Audio System Toolbox™

Reference

MATLAB®&SIMULINK®

R2017a

MathWorks®

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

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)

Contents

1

Apps in Audio System Toolbox

Audio Test Bench

Develop, 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.
■	Stop	Stop the audio stream loop. The MATLAB command line and objects used by the test bench are released.

Button		Description
	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.
	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.
	Synchronize to MIDI Controls	Start the `configureMIDI` user interface (UI) for your plugin object.
	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™.
	Help	Open MATLAB documentation for **Audio Test Bench**.
	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

Button		Description
⚙	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

See Also

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

Classes
audioPluginSource | audioPlugin

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

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) | string

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) | string

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

`'VendorVersion'` — Vendor version

'1.0.0' (default) | dot-separated string

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-character string

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

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 2-23.

`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

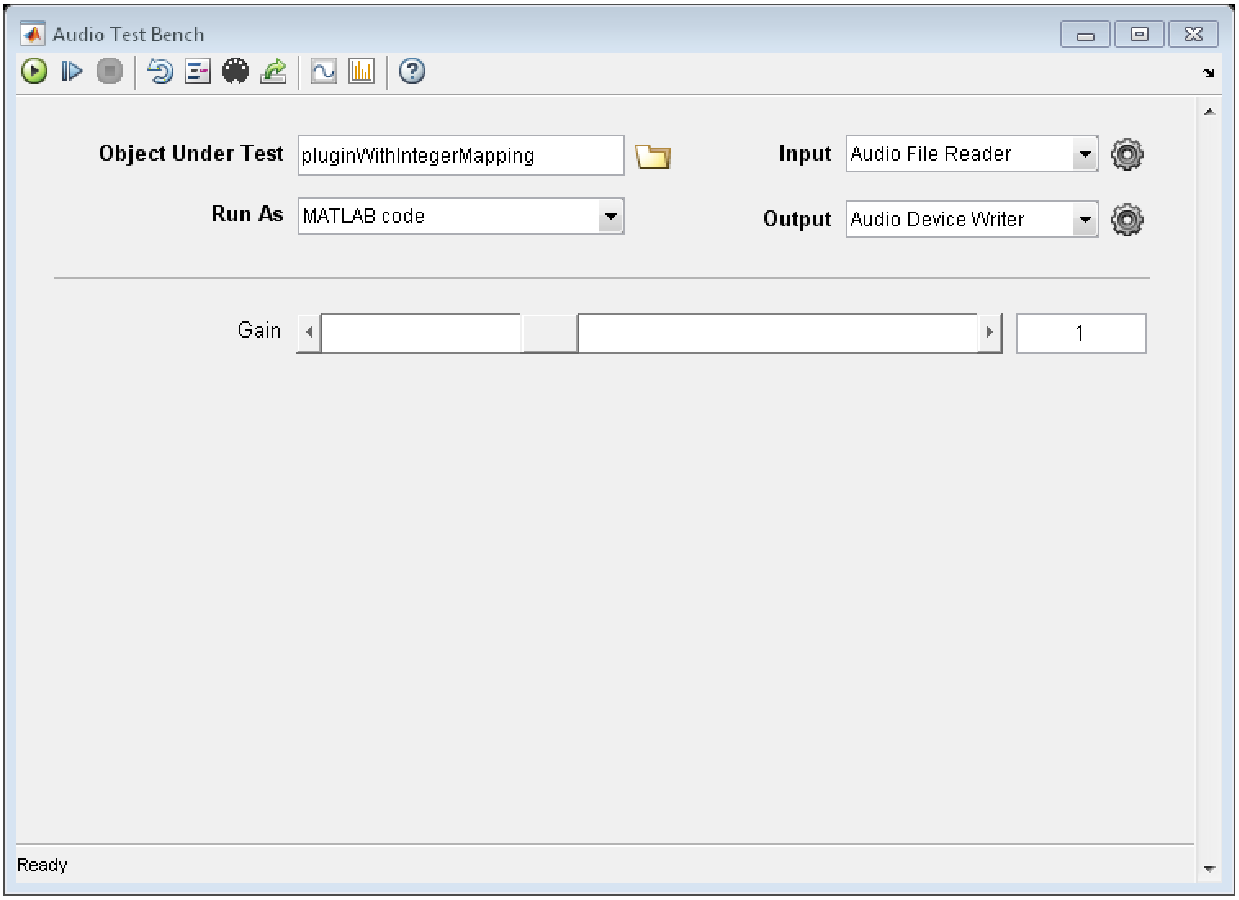
The following class definition uses integer parameter mapping to define the relationship between a property and a parameter. You can use the plugin created from this class to tune the linear gain of an audio signal in integer steps from 0 to 3.

```
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)
```



Object Under Test: pluginWithIntegerMapping
Run As: MATLAB code
Input: Audio File Reader
Output: Audio Device Writer
Gain: 1
Ready

Power Parameter Mapping

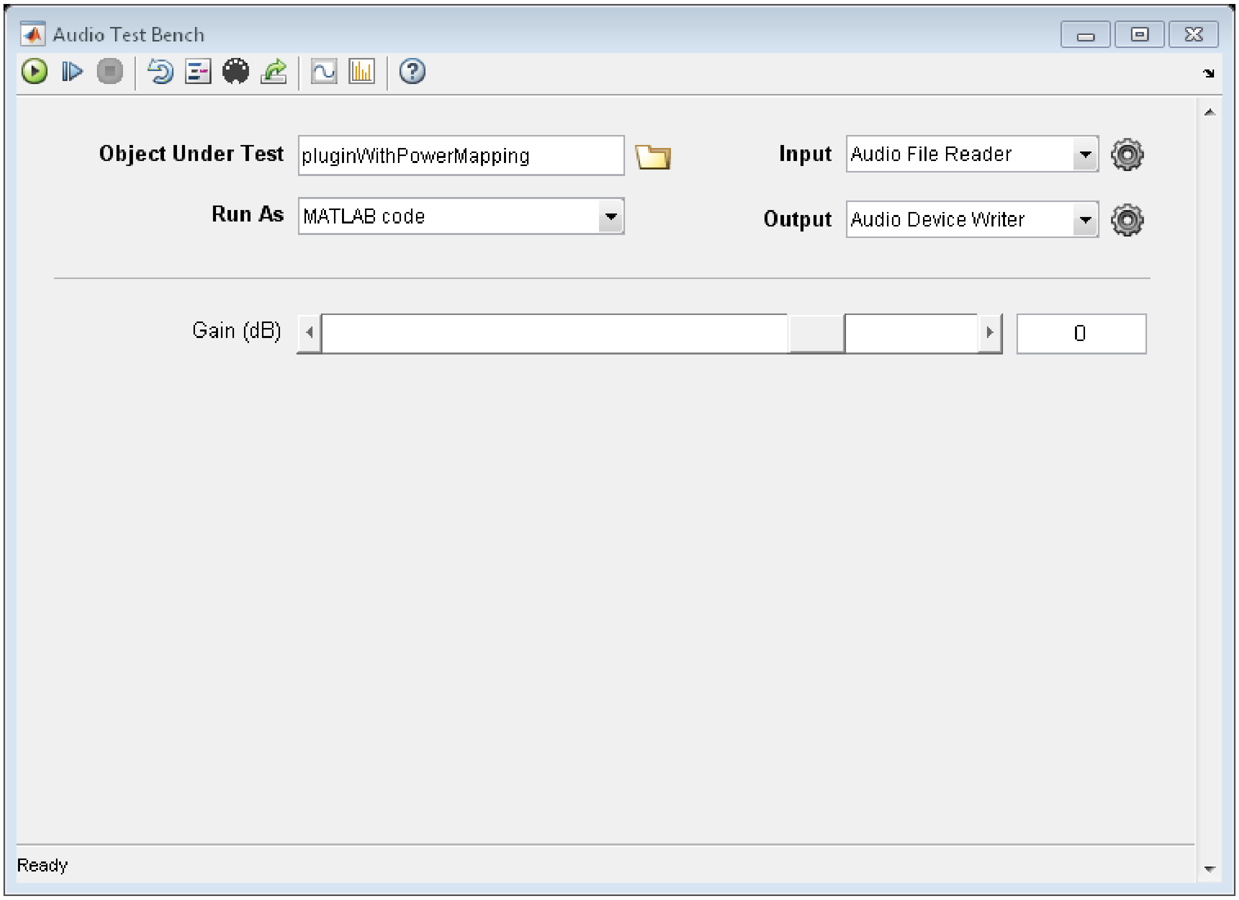
The following class definition uses power parameter mapping to define the relationship between a property and a parameter. You can use the plugin created from this class to tune the gain of an audio signal in dB.

```
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)
```



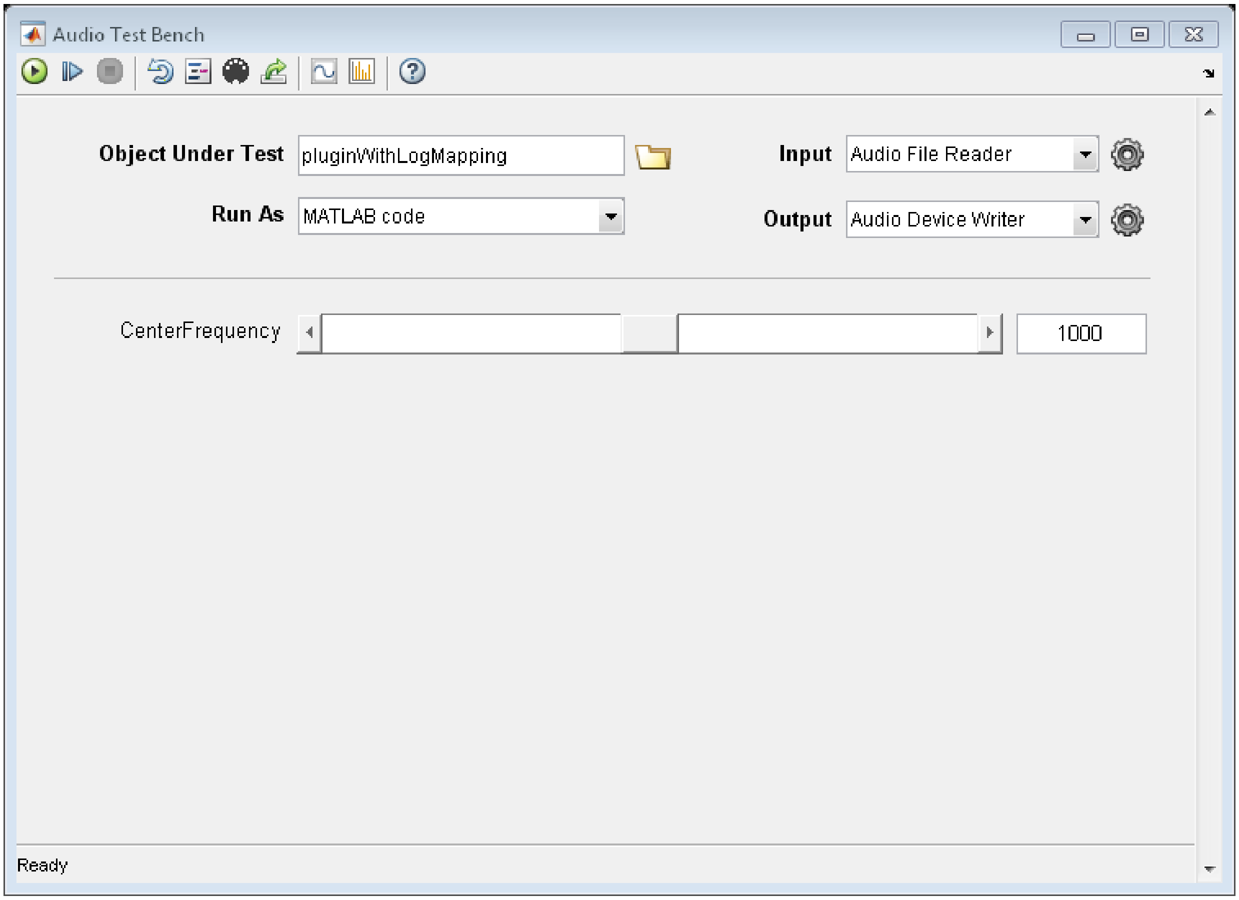
Logarithmic Parameter Mapping

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)
```



Object Under Test: pluginWithLogMapping
Run As: MATLAB code
Input: Audio File Reader
Output: Audio Device Writer
CenterFrequency: 1000

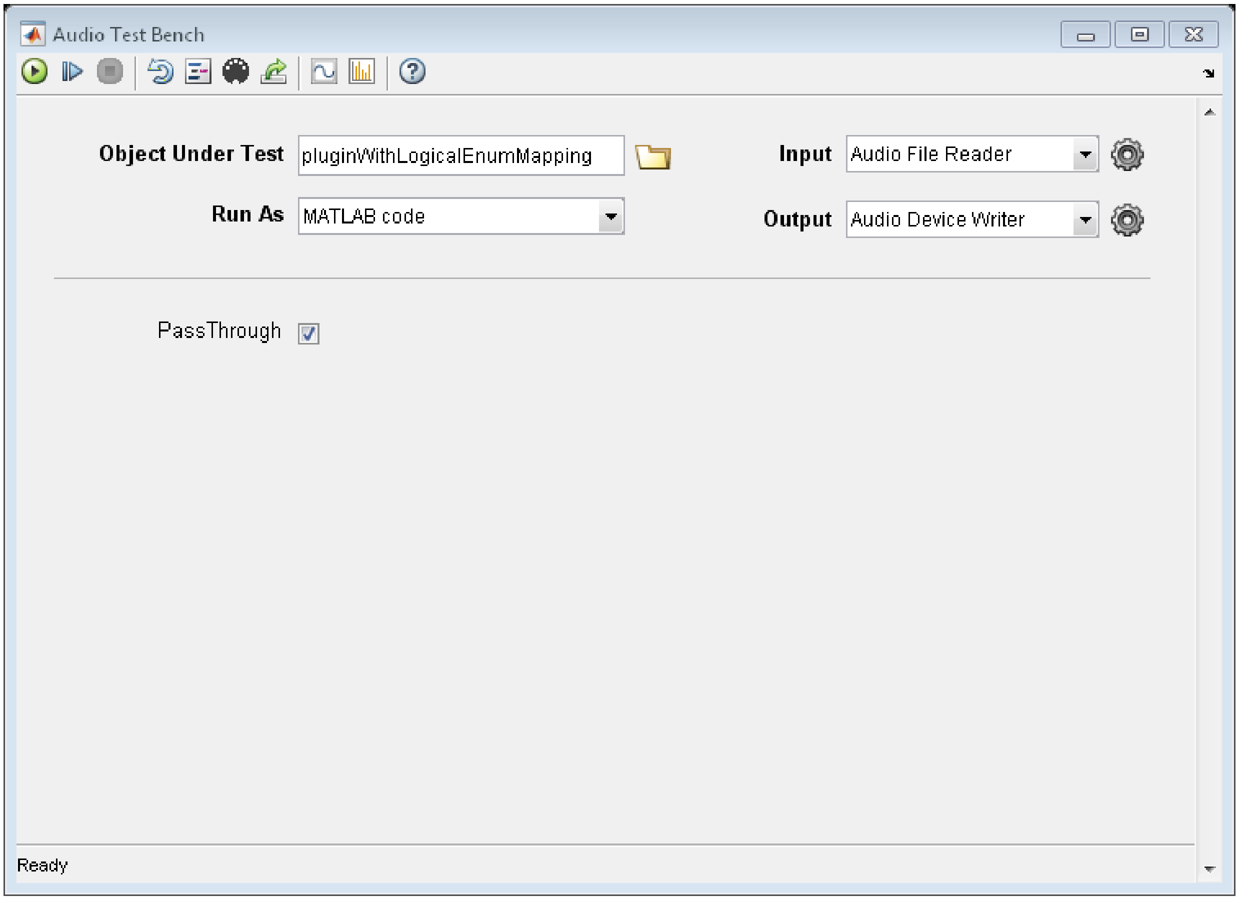
Enumeration for Logical Properties Parameter Mapping

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)
```



