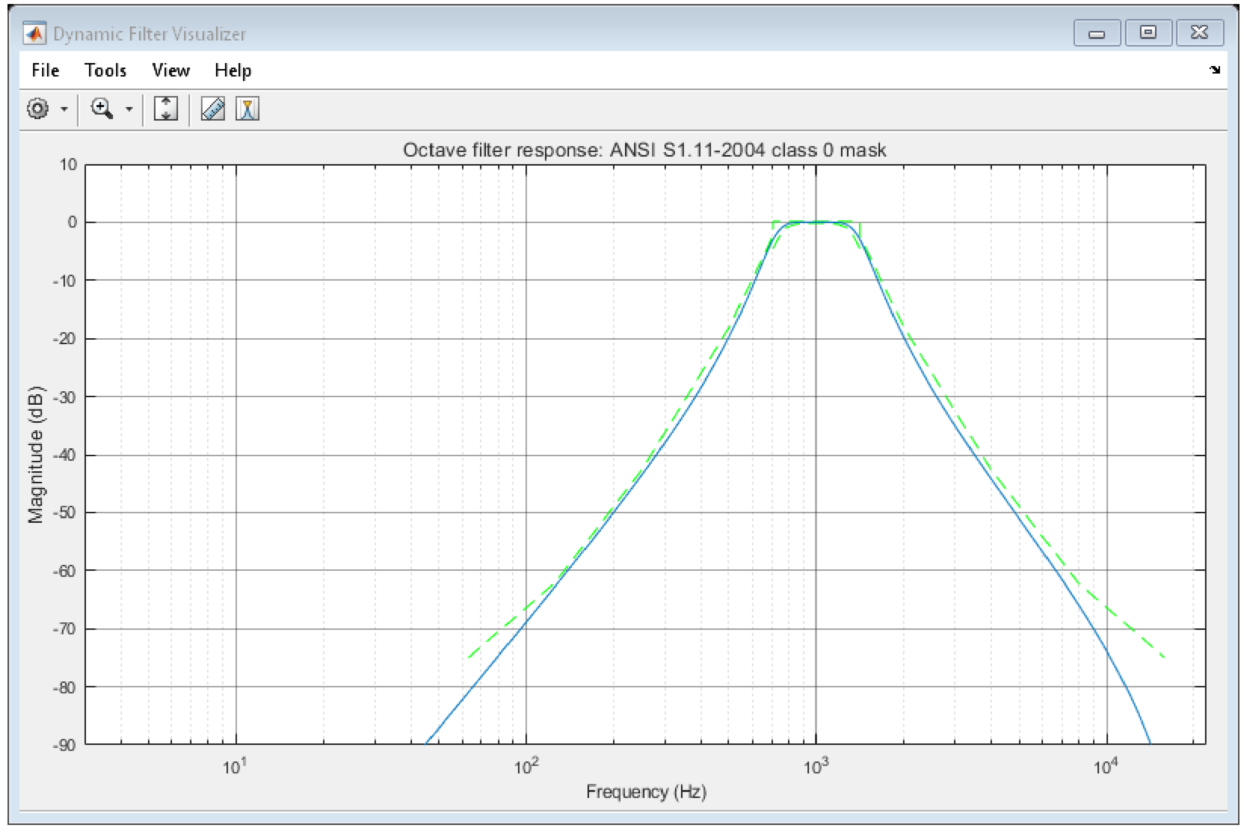
```
tic;
while toc < 20
    x = reader();
    y = octFilt(x);
    scope([x(:,1),y(:,1)])
end

release(octFilt)
release(scope)
release(reader)
```

Create Octave-Band Filter Bank

Create an octave-band filter bank that conforms to ANSI S1.11-2004. Pass white noise through the filter bank and inspect the resulting power in each band.

Create an octave filter with default settings. Visualize the filter design and verify that it conforms to ANSI S1.11-2004 for class 0.

```
octFilt = octaveFilter;
visualize(octFilt,'class 0')
```



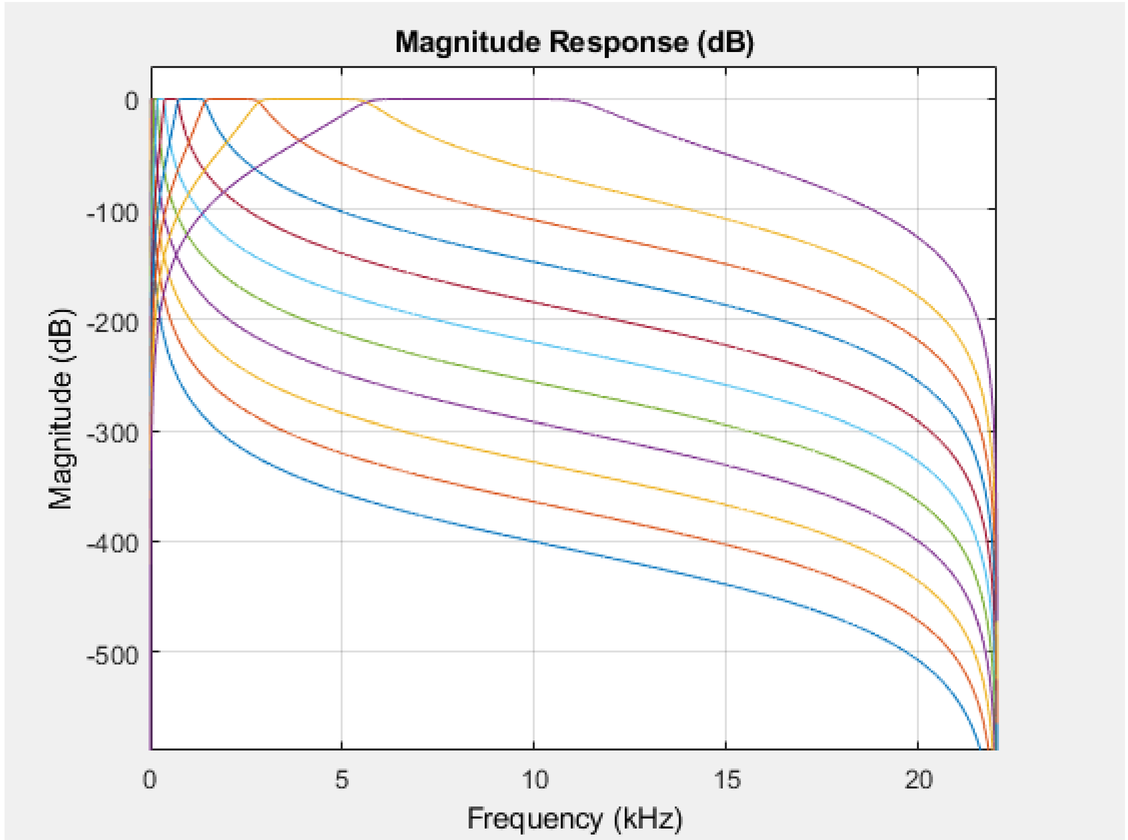
Get a vector of valid center frequencies, given the center frequency of `octFilt`. Create an octave filter bank using the valid center frequencies.

```
centerFrequencies = getANSICenterFrequencies(octFilt);
for i = 1:11
    octaveFilterBank{i} = octaveFilter(centerFrequencies(i),'FilterOrder',12);
end
```

Use `getFilter` to return biquad filter objects for each filter in your octave filter bank. Visualize the octave filter bank with a linear frequency scale.

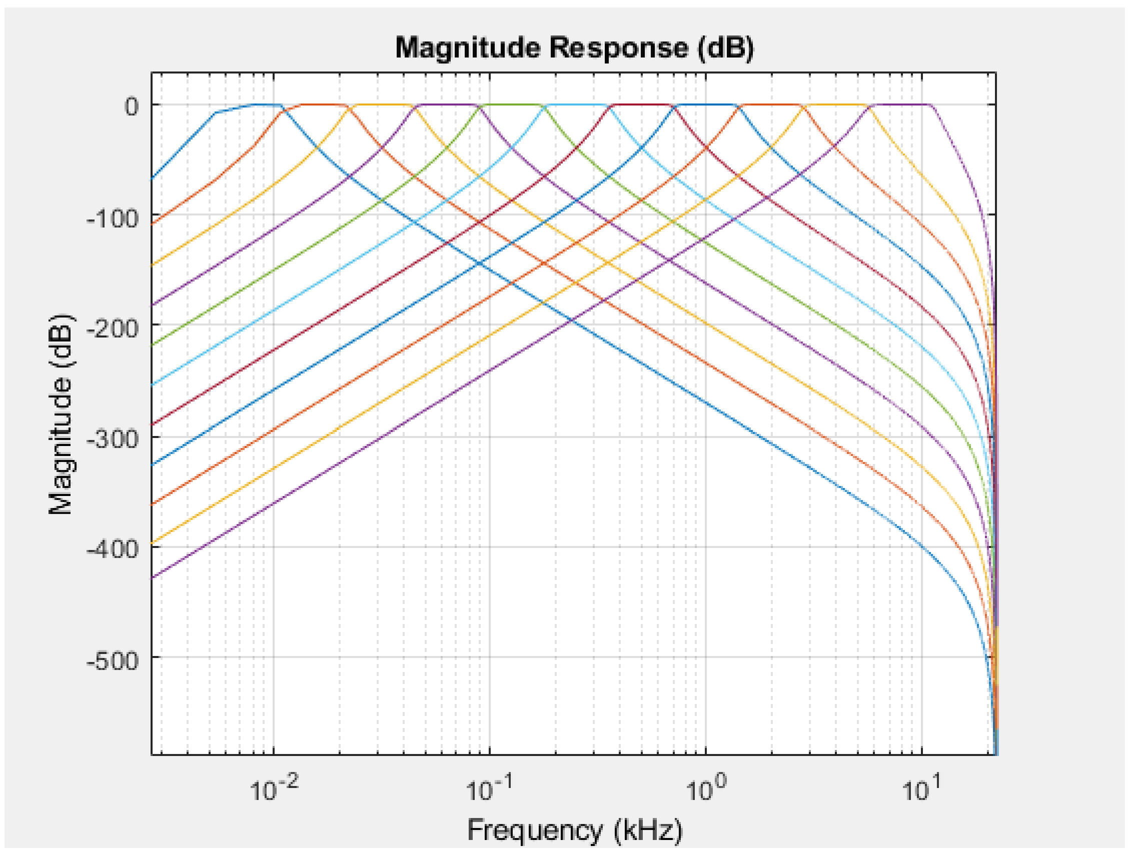
```
plotter = fvtool(getFilter(octaveFilterBank{1}), ...
    getFilter(octaveFilterBank{2}), ...
    getFilter(octaveFilterBank{3}), ...
    getFilter(octaveFilterBank{4}), ...
```

```
getFilter(octaveFilterBank{5}), ...
getFilter(octaveFilterBank{6}), ...
getFilter(octaveFilterBank{7}), ...
getFilter(octaveFilterBank{8}), ...
getFilter(octaveFilterBank{9}), ...
getFilter(octaveFilterBank{10}), ...
getFilter(octaveFilterBank{11}), ...
'Fs',octaveFilterBank{1}.SampleRate);
```



Visualize the octave filter bank with a logarithmic frequency scale. The logarithmic frequency scale makes the center frequencies appear evenly distributed.

```
set(plotter,'FrequencyScale','Log')
```



Create a white noise signal. By definition, white noise has a flat power spectral density.

```
whiteNoiseGenerator = dsp.ColoredNoise(0,1024);
whiteNoise = whiteNoiseGenerator();
```

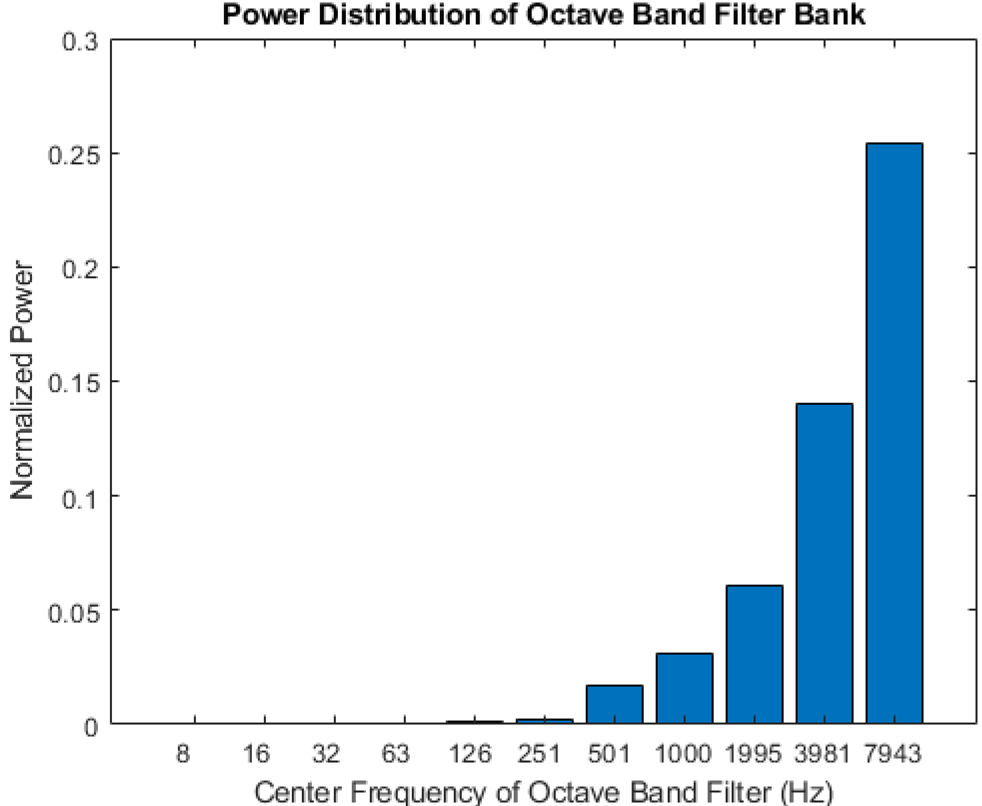
Pass the white noise signal through the octave-band filter bank.

```
for i = 1:11
    filteredWhiteNoise(:,i) = octaveFilterBank{i}(whiteNoise);
end
```

Calculate and plot the power in each octave.

```
for i = 1:11
    powerPerBand(i) = bandpower(filteredWhiteNoise(:,i));
```

```
end

bar(powerPerBand)
title('Power Distribution of Octave Band Filter Bank')
set(gca,'XTickLabel',{round(centerFrequencies)})
xlabel('Center Frequency of Octave Band Filter (Hz)')
ylabel('Normalized Power')
```



The band power increases by a factor of approximately two because the octave bandwidth increases by a factor of two. The power distribution of an octave filter bank mimics how higher frequencies are perceived louder in white noise. You can use octave filter banks to weight a spectrum for perceived loudness.

Effect of Center Frequency on Octave-Band Filtering

Process a speech signal using different octave bands from an octave-band filter bank.

Design a 1/2 octave filter with an estimated center frequency of 800 Hz. Use `isStandardCompliant` to find the nearest compliant center frequency.

```
octFilt = octaveFilter(800,'1/2 octave');
[complianceStatus,suggestedCenterFrequency] = isStandardCompliant(octFilt,'class 0')
```

```
complianceStatus =

  logical

   0


suggestedCenterFrequency =

  841.3951
```

Change the center frequency of the `octFilt` object to the suggested center frequency returned by `isStandardCompliant`. Get a list of valid ANSI S1.11-2004 center frequencies, given your specified `octFilt` center frequency.

```
octFilt.CenterFrequency = suggestedCenterFrequency;
Fo = getANSICenterFrequencies(octFilt);
```

Create an audio file reader and audio device writer.

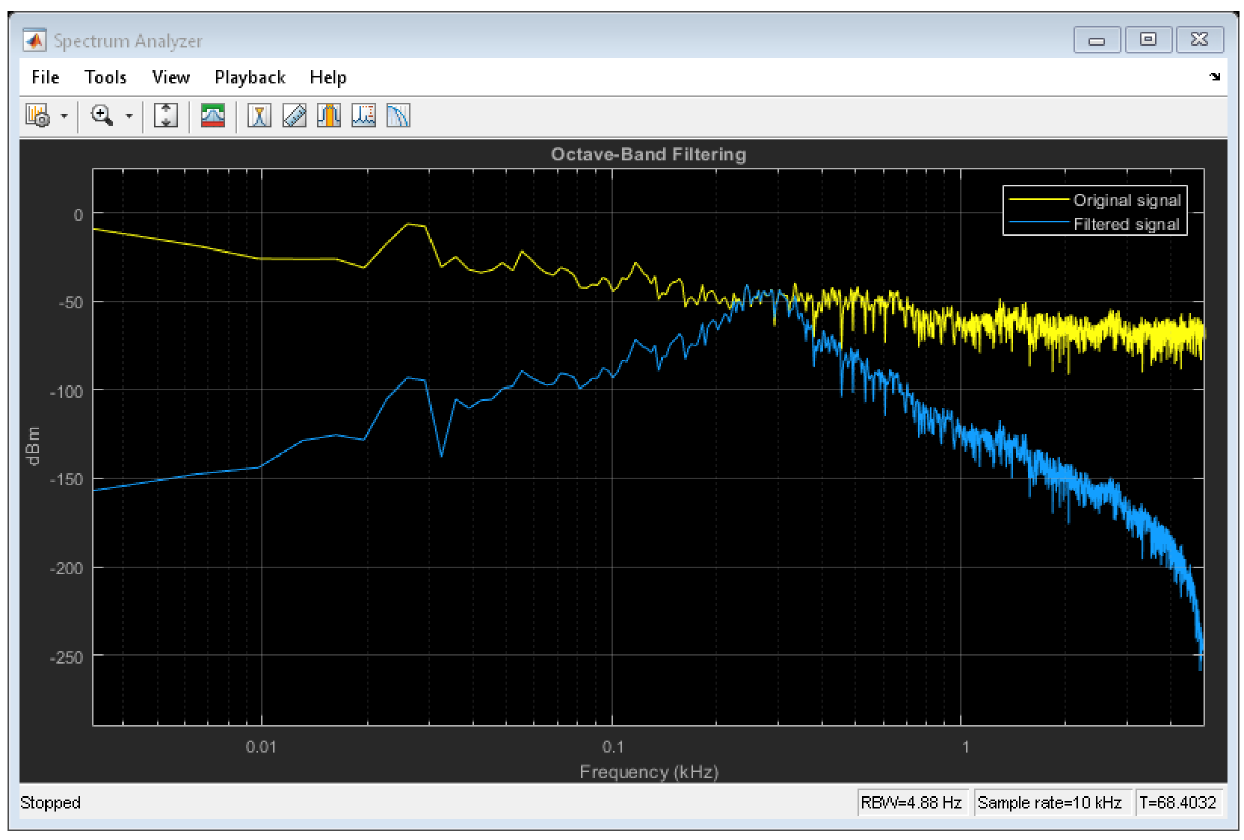
```
fileReader = dsp.AudioFileReader('Counting-16-44p1-mono-15secs.wav');
deviceWriter = audioDeviceWriter('SampleRate',fileReader.SampleRate);
```

Create a scope to visualize the filtered and unfiltered signals.

```
scope = dsp.SpectrumAnalyzer(...
    'PlotAsTwoSidedSpectrum',false,...
    'FrequencyScale','Log',...
    'Title','Octave-Band Filtering',...
    'ShowLegend',true,...
    'ChannelNames',{'Original signal','Filtered signal'});
```

In an audio stream loop, process the audio signal using your octave-band filter. Vary the center frequency to hear the effect. As a best practice, release your objects after processing.

```
index = 12;
octFilt.CenterFrequency = Fo(index);
count = 1;
while ~isDone(fileReader)
    x = fileReader();
    y = octFilt(x);
    scope([x,y])
    deviceWriter(y);

    if mod(count,100)==0
        octFilt.CenterFrequency = Fo(index);
        index = index+1;
    end
    count = count+1;
end

release(scope)
release(deviceWriter)
release(fileReader)
```



Remove Noise from Tone Scale

Remove additive noise from an audio tone scale using an `octaveFilter` System object™.

Create `audioOscillator` and `audioDeviceWriter` System objects with default properties. Create an `octaveFilter` System object with the center frequency set to 100 Hz.

```
osc = audioOscillator;
deviceWriter = audioDeviceWriter;
octFilt = octaveFilter(100);
```

In an audio stream loop, listen to a tone created by your audio oscillator. The tone contains additive Gaussian noise.

```
for i = 1:400
    x = osc();
    x1 = x + 0.1*randn(512,1);
    deviceWriter(x1);
    if rem(i,100)==0
        osc.Frequency = osc.Frequency*2;
    end
end
```

Create a spectrum analyzer to view your filtered and unfiltered signals.

```
scope = dsp.SpectrumAnalyzer( ...
    'PlotAsTwoSidedSpectrum',false, ...
    'FrequencyScale','Log', ...
    'FrequencyResolutionMethod','WindowLength', ...
    'Title','Octave-Band Filtering', ...
    'ShowLegend',true, ...
    'SpectralAverages',10, ...
    'ChannelNames',{'Original signal','Filtered signal'});
```

Reset the frequency of your audio oscillator to its default, 100 Hz.

```
osc.Frequency = 100;
```

In an audio stream loop, filter the corrupted tone using your octave-band filter. When the tone changes frequency in the loop, change the center frequency of your octave filter to match. As a best practice, release your audio device once done.

```
for i = 1:400
    x = osc();
    x1 = x + 0.1*randn(512,1);
    x2 = octFilt(x1);
    deviceWriter(x2);
    if rem(i,100)==0
        osc.Frequency = osc.Frequency*2;
        octFilt.CenterFrequency = octFilt.CenterFrequency*2;
    end
    scope([x1,x2])
end

release(deviceWriter)
```

Design Compliant High-Frequency Filters

Design a sixth-order 1/3 octave filter with a sample rate of 96 kHz.

```
octFilt = octaveFilter('FilterOrder',6, ...
    'Bandwidth','1/3 octave', ...
    'SampleRate',96e3);
```

Get the center frequencies defined by the ANSI S1.11-2004 standard. The center frequencies defined by the standard depend on the `Bandwidth` and `SampleRate` properties.

```
centerFrequencies = getANSICenterFrequencies(octFilt)
```

Set the center frequency of the octave filter to 19.953 kHz and visualize the response with a 'class 0' compliance mask.

```
octFilt.CenterFrequency = centerFrequencies(38);
visualize(octFilt,'class 0')
```

The red mask on the plot defines the bounds for the magnitude response of the filter. The magnitude response of this filter goes above the upper bound of the compliance mask around 6.6 kHz. One way to counter this is to increase the filter order so that the filter's rolloff is steeper.

To bring the octave filter design into compliance, set the octave filter order to 8.

```
octFilt.FilterOrder = 8;
```

Another option to bring the octave filter design into compliance is to set the `Overample` property to `true`. This designs and runs the filter at twice the specified `SampleRate` to reduce the effects of the bilinear transformation during the design stage.

```
octFilt.FilterOrder = 6;
octFilt.Oversample = true;
```

Design Compliant Low-Frequency Filters

Design a sixth-order 2/3 octave filter with a 96 kHz sample rate.

```
octFilt = octaveFilter('FilterOrder',6, ...
    'Bandwidth','2/3 octave', ...
    'SampleRate',96e3);
```

Get the center frequencies defined by the ANSI S1.11-2004 standard. The center frequencies defined by the standard depend on the `Bandwidth` and `SampleRate` properties.

```
centerFrequencies = getANSICenterFrequencies(octFilt)
```

Set the center frequency of the octave filter to ~6 Hz and visualize the response with a 'class 0' compliance mask.

```
octFilt.CenterFrequency = centerFrequencies(2);
visualize(octFilt,'class 0')
```

The red mask on the plot defines the bounds for the magnitude response of the filter. The magnitude response of this filter goes below the lower bound of the compliance mask between 5.5 and 7.5 Hz.

Low-frequency filters in an octave filter bank have very low normalized center frequencies, and the filters designed for them have poles that are almost on the unit circle. To make this filter ANSI compliant, it has to be designed and operated at a lower sample rate.

To bring the octave filter design into compliance, set the sample rate to 48 kHz.

```
octFilt.SampleRate = 48e3;
```

Definitions

Band Edge

A band edge frequency refers to the lower or upper edge of the passband of a bandpass filter.

Center Frequency of Octave Filter

The center frequency of an octave filter is the geometric mean of the lower and upper band edge frequencies.

Algorithms

Octave Bandwidth to Band Edge Conversion

The `octaveFilter` System object uses the specified center frequency and filter bandwidth in octaves to determine the normalized band edges [2].

The object computes the upper and lower band edge frequencies:

$$f_{\text{pa}} = f_{\text{c}} \times G^{-1/2b}$$

$$f_{\text{pb}} = f_{\text{c}} \times G^{1/2b}$$

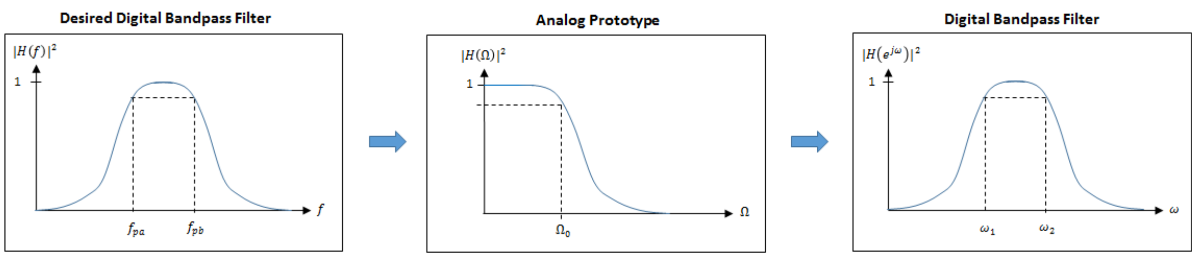
- f_{c} is the normalized center frequency specified by the `CenterFrequency` property.
- b is the octave bandwidth specified by the `Bandwidth` property. For example, if `Bandwidth` is specified as `'1/3 octave'`, the value of b is 3.
- G is a conversion constant:

$$G = 10^{3/10}.$$

Digital Filter Design

The `octaveFilter` System object implements a higher-order digital bandpass filter design method specified in [1].

In this design method, a desired digital bandpass filter maps to a Butterworth lowpass analog prototype, which is then mapped back to a digital bandpass filter:



1. The analog Butterworth filter is expressed as a cascade of second-order sections:

   $$H(s) = H_1(s)H_2(s)\cdots H_{2N}(s)\ ,$$

   where:

   $$H_i(s) = \frac{1}{1 - 2\frac{s}{\Omega_0}\cos\theta_i + \frac{s^2}{\Omega_0^2}}, \quad i = 1,2,\ldots,2N$$

   $$\theta_i = \frac{\pi}{2N}\left(N - 1 + 2i\right), \quad i = 1,2,..,2N$$

   N is the filter order specified by the `FilterOrder` property.
2. The analog Butterworth filter is mapped to a digital filter using a bandpass version of the bilinear transformation:

   $$s = \frac{1 - cz^{-1} + z^{-2}}{1 - z^{-2}},$$

   where

   $$c = \frac{\sin\left(\omega_{\text{pa}} + \omega_{\text{pb}}\right)}{\sin\omega_{\text{pa}} + \sin\omega_{\text{pb}}}.$$

   This mapping results in the following substitution:

   $$\Omega_0 = \frac{c - \cos\omega_{\text{pb}}}{\sin\omega_{\text{pb}}}.$$
3. The analog prototype is evaluated:

$$H_i(z) = \left. \frac{1}{1 - 2\frac{s}{\Omega_0}\cos\theta_i + \frac{s^2}{\Omega_0^2}} \right|_{s = \frac{1 - 2cz^{-1} + z^{-2}}{1 - z^{-2}}}$$

Because s is second-order in z, the bandpass version of the bilinear transformation is fourth-order in z.

References

[1] Orfanidis, Sophocles J. *Introduction to Signal Processing*. Englewood Cliffs, NJ: Prentice Hall, 2010.

[2] Acoustical Society of America. *American National Standard Specification for Octave-Band and Fractional-Octave-Band Analog and Digital Filters*. ANSI S1.11-2004. Melville, NY: Acoustical Society of America, 2009.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

“System Objects in MATLAB Code Generation” (MATLAB Coder)

See Also

Blocks
Octave Filter

System Objects
`dsp.BiquadFilter` | `multibandParametricEQ` | `weightingFilter`

Topics

"Octave-Band and Fractional Octave-Band Filters"

Introduced in R2016b

getANSICenterFrequencies

Get the list of valid ANSI S1.11-2004 center frequencies

Syntax

`centerFrequencies = getANSICenterFrequencies(octFilt)`

Description

`centerFrequencies = getANSICenterFrequencies(octFilt)` returns a vector of valid center frequencies as specified by the ANSI S1.11-2004 standard.

Examples

Get ANSI Center Frequencies

Create an object of the `octaveFilter` System object™. Call `getANSICenterFrequencies` to get a list of valid center frequencies.

```
octFilt = octaveFilter;
centerFrequencies = getANSICenterFrequencies(octFilt)
```

Input Arguments

`octFilt` — Object of `octaveFilter`
object

Object of the `octaveFilter` System object.

Output Arguments

`centerFrequencies` — Center frequencies
vector

Center frequencies specified by the ANSI S1.11-2004 standard, returned as a vector.

The range for computing valid center frequencies is 3 Hz to (Fs/2) Hz, where the `SampleRate` property of your octave filter defines Fs.

Data Types: `single` | `double`

See Also

Blocks
Octave Filter

Topics
"Octave-Band and Fractional Octave-Band Filters"

Introduced in R2016b

isStandardCompliant

Verify octave filter design is ANSI S1.11-2004 compliant

Syntax

```
complianceStatus = isStandardCompliant(octFilt,classType)
[complianceStatus,centerFreq] = isStandardCompliant(octFilt,
classType)
```

Description

`complianceStatus = isStandardCompliant(octFilt,classType)` returns a logical scalar, `complianceStatus`, indicating whether the `complianceStatus` filter design is compliant with the ANSI S1.11-2004 standard for `classType`.

The mask used to determine compliance is centered on the nearest ANSI-compliant center frequency that ensures the center frequency of the object falls between the upper and lower band edges of the mask.

`[complianceStatus,centerFreq] = isStandardCompliant(octFilt, classType)` also returns the ANSI-compliant center frequency used to create the mask.

Examples

Verify Standard Compliance

Create an object of the `octaveFilter` System object™. Call `isStandardCompliant`, specifying the compliance class type to check as the second argument.

```
octFilt = octaveFilter;
complianceStatus = isStandardCompliant(octFilt,'class 2')
```

Get ANSI-Compliant Center Frequency

Create an object of the octaveFilter System object. Check the compliance to class 0 status of your object, and get the center frequency used to create the compliance mask.

```
octFilt = octaveFilter('CenterFrequency',1266);
[compliant, centerFreq] = isStandardCompliant(octFilt,'class 0')
```

Input Arguments

`octFilt` — Object of `octaveFilter`
object

Object of the `octaveFilter` System object.

`classType` — Compliance class type
`'class 0'` | `'class 1'` | `'class 2'`

Compliance class type to verify, specified as `'class 0'`, `'class 1` or `'class 2'`.

Data Types: `char`

Output Arguments

`complianceStatus` — Compliance status of filter design
scalar

Compliance status of filter design, returned as a logical scalar. The compliance status indicates whether the `octFilt` filter design is compliant with the ANSI S1.11-2004 standard for `classType`.

If your octave filter is noncompliant, try any of the following:

- Set the center frequency to one of the values returned by `getANSICenterFrequencies`
- Increase filter order
- Increase sample rate

Data Types: `logical`

`centerFreq` — Center frequency of mask
scalar

Center frequency used to create the compliance mask, returned as a scalar.

Data Types: `single` | `double`

See Also

Blocks
Octave Filter

System Objects
`dsp.BiquadFilter` | `multibandParametricEQ` | `weightingFilter`

Topics
"Octave-Band and Fractional Octave-Band Filters"

Introduced in R2016b

visualize

Visualize and validate filter response

Syntax

```
visualize(octFilt)
visualize(octFilt,N)
visualize(___,mType)
```

Description

`visualize(octFilt)` plots the magnitude response of the octave-band filter, `octFilt`. The plot is updated automatically when properties of the object change.

`visualize(octFilt,N)` uses an `N`-point FFT to calculate the magnitude response.

`visualize(___,mType)` creates a mask based on the class of filter specified by `mType`, using either of the previous syntaxes. Specify `mType` as `'class 0'`, `'class 1'`, or `'class 2'`. The mask attenuation limits are defined in the ANSI S1.11-2004 standard. The mask center frequency is the ANSI standard center frequency, with band edge frequencies on either side of the `CenterFrequency` set in `octFilt`.

- If the mask is green, the design is compliant with the ANSI S1.11-2004 standard.
- If the mask is red, the design breaks compliance.

Examples

Plot Octave Filter Magnitude Response

Create an object of the `octaveFilter` System object™ and then plot the magnitude response of the filter.

```
octFilt = octaveFilter;
visualize(octFilt)
```

Specify Number of Frequency Bins

Create an object of the `octaveFilter` System object™. Plot a 5096-point frequency representation.

```
octFilt = octaveFilter;
visualize(octFilt,5096)
```

Visualize Standard-Compliance Mask

Create an object of the `octaveFilter` System object™. Visualize the class 1 compliance of the filter design.

```
octFilt = octaveFilter;
visualize(octFilt,'class 1')
```

Input Arguments

`octFilt` — Object of `octaveFilter`
object

Object of the `octaveFilter` System object.

`N` — Number of DFT bins
`2048` | positive scalar

Number of DFT bins in frequency-domain representation, specified as a positive scalar. The default is `2048`.

Data Types: `single` | `double`

`mType` — Type of mask
`'class 0'` | `'class 1'` | `'class 2'`

Type of mask, specified as `'class 0'`, `'class 1`, or `'class 2'`.

The mask attenuation limits are defined in the ANSI S1.11-2004 standard. The mask center frequency is the ANSI standard center frequency, with band edge frequencies on either side of the `CenterFrequency` set in `octFilt`.

- If the mask is green, the design is compliant with the ANSI S1.11-2004 standard.
- If the mask is red, the design breaks compliance.

Data Types: `char`

See Also

Topics

"Octave-Band and Fractional Octave-Band Filters"

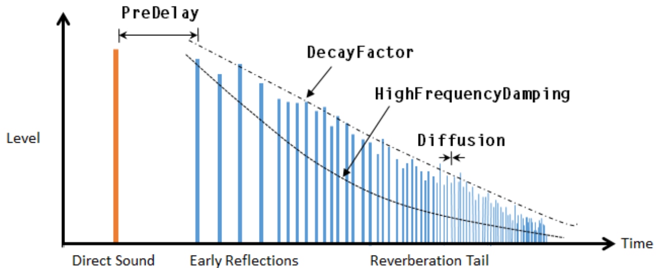
Introduced in R2016b

reverberator System object

Add reverberation to audio signal

Description

The `reverberator` System object adds reverberation to mono or stereo audio signals.



To add reverberation to your input:

1. Create the reverberator object and set its properties.
2. Call the object with arguments, as if it were a function.

To learn more about how System objects work, see What Are System Objects? (MATLAB).

Creation

Syntax

```
reverb = reverberator
reverb = reverberator(Name,Value)
```

Description

`reverb = reverberator` creates a System object, `reverb`, that adds artificial reverberation to an audio signal.

`reverb = reverberator(Name,Value)` sets each property `Name` to the specified `Value`. Unspecified properties have default values.

Example: `reverb = reverberator('PreDelay',0.5,'WetDryMix',1)` creates a System object, `reverb`, with a 0.5 second pre-delay and a wet-to-dry mix ratio of one.

Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see System Design in MATLAB Using System Objects (MATLAB).

`PreDelay` — Pre-delay for reverberation (s)
`0` (default) | real positive scalar

Pre-delay for reverberation in seconds, specified as a real scalar in the range [0, 1].

Pre-delay for reverberation is the time between hearing direct sound and the first early reflection. The value of `PreDelay` is proportional to the size of the room being modeled.

Tunable: Yes

Data Types: `single` | `double`

`HighCutFrequency` — Lowpass filter cutoff (Hz)
`20000` (default) | real positive scalar

Lowpass filter cutoff in Hz, specified as a real positive scalar in the range 0 to

$$\left(\frac{SampleRate}{2}\right).$$

Lowpass filter cutoff is the –3 dB cutoff frequency for the single-pole lowpass filter at the front of the reverberator structure. It prevents the application of reverberation to high-frequency components of the input.

Tunable: Yes

Data Types: `single` | `double`

`Diffusion` — Density of reverb tail

`0.5` (default) | real scalar

Density of reverb tail, specified as a real positive scalar in the range [0, 1].

`Diffusion` is proportional to the rate at which the reverb tail builds in density. Increasing `Diffusion` pushes the reflections closer together, thickening the sound. Reducing `Diffusion` creates more discrete echoes.

Tunable: Yes

Data Types: `single` | `double`

`DecayFactor` — Decay factor of reverb tail

`0.5` (default) | real scalar

Decay factor of reverb tail, specified as a real positive scalar in the range [0, 1].

`DecayFactor` is proportional to the time it takes for reflections to run out of energy. To model a large room, use a long reverb tail (low decay factor). To model a small room, use a short reverb tail (high decay factor).

Tunable: Yes

Data Types: `single` | `double`

`HighFrequencyDamping` — High-frequency damping

`0.0005` (default) | real scalar

High-frequency damping, specified as a real positive scalar in the range [0, 1].

`HighFrequencyDamping` is proportional to the attenuation of high frequencies in the reverberation output. Setting `HighFrequencyDamping` to a large value makes high-frequency reflections decay faster than low-frequency reflections.

Tunable: Yes

Data Types: `single` | `double`

`WetDryMix` — Wet-dry mix
`0.3` (default) | real scalar

Wet-dry mix, specified as a real positive scalar in the range [0, 1].

Wet-dry mix is the ratio of wet (reverberated) to dry (original) signal that your `reverberator` System object outputs.

Tunable: Yes

Data Types: `single` | `double`

`SampleRate` — Input sample rate (Hz)
`44100` (default) | positive scalar

Input sample rate in Hz, specified as a positive scalar.

Tunable: Yes

Data Types: `single` | `double`

Usage

Note For versions earlier than R2016b, use the `step` function to run the System object™ algorithm. The arguments to `step` are the object you created, followed by the arguments shown in this section.

For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Syntax

```
audioOut = reverb(audioIn)
```

Description

`audioOut = reverb(audioIn)` adds reverberation to the input signal, `audioIn`, and returns the mixed signal, `audioOut`. The type of reverberation is specified by the algorithm and properties of the reverberator System object, `reverb`.

Input Arguments

`audioIn` — Audio input to reverberator
column vector | *N*-by-2 matrix

Audio input to the reverberator, specified as a column vector or two-column matrix. The columns of the matrix are treated as independent audio channels.