Object Under Test: pluginWithLogicalEnumMapping
Run As: MATLAB code
Input: Audio File Reader
Output: Audio Device Writer
PassThrough ✓

'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

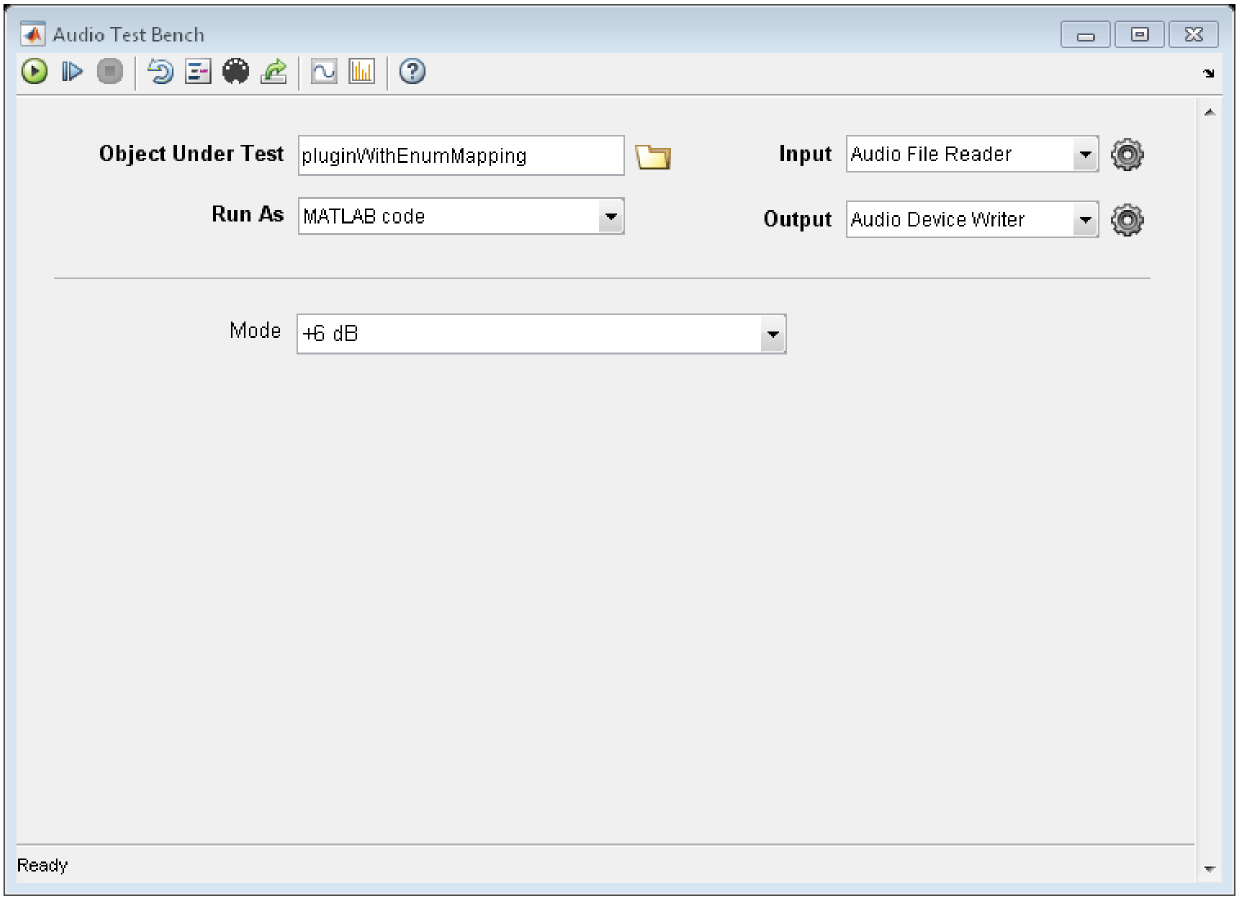
```
classdef pluginWithEnumMapping < audioPlugin
    properties
        Mode = OperatingMode.boost;
    end
    properties (Constant)
        PluginInterface = audioPluginInterface(...
            audioPluginParameter('Mode',...
                'Mapping',{'enum','+6 dB','-6 dB','silence','white noise'}));
    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)
```



Object Under Test: pluginWithEnumMapping

Run As: MATLAB code

Input: Audio File Reader

Output: Audio Device Writer

Mode: +6 dB

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

Mappin	Description	Example
`'lin'`	Specifies a linear relationship with given minimum and maximum values. $(\textit{property value}) = \min + (\max - \min) \times (p$	{`'lin'`, `0`, `24`} specifies a linear relationship with a minimum of 0 and maximum of 24. **Simple Example:** "Specify Parameter Information" on page 2-10
`'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`. $(\textit{property value}) = \min \times (\max / \min)^{(\textit{para}}$	{`'log'`, `1`, `22050`} specifies a logarithmic relationship with a minimum of 1 and a maximum of 22050. **Simple Example:** "Logarithmic Parameter Mapping" on page 2-14
`'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: $(\textit{property value}) = \min + (\max - \min) \times (p$	{`'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 2-12
`'int'`	Quantizes the control position and maps it to the range of consecutive integers with given minimum and maximum values.	{`'int'`, `0`, `3`} specifies a linear, quantized relationship with a minimum of 0 and maximum of 3. The property

Mappin	Description	Example
	$(property\ value) = floor\left(0.5 + \min + (\mathrm{ma}\right.$	value is mapped as an integer in the range 0 to 3. **Simple Example:** “Integer Parameter Mapping” on page 2-10
`'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 2-16
`'enum'` (enume 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 2-18

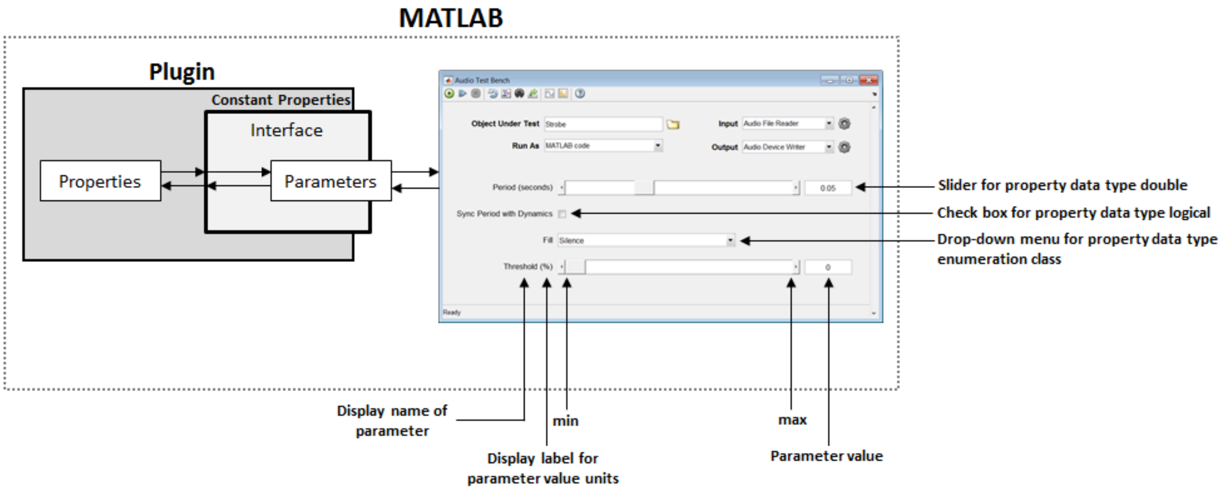
For nontrivial examples of audio plugin parameter mapping, see “Audio Plugin Example Gallery”.

Definitions

Implementation of Audio Plugin Parameters

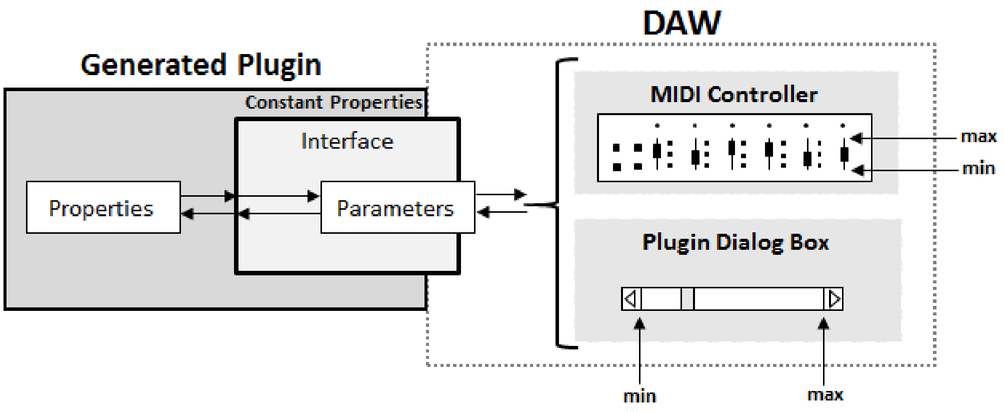
Audio plugin parameters are visible and tunable in both the MATLAB and digital audio workstation (DAW) environments.

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


MATLAB
Plugin
Constant Properties
Interface
Properties
Parameters
Audio Test Bench
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

Object Under Test Strobe
Run As MATLAB code
Input Audio File Reader
Output Audio Device Writer
Period (seconds) 0.05
Sync Period with Dynamics
Fill Silence
Threshold (%) 0
Ready

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

See Also

Classes
audioPluginSource | audioPlugin

Functions
`audioPluginInterface` | `generateAudioPlugin` | `validateAudioPlugin`

Topics
“Design an Audio Plugin”

Introduced in R2016a

configureMIDI

Configure MIDI connections between audio plugin and MIDI controller

Syntax

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

Description

`configureMIDI(myAudioPlugin)` opens a MIDI configuration user interface (UI). Use the UI to synchronize parameters of the plugin, `myAudioPlugin`, 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.

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

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

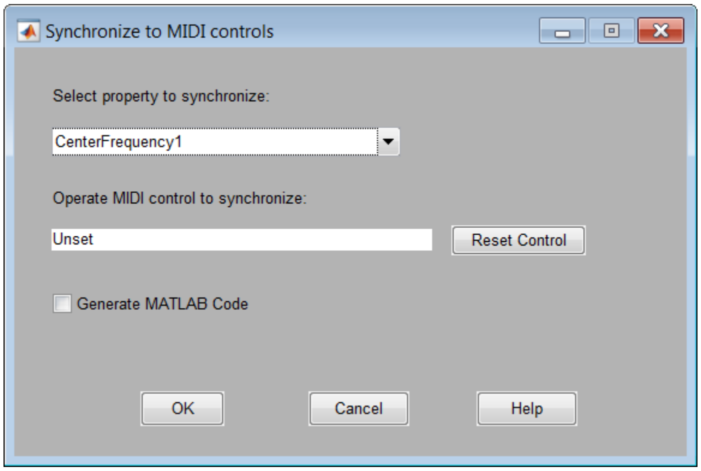
`configureMIDI(myAudioPlugin,propertyName,controlNumber,'DeviceName', deviceNameValue)` makes the plugin property respond to the MIDI control specified by `controlNumber` on the device specified by `deviceNameValue`.

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.

   Synchronize to MIDI controls

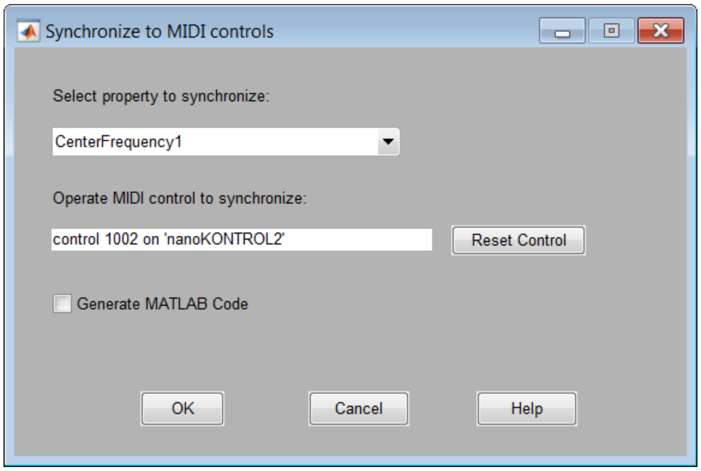
   Select property to synchronize:

   CenterFrequency1

   Operate MIDI control to synchronize:

   Unset

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.


Synchronize to MIDI controls
Select property to synchronize:
CenterFrequency1
Operate MIDI control to synchronize:
control 1002 on 'nanoKONTROL2'
Reset Control
Generate MATLAB Code
OK
Cancel
Help

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.

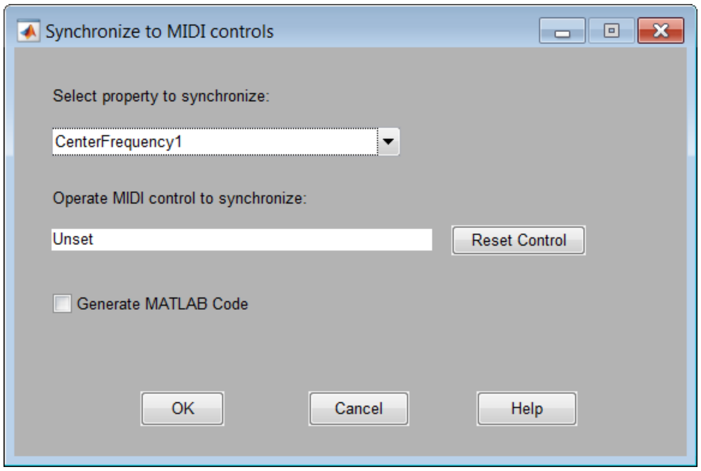
Generate MATLAB Code from `configureMIDI` UI

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.

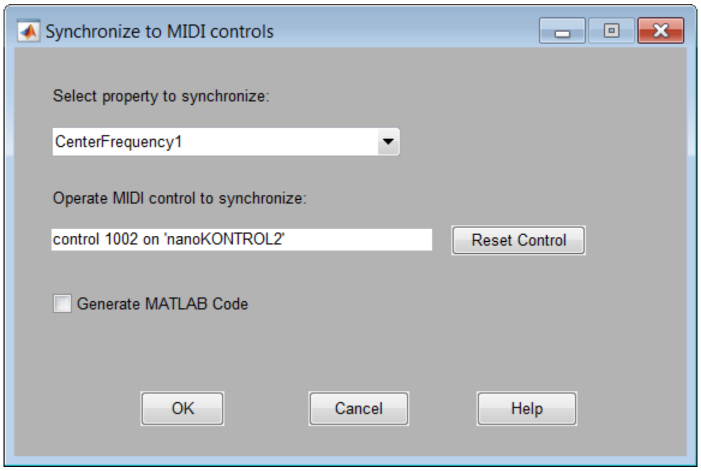


Select property to synchronize: CenterFrequency1

Operate MIDI control to synchronize: Unset

☐ Generate MATLAB Code

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.

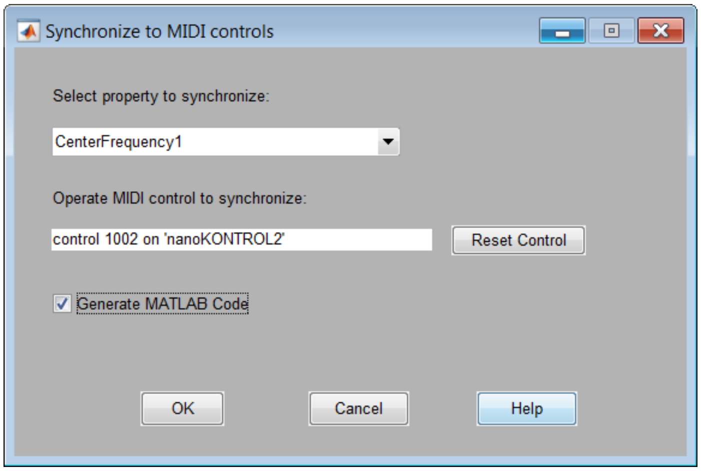


Select property to synchronize: CenterFrequency1

Operate MIDI control to synchronize: control 1002 on 'nanoKONTROL2'

☐ Generate MATLAB Code

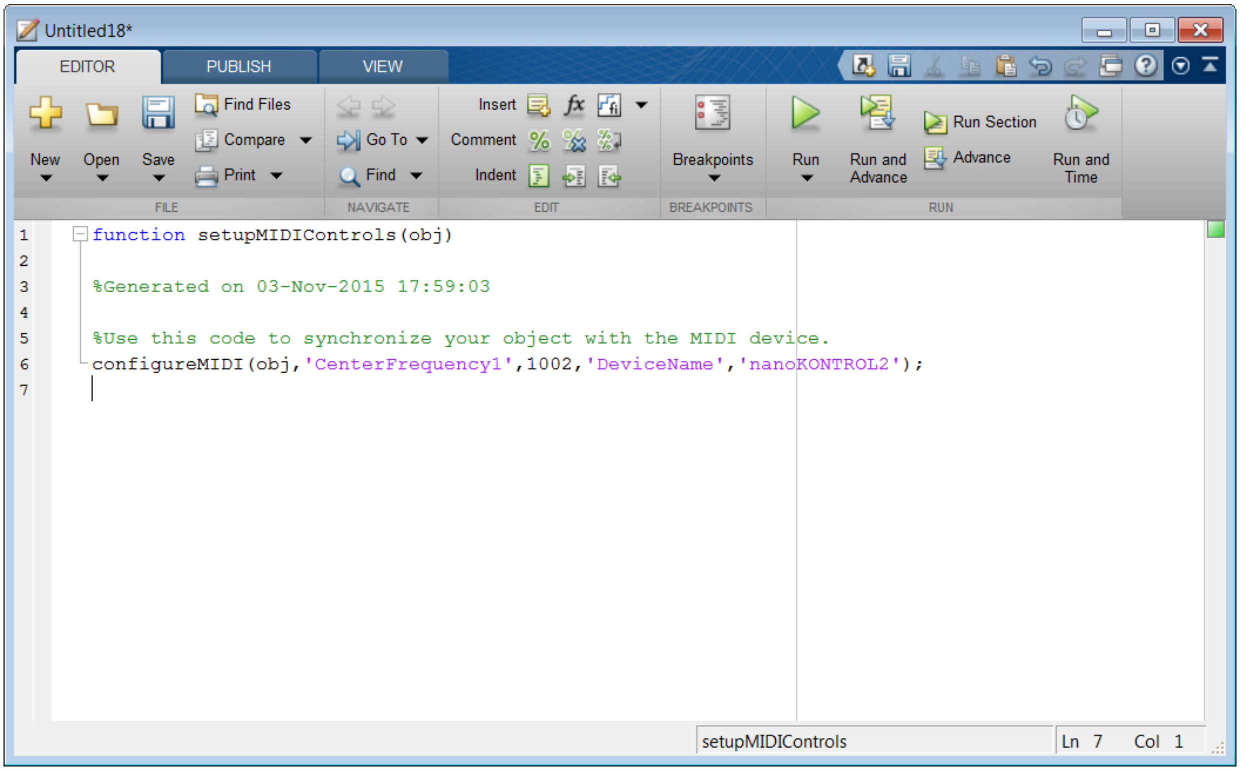
4 Select the **Generate MATLAB Code** check box.



Select property to synchronize:
CenterFrequency1
Operate MIDI control to synchronize:
control 1002 on 'nanoKONTROL2'

☑ Generate MATLAB Code

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



```
function setupMIDIControls(obj)

%Generated on 03-Nov-2015 17:59:03

%Use this code to synchronize your object with the MIDI device.
configureMIDI(obj,'CenterFrequency1',1002,'DeviceName','nanoKONTROL2');

```

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

`myAudioPlugin` — Audio plugin

object

Audio plugin, specified as an object that inherits from the `audioPlugin` class.

`propertyName` — Name of audio plugin property

character vector

Name of the audio plugin property, specified as a character vector. Enter the property name exactly as it is defined in the property section of your audio plugin class.

`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 plugin 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 plugin, 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

See Also

Classes
audioPlugin | audioPluginSource

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

Topics
"MIDI Control for Audio Plugins"
"Musical Instrument Digital Interface (MIDI)"

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 specifed parameters.

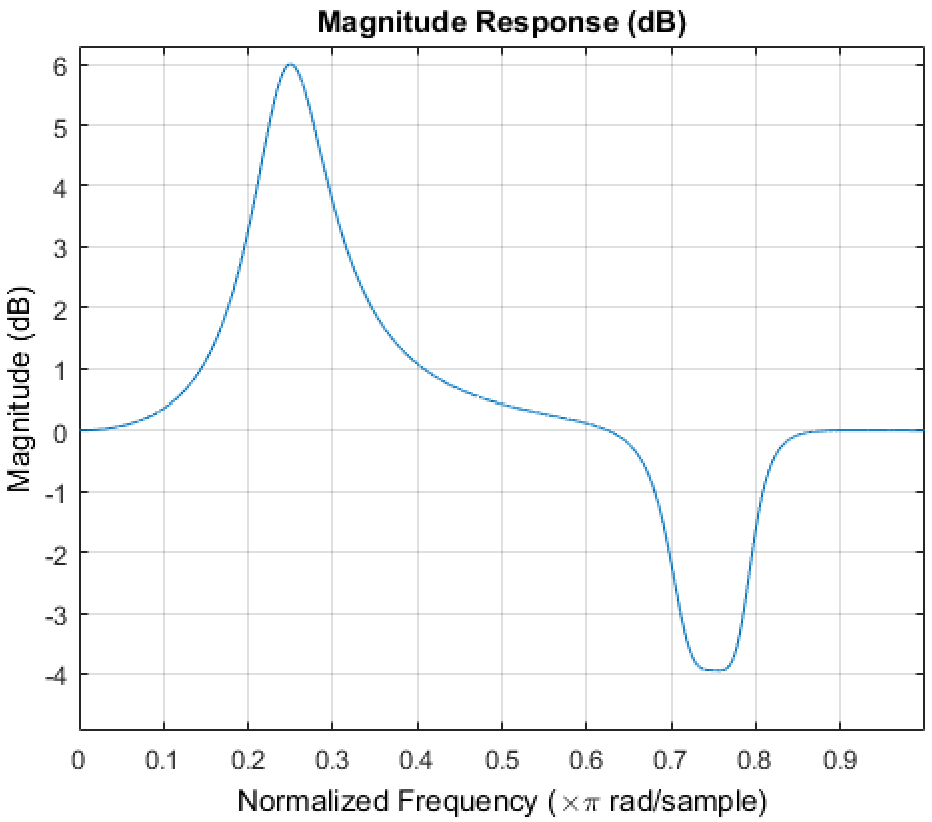
```
[B,A] = designParamEQ(N,gain,centerFreq,bandwidth);
```

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.

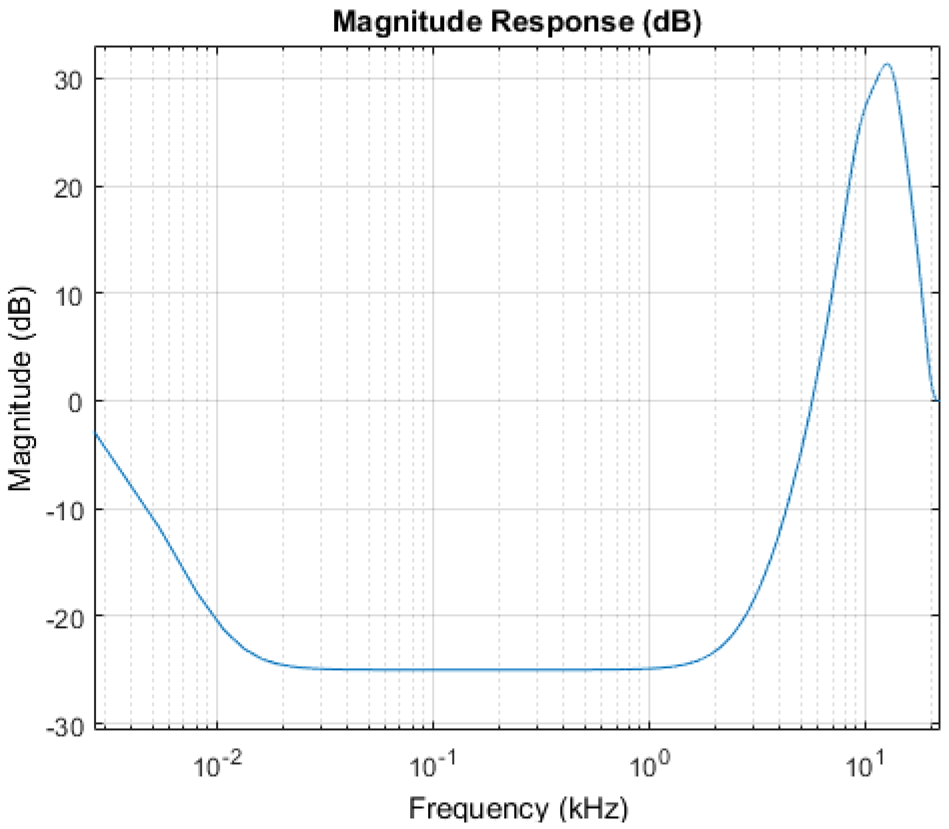
```
count = 0;
while count < 2500
    audio = fileReader();
    deviceWriter(audio);
    count = count+1;
end
reset(fileReader);
```

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.