Data Types: `single` | `double`

Output Arguments

`audioOut` — Audio output from reverberator
N-by-2 matrix (default)

Audio output from the reverberator, returned as a two-column matrix.

Data Types: `single` | `double`

Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

Specific to reverberator

createAudioPluginClass	Create audio plugin class that implements functionality of System object

MIDI

configureMIDI	Configure MIDI connections between audio object and MIDI controller
disconnectMIDI	Disconnect MIDI controls from audio object
getMIDIConnections	Get MIDI connections of audio object

Common to All System Objects

clone	Create duplicate System object
isLocked	Determine if System object is in use
release	Release resources and allow changes to System object property values and input characteristics
reset	Reset internal states of System object
step	Run System object algorithm

The `createAudioPluginClass` and `configureMIDI` functions map tunable properties of the `compressor` System object to user-facing parameters:

Property	Range	Mapping	Unit
`PreDelay`	[0, 1]	linear	s
`HighCutFrequency`	[20, 20000]	log	Hz
`Diffusion`	[0, 1]	linear	none
`DecayFactor`	[0, 1]	linear	none
`HighFrequencyDamping`	[0, 1]	linear	none
`WetDryMix`	[0, 1]	linear	none

Examples

Add Reverberation to Audio Signal

Use the `reverberator` System object™ to add artificial reverberation to an audio signal read from a file.

Create the `dsp.AudioFileReader` and `audioDeviceWriter` System objects. Use the sample rate of the reader as the sample rate of the writer.

```
fileReader = dsp.AudioFileReader( ...
    'FunkyDrums-44p1-stereo-25secs.mp3', ...
    'SamplesPerFrame',1024);
deviceWriter = audioDeviceWriter( ...
    'SampleRate',fileReader.SampleRate);
```

Play 10 seconds of the audio signal through your device.

```
tic
while toc < 10
    audio = fileReader();
    deviceWriter(audio);
end
release(fileReader)
```

Construct a `reverberator` System object with default settings.

```
reverb = reverberator
```

Construct a time scope to visualize the original audio signal and the audio signal with added artificial reverberation.

```
scope = dsp.TimeScope( ...
    'SampleRate',fileReader.SampleRate, ...
    'TimeSpanOverrunAction','Scroll', ...
    'TimeSpan',10, ...
    'BufferLength',1.5e6, ...
    'YLimits',[-1,1], ...
    'ShowGrid',true, ...
    'ShowLegend',true, ...
    'Title','Audio with Reverberation vs. Original');
```

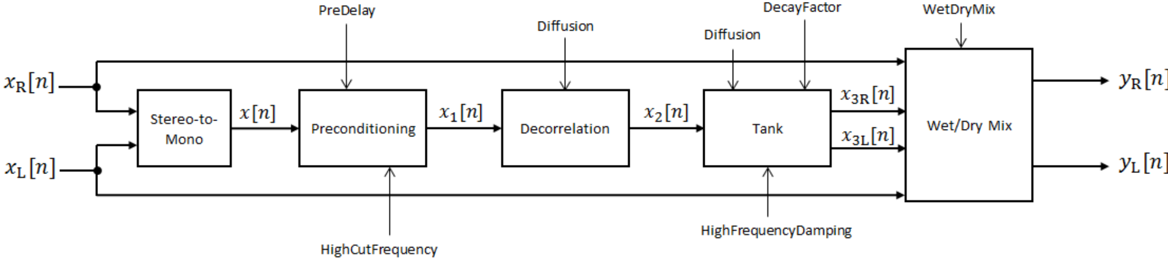
Play the audio signal with artificial reverberation. Visualize the audio with reverberation and the original audio.

```
while ~isDone(fileReader)
    audio = fileReader();
    audioWithReverb = reverb(audio);
    deviceWriter(audioWithReverb);
    scope([audioWithReverb(:,1),audio(:,1)])
end

release(fileReader)
release(deviceWriter)
```

Algorithms

The algorithm to add reverberation follows the plate-class reverberation topology described in [1] and is based on a 29,761 Hz sample rate.

The algorithm has five stages.



The description for the algorithm that follows is for a stereo input. A mono input is a simplified case.

Stereo-to-Mono

A stereo signal is converted to a mono signal: $x[n] = 0.5 \times \left(x_R[n] + x_L[n]\right)$.

Preconditioning

A delay followed by a lowpass filter preconditions the mono signal.

- The pre-delay output is determined as $x_p[n] = x[n-k]$, where the `PreDelay` property determines the value of k.
- The signal is fed through a single-pole lowpass filter with transfer function

$$LP(z) = \frac{1-\alpha}{1-\alpha z^{-1}},$$

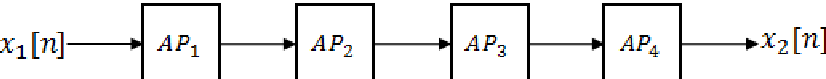
where

$$\alpha = \exp\left(-2\pi \times \frac{f_c}{f_s}\right).$$

- f_c is the cutoff frequency specified by the `HighCutFrequency` property.
- f_s is the sampling frequency specified by the `SampleRate` property.

Decorrelation

The signal is decorrelated by passing through a series of four allpass filters.



The allpass filters are of the form

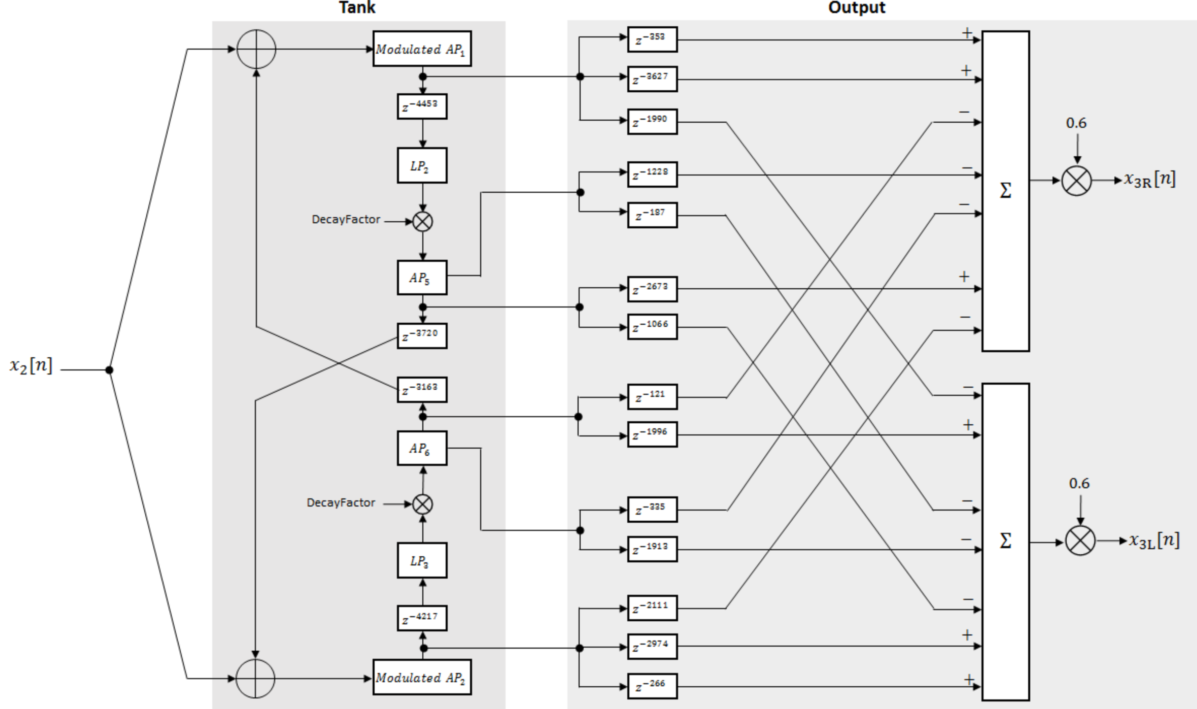
$$AP(z) = \frac{\beta + z^{-k}}{1 + \beta z^{-k}},$$

where β is the coefficient specified by the `Diffusion` property and k is the delay as follows:

- For AP_1, $k = 142$.
- For AP_2, $k = 107$.
- For AP_3, $k = 379$.
- For AP_4, $k = 277$.

Tank

The signal is fed into the tank, where it circulates to simulate the decay of a reverberation tail.



The following description tracks the signal as it progresses through the top of the tank. The signal progression through the bottom of the tank follows the same pattern, with different delay specifications.

1. The new signal enters the top of the tank and is added to the circulated signal from the bottom of the tank.
2. The signal passes through a modulated allpass filter:

$$Modulated\ AP_1(z) = \frac{-\beta + z^{-k}}{1 - \beta z^{-k}}$$

- β is the coefficient specified by the `Diffusion` property.

- k is the variable delay specified by a 1 Hz sinusoid with `amplitude = (8/29761)` * `SampleRate`. To account for fractional delay resulting from the modulating k, allpass interpolation is used [2].

3. The signal is delayed again, and then passes through a lowpass filter:

$$LP_2(z) = \frac{1-\varphi}{1-\varphi z^{-1}}$$

   - φ is the coefficient specified by the `HighFrequencyDamping` property.

4. The signal is multiplied by a gain specified by the `DecayFactor` property. The signal then passes through an allpass filter:

$$AP_5(z) = \frac{\beta + z^{-k}}{1 + \beta z^{-k}}.$$

   - β is the coefficient specified by the `Diffusion` property.
   - k is set to `1800` for the top of the tank and `2656` for the bottom of the tank.

5. The signal is delayed again and then circulated to the bottom half of the tank for the next iteration.

A similar pattern is executed in parallel for the bottom half of the tank. The output of the tank is calculated as the signed sum of delay lines picked off at various points from the tank. The summed output is multiplied by `0.6`.

Wet/Dry Mix

The wet (processed) signal is then added to the dry (original) signal:

$$y_R[n] = (1-\kappa)x_R[n] + \kappa x_{3R}[n],$$

$$y_L[n] = (1-\kappa)x_L[n] + \kappa x_{3L}[n],$$

where the `WetDryMix` property determines κ.

References

[1] Dattorro, Jon. "Effect Design, Part 1: Reverberator and Other Filters." *Journal of the Audio Engineering Society*. Vol. 45, Issue 9, 1997, pp. 660–684.

[2] Dattorro, Jon. "Effect Design, Part 2: Delay-Line Modulation and Chorus." *Journal of the Audio Engineering Society*. Vol. 45, Issue 10, 1997, pp. 764–788.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

“System Objects in MATLAB Code Generation” (MATLAB Coder)

See Also

Blocks
Reverberator

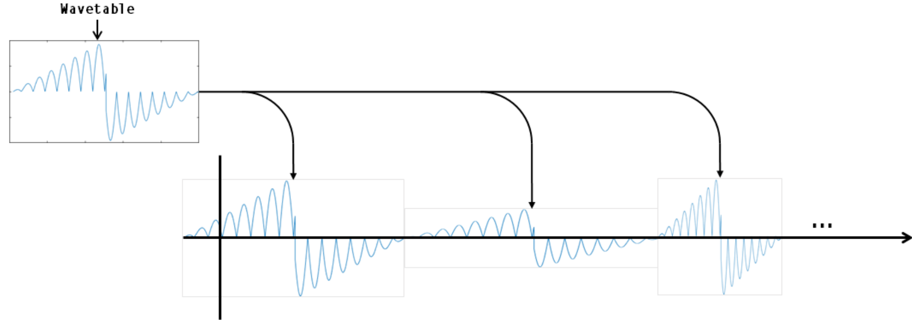
Introduced in R2016a

wavetableSynthesizer System object

Generate periodic signal from single-cycle waveforms

Description

The `wavetableSynthesizer` System object generates a periodic signal with tunable properties. The periodic signal is defined by a single-cycle waveform cached as the `Wavetable` property of your `wavetableSynthesizer` object.



To generate a periodic signal:

1. Create the wavetableSynthesizer object and set its properties.
2. Call the object with arguments, as if it were a function.

To learn more about how System objects work, see What Are System Objects? (MATLAB).

Creation

Syntax

```
waveSynth = wavetableSynthesizer
waveSynth = wavetableSynthesizer(wavetableValue)
```

```
waveSynth = wavetableSynthesizer(wavetableValue,frequencyValue)
waveSynth = wavetableSynthesizer( ___ ,Name,Value)
```

Description

`waveSynth = wavetableSynthesizer` creates a wavetable synthesizer System object, `waveSynth`, with default property values.

`waveSynth = wavetableSynthesizer(wavetableValue)` sets the Wavetable property to `wavetableValue`.

`waveSynth = wavetableSynthesizer(wavetableValue,frequencyValue)` sets the Frequency property to `frequencyValue`.

`waveSynth = wavetableSynthesizer( ___ ,Name,Value)` sets each property `Name` to the specified `Value`. Unspecified properties have default values.

Example: `waveSynth = wavetableSynthesizer('Amplitude',2,'DCOffset',2.5)` creates a System object, `waveSynth`, that generates the default sine waveform with an amplitude of 2 and a DC offset of 2.5.

Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see System Design in MATLAB Using System Objects (MATLAB).

`Wavetable` — Single-cycle waveform

`sin(2*pi*(0:511)/512)` (default) | vector of real values

Single-cycle waveform, specified as a vector of real values. The algorithm of the `wavetableSynthesizer` indexes into the single-cycle waveform to synthesize a periodic wave.

This property is semi-tunable. You can tune the values of the wavetable when the object is locked. However, you cannot tune the length of the wavetable when the object is locked.

Tunable: Yes

Data Types: `single` | `double`

`Frequency` — Frequency of generated signal (Hz)

`100` (default) | real scalar

Frequency of generated signal in Hz, specified as a real scalar greater than or equal to 0.

Tunable: Yes

Data Types: `single` | `double`

`Amplitude` — Amplitude of generated signal

`1` (default) | real scalar

Amplitude of generated signal, specified as a real scalar greater than or equal to 0.

The generated signal is multiplied by the value specified by `Amplitude` at the output, before DCOffset is applied.

Tunable: Yes

Data Types: `single` | `double`

`PhaseOffset` — Normalized phase offset of generated signal

`0` (default) | real scalar

Normalized phase offset of generated signal, specified as a real scalar with values in the range [0, 1]. The range is a normalized 2π radians interval.

Tunable: No

Data Types: `single` | `double`

`DCOffset` — Value added to each element of generated signal

`0` (default) | real scalar

Value added to each element of the generated signal, specified as a real scalar.

Tunable: Yes

Data Types: `single` | `double`

`SamplesPerFrame` — Number of samples per frame

`512` (default) | positive integer

Number of samples per frame, specified as a positive integer in the range [1, 192000].

This property determines the vector length that your `wavetableSynthesizer` object outputs.

Tunable: Yes

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`SampleRate` — Sample rate of generated signal (Hz)
`44100` (default) | real positive scalar

Sample rate of generated signal in Hz, specified as a real positive scalar.

Tunable: Yes

`OutputDataType` — Data type of generated signal
`'double'` (default) | `'single'`

Data type of generated signal, specified as `'double'` or `'single'`.

Tunable: No

Data Types: `char` | `string`

Usage

Note For versions earlier than R2016b, use the `step` function to run the System object™ algorithm. The arguments to `step` are the object you created, followed by the arguments shown in this section.

For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Syntax

```
waveform = waveSynth()
```

Description

`waveform = waveSynth()` generates a periodic signal, `waveform`. The type of signal is specified by the algorithm and properties of the `wavetableSynthesizer` System object, `waveSynth`.

Output Arguments

`waveform` — Waveform output from wavetable synthesizer
column vector (default)

Waveform output from the wavetable synthesizer, returned as a column vector with length specified by the SamplesPerFrame property and data type specified by the OutputDataType property.

Data Types: `single` | `double`

Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

Specific to wavetableSynthesizer

createAudioPluginClass	Create audio plugin class that implements functionality of System object

MIDI

configureMIDI	Configure MIDI connections between audio object and MIDI controller
disconnectMIDI	Disconnect MIDI controls from audio object
getMIDIConnections	Get MIDI connections of audio object

Common to All System Objects

clone	Create duplicate System object
isLocked	Determine if System object is in use

release	Release resources and allow changes to System object property values and input characteristics
reset	Reset internal states of System object
step	Run System object algorithm

The `createAudioPluginClass` and `configureMIDI` functions map tunable properties of the `wavetableSynthesizer` System object to user-facing parameters:

Property	Range	Mapping	Unit
`Frequency`	[0.1, 20000]	log	Hz
`Amplitude`	[0, 10]	linear	none
`DCOffset`	[–10, 10]	linear	none

Examples

Generate Variable-Frequency Staircase Wave

Define and plot a single-cycle waveform.

```
values = -1:0.1:1;
singleCycleWave = ones(100,1) * values;
singleCycleWave = reshape(singleCycleWave,numel(singleCycleWave),1);

plot(singleCycleWave)
xlabel('Index')
ylabel('Amplitude')
```

Create a wavetable synthesizer, `waveSynth`, to generate a staircase wave using the single-cycle waveform. Specify a frequency of 10 Hz.

```
waveSynth = wavetableSynthesizer(singleCycleWave,10);
```

Create a time scope to visualize the staircase wave generated by `waveSynth`.

```
scope = dsp.TimeScope( ...
    'SampleRate',waveSynth.SampleRate, ...
    'TimeSpan',0.1, ...
    'YLimits',[-1.5,1.5], ...
    'TimeSpanOverrunAction','Scroll', ...
```

```
    'ShowGrid',true, ...
    'Title','Variable-Frequency Staircase Wave');
```

Place the wavetable synthesizer in an audio stream loop. Increase the frequency of your staircase wave in 10 Hz increments.

```
counter = 0;
while (counter < 1e4)
    counter = counter + 1;
    staircaseWave = waveSynth();
    scope(staircaseWave)
    if mod(counter,1000)==0
        waveSynth.Frequency = waveSynth.Frequency + 10;
    end
end
```

Manipulate Audio Samples Using Wavetable Synthesizer

Sample an audio file and save it to the `Wavetable` property of a `wavetableSynthesizer` System object™. Use the wavetable synthesizer to manipulate your audio sample.

Read in an entire audio file. Clip out an interesting sound from the audio and then play it.

```
[audio,fs] = audioread('MainStreetOne-24-96-stereo-63secs.wav');

engine = audio(5.35e6:5.45e6);
sound(engine,fs)
```

Create a wavetable synthesizer using your audio clip. The duration of the `engine` audio clip is `numel(engine)/fs` seconds. In the `wavetableSynthesizer`, set the `Frequency` property to 1/(clip duration). The generated signal now plays back at the same rate it was recorded at.

```
duration = numel(engine)/fs;
waveSynth = wavetableSynthesizer('Wavetable',engine,'SampleRate',fs, ...
    'Frequency',1/duration);
```

Create an `audioDeviceWriter` to write to your audio device.

```
deviceWriter = audioDeviceWriter('SampleRate',fs);
```

In a loop, play the wavetable synthesizer to your device. After three seconds, begin increasing the frequency of the wavetable synthesizer. After six seconds, begin decreasing the frequency of the wavetable synthesizer.

```
timeElapsed = 0;
while timeElapsed < 9
    audioWave = waveSynth();
    deviceWriter(audioWave);

    if (timeElapsed > 3) && (timeElapsed < 6)
        waveSynth.Frequency = waveSynth.Frequency + 0.001;
    elseif timeElapsed > 6
        waveSynth.Frequency = waveSynth.Frequency - 0.002;
    end

    timeElapsed = timeElapsed + waveSynth.SamplesPerFrame*(1/fs);
end
```

Modify Wavetable While Stream Processing

Modify the `Wavetable` property of a `wavetableSynthesizer` System object™ while stream processing. Visualize the wavetable and play the resulting audio.

Create a single-cycle waveform for the `wavetableSynthesizer` to index into. Create a wavetable synthesizer object.

```
t = 0:0.001:1;
exponent = 5;
waveTable = [t.^exponent,fliplr(t.^exponent)] - 0.5;

waveSynth = wavetableSynthesizer('Wavetable',waveTable);
```

Create a `dsp.ArrayPlot` object to plot the wavetable as it is modified over time. Create an `audioDeviceWriter` object to listen to the signal output by your wavetable synthesizer.

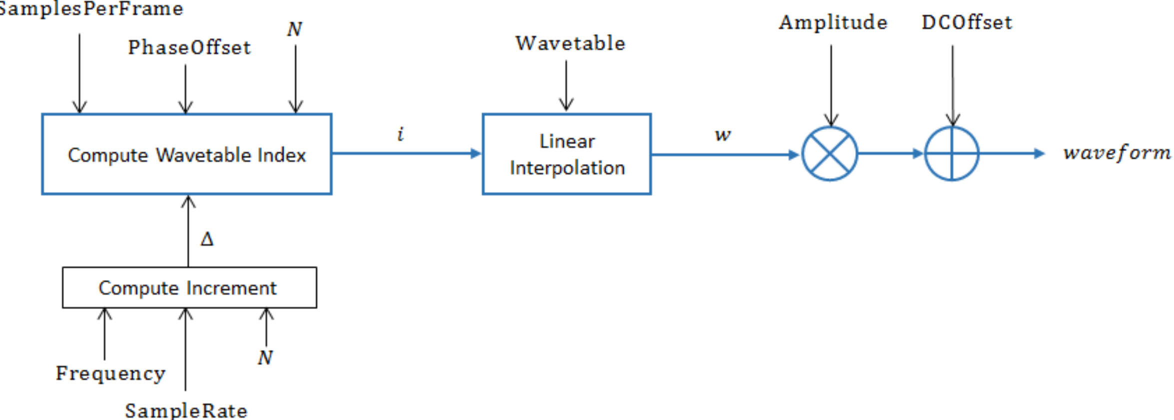
```
arrayPlotter = dsp.ArrayPlot('YLimits',[-1,1],'PlotType','Line');
deviceWriter = audioDeviceWriter;
```

In an audio stream loop, incrementally modify the `Wavetable` property of the wavetable synthesizer and plot it. Call the synthesizer to output a waveform and play the waveform to your audio device.

```
tic
while toc < 10
    exponent = exponent - 0.01;
    waveSynth.Wavetable = [t.^abs(exponent),fliplr(t.^abs(exponent))] - 0.5;

    arrayPlotter(waveSynth.Wavetable')

    deviceWriter(waveSynth());
end

release(deviceWriter)
```

Algorithms

The `wavetableSynthesizer` System object synthesizes periodic signals using a cached single-cycle waveform, specified waveform properties, and phase memory.



Compute Increment

Compute the increment step size:

$$\Delta = \frac{\text{Frequency}}{\text{SampleRate}} \times N \ ,$$

where N is the number of elements in your wavetable.

Compute Wavetable Index

Compute `Wavetable` index,

$$i[n] = \begin{cases} i[n-1] + \Delta & i[n-1] < N \\ i[n-1] + \Delta - N & i[n-1] \geq N \end{cases},$$

for $2 \leq n \leq \text{SamplesPerFrame}$. The `PhaseOffset` property determines $i[n=1]$.

Linear Interpolation

Index into the `Wavetable` and perform linear interpolation:

$$w = \begin{cases} \left(\text{Wavetable}[1] - \text{Wavetable}[i_{\text{L}}]\right) \times \varepsilon + \text{Wavetable}[i_{\text{L}}] & i_{\text{H}} > N \\ \left(\text{Wavetable}[i_{\text{H}}] - \text{Wavetable}[i_{\text{L}}]\right) \times \varepsilon + \text{Wavetable}[i_{\text{L}}] & i_{\text{H}} \leq N \end{cases}.$$

- $i_{\text{L}} = \text{floor}(i[n] + 1)$
- $i_{\text{H}} = i_{\text{L}} + 1$
- $\varepsilon = i - \text{floor}(i)$

Apply Amplitude and DC Offset

Multiply by `Amplitude` and add `DCOffset`.

$$waveform = w \times \text{Amplitude} + \text{DCOffset}$$

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

"System Objects in MATLAB Code Generation" (MATLAB Coder)

See Also

System Objects
`audioOscillator`

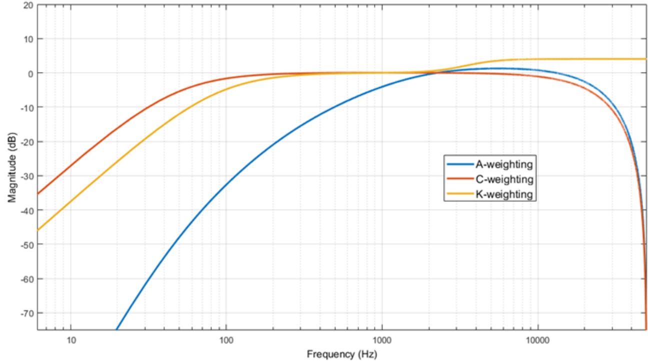
Introduced in R2016a

weightingFilter System object

Frequency-weighted filter

Description

The `weightingFilter` System object performs frequency-weighted filtering independently across each input channel.



To perform frequency-weighted filtering:

1. Create the weightingFilter object and set its properties.
2. Call the object with arguments, as if it were a function.

To learn more about how System objects work, see What Are System Objects? (MATLAB).

Creation

Syntax

`weightFilt = weightingFilter`

```
weightFilt = weightingFilter(weightType)
weightFilt = weightingFilter(weightType,Fs)
weightFilt = weightingFilter(___,Name,Value)
```

Description

`weightFilt = weightingFilter` creates a System object, `weightFilt`, that performs frequency-weighted filtering independently across each input channel.

`weightFilt = weightingFilter(weightType)` sets the Method property to `weightType`.

`weightFilt = weightingFilter(weightType,Fs)` sets the SampleRate property to `Fs`.

`weightFilt = weightingFilter(___,Name,Value)` sets each property `Name` to the specified `Value`. Unspecified properties have default values.

Example: `weightFilt = weightingFilter('C-weighting','SampleRate', 96000)` creates a C-weighting filter with a sample rate of 96,000 Hz.

Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see System Design in MATLAB Using System Objects (MATLAB).

`Method` — Type of weighting

`'A-weighting'` (default) | `'C-weighting'` | `'K-weighting'`

Type of weighting, specified as `'A-weighting'`, `'C-weighting'`, or `'K-weighting'`. See “Algorithms” on page 3-327 for more information.

Tunable: No

Data Types: `char` | `string`

`SampleRate` — Input sample rate (Hz)
`44100` (default) | positive scalar

Input sample rate in Hz, specified as a positive scalar.

Tunable: Yes

Data Types: `single` | `double`

Usage

Syntax

`audioOut = weightFilt(audioIn)`

Description

`audioOut = weightFilt(audioIn)` applies frequency-weighted filtering to the input signal, `audioIn`, and returns the filtered signal, `audioOut`. The type of filtering is specified by the algorithm and properties of the `weightingFilter` System object, `weightFilt`.

Input Arguments

`audioIn` — Audio input to weighting filter
matrix

Audio input to the weighting filter, specified as a matrix. The columns of the matrix are treated as independent audio channels.

Data Types: `single` | `double`

Output Arguments

`audioOut` — Audio output from weighting filter
matrix

Audio output from the weighting filter, returned as a matrix the same size as `audioIn`.

Data Types: `single` | `double`

Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

Specific to weightingFilter

visualize	Visualize and validate filter response
getFilter	Return biquad filter object with design parameters set
createAudioPluginClass	Create audio plugin class that implements functionality of System object
isStandardCompliant	Verify filter design is IEC 61672-1:2002 compliant

Common to All System Objects

clone	Create duplicate System object
isLocked	Determine if System object is in use
release	Release resources and allow changes to System object property values and input characteristics
reset	Reset internal states of System object
step	Run System object algorithm

Examples

Validate Filter Compliance

Check the compliance status of filter designs and visualize them.

Create an A-weighting filter with a 22.5 kHz sample rate. Verify that the filter is standard compliant and visualize the filter design.

```
aWeight = weightingFilter('A-weighting','SampleRate',22500);
complianceStatus = isStandardCompliant(aWeight,'class 1')
visualize(aWeight,'class 1')
```

Change your A-weighting filter sample rate to 44.1 kHz. Verify that the filter is standard compliant and visualize the filter design.

```
aWeight.SampleRate = 44100;
complianceStatus = isStandardCompliant(aWeight,'class 1')
visualize(aWeight,'class 1')
```

Perform A-Weighted Filtering

Use the `weightingFilter` System object™ to design an A-weighted filter, and then process an audio signal using your frequency-weighted filter design.

Create a `dsp.AudioFileReader` System object.

```
samplesPerFrame = 1024;
reader = dsp.AudioFileReader('Filename', ...
    'RockGuitar-16-44p1-stereo-72secs.wav', ...
    'SamplesPerFrame',samplesPerFrame, ...
    'PlayCount',Inf);
```

Create a `weightingFilter` System object. Use the sample rate of the reader as the sample rate of the weighting filter.

```
Fs = reader.SampleRate;
weightFilt = weightingFilter('A-weighting',Fs);
```

Visualize the filter response and verify that it fits within the class 1 mask of the IEC 61672-1:2002 standard.

```
visualize(weightFilt,'class 1')
```

Create a spectrum analyzer to visualize the original audio signal and the audio signal after frequency-weighted filtering.

```
scope = dsp.SpectrumAnalyzer( ...
    'SampleRate',Fs, ...
    'PlotAsTwoSidedSpectrum',false, ...
    'FrequencyScale','Log', ...
    'FrequencyResolutionMethod','WindowLength', ...
    'WindowLength',samplesPerFrame, ...
    'Title','A-Weighted Filtering', ...
```

```
    'ShowLegend',true, ...
    'ChannelNames',{'Original signal','Filtered signal'});
```

Process the audio signal in an audio stream loop. Visualize the filtered audio and the original audio. As a best practice, release the System objects when complete.

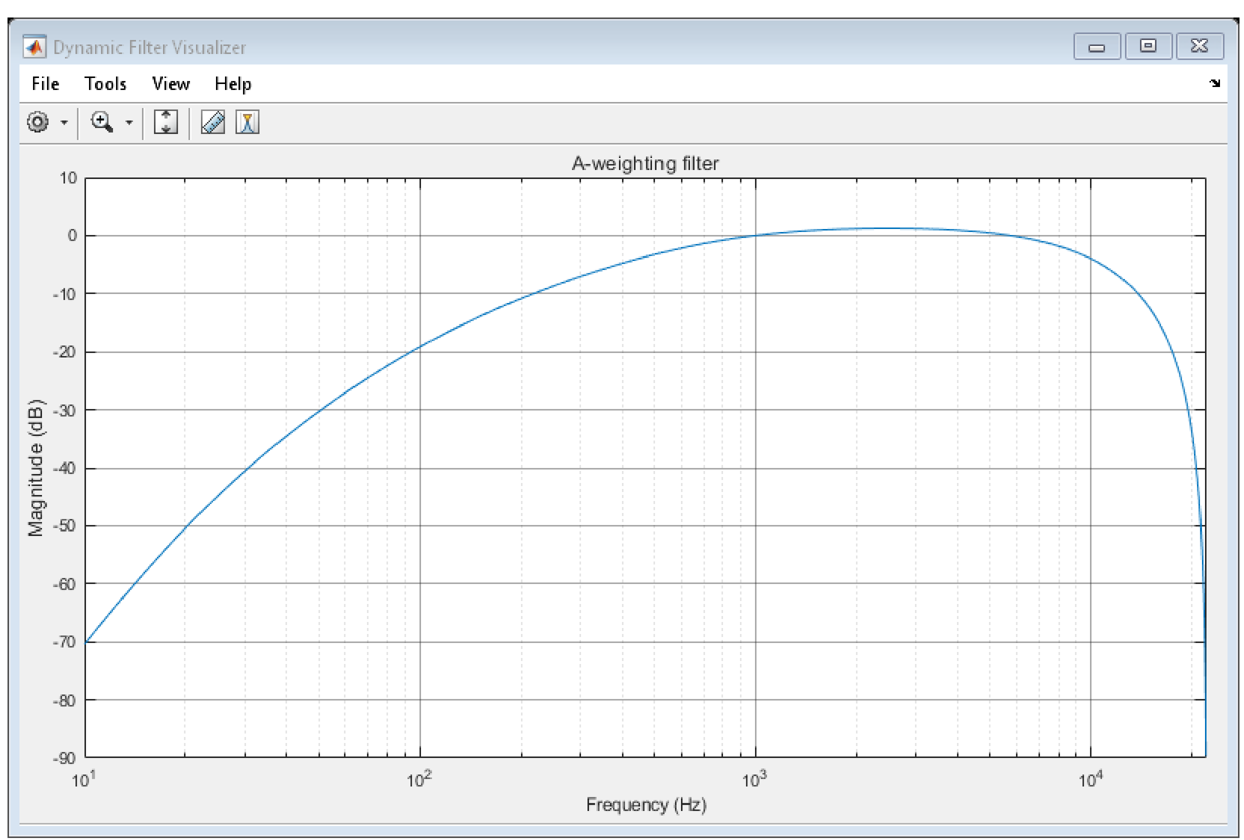
```
tic
while toc < 20
    x = reader();
    y = weightFilt(x);
    scope([x(:,1),y(:,1)])
end

release(weightFilt)
release(scope)
release(reader)
```

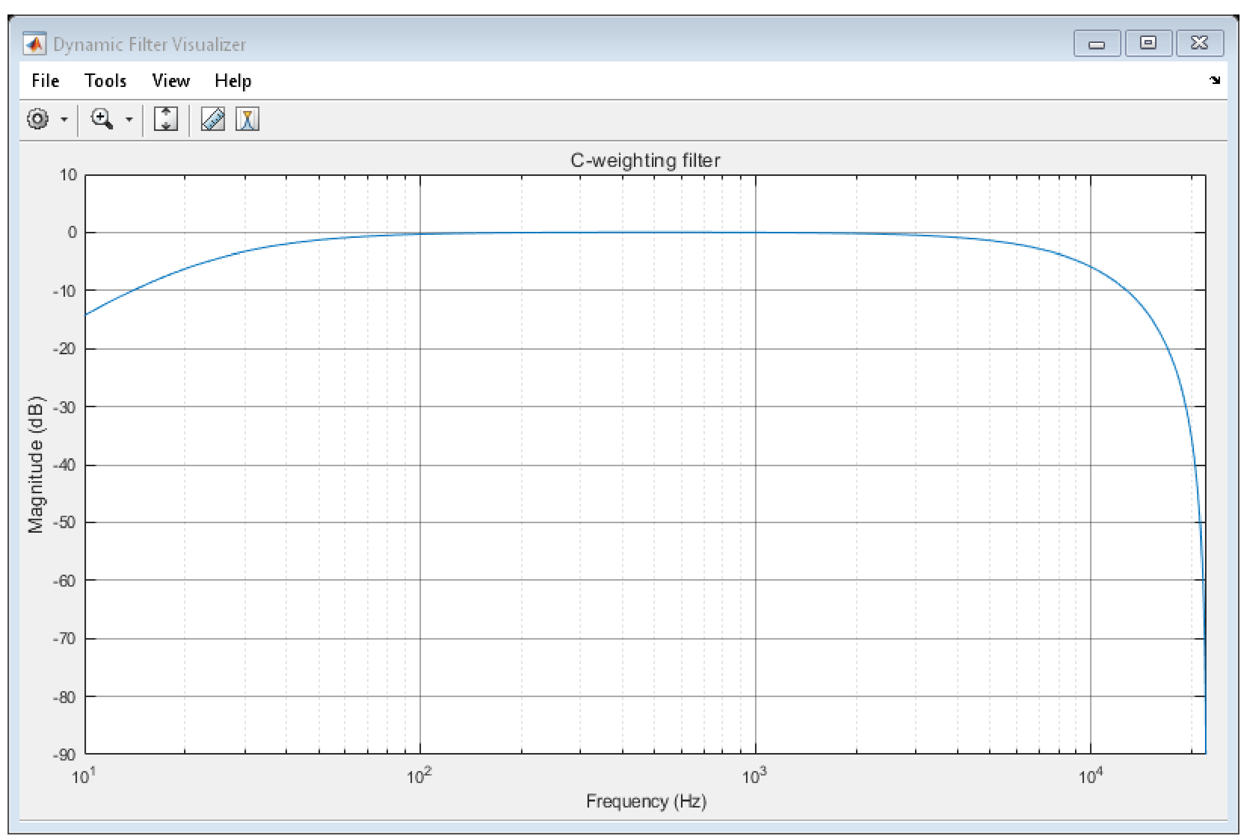
Compare Weighting Types

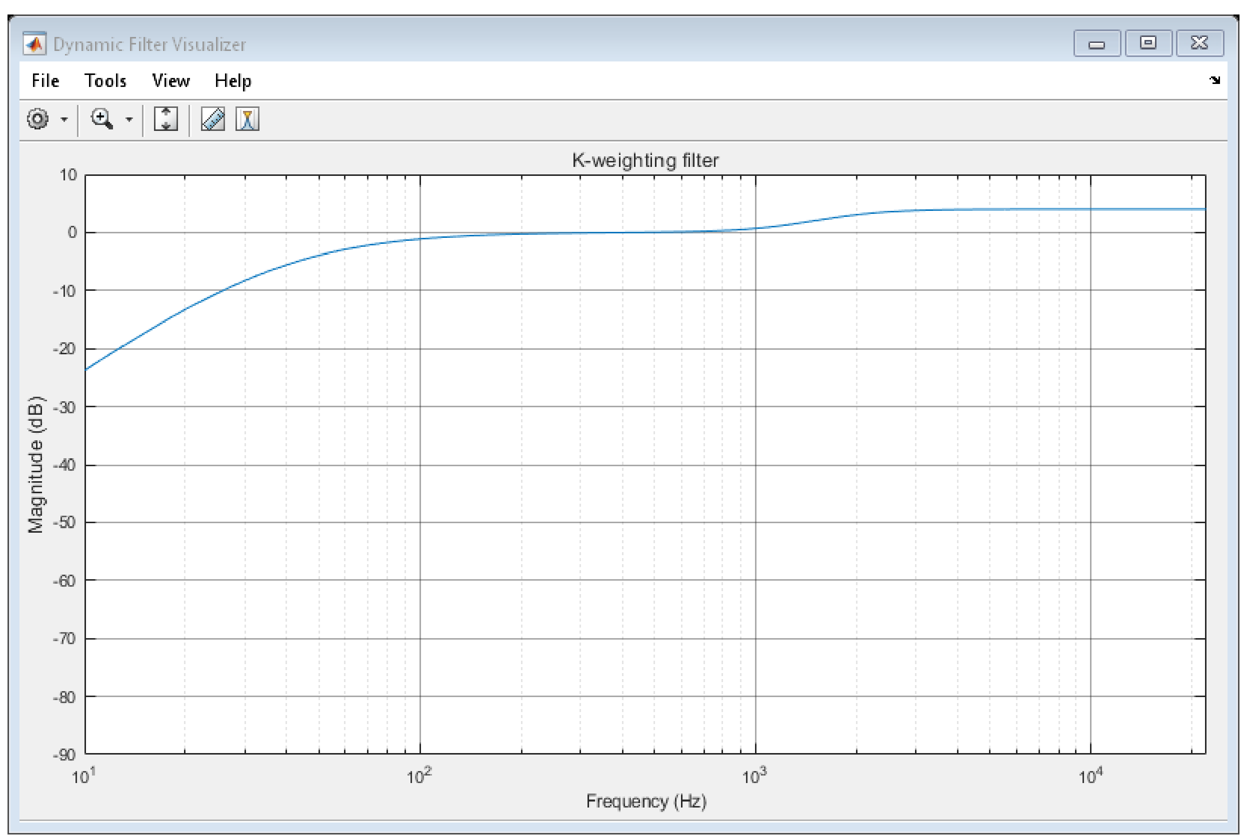
Compare the A-weighted, C-weighted, and K-weighted filtering of an engine sound.