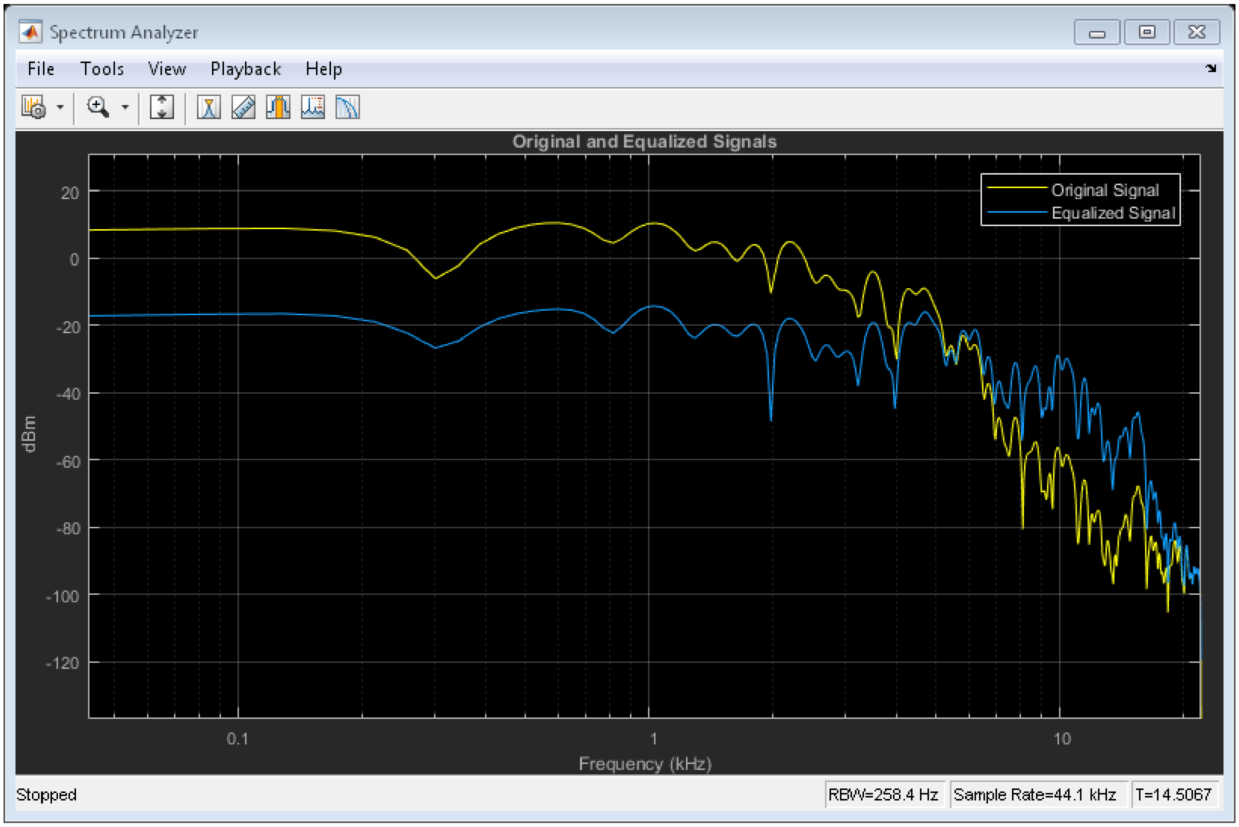
```
myFilter = dsp.BiquadFilter(...
    'SOSMatrixSource','Input port',...
    'ScaleValuesInputPort',false);
```

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.

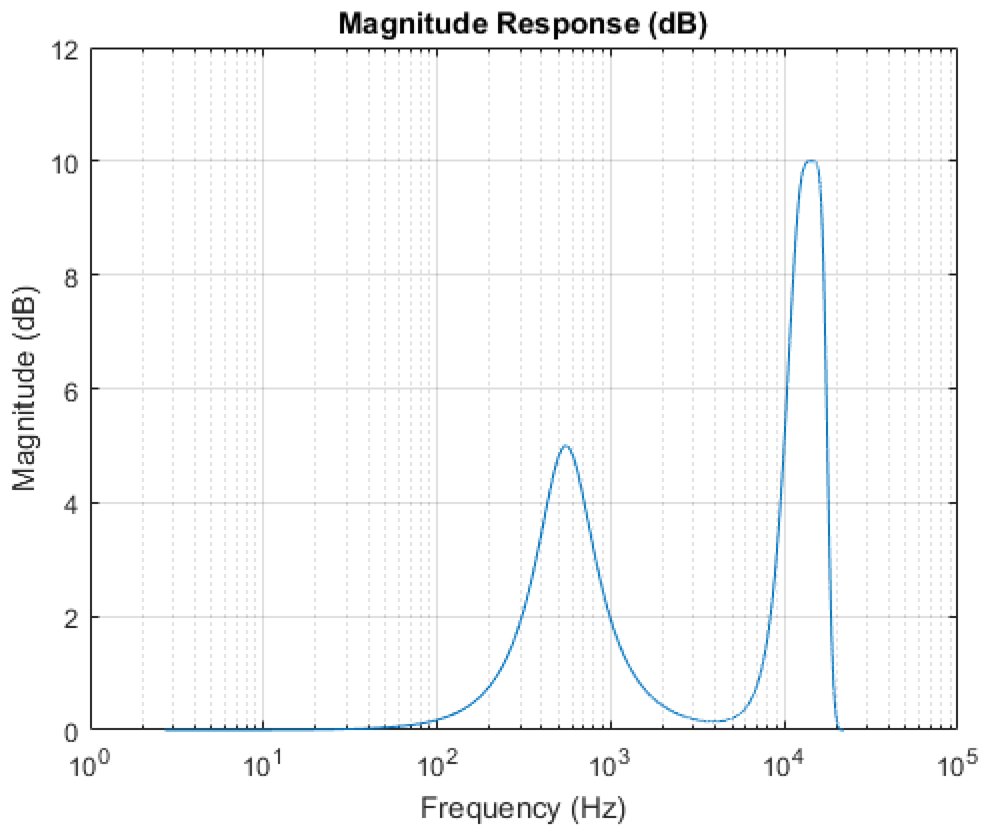
```
N = [2,4];
gain = [5,10];
centerFreq = [0.025,0.65];
bandwidth = [0.025,0.35];
mode = 'fos';

[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 parameteric 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
```