Create an A-weighting filter, a C-weighting filter, and a K-weighting filter. Visualize the filters for analysis and comparison.

```
wF{1} = weightingFilter;
visualize(wF{1})

wF{2} = weightingFilter('C-weighting');
visualize(wF{2})

wF{3} = weightingFilter('K-weighting');
visualize(wF{3})
```


Dynamic Filter Visualizer
File
Tools
View
Help
A-weighting filter
Magnitude (dB)
10
0
-10
-20
-30
-40
-50
-60
-70
-80
-90
10^1
10^2
10^3
10^4
Frequency (Hz)


Dynamic Filter Visualizer
File Tools View Help
C-weighting filter
Magnitude (dB)
Frequency (Hz)
10
0
-10
-20
-30
-40
-50
-60
-70
-80
-90
10^1
10^2
10^3
10^4



Create a `dsp.AudioFileReader` and specify a sound file. Create an `audioDeviceWriter` with default properties. In an audio stream loop, play the white noise, and then listen to it filtered through the A-weighted, C-weighted, and K-weighted filters, successively.

```
fileReader = dsp.AudioFileReader('Engine-16-44p1-stereo-20sec.wav');
deviceWriter = audioDeviceWriter('SampleRate',fileReader.SampleRate);

fprintf('No filtering...')
for i = 1:400
    x = fileReader();
    if i==100
        index = 1;
        fprintf('A-weighted filtering...')
```

```
    elseif i==200
        index = 2;
        fprintf('C-weighted filtering...')
    elseif i==300
        index = 3;
        fprintf('K-weighted filtering...\n')
    end
    if i>99
        y = wF{index}(x);
    else
        y = x;
    end
    deviceWriter(y);
end

release(deviceWriter)
release(fileReader)
```

```
No filtering...A-weighted filtering...C-weighted filtering...K-weighted filtering...
```

Use Weighting Filter Design with Biquad Filter

The `weightingFilter` object uses second-order sections (SOS) for filtering. To extract the weighting filter design, use `getFilter` to return a `dsp.BiquadFilter` object with the `SOSMatrix` and `ScaleValues` properties set.

Use `weightingFilter` to create C-weighted and A-weighted filter objects. Use `getFilter` to return corresponding `dsp.BiquadFilter` objects.

```
cFilt = weightingFilter('C-weighting');
aFilt = weightingFilter('A-weighting');
cSOSFilter = getFilter(cFilt);
aSOSFilter = getFilter(aFilt);
```

Create an audio file reader and audio device writer for audio input/output. Use the sample rate of your reader as the sample rate of your writer.

```
fileReader = dsp.AudioFileReader('JetAirplane-16-11p025-mono-16secs.wav');
deviceWriter = audioDeviceWriter('SampleRate',fileReader.SampleRate);
```

In an audio stream loop, play the unfiltered signal. Release your file reader so that the next time you call it, it reads from the beginning of the file.

```
tic
while toc<8
    x = fileReader();
    deviceWriter(x);
end
release(fileReader)
```

Play the signal processed by the A-weighted filter. Then play the signal processed by the C-weighted filter. Cache the power in each frame of the original and filtered signals for analysis. As a best practice, release your file reader and device writer once complete.

```
y = [];
count = 1;
tic
while ~isDone(fileReader)
    x = fileReader();
    aFiltered = aSOSFilter(x);
    cFiltered = cSOSFilter(x);
    if toc>8
        deviceWriter(cFiltered);
    else
        deviceWriter(aFiltered);
    end
    xPower(count) = var(x);
    aPower(count) = var(aFiltered);
    cPower(count) = var(cFiltered);
    y = [y;x];
    count = count+1;
end

release(fileReader)
release(deviceWriter)
```

Plot the power of the original signal, the A-weighted signal, and the C-weighted signal over time.

```
subplot(2,1,1)
    spectrogram(y,512,256,4096,fileReader.SampleRate,'yaxis')
    title('Original Signal')
subplot(2,1,2)
    t = linspace(0,16.3468,count-1);
    plot(t,xPower,'r',t,aPower,'b',t,cPower,'g')
        legend('Original Signal','A-Weighted','C-Weighted')
        xlabel('Time (s)')
        ylabel('Power')
```

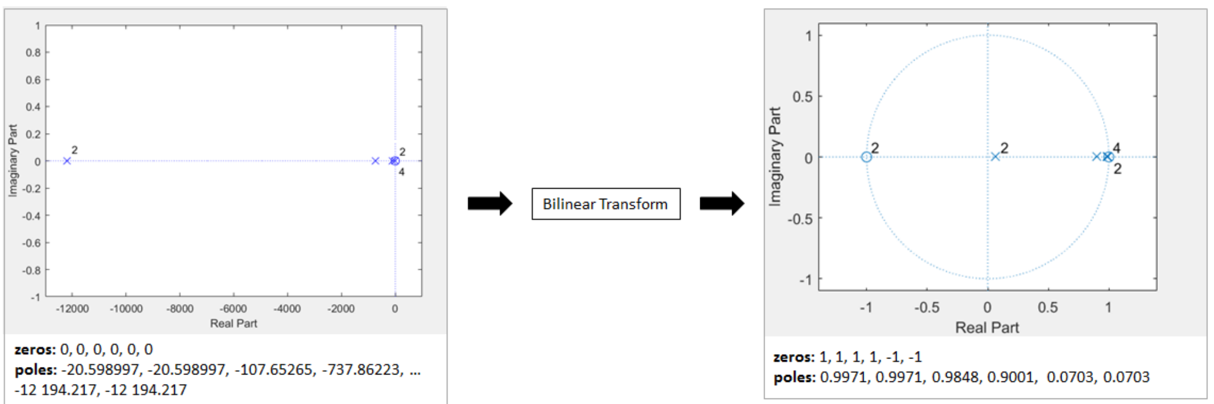
Algorithms

A-Weighting

The A-curve is a wide bandpass filter centered at 2.5 kHz, with approximately 20 dB attenuation at 100 Hz and 10 dB attenuation at 20 kHz. A-weighted SPL measurements of noise level are increasingly found in sales literature for domestic appliances. In most countries, the use of A-weighting is mandated for the protection of workers against noise-induced deafness. The ISO and ICOA standards mandate A-weighting for all civil aircraft noise measurements.

The ANSI S1.42.2001 [1] defines this weighting curve. The IEC 61672-1:2002 [2] standard defines the minimum and maximum attenuation limits for an A-weighting filter.

ANSI S1.42.2001 defines the weighting curve by specifying analog poles and zeros. Audio System Toolbox converts the specified poles and zeros to the digital domain using a bilinear transform:

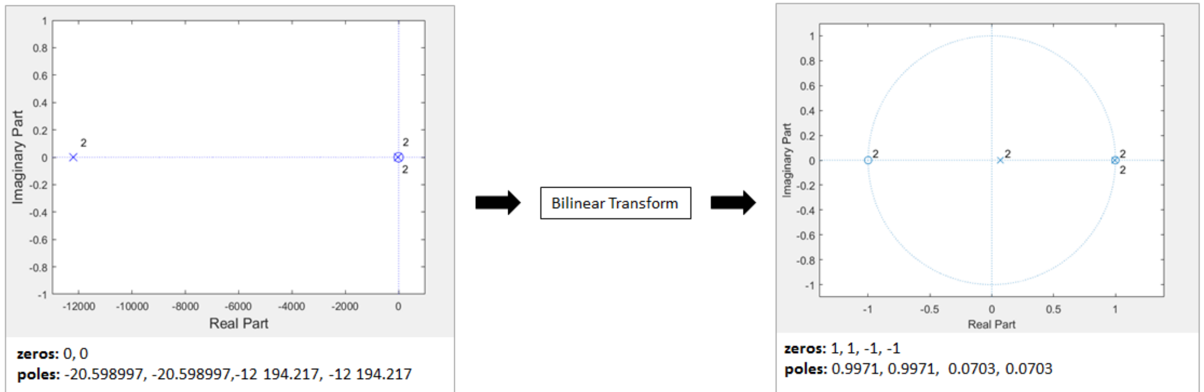


C-Weighting

The C-curve is "flat," but with limited bandwidth: It has –3 dB corners at 31.5 Hz and 8 kHz. C-curves are used in sound level meters for sounds that are louder than those intended for A-weighting filters.

The ANSI S1.42-2001 [1] defines the C-weighting curve. The IEC 61672-1:2002 [2] standard defines the minimum and maximum attenuation limits for C-weighting filters.

ANSI S1.42.2001 defines the weighting curve by specifying analog poles and zeros. Audio System Toolbox converts the specified poles and zeros to the digital domain using a bilinear transform:

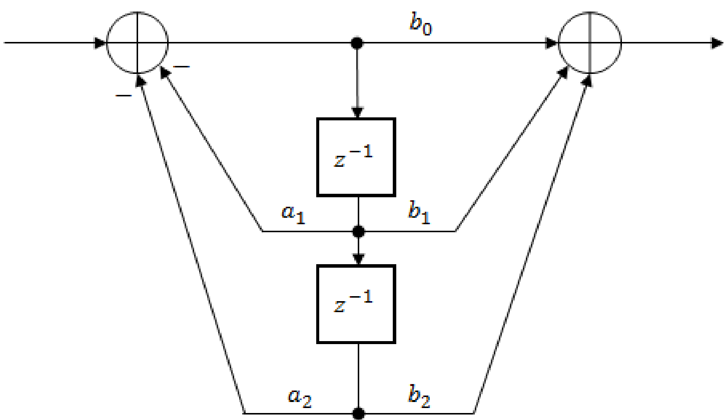


K-Weighting

The K-weighting filter is used for loudness normalization in broadcast. It is composed of two stages of filtering: a first stage shelving filter and a second stage highpass filter.

The ITU-R BS.1770-4 [3] standard defines this curve.

Assume a second-order filter.



The table shows the coefficients for the filters.

First Stage Shelving Coefficients	Second Stage Highpass Coefficients
$a_1 = -1.69065929318241$	$a_1 = -1.99004745483398$
$a_2 = 0.73248077421585$	$a_2 = 0.99007225036621$
$b_0 = 1.53512485958697$	$b_0 = 1.0$
$b_1 = -2.69169618940638$	$b_1 = -2.0$
$b_2 = 1.19839281085285$	$b_2 = 1.0$

The coefficients presented by ITU-R BS.1770-4 are defined for 48 kHz. These coefficients are recomputed for nonstandard sample rates using the algorithm described in [4].

References

[1] Acoustical Society of America. *Design Response of Weighting Networks for Acoustical Measurements*. ANSI S1.42-2001. New York, NY: American National Standards Institute, 2001.

[2] International Electrotechnical Commission. *Electroacoustics Sound Level Meters Part 1: Specifications*. First Edition. IEC 61672-1. 2002–2005.

[3] International Telecommunication Union. *Algorithms to measure audio programme loudness and true-peak audio level*. ITU-R BS.1770-4. 2015.

[4] Mansbridge, Stuart, Saoirse Finn, and Joshua D. Reiss. "Implementation and Evaluation of Autonomous Multi-track Fader Control." Paper presented at the 132nd Audio Engineering Society Convention, Budapest, Hungary, 2012.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

"System Objects in MATLAB Code Generation" (MATLAB Coder)

See Also

Blocks
Weighting Filter

System Objects
`dsp.BiquadFilter` | `multibandParametricEQ` | `octaveFilter`

Topics

"Audio Weighting Filters"
"Sound Pressure Measurement Using Weighting Filters"

Introduced in R2016b

isStandardCompliant

Verify filter design is IEC 61672-1:2002 compliant

Syntax

```
complianceStatus = isStandardCompliant(weightFilt,classType)
complianceStatus = isStandardCompliant( ___ ,freqRange)
```

Description

`complianceStatus = isStandardCompliant(weightFilt,classType)` returns a logical scalar, `complianceStatus`, indicating whether the `weightFilt` filter design is compliant with the minimum and maximum attenuation specifications for the `classType` design specified in IEC 61672-1:2002. You can check compliance for A-weighting and C-weighting filters only.

`complianceStatus = isStandardCompliant( ___ ,freqRange)` specifies the range of frequencies checked for compliance.

Examples

Verify Class 1 Standard Compliance

Create an object of the `weightingFilter` System object™. Call `isStandardCompliant`, specifying the compliance class type to check as the second argument.

```
weightFilt = weightingFilter;
complianceStatus = isStandardCompliant(weightFilt,'class 1')
```

Specify Frequency Range Checked for Compliance

Create an object of the `weightingFilter` System object™. Check the 'class 2' compliance status of the filter design over a specified frequency range.

```
weightFilt = weightingFilter;
isStandardCompliant(weightFilt,'class 2',[120,2000])
```

Input Arguments

`weightFilt` — Object of `weightingFilter`
object

Object of the `weightingFilter` System object.

`classType` — Compliance class type
`'class 1'` | `'class 2'`

Compliance class type to verify, specified as `'class 1` or `'class 2'`.

Data Types: `char`

`freqRange` — Frequency range checked for compliance (Hz)
`[minFreq,maxFreq]` | two-element vector of increasing values

Specify the frequency range, in Hz, checked for compliance as a two-element vector of increasing values: `[minFreq,maxFreq]`.

Data Types: `single` | `double`

Output Arguments

`complianceStatus` — Compliance status of filter design
scalar

Compliance status of filter design, returned as a logical scalar. The compliance status indicates whether the `weightFilt` filter design is compliant with the minimum and maximum attenuation specifications for the class type design specified by IEC 61672-1:2002 standard. Compliance can only be checked for A-weighting and C-weighting filters.

Data Types: `logical`

Note The pole-zero values defined in the ANSI S1.42-2001 standard are used for designing the A-weighted and C-weighted filters. The pole-zero values are based on analog filters, so the design can break compliance for lower sample rates.

See Also

Topics

"Audio Weighting Filters"
"Sound Pressure Measurement Using Weighting Filters"

Introduced in R2016b

visualize

Visualize and validate filter response

Syntax

```
visualize(weightFilt)
visualize(weightFilt,N)
visualize( ___ ,mType)
```

Description

`visualize(weightFilt)` plots the magnitude response of the frequency-weighted filter, `weightFilt`. The plot is updated automatically when properties of the object change.

`visualize(weightFilt,N)` uses an `N`-point FFT to calculate the magnitude response.

`visualize( ___ ,mType)` creates a mask based on the class of filter specified by `mType`, using either of the previous syntaxes.

Examples

Plot Weighting Filter Magnitude Response

Create an object of the `weightingFilter` System object™ and then plot the magnitude response of the filter.

```
weightFilt = weightingFilter;
visualize(weightFilt)
```

Specify Number of Frequency Bins in FFT Calculation

Create an object of the `octaveFilter` System object™. Plot a 1024-point frequency representation.

```
weightFilt = weightingFilter;
visualize(weightFilt,1024)
```

Visualize Class 2 Standard-Compliance Mask

Create an object of the `weightFilt` System object™. Visualize the class 2 compliance of the filter design.

```
weightFilt = weightingFilter;
visualize(weightFilt,'class 2')
```

Input Arguments

`weightFilt` — Object of `weightingFilter`
object

Object of the `weightingFilter` System object.

`N` — Number of DFT bins
`2048` | positive scalar

Number of DFT bins in frequency-domain representation, specified as a positive scalar. The default is `2048`.

Data Types: `single` | `double`

`mType` — Type of mask
`'class 1'` (default) | `'class 2'`

Type of mask, specified as `'class 1'` or `'class 2'`.

The mask attenuation limits are defined in the IEC 61672-1:2002 standard. The mask is defined for A-weighting and C-weighting filters only.

- If the mask is green, the design is compliant with the IEC 61672-1:2002 standard.
- If the mask is red, the design breaks compliance.

Note The pole-zero values defined in the ANSI S1.42-2001 standard are used for designing the A-weighted and C-weighted filters. The pole-zero values are based on analog filters, so the design can break compliance for lower sample rates.

Data Types: `char`

See Also

Topics

“Audio Weighting Filters”
“Sound Pressure Measurement Using Weighting Filters”

Introduced in R2016b

4

Classes in Audio System Toolbox

midimsg

Create MIDI message

Description

Create a MIDI message in MATLAB using `midimsg`. Create a MIDI device interface using `mididevice`. Send and receive messages using `midisend` and `midireceive`. When you create a MIDI message, you specify it as a MIDI message type.

MATLAB

mididevice

midimsg ← midireceive ← Buffer ← MIDI message ← MIDI Device

midimsg → midisend → Buffer → MIDI message → MIDI Device

For a tutorial on MIDI messages and interfacing with MIDI devices, see “MIDI Device Interface”.

Creation

Syntax

```
msg = midimsg('Note',channel,note,velocity,duration,timestamp)
msg = midimsg('NoteOn',channel,note,velocity,timestamp)
msg = midimsg('NoteOff',channel,note,velocity,timestamp)
msg = midimsg('ControlChange',channel,ccnumber,ccvalue,timestamp)
msg = midimsg('ProgramChange',channel,program,timestamp)
msg = midimsg('SystemExclusive',bytes,timestamp)
```

```
msg = midimsg('SystemExclusive',timestamp)
msg = midimsg('Data',bytes,timestamp)
msg = midimsg('EOX',timestamp)
msg = midimsg('TimingClock',timestamp)
msg = midimsg('Start',timestamp)
msg = midimsg('Continue',timestamp)
msg = midimsg('Stop',timestamp)
msg = midimsg('ActiveSensing',timestamp)
msg = midimsg('SystemReset',timestamp)
msg = midimsg('TuneRequest',timestamp)
msg = midimsg('MIDITimeCodeQuarterFrame',seq,value,timestamp)
msg = midimsg('SongPositionPointer',position,timestamp)
msg = midimsg('SongSelect',song,timestamp)
msg = midimsg('AllSoundOff',channel,timestamp)
msg = midimsg('ResetAllControllers',channel,timestamp)
msg = midimsg('LocalControl',channel,localcontrol,timestamp)
msg = midimsg('PolyOn',channel,timestamp)
msg = midimsg('MonoOn',channel,monoChannels,timestamp)
msg = midimsg('OmniOn',channel,timestamp)
msg = midimsg('OmniOff',channel,timestamp)
msg = midimsg('AllNotesOff',channel,timestamp)
msg = midimsg('PolyKeyPressure',channel,note,pressure,timestamp)
msg = midimsg('ChannelPressure',channel,pressure,timestamp)
msg = midimsg('PitchBend',channel,change,timestamp)
msg = midimsg
msg = midimsg(size)
msg = midimsg(0)
```

Description

`msg = midimsg('Note',channel,note,velocity,duration,timestamp)` returns two MIDI messages: `NoteOn` and `NoteOff`, with specified `Channel`, `Note`, `Velocity`, and `Timestamp` properties. The `Timestamp` property of the `NoteOff` message is determined as the `Timestamp` property of the `NoteOn` message plus the `duration`.

`msg = midimsg('NoteOn',channel,note,velocity,timestamp)` returns a `NoteOn` `midimsg`, with specified `Channel`, `Note`, `Velocity`, and `Timestamp` properties.

`msg = midimsg('NoteOff',channel,note,velocity,timestamp)` returns a `NoteOff` `midimsg`, with specified `Channel`, `Note`, `Velocity`, and `Timestamp` properties.

`msg = midimsg('ControlChange',channel,ccnumber,ccvalue,timestamp)` returns a `ControlChange` `midimsg`, with specified `Channel`, `CCNumber`, `CCValue`, and `Timestamp` properties.

`msg = midimsg('ProgramChange',channel,program,timestamp)` returns a `ProgramChange` `midimsg`, with specified `Channel`, `Program`, and `Timestamp` properties.

`msg = midimsg('SystemExclusive',bytes,timestamp)` returns a complete `SystemExclusive` message sequence, with specified `Timestamp` property.

`msg = midimsg('SystemExclusive',timestamp)` returns a `SystemExclusive` `midimsg`, with specified `Timestamp` property.

`msg = midimsg('Data',bytes,timestamp)` returns a `Data` `midimsg` for use in a System Exclusive message, with specified `MsgBytes` and `Timestamp` properties. `bytes` is specified as a scalar, vector, or multi-dimensional array of elements. Each element of `bytes` must be in the range [0,127].

`msg = midimsg('EOX',timestamp)` returns an `EOX` `midimsg`, with specified `Timestamp` property.

`msg = midimsg('TimingClock',timestamp)` returns a `TimingClock` `midimsg`, with specified `Timestamp` property.

`msg = midimsg('Start',timestamp)` returns a `Start` `midimsg`, with specified `Timestamp` property.

`msg = midimsg('Continue',timestamp)` returns a `Continue` `midimsg`, with specified `Timestamp` property.

`msg = midimsg('Stop',timestamp)` returns a `Stop` `midimsg`, with specified `Timestamp` property.

`msg = midimsg('ActiveSensing',timestamp)` returns a `ActiveSensing` `midimsg`, with specified `Timestamp` property.

`msg = midimsg('SystemReset',timestamp)` returns a `SystemReset` `midimsg`, with specified `Timestamp` property.

msg = midimsg('TuneRequest',timestamp) returns a TuneRequest midimsg, with specified Timestamp property.

msg = midimsg('MIDITimeCodeQuarterFrame',seq,value,timestamp) returns a MIDITimeCodeQuarterFrame midimsg, with specified TimeCodeSequence, TimeCodeValue, and Timestamp properties.

msg = midimsg('SongPositionPointer',position,timestamp) returns a SongPositionPointer midimsg, with specified SongPosition and Timestamp properties.

msg = midimsg('SongSelect',song,timestamp) returns a SongSelect midimsg, with specified Song and Timestamp properties.

msg = midimsg('AllSoundOff',channel,timestamp) returns a AllSoundOff midimsg, with specified Channel and Timestamp properties.

msg = midimsg('ResetAllControllers',channel,timestamp) returns a ResetAllControllers midimsg, with specified Channel and Timestamp properties.

msg = midimsg('LocalControl',channel,localcontrol,timestamp) returns a LocalControl midimsg, with specified Channel, LocalControl, and Timestamp properties.

msg = midimsg('PolyOn',channel,timestamp) returns a PolyOn midimsg, with specified Channel and Timestamp properties.

msg = midimsg('MonoOn',channel,monoChannels,timestamp) returns a MonoOn midimsg, with specified Channel, MonoChannels, and Timestamp properties.

msg = midimsg('OmniOn',channel,timestamp) returns an OmniOn midimsg, with specified Channel and Timestamp properties.

msg = midimsg('OmniOff',channel,timestamp) returns an OmniOff midimsg, with specified Channel and Timestamp properties.

msg = midimsg('AllNotesOff',channel,timestamp) returns an AllNotesOff midimsg, with specified Channel and Timestamp properties.

msg = midimsg('PolyKeyPressure',channel,note,pressure,timestamp) returns a PolyKeyPressure midimsg, with specified Channel, Note, Pressure, and Timestamp properties.

`msg = midimsg('ChannelPressure',channel,pressure,timestamp)` returns a `ChannelPressure` `midimsg`, with specified `Channel`, `Pressure`, and `Timestamp` properties.

`msg = midimsg('PitchBend',channel,change,timestamp)` returns a `PitchBend` `midimsg`, with specified `Channel`, `PitchChange`, and `Timestamp` properties.

`msg = midimsg` returns a scalar `midimsg` with all zero bytes. All zero bytes indicates a MIDI message with `Type` set to `Data`.

`msg = midimsg(size)` returns a `midimsg` array of `size` with all zero bytes.

`msg = midimsg(0)` returns an empty `midimsg`.

Note If `timestamp` is listed as an argument, it is optional and defaults to zero. The exception is the `'SystemExclusive',bytes,timestamp` syntax, in which case the timestamp argument is required.

Properties

Type — Type of MIDI message
`NoteOn` | `NoteOff` | `ControlChange` | `ProgramChange` | `SystemExclusive` | `Data` | `EOX` | ...

This property is read-only.

Type of MIDI message, returned as one of the following `midimsgtype` enumeration values:

NoteOn	Data	Stop	SongPositionPointer	PolyOn	PolyKeyPressure
NoteOff	EOX	ActiveSensing	SongSelect	MonoOn	ChannelPressure
ControlChange	TimingClock	SystemReset	AllSoundOff	OmniOn	PitchBendChange
ProgramChange	Start	TuneRequest	ResetAllControllers	OmniOff	Undefined

SystemExclusive	Continue	MIDITimeCodeQuarterFrame	LocalControl	AllNotesOff	

You can specify the type of MIDI message during creation as a character vector, string, or member of the `midimsgtype` enumeration.

For example, the following create equivalent MIDI messages:

- `midimsg('SongPositionPointer',1)`
- `midimsg("SongPositionPointer",1)`
- `midimsg(midimsgtype.SongPositionPointer,1)`

`NumMsgBytes` — Number of bytes in MIDI message
scalar | vector | array

This property is read-only.

Number of bytes in the MIDI message, returned as a scalar, vector, or array the same size as `msg`.

Data Types: `double`

`MsgBytes` — Actual bytes of constructed MIDI message (decimal)
scalar | vector | array

This property is read-only.

Actual bytes of the constructed MIDI message in decimal, returned as a scalar, vector, or array the same size as `msg`.

Data Types: `uint8`

`Timestamp` — Location in time for MIDI message
scalar | vector | array

Location in time for the MIDI message, specified as a scalar, vector, or array the same size as `msg`.

You can specify the timestamp as any numeric value. However, the timestamp is always stored and returned as type `double`.

For more on how MIDI timestamps are implemented in Audio System Toolbox, see “MIDI Message Timing”.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`Channel` — MIDI channel to which message is addressed
integer in the range [1,16]

MIDI channel to which message is addressed, specified as an integer in the range [1,16].

Dependencies

This property is valid only for `NoteOn`, `NoteOff`, `PolyKeyPressure`, `AllSoundOff`, `ResetAllControllers`, `LocalControl`, `AllNotesOff`, `OmniOn`, `OmniOff`, `MonoOn`, `PolyOn`, `ControlChange`, `ProgramChange`, `ChannelPressure`, and `PitchBend` `midimsg` objects.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`Note` — MIDI note number
integer in the range [0,127]

MIDI note number, specified as an integer in the range [0,127]. The MIDI specification defines note number 60 as Middle C, and all other notes are relative. MIDI devices and software define the mapping between a note and a MIDI note number. If Middle C is arbitrarily assumed to be C5 for the target MIDI hardware or software, the following table maps between MIDI note numbers and notes:

	Note Numbers											
Octave	C	C#	D	D#	E	F	F#	G	G#	A	A#	B
0	0	1	2	3	4	5	6	7	8	9	10	11
1	12	13	14	15	16	17	18	19	20	21	22	23
2	24	25	26	27	28	29	30	31	32	33	34	35
3	36	37	38	39	40	41	42	43	44	45	46	47
4	48	49	50	51	52	53	54	55	56	57	58	59
5	60	61	62	63	64	65	66	67	68	69	70	71
6	72	73	74	75	76	77	78	79	80	81	82	83
7	84	85	86	87	88	89	90	91	92	93	94	95
8	96	97	98	99	100	101	102	103	104	105	106	107
9	108	109	110	111	112	113	114	115	116	117	118	119
10	120	121	122	123	124	125	126	127				

Dependencies

This property is valid only for `NoteOn`, `NoteOff`, and `PolyKeyPressure` `midimsg` objects.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`Velocity` — Velocity of MIDI message
integer in the range [0,127]

Velocity of MIDI message, specified as a scalar integer in the range [0,127]. Velocity describes how fast, or "hard," a note is played. A higher number corresponds to faster velocity.

Dependencies

This property is valid only for `NoteOn` and `NoteOff` `midimsg` objects.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`KeyPressure` — Key pressure
integer in the range [0,127]

Key pressure, specified as a scalar integer in the range [0,127]. Key pressure applies *aftertouch* to an individual note. For example, on a keyboard, key pressure describes the

pressure applied to a key after that key has been struck (after a `NoteOn` message is sent). You can use `KeyPressure` to add expression to held notes.

Dependencies

This property is valid only for `PolyKeyPressure` `midimsg` objects.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`LocalControl` — Enable local control
`true` | `false`

Enable local control, specified as `true` or `false`. When local control is set to `false`, all devices on a given channel respond only to data received over MIDI.

Dependencies

This property is valid only for `LocalControl` `midimsg` objects.

Data Types: `logical`

`MonoChannels` — Channels for `MonoOn` messages
integer in the range [0,16]

Channels for `MonoOn` messages, specified as a scalar integer in the range [0,16].

Dependencies

This property is valid only for `MonoOn` `midimsg` objects.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`CCNumber` — Control change number
integer in the range [0,119]

Control change number, specified as an integer in the range [0,119].

Dependencies

This property is valid only for `ControlChange` `midimsg` objects.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`CCValue` — Control change value

integer in the range [0,127]

Control change value, specified as an integer in the range [0,127].

Dependencies

This property is valid only for `ControlChange` `midimsg` objects.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`Program` — Program number to switch to

integer in the range [0,127]

Program number to switch to, specified as an integer in the range [0,127].

Dependencies

This property is valid only for `ProgramChange` `midimsg` objects.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`ChannelPressure` — Channel pressure

integer in the range [0,127]

Channel pressure, specified as an integer in the range [0,127]. Key pressure applies *aftertouch* to all notes in a channel.

Dependencies

This property is valid only for `ChannelPressure` `midimsg` objects.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`PitchChange` — Amount of pitch change to apply

integer in the range [0,16383]

Amount of pitch change to apply, specified as an integer in the range [0,16383]. The center position (no effect) is `8192`. Sensitivity is a function of the receiver.

Dependencies

This property is valid only for `PitchBend` `midimsg` objects.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

TimeCodeSequence — Sequence number
integer in the range [0,7]

Sequence number, specified as an integer in the range [0,7].

Dependencies

This property is valid only for MIDITimeCodeQuarterFrame midimsg objects.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

TimeCodeValue — Time code value
integer in the range [0,15]

Time code value, specified as an integer in the range [0,15].

Dependencies

This property is valid only for MIDITimeCodeQuarterFrame midimsg objects.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

SongPosition — Position in song to go to
integer in the range [0,16383]

Position in song to go to, specified as an integer in the range [0,16383].

Dependencies

This property is valid only for SongPositionPointer midimsg objects.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

Song — Song number to switch to
integer in the range [0,127]

Song number to switch to, specified as an integer in the range [0,127].

Dependencies

This property is valid only for `SongSelect` `midimsg` objects.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Examples

Create Note Messages

You can create MIDI note messages using the `NoteOn` and `NoteOff` `midimsg` objects. A `NoteOn` message indicates that a note should begin playing. A `NoteOff` message indicates that a note should stop playing. Alternatively, you can send a second `NoteOn` message with velocity set to zero to indicate that the note should stop playing. The Audio System Toolbox® provides a convenience syntax to create pairs of note on and note off messages.

Create a pair of MIDI messages to indicate a Note On and Note Off sequence using the `Note` convenience syntax. Specify that the note starts after one second, and has a duration of two seconds.

```
channel = 1;
note = 60;
velocity = 64;
duration = 2;
timestamp = 1;
msgs = midimsg('Note',channel,note,velocity,duration,timestamp)
```

```
msgs = 
  MIDI message:
    NoteOn          Channel: 1  Note: 60  Velocity: 64  Timestamp: 1  [ 90 3C 40 ]
    NoteOn          Channel: 1  Note: 60  Velocity: 0   Timestamp: 3  [ 90 3C 00 ]
```

Two `midimsg` objects are created and returned as an array. The `Note` syntax returns the note off message as a `NoteOn` `midimsg` object with `Velocity` set to zero.

To create Note On and Note Off messages separately, create two `NoteOn` messages and concatenate them.

```
msgs = [midimsg('NoteOn',channel,note,velocity,timestamp), ...
        midimsg('NoteOn',channel,note,0,3)]
```

```
msgs = 
  MIDI message:
    NoteOn          Channel: 1  Note: 60  Velocity: 64  Timestamp: 1  [ 90 3C 40 ]
    NoteOn          Channel: 1  Note: 60  Velocity: 0   Timestamp: 3  [ 90 3C 00 ]
```

You can also specify the Note Off using a `NoteOff` `midimsg` object. Using the `NoteOff` syntax enables you to specify a release velocity.

```
msgs = [midimsg('NoteOn',channel,note,velocity,timestamp), ...
        midimsg('NoteOff',channel,note,velocity,3)]
```

```
msgs = 
  MIDI message:
    NoteOn          Channel: 1  Note: 60  Velocity: 64  Timestamp: 1  [ 90 3C 40 ]
    NoteOff         Channel: 1  Note: 60  Velocity: 64  Timestamp: 3  [ 80 3C 40 ]
```

Control Change Messages for Control Surfaces

To create a control change message, specify the `midimsg` `Type` as `ControlChange` and set the required parameters: `Channel`, `CCNumber`, and `CCValue`. To determine the channel and control number assigned to your MIDI control surface, use `midiid`. Enter `midiid` at the Command Prompt and then move the control you want to identify.

```
[ccInfo,deviceName] = midiid;
```

```
Move the control you wish to identify; type ^C to abort.
Waiting for control message... done
```

`midiid` returns the control change number and channel as a single number according to the following formula: `ccInfo = (Channel*1000 + CCNumber)`. Define a MIDI Control Change message to move the identified controller. Your MIDI Control Surface must be bidirectional to receive Control Change messages.

```
channel = floor(ccInfo/1000);
ccnumber = ccInfo - channel*1000;
ccvalue = 1;
msg = midimsg('ControlChange',channel,ccnumber,ccvalue)
```

```
msg = 
  MIDI message:
    ControlChange   Channel: 1  CCNumber: 16  CCValue: 1   Timestamp: 0  [ B0 10 01 ]
```

Create a `mididevice` object using the `deviceName` identified using `midiid`. Send the MIDI message to your device.

```
device = mididevice(deviceName);
midisend(device,msg);
```

Create a Program Change Message

Program Change messages, sometimes called "patch change" messages, specify how notes are interpreted. For example, a Program Change message can specify the instrument being played. To create a `ProgramChange` `midimsg` object, specify the `midimsg` type as `ProgramChange`, and the required property values: `Channel` and `Program`.

```
channel = 4;
program = 7;
msg = midimsg('ProgramChange',channel,program)
```

```
msg =
  MIDI message:
    ProgramChange   Channel: 4  Program: 7   Timestamp: 0  [ C3 07 ]
```

Create a System Exclusive Message

System Exclusive messages are defined by a sequence of `midimsg` objects: `SystemExclusive`, `Data`, and `EOX`. To create a System Exclusive sequence, specify the `SystemExclusive` `midimsg` type during creation and then specify the bytes of the message. This syntax requires a timestamp.

```
bytes = [0 1 2];
timestamp = 0;
msg = midimsg('SystemExclusive',bytes,timestamp)
```

```
msg =
  MIDI message:
    SystemExclusive Timestamp: 0  [ F0 ]
    Data            Timestamp: 0  [ 00 01 02 ]
    EOX             Timestamp: 0  [ F7 ]
```

You can also create the `SystemExclusive`, `Data`, and `EOX` `midimsg` objects individually. For example, the following `midimsg` array is the same as the preceding.

```
msg = [midimsg('SystemExclusive',timestamp), ...
       midimsg('Data',bytes,timestamp), ...
       midimsg('EOX',timestamp)]
```

```
msg = 
  MIDI message:
    SystemExclusive Timestamp: 0  [ F0 ]
    Data            Timestamp: 0  [ 00 01 02 ]
    EOX             Timestamp: 0  [ F7 ]
```

Create a Scalar Default MIDI Message

The default MIDI message is a scalar with all zero bytes, and `Type` is `Data`.

```
msg = midimsg
```

```
msg = 
  MIDI message:
    Data            Timestamp: 0  [ 00 00 00 00 00 00 00 00 ]
```

Preallocate Array of MIDI Messages

You can create a MIDI message array by specifying the size by a scalar or row vector.

If you specify the size as a scalar *M*, `midimsg` returns an *M*-by-*M* array with all zero bytes.

```
msg = midimsg(2)
```

```
msg = 
  MIDI message:
    Data            Timestamp: 0  [ 00 00 00 00 00 00 00 00 ]
    Data            Timestamp: 0  [ 00 00 00 00 00 00 00 00 ]
    Data            Timestamp: 0  [ 00 00 00 00 00 00 00 00 ]
    Data            Timestamp: 0  [ 00 00 00 00 00 00 00 00 ]
```

An array of MIDI messages is always displayed vertically in order of their linear indexing. You can refer to individual elements of the array by specifying its position in each

dimension, or by its linear index. For example, change the `Timestamp` of the third element from `0` to `2 using linear indexing`, and then from `2` to `3` using first dimensional indexing.

```
msg(3).Timestamp = 2
```

```
msg =
  MIDI message:
    Data           Timestamp: 0  [ 00 00 00 00 00 00 00 00 ]
    Data           Timestamp: 0  [ 00 00 00 00 00 00 00 00 ]
    Data           Timestamp: 2  [ 00 00 00 00 00 00 00 00 ]
    Data           Timestamp: 0  [ 00 00 00 00 00 00 00 00 ]
```

```
msg(1,2).Timestamp = 3
```

```
msg =
  MIDI message:
    Data           Timestamp: 0  [ 00 00 00 00 00 00 00 00 ]
    Data           Timestamp: 0  [ 00 00 00 00 00 00 00 00 ]
    Data           Timestamp: 3  [ 00 00 00 00 00 00 00 00 ]
    Data           Timestamp: 0  [ 00 00 00 00 00 00 00 00 ]
```

You can also specify nonsymmetric arrays. If you specify the size as a row vector of two or more elements, `midimsg` returns an *M*-by-*N*-by-...-*X* multidimensional array. For example, to specify a three dimensional array with each dimension having a different number of elements, specify the size as a row vector of three elements.

```
msg = midimsg([2,1,3])
```

```
msg =
  MIDI message:
    Data           Timestamp: 0  [ 00 00 00 00 00 00 00 00 ]
    Data           Timestamp: 0  [ 00 00 00 00 00 00 00 00 ]
    Data           Timestamp: 0  [ 00 00 00 00 00 00 00 00 ]
    Data           Timestamp: 0  [ 00 00 00 00 00 00 00 00 ]
    Data           Timestamp: 0  [ 00 00 00 00 00 00 00 00 ]
    Data           Timestamp: 0  [ 00 00 00 00 00 00 00 00 ]
```

```
size(msg)
```

```
ans = 1×3

     2     1     3
```

Create Empty MIDI Message

```
msg = midimsg(0)
```

```
msg =

  empty MIDI message array
```

Manipulate Array of MIDI Messages

In this example, you create an array of MIDI messages, and then index into the array in a loop to define a melody.