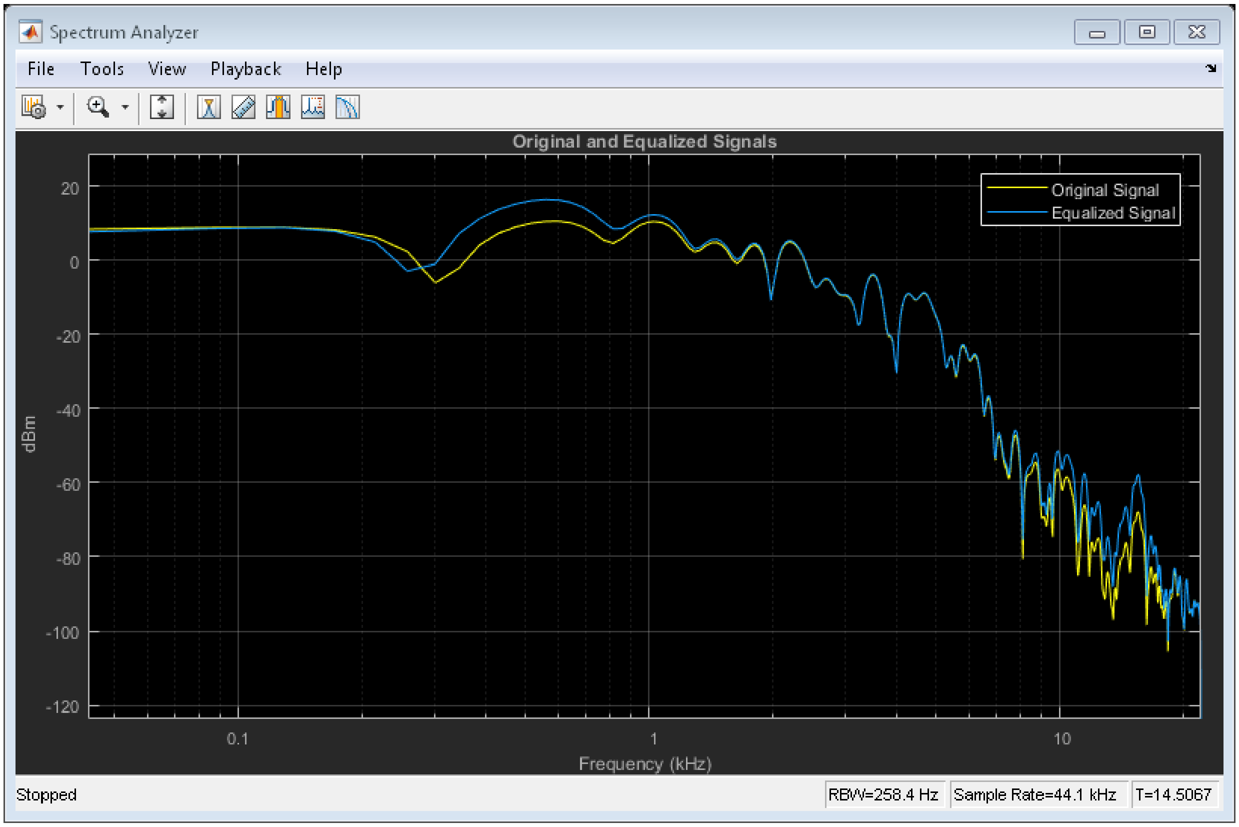


Construct a spectrum analyzer to visualize the original audio signal and the audio signal passed through your parametric equalizer.

```
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.

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.

`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`.

`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

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

`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.

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

See Also

Functions
`designShelvingEQ` | `designVarSlopeFilter`

System Objects
multibandParametricEQ | dsp.BiquadFilter

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.

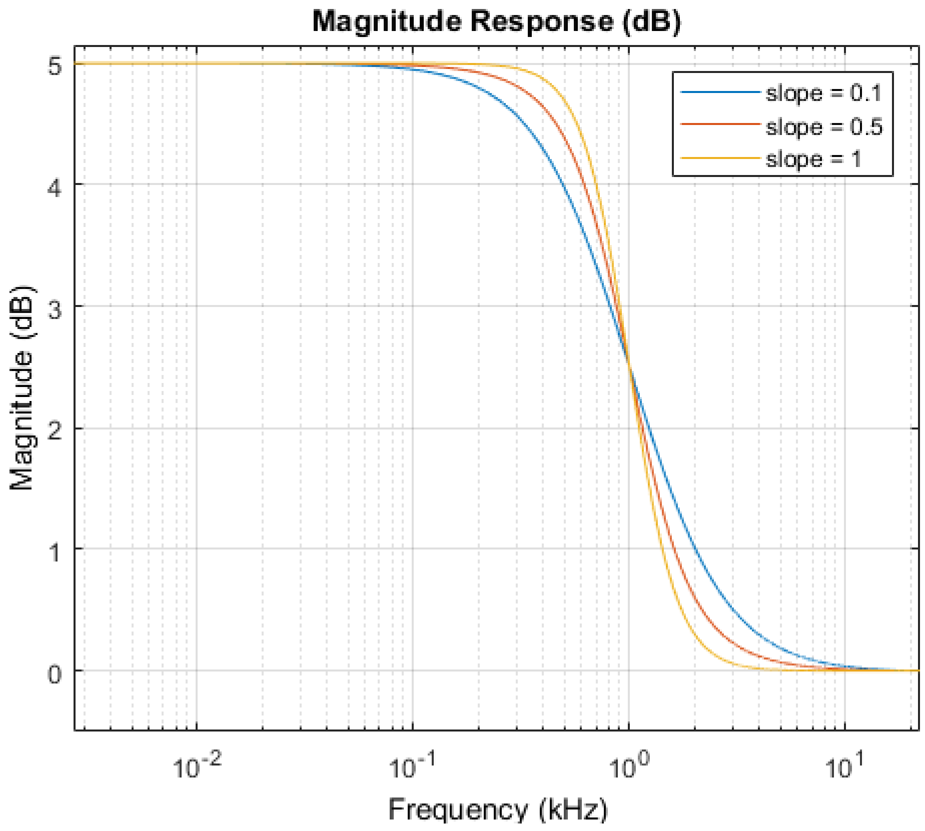
```
[B1,A1] = designShelvingEQ(gain,slope1,Fc);
[B2,A2] = designShelvingEQ(gain,slope2,Fc);
[B3,A3] = designShelvingEQ(gain,slope3,Fc);
```

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.

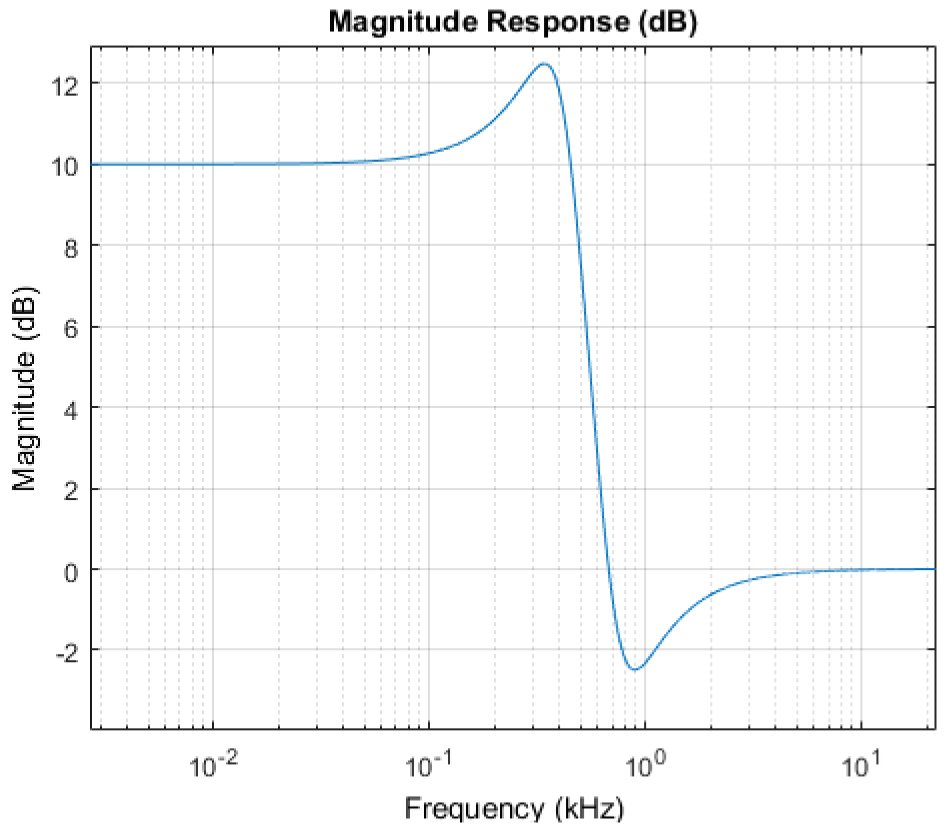
```
count = 0;
while count < 2500
    audio = step(fileReader);
    play(deviceWriter,audio);
    count = count+1;
end
reset(fileReader)
```

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.

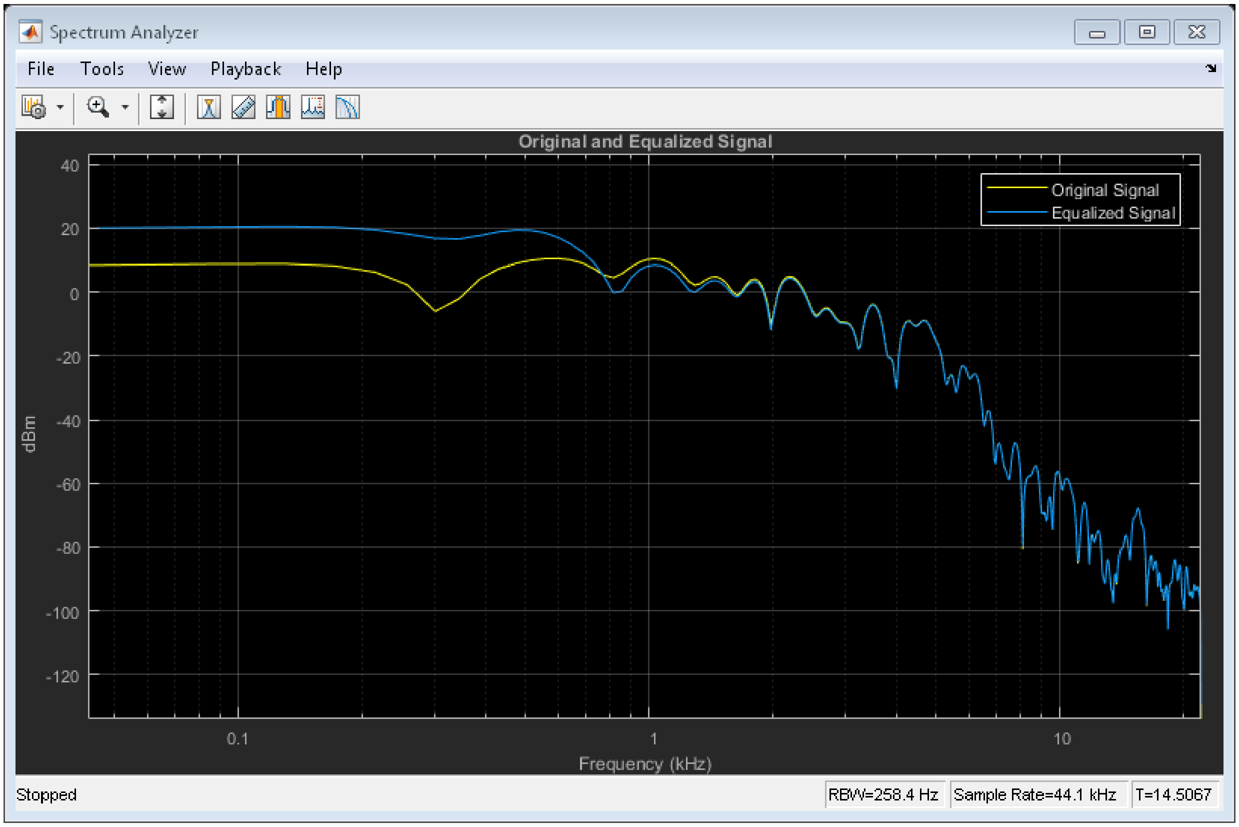
```
myFilter = dsp.BiquadFilter(...
    'SOSMatrixSource','Input port',...
    'ScaleValuesInputPort',false);
```

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 coefficents using the specified parameters.

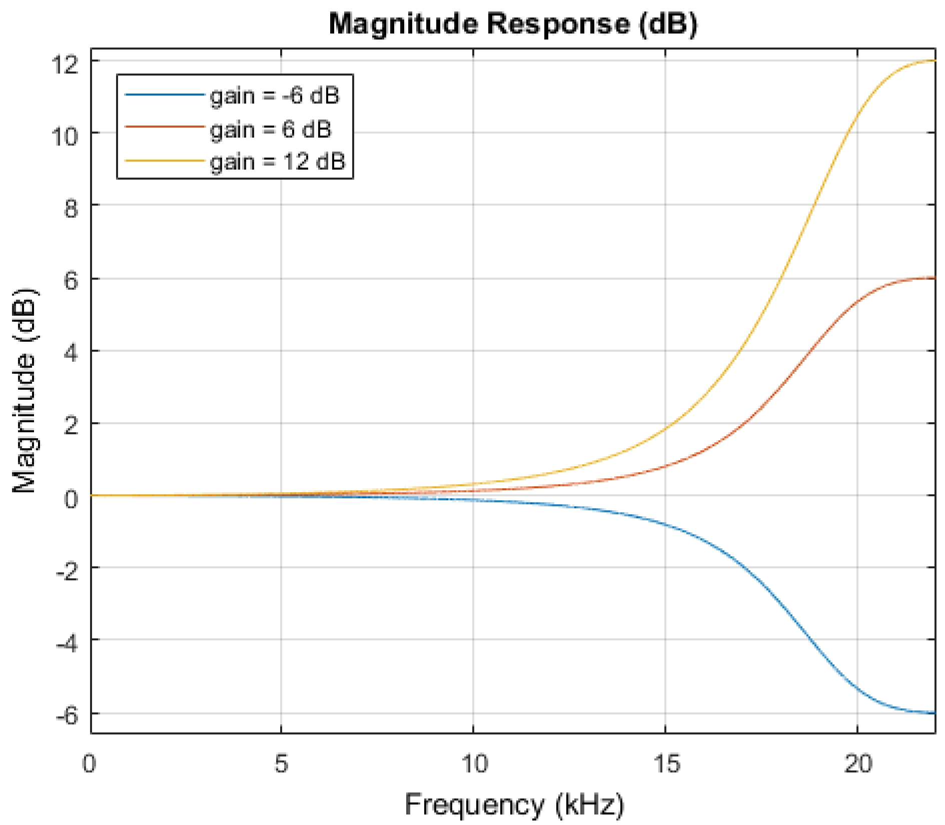
```
[B1,A1] = designShelvingEQ(gain1,slope,Fc,'hi');
[B2,A2] = designShelvingEQ(gain2,slope,Fc,'hi');
[B3,A3] = designShelvingEQ(gain3,slope,Fc,'hi');
```

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.

`slope` — Slope coefficient

real scalar in the range 0 to 5

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

`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.

`type` — Filter type

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

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

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

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

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.

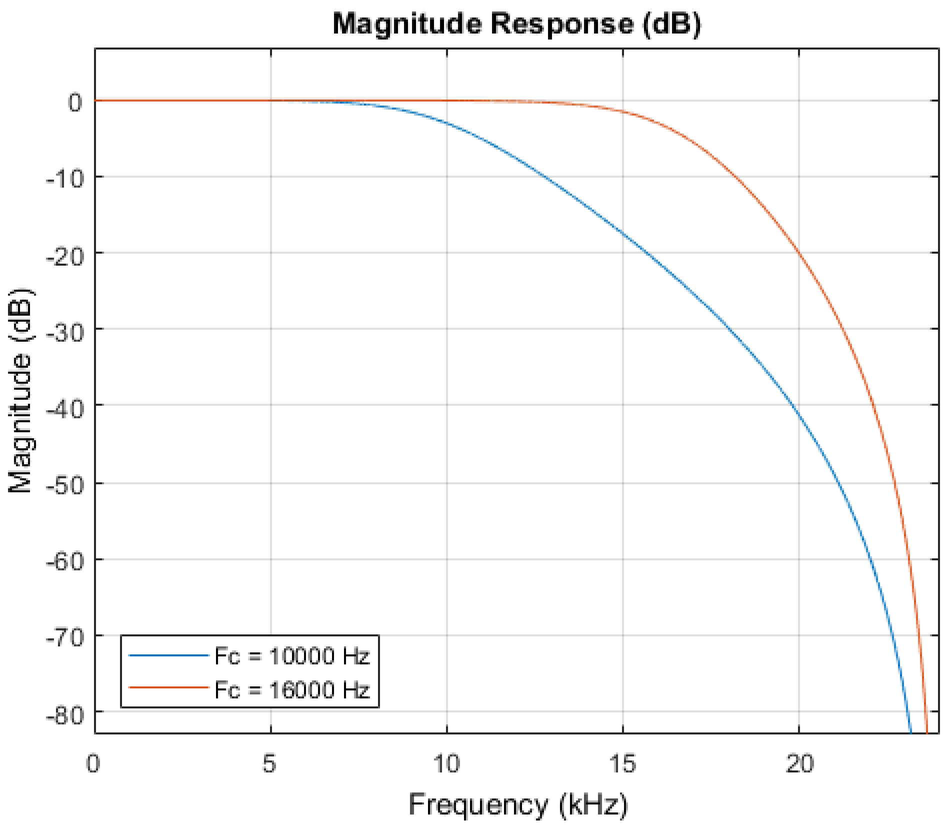
```
[B1,A1] = designVarSlopeFilter(slope,Fc1);
[B2,A2] = designVarSlopeFilter(slope,Fc2);
```

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.

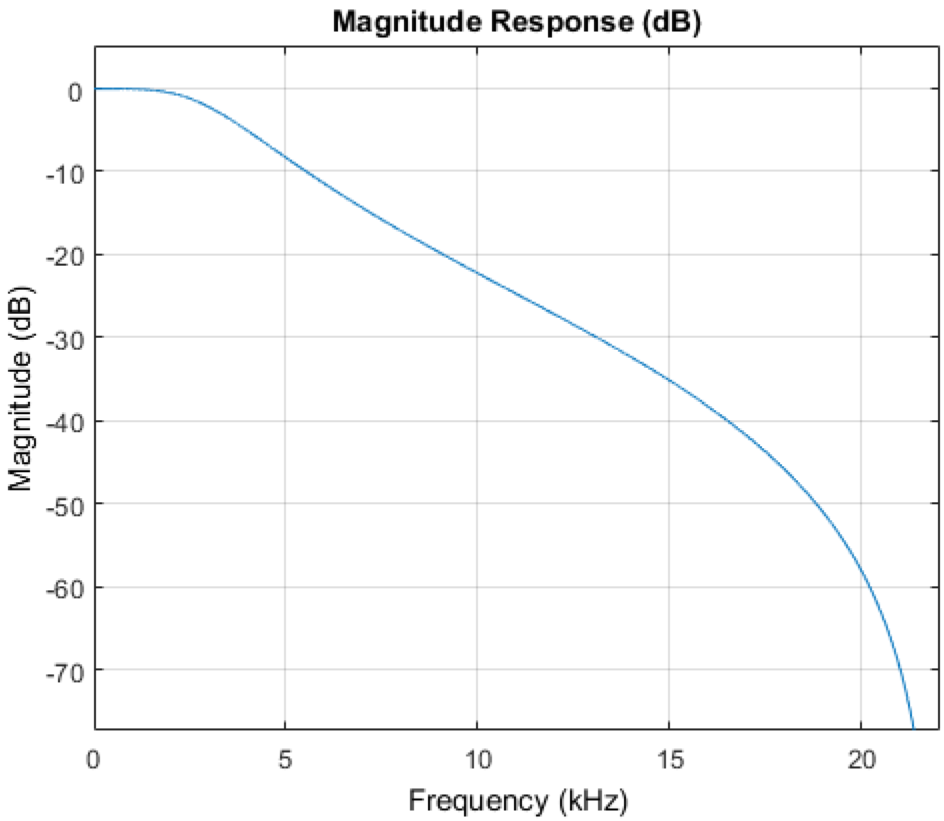
```
count = 0;
while count < 2500
    audio = fileReader();
    deviceWriter(audio);
    count = count+1;
end
reset(fileReader);
```

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.

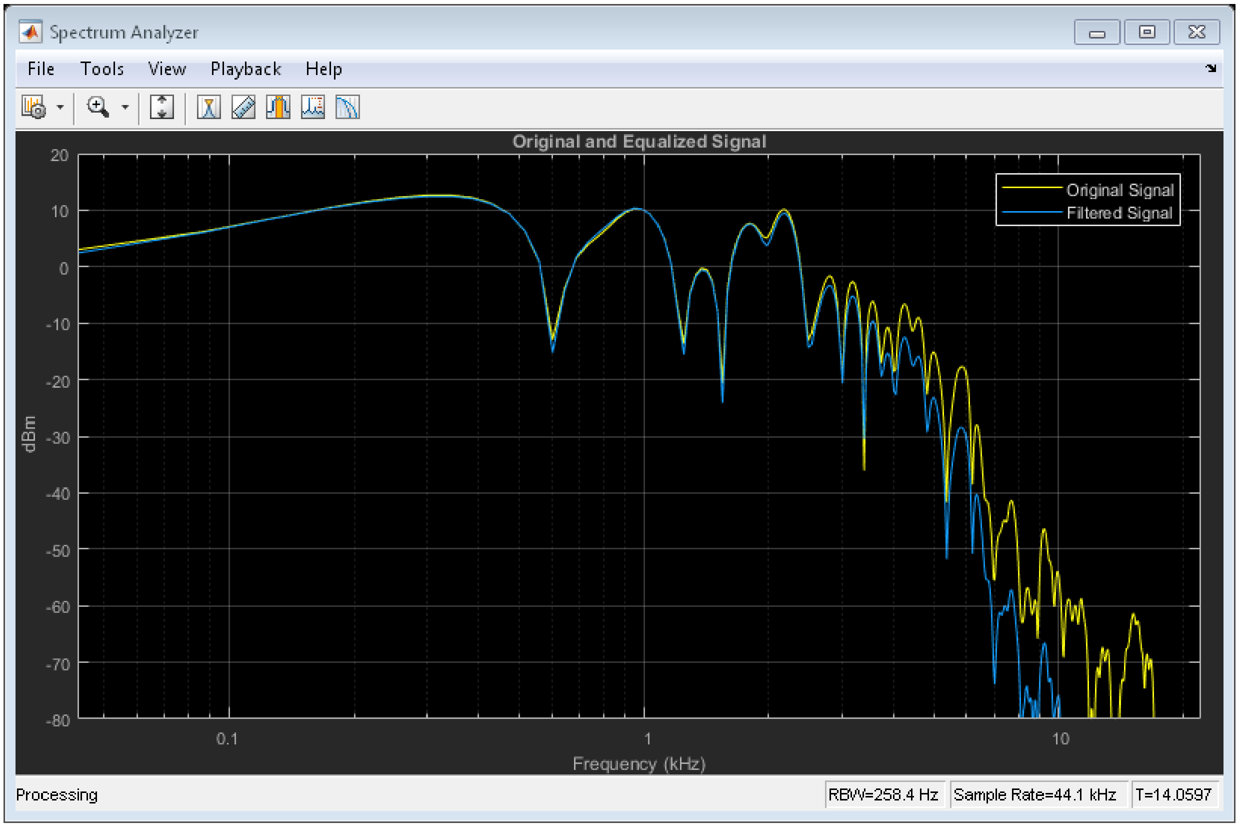
```
myFilter = dsp.BiquadFilter( ...
    'SOSMatrixSource','Input port', ...
    'ScaleValuesInputPort',false);
```

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 specifed parameters.

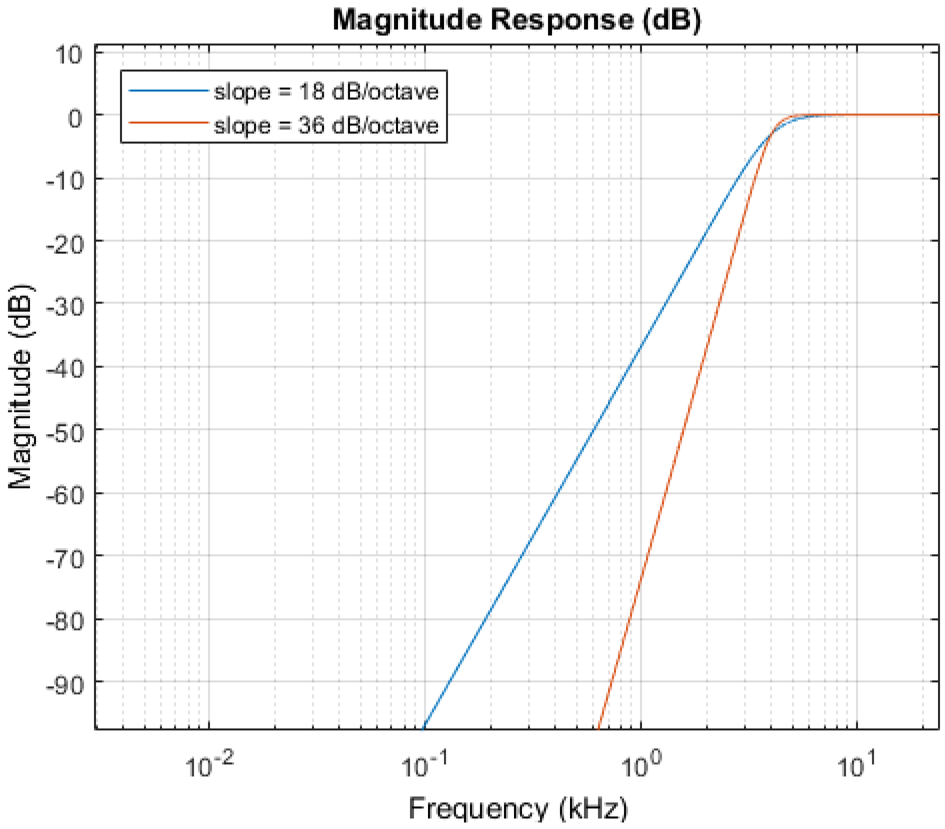
```
[B1,A1] = designVarSlopeFilter(slope1,Fc,'hi');
[B2,A2] = designVarSlopeFilter(slope2,Fc,'hi');
```

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 *p*, *d*, and *g* words. 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 `audioDeviceWriter` System object™ to read an audio signal from a file and write an audio signal to a device. Listen to 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.

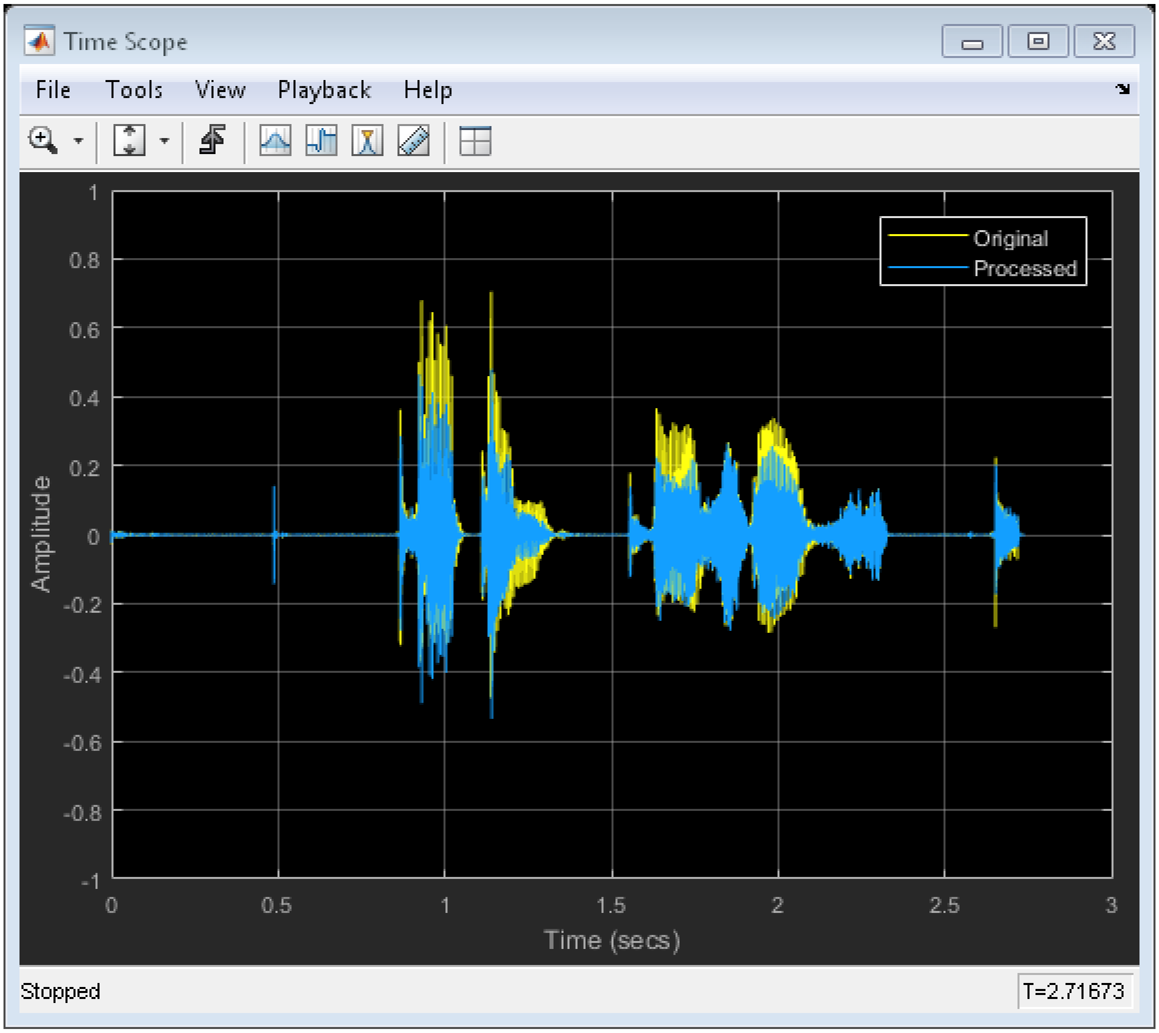
```
[B,A] = designVarSlopeFilter(48,120/(44100/2),'hi');
biquadFilter = dsp.BiquadFilter( ...
    'SOSMatrixSource','Input port', ...
    'ScaleValuesInputPort',false);