Create a 22-by-1 array of MIDI messages with all zero data.

```
msgArray = midimsg([22,1]);
```

To create a melody, create MIDI `NoteOn` and `NoteOff` messages by indexing in a loop. Display the result.

```
melody = [60,65,60,57,55,53,60,65,60,67,60];
for i = 1:numel(melody)
    idx = (2*i-1):(2*i);
    msgArray(idx) = midimsg('Note',1,melody(i),50,0.5,i);
end
msgArray
```

```
msgArray =
  MIDI message:
    NoteOn          Channel: 1  Note: 60  Velocity: 50  Timestamp: 1  [ 90 3C 32 ]
    NoteOn          Channel: 1  Note: 60  Velocity: 0   Timestamp: 1.5  [ 90 3C 00 ]
    NoteOn          Channel: 1  Note: 65  Velocity: 50  Timestamp: 2  [ 90 41 32 ]
    NoteOn          Channel: 1  Note: 65  Velocity: 0   Timestamp: 2.5  [ 90 41 00 ]
    NoteOn          Channel: 1  Note: 60  Velocity: 50  Timestamp: 3  [ 90 3C 32 ]
    NoteOn          Channel: 1  Note: 60  Velocity: 0   Timestamp: 3.5  [ 90 3C 00 ]
    NoteOn          Channel: 1  Note: 57  Velocity: 50  Timestamp: 4  [ 90 39 32 ]
    NoteOn          Channel: 1  Note: 57  Velocity: 0   Timestamp: 4.5  [ 90 39 00 ]
    NoteOn          Channel: 1  Note: 55  Velocity: 50  Timestamp: 5  [ 90 37 32 ]
    NoteOn          Channel: 1  Note: 55  Velocity: 0   Timestamp: 5.5  [ 90 37 00 ]
    NoteOn          Channel: 1  Note: 53  Velocity: 50  Timestamp: 6  [ 90 35 32 ]
```

```
    NoteOn          Channel: 1  Note: 53  Velocity: 0   Timestamp: 6.5  [ 90 35 00 ]
    NoteOn          Channel: 1  Note: 60  Velocity: 50  Timestamp: 7  [ 90 3C 32 ]
    NoteOn          Channel: 1  Note: 60  Velocity: 0   Timestamp: 7.5  [ 90 3C 00 ]
    NoteOn          Channel: 1  Note: 65  Velocity: 50  Timestamp: 8  [ 90 41 32 ]
    NoteOn          Channel: 1  Note: 65  Velocity: 0   Timestamp: 8.5  [ 90 41 00 ]
    NoteOn          Channel: 1  Note: 60  Velocity: 50  Timestamp: 9  [ 90 3C 32 ]
    NoteOn          Channel: 1  Note: 60  Velocity: 0   Timestamp: 9.5  [ 90 3C 00 ]
    NoteOn          Channel: 1  Note: 67  Velocity: 50  Timestamp: 10  [ 90 43 32 ]
    NoteOn          Channel: 1  Note: 67  Velocity: 0   Timestamp: 10.5  [ 90 43 00 ]
    NoteOn          Channel: 1  Note: 60  Velocity: 50  Timestamp: 11  [ 90 3C 32 ]
    NoteOn          Channel: 1  Note: 60  Velocity: 0   Timestamp: 11.5  [ 90 3C 00 ]
```

The order of the MIDI messages in the array is only important for readability. When you send MIDI messages using a `mididevice` object, the `mididevice` object reorders your MIDI messages according to their timestamps and sends them in chronological order.

Create a `PitchBend` MIDI message to bend the fourth note downward and add it to the MIDI message array. For readability, sort the MIDI message array by `Timestamp`.

```
msg = midimsg('PitchBend',1,7192,4.01);
msgArray = [msgArray;msg]
```

```
msgArray =
  MIDI message:
    NoteOn          Channel: 1  Note: 60  Velocity: 50  Timestamp: 1  [ 90 3C 32 ]
    NoteOn          Channel: 1  Note: 60  Velocity: 0   Timestamp: 1.5  [ 90 3C 00 ]
    NoteOn          Channel: 1  Note: 65  Velocity: 50  Timestamp: 2  [ 90 41 32 ]
    NoteOn          Channel: 1  Note: 65  Velocity: 0   Timestamp: 2.5  [ 90 41 00 ]
    NoteOn          Channel: 1  Note: 60  Velocity: 50  Timestamp: 3  [ 90 3C 32 ]
    NoteOn          Channel: 1  Note: 60  Velocity: 0   Timestamp: 3.5  [ 90 3C 00 ]
    NoteOn          Channel: 1  Note: 57  Velocity: 50  Timestamp: 4  [ 90 39 32 ]
    NoteOn          Channel: 1  Note: 57  Velocity: 0   Timestamp: 4.5  [ 90 39 00 ]
    NoteOn          Channel: 1  Note: 55  Velocity: 50  Timestamp: 5  [ 90 37 32 ]
    NoteOn          Channel: 1  Note: 55  Velocity: 0   Timestamp: 5.5  [ 90 37 00 ]
    NoteOn          Channel: 1  Note: 53  Velocity: 50  Timestamp: 6  [ 90 35 32 ]
    NoteOn          Channel: 1  Note: 53  Velocity: 0   Timestamp: 6.5  [ 90 35 00 ]
    NoteOn          Channel: 1  Note: 60  Velocity: 50  Timestamp: 7  [ 90 3C 32 ]
    NoteOn          Channel: 1  Note: 60  Velocity: 0   Timestamp: 7.5  [ 90 3C 00 ]
    NoteOn          Channel: 1  Note: 65  Velocity: 50  Timestamp: 8  [ 90 41 32 ]
    NoteOn          Channel: 1  Note: 65  Velocity: 0   Timestamp: 8.5  [ 90 41 00 ]
    NoteOn          Channel: 1  Note: 60  Velocity: 50  Timestamp: 9  [ 90 3C 32 ]
    NoteOn          Channel: 1  Note: 60  Velocity: 0   Timestamp: 9.5  [ 90 3C 00 ]
    NoteOn          Channel: 1  Note: 67  Velocity: 50  Timestamp: 10  [ 90 43 32 ]
    NoteOn          Channel: 1  Note: 67  Velocity: 0   Timestamp: 10.5  [ 90 43 00 ]
    NoteOn          Channel: 1  Note: 60  Velocity: 50  Timestamp: 11  [ 90 3C 32 ]
```

```
    NoteOn          Channel: 1  Note: 60  Velocity: 0   Timestamp: 11.5  [ 90 3C 00 ]
    PitchBend       Channel: 1  PitchChange: 7192 Timestamp: 4.01  [ E0 18 38 ]

timeStamps = [msgArray.Timestamp];
[~,idx] = sort(timeStamps);

msgArray = msgArray(idx)

msgArray =
  MIDI message:
    NoteOn          Channel: 1  Note: 60  Velocity: 50  Timestamp: 1  [ 90 3C 32 ]
    NoteOn          Channel: 1  Note: 60  Velocity: 0   Timestamp: 1.5  [ 90 3C 00 ]
    NoteOn          Channel: 1  Note: 65  Velocity: 50  Timestamp: 2  [ 90 41 32 ]
    NoteOn          Channel: 1  Note: 65  Velocity: 0   Timestamp: 2.5  [ 90 41 00 ]
    NoteOn          Channel: 1  Note: 60  Velocity: 50  Timestamp: 3  [ 90 3C 32 ]
    NoteOn          Channel: 1  Note: 60  Velocity: 0   Timestamp: 3.5  [ 90 3C 00 ]
    NoteOn          Channel: 1  Note: 57  Velocity: 50  Timestamp: 4  [ 90 39 32 ]
    PitchBend       Channel: 1  PitchChange: 7192 Timestamp: 4.01  [ E0 18 38 ]
    NoteOn          Channel: 1  Note: 57  Velocity: 0   Timestamp: 4.5  [ 90 39 00 ]
    NoteOn          Channel: 1  Note: 55  Velocity: 50  Timestamp: 5  [ 90 37 32 ]
    NoteOn          Channel: 1  Note: 55  Velocity: 0   Timestamp: 5.5  [ 90 37 00 ]
    NoteOn          Channel: 1  Note: 53  Velocity: 50  Timestamp: 6  [ 90 35 32 ]
    NoteOn          Channel: 1  Note: 53  Velocity: 0   Timestamp: 6.5  [ 90 35 00 ]
    NoteOn          Channel: 1  Note: 60  Velocity: 50  Timestamp: 7  [ 90 3C 32 ]
    NoteOn          Channel: 1  Note: 60  Velocity: 0   Timestamp: 7.5  [ 90 3C 00 ]
    NoteOn          Channel: 1  Note: 65  Velocity: 50  Timestamp: 8  [ 90 41 32 ]
    NoteOn          Channel: 1  Note: 65  Velocity: 0   Timestamp: 8.5  [ 90 41 00 ]
    NoteOn          Channel: 1  Note: 60  Velocity: 50  Timestamp: 9  [ 90 3C 32 ]
    NoteOn          Channel: 1  Note: 60  Velocity: 0   Timestamp: 9.5  [ 90 3C 00 ]
    NoteOn          Channel: 1  Note: 67  Velocity: 50  Timestamp: 10  [ 90 43 32 ]
    NoteOn          Channel: 1  Note: 67  Velocity: 0   Timestamp: 10.5  [ 90 43 00 ]
    NoteOn          Channel: 1  Note: 60  Velocity: 50  Timestamp: 11  [ 90 3C 32 ]
    NoteOn          Channel: 1  Note: 60  Velocity: 0   Timestamp: 11.5  [ 90 3C 00 ]
```

See Also

`mididevice` | `midireceive` | `midisend`

Topics

"MIDI Device Interface"

External Websites

MIDI Manufacturers Association

Introduced in R2018a

mididevice

Send and receive MIDI messages

Description

Interface to a MIDI device in MATLAB using `mididevice`. Package MIDI messages using `midimsg`. Send and receive messages using `midisend` and `midireceive`. Use `mididevinfo` to query your system for available MIDI devices.

For a tutorial on interfacing with MIDI devices, see “MIDI Device Interface”.

Creation

Syntax

```
device = mididevice(deviceNameOrID)
device = mididevice('Input',inDeviceNameOrID)
device = mididevice('Output',outDeviceNameOrID)
device = mididevice('Input',inDeviceNameOrID,
'Output',outDeviceNameOrID)
```

Description

`device = mididevice(deviceNameOrID)` returns an interface to the MIDI device specified by `deviceNameOrID`. If the MIDI device supports MIDI in and MIDI out, then `device` also supports MIDI in and MIDI out.

`device = mididevice('Input',inDeviceNameOrID)` returns an input interface to the MIDI input device, `inDeviceNameOrID`.

`device = mididevice('Output',outDeviceNameOrID)` returns an output interface to the MIDI output device, `outDeviceNameOrID`.

`device = mididevice('Input',inDeviceNameOrID, 'Output',outDeviceNameOrID)` returns a MIDI I/O interface, where input is received from `inDeviceNameOrID` and output is sent to `outDeviceNameOrID`.

Properties

`Input` — Input device name associated with `mididevice`
empty `char` array (default)

This property is read-only.

Input device name attached to your `mididevice` object, returned as a character array.

`Input` is set during the creation of the `mididevice` object and cannot be modified later.

Data Types: `char`

`Output` — Output device name associated with `mididevice`
empty `char` array (default)

This property is read-only.

Output device name attached to your `mididevice` object, returned as a character array

`Output` is set during the creation of the `mididevice` object and cannot be modified later.

Data Types: `char`

`InputID` — Input device ID associated with `mididevice`
`-1` (default)

This property is read-only.

Unique MIDI input device ID attached to your `midídevice` object, returned as a scalar double. If your system includes different MIDI devices with the same name, using the device ID removes ambiguity.

`InputID` is set during the creation of the `mididevice` object and cannot be modified later.

Data Types: `double`

`OutputID` — Output device name associated with `mididevice`
`-1` (default)

This property is read-only.

Unique MIDI output device ID attached to your `mididevice` object, returned as a scalar double. If your system includes different MIDI devices with the same name, using the device ID removes ambiguity.

`OutputID` is set during the creation of the `mididevice` object and cannot be modified later.

Data Types: `double`

Object Functions

midisend	Send MIDI message to MIDI device
midireceive	Receive MIDI message from MIDI device
hasdata	Determine if data is available to read from MIDI device

Examples

Connect Input and Output to Single MIDI Device

Query your system for available MIDI devices.

```
mididevinfo
```

```
  MIDI devices available:
  ID  Direction  Interface   Name
```

```
  0   output    MMSystem   'Microsoft MIDI Mapper'
  1    input    MMSystem   'USB MIDI Interface '
  2   output    MMSystem   'Microsoft GS Wavetable Synth'
  3   output    MMSystem   'USB MIDI Interface '
```

Create a MIDI device object to interface with your selected device. If you specify a single MIDI device object, and it is capable of both input and output, `mididevice` connects to both the input and output.

```
device = mididevice('USB MIDI Interface ')
```

```
device = 
  mididevice connected to
     Input: 'USB MIDI Interface ' (1)
    Output: 'USB MIDI Interface ' (3)
```

Connect Input to MIDI Device

Query your system for MIDI devices.

```
mididevinfo
```

```
  MIDI devices available:
  ID  Direction  Interface   Name
   0   output    MMSystem   'Microsoft MIDI Mapper'
   1    input    MMSystem   'USB MIDI Interface '
   2   output    MMSystem   'Microsoft GS Wavetable Synth'
   3   output    MMSystem   'USB MIDI Interface '
```

Create a MIDI device object to interface with your selected input device. As soon as you create the MIDI device object, it begins listening for MIDI messages and storing them in a buffer.

```
device = mididevice('Input','USB MIDI Interface ');
```

Connect Output to MIDI Device

Query your system for available MIDI devices.

```
mididevinfo
```

```
  MIDI devices available:
  ID  Direction  Interface   Name
   0   output    MMSystem   'Microsoft MIDI Mapper'
   1    input    MMSystem   'USB MIDI Interface '
   2   output    MMSystem   'Microsoft GS Wavetable Synth'
   3   output    MMSystem   'USB MIDI Interface '
```

Create a MIDI device object to interface with your selected output device.

```
device = mididevice('Output','USB MIDI Interface ')
```

```
device =
  mididevice connected to
    Output: 'USB MIDI Interface ' (3)
```

Connect Input and Output to Different MIDI Devices

Query your system for available MIDI devices.

```
mididevinfo
```

```
  MIDI devices available:
  ID  Direction  Interface   Name
   0   output    MMSystem   'Microsoft MIDI Mapper'
   1    input    MMSystem   'USB MIDI Interface '
   2   output    MMSystem   'Microsoft GS Wavetable Synth'
   3   output    MMSystem   'USB MIDI Interface '
```

Create a MIDI device object that receives data from one device and sends data to another device. In this example, the MIDI device object receives MIDI messages from the `'USB MIDI Interface '` device and sends data to the `'Microsoft GS Wavetable Synth'` virtual output device. To avoid ambiguity, the MIDI devices are specified by the device IDs.

```
device = mididevice('Input',1,'Output',2)
```

```
device =
  mididevice connected to
     Input: 'USB MIDI Interface ' (1)
    Output: 'Microsoft GS Wavetable Synth' (2)
```

See Also

`mididevinfo` | `midimsg` | `midireceive` | `midisend`

Topics

"MIDI Device Interface"

External Websites

MIDI Manufacturers Association

Introduced in R2018a

hasdata

Determine if data is available to read from MIDI device

Syntax

`tf = hasdata(device)`

Description

`tf = hasdata(device)` returns logical `1` (`true`) if there is data available to read from the `mididevice` specified by `device`. Otherwise, it returns logical `0` (`false`).

Examples

Determine if Data Is Available to Receive

Create a `mididevice` object to interface with your MIDI device. Query your system for available MIDI devices.

```
mididevinfo
```

```
  MIDI devices available:
  ID  Direction  Interface   Name
   0   output    MMSystem   'Microsoft MIDI Mapper'
   1    input    MMSystem   'nanoKONTROL2'
   2    input    MMSystem   'USB Uno MIDI Interface'
   3   output    MMSystem   'Microsoft GS Wavetable Synth'
   4   output    MMSystem   'nanoKONTROL2'
   5   output    MMSystem   'USB Uno MIDI Interface'
```

```
device = mididevice('USB Uno MIDI Interface')
```

```
device =
  mididevice connected to
     Input: 'USB Uno MIDI Interface' (2)
    Output: 'USB Uno MIDI Interface' (5)
```

As soon as your `mididevice` object is created, it begins listening for MIDI messages and storing them in a buffer. When you call `midireceive`, MIDI messages are retrieved from the buffer and returned. You can use `hasdata` to query whether your `mididevice` object buffer contains unread MIDI messages.

```
hasdata(device)

ans = logical
   0
```

Input Arguments

device — mididevice object
`mididevice` object

Specify `device` as an object created by `mididevice`.

See Also

`mididevice` | `midideviceinfo` | `midimsg` | `midisend`

Topics

"MIDI Device Interface"

External Websites

MIDI Manufacturers Association

Introduced in R2018a

midireceive

Receive MIDI message from MIDI device

Syntax

```
msgs = midireceive(device)
msgs = midireceive(device,maxmsgs)
```

Description

`msgs = midireceive(device)` returns the MIDI messages, `msgs`, received from a MIDI device using the MIDI device interface, `device`.

`msgs = midireceive(device,maxmsgs)` specifies the maximum number of MIDI messages to return as `maxmsgs`.

Examples

Receive MIDI Messages

To determine what MIDI devices are attached to your MIDI input ports, call `mididevinfo`. Use the `availableDevices` struct to specify a valid MIDI device to create a `mididevice` object.

```
availableDevices = mididevinfo;
device = mididevice(availableDevices.input(1).ID);
```

Once your MIDI `device` object is created, it begins listening to MIDI messages from your specified device and storing them in a buffer. To get all MIDI messages in the buffer, call `midireceive`. In this example, several keys on a MIDI keyboard are played.

```
msgs = midireceive(device)
```

```
msgs =
```

```
  MIDI message:
    NoteOn         Channel: 1  Note: 52  Velocity: 64  Timestamp: 3.94  [ 90 34 40 ]
    NoteOn         Channel: 1  Note: 52  Velocity: 0   Timestamp: 4.179  [ 90 34 00 ]
    NoteOn         Channel: 1  Note: 48  Velocity: 64  Timestamp: 4.19  [ 90 30 40 ]
    NoteOn         Channel: 1  Note: 47  Velocity: 64  Timestamp: 4.382  [ 90 2F 40 ]
    NoteOn         Channel: 1  Note: 48  Velocity: 0   Timestamp: 4.459  [ 90 30 00 ]
    NoteOn         Channel: 1  Note: 48  Velocity: 64  Timestamp: 4.59  [ 90 30 40 ]
    NoteOn         Channel: 1  Note: 47  Velocity: 0   Timestamp: 4.776  [ 90 2F 00 ]
    NoteOn         Channel: 1  Note: 50  Velocity: 64  Timestamp: 4.788  [ 90 32 40 ]
    NoteOn         Channel: 1  Note: 47  Velocity: 64  Timestamp: 4.802  [ 90 2F 40 ]
    NoteOn         Channel: 1  Note: 52  Velocity: 64  Timestamp: 4.831  [ 90 34 40 ]
    NoteOn         Channel: 1  Note: 47  Velocity: 0   Timestamp: 4.84  [ 90 2F 00 ]
    NoteOn         Channel: 1  Note: 48  Velocity: 0   Timestamp: 4.912  [ 90 30 00 ]
    NoteOn         Channel: 1  Note: 52  Velocity: 0   Timestamp: 4.953  [ 90 34 00 ]
    NoteOn         Channel: 1  Note: 50  Velocity: 0   Timestamp: 5.079  [ 90 32 00 ]
```

Reading from the buffer clears the data. For example, if no more MIDI messages are sent, and the buffer is reread, `midireceive` returns an empty MIDI message.

```
msgs = midireceive(device)
```

```
msgs = 

  empty MIDI message array
```

Receive Limited Number of MIDI Messages

Query your system for available output from MIDI devices. Specify that the output of a MIDI device is connected to the input of your `mididevice` object.

```
mididevinfo
```

```
  MIDI devices available:
  ID  Direction  Interface   Name
   0   output    MMSystem   'Microsoft MIDI Mapper'
   1    input    MMSystem   'USB MIDI Interface '
   2   output    MMSystem   'Microsoft GS Wavetable Synth'
   3   output    MMSystem   'USB MIDI Interface '
```

```
device = mididevice('Input','USB MIDI Interface ');
```

Once your MIDI `device` object is created, it begins listening to MIDI messages from your specified device and storing them in a buffer. To get a limited number of MIDI messages from the buffer, call `midireceive` and specify the maximum number of messages to return. In this example, five keys are played on a MIDI device. A maximum of four MIDI messages are received at each call to `midireceive`.

```
midireceive(device,4)

ans = 

  MIDI message:
    NoteOn          Channel: 1  Note: 36  Velocity: 64  Timestamp: 2929.71  [ 90 24 40 ]
    NoteOn          Channel: 1  Note: 36  Velocity: 0   Timestamp: 2929.91  [ 90 24 00 ]
    NoteOn          Channel: 1  Note: 37  Velocity: 64  Timestamp: 2930.43  [ 90 25 40 ]
    NoteOn          Channel: 1  Note: 37  Velocity: 0   Timestamp: 2930.59  [ 90 25 00 ]

midireceive(device,4)

ans = 

  MIDI message:
    NoteOn          Channel: 1  Note: 38  Velocity: 64  Timestamp: 2931.16  [ 90 26 40 ]
    NoteOn          Channel: 1  Note: 38  Velocity: 0   Timestamp: 2931.32  [ 90 26 00 ]
    NoteOn          Channel: 1  Note: 39  Velocity: 64  Timestamp: 2931.87  [ 90 27 40 ]
    NoteOn          Channel: 1  Note: 39  Velocity: 0   Timestamp: 2932.01  [ 90 27 00 ]

midireceive(device,4)

ans = 

  MIDI message:
    NoteOn          Channel: 1  Note: 40  Velocity: 64  Timestamp: 2932.52  [ 90 28 40 ]
    NoteOn          Channel: 1  Note: 40  Velocity: 0   Timestamp: 2932.66  [ 90 28 00 ]
```

Input Arguments

device — Object of mididevice
object of mididevice

Specify device as an object created by mididevice.

maxmsgs — Maximum number of messages to return
positive integer scalar

Maximum number of messages to return, specified as a positive integer scalar.

Data Types: double

Output Arguments

msgs — Object of midimsg
scalar | column vector

Object of `midimsg`, returned as a scalar or column vector. The number of MIDI messages in the `mididevice` buffer and `maxmsgs` determine the size of `msgs`.

See Also

`mididevice` | `mididevinfo` | `midimsg` | `midisend`

Topics

"MIDI Device Interface"

External Websites

MIDI Manufacturers Association

Introduced in R2018a

midisend

Send MIDI message to MIDI device

Syntax

```
midisend(device,msg)
midisend(device,varargin)
```

Description

`midisend(device,msg)` sends the MIDI message, `msg`, to a MIDI device using the MIDI device interface, `device`.

`midisend(device,varargin)` creates MIDI messages using `varargin` and then sends the MIDI messages. The `varargin` syntax is for convenience and includes a call to `midimsg` with the call to `midisend`.

Examples

Send MIDI Messages to Device

Query your system for available MIDI device output ports. Use the `availableDevices` struct to specify a valid MIDI device and create a `mididevice` object.

```
availableDevices = mididevinfo;
device = mididevice(availableDevices.output(2).ID);
```

Create a pair of `NoteOn` messages (to indicate Note On and Note Off) and send them to your selected MIDI device.

```
msgs = midimsg('Note',1,48,64,0.25);
midisend(device,msgs)
```

Define and Send MIDI Messages to Device

`midisend` enables you to combine the definition and sending of a `midimsg` into a single function call. Send middle C on channel 3 with velocity 64.

```
mididevinfo
```

```
  MIDI devices available:
  ID  Direction  Interface   Name
   0   output    MMSystem   'Microsoft MIDI Mapper'
   1    input    MMSystem   'nanoKONTROL2'
   2    input    MMSystem   'USB Uno MIDI Interface'
   3   output    MMSystem   'Microsoft GS Wavetable Synth'
   4   output    MMSystem   'nanoKONTROL2'
   5   output    MMSystem   'USB Uno MIDI Interface'
```

```
device = mididevice('USB Uno MIDI Interface')
```

```
device = 
  mididevice connected to
     Input: 'USB Uno MIDI Interface' (2)
    Output: 'USB Uno MIDI Interface' (5)
```

```
midisend(device,'NoteOn',3,60,64)
```

Input Arguments

device — Object of `mididevice`

scalar

Specify `device` as an object created by `mididevice`.

msg — Object of `midimsg`

scalar | vector | array

Specify `msg` as an object created by `midimsg`.

varargin — Variable number of arguments describing MIDI message

`midimsg` input arguments

Specify `varargin` as a valid combination of arguments that can construct a MIDI message. See `midimsg` for a description of valid arguments.

See Also
`mididevice` | `mididevinfo` | `midimsg` | `midireceive`

Topics
"MIDI Device Interface"

External Websites
MIDI Manufacturers Association

Introduced in R2018a

audioPlugin class

Base class for audio plugins

Description

`audioPlugin` is the base class for audio plugins. In your class definition file, you must subclass your object from this base class or from the `audioPluginSource` class, which inherits from `audioPlugin`. Subclassing enables you to inherit the attributes necessary to generate plugins and access Audio System Toolbox functionality.

To inherit from the `audioPlugin` base class directly, type this syntax as the first line of your class definition file:

```
classdef myAudioPlugin < audioPlugin
```

`myAudioPlugin` is the name of your object.

For a tutorial on designing audio plugins, see “Design an Audio Plugin”.

Methods

getSampleRate	Get sample rate at which the plugin is run
setSampleRate	Set sample rate at which the plugin is run

Copy Semantics

Handle. To learn how handle classes affect copy operations, see “Object Behavior” (MATLAB) in the MATLAB documentation.

Examples

Design Valid Audio Plugin

Design a valid basic audio plugin class

Terminology:

- A valid audio plugin is one that can be deployed in a digital audio workstation (DAW) environment. To validate it, use the `validateAudioPlugin` function. To generate it, use the `generateAudioPlugin` function.
- A basic audio plugin inherits from the `audioPlugin` class but not the `matlab.System` class.

Define a basic audio plugin class that inherits from `audioPlugin`.

```
classdef myAudioPlugin < audioPlugin
end
```

Add a processing function to your plugin class.

All valid audio plugins include a processing function. For basic audio plugins, the processing function is named `process`. The processing function is where audio processing occurs. It always has an output.

```
classdef myAudioPlugin < audioPlugin
    methods
        function out = process(~,in)
            out = in;
        end
    end
end
```

Design Valid Audio Plugin That Uses getSampleRate

Design an `audioPlugin` class that uses the `getSampleRate` method to get the sample rate at which the plugin is run. The plugin in this example, `simpleStrobe`, uses the sample rate to determine a constant 50 ms strobe period.

```
classdef simpleStrobe < audioPlugin
    % simpleStrobe Add audio strobe effect
    %   Add a strobe effect by gain switching between 0 and 1 in
    %   50 ms increments.  Although the input sample rate can change,
    %   the strobe period remains constant.
```

```
%
%   simpleStrobe properties:
%   period    - Number of samples between gain switches
%   gain    - Gain multiplier, one or zero
%   count    - Number of samples since last gain switch
%
%
%   simpleStrobe methods:
%   process    - Multiply input frame by gain, element by element
%   reset    - Reset count and gain to initial conditions
%              and get sample rate

properties
    Period = 44100*0.05;
    Gain = 1;
end
properties (Access = private)
    Count = 1;
end
methods
    function out = process(plugin,in)
        for i = 1:size(in,1)
            if plugin.Count == plugin.Period
                plugin.Gain = 1 - plugin.Gain;
                plugin.Count = 1;
            end
            in(i,:) = in(i,:)*plugin.Gain;
            plugin.Count = plugin.Count + 1;
        end
        out = in;
    end
    function reset(plugin)
        plugin.Period = floor( getSampleRate(plugin)*0.05 );
        plugin.Count = 1;
        plugin.Gain = 1;
    end
```

```
    end
end
```

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

Classes

audioPluginSource

Functions

audioPluginInterface | audioPluginParameter | generateAudioPlugin | validateAudioPlugin

Topics

"Design an Audio Plugin"
"Audio Plugin Example Gallery"
"Hierarchies of Classes — Concepts" (MATLAB)

Introduced in R2016a

getSampleRate

Class: audioPlugin

Get sample rate at which the plugin is run

Syntax

```
sampleRate = getSampleRate(myAudioPlugin)
```

Description

`sampleRate = getSampleRate(myAudioPlugin)` returns the sample rate in Hz at which the plugin is being run.

- In a digital audio workstation (DAW) environment, the DAW user sets the sample rate. `getSampleRate` interacts with the DAW to determine the sample rate.
- In the MATLAB environment, `getSampleRate` returns the value set by a previous call to `setSampleRate`. If `setSampleRate` has not been called, `getSampleRate` returns the default value, `44100`.

Introduced in R2016a

setSampleRate

Class: audioPlugin

Set sample rate at which the plugin is run

Syntax

```
setSampleRate(myAudioPlugin,sampleRate)
```

Description

`setSampleRate(myAudioPlugin,sampleRate)` sets the sample rate of the plugin, `myAudioPlugin`, to the value specified by `sampleRate`. Specify `sampleRate` as a positive real integer. `setSampleRate` enables the MATLAB environment to mimic behavior in a digital audio workstation (DAW) environment.

Note A plugin must not call `setSampleRate` on itself. If the plugin attempts to call `setSampleRate` on itself, `generateAudioPlugin` throws an error.

Introduced in R2016a

audioPluginSource class

Base class for audio source plugins

Description

`audioPluginSource` is the base class for audio source plugins. Use audio source plugins to produce audio signals.

To create a valid audio source plugin, in your class definition file, subclass your object from the `audioPluginSource` class. Subclassing enables you to inherit the attributes necessary to generate audio source plugins and access Audio System Toolbox functionality. To inherit from the `audioPluginSource` base class directly, type this syntax as the first line of your class definition file:

```
classdef myAudioSourcePlugin < audioPluginSource
```

`myAudioSourcePlugin` is the name of your object.

Methods

getSamplesPerFrame	Get frame size returned by the plugin
setSamplesPerFrame	Set frame size returned by the plugin (MATLAB environment only)

Inherited Methods

getSampleRate	Get sample rate at which the plugin is run
setSampleRate	Set sample rate at which the plugin is run

Copy Semantics

Handle. To learn how handle classes affect copy operations, see “Object Behavior” (MATLAB) in the MATLAB documentation.

Examples

Design Valid Audio Plugin

Design a valid basic audio source plugin class

Terminology:

- A valid audio source plugin is one that can be deployed in a digital audio workstation (DAW) environment. To validate it, use the `validateAudioPlugin` function. To generate it, use the `generateAudioPlugin` function.
- A basic audio source plugin inherits from the `audioPluginSource` class but not the `matlab.System` class.

Define a basic audio source plugin class that inherits from `audioPluginSource`.

```
classdef myAudioSourcePlugin < audioPluginSource
end
```

Add a processing function to your audio source plugin class.

All valid audio source plugins include a processing function. For basic audio source plugins, the processing function is named `process`. The processing function defines the audio signal that your plugin outputs. Audio source plugins do not accept audio signals as input to the processing function.

The default audio plugin interface assumes a stereo output. Specify the processing output as a matrix with two columns. These columns correspond to the left and right channels of a stereo signal. The number of rows in the output matrix correspond to the frame size.

The output frame size must match the frame size of the environment in which the plugin is run. A DAW environment has variable frame size. To determine the current environment frame size, call `getSamplesPerFrame` in the `process` function.

```
classdef myAudioSourcePlugin < audioPluginSource
    methods
        function out = process(plugin)
            out = 0.5*randn(getSamplesPerFrame(plugin),2);
        end
    end
end
```

myAudioSourcePlugin generates a Gaussian white noise audio signal with 0.5 standard deviation.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

Classes
audioPlugin

Functions
audioPluginInterface | audioPluginParameter | generateAudioPlugin | validateAudioPlugin

Topics

"Design an Audio Plugin"
"Audio Plugin Example Gallery"
"Hierarchies of Classes — Concepts" (MATLAB)

Introduced in R2016a

getSamplesPerFrame

Class: audioPluginSource

Get frame size returned by the plugin

Syntax

```
frameSize = getSamplesPerFrame(myAudioSourcePlugin)
```

Description

`frameSize = getSamplesPerFrame(myAudioSourcePlugin)` returns the frame size at which the plugin is run. `frameSize` is the number of output samples (rows) that the current call to the processing function of `myAudioSourcePlugin` must return.

- In a digital audio workstation (DAW) environment, `getSamplesPerFrame` interacts with the DAW to determine the frame size. Frame size can vary from call to call, as determined by the DAW environment.
- In the MATLAB environment, `getSamplesPerFrame` returns the value set by a previous call to the `setSamplesPerFrame` method. If `setSamplesPerFrame` has not been called, then `getSamplesPerFrame` returns the default value, 256.

Note When authoring source plugins in MATLAB, `getSamplesPerFrame` is valid only when called in the processing function.

Introduced in R2016a

setSamplesPerFrame

Class: audioPluginSource

Set frame size returned by the plugin (MATLAB environment only)

Syntax

```
setSamplesPerFrame(myAudioSourcePlugin,frameSize)
```

Description

`setSamplesPerFrame(myAudioSourcePlugin,frameSize)` sets the frame size (rows) that the source plugin, `myAudioSourcePlugin`, must return in subsequent calls to its processing function. Specify `frameSize` as a real integer greater than or equal to 0. `setSamplesPerFrame` enables the MATLAB environment to mimic behavior in a digital audio workstation (DAW) environment.

Note Do not use `setSamplesPerFrame` in a generated plugin. If you call `setSamplesPerFrame` in your authored plugin, `generateAudioPlugin` throws an error.

Introduced in R2016a

externalAudioPlugin class

Base class for external audio plugins

Description

`externalAudioPlugin` is the base class for hosted audio plugins. When you load an external plugin using `loadAudioPlugin`, an object of that plugin is created having `externalAudioPlugin` or `externalAudioPluginSource` as a base class. The `externalAudioPluginSource` class is used when the external audio plugin is a source plugin.

For a tutorial on hosting audio plugins, see “Host External Audio Plugins”.

Methods

dispParameter	Display information of single or multiple parameters
getParameter	Get normalized value and information about parameter
info	Get information about hosted plugin
process	Process audio stream
setParameter	Set normalized parameter value of hosted plugin

Inherited Methods

getSampleRate	Get sample rate at which the plugin is run
setSampleRate	Set sample rate at which the plugin is run

Copy Semantics

Handle. To learn how handle classes affect copy operations, see “Object Behavior” (MATLAB) in the MATLAB documentation.

Examples

Specify Hosted Plugin Parameter Values

Load a VST audio plugin into MATLAB® by specifying its full path. If you are using a Mac, replace the `.dll` file extension with `.vst`.

```
pluginPath = fullfile(matlabroot,'toolbox/audio/samples/ParametricEqualizer.dll');
hostedPlugin = loadAudioPlugin(pluginPath)
```

```
hostedPlugin = 

  VST plugin 'ParametricEQ'  2 in, 2 out

                     Parameter    Value    Display
        _____________________________________________
     1              Low Peak Gain:   0.5000      0.000 dB
     2       Low Center Frequency:   0.2330    100.000 Hz
     3               Low Q Factor:   0.2822      2.000
     4           Medium Peak Gain:   0.5000      0.000 dB
     5    Medium Center Frequency:   0.5663   1000.000 Hz
   4 parameters not displayed. Use dispParameter(hostedPlugin) to see all 9 params.
```

Use `info` to return information about the hosted plugin.

```
info(hostedPlugin)
```

```
ans = 

  struct with fields:

        PluginName: 'ParametricEQ'
            Format: 'VST'
     InputChannels: 2
    OutputChannels: 2
         NumParams: 9
        PluginPath: 'E:\jobarchive\Bdoc16b\2016_07_05_h07m05s16_job410158_...'
        VendorName: ''
     VendorVersion: 'V1.0.0'
          UniqueId: 'MWap'
```

Use `setParameter` to change the normalized value of the `Medium Center Frequency` parameter to 0.75. Specify the parameter by its index.

```
setParameter(hostedPlugin,5,0.75)
```

When you set the normalized parameter value, the parameter display value is automatically updated. The normalized parameter value generally corresponds to the position of a UI widget or MIDI controller. The parameter display value typically reflects the value used internally for processing.

Use `dispParameter` to display the updated table of parameters.

```
dispParameter(hostedPlugin)
```

```
                     Parameter    Value    Display
        _____________________________________________
     1           Low Peak Gain:   0.5000       0.000 dB
     2     Low Center Frequency:   0.2330     100.000 Hz
     3            Low Q Factor:   0.2822       2.000
     4        Medium Peak Gain:   0.5000       0.000 dB
     5  Medium Center Frequency:   0.7500    3556.559 Hz
     6         Medium Q Factor:   0.2822       2.000
     7          High Peak Gain:   0.5000       0.000 dB
     8    High Center Frequency:   0.8997   10000.000 Hz
     9           High Q Factor:   0.2822       2.000
```

Alternatively, you can use `getParameter` to return the normalized value of a single parameter.

```
parameterIndex = 5;
parameterValue = getParameter(hostedPlugin,parameterIndex)
```

```
parameterValue =

    0.7500
```

Run External Plugin in MATLAB

Load a VST audio plugin into MATLAB™ by specifying its full path. If you are using a Mac, replace the `.dll` file extension with `.vst`.

```
pluginPath = ...
    fullfile(matlabroot,'toolbox/audio/samples/ParametricEqualizer.dll');
hostedPlugin = loadAudioPlugin(pluginPath);
```

Create input and output objects for an audio stream loop that reads from a file and writes to your audio device. Set the sample rate of the hosted plugin to the sample rate of the input to the plugin.

```
fileReader   = dsp.AudioFileReader('FunkyDrums-44p1-stereo-25secs.mp3');
deviceWriter = audioDeviceWriter('SampleRate',fileReader.SampleRate);
setSampleRate(hostedPlugin,fileReader.SampleRate);
```

Set the `MediumPeakGain` property to -20 dB.

```
hostedPlugin.MediumPeakGain = -20;
```

Use the hosted plugin to process the audio file in an audio stream loop. Sweep the medium peak gain upward in the loop to hear the effect.

```
while hostedPlugin.MediumPeakGain < 19
    hostedPlugin.MediumPeakGain = hostedPlugin.MediumPeakGain + 0.04;
    x = fileReader();
    y = process(hostedPlugin,x);
    deviceWriter(y);
end

release(fileReader)
release(deviceWriter)
```

See Also

Functions
`loadAudioPlugin`

Classes
`audioPlugin` | `audioPluginSource` | `externalAudioPluginSource`

Topics

"Host External Audio Plugins"
"Hierarchies of Classes — Concepts" (MATLAB)

Introduced in R2016b

dispParameter

Class: externalAudioPlugin

Display information of single or multiple parameters

Syntax

```
dispParameter(hostedPlugin)
dispParameter(hostedPlugin,parameter)
```

Description

`dispParameter(hostedPlugin)` displays all parameters and associated indices, values, displayed values, and display labels. For example:

```
dispParameter(hostedPlugin)
```

```
             Parameter    Value    Display
        _____________________________________
    1              Wet:   1.0000    +0.0 dB
    2              Dry:   1.0000    +0.0 dB
    3      1: Enabled:    1.0000      ON
    4       1: Length:    0.0000     0.0 ms
    5       1: Length:    0.0156    4.00 8N
    6     1: Feedback:    0.0000    -inf dB
    7      1: Lowpass:    1.0000   20000 Hz
    8       1: Hipass:    0.0000       0 Hz
    9   1: Resolution:    1.0000      24 bits
   10 1: Stereo width:    1.0000    1.00
   11       1: Volume:    1.0000    +0.0 dB
   12          1: Pan:    0.5000     0.0 %
```

The `Value` column corresponds to the normalized parameter value. Generally, the normalized parameter value represents the position of a UI widget or MIDI controller. The `Display` column corresponds to an internal parameter value used for processing. The `Value` and `Display` are related by an unknown mapping that is internal to the hosted plugin.

`dispParameter(hostedPlugin,parameter)` displays a subset of parameters. You can specify a parameter by its name as a character vector, string, or as a vector of one or more parameter indices. For example:

- `dispParameter(hostedPlugin,'Gain')` displays information about the `'Gain'` parameter of `hostedPlugin`.
- `dispParameter(hostedPlugin,[1,3])` displays information about parameters specified by indices `1` and `3`.

Introduced in R2016b

getParameter

Class: externalAudioPlugin

Get normalized value and information about parameter

Syntax

```
value = getParameter(hostedPlugin,parameter)
[value, parameterInformation] = getParameter(hostedPlugin,parameter)
```

Description

`value = getParameter(hostedPlugin,parameter)` returns the normalized value of the `parameter` of `hostedPlugin`. You can specify a parameter by its name as a character vector, string, or by its index. For example:

- `getParameter(hostedPlugin,'Gain')` returns the normalized value of the hosted plugin parameter named `'Gain'`. If the parameter name is not unique, `getParameter` returns an error.
- `getParameter(hostedPlugin,2)` returns information about the parameter specified by index `2`.

`[value, parameterInformation] = getParameter(hostedPlugin,parameter)` returns a structure containing additional information about the specified parameter of the hosted plugin.

Field	Description
`DisplayName`	Display name or prompt of the plugin parameter, returned as a character vector. The display name is intended for display on the plugin’s user interface (UI).

Field	Description
`DisplayValue`	Display value of the plugin parameter, returned as a character vector. The parameter `DisplayValue` corresponds to the normalized parameter `value` by an unknown mapping internal to the hosted plugin. Generally, the display value reflects the value used internally by the plugin for processing, while the normalized parameter value corresponds to the position of a MIDI control or widget on a UI.
`Label`	Label intended for display with `DisplayValue` on the plugin's UI, returned as a character vector. Typical labels include `dB` and `Hz`.

Introduced in R2016b

info

Class: externalAudioPlugin

Get information about hosted plugin

Syntax

```
pluginInfo = info(hostedPlugin)
```

Description

`pluginInfo = info(hostedPlugin)` returns a structure containing information about the hosted plugin.

Field	Description
`PluginName`	Display name of plugin.
`Format`	Software interface. Supported formats include VST, VST3, and AU.
`InputChannels`	Number of channels passed to the processing function of the plugin.
`OutputChannels`	Number of channels returned from the processing function of the plugin.
`NumParams`	Total number of plugin parameters.
`PluginPath`	Path specified when plugin is loaded using `loadAudioPlugin`.
`VendorName`	Name of the plugin creator.
`VendorVersion`	Version number. Typically used to track plugin releases.
`UniqueID`	Unique identifier of plugin used for recognition in certain digital audio workstation (DAW) environments.

Introduced in R2016b

process

Class: externalAudioPlugin

Process audio stream

Syntax

```
audioOut = process(hostedPlugin,audioIn)
```

Description

`audioOut = process(hostedPlugin,audioIn)` returns an audio signal processed according to the algorithm and parameters of `hostedPlugin`. For source plugins, call `process` without an audio input. Use `info(hostedPlugin)` to determine the number of channels (columns) of the input and output audio signal.

Use `setSamplesPerFrame(hostedPlugin)` to specify the frame size returned by hosted source plugins.

Introduced in R2016b

setParameter

Class: externalAudioPlugin

Set normalized parameter value of hosted plugin

Syntax

`setParameter(hostedPlugin,parameter,newValue)`

Description

`setParameter(hostedPlugin,parameter,newValue)` sets the normalized value corresponding to the `parameter` of `hostedPlugin` to `newValue`. Specify the parameter by its unique display name or its index. Specify the new normalized parameter value as a scalar in the range 0–1.

For example, assume `hostedPlugin` has a parameter with index 3 and a unique display name, `'Gain'`. These commands are identical:

- `setParameter(hostedPlugin,'Gain',0.2)`
- `setParameter(hostedPlugin,3,0.2)`

Note A hosted plugin might quantize its parameters. The result of `setParameter` for quantized parameters depends on the type of quantization.

Introduced in R2016b

externalAudioPluginSource class

Base class for external audio source plugins

Description

`externalAudioPluginSource` is the base class for hosted audio source plugins. When you load an external plugin using `loadAudioPlugin`, an object of that plugin is created having `externalAudioPlugin` or `externalAudioPluginSource` as a base class. The `externalAudioPluginSource` class is used when the external audio plugin is a source plugin.

For a tutorial on hosting audio plugins, see "Host External Audio Plugins".

Methods

Inherited Methods

dispParameter	Display information of single or multiple parameters
getParameter	Get normalized value and information about parameter
info	Get information about hosted plugin
process	Process audio stream
setParameter	Set normalized parameter value of hosted plugin

getSampleRate	Get sample rate at which the plugin is run
setSampleRate	Set sample rate at which the plugin is run

getSamplesPerFrame	Get frame size returned by the plugin
setSamplesPerFrame	Set frame size returned by the plugin (MATLAB environment only)

Copy Semantics

Handle. To learn how handle classes affect copy operations, see “Object Behavior” (MATLAB) in the MATLAB documentation.

Examples

Specify Hosted Source Plugin Parameter Values

Load a VST audio source plugin into MATLAB® by specifying its full path. If you are using a Mac, replace the `.dll` file extension with `.vst`.

```
pluginPath = fullfile(matlabroot,'toolbox/audio/samples/oscillator.dll');
hostedSourcePlugin = loadAudioPlugin(pluginPath)
```

```
hostedSourcePlugin =

  VST plugin 'oscillator'  source, 1 out, 256 samples

         Parameter    Value    Display
        _______________________________
     1   Frequency:   0.5659   100.000 Hz
     2   Amplitude:   0.1000     1.000 AU
     3   DC Offset:   0.5000     0.000 AU
```

Use `info` to return information about the hosted plugin.

```
info(hostedSourcePlugin)
```

```
ans =

  struct with fields:

        PluginName: 'oscillator'
            Format: 'VST'
     InputChannels: 0
    OutputChannels: 1
         NumParams: 3
        PluginPath: 'E:\jobarchive\Bdoc16b\2016_07_05_h07m05s16_job410158_...'
```

```
      VendorName: ''
   VendorVersion: 'V1.0.0'
        UniqueId: 'MWap'
```

Use `setParameter` to change the normalized value of the `Frequency` parameter to 0.8. Specify the parameter by its index.

```
setParameter(hostedSourcePlugin,1,0.8)
```

When you set the normalized parameter value, the parameter display value is automatically updated. Generally, the normalized parameter value corresponds to the position of a UI widget or MIDI controller. The parameter display value typically reflects the value used internally by the plugin for processing.

Use `dispParameter` to display the updated table of parameters.

```
dispParameter(hostedSourcePlugin)
```

```
      Parameter    Value    Display
     _______________________________
 1   Frequency:   0.8000   1741.101 Hz
 2   Amplitude:   0.1000      1.000 AU
 3   DC Offset:   0.5000      0.000 AU
```

Alternatively, you can use `getParameter` to return the normalized value of a single parameter.

```
getParameter(hostedSourcePlugin,1)
```

```
ans =

    0.8000
```

Run External Source Plugin in MATLAB

Load a VST audio source plugin into MATLAB™ by specifying its full path. If you are using a Mac, replace the `.dll` file extension with `.vst`.

```
pluginPath = fullfile(matlabroot,'toolbox','audio','samples','oscillator.dll');
hostedSourcePlugin = loadAudioPlugin(pluginPath);
```

Set the `Amplitude` property to 0.5. Set the `Frequency` property to 16 kHz.

```
hostedSourcePlugin.Amplitude = 0.5;
hostedSourcePlugin.Frequency = 16000;
```

Set the sample rate at which to run the plugin. Create an output object to write to your audio device.

```
setSampleRate(hostedSourcePlugin,44100);
deviceWriter = audioDeviceWriter('SampleRate',44100);
```

Use the hosted source plugin to output an audio stream. The processing in the audio stream loop ramps the frequency parameter down and then up.

```
k = 1;
for i = 1:1000
    hostedSourcePlugin.Frequency = hostedSourcePlugin.Frequency - 30*k;
    y = process(hostedSourcePlugin);
    deviceWriter(y);
    if (hostedSourcePlugin.Frequency - 30 <= 0.1) || ...
            (hostedSourcePlugin.Frequency + 30 >= 20e3)
        k = -1*k;
    end
end

release(deviceWriter)
```

See Also

Functions
`loadAudioPlugin`

Classes
`audioPlugin` | `audioPluginSource` | `externalAudioPlugin`

Topics
“Host External Audio Plugins”
“Hierarchies of Classes — Concepts” (MATLAB)

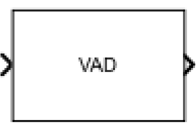
Introduced in R2016b

5

Blocks in Audio System Toolbox

Voice Activity Detector

Detect presence of speech in audio signal
Library: Audio System Toolbox / Measurements



Description

The Voice Activity Detector block detects the presence of speech in an audio signal. You can also use the Voice Activity Detector block to output an estimate of the noise variance per frequency bin.

Ports

Input

`Port_1` — Input signal
matrix | 1-D vector

- Matrix input –– Each column of the input is treated as an independent channel.
- 1-D vector input –– The input is treated as a single channel.

Data Types: `single` | `double`

Output

P — Probability that speech is present
scalar | row vector

The block outputs a scalar or row vector with the same number of columns as the input signal.

This port is unnamed until you select the **Output noise variance** parameter.

Data Types: `single` | `double`

N — Estimate of noise variance per frequency bin

column vector | matrix

The block outputs a column vector or a matrix with the same number of columns as the input signal.

Dependencies

To enable this port, select the **Output noise variance** parameter.

Data Types: `single` | `double`

Parameters

If a parameter is listed as tunable, then you can change its value during simulation.

`Domain of the input` — Domain of the input

`Time` (default) | `Frequency`

Tunable: No

`Window` — Windowing function applied before FFT

`Hann` (default) | `Chebyshev` | `Flat Top` | `Hamming` | `Kaiser` | `Rectangular`

The window function is designed using the algorithms of the following functions:

- `Hann` -- `hann`
- `Chebyshev` -- `chebwin`
- `Flat Top` -- `flattopwin`
- `Hamming` -- `hamming`
- `Kaiser` -- `kaiser`

Tunable: No

Dependencies

To enable this parameter, set **Domain of the input** to `Time`.

`Sidelobe attenuation of the window (dB)` — Sidelobe attenuation of the window (dB)
`60` (default) | positive finite scalar

Tunable: No

Dependencies

To enable this parameter, set **Domain of the input** to `Time` and **Window** to `Chebyshev` or `Kaiser`.

Data Types: `single` | `double`

`Inherit FFT length from input dimensions` — Set FFT length to number of input samples
on (default) | off

Tunable: No

Dependencies

To enable this parameter, set **Domain of the input** to `Time`.

`FFT length` — Number of bins in frequency domain
`1024` (default) | positive finite integer

Tunable: No

Dependencies

To enable this parameter, set **Domain of the input** to `Time` and clear the **Inherit FFT length from input dimensions** parameter.

Data Types: `single` | `double`

`Probability of transition from a silence frame to a speech frame` — Probability that a speech frame follows a silence frame
`0.2` (default) | value in the range [0,1]

Tunable: Yes

Data Types: `single` | `double`

`Probability of transition from a speech frame to a silence frame` — Probability that a silence frame follows a speech frame
`0.1` (default) | value in the range [0,1]

Tunable: Yes

Data Types: `single` | `double`

`Output noise variance` — Output estimate of noise variance per frequency bin
`off` (default) | `on`

When you select this parameter, an additional output port, **N**, is added to the block.

Tunable: No

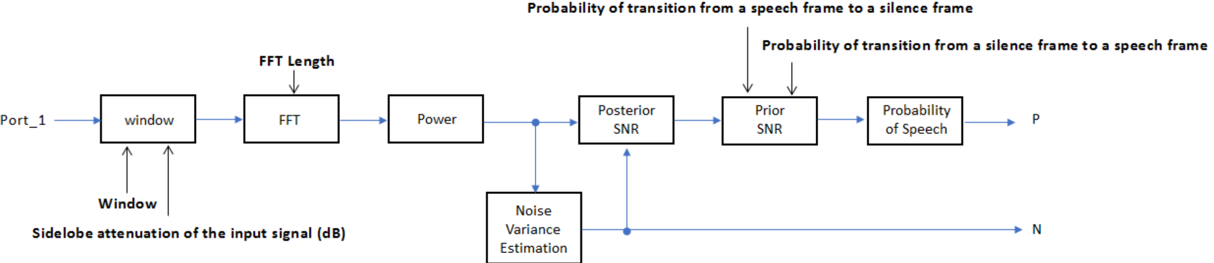
`Simulate using` — Specify type of simulation to run
`Code generation` (default) | `Interpreted execution`

- `Code generation` -- Simulate model using generated C code. The first time you run a simulation, Simulink® generates C code for the block. The C code is reused for subsequent simulations, as long as the model does not change. This option requires additional startup time but the speed of the subsequent simulations is comparable to `Interpreted execution`.
- `Interpreted execution` -- Simulate model using the MATLAB interpreter. This option shortens startup time but has a slower simulation speed than `Code generation`. In this mode, you can debug the source code of the block.

Tunable: No

Algorithms

The Voice Activity Detector implements the algorithm described in [1].



If **Domain of the input** is specified as `Time`, the input signal is windowed and then converted to the frequency domain according to the **Window**, **Sidelobe attenuation of**

the window (dB), and **FFT length** parameters. If **Domain of the input** is specified as `Frequency`, the input is assumed to be a windowed discrete time Fourier transform (DTFT) of an audio signal. The signal is then converted to the power domain. Noise variance is estimated according to [2]. The posterior and prior SNR are estimated according to the Minimum Mean-Square Error (MMSE) formula described in [3]. A log likelihood ratio test with a Hidden Markov Model (HMM)-based hang-over scheme is used, according to [1].

References

[1] Sohn, Jongseo., Nam Soo Kim, and Wonyong Sung. "A Statistical Model-Based Voice Activity Detection." *Signal Processing Letters IEEE*. Vol. 6, No. 1, 1999.

[2] Martin, R. "Noise Power Spectral Density Estimation Based on Optimal Smoothing and Minimum Statistics." *IEEE Transactions on Speech and Audio Processing*. Vol. 9, No. 5, 2001, pp. 504–512.

[3] Ephraim, Y., and D. Malah. "Speech Enhancement Using a Minimum Mean-Square Error Short-Time Spectral Amplitude Estimator." *IEEE Transactions on Acoustics, Speech, and Signal Processing*. Vol. 32, No. 6, 1984, pp. 1109–1121.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

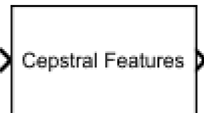
See Also

`voiceActivityDetector`

Introduced in R2018a

Cepstral Feature Extractor

Extract cepstral features from audio segment
Library: Audio System Toolbox / Measurements



Description

The Cepstral Feature Extractor block extracts cepstral features from an audio segment. Cepstral features are commonly used to characterize speech and music signals.

Ports

Input

`Port_1` — Audio input to cepstral feature extractor
column vector | matrix

Audio input to the cepstral feature extractor, specified as a column vector or a matrix. If specified as a matrix, the columns are treated as independent audio channels.

Data Types: `single` | `double`

Output

`coeffs` — Cepstral coefficients
column vector | matrix

Cepstral coefficients, returned as a column vector or a matrix. If the coefficients matrix is an *N*-by-*M* matrix, *N* is determined by the values you specify in the **Number of coefficients to return** and **Log energy usage** parameters. *M* equals the number of input audio channels.

When the **Log energy usage** parameter is set to:

- `Append` –– The block prepends the log energy value to the coefficients vector. The length of the coefficients vector is 1 + *NumCoeffs*, where *NumCoeffs* is the value specified in the **Number of coefficients to return** parameter.
- `Replace` –– The block replaces the first coefficient with the log energy of the signal. The length of the coefficients vector is *NumCoeffs*.
- `Ignore` –– The block does not calculate or return the log energy.

This port is unnamed until you select **Output delta** parameter, the **Output delta-delta** parameter, or both.

Data Types: `single` | `double`

`delta` — Change in coefficients
column vector | matrix

Change in coefficients over consecutive calls to the algorithm, returned as a column vector or a matrix. The **delta** array is of the same size and data type as the **coeffs** array.

Dependencies

To enable this port, select the **Output delta** parameter.

Data Types: `single` | `double`

`deltaDelta` — Change in delta values
column vector | matrix

Change in **delta** values over consecutive calls to the algorithm, returned as a column vector or a matrix. The **deltaDelta** array is the same size and data type as the **coeffs** and **delta** arrays.

Dependencies

To enable this port, select the **Output delta-delta** parameter.

Data Types: `single` | `double`

Parameters

If a parameter is listed as tunable, then you can change its value during simulation.

`Domain of the input signal` — Input signal domain
`Time` (default) | `Frequency`

Input signal domain, specified as either `Time` or `Frequency`.

Tunable: No

`Number of coefficients to return` — Number of coefficients to return

`13` (default) | positive integer

Number of coefficients to return, specified as an integer in the range [2, *v*], where *v* is the number of valid passbands.

The number of valid passbands is defined as `sum(BandEdges <= floor(fs/2))-2`.

- *BandEdges* –– Vector containing the band edges of the filter bank, specified through the **Band edges of filter bank (Hz)** parameter.
- *fs* –– Sample rate of the input audio signal, specified through the **Input sample rate (Hz)** parameter. If the **Inherit sample rate from input** parameter is selected, the sample rate is inherited from the input signal.

Tunable: No

Data Types: `single` | `double`

`Inherit FFT length from input dimensions` — Inherit FFT length from input

`on` (default) | `off`

When you select this parameter, the FFT length is equal to the number of rows in the input signal.

Tunable: No

Dependencies

To enable this parameter, set **Domain of the input signal** to `Time`.

`FFTLength` — FFT length

`[]` (default) | positive integer

FFT length, specified as a positive integer. The default, `[]`, means that the FFT length is equal to the number of rows in the input signal.

Tunable: No

Dependencies

To enable this parameter, set **Domain of the input signal** to `Time` and select the **Inherit FFT length from input dimensions** paramerter.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`Log energy usage` — Specify how the log energy is shown

`Append` (default) | `Replace` | `Ignore`

Specify how the log energy is shown in the coefficients vector output, specified as:

- `Append` –– The block prepends the log energy to the coefficients vector. The length of the coefficients vector is 1 + *NumCoeffs*, where *NumCoeffs* is the value specified in the **Number of coefficients to return** parameter.
- `Replace` –– The block replaces the first coefficient with the log energy of the signal. The length of the coefficients vector is *NumCoeffs*.
- `Ignore` –– The block does not calculate or return the log energy.

Tunable: No

`Output delta` — Output delta values

`off` (default) | `on`

When you select this parameter, an additional output port, **delta**, is added to the block. This port outputs the change in coefficients over consecutive calls to the algorithm.

Tunable: No

`Output delta-delta` — Output delta-delta values

`off` (default) | `on`

When you select this parameter, an additional output port, **deltaDelta**, is added to the block. This port outputs the change in delta values over consecutive calls to the alorithm.

Tunable: No

`Inherit sample rate from input` — Specify source of input sample rate

`off` (default) | `on`

When you select this parameter, the block inherits its sample rate from the input signal. When you clear this parameter, you specify the sample rate in **Input sample rate (Hz)** parameter.

Tunable: No

`Input sample rate (Hz)` — Sample rate of input

`16000` (default) | positive scalar

Input sample rate in Hz, specified as a real positive scalar.

Tunable: Yes

Dependencies

To enable this parameter, clear the **Inherit sample rate from input** parameter.

`Simulate using` — Specify type of simulation to run

`Code generation` (default) | `Interpreted execution`

- `Code generation` -- Simulate model using generated C code. The first time you run a simulation, Simulink generates C code for the block. The C code is reused for subsequent simulations, as long as the model does not change. This option requires additional startup time, but the speed of the subsequent simulations is comparable to `Interpreted execution`.
- `Interpreted execution` -- Simulate model using the MATLAB interpreter. This option shortens startup time but has a slower simulation speed than `Code generation`. In this mode, you can debug the source code of the block.

Tunable: No

Advanced Tab

`Band edges of filter bank (Hz)` — Band edges of auditory filter bank

row vector

Band edges of the filter bank in Hz, specified as a nonnegative monotonically increasing row vector in the range [0, ∞). The maximum bandedge frequency can be any finite number. The number of bandedges must be in the range [4, 80].

The default band edges are spaced linearly for the first ten and then logarithmically thereafter. The default band edges are set as recommended by [1].

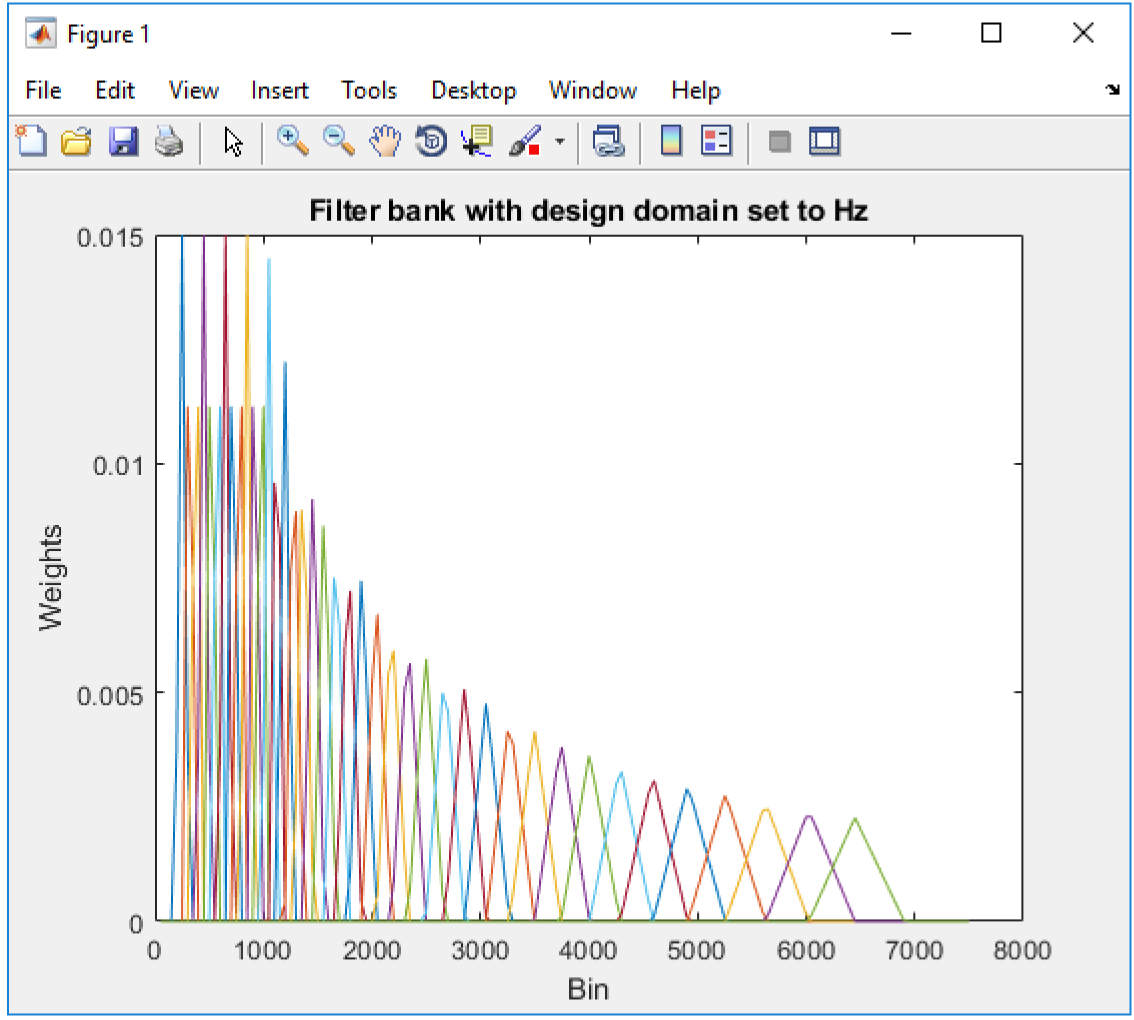
Tunable: No

`Domain for filter bank design` — Filter bank design domain
`Hz` (default) | `Bin`

Domain for filter bank design, specified as either `Hz` or `Bin`. The filterbank is designed as overlapped triangles with band edges specified by the **Band edges of filter bank (Hz)** parameter.

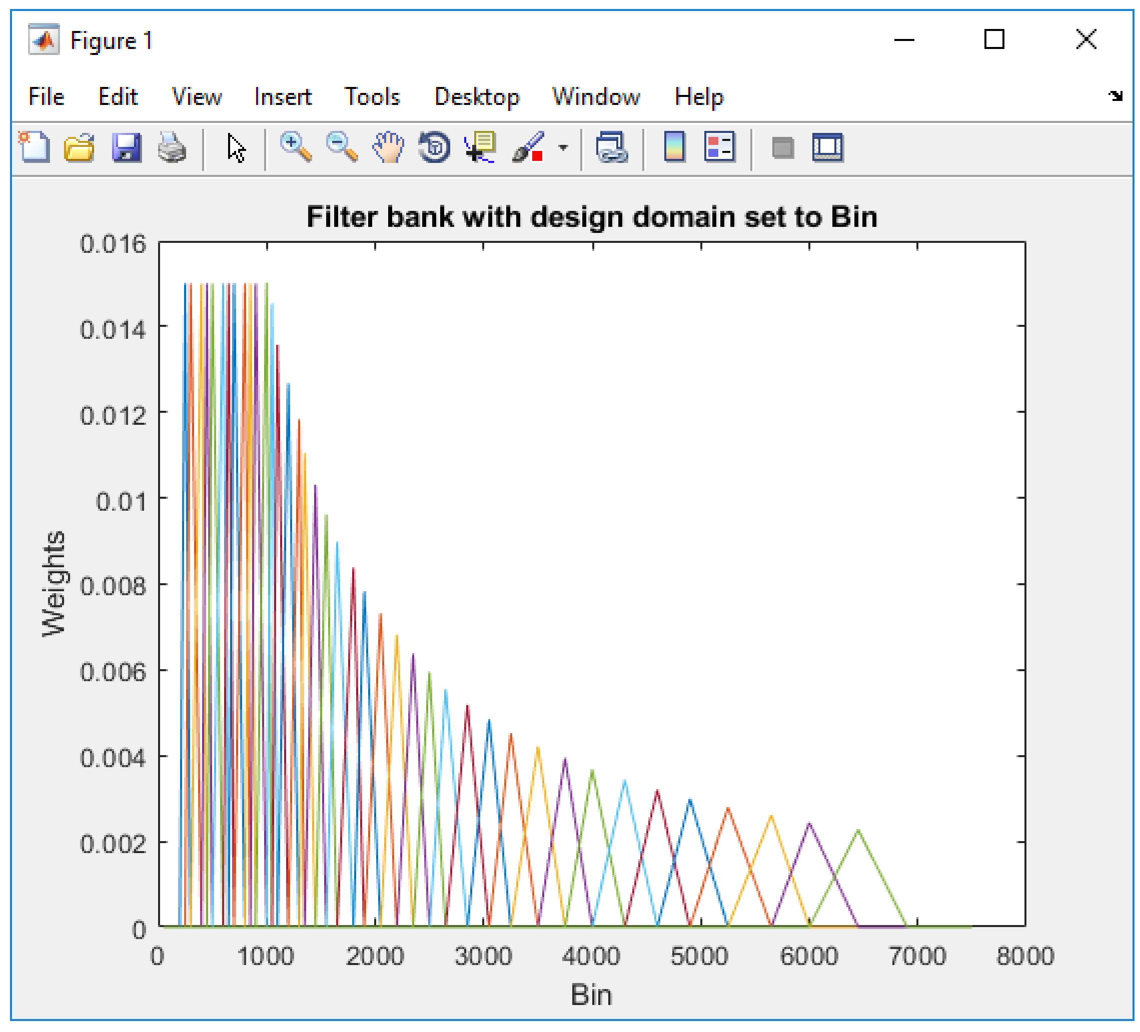
The band edges are specified in Hz. When you set the design domain to:

- `Hz` –– Filter bank triangles are drawn in Hz and are mapped onto bins.



For details, see [1].

- `Bin` -- The band edge frequencies in `Hz` are converted to bins. The filter bank triangles are drawn symmetrically in bins.



For details, see [2].

Tunable: No

`Filter bank normalization` — Normalize filter bank
`Bandwidth` (default) | `Area` | `None`

Normalization technique used to normalize the weights of the filter bank, specified as:

- `Bandwidth` –– The weights of each bandpass filter are normalized by the corresponding bandwidth of the filter.
- `Area` –– The weights of each bandpass filter are normalized by the corresponding area of the bandpass filter.
- `None` –– The weights of the filter are not normalized.

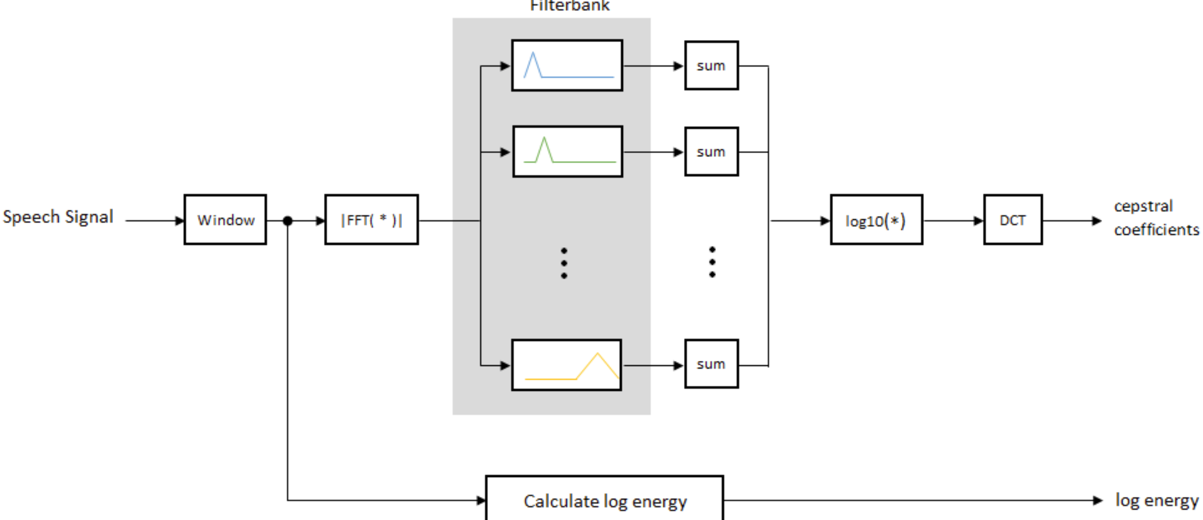
Tunable: No

Algorithms

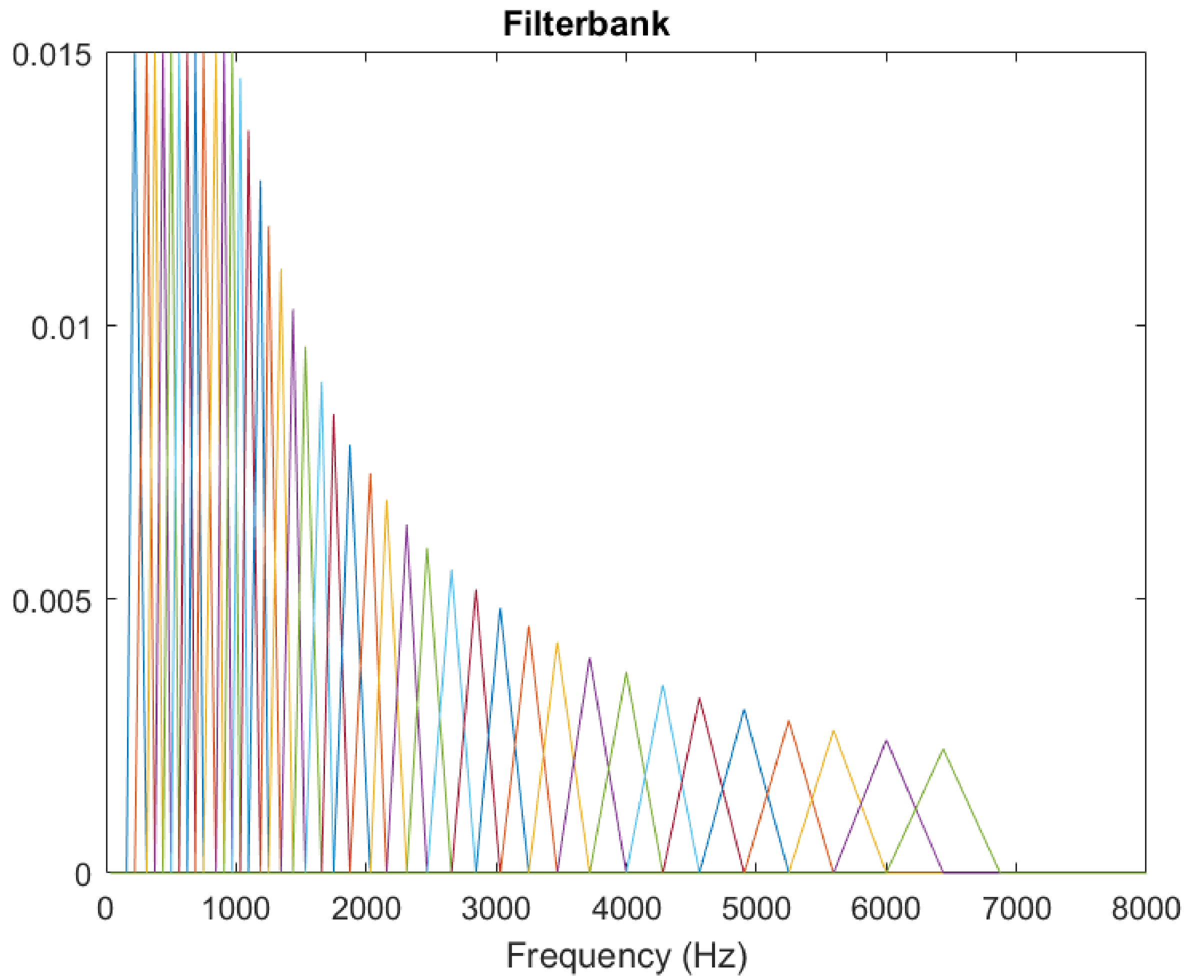
Auditory Cepstrum Coefficients

Auditory cepstrum coefficients are popular features extracted from speech signals for use in recognition tasks. In the source-filter model of speech, cepstral coefficients are understood to represent the filter (vocal tract). The vocal tract frequency response is relatively smooth, whereas the source of voiced speech can be modeled as an impulse train. As a result, the vocal tract can be estimated by the spectral envelope of a speech segment.

The motivating idea of cepstral coefficients is to compress information about the vocal tract (smoothed spectrum) into a small number of coefficients based on an understanding of the cochlea. Although there is no hard standard for calculating the coefficients, the basic steps are outlined by the diagram.



The windowing is done by a Hamming function. The default filter bank linearly spaces the first 10 triangular filters and logarithmically spaces the remaining filters.



Log Energy

If the input (x) is a time-domain signal, the log energy is computed using the following equation:

$$\log E = \log(\text{sum}(x^2))$$

If the input (x) is a frequency-domain signal, the log energy is computed using the following equation:

$$\log E = \log\left(\text{sum}\left(|x|^2\right) \Big/ FFTLength\right)$$

References

[1] Auditory Toolbox. https://engineering.purdue.edu/~malcolm/interval/1998-010/AuditoryToolboxTechReport.pdf

[2] ETSI ES 201 108 V1.1.3 (2003-09). http://www.etsi.org/deliver/etsi_es/201100_201199/201108/01.01.03_60/es_201108v010103p.pdf

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

Functions
`mfcc` | `pitch`

System Objects
`cepstralFeatureExtractor` | `voiceActivityDetector`

Blocks
Voice Activity Detector

Topics

"Speaker Identification Using Pitch and MFCC"

Introduced in R2018a

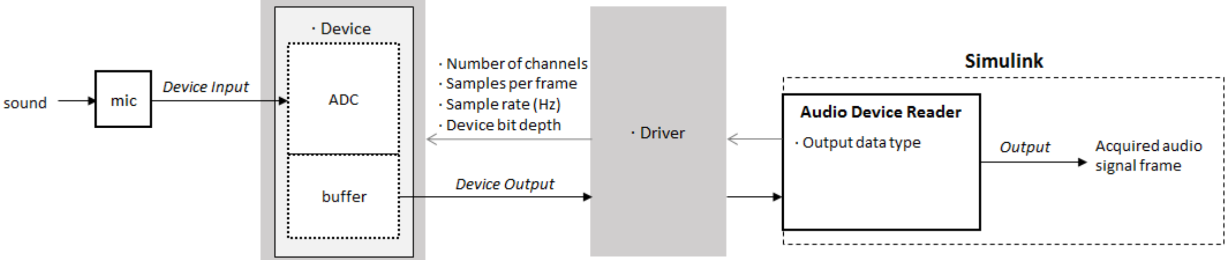
Audio Device Reader

Record from sound card

Library: Audio System Toolbox / Sources

Description

The Audio Device Reader block reads audio samples using your computer's audio device. The Audio Device Reader block specifies the driver, the device and its attributes, and the data type and size output from your Audio Device Reader block.