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.

`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).

`type` — Filter type

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

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

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

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

See Also

Functions
`designParamEQ` | `designShelvingEQ`

System Objects
multibandParametricEQ

Topics
"Parametric Equalizer Design"
"Equalization"

Introduced in R2016a

disconnectMIDI

Disconnect MIDI controls from audio plugin

Syntax

```
disconnectMIDI(myAudioPlugin)
```

Description

`disconnectMIDI(myAudioPlugin)` disconnects MIDI controls from your audio plugin object, `myAudioPlugin`. 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

`myAudioPlugin` — Audio plugin
object

Audio plugin, specified as an object that inherits from the `audioPlugin` class or the `audioPluginSource` class.

See Also

See Also

Classes
audioPlugin | audioPluginSource

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

Topics
"MIDI Control for Audio Plugins"
"Musical Instrument Digital Interface (MIDI)"

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	

Parameter	Definition	Unit
`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`

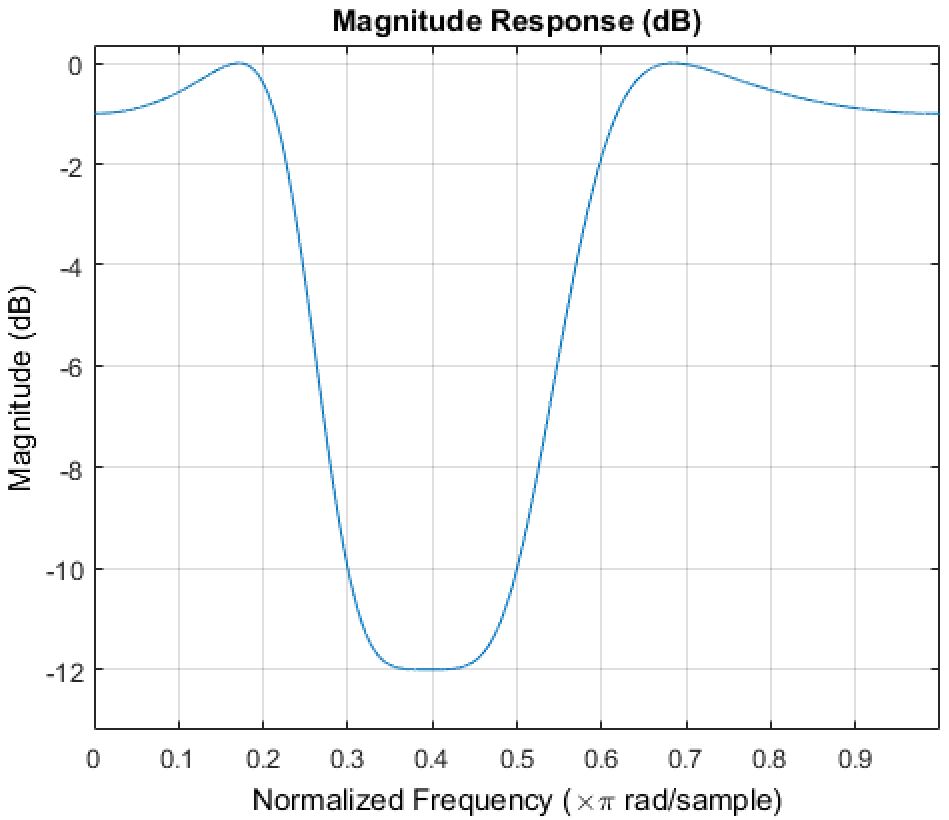
`d = fdesign.parameq(... fs)` adds the input sampling frequency. `fs` must be specified as a scalar trailing the other numerical values provided, and is assumed to be in Hz.

Examples

Design Parametric Equalizers

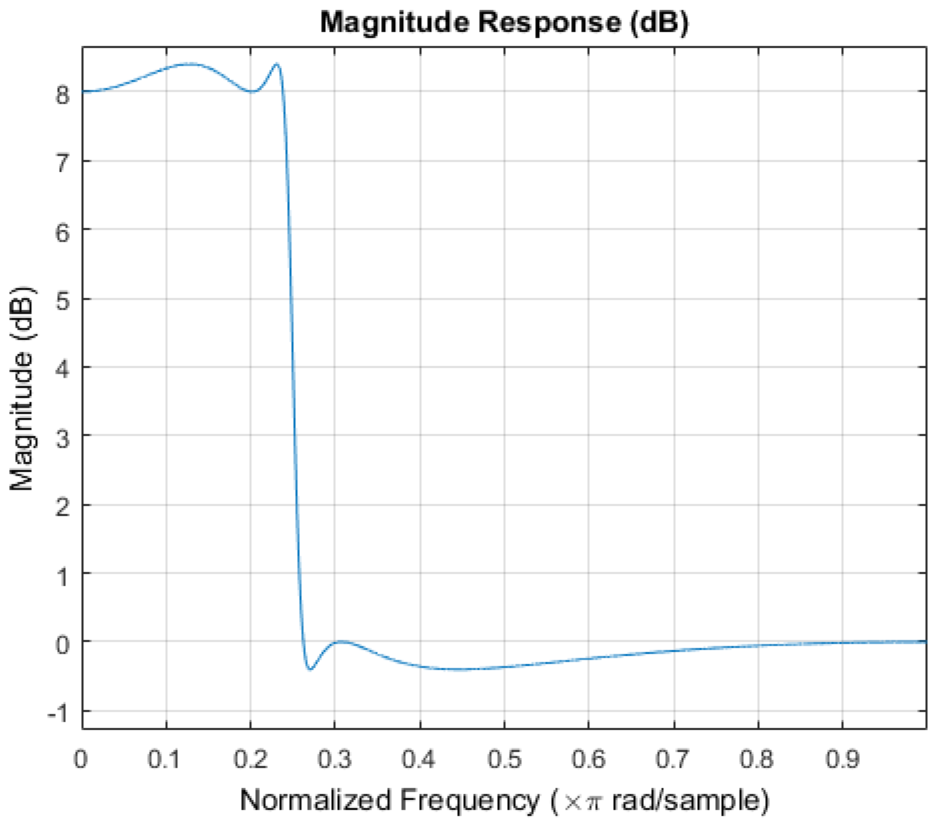
Design a Chebyshev Type II parametric equalizer filter that cuts by 12 dB.

```
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.

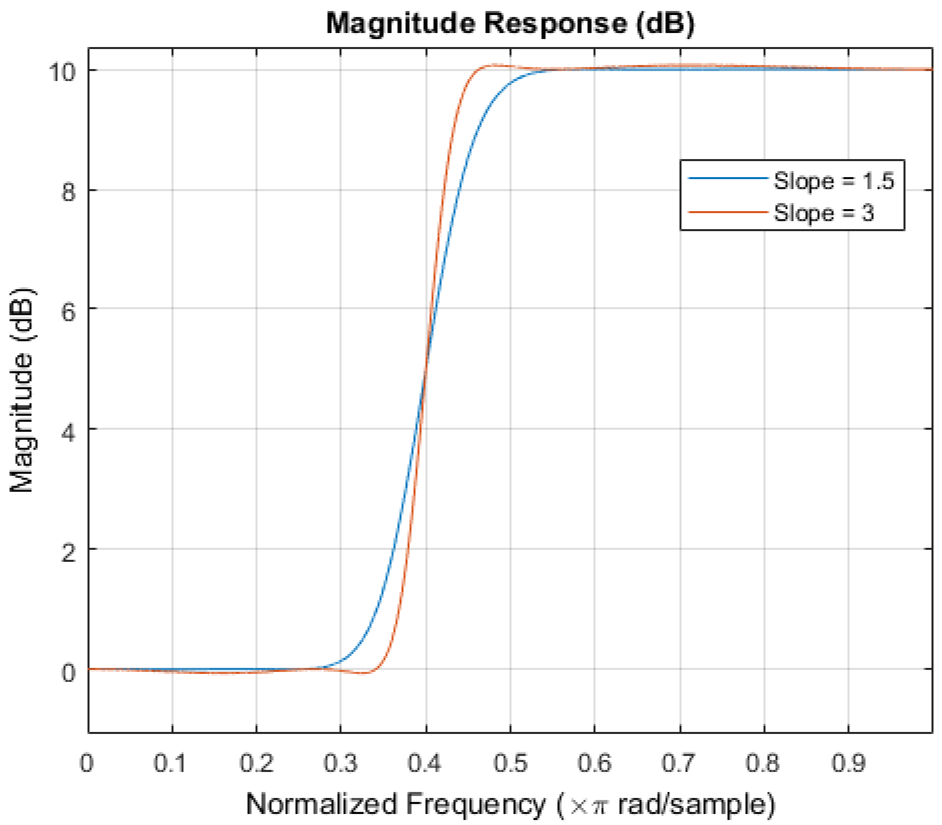
```
parametricEQ  = fdesign.parameq('N,F0,Fc,Qa,G0',4,0,0.25,10,8);
parametricEQBiquad = design(parametricEQ,'SystemObject',true);
fvtool(parametricEQBiquad)
```



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