Ports

Output

A — Output signal
scalar | vector | matrix

The output of the Audio Device Reader block is determined by the block’s parameters. If the block output is a matrix, the columns correspond to independent channels.

Data Types: `single` | `double` | `int16` | `int32` | `uint8`

0 — Number of samples overrun
scalar

This port outputs the number of samples overrun while acquiring a frame of data (one output matrix).

Dependencies

To enable this port, select the **Output number of samples overrun** parameter.

Data Types: `uint32`

Parameters

Main Tab

`Driver` — Driver used to access your audio device
`DirectSound` (default) | `ASIO` | `WASAPI`

- ASIO drivers do not come pre-installed on Windows machines. To use the `ASIO` driver option, install an ASIO driver outside of MATLAB.

  Note If **Driver** is set to `ASIO`, open the ASIO UI outside of MATLAB to set the sound card buffer size to the value specified by the **Samples per frame** parameter. See the documentation of your ASIO driver for more information.

- WASAPI drivers are supported for exclusive-mode only.

ASIO and WASAPI drivers do not provide sample rate conversion. For ASIO and WASAPI drivers, set **Sample rate (Hz)** to a sample rate supported by your audio device.

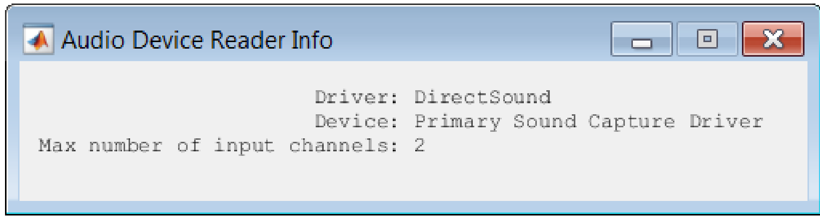
This parameter applies only on Windows machines. Linux machines always use the ALSA driver. Mac machines always use the CoreAudio driver.

`Device` — Device used to acquire audio samples
default audio device (default)

The device list is populated with devices available on your computer.

`Info` — View information about your audio input configuration
button

This button opens a dialog box that lists your selected audio driver, the full name of your audio device, and the maximum input channels for your configuration. For example:



Sample rate (Hz) — Sample rate your device uses to acquire audio data
44100 (default) | integer

The possible range of **Sample rate (Hz)** depends on your audio hardware.

Number of channels — Number of channels acquired by your audio device
1 (default) | integer

The number of input channels is also the number of channels (matrix columns) output by the Audio Device Reader block.

Dependencies

To specify which input channels your audio device acquires, on the **Advanced** tab, select the **Use default mapping between sound card's input channels and columns of output of this block** parameter.

Samples per frame — Frame size read from audio device
1024 (default) | integer

Samples per frame is also the device buffer size, and the frame size (number of matrix rows) output by the Audio Device Reader block.

Advanced Tab

Device bit depth — Data type used by device to acquire audio data
16-bit integer (default) | 8-bit integer | 16-bit integer | 24-bit integer | 32-bit integer

Use default mapping between sound card's input channels and columns of output of this block — Toggle channel mapping source
on (default) | off

When you select this parameter, the block uses the default mapping between the sound card's input channels and the matrix columns output by this block. When you clear this parameter, you specify the mapping in **Device input channels**.

`Device input channels` — Specify nondefault channel mapping

`[1:MaximumInputChannels]` (default) | scalar | vector

Nondefault map of device channels and matrix output by the Audio Device Reader block, specified as a scalar or vector. For example:

If **Device input channels** is specified as `1:3`, then:

- Channel 1 maps to the first column of the output matrix.
- Channel 2 maps to the second column of the output matrix.
- Channel 3 maps to the third column of the output matrix.

If **Device input channels** is specified as `[3,1,2]`, then:

- Channel 3 maps to the first column of the output matrix.
- Channel 1 maps to the second column of the output matrix.
- Channel 2 maps to the third column of the output matrix.

Dependencies

To specify a nondefault mapping, clear the **Use default mapping between sound card's input channels and columns of output of this block** parameter.

`Output number of samples overrun` — Specify additional output port for number of samples overrun

off (default) | on

When you select this parameter, an additional output port, **O**, is added to the block. The **O** port outputs the number of samples overrun while acquiring a frame of data (one output matrix).

`Output data type` — Data type output from block

`double` (default) | `single` | `int32` | `int16` | `uint8`

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

The executable generated from this block relies on prebuilt dynamic library files (.dll files) included with MATLAB. Use the `packNGo` function to package the code generated from this object and all the relevant files in a compressed zip file. Using this zip file, you can relocate, unpack, and rebuild your project in another development environment where MATLAB is not installed. For more details, see “Run Audio I/O Features Outside MATLAB and Simulink”.

See Also

System Objects
`audioDeviceReader` | `audioDeviceWriter`

Blocks
Audio Device Writer

Topics

“Run Audio I/O Features Outside MATLAB and Simulink”
“Audio I/O: Buffering, Latency, and Throughput”

Introduced in R2016a

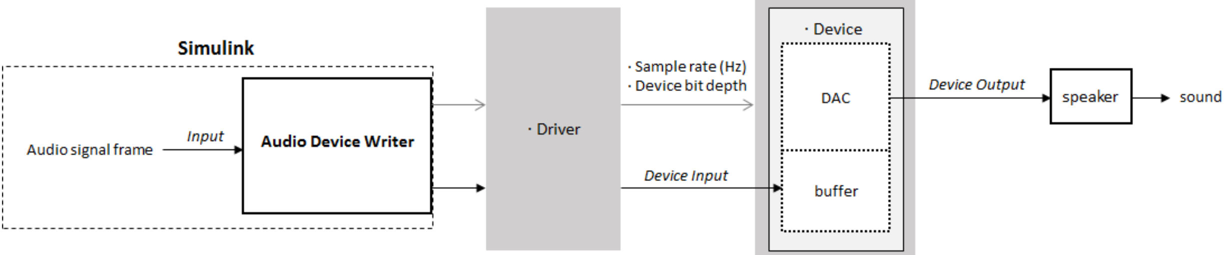
Audio Device Writer

Play to sound card
Library: Audio System Toolbox / Sinks

Description

The Audio Device Writer block writes audio samples to an audio output device.

Parameters of the Audio Device Writer block specify the driver, the device, and device attributes such as sample rate and bit depth.



Ports

Input

`Port_1` — Input signal

scalar | vector | matrix

If input to the Audio Device Writer block is of data type `double` or `single`, the block clips values outside the range [–1, 1]. For other data types, the allowed input range is [min, max] of the specified data type.

Data Types: `single` | `double` | `int16` | `int32` | `uint8`

Output

`Port_1` — Number of samples underrun
scalar

This port outputs the number of samples underrun while writing a frame of data (one input matrix).

Dependencies

To enable this port, select the **Output number of samples underrun** parameter.

Data Types: `uint32`

Parameters

Main Tab

`Driver` — Driver used to access your audio device
`DirectSound` (default) | `ASIO` | `WASAPI`

- ASIO drivers do not come pre-installed on Windows machines. To use the `ASIO` driver option, install an ASIO driver outside of MATLAB.

  Note If **Driver** is set to `ASIO`, open the ASIO UI outside of MATLAB to set the sound card buffer size to the frame size (number of rows) input to the Audio Device Writer block. See the documentation of your ASIO driver for more information.

- WASAPI drivers are supported for exclusive-mode only.

ASIO and WASAPI drivers do not provide sample rate conversion. For ASIO and WASAPI drivers, supply an audio stream with a sample rate supported by your audio device.

This parameter applies only on Windows machines. Linux machines always use the ALSA driver. Mac machines always use the CoreAudio driver.

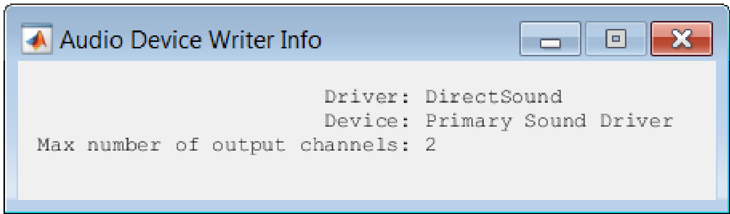
To specify nondefault **Driver** values, you must install Audio System Toolbox. If the toolbox is not installed, specifying nondefault **Driver** values returns an error.

`Device` — Device used to play audio samples
default audio device (default)

The device list is populated with devices available on your computer.

`Info` — View information about your audio output configuration
button

This button opens a dialog box that lists your selected audio driver, the full name of your audio device, and the maximum output channels for your configuration. For example:



`Inherit sample rate from input` — Specify source of input sample rate
on (default) | off

When you select this parameter, the block inherits its sample rate from the input signal. When you clear this parameter, you specify the sample rate in **Sample rate (Hz)**.

`Sample rate (Hz)` — Sample rate used by device to play audio data
`44100` (default) | positive scalar

The possible range of **Sample rate (Hz)** depends on your audio hardware.

Dependencies

To enable this parameter, clear the **Inherit sample rate from input** parameter.

Advanced Tab

`Device bit depth` — Data type used by device to perform digital-to-analog conversion
`16-bit integer` (default) | `8-bit integer` | `24-bit integer` | `32-bit float`

Before performing digital-to-analog conversion, the input data is cast to a data type specified by this parameter.

Note To specify a nondefault **Device bit depth**, you must install Audio System Toolbox. If the toolbox is not installed, specifying a nondefault **Device bit depth** returns an error.

`Use default mapping between columns of input of this block and sound card’s output channels` — Toggle channel mapping source
on (default) | off

When you select this parameter, the block uses the default mapping between columns of the matrix input to this block and the channels of your device. When you clear this parameter, you specify the mapping in **Device output channels**.

`Device output channels` — Specify nondefault channel mapping
`[1:MaximumOutputChannels]` (default) | scalar | vector

Nondefault mapping between columns of matrix input to the Audio Device Writer block and channels of output device, specified as a scalar or vector. For example:

If **Device output channels** is specified as `1:3`, then:

- The first column of the input matrix maps to channel 1.
- The second column of the input matrix maps to channel 2.
- The third column of the input matrix maps to channel 3.

If **Device output channels** is specified as `[3,1,2]`, then:

- The first column of the input matrix maps to channel 3.
- The second column of the input matrix maps to channel 1.
- The third column of the input matrix maps to channel 2.

Note To selectively map between columns of the input matrix and your sound card's output channels, you must install Audio System Toolbox. If the toolbox is not installed, specifying nondefault values for **Device output channels** returns an error.

Dependencies

To enable this parameter, clear the **Use default mapping between columns of input of this block and sound card’s output channels** parameter.

`Output number of samples underrun` — Specify output port for number of samples underrun
off (default) | on

When you select this parameter, an output port is added to the block. The port outputs the number of samples underrun while writing a frame of data (one input matrix).

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

Usage notes and limitations:

- "System Objects in MATLAB Code Generation" (MATLAB Coder)
- The executable generated from this System object relies on prebuilt dynamic library files (`.dll` files) included with MATLAB. Use the `packNGo` function to package the code generated from this object and all the relevant files in a compressed zip file. Using this zip file, you can relocate, unpack, and rebuild your project in another development environment where MATLAB is not installed. For more details, see "Run Audio I/O Features Outside MATLAB and Simulink".

See Also

Blocks
Audio Device Reader | Binary File Reader

System Objects
`audioDeviceReader` | `audioDeviceWriter`

Topics

"Run Audio I/O Features Outside MATLAB and Simulink"
"Audio I/O: Buffering, Latency, and Throughput"

Introduced in R2016a

Compressor

Dynamic range compressor
Library: Audio System Toolbox / Dynamic Range Control

Description

The Compressor block performs dynamic range compression independently across each input channel. Dynamic range compression attenuates the volume of loud sounds that cross a given threshold. The block uses specified attack and release times to achieve a smooth applied gain curve.

Ports

Input

x — Input signal
matrix | 1-D vector

- Matrix input –– Each column of the input is treated as an independent channel.
- 1-D vector input –– The input is treated as a single channel.

This port is unnamed unless you specify additional input ports.

Data Types: `single` | `double`

T — Threshold (dB)
scalar

Dependencies

To enable this port, select **Specify from input port** for the “Threshold (dB)” on page 5-0 parameter.

Data Types: `single` | `double`

R — Ratio
scalar

Dependencies

To enable this port, select **Specify from input port** for the “Ratio” on page 5-0 parameter.

Data Types: `single` | `double`

K — Knee width (dB)
scalar

Dependencies

To enable this port, select **Specify from input port** for the “Knee width (dB)” on page 5-0 parameter.

Data Types: `single` | `double`

AT — Attack time (s)
scalar

Dependencies

To enable this port, select **Specify from input port** for the “Attack time (s)” on page 5-0 parameter.

Data Types: `single` | `double`

RT — Release time (s)
scalar

Dependencies

To enable this port, select **Specify from input port** for the “Release time (s)” on page 5-0 parameter.

Data Types: `single` | `double`

Output

Y — Output signal
matrix

The Compressor block outputs a signal with the same data type as the input signal. The size of the output depends on the size of the input:

- Matrix input –– The block outputs a matrix the same size and data type as the input signal.
- 1-D vector input –– The block outputs an *N*-by-1 matrix (column vector), where *N* is the number of elements in the 1-D vector.

Data Types: `single` | `double`

`G` — Gain applied to each input sample
matrix

Dependencies

To enable this port, select the **Output gain (dB)** parameter.

Data Types: `single` | `double`

Parameters

If a parameter is listed as tunable, then you can change its value during simulation.

Main Tab

`Threshold (dB)` — Operation threshold
`–10` (default) | scalar in the range –50 to 0 inclusive

Operation threshold is the level above which gain is applied to the input signal.

To specify **Threshold (dB)** from an input port, select **Specify from input port** for the parameter.

Tunable: Yes

`Ratio` — Compression ratio
`5` (default) | scalar in the range 1 to 50 inclusive

Compression ratio is the input/output ratio for signals that overshoot the operation threshold.

Assuming a hard knee characteristic and a steady-state input such that $x[n]$ dB > **Threshold (dB)**, the compression ratio is defined as $R = \frac{(x[n]-T)}{(y[n]-T)}$, where

- R is the compression ratio.
- $x[n]$ is the input signal in dB.
- $y[n]$ is the output signal in dB.
- T is the threshold in dB.

To specify **Ratio** from an input port, select **Specify from input port** for the parameter.

Tunable: Yes

`Knee width (dB)` — Transition area in compression characteristic

`0` (default) | scalar in the range 0 to 20 inclusive

For soft knee characteristics, the transition area is defined by the relation

$$y = x + \frac{\left(\frac{1}{R} - 1\right) \times \left(x - T + \frac{W}{2}\right)^2}{(2 \times W)}$$

for the range $(2 \times |x - T|) \le W$, where

- y is the output level in dB.
- x is the input level in dB.
- R is the compression ratio.
- T is the threshold in dB.
- W is the knee width in dB.

To specify **Knee width (dB)** from an input port, select **Specify from input port** for the parameter.

Tunable: Yes

`View static characteristic` — Open static characteristic plot of dynamic range compressor
button

The plot is updated automatically when parameters of the Compressor block change.

Tunable: Yes

`Attack time (s)` — Time for applied gain to ramp up
`0.05` (default) | scalar in the range 0 to 4 inclusive

Attack time is the time the compressor gain takes to rise from 10% to 90% of its final value when the input goes above the threshold. The **Attack time (s)** parameter smooths the applied gain curve.

To specify **Attack time (s)** from an input port, select **Specify from input port** for the parameter.

Tunable: Yes

`Release time (s)` — Time for applied gain to ramp down
`0.2` (default) | scalar in the range 0 to 4 inclusive

Release time is the time the compressor gain takes to drop from 90% to 10% of its final value when the input goes below the threshold. The **Release time (s)** parameter smooths the applied gain curve.

To specify **Release time (s)** from an input port, select **Specify from input port** for the parameter.

Tunable: Yes

`Make-up gain mode` — Make-up gain mode
`Property` (default) | `Auto`

- `Property` –– Make-up gain is set to the value specified by the **Make-up gain (dB)** parameter.
- `Auto` –– Make-up gain is applied at the output of the Compressor block such that a steady-state 0 dB input has a 0 dB output.

Tunable: No

`Make-up gain (dB)` — Applied make-up gain

`0` (default) | scalar in the range –10 to 24 inclusive

Make-up gain compensates for gain lost during compression. It is applied at the output of the Compressor block.

Tunable: Yes

Dependencies

To enable this parameter, set the **Make-up gain mode** parameter to `Property`.

`Inherit sample rate from input` — Specify source of input sample rate

on (default) | off

When you select this parameter, the block inherits its sample rate from the input signal. When you clear this parameter, specify the sample rate in the **Input sample rate (Hz)** parameter.

Tunable: No

`Input sample rate (Hz)` — Sample rate of input

`44100` (default) | positive scalar

Tunable: Yes

Dependencies

To enable this parameter, clear the **Inherit sample rate from input** parameter.

Advanced Tab

`Output gain (dB)` — Gain applied on each input sample

off (default) | on

When you select this parameter, an additional output port, **G**, is added to the block. The **G** port outputs the gain applied on each input channel in dB.

Tunable: No

`Simulate using` — Specify type of simulation to run

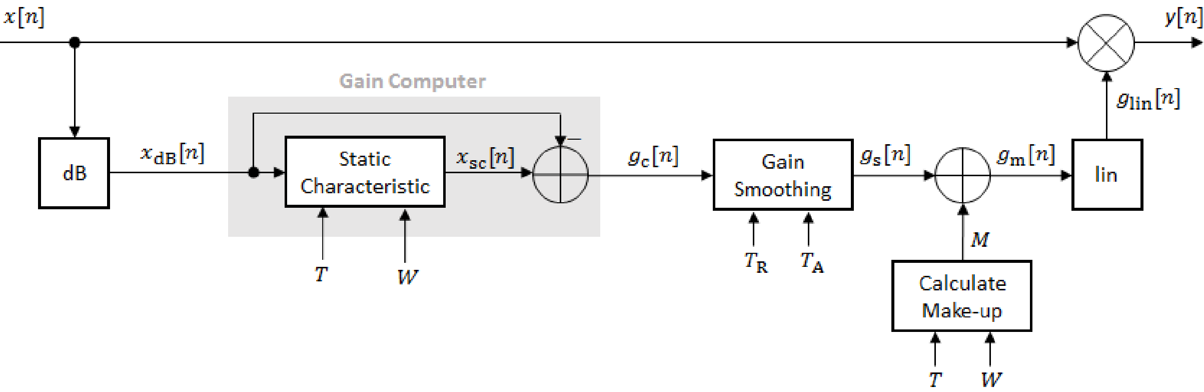
`Interpreted execution` (default) | `Code generation`

- `Interpreted execution` -- Simulate model using the MATLAB interpreter. This option shortens startup time and has a simulation speed comparable to `Code generation`. In this mode, you can debug the source code of the block.
- `Code generation` -- Simulate model using generated C code. The first time you run a simulation, Simulink generates C code for the block. The C code is reused for subsequent simulations, as long as the model does not change. This option requires additional startup time, but the speed of the subsequent simulations is comparable to `Interpreted execution`.

Tunable: No

Algorithms

The Compressor block processes a signal frame by frame and element by element.



1 The N-point signal, $x[n]$, is converted to decibels:

$$x_{dB}[n] = 20 \times \log_{10} |x[n]|$$

2 $x_{dB}[n]$ passes through the gain computer. The gain computer uses the static compression characteristic of the Compressor block to attenuate gain that is above the threshold.

If you specified a soft knee, the gain computer has the following static characteristic:

$$x_{sc}(x_{dB}) = \begin{cases} x_{dB} & x_{dB} < \left(T - \frac{W}{2}\right) \\ x_{dB} + \frac{\left(\frac{1}{R} - 1\right)\left(x_{dB} - T + \frac{W}{2}\right)^2}{2W} & \left(T - \frac{W}{2}\right) \le x_{dB} \le \left(T + \frac{W}{2}\right) \\ T + \frac{(x_{dB} - T)}{R} & x_{dB} > \left(T + \frac{W}{2}\right) \end{cases},$$

where T is the threshold, R is the compression ratio, and W is the knee width.

If you specified a hard knee, the gain computer has the following static characteristic:

$$x_{sc}(x_{dB}) = \begin{cases} x_{dB} & x_{dB} < T \\ T + \frac{(x_{dB} - T)}{R} & x_{dB} \ge T \end{cases}$$

3 The computed gain, $g_c[n]$, is calculated as

$$g_c[n] = x_{sc}[n] - x_{dB}[n].$$

4 $g_c[n]$ is smoothed using specified attack and release time parameters:

$$g_s[n] = \begin{cases} \alpha_A g_s[n-1] + (1 - \alpha_A) g_c[n], & g_c[n] > g_s[n-1] \\ \alpha_R g_s[n-1] + (1 - \alpha_R) g_c[n], & g_c[n] \le g_s[n-1] \end{cases}$$

The attack time coefficient, α_A, is calculated as

$$\alpha_A = \exp\left(\frac{-\log(9)}{Fs \times T_A}\right).$$

The release time coefficient, α_R, is calculated as

$$\alpha_R = \exp\left(\frac{-\log(9)}{Fs \times T_R}\right).$$

T_A is the attack time period, specified by the **Attack time (s)** parameter. T_R is the release time period, specified by the **Release time (s)** parameter. *Fs* is the input sampling rate, specified by the **Inherit sample rate from input** or the **Input sample rate (Hz)** parameter.

5. If **Make-up gain (dB)** is set to `Auto`, the make-up gain is calculated as the negative of the computed gain for a 0 dB input:

$$M = -x_{sc}\Big|_{x_{dB}=0}.$$

   Given a steady-state input of 0 dB, this configuration achieves a steady-state output of 0 dB. The make-up gain is determined by the **Threshold (dB)**, **Ratio**, and **Knee width (dB)** parameters. It does not depend on the input signal.

6. The make-up gain, *M*, is added to the smoothed gain, $g_s[n]$:

$$g_m[n] = g_s[n] + M$$

7. The calculated gain in dB, $g_{dB}[n]$, is translated to a linear domain:

$$g_{lin}[n] = 10^{\left(\frac{g_m[n]}{20}\right)}$$

8. The output of the dynamic range compressor is given as

$$y[n] = x[n] \times g_{lin}[n].$$

References

[1] Giannoulis, Dimitrios, Michael Massberg, and Joshua D. Reiss. "Digital Dynamic Range Compressor Design –– A Tutorial And Analysis." *Journal of Audio Engineering Society*. Vol. 60, Issue 6, 2012, pp. 399–408.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

Blocks
Expander | Limiter | Noise Gate

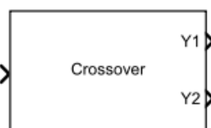
System Objects
`compressor`

Topics
"Dynamic Range Control"

Introduced in R2016a

Crossover Filter

Audio crossover filter
Library: Audio System Toolbox / Filters



Description

The Crossover Filter block implements an audio crossover filter, which is used to split an audio signal into two or more frequency bands. Crossover filters are multiband filters whose overall magnitude frequency response is flat.

Ports

Input

`Port_1` — Input signal
matrix | 1-D vector

- Matrix input –– Each column of the input is treated as an independent channel.
- 1-D vector input –– The input is treated as a single channel.

Data Types: `single` | `double`

Output

Y1 — Output signal
matrix

Available if **Number of crossovers** is set to 1, 2, 3, or 4. Port **Y1** always corresponds to a lowpass filter.

Data Types: `single` | `double`

Y2 — Output signal
matrix

Depending on the number of crossovers specified, port **Y2** outputs the original audio signal passed through a bandpass or highpass filter.

Available if **Number of crossovers** is set to 1, 2, 3, or 4.

Data Types: single | double

Y3 — Output signal
matrix

Depending on the number of crossovers specified, port **Y3** corresponds to a bandpass or highpass filter of the original audio signal.

Available if **Number of crossovers** is set to 2, 3, or 4.

Data Types: single | double

Y4 — Output signal
matrix

Available if **Number of crossovers** is set to 3 or 4.

Data Types: single | double

Y5 — Output signal
matrix

Available if **Number of crossovers** is set to 4.

Data Types: single | double

Parameters

If a parameter is listed as tunable, then you can change its value during simulation.

Number of crossovers — Number of magnitude response band crossings
1 (default) | 2 | 3 | 4

If you specify multiple crossovers, the corresponding **Crossover frequency (Hz)** and **Crossover order** parameters populate in the dialog box automatically.

The number of bands output by the Crossover Filter block is one more than the **Number of crossovers**.

Number of Crossovers	Number of Bands Output
1	two bands
2	three bands
3	four bands
4	five bands

Tunable: No

`Crossover frequency (Hz)` — Intersections of magnitude response bands
`100` (default) | real scalar in the range `20` to `20000`

Crossover frequencies are the intersections of magnitude response bands of the individual two-band crossover filters used in the multiband crossover filter.

Tunable: Yes

`Crossover order` — Order of individual crossover filters
2 (default) | 1 | 3 | 4 | 5 | 6 | 7 | 8

The crossover filter order relates to the crossover filter slope in dB/octave: $slope = N \times 6$, where N is the crossover order.

Tunable: Yes

`View filter response` — Open plot of magnitude response of each filter band
button

The plot is updated automatically when parameters of the Crossover Filter block change.

Tunable: Yes

`Inherit sample rate from input` — Specify source of input sample rate
off (default) | on

When you select this parameter, the block inherits its sample rate from the input signal. When you clear this parameter, you specify the sample rate in **Input sample rate (Hz)**.

Tunable: No

`Input sample rate (Hz)` — Sample rate of input
`44100` (default) | positive scalar

Tunable: Yes

Dependencies

To enable this parameter, clear the **Inherit sample rate from input** parameter.

`Simulate using` — Specify type of simulation to run
`Interpreted execution` (default) | `Code generation`

- `Interpreted execution` -– Simulate model using the MATLAB interpreter. This option shortens startup time and has simulation speed comparable to `Code generation`. In this mode, you can debug the source code of the block.
- `Code generation` -– Simulate model using generated C code. The first time you run a simulation, Simulink generates C code for the block. The C code is reused for subsequent simulations, as long as the model does not change. This option requires additional startup time but the speed of the subsequent simulations is comparable to `Interpreted execution`.

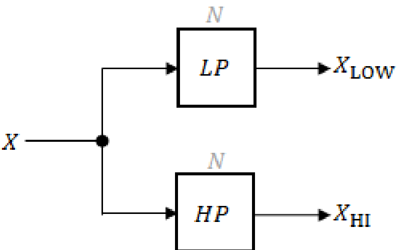
Tunable: No

Algorithms

The Crossover Filter block is implemented as a binary tree of crossover pairs with additional phase-compensating sections [1]. Odd-order crossovers are implemented with Butterworth filters, while even-order crossovers are implemented with cascaded Butterworth filters (Linkwitz-Riley filters).

Odd-Order Crossover Pair

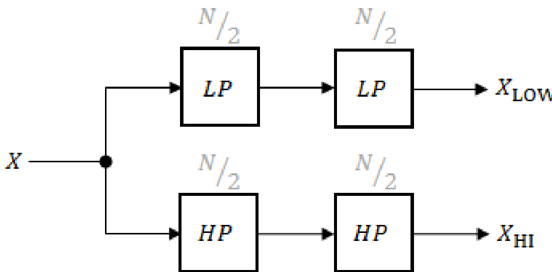
Odd-order two-band (one crossover) filters are implemented as parallel complementary highpass and lowpass filters.



LP and *HP* are Butterworth filters of order *N*, implemented as direct-form II transposed second-order sections. The shared cutoff frequency used in their design corresponds to the crossover of the resulting bands.

Even-Order Crossover Pair

Even-order two-band (one crossover) filters are implemented as parallel complementary highpass and lowpass filters.

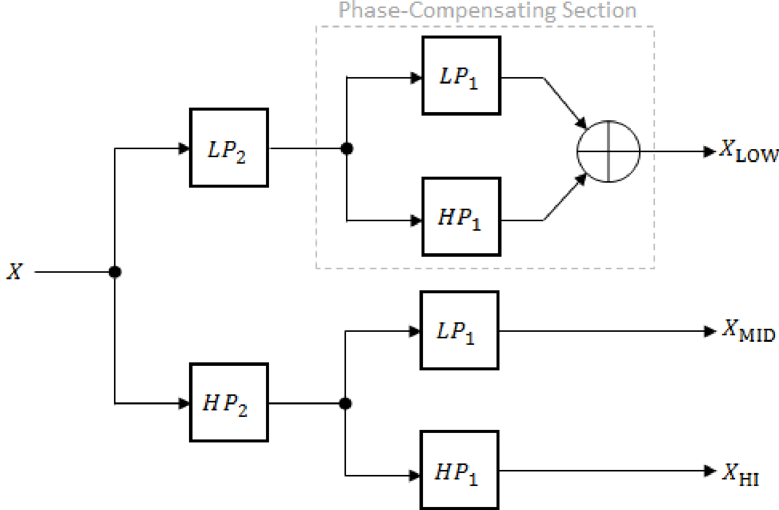


LP and *HP* are Butterworth filters of order *N/2*, where *N* is the order of the overall filter. The filters are implemented as direct-form II transposed second-order sections.

For overall filters of orders 2 and 6, X_{HI} is multiplied by –1 internally so that the branches of your crossover pair are in-phase.

Even-Order Three-Band Filter

Even-order three-band (two crossovers) filters are implemented as parallel complementary highpass and lowpass filters organized in a tree structure.



The phase-compensating section is equivalent to an allpass filter.

The design of four-band and five-band filters (three and four crossovers) are extensions of the pattern developed for even-order and odd-order crossovers and the tree structure specified for three-band (two crossover) filters.

References

[1] D'Appolito, Joseph A. "Active Realization of Multiway All-Pass Crossover Systems." *Journal of Audio Engineering Society*. Vol. 35, Issue 4, 1987, pp. 239–245.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

System Objects
`crossoverFilter`

Topics

"Multiband Dynamic Range Compression"

Introduced in R2016a

Expander

Dynamic range expander
Library: Audio System Toolbox / Dynamic Range Control

Description

The Expander block performs dynamic range expansion independently across each input channel. Dynamic range expansion attenuates the volume of quiet sounds below a given threshold. The block uses specified attack, release, and hold times to achieve a smooth applied gain curve.

Ports

Input

x — Input signal
matrix | 1-D vector

- Matrix input –– Each column of the input is treated as an independent channel.
- 1-D vector input –– The input is treated as a single channel.

This port is unnamed unless you specify additional input ports.

Data Types: `single` | `double`

R — Ratio
scalar

Dependencies

To enable this port, select **Specify from input port** for the “Ratio” on page 5-0 parameter.

Data Types: `single` | `double`

T — Threshold (dB)
scalar

Dependencies

To enable this port, select **Specify from input port** for the "Threshold (dB)" on page 5-0 parameter.

Data Types: `single` | `double`

K — Knee width (dB)
scalar

Dependencies

To enable this port, select **Specify from input port** for the "Knee width (dB)" on page 5-0 parameter.

Data Types: `single` | `double`

AT — Attack time (s)
scalar

Dependencies

To enable this port, select **Specify from input port** for the "Attack time (s)" on page 5-0 parameter.

Data Types: `single` | `double`

RT — Release time (s)
scalar

Dependencies

To enable this port, select **Specify from input port** for the "Release time (s)" on page 5-0 parameter.

Data Types: `single` | `double`

HT — Hold time (s)
scalar

Dependencies

To enable this port, select **Specify from input port** for the "Hold time (s)" on page 5-0 parameter.

Data Types: `single` | `double`

Output

Y — Output signal
matrix

The Expander block outputs a signal with the same data type as the input signal. The size of the output depends on the size of the input:

- Matrix input –– The block outputs a matrix the same size and data type as the input signal.
- 1-D vector input –– The block outputs an *N*-by-1 matrix (column vector), where *N* is the number of elements in the 1-D vector.

This port is unnamed until you select the **Output gain (dB)** parameter.

Data Types: `single` | `double`

G — Gain applied to each input sample
matrix

Dependencies

To enable this port, select the **Output gain (dB)** parameter.

Data Types: `single` | `double`

Parameters

If a parameter is listed as tunable, then you can change its value during simulation.

Main Tab

Ratio — Expansion ratio
5 (default) | scalar in the range 1 to 50 inclusive

Expansion ratio is the input/output ratio for signals that undershoot the operation threshold.

Assuming a hard knee characteristic and a steady-state input such that $x[n]$ dB < **Threshold (dB)**, the expansion ratio is defined as $R = \frac{(y[n]-T)}{(x[n]-T)}$, where

- *R* is the expansion ratio.
- $y[n]$ is the output signal in dB.
- $x[n]$ is the input signal in dB.
- *T* is the threshold in dB.

To specify **Ratio** from an input port, select **Specify from input port** for the parameter.

Tunable: Yes

`Threshold (dB)` — Operation threshold

`–10` (default) | scalar in the range –140 to 0 inclusive

Operation threshold is the level below which gain is applied to the input signal.

To specify **Threshold (dB)** from an input port, select **Specify from input port** for the parameter.

Tunable: Yes

`Knee width (dB)` — Transition area in the compression characteristic

`0` (default) | scalar in the range 0 to 20

For soft knee characteristics, the transition area is defined by the relation

$$y = x + \frac{(1-R)\times\left(x - T - \frac{W}{2}\right)^2}{(2\times W)}$$

for the range $(2\times|x-T|) \leq W$, where

- *y* is the output level in dB.
- *x* is the input level in dB.
- *R* is the expansion ratio.
- *T* is the threshold in dB.
- *W* is the knee width in dB.

To specify **Knee width (dB)** from an input port, select **Specify from input port** for the parameter.

Tunable: Yes

`View static characteristic` — Open static characteristic plot of dynamic range expander

button

The plot is updated automatically when parameters of the Expander block change.

Tunable: Yes

`Attack time (s)` — Time for applied gain to ramp up

`0.05` (default) | scalar in the range 0 to 4 inclusive

Attack time is the time the expander gain takes to rise from 10% to 90% of its final value when the input goes below the threshold. The **Attack time (s)** parameter smooths the applied gain curve.

To specify **Attack time (s)** from an input port, select **Specify from input port** for the parameter.

Tunable: Yes

`Release time (s)` — Time for applied gain to ramp down

`0.2` (default) | scalar in the range 0 to 4 inclusive

Release time is the time the expander gain takes to drop from 90% to 10% of its final value when the input goes above the threshold. The **Release time (s)** parameter smooths the applied gain curve.

To specify **Release time (s)** from an input port, select **Specify from input port** for the parameter.

Tunable: Yes

`Hold time (s)` — Time during which applied gain holds steady

`0.05` (default) | scalar in the range 0 to 4 inclusive

Hold time is the period in which the applied gain is held constant before it starts moving toward its steady-state value. Hold time begins when the input level crosses the operation threshold.

To specify **Hold time (s)** from an input port, select **Specify from input port** for the parameter.

Tunable: Yes

`Inherit sample rate from input` — Specify source of input sample rate
on (default) | off

When you select this parameter, the block inherits its sample rate from the input signal. When you clear this parameter, specify the sample rate in the **Input sample rate (Hz)** parameter.

Tunable: No

`Input sample rate (Hz)` — Sample rate of input
`44100` (default) | positive scalar

Tunable: Yes

Dependencies

To enable this parameter, clear the **Inherit sample rate from input** parameter.

Advanced Tab

`Output gain (dB)` — Gain applied on each input sample
off (default) | on

When you select this parameter, an additional output port, **G**, is added to the block. The **G** port outputs the gain applied on each input channel in dB.

Tunable: No

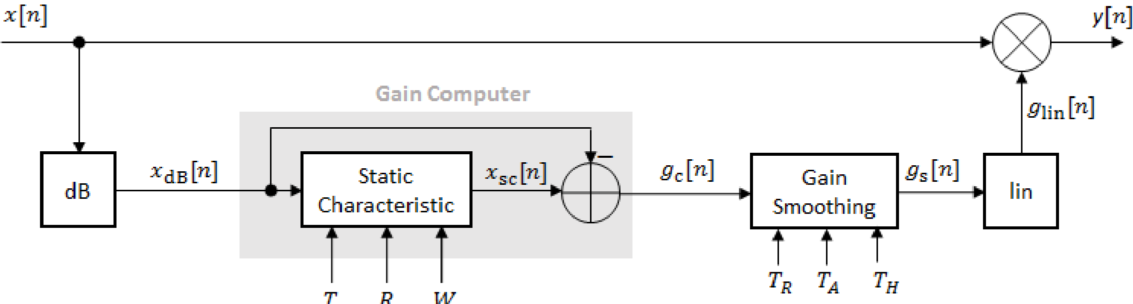
`Simulate using` — Specify type of simulation to run
`Interpreted execution` (default) | `Code generation`

- `Interpreted execution` –– Simulate model using the MATLAB interpreter. This option shortens startup time and has a simulation speed comparable to `Code generation`. In this mode, you can debug the source code of the block.
- `Code generation` –– Simulate model using generated C code. The first time you run a simulation, Simulink generates C code for the block. The C code is reused for subsequent simulations, as long as the model does not change. This option requires additional startup time, but the speed of the subsequent simulations is comparable to `Interpreted execution`.

Tunable: No

Algorithms

The Expander block processes a signal frame by frame and element by element.



1. The *N*-point signal, $x[n]$, is converted to decibels:

   $$x_{dB}[n] = 20 \times \log_{10}|x[n]|$$

2. $x_{dB}[n]$ passes through the gain computer. The gain computer uses the static characteristic properties of the dynamic range expander to attenuate gain that is below the threshold.

   If you specified a soft knee, the gain computer has the following static characteristic:

   $$x_{sc}(x_{dB}) = \begin{cases} T + (x_{dB} - T) \times R & x_{dB} < \left(T - \frac{W}{2}\right) \\ x_{dB} + \frac{(1-R)\left(x_{dB} - T - \frac{W}{2}\right)^2}{2W} & \left(T - \frac{W}{2}\right) \leq x_{dB} \leq \left(T + \frac{W}{2}\right) \\ x_{dB} & x_{dB} > \left(T + \frac{W}{2}\right) \end{cases},$$

   where *T* is the threshold, *R* is the expansion ratio, and *W* is the knee width.

   If you specified a hard knee, the gain computer has the following static characteristic:

$$x_{sc}(x_{dB}) = \begin{cases} T + (x_{dB} - T) \times R & x_{dB} < T \\ x_{dB} & x_{dB} \geq T \end{cases}$$

3 The computed gain, $g_c[n]$, is calculated as

$$g_c[n] = x_{sc}[n] - x_{dB}[n].$$

4 $g_c[n]$ is smoothed using specified attack, release, and hold time parameters:

$$g_s[n] = \begin{cases} \alpha_A g_s[n-1] + (1-\alpha_A) g_c[n] & (C_A > T_H) \ \& \ (g_c[n] > g_s[n-1]) \\ g_s[n-1] & C_A \leq T_H \\ \alpha_R g_s[n-1] + (1-\alpha_R) g_c[n] & (C_R > T_H) \ \& \ (g_c[n] \leq g_s[n-1]) \\ g_s[n-1] & C_R \leq T_H \end{cases}$$

C_A and C_R are hold counters for attack and release, respectively. The limit, T_H, is determined by the **Hold time (s)** parameter.

The attack time coefficient, α_A, is calculated as

$$\alpha_A = \exp\left(\frac{-\log(9)}{Fs \times T_A}\right).$$

The release time coefficient, α_R, is calculated as

$$\alpha_R = \exp\left(\frac{-\log(9)}{Fs \times T_R}\right).$$

T_A is the attack time period, specified by the **Attack time (s)** parameter. T_R is the release time period, specified by the **Release time (s)** parameter. *Fs* is the input sampling rate, specified by the **Inherit sample rate from input** or **Input sample rate (Hz)** parameter.

5 The smoothed gain in dB, $g_s[n]$, is translated to a linear domain:

$$g_{lin}[n] = 10^{\left(\frac{g_s[n]}{20}\right)}.$$

6 The output of the dynamic range expander is given as

$$y[n] = x[n] \times g_{\text{lin}}[n].$$

References

[1] Giannoulis, Dimitrios, Michael Massberg, and Joshua D. Reiss. "Digital Dynamic Range Compressor Design –– A Tutorial And Analysis." *Journal of Audio Engineering Society*. Vol. 60, Issue 6, 2012, pp. 399–408.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

Blocks
Compressor | Limiter | Noise Gate

System Objects
`expander`

Topics

“Dynamic Range Control”

Introduced in R2016a

Graphic EQ

Standards-based graphic equalizer
Library: Audio System Toolbox / Filters

Description

The Graphic EQ block implements a graphic equalizer that can tune the gain on individual octave or fractional octave bands. The block filters the data independently across each input channel over time using the filter specifications. Center frequencies for bands in the graphic equalizer are based on the ANSI S1.11-2004 standard.

Ports

Input

`Port_1` — Input signal
matrix | 1-D vector

- Matrix input –– Each column of the input is treated as an independent channel.
- 1-D vector input –– The input is treated as a signal channel.

Data Types: `single` | `double`

Output

`Port_1` — Output signal
matrix

The Graphic EQ block outputs a signal with the same data type as the input signal. The size of the output depends on the size of the input:

- Matrix input –– The block outputs a matrix the same size and data type as the input signal.
- 1-D vector input –– The block outputs an *N*-by-1 matrix (column vector), where *N* is the number of elements in the 1-D vector input.

Data Types: `single` | `double`

Parameters

If a parameter is listed as tunable, then you can change its value during simulation.

`EQ Order` — Order of individual equalizer bands
2 (default) | positive even integer

Specify the order of individual equalizer bands as a positive even integer. All equalizer bands have the same order.

Tunable: Yes

`Bandwidth` — Filter bandwidth (octaves)
`1 octave` (default) | `2/3 octave` | `1/3 octave`

Specify the filter bandwidth as `1 octave`, `2/3 octave`, or `1/3 octave`.

The ANSI S1.11-2004 standard defines the center and edge frequencies of your equalizer. The ISO 266:1997(E) standard specifies corresponding preferred frequencies for labeling purposes.

1-Octave Bandwidth

Center frequencies	`32 63 126 251 501 1000 1995 3981 7943 15849`
Edge frequencies	`22 45 89 178 355 708 1413 2818 5623 1122 22387`
Preferred frequencies	`31.5 63 125 250 500 1000 2000 4000 8000 16000`

2/3-Octave Bandwidth

Center frequencies	25 40 63 100 158 251 398 631 1000 1585 2512 3981 6310 10000 15849
Edge frequencies	20 32 50 79 126 200 316 501 794 1259 1995 3162 5012 7943 12589 19953
Preferred frequencies	25 40 63 100 160 250 400 630 1000 1600 2500 4000 6300 10000 16000

1/3-Octave Bandwidth

Center frequencies	25 32 40 50 63 79 100 126 158 200 251 316 398 501 631 794 1000 1259 1585 1995 2512 3162 3981 5012 6310 7943 10000 12589 15849 19953
Edge frequencies	22 28 35 45 56 71 89 112 141 178 224 282 355 447 562 708 891 1122 1413 1778 2239 2818 3548 4467 5623 7079 8913 11220 14125 17783 22387
Preferred frequencies	25 31.5 40 50 63 80 100 125 160 200 250 315 400 500 630 800 1000 1250 1600 2000 2500 3150 4000 5000 6300 8000 10000 12500 16000 20000

Tunable: Yes

`Structure` — Type of implementation

`Cascade` (default) | `Parallel`

Specify the type of implementation as `Cascade` or `Parallel`. See “Algorithms” on page 5-59 and “Graphic Equalization” for information about these implementation structures.

Tunable: No

`Gains` — Gain of each octave or fractional octave band (dB)

`0` | scalar

Specify the gain of each octave or fractional octave band in dB. The number and position of filters in the graphic equalizer depends on the **Bandwidth** on page 5-0 parameter.

Tunable: Yes

`Inherit sample rate from input` — Specify source of input sample rate
off (default) | on

When you select this parameter, the block inherits its sample rate from the input signal. When you clear this parameter, specify the sample rate in **Input sample rate (Hz)** on page 5-0.

Tunable: No

`Input sample rate (Hz)` — Sample rate of input
`44100` (default) | scalar

Tunable: Yes

Dependencies

To enable this parameter, clear the **Inherit sample rate from input** on page 5-0 parameter.

`Simulate using` — Specify type of simulation to run
`Code generation` (default) | `Interpreted execution`

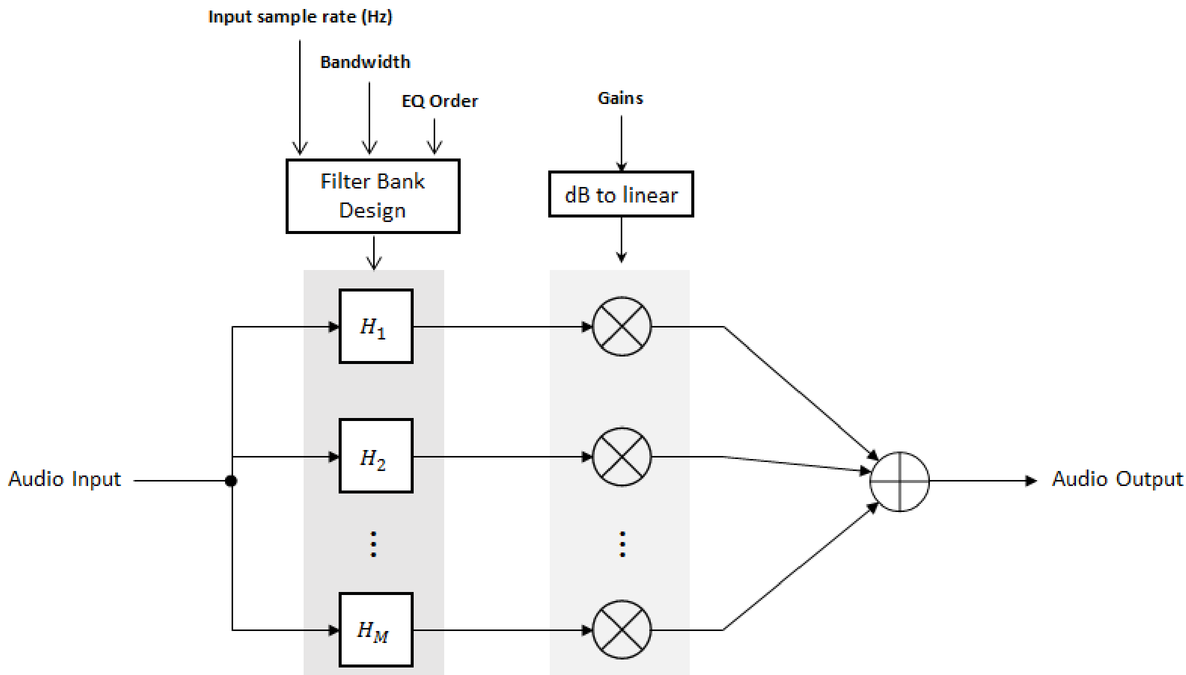
- `Interpreted execution` –– Simulate model using the MATLAB interpreter. This option shortens startup time and has simulation speed comparable to `Code generation`. In this mode, you can debug the source code of the block.
- `Code generation` –– Simulate model using generated C code. The first time you run a simulation, Simulink generates C code for the block. The C code is reused for subsequent simulations, as long as the model does not change. This option requires additional startup time but the speed of the subsequent simulations is faster than `Interpreted execution`.

Tunable: No

Algorithms

The implementation of your graphic equalizer depends on the **Structure** on page 5-0 parameter. See “Graphic Equalization” for a discussion of the pros and cons of the parallel and cascade implementations. Refer to the following sections to understand how these algorithms are implemented in Audio System Toolbox.

Parallel Structure



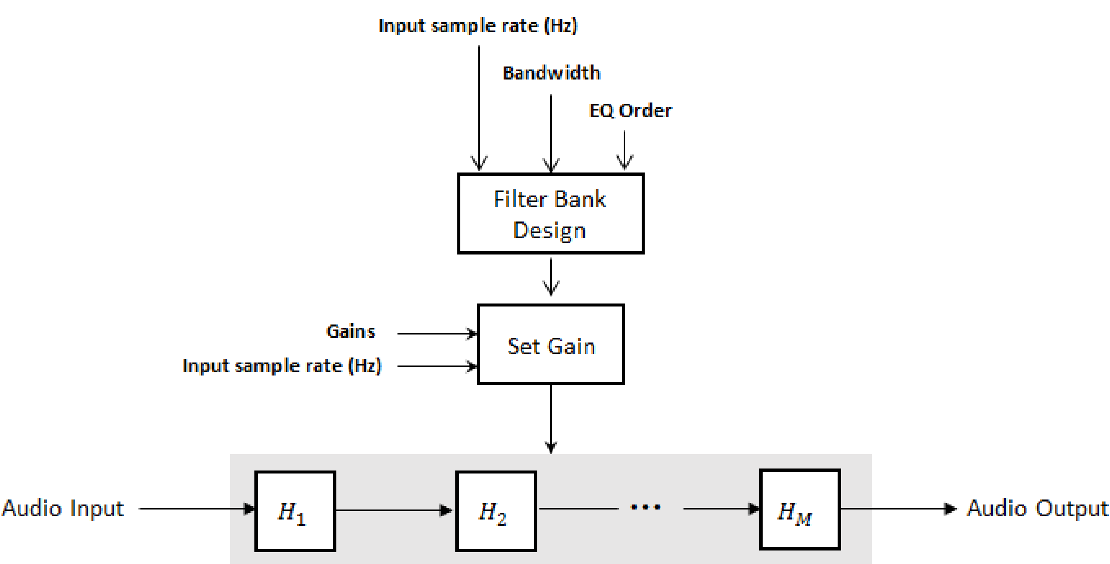
The parallel implementation designs the individual equalizers using the `octaveFilter design method` and spaces them on the spectrum according to the ANSI S1.11-2004 standard.

If you set the **Input sample rate (Hz)** parameter so that the Nyquist frequency (**Input sample rate (Hz)**`/2`) is less than the final bandpass edge defined by the ANSI S1.11-2004 standard, then:

- The final bandpass filter is the one whose upper bandpass edge is less than the Nyquist frequency.
- The final filter is implemented as a highpass filter designed by the `designParamEQ` function.

1. The input signal is fed into a filterbank of M filters, where M depends on the specified **Bandwidth** and **Input sample rate (Hz)** parameters.
2. Each branch of the filterbank is multiplied by the linear form of the corresponding element of the **Gains** parameter.
3. The branches are summed and the output signal is returned.

Cascade Structure



The cascade implementation designs the graphic equalizer filter bank using the `multibandParametricEQ` System object.

If the **EQ Order** on page 5-0 parameter is set to 2, then a gain correction is calculated according to [1]. The gain correction is independent of the requested gains. The gain correction is recomputed during the real-time processing only if the **Input sample rate (Hz)** parameter is modified.

If the **EQ Order** parameter is not set to 2, no gain correction is applied and the requested gains are passed on to the `multibandParametricEQ` object.

The input signal is fed into a cascade of *M* biquad filters, where *M* depends on the specified **Bandwidth** and **Input sample rate (Hz)** parameters.

References

[1] Oliver, Richard J., and Jean-Marc Jot. "Efficient Multi-Band Digital Audio Graphic Equalizer with Accurate Frequency Response Control." Presented at the 139th Convention of the AES, New York, October 2015.

[2] Acoustical Society of America. *American National Standard Specification for Octave-Band and Fractional-Octave-Band Analog and Digital Filters*. ANSI S1.11-2004. Melville, NY: Acoustical Society of America, 2009.

[3] International Organization for Standardization. *Acoustics –– Preferred frequencies*. ISO 266:1997(E). Second Edition. 1997.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

System Objects

`graphicEQ` | `multibandParametricEQ`

Functions

`designParamEQ` | `designShelvingEQ` | `designVarSlopeFilter`

Topics

“Parametric Equalizer Design”

“Equalization”

Introduced in R2017b

Limiter

Dynamic range limiter
Library: Audio System Toolbox / Dynamic Range Control

Description

The Limiter block performs dynamic range limiting independently across each input channel. Dynamic range limiting suppresses the volume of loud sounds that cross a given threshold. The block uses specified attack and release times to achieve a smooth applied gain curve.

Ports

Input

x — Input signal
1-D vector | matrix

- Matrix input –– Each column of the input is treated as an independent channel.
- 1-D vector input –– The input is treated as a single channel.

This port is unnamed unless you specify additional input ports.

Data Types: `single` | `double`

T — Threshold (dB)
scalar

Dependencies

To enable this port, select **Specify from input port** for the “Threshold (dB)” on page 5-0 parameter.

Data Types: `single` | `double`

K — Knee width (dB)

scalar

Dependencies

To enable this port, select **Specify from input port** for the “Knee width (dB)” on page 5-0 parameter.

Data Types: `single` | `double`

AT — Attack time (s)

scalar

Dependencies

To enable this port, select **Specify from input port** for the “Attack time (s)” on page 5-0 parameter.

Data Types: `single` | `double`

RT — Release time (s)

scalar

Dependencies

To enable this port, select **Specify from input port** for the “Release time (s)” on page 5-0 parameter.

Data Types: `single` | `double`

Output

Y — Output signal

matrix

The Limiter block outputs a signal with the same data type as the input signal. The size of the output depends on the size of the input:

- Matrix input –– The block outputs a matrix the same size and data type as the input signal.
- 1-D vector input –– The block outputs an *N*-by-1 matrix (column vector), where *N* is the number of elements in the 1-D vector.

This port is unnamed until you select the **Output gain (dB)** parameter.

Data Types: `single` | `double`

G — Gain applied to each input sample
matrix

Dependencies

To enable this port, select the **Output gain (dB)** parameter.

Data Types: `single` | `double`

Parameters

If a parameter is listed as tunable, then you can change its value during simulation.

Main Tab

`Threshold (dB)` — Operation threshold
`–10` (default) | scalar in the range –50 to 0 inclusive

Operation threshold is the level above which gain is applied to the input signal.

To specify **Threshold (dB)** from an input port, select **Specify from input port** for the parameter.

Tunable: Yes

`Knee width (dB)` — Transition area in the limiter characteristic
`0` (default) | scalar in the range 0 to 20 inclusive

For soft knee characteristics, the transition area is defined by the relation

$$y = x - \frac{\left(x - T + \frac{W}{2}\right)^2}{(2 \times W)}$$

for the range $(2 \times |x - T|) \leq W$, where

- y is the output level in dB.

- *x* is the input level in dB.
- *T* is the threshold in dB.
- *W* is the knee width in dB.

To specify **Knee width (dB)** from an input port, select **Specify from input port** for the parameter.

Tunable: Yes

`View static characteristic` — Open static characteristic plot of dynamic range limiter

button

The plot is updated automatically when parameters of the Limiter block change.

Tunable: Yes

`Attack time (s)` — Time for applied gain to ramp up

`0` (default) | scalar in the range 0 to 4 inclusive

Attack time is the time the limiter gain takes to rise from 10% to 90% of its final value when the input goes above the threshold. The **Attack time (s)** parameter smooths the applied gain curve.

To specify **Attack time (s)** from an input port, select **Specify from input port** for the parameter.

Tunable: Yes

`Release time (s)` — Time for applied gain to ramp down

`0.2` (default) | scalar in the range 0 to 4 inclusive

Release time is the time the limiter gain takes to drop from 90% to 10% of its final value when the input goes below the threshold. The **Release time (s)** parameter smooths the applied gain curve.

To specify **Release time (s)** from an input port, select **Specify from input port** for the parameter.

Tunable: Yes

`Make-up gain mode` — Make-up gain mode
`Property` (default) | `Auto`

- `Property` –– Make-up gain is set to the value specified by the **Make-up gain (dB)** parameter.
- `Auto` –– Make-up gain is applied at the output of the Limiter block such that a steady-state 0 dB input has a 0 dB output.

Tunable: No

`Make-up gain (dB)` — Applied make-up gain
`0` (default) | scalar in the range –10 to 24 inclusive

Make-up gain compensates for gain lost during limiting. It is applied at the output of the Limiter block.

Tunable: Yes

Dependencies

To enable this parameter, set the **Make-up gain mode** parameter to `Property`.

`Inherit sample rate from input` — Specify source of input sample rate
on (default) | off

When you select this parameter, the block inherits its sample rate from the input signal. When you clear this parameter, specify the sample rate in the **Input sample rate (Hz)** parameter.

Tunable: No

`Input sample rate (Hz)` — Specify input sample rate
`44100` (default) | positive scalar

Tunable: Yes

Dependencies

To enable this parameter, clear the **Inherit sample rate from input** parameter.

Advanced Tab

`Output gain (dB)` — Gain applied on each input sample
off (default) | on

When you select this parameter, an additional output port, **G**, is added to the block. The **G** port outputs the gain applied on each input channel in dB.

Tunable: No

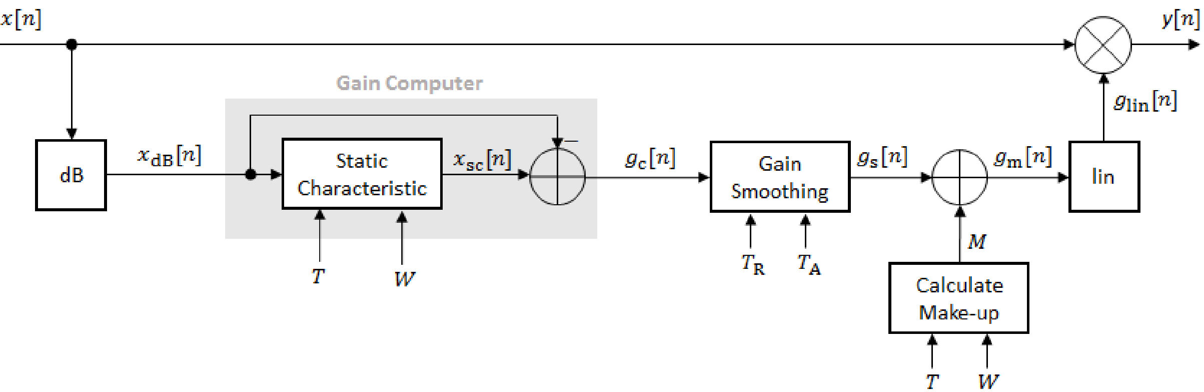
`Simulate using` — Specify type of simulation to run
`Interpreted execution` (default) | `Code generation`

- `Interpreted execution` -- Simulate model using the MATLAB interpreter. This option shortens startup time and has a simulation speed comparable to `Code generation`. In this mode, you can debug the source code of the block.
- `Code generation` -- Simulate model using generated C code. The first time you run a simulation, Simulink generates C code for the block. The C code is reused for subsequent simulations, as long as the model does not change. This option requires additional startup time, but the speed of the subsequent simulations is comparable to `Interpreted execution`.

Tunable: No

Algorithms

The Limiter block processes a signal frame by frame and element by element.



1 The *N*-point signal, *x*[*n*], is converted to decibels:

$$x_{\mathrm{dB}}[n] = 20 \times \log_{10} |x[n]|$$

2 $x_{dB}[n]$ passes through the gain computer. The gain computer uses the static characteristic properties of the dynamic range limiter to brickwall gain that is above the threshold.

If you specified a soft knee, the gain computer has the following static characteristic:

$$x_{sc}(x_{dB}) = \begin{cases} x_{dB} & x_{dB} < \left(T - \frac{W}{2}\right) \\ x_{dB} - \frac{\left(x_{dB} - T + \frac{W}{2}\right)^2}{2W} & \left(T - \frac{W}{2}\right) \le x_{dB} \le \left(T + \frac{W}{2}\right) \\ T & x_{dB} > \left(T + \frac{W}{2}\right) \end{cases},$$

where T is the threshold and W is the knee width.

If you specified a hard knee, the gain computer has the following static characteristic:

$$x_{sc}(x_{dB}) = \begin{cases} x_{dB} & x_{dB} < T \\ T & x_{dB} \ge T \end{cases}$$

3 The computed gain, $g_c[n]$, is calculated as

$$g_c[n] = x_{sc}[n] - x_{dB}[n].$$

4 $g_c[n]$ is smoothed using specified attack and release time parameters:

$$g_s[n] = \begin{cases} \alpha_A g_s[n-1] + (1-\alpha_A) g_c[n], & g_c[n] > g_s[n-1] \\ \alpha_R g_s[n-1] + (1-\alpha_R) g_c[n], & g_c[n] \le g_s[n-1] \end{cases}$$

The attack time coefficient, α_A, is calculated as

$$\alpha_A = \exp\left(\frac{-\log(9)}{Fs \times T_A}\right).$$

The release time coefficient, α_R , is calculated as

$$\alpha_R = \exp\left(\frac{-\log(9)}{Fs \times T_R}\right).$$

T_A is the attack time period, specified by the **Attack time (s)** parameter. T_R is the release time period, specified by the **Release time (s)** parameter. *Fs* is the input sampling rate, specified by the **Inherit sample rate from input** or **Input sample rate (Hz)** parameter.

5 If the **Make-up gain (dB)** parameter is set to `Auto`, the make-up gain is calculated as the negative of the computed gain for a 0 dB input:

$$M = -x_{sc}(x_{dB} = 0)$$

Given a steady-state input of 0 dB, this configuration achieves a steady-state output of 0 dB. The make-up gain is determined by the **Threshold (dB)** and **Knee width (dB)** parameters. It does not depend on the input signal.

6 The make-up gain, M, is added to the smoothed gain, $g_s[n]$:

$$g_m[n] = g_s[n] + M$$

7 The calculated gain in dB, $g_m[n]$, is translated to a linear domain:

$$g_{lin}[n] = 10^{\left(\frac{g_m[n]}{20}\right)}$$

8 The output of the dynamic range limiter is given as

$$y[n] = x[n] \times g_{lin}[n].$$

References

[1] Giannoulis, Dimitrios, Michael Massberg, and Joshua D. Reiss. "Digital Dynamic Range Compressor Design –– A Tutorial And Analysis." *Journal of Audio Engineering Society*. Vol. 60, Issue 6, 2012, pp. 399–408.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

Blocks

Compressor | Expander | Noise Gate

System Objects

`limiter`

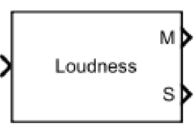
Topics

"Dynamic Range Control"

Introduced in R2016a

Loudness Meter

Standard-compliant loudness measurements
Library: Audio System Toolbox / Measurements



Description

The Loudness Meter block measures the loudness and true-peak of an audio signal based on EBU R 128 and ITU-R BS.1770-4 standards.

Ports

Input

`Port_1` — Input signal
matrix | 1-D vector

- Matrix input –– Each column of the input is treated as an independent channel. If you use the default **Channel weights**, specify the input channels in order: [Left, Right, Center, Left surround, Right surround].
- 1-D vector input –– The input is treated as a single channel.

Data Types: `single` | `double`

Output

`M` — Momentary loudness measurement
column vector

The block outputs a column vector with the same data type and number of rows as the input signal.

Data Types: `single` | `double`

S — Short-term loudness measurement
column vector

The block outputs a column vector with the same data type and number of rows as the input signal.

Data Types: `single` | `double`

TP — True-peak value
real scalar

The block outputs a real scalar with the same data type as the input signal.

Dependencies

To enable this port, select the **Output true-peak value** parameter.

Data Types: `single` | `double`

Parameters

If a parameter is listed as tunable, then you can change its value during simulation.

`Channel weights` — Linear weighting applied to each input channel
`[1, 1, 1, 1.41, 1.41]` (default) | nonnegative row vector

The number of elements of the row vector must be equal to or greater than the number of input channels. Excess values in the vector are ignored.

The default channel weights follow the ITU-R BS.1170-4 standard. To use the default channel weights, specify the input to the Loudness Meter block as a matrix whose columns correspond to channels in this order: [Left, Right, Center, Left surround, Right surround].

It is a best practice to specify the channel weights in order: [Left, Right, Center, Left surround, Right surround].

Tunable: Yes

`Use relative scale for loudness measurements` — Specify block to output loudness measurements relative to target level
off (default) | on

- On — The loudness measurements are relative to the value specified by **Target loudness level (LUFS)**. The output of the block is returned in loudness units (LU).
- Off — The loudness measurements are absolute, and returned in loudness units full scale (LUFS).

Tunable: No

`Target loudness level (LUFS)` — Reference level for relative loudness measurements

–23 (default) | real scalar

For example, if the **Target loudness level (LUFS)** is –23, then a loudness value of –24 LUFS is reported as –1 LU.

Tunable: Yes

Dependencies

To enable this parameter, select the **Use relative scale for loudness measurements** parameter.

`Output true-peak value` — Add output port for true-peak value

off (default) | on

When you select this parameter, an additional output port, **TP**, is added to the block. The **TP** port outputs the true-peak value of the input frame.

Tunable: No

`Inherit sample rate from input` — Specify source of input sample rate

on (default) | off

When you select this parameter, the block inherits its sample rate from the input signal. When you clear this parameter, you specify the sample rate in **Input sample rate (Hz)**.

Tunable: No

`Input sample rate (Hz)` — Sample rate of input

`44100` (default) | scalar

Tunable: Yes

Dependencies

To enable this parameter, clear the **Inherit sample rate from input** parameter.

`Simulate using` — Specify type of simulation to run

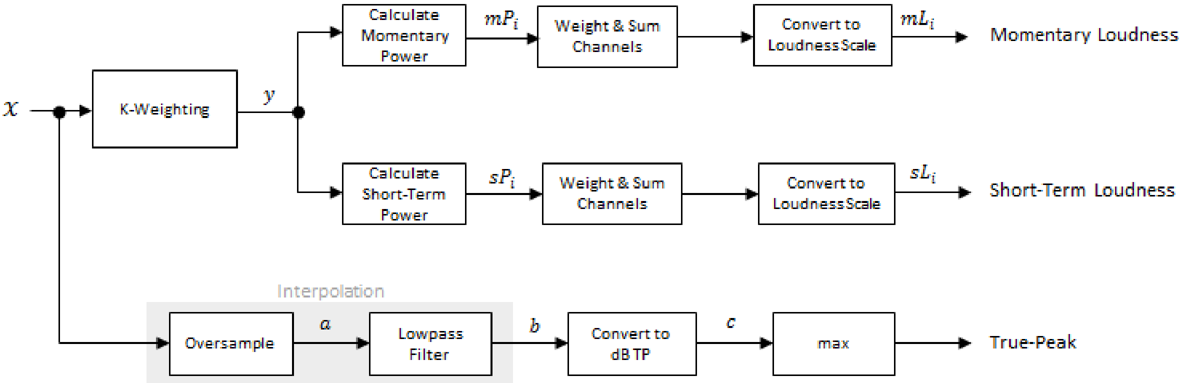
`Code generation` (default) | `Interpreted execution`

- `Code generation` -- Simulate model using generated C code. The first time you run a simulation, Simulink generates C code for the block. The C code is reused for subsequent simulations, as long as the model does not change. This option requires additional startup time but the speed of the subsequent simulations is comparable to `Interpreted execution`.
- `Interpreted execution` -- Simulate model using the MATLAB interpreter. This option shortens startup time but has a slower simulation speed than `Code generation`. In this mode, you can debug the source code of the block.

Tunable: No

Algorithms

The Loudness Meter block calculates the momentary loudness, short-term loudness, and true-peak value of an audio signal. You can specify any number of channels and nondefault channel weights used for loudness measurements. The block algorithm is described for the general case of *n* channels and default channel weights.



Loudness Measurements

The input channels, x, pass through a K-weighted filter implemented using the algorithm of the Weighting Filter block. The K-weighted filter shapes the frequency spectrum to reflect perceived loudness.

1. The K-weighted channels, y, are divided into 0.4-second segments with 0.3-second overlap. If the required number of samples have not been collected yet, the Loudness Meter block returns the last computed value for momentary loudness. If enough samples have been collected, then the power (mean square) of each segment of the K-weighted channels is calculated:

$$mP_i = \frac{1}{w}\sum_{k=1}^{w} y_i^2[k]$$

   - mP_i is the momentary power of the ith segment.
   - w is the segment length in samples.

2. The momentary loudness, mL, is computed for each segment:

$$mL_i = -0.691 + 10\log_{10}\left(\sum_{c=1}^{n} G_c \times mP_{(i,c)}\right) \quad LUFS$$

   - G_c is the weighting for channel c.

   mL is the momentary loudness returned by your Loudness Meter block.

1. The K-weighted channels, y, are divided into 3-second segments with 2.9-second overlap. If the required number of samples have not been collected yet, the Loudness Meter block returns the last computed values for short-term loudness and loudness range. If enough samples have been collected, then the power (mean square) of each K-weighted channel is calculated:

$$sP_i = \frac{1}{w}\sum_{k=1}^{w} y_i^2[k]$$

- sP_i is the short-term power of the ith segment of a channel.
- w is the segment length in samples.

2 The short-term loudness, sL, is computed for each segment:

$$sL_i = -0.691 + 10\log_{10}\left(\sum_{c=1}^{n} G_c \times sP_{(i,c)}\right) \quad LUFS$$

- G_c is the weighting for channel c.

sL is the short-term loudness returned by your Loudness Meter block.

True-Peak

The *true-peak* measurement considers only the current input frame of a call to your loudness meter.

1 The signal is oversampled to at least 192 kHz. To optimize processing, the input sample rate determines the exact oversampling. An input sample rate below 750 Hz is not considered.

Input Sample Rate (kHz)	Upsample Factor
[0.75,1.5)	256
[1.5,3)	128
[3,6)	64
[6,12)	32
[12,24)	16
[24,48)	8
[48,96)	4
[96,192)	2
[192,∞)	not required

2 The oversampled signal, a, passes through a lowpass filter with a half-polyphase length of 12 and stopband attenuation of 80 dB. The filter design uses `designMultirateFIR`.

3 The filtered signal, b, is rectified and converted to the dB TP scale:

$$c = 20 \times \log_{10}\left(|b|\right)$$

4 The true-peak is determined as the maximum of the converted signal, c.

References

[1] International Telecommunication Union; Radiocommunication Sector. *Algorithms to Measure Audio Programme Loudness and True-Peak Audio Level*. ITU-R BS. 1770-4. 2015.

[2] European Broadcasting Union. *Loudness Normalisation and Permitted Maximum Level of Audio Signals*. EBU R 128. 2014.

[3] European Broadcasting Union. *Loudness Metering: 'EBU Mode' Metering to Supplement EBU R 128 Loudness Normalization*. EBU R 128 Tech 3341. 2014.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

Functions
`integratedLoudness`

System Objects
`loudnessMeter`

Introduced in R2016b

MIDI Controls

Output values from controls on MIDI control surface
Library: Audio System Toolbox / Sources

MIDI Controls

Description

The MIDI Controls block outputs values from controls on a MIDI control surface in real time. Use the MIDI Controls block to interact with your audio processing model.

The MIDI Controls block combines the functionality of the general MIDI functions in MATLAB: `midicontrols`, `midiread`, `midisync`. Use the MATLAB `midiid` command to discover MIDI device names or MIDI device control numbers.

Ports

Output

`Port_1` — Output signal
matrix

The output size of the MIDI Controls block is determined by the **MIDI controls** and **MIDI control numbers** parameters.

The output data type is determined by the **Output mode** parameter.

Data Type	Output Mode
`double`	`Normalized (0-1)`
`uint8`	`RAW MIDI (0-127)`

Data Types: `double` | `uint8`

Parameters

`MIDI device` — MIDI control surface your block listens to
`Default` (default) | `Specify other`

To set the default MIDI device, use the `setpref` function. For example, if the device is named `BCF2000`, at the MATLAB command line, enter:

```
setpref('midi','DefaultDevice','BCF2000');
```

`MIDI device name` — Device name of MIDI control surface your block listens to
character vector

The MIDI device name is assigned by the device manufacturer or host operating system, and specified as a character vector. Use `midiid` to interactively identify your MIDI device.

To enable this parameter, set **MIDI device** to `Specify other`.

`MIDI controls` — Specify if block responds to all controllers or specific controllers on MIDI surface
`Respond to any control` (default) | `Respond to specified controls`

This parameter also determines the size of the block output port. If you choose `Respond to any control`, then the block output is a scalar corresponding to the value of the most recently manipulated control.

`MIDI control numbers` — Control numbers associated with MIDI surface controllers that your block responds to
`0` (default) | integer | array of integers

Use `midiid` to interactively identify the control numbers of your MIDI device. This parameter is available when you set **MIDI controls** to `Respond to specified controls`.

`Initial values` — Control numbers associated with MIDI surface controllers that your block responds to
`0` (default) | scalar | array

If you specify **Initial values** as a scalar, all controls specified by **MIDI control numbers** are assigned that value.

If you specify **Initial values** as an array, the array must be the same size as **MIDI control numbers**.

`Send initial values to device at start` — Synchronize MIDI surface with values specified initial values
`off` (default) | `on`

Select this parameter to synchronize a MIDI device with values specified by the **Initial values** when simulation starts. If your MIDI device can receive and respond to messages, it adjusts its controls as specified. This parameter is valid only when **MIDI controls** is set to `Respond to specified controls`.

Many MIDI devices are not bidirectional. Selecting this parameter with a unidirectional device has no effect. The MIDI Controls block cannot tell whether a value is successfully sent to a device or even whether the device is bidirectional. If sending a value fails, no errors or warnings are generated.

`Output Mode` — Output mode for MIDI control value
`Normalized (0-1)` (default) | `RAW MIDI (0-127)`

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

Functions

`midicontrols` | `midiid` | `midiread` | `midisync`

Topics

“MIDI Control Surface Interface”

Noise Gate

Dynamic range gate
Library: Audio System Toolbox / Dynamic Range Control

Description

The Noise Gate block performs dynamic range gating independently across each input channel. Dynamic range gating suppresses signals below a given threshold. The block uses specified attack, release, and hold times to achieve a smooth applied gain curve.

Ports

Input

x — Input signal
matrix | 1-D vector

- Matrix input –– Each column of the input is treated as an independent channel.
- 1-D vector input –– The input is treated as a single channel.

This port is unnamed unless you specify additional input ports.

Data Types: `single` | `double`

T — Threshold (dB)
scalar

Dependencies

To enable this port, select **Specify from input port** for the “Threshold (dB)” on page 5-0 parameter.

Data Types: `single` | `double`

AT — Attack time (s)
scalar

Dependencies

To enable this port, select **Specify from input port** for the “Attack time (s)” on page 5-0 parameter.

Data Types: `single` | `double`

RT — Release time (s)
scalar

Dependencies

To enable this port, select **Specify from input port** for the “Release time (s)” on page 5-0 parameter.

Data Types: `single` | `double`

HT — Hold time (s)
scalar

Dependencies

To enable this port, select **Specify from input port** for the “Hold time (s)” on page 5-0 parameter.

Data Types: `single` | `double`

Output

Y — Output signal
matrix

The Noise Gate block outputs a signal with the same data type as the input signal. The size of the output depends on the size of the input:

- Matrix input –– The block outputs a matrix the same size and data type as the input signal.
- 1-D vector input –– The block outputs an *N*-by-1 matrix (column vector), where *N* is the number of elements in the 1-D vector.

This port is unnamed until you select the **Output gain (dB)** parameter.

Data Types: `single` | `double`

`G` — Gain applied to each input sample
matrix

Dependencies

To enable this port, select the **Output gain (dB)** parameter.

Data Types: `single` | `double`

Parameters

If a parameter is listed as tunable, then you can change its value during simulation.

Main Tab

`Threshold (dB)` — Operation threshold
`–10` (default) | scalar in the range –140 to 0 inclusive

Operation threshold is the level below which gain is applied to the input signal.

To specify **Threshold (dB)** from an input port, select **Specify from input port** for the parameter.

Tunable: Yes

`View static characteristic` — Open static characteristic plot of dynamic range gate
button

The plot is updated automatically when parameters of the Noise Gate block change.

Tunable: Yes

`Attack time (s)` — Time for applied gain to ramp up
`0.05` (default) | scalar in the range 0 to 4 inclusive

Attack time is the time the applied gain takes to rise from 10% to 90% of its final value when the input goes below the threshold. The **Attack time (s)** parameter smooths the applied gain curve.

To specify **Attack time (s)** from an input port, select **Specify from input port** for the parameter.

Tunable: Yes

`Release time (s)` — Time for applied gain to ramp down

`0.2` (default) | scalar in the range 0 to 4 inclusive

Release time is the time the applied gain takes to drop from 90% to 10% of its final value when the input goes above the threshold. The **Release time (s)** parameter smooths the applied gain curve.

To specify **Release time (s)** from an input port, select **Specify from input port** for the parameter.

Tunable: Yes

`Hold time (s)` — Time during which applied gain holds steady

`0.05` (default) | scalar in the range 0 to 4

Hold time is the period in which the applied gain is held constant before it starts moving toward its steady-state value. Hold time begins when the input level crosses the operation threshold.

To specify **Hold time (s)** from an input port, select **Specify from input port** for the parameter.

Tunable: Yes

`Inherit sample rate from input` — Specify source of input sample rate

on (default) | off

When you select this parameter, the block inherits its sample rate from the input signal. When you clear this parameter, you specify the sample rate in **Input sample rate (Hz)**.

Tunable: No

`Input sample rate (Hz)` — Specify input sample rate

`44100` (default) | scalar

Tunable: Yes

Dependencies

To enable this parameter, clear the **Inherit sample rate from input** parameter.