```
N  = 4;   % Filter order
F0 = 1;   % Center Frequency
Fc = 0.4; % Cutoff Frequency
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

fdesign | design | designShelvingEQ | multibandParametricEQ | designParamEQ | designVarSlopeFilter

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” on page 2-84 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.
`-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.

Note: `className` is not the name of a file. Arguments such as `'myPlugin.m'` issue an error.

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

Supported Compilers

Compilers supported by `generateAudioPlugin`.

Operating System	Supported Compilers
win64	Microsoft Visual C++ 2015 Professional Microsoft Visual C++ 2013 Professional Microsoft Visual C++ 2012 Professional Visual Studio Community 2013 is reported to work. Full support of Visual Studio Community 2013 has not been qualified by MathWorks®. Visual Studio Community 2015 is reported to work. Full support of Visual Studio Community 2015 has not been qualified by MathWorks.
maci64	Xcode 6.2

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

See Also

Apps
Audio Test Bench

Functions
`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)
```

```
loudness =

   -3.0036
```

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)
```

```
standardLoudness =

  -11.6825


nonStandardLoudness =

  -11.0121
```

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);
```

```
Loudness: -22.93
Loudness range: 1.50

Beginning loudness: -23.29
Beginning loudness range: 1.17

Middle loudness: -22.99
Middle loudness range: 1.12

End loudness: -22.10
End loudness range: 1.82
```

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

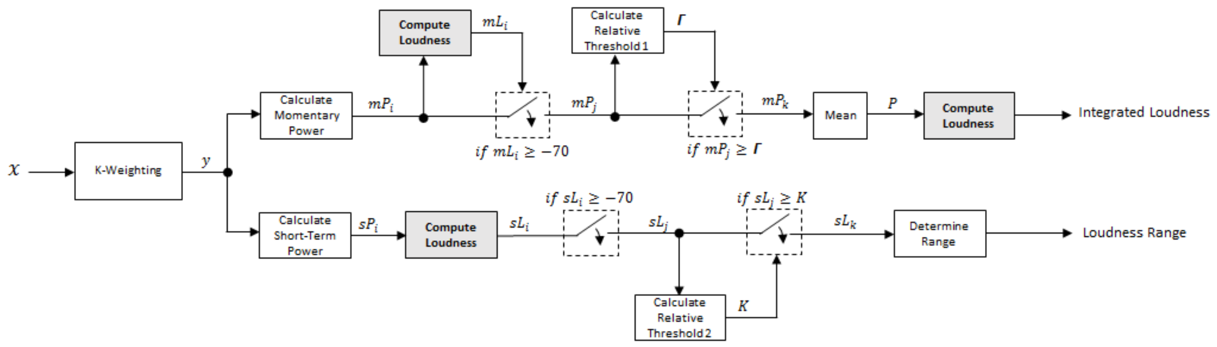
Loudness range in loudness units (LU), returned as a scalar.

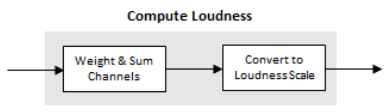
The EBU R 128 Tech 3342 standard defines the loudness range. The algorithm computes the loudness range by breaking down the audio into 3-second segments with 2.9-second overlap. If the input signal is less than three seconds, `loudnessRange` is returned empty.

Data Types: `single` | `double`

Algorithm

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.

Integrated 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:

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

Loudness Range

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:

$$sL_j \rightarrow sL_k$$

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

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

See Also

System Objects
loudnessMeter | weightingFilter

Blocks
`Loudness Meter`

Introduced in R2016b

getMIDIConnections

Get MIDI connections of audio plugin

Syntax

```
connectionInfo = getMIDIConnections(myAudioPlugin)
```

Description

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

The `connectionInfo` structure contains a substructure for each tunable property of `myAudioPlugin` 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

`myAudioPlugin` — Audio plugin

object

Audio plugin, specified as an object that inherits from the `audioPlugin` class.

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

See Also

Classes
audioPlugin | audioPluginSource

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

Topics
"MIDI Control for Audio Plugins"
"Musical Instrument Digital Interface (MIDI)"

Introduced in R2016a

loadAudioPlugin

Load VST and VST3 plugins into MATLAB environment

Syntax

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

Description

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

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

- `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.
- `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].
- `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.
- `sampleRate = getSampleRate(hostedPlugin)`

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

- `setSampleRate(hostedPlugin,sampleRate)`

  Sets the sample rate of the hosted plugin to the value specified by `sampleRate`.
- `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).

Examples

Host External Plugin in MATLAB

Use `loadAudioPlugin` to host a VST external plugin and a VST external source plugin.

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 `externalAudioPlugin` or `externalAudioSourcePlugin`. Because `oscillator.dll` is a source audio plugin, the hosted object derives from `externalAudioSourcePlugin`. Use `class()` to verify the class 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 parameters of the plugin, which are summarized in a table. Source plugins also display the frame size of the plugin.

```
hostedSourcePlugin
hostedPlugin
```

```
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

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.
```

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 normalized value of the `Medium Peak Gain` parameter value to zero.

```
parameterValue = 0;
setParameter(hostedPlugin,'Medium Peak Gain',parameterValue)
```

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 parameterValue < 0.995
    parameterValue = parameterValue + 0.001;
    setParameter(hostedPlugin,'Medium Peak Gain',parameterValue);
    x = fileReader();
    y = process(hostedPlugin,x);
    deviceWriter(y);
end

release(fileReader)
release(deviceWriter)
```

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 normalized value of the `Amplitude` parameter to 0.05. Set the normalized value of the `Frequency` parameter to 0.8.

```
setParameter(hostedSourcePlugin,'Amplitude',0.05);
frequencyParameterValue = 0.8;
setParameter(hostedSourcePlugin,'Frequency',frequencyParameterValue);
```

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
    frequencyParameterValue = frequencyParameterValue - 0.0004*k;
    setParameter(hostedSourcePlugin,'Frequency',frequencyParameterValue);
    y = process(hostedSourcePlugin);
    deviceWriter(y);
    if i == 500
        k = -1;
    end
end
```

```
release(deviceWriter)
```

Input Arguments

`pluginpath` — Location of external plugin
character vector

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 Macs

For Macs, 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`

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`

Output Arguments

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

Object of an external plugin, derived from `externalAudioPlugin` or `externalAudioSourcePlugin`. 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

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;
```

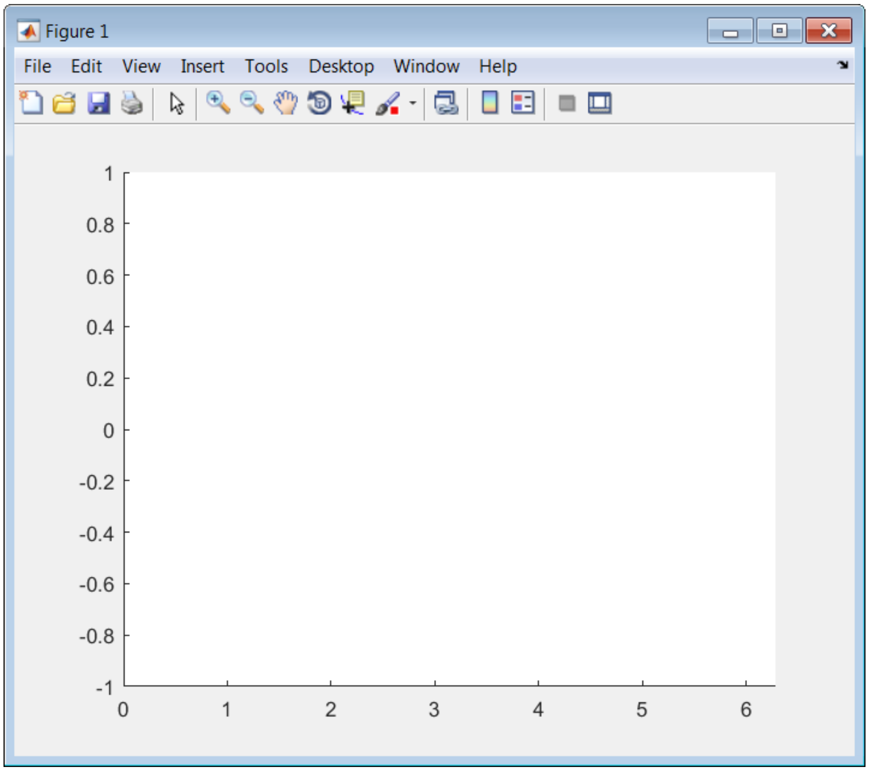
```
Move the control you wish to identify; type ^C to abort.
Waiting for control message...
```

Create an object that responds to the control you specified.

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

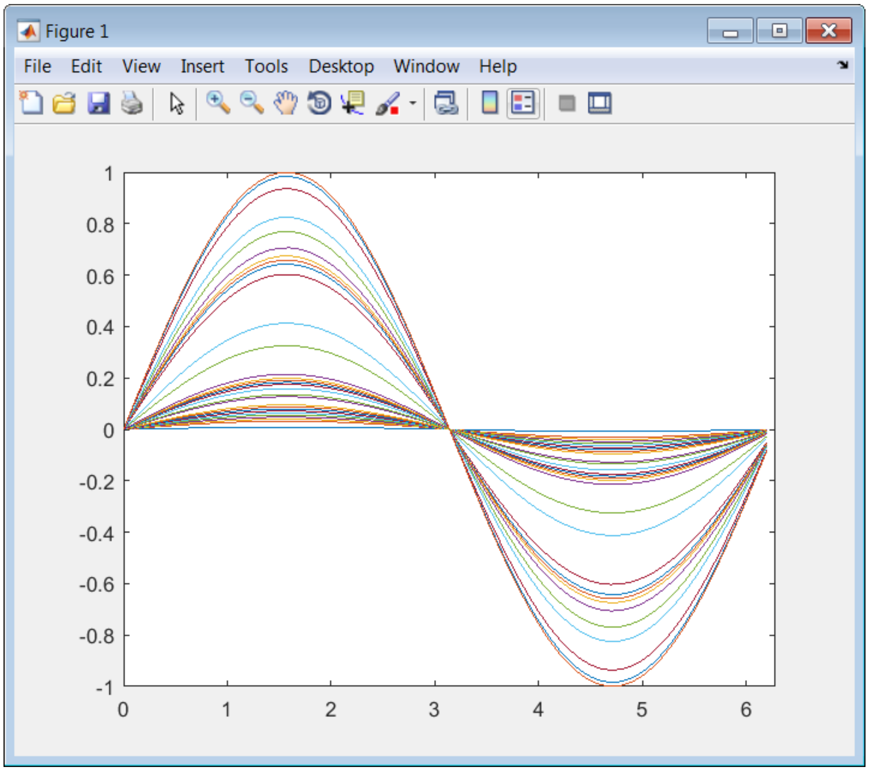
Define a function that plots a sinusoid with the amplitude set by your MIDI control. Make the axis constant.

```
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

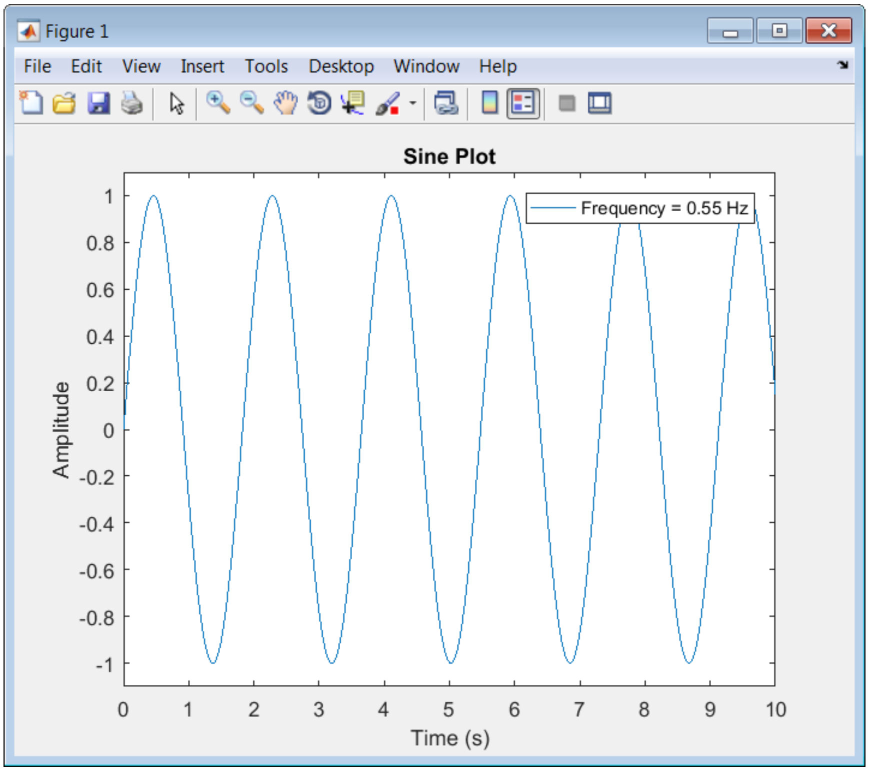
Define this function and save it to your current folder.