Advanced Tab

`Output gain (dB)` — Gain applied on each input sample

off (default) | on

When you select this parameter, an additional output port, **G**, is added to the block. The **G** port outputs the gain applied on each input channel in dB.

Tunable: No

`Simulate using` — Specify type of simulation to run

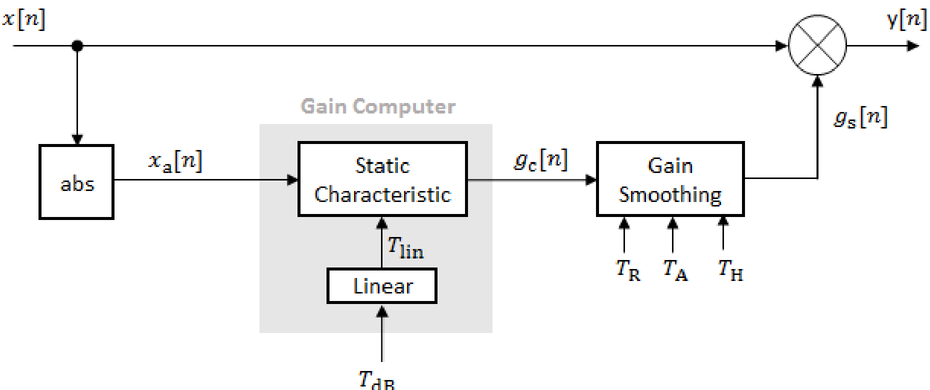
`Interpreted execution` (default) | `Code generation`

- `Interpreted execution` –– Simulate model using the MATLAB interpreter. This option shortens startup time and has a simulation speed comparable to `Code generation`. In this mode, you can debug the source code of the block.
- `Code generation` –– Simulate model using generated C code. The first time you run a simulation, Simulink generates C code for the block. The C code is reused for subsequent simulations, as long as the model does not change. This option requires additional startup time, but the speed of the subsequent simulations is comparable to `Interpreted execution`.

Tunable: No

Algorithms

The Noise Gate block processes a signal frame by frame and element by element.



1. The N-point signal, $x[n]$, is converted to magnitude:

$$x_a[n] = |x[n]|$$

2. $x_a[n]$ passes through the gain computer. The gain computer uses the static characteristic properties of the dynamic range gate to apply a brickwall gain for signal below the threshold:

$$g_c(x_a) = \begin{cases} 0 & x_a < T_{lin} \\ 1 & x_a \geq T_{lin} \end{cases}$$

T_{lin} is the threshold property converted to a linear domain:

$$T_{lin} = 10^{\left(T_{dB}/20\right)}.$$

3. The computed gain, $g_c[n]$, is smoothed using specified attack, release, and hold time parameters:

$$g_s[n] = \begin{cases} \alpha_A g_s[n-1] + (1-\alpha_A) g_c[n] & \text{if } (C_A > T_H) \ \& \ (g_c[n] > g_s[n-1]) \\ g_s[n-1] & \text{if } C_A \leq T_H \\ \alpha_R g_s[n-1] + (1-\alpha_R) g_c[n] & \text{if } (C_R > T_H) \ \& \ (g_c[n] \leq g_s[n-1]) \\ g_s[n-1] & \text{if } C_R \leq T_H \end{cases}$$

C_A and C_R are hold counters for attack and release, respectively. The limit, T_H, is determined by the **Hold time (s)** parameter.

The attack time coefficient, α_A, is calculated as

$$\alpha_A = \exp\left(\frac{-\log(9)}{Fs \times T_A}\right).$$

The release time coefficient, α_R, is calculated as

$$\alpha_R = \exp\left(\frac{-\log(9)}{Fs \times T_R}\right).$$

T_A is the attack time period, specified by the **Attack time (s)** parameter. T_R is the release time period, specified by the **Release time (s)** parameter. *Fs* is the input sampling rate, specified by the **Inherit sample rate from input** or **Input sample rate (Hz)** parameter.

4 The output of the dynamic range gate is given as

$$y[n] = x[n] \times g_s[n].$$

References

[1] Giannoulis, Dimitrios, Michael Massberg, and Joshua D. Reiss. "Digital Dynamic Range Compressor Design –– A Tutorial And Analysis." *Journal of Audio Engineering Society*. Vol. 60, Issue 6, 2012, pp. 399–408.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

Blocks
Compressor | Expander | Limiter

System Objects
`noiseGate`

Topics
"Dynamic Range Control"

Introduced in R2016a

Octave Filter

Octave-band and fractional octave-band filter
Library: Audio System Toolbox / Filters

Octave

Description

The Octave Filter block performs octave-band or fractional octave-band filtering independently across each input channel. An octave-band is a frequency band where the highest frequency is twice the lowest frequency. Octave-band and fractional octave-band filters are commonly used to mimic how humans perceive loudness. Octave filters are best understood when viewed on a logarithmic scale, which models how the human ear weights the spectrum.

Ports

Input

x — Input signal
matrix | 1-D vector

- Matrix input –– Each column of the input is treated as an independent channel.
- 1-D vector input –– The input is treated as a single channel.

This port is unnamed unless you specify additional input ports.

Data Types: `single` | `double`

CF — Center frequency (Hz)
scalar in the range `3` to `22,000` inclusive

Dependencies

To enable this port, select **Specify from input port** for the “Center frequency (Hz)” on page 5-0 parameter.

Data Types: `single` | `double`

Output

`Port_1` — Output signal
matrix

The Octave Filter block outputs a signal with the same data type as the input signal. The size of the output depends on the size of the input:

- Matrix input –– The block outputs a matrix the same size and data type as the input signal.
- 1-D vector input –– The block outputs an *N*-by-1 matrix (column vector), where *N* is the number of elements in the 1-D vector.

Data Types: `single` | `double`

Parameters

If a parameter is listed as tunable, then you can change its value during simulation.

`Filter order` — Order of octave filter
`6` (default) | even integer

Tunable: No

`Center frequency (Hz)` — Center frequency of octave filter
`1000` (default) | scalar in the range `3` to `22,000` inclusive

- The maximum center frequency is the value that causes the upper band edge to be equal to the Nyquist frequency, *Fs*/2. Frequencies above this value are saturated.
- The minimum center frequency is the value that causes the lower band edge to be equal to 1 Hz. Frequencies below this value are quantized to 1 Hz.

To specify **Center frequency (Hz)** from an input port, select **Specify from input port** for the parameter.

Tunable: Yes

`Bandwidth` — Filter bandwidth in octaves
`1 octave` (default) | `2/3 octave` | `1/2 octave` | `1/3 octave` | `1/6 octave` | `1/12 octave` | `1/24 octave` | `1/48 octave`

Tunable: Yes

`Oversample the input by 2 for this filter` — Oversample toggle
off (default) | on

- off –– The Octave Filter block runs at the input sample rate.
- on –– The Octave Filter block runs at two times the input sample rate. Oversampling minimizes the frequency warping effects introduced by the bilinear transformation. An FIR halfband interpolator implements oversampling before octave filtering. A halfband decimator reduces the sample rate back the input sampling rate after octave filtering.

Tunable: No

`Inherit sample rate from input` — Specify source of input sample rate
off (default) | on

When you select this parameter, the block inherits its sample rate from the input signal. When you clear this parameter, you specify the sample rate in **Input sample rate (Hz)**.

Tunable: No

`Input sample rate (Hz)` — Sample rate of input
`44100` (default) | scalar

Tunable: Yes

Dependencies

To enable this parameter, clear the **Inherit sample rate from input** parameter.

`Simulate using` — Specify type of simulation to run
`Code generation` (default) | `Interpreted execution`

- `Code generation` –– Simulate the model using generated C code. The first time you run a simulation, Simulink generates C code for the block. The C code is reused for subsequent simulations as long as the model does not change. This option requires additional startup time, but the speed of the subsequent simulations is comparable to `Interpreted execution`.

- `Interpreted execution` -- Simulate the model using the MATLAB interpreter. This option shortens startup time but has a slower simulation speed than `Code generation`. In this mode, you can debug the source code of the block.

Tunable: No

`Mask for attenuation limits` — Create a mask for filter response visualization
`No mask` (default) | `Class 0` | `Class 1` | `Class 2`

The mask attenuation limits are defined in the ANSI S1.11-2004 standard.

- If the mask is green, the design is compliant.
- If the mask is red, the design breaks compliance.

Tunable: Yes

`Visualize filter response` — Open plot to visualize magnitude response and compliance mask
button

A 2048-point FFT is used to calculate the magnitude response.

Tunable: Yes

Definitions

Band Edge

A band edge frequency refers to the lower or upper edge of the passband of a bandpass filter.

Center Frequency of Octave Filter

The center frequency of an octave filter is the geometric mean of the lower- and upper-band edge frequencies.

Algorithms

Octave Bandwidth to Band Edge Conversion

The Octave Filter block uses the specified center frequency and filter bandwidth in octaves to determine the normalized band edges [2].

First the block computes the upper and lower band edge frequencies:

$$f_{pa} = f_c \times G^{-1/2b}$$

$$f_{pb} = f_c \times G^{1/2b}$$

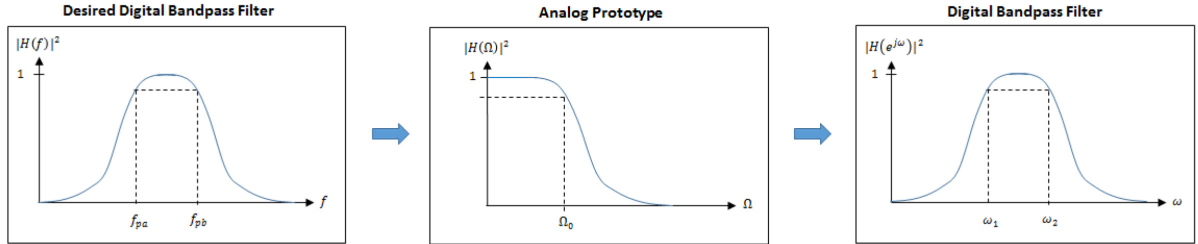
- f_c is the normalized center frequency specified by the **Center frequency (Hz)** parameter.
- b is the octave bandwidth specified by the **Bandwidth** parameter. For example, if **Bandwidth** is specified as `1/3 octave`, the value of b is 3.
- G is a conversion constant:

$$G = 10^{3/10}.$$

Digital Filter Design

The Octave Filter block implements a higher-order digital bandpass filter design method as specified in [1].

In this design method, a desired digital bandpass filter maps to a Butterworth lowpass analog prototype, which is then mapped back to a digital bandpass filter:



1. The analog Butterworth filter is expressed as a cascade of second-order sections:

   $H(s) = H_1(s)H_2(s)\cdots H_{2N}(s)$, where:

   - $$H_i(s) = \frac{1}{1 - 2\frac{s}{\Omega_0}\cos\theta_i + \frac{s^2}{\Omega_0^2}}, \quad i = 1,2,\ldots,2N$$
   - $$\theta_i = \frac{\pi}{2N}(N - 1 + 2i), \quad i = 1,2,\ldots,N,\ldots,2N$$

   N is the filter order specified by the **Filter order** parameter.
2. The analog Butterworth filter is mapped to a digital filter using a bandpass version of the bilinear transformation:

   $$s = \frac{1 - cz^{-1} + z^{-2}}{1 - z^{-2}},$$

   where

   $$c = \frac{\sin(\omega_{pa} + \omega_{pb})}{\sin\omega_{pa} + \sin\omega_{pb}}.$$

   This mapping results in the following substitution:

$$\Omega_0 = \frac{c - \cos\omega_{\text{pb}}}{\sin\omega_{\text{pb}}}$$

3 The analog prototype is evaluated:

$$H_{\text{i}}(z) = \left. \frac{1}{1 - 2\frac{s}{\Omega_0}\cos\theta_{\text{i}} + \frac{s^2}{\Omega_0^2}} \right|_{s = \frac{1 - 2cz^{-1} + z^{-2}}{1 - z^{-2}}}$$

Because s is second-order in z, the bandpass version of the bilinear transformation is fourth-order in z.

References

[1] Orfanidis, Sophocles J. *Introduction to Signal Processing*. Englewood Cliffs, NJ: Prentice Hall, 2010.

[2] Acoustical Society of America. *American National Standard Specification for Octave-Band and Fractional-Octave-Band Analog and Digital Filters: ANSI S1.11-2004*. Melville, NY: Acoustical Society of America, 2009.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

System Objects

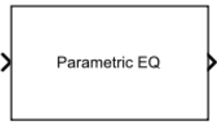
`octaveFilter` | `weightingFilter`

Blocks
Weighting Filter

Introduced in R2016b

Parametric EQ Filter

Second-order parametric equalizer filter
Library: Audio System Toolbox / Filters



Description

The Parametric EQ Filter block filters each channel of the input signal over time using a specified center frequency, bandwidth, and peak (dip) gain. This block offers tunable filter design parameters, which enable you to tune the filter characteristics while the simulation is running. The filter uses a coupled allpass structure to optimize joint computation of the peak and notch response.

This block supports variable-size input, enabling you to change the channel length during simulation. To enable variable-size input, clear the **Inherit sample rate from input** parameter. The number of channels must remain constant.

Ports

Input

`Port_1` — Input signal
matrix | 1-D vector

- Matrix input –– Each column of the input is treated as an independent channel.
- 1-D vector input –– The input is treated as a signal channel.

Data Types: `single` | `double`

Output

`Port_1` — Output signal
matrix

The Parametric EQ Filter block outputs a signal with the same data type as the input signal. The size of the output depends on the size of the input:

- Matrix input –– The block outputs a matrix the same size and data type as the input signal.
- 1-D vector input –– The block outputs an *N*-by-1 matrix (column vector), where *N* is the number of elements in the 1-D vector.

Data Types: `single` | `double`

Parameters

If a parameter is listed as tunable, then you can change its value during simulation.

`Filter specification` — Specify parameters or coefficients used to design filter

`Bandwidth and center frequency` (default) | `Coefficients` | `Quality factor and center frequency`

- `Bandwidth and center frequency` –– Design the filter using **Filter bandwidth (Hz)**, **Equalizer center frequency (Hz)**, and **Gain (dB)**.
- `Coefficients` –– Design the filter using **Bandwidth coefficient**, **Center frequency coefficient**, and **Gain (Linear Units)**.
- `Quality factor and center frequency` –– Design the filter using **Equalizer center frequency (Hz)**, **Gain (dB)**, and **Quality factor**.

Tunable: No

`Filter bandwidth (Hz)` — Bandwidth of the filter

`2205` (default) | positive scalar

Specify the filter bandwidth as a positive scalar that is less than half the sample rate of the input signal.

Tunable: Yes

Dependencies

To enable this parameter, set **Filter specification** to `Bandwidth and center frequency`.

`Equalizer center frequency (Hz)` — Center frequency of the filter
`11025` (default) | positive scalar

Specify the center frequency as a positive scalar that is less than half the sample rate of the input signal.

Tunable: Yes

Dependencies

To enable this parameter, set **Filter specification** to `Bandwidth and center frequency` or `Quality factor and center frequency`.

`Gain (dB)` — Peak or dip gain of the filter
`6.0206` (default) | real scalar

Tunable: Yes

Dependencies

To enable this parameter, set **Filter specification** to `Bandwidth and center frequency` or `Quality factor and center frequency`.

`Bandwidth coefficient` — Coefficient that determines the filter bandwidth
`0.72654` (default) | scalar in the range –`1` to `1`

- `-1` corresponds to the maximum bandwidth (one-fourth the sample rate of the input signal).
- `1` corresponds to the minimum bandwidth (0 Hz, that is, an allpass filter).

Tunable: Yes

Dependencies

To enable this parameter, set **Filter specification** to `Coefficients`.

`Center frequency coefficient` — Coefficient that determines the filter center frequency
`0` (default) | scalar in the range –`1` to `1`

- `-1` corresponds to the minimum center frequency (0 Hz).
- `1` corresponds to the maximum center frequency (half the sample rate of the input signal).

Tunable: Yes

Dependencies

To enable this parameter, set **Filter specification** to `Coefficients`.

`Gain (Linear Units)` — Peak or dip gain of the filter

`2` (default) | positive scalar

A value greater than one boosts the input signal. A value less than one attenuates the input signal.

Tunable: Yes

Dependencies

To enable this parameter, set **Filter specification** to `Coefficients`.

`Quality factor` — Quality factor of the filter

`5` (default) | positive scalar

Tunable: Yes

Dependencies

To enable this parameter, set **Filter specification** to `Quality factor and center frequency`.

`Inherit sample rate from input` — Specify source of input sample rate

on (default) | off

When you select this parameter, the block inherits its sample rate from the input signal. When you clear this parameter, you specify the sample rate in **Input sample rate (Hz)**.

Tunable: No

`Input sample rate (Hz)` — Sample rate of input

`44100` (default) | scalar

Tunable: Yes

Dependencies

To enable this parameter, clear the **Inherit sample rate from input** parameter.

`View Filter Response` — Open plot to visualize magnitude response
button

`Simulate using` — Specify type of simulation to run
`Code generation` (default) | `Interpreted execution`

- `Interpreted execution` -– Simulate model using the MATLAB interpreter. This option shortens startup time and has simulation speed comparable to `Code generation`. In this mode, you can debug the source code of the block.
- `Code generation` -– Simulate model using generated C code. The first time you run a simulation, Simulink generates C code for the block. The C code is reused for subsequent simulations, as long as the model does not change. This option requires additional startup time but the speed of the subsequent simulations is faster compared to `Interpreted execution`.

Tunable: No

References

[1] Orfanidis, Sophocles J. *Introduction to Signal Processing*. Upper Saddle River, NJ: Prentice-Hall, 1996.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

System Objects
`multibandParametricEQ`

Functions
`designParamEQ` | `designShelvingEQ` | `designVarSlopeFilter`

Topics

“Parametric Equalizer Design”
“Equalization”

Reverberator

Add reverberation to audio signal
Library: Audio System Toolbox / Effects

Description

The Reverberator block adds reverberation to mono or stereo audio signals. You can tune parameters of the Reverberator block to mimic different acoustic environments.

Ports

Input

`Port_1` — Input signal
matrix | 1-D vector

- Matrix input –– Each column of the input is treated as an independent channel.
- 1-D vector input –– The input is treated as a single channel.

Data Types: `single` | `double`

Output

`Port_1` — Output signal
matrix

The Reverberator block outputs a signal with the same data type as the input signal. The size of the output depends on the size of the input:

- Matrix input –– The block outputs a matrix the same size and data type as the input signal.

- 1-D vector input –– The block outputs an *N*-by-1 matrix (column vector), where *N* is the number of elements in the 1-D vector.

Data Types: `single` | `double`

Parameters

If a parameter is listed as tunable, then you can change its value during simulation.

`Pre-delay (s)` — Pre-delay for reverberation
`0` (default) | scalar in the range 0 to 1

Pre-delay for reverberation is the time between hearing direct sound and the first early reflection. The value of **Pre-delay (s)** is proportional to the size of the room being modeled.

Tunable: Yes

`Highcut frequency (Hz)` — Lowpass filter cutoff in the range 0 to (Sample Rate)/2
`20000` (default) | real positive scalar

Lowpass filter cutoff is the –3 dB cutoff frequency for the single-pole lowpass filter at the front of the reverberator structure. It prevents the application of reverberation to high-frequency components of the input.

Tunable: Yes

`Diffusion` — Density of reverb tail
`0.50` (default) | scalar in the range 0 to 1

Diffusion is proportional to the rate at which the reverb tail builds in density. Increasing **Diffusion** pushes the reflections closer together, thickening the sound. Reducing **Diffusion** creates more discrete echoes.

Tunable: Yes

`Decay factor` — Decay factor of reverb tail
`0.50` (default) | scalar in the range 0 to 1

Decay factor is proportional to the time it takes for reflections to run out of energy. To model a large room, use a long reverb tail (low decay factor). To model a small room, use a short reverb tail (high decay factor).

Tunable: Yes

`High frequency damping` — High-frequency damping
`0.0005` (default) | scalar in the range 0 to 1

High frequency damping is proportional to the attenuation of high frequencies in the reverberation output. Setting **High frequency damping** to a large value makes high-frequency reflections decay faster than low-frequency reflections.

Tunable: Yes

`Wet/dry mix` — Ratio of wet (reverberated) signal to dry (original) signal
`0.3` (default) | scalar in the range 0 to 1

Wet/dry mix is the ratio of wet (reverberated) signal to dry (original) signal that your Reverberator block outputs.

Tunable: Yes

`Inherit sample rate from input` — Specify source of input sample rate
on (default) | off

When you select this parameter, the block inherits its sample rate from the input signal. When you clear this parameter, you specify the sample rate in **Input sample rate (Hz)**.

Tunable: No

`Input sample rate (Hz)` — Sample rate of input
`44100` (default) | positive scalar

Tunable: Yes

Dependencies

To enable this parameter, clear the **Inherit sample rate from input** parameter.

`Simulate using` — Specify type of simulation to run
`Interpreted execution` (default) | `Code generation`

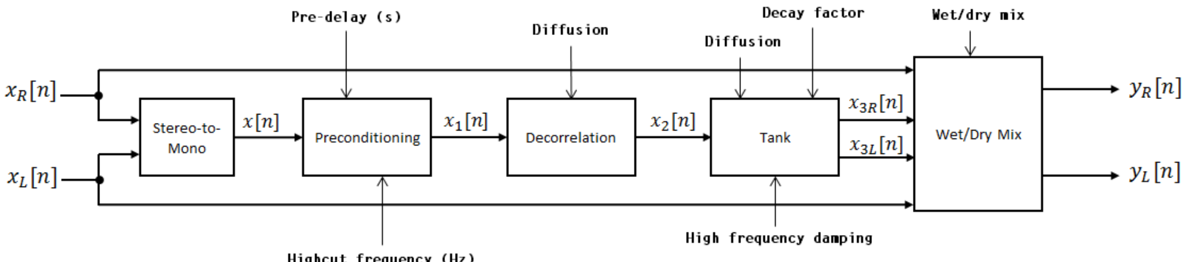
- `Interpreted execution` –– Simulate model using the MATLAB interpreter. This option shortens startup time and has simulation speed comparable to `Code generation`. In this mode, you can debug the source code of the block.
- `Code generation` –– Simulate model using generated C code. The first time you run a simulation, Simulink generates C code for the block. The C code is reused for subsequent simulations, as long as the model does not change. This option requires additional startup time but the speed of the subsequent simulations is comparable to `Interpreted execution`.

Tunable: No

Algorithms

The algorithm to add reverberation follows the plate-class reverberation topology described in [1] and is based on a 29,761 Hz sample rate.

The algorithm has five stages.



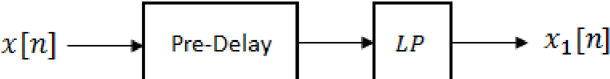
The description for the algorithm that follows is for a stereo input. A mono input is a simplified case.

Stereo-to-Mono

A stereo signal is converted to a mono signal: $x[n] = 0.5 \times (x_R[n] + x_L[n])$.

Preconditioning

A delay followed by a lowpass filter preconditions the mono signal.



- The pre-delay output is determined as $x_p[n] = x[n-k]$, where the **Pre-delay (s)** parameter determines the value of k.
- The signal is fed through a single-pole lowpass filter with transfer function

  $$LP(z) = \frac{1-\alpha}{1-\alpha z^{-1}},$$

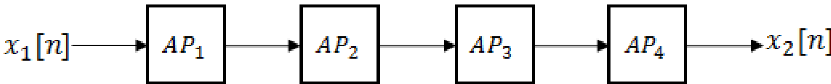
  where

  $$\alpha = \exp\left(-2\pi \times \frac{f_c}{f_s}\right).$$

  - f_c is the cutoff frequency specified by the **Pre-delay (s)** parameter.
  - f_s is the sampling frequency specified by the **Inherit sample rate from input** parameter or the **Input sample rate (Hz)** parameter.

Decorrelation

The signal is decorrelated by passing through a series of four allpass filters.



The allpass filters are of the form

$$AP(z) = \frac{\beta + z^{-k}}{1 + \beta z^{-k}},$$

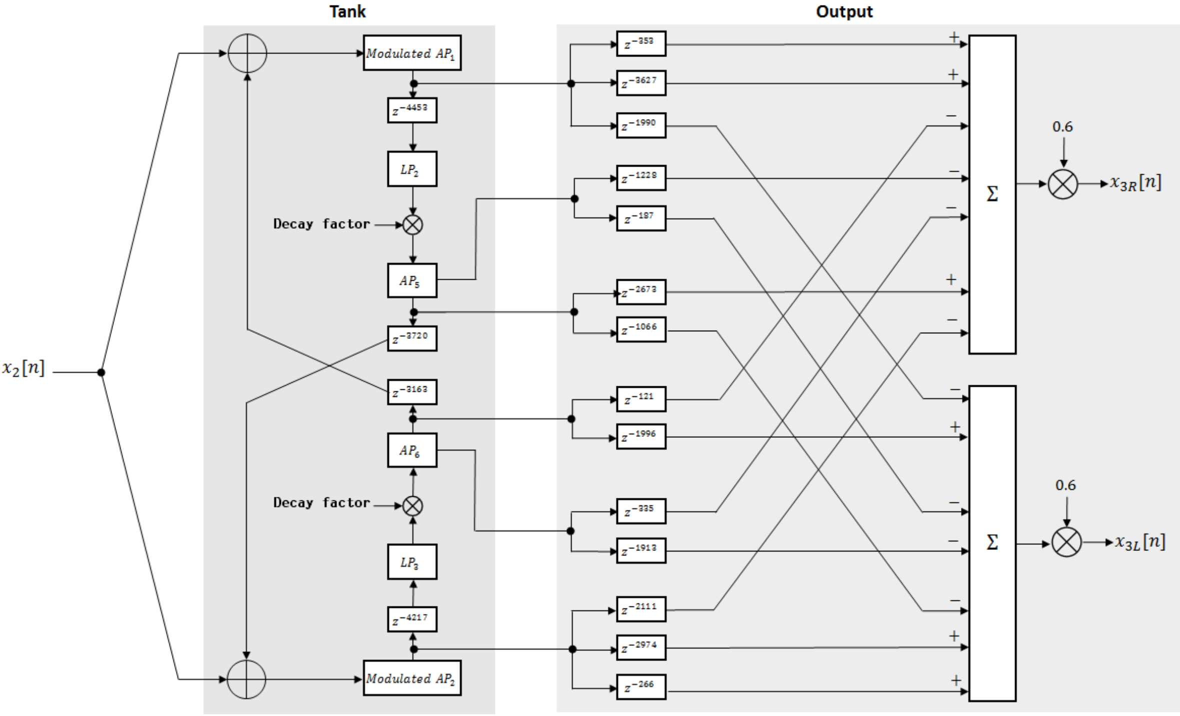
where β is the coefficient specified by the `Diffusion` property and k is the delay as follows:

- For AP_1, k = 142.

- For AP_2, k = 107.
- For AP_3, k = 379.
- For AP_4, k = 277.

Tank

The signal is fed into the tank, where it circulates to simulate the decay of a reverberation tail.



The following description tracks the signal as it progresses through the top of the tank. The signal progression through the bottom of the tank follows the same pattern, with different delay specifications.

1. The new signal enters the top of the tank and is added to the circulated signal from the bottom of the tank.

2 The signal passes through a modulated allpass filter:

$$Modulated\ AP_1(z) = \frac{-\beta + z^{-k}}{1 - \beta z^{-k}}$$

- β is the coefficient specified by the **Diffusion** parameter.
- k is the variable delay specified by a 1 Hz sinusoid with amplitude = (8/29761) × (sample rate). To account for fractional delay resulting from the modulating k, allpass interpolation is used [2].

3 The signal is delayed again, and then passes through a lowpass filter:

$$LP_2(z) = \frac{1 - \varphi}{1 - \varphi z^{-1}}$$

- φ is the coefficient specified by the **High frequency damping** parameter.

4 The signal is multiplied by a gain specified by the **Decay factor** parameter. The signal then passes through an allpass filter:

$$AP_5(z) = \frac{\beta + z^{-k}}{1 + \beta z^{-k}}.$$

- β is the coefficient specified by the **Diffusion** parameter.
- k is set to `1800` for the top of the tank and `2656` for the bottom of the tank.

5 The signal is delayed again and then circulated to the bottom half of the tank for the next iteration.

A similar pattern is executed in parallel for the bottom half of the tank. The output of the tank is calculated as the signed sum of delay lines picked off at various points from the tank. The summed output is multiplied by `0.6`.

Wet/Dry Mix

The wet (processed) signal is then added to the dry (original) signal:

$$y_R[n] = (1 - \kappa) x_R[n] + \kappa x_{3R}[n],$$

$$y_L[n] = (1-\kappa)\, x_L[n] + \kappa x_{3L}[n],$$

where the **Wet/dry mix** parameter determines κ.

References

[1] Dattorro, Jon. "Effect Design, Part 1: Reverberator and Other Filters." *Journal of the Audio Engineering Society*. Vol. 45, Issue 9, 1997, pp. 660–684.

[2] Dattorro, Jon. "Effect Design, Part 2: Delay-Line Modulation and Chorus." *Journal of the Audio Engineering Society*. Vol. 45, Issue 10, 1997, pp. 764–788.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

System Objects
`reverberator`

Introduced in R2016a

Weighting Filter

Weighted frequency response filter
Library: Audio System Toolbox / Filters

Description

The Weighting Filter block performs frequency-weighted filtering independently across each input channel.

Ports

Input

`Port_1` — Input signal
matrix | 1-D vector

- Matrix input –– Each column of the input is treated as an independent channel.
- 1-D vector input –– The input is treated as a single channel.

Data Types: `single` | `double`

Output

`Port_1` — Output signal
matrix

The Weighting Filter block outputs a signal with the same data type as the input signal. The size of the output depends on the size of the input:

- Matrix input –– The block outputs a matrix the same size and data type as the input signal.

- 1-D vector input –– The block outputs an *N*-by-1 matrix (column vector), where *N* is the number of elements in the 1-D vector.

Data Types: `single` | `double`

Parameters

If a parameter is listed as tunable, then you can change its value during simulation.

`Weighting method` — Type of frequency weighting
`A-weighting` (default) | `C-weighting` | `K-weighting`

See "A-Weighting" on page 5-115, "C-Weighting" on page 5-116, and "K-Weighting" on page 5-117 for the definition of the weighting curves.

Tunable: No

`Inherit sample rate from input` — Specify source of input sample rate
`off` (default) | `on`

When you select this parameter, the block inherits its sample rate from the input signal. When you clear this parameter, you specify the sample rate in **Input sample rate (Hz)**.

Tunable: No

`Input sample rate (Hz)` — Sample rate of input
`44100` (default) | `positive scalar`

Tunable: Yes

Dependencies

To enable this parameter, clear the **Inherit sample rate from input** parameter.

`Simulate using` — Specify type of simulation to run
`Code generation` (default) | `Interpreted execution`

- `Code generation` –– Simulate model using generated C code. The first time you run a simulation, Simulink generates C code for the block. The C code is reused for subsequent simulations, as long as the model does not change. This option requires additional startup time but the speed of the subsequent simulations is faster than `Interpreted execution`.

- `Interpreted execution` –– Simulate model using the MATLAB interpreter. This option shortens startup time but has a slower simulation speed compared to `Code generation`. In this mode, you can debug the source code of the block.

Tunable: No

`Mask for attenuation limits` — Creates a mask for filter response visualization

`No mask` (default) | `Class 1` | `Class 2`

The mask attenuation limits are defined in the IEC 61672-1:2002 standard.

- If the mask is green, the design is compliant.
- If the mask is red, the design breaks compliance.

Tunable: Yes

Dependencies

To enable this parameter, set **Weighting method** to `A-weighting` or `C-weighting`.

`Visualize filter response` — Open plot to visualize magnitude response and compliance mask

button

A 2048-point FFT is used to calculate the magnitude response.

Tunable: Yes

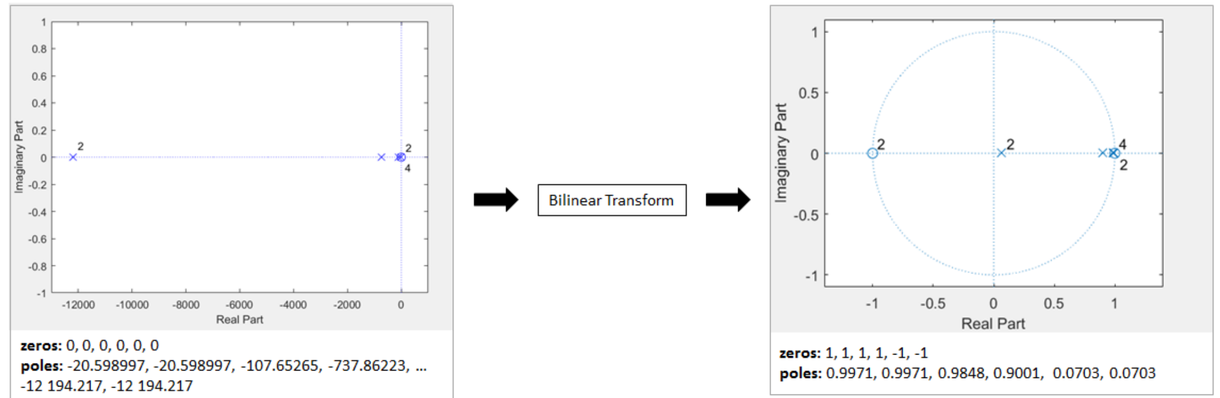
Definitions

A-Weighting

The A-curve is a wide bandpass filter centered at 2.5 kHz, with approximately 20 dB attenuation at 100 Hz and 10 dB attenuation at 20 kHz. A-weighted SPL measurements of noise level are increasingly found in sales literature for domestic appliances. In most countries, the use of A-weighting is mandated for the protection of workers against noise-induced deafness. The ISO and ICOA standards mandate A-weighting for all civil aircraft noise measurements.

The ANSI S1.42.2001 [1] defines this weighting curve. The IEC 61672-1:2002 [2] standard defines the minimum and maximum attenuation limits for an A-weighting filter.

ANSI S1.42.2001 defines the weighting curve by specifying analog poles and zeros. Audio System Toolbox converts the specified poles and zeros to the digital domain using a bilinear transform:

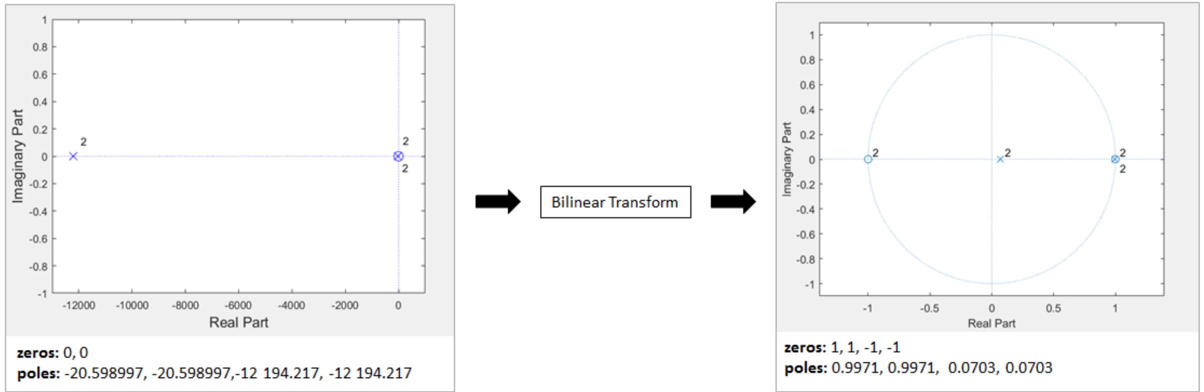


C-Weighting

The C-curve is "flat," but with limited bandwidth: It has –3 dB corners at 31.5 Hz and 8 kHz. C-curves are used in sound level meters for sounds that are louder than sounds intended for A-weighting filters.

The ANSI S1.42-2001 [1] defines the C-weighting curve. The IEC 61672-1:2002 [2] standard defines the minimum and maximum attenuation limits for C-weighting filters.

ANSI S1.42.2001 defines the weighting curve by specifying analog poles and zeros. Audio System Toolbox converts the specified poles and zeros to the digital domain using a bilinear transform:

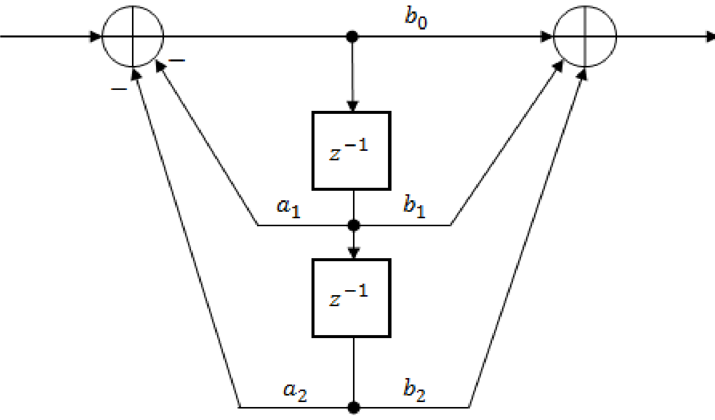


K-Weighting

The K-weighting filter is used for loudness normalization in broadcast. It is composed of two stages of filtering: a first stage shelving filter and a second stage highpass filter.

The ITU-R BS.1770-4 [3] standard defines this curve.

Assume a second-order filter.



The table shows the coefficients for the filters.

First Stage Shelving Coefficients	Second Stage Highpass Coefficients
$a_1 = -1.69065929318241$	$a_1 = -1.99004745483398$
$a_2 = 0.73248077421585$	$a_2 = 0.99007225036621$
$b_0 = 1.53512485958697$	$b_0 = 1.0$
$b_1 = -2.6916918940638$	$b_1 = -2.0$
$b_2 = 1.19839281085285$	$b_2 = 1.0$

The coefficients presented by ITU-R BS.1770-4 are defined for 48 kHz. These coefficients are recomputed for nonstandard sample rates using the algorithm described in [4].

References

[1] Acoustical Society of America. *Design Response of Weighting Networks for Acoustical Measurements*. ANSI S1.42-2001. New York, NY: American National Standards Institute, 2001.

[2] International Electrotechnical Commission. *Electroacoustics Sound Level Meters Part 1: Specifications*. First Edition. IEC 61672-1. 2002-2005.

[3] International Telecommunication Union. *Algorithms to measure audio programme loudness and true-peak audio level*. ITU-R BS.1770-4. 2015.

[4] Mansbridge, Stuart, Saoirse Finn, and Joshua D. Reiss. "Implementation and Evaluation of Autonomous Multi-track Fader Control." Paper presented at the 132nd Audio Engineering Society Convention, Budapest, Hungary, 2012.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

Blocks
Loudness Meter | Octave Filter

System Objects
`loudnessMeter` | `octaveFilter` | `weightingFilter`

Introduced in R2016b