```
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

See Also

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

Topics
"Musical Instrument Digital Interface (MIDI)"
"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 2-122 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

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 2-119 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

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

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

See Also

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

Topics
"Musical Instrument Digital Interface (MIDI)"
"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

See Also

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

Topics
"Musical Instrument Digital Interface (MIDI)"
"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

See Also

Functions

`configureMIDI` | `disconnectMIDI` | `getMIDIConnections` | `midicallback` | `midicontrols` | `midiid` | `midisync` | `setpref`

Topics

"Musical Instrument Digital Interface (MIDI)"
"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

See Also

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

Topics
“Musical Instrument Digital Interface (MIDI)”
“MIDI Control for Audio Plugins”

validateAudioPlugin

Test MATLAB source code for audio plugin

Syntax

```
validateAudioPlugin pluginClass
validateAudioPlugin options pluginClass
```

Description

`validateAudioPlugin pluginClass` generates and runs a “Test Bench Procedure” on page 2-135 that exercises your audio plugin class.

`validateAudioPlugin options pluginClass` specifies options to modify the default “Test Bench Procedure” on page 2-135.

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.

`pluginClass` — 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.

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

See Also

Functions
`generateAudioPlugin`

Classes
audioPlugin | audioPluginSource

Topics
“Design an Audio Plugin”

Introduced in R2016a

3

System objects in Audio System Toolbox

audioPlayerRecorder System object

Simultaneously play and record using an audio device

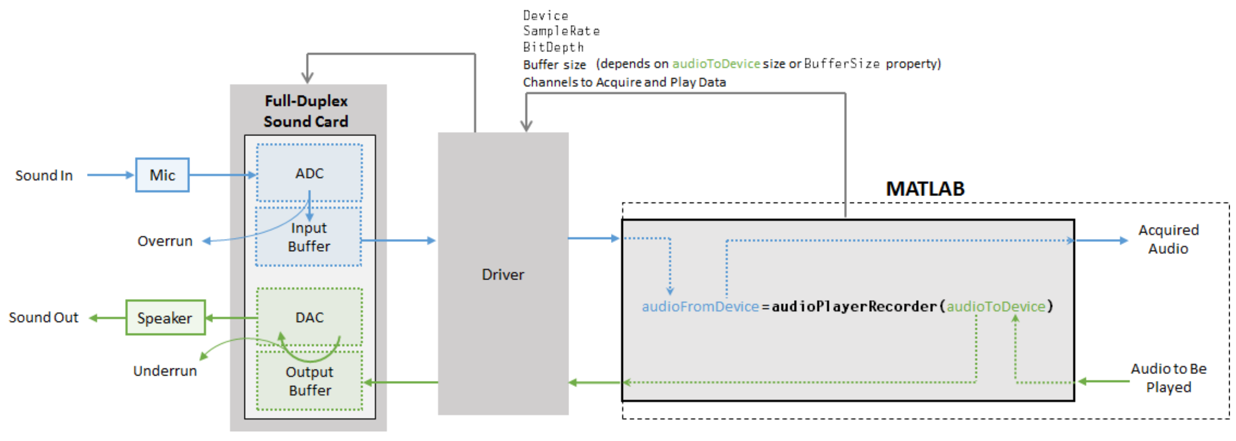
Description

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.

To simultaneously play and record:

1. Define and set up your audio player recorder. See "Construction" on page 3-3.
2. Call step to stream data from and to your audio device.

Note: Alternatively, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.



Construction

`aPR = audioPlayerRecorder` returns a System object, `aPR`, that plays audio samples to an audio device, and records samples from the same audio device, in real time.

`aPR = audioPlayerRecorder(sampleRateValue)` sets the `SampleRate` property to `sampleRateValue`.

`aPR = audioPlayerRecorder(Name,Value)` sets each property `Name` to the specified `Value`. Unspecified properties have default values.

Example: `aPR = audioPlayerRecorder(12000,'BitDepth','8-bit integer')` creates a System object, `aPR`, with the `SampleRate` property set to `12000` and the `BitDepth` property set to `'8-bit integer'`.

Properties

`Device` — Device used to play and record audio data

default audio device (default) | character vector

Device used to play and record audio data, specified as a character vector. The object supports only devices enabled for simultaneous playback and recording (full-duplex mode). Use the `getAudioDevices` method 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.

`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.

`BitDepth` — Data type used by device
`'16-bit integer'` (default) | `'8-bit integer'` | `'32-bit float'` | `'24-bit float'`

Data type used by device, specified as a character vector.

`SupportVariableSize` — Option to 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`.

`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, open the ASIO UI to set the sound card buffer size to the `BufferSize` value of your `audioPlayerRecorder` System object.

To enable this property, set `SupportVariableSize` to `true`.

`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.

`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.

Methods

getAudioDevices	List available audio devices
info	Get information about audio I/O system
step	Stream audio data from and to device

Common to All System Objects	
`clone`	Create System object with same property values
`getNumInpu`	Expected number of inputs to a System object
`getNumOutp`	Expected number of outputs of a System object
`isLocked`	Check locked states of a System object (logical)
`release`	Allow System object property value changes

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 resoures.

```
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 6400 samples.
Audio recorder queue was overrun by 3840 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.

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`

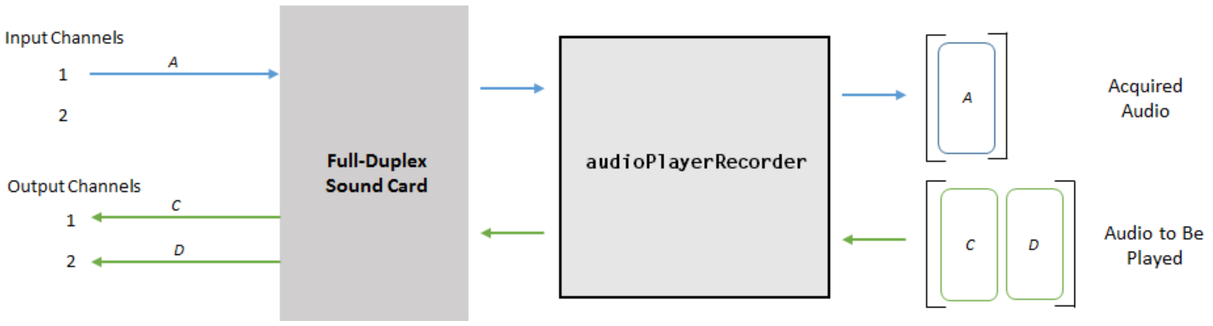
By default, `audioFromDevice` is a one-column matrix corresponding to channel 1 of your audio device. To view the maximum number of input and output channels of your device, use the `info` method.

```
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 one is mapped to output channel one, and column two is mapped to output channel two. The `audioPlayerRecorder` returns a one-column matrix with the same number of rows as the `audioToDevice` matrix.

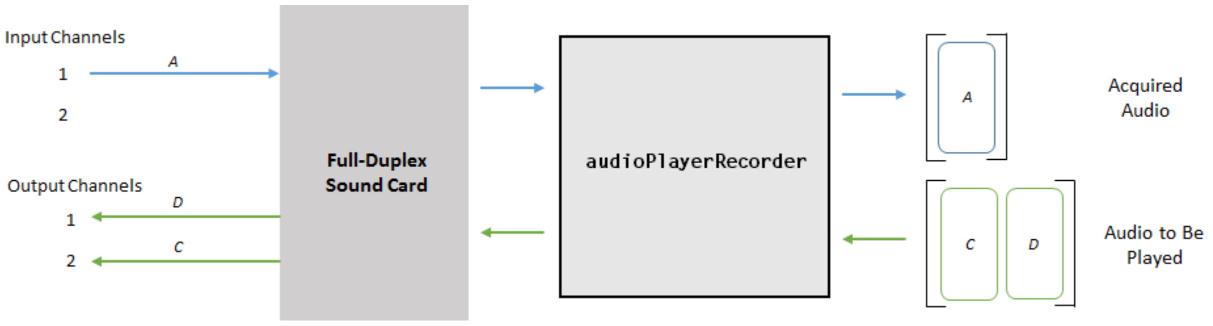
```
highToneGenerator = audioOscillator('Frequency',600,'SamplesPerFrame',256);
lowToneGenerator = audioOscillator('Frequency',200,'SamplesPerFrame',256);

for i = 1:250
    C = highToneGenerator();
    D = lowToneGenerator();
    audioToDevice = [C,D];
    audioFromDevice = aPR(audioToDevice);
end
```



Specify a nondefault channel mapping for your audio output. Specify that column one of `audioToDevice` maps to channel two, and that column two of `audioToDevice` maps to channel one. 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.

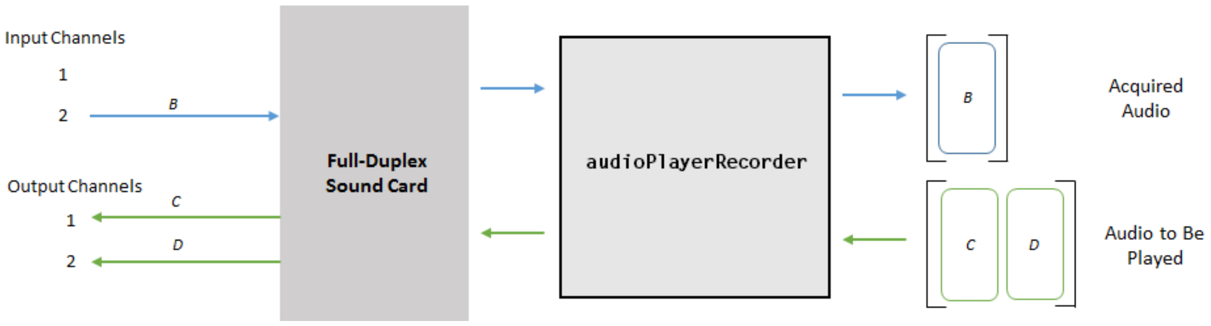
```
release(aPR)
aPR.PlayerChannelMapping = [2,1];

for i = 1:250
    C = highToneGenerator();
    D = lowToneGenerator();
    audioToDevice = [C,D];
    audioFromDevice = aPR(audioToDevice);
end
```



Specify a nondefault channel mapping for your audio input. Record data from only channel two of your device. In this case, channel two 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)

See Also

See Also

Blocks
`Audio Device Reader` | `Audio Device Writer`

System Objects
audioDeviceWriter | audioDeviceReader | 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

getAudioDevices

System object: audioPlayerRecorder

List available audio devices

Syntax

```
devices = getAudioDevices(aPR)
```

Description

`devices = getAudioDevices(aPR)` returns a cell array listing available audio devices. The list of available audio devices support full-duplex mode and have an appropriate driver:

- Windows — ASIO
- Mac — CoreAudio
- Linux — ALSA

Introduced in R2017a

info

System object: audioPlayerRecorder

Get information about audio I/O system

Syntax

```
aPRInfo = info(aPR)
```

Description

`aPRInfo = info(aPR)` returns a structure containing information about your `audioPlayerRecorder` System object. The structure contains information about the driver, device, maximum recorder channels, and maximum player channels for your `audioPlayerRecorder` System object.

Introduced in R2017a

step

System object: audioPlayerRecorder

Stream audio data from and to device

Syntax

```
[audioFromDevice,numUnderrun,numOverrun] = step(aPR,audioToDevice)
```

Description

Note: Alternatively, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj)` and `y = obj()` perform equivalent operations.

`[audioFromDevice,numUnderrun,numOverrun] = step(aPR,audioToDevice)` writes one frame of audio samples, `audioToDevice`, to the selected audio device and returns a frame of audio samples from the device. `numUnderrun` is the number of samples by which the player queue was underrun since the last call to the `step` method. `numOverrun` is the number of samples by which the recorder queue was overrun since the last call to the `step` method.

When you call the `step` method of an `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 the `release` method of the `audioPlayerRecorder` System object.

Note: The System object performs an internal initialization the first time you execute `step`. This initialization locks nontunable properties and input specifications, such as the dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

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 Device Reader System Interaction" on page 3-25 for a visualization of how the `audioDeviceReader` acquires data.

To stream data from an audio device:

1. Define and set up your audio device reader. See "Construction" on page 3-15.
2. Call step or record to stream data from your audio device.

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`aDR = audioDeviceReader` returns a System object, `aDR`, that reads audio samples using an audio input device in real time.

`aDR = audioDeviceReader(sampleRateValue)` sets the `SampleRate` property to `sampleRateValue`.

`aDR = audioDeviceReader(sampleRateValue,samplesPerFrameValue)` sets the `SamplesPerFrame` property to `samplesPerFrameValue`.

`aDR = audioDeviceReader(Name,Value)` sets each property `Name` to the specified `Value`. Unspecified properties have default values.

Example: `aDR = audioDeviceReader(12000,'BitDepth','8-bit integer')` creates a System object, `aDR`, with the `SampleRate` property set to `12000` and the `BitDepth` property set to `'8-bit integer'`.

Properties

`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'`, open the ASIO UI outside of MATLAB to set the sound card buffer size to the `SamplesPerFrame` value of your `audioDeviceReader` System object. See your ASIO driver documentation 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 `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.

`Device` — Device used to acquire audio samples
default audio device (default) | character vector

Device used to acquire audio samples, specified as a character vector. Use the `getAudioDevices` method to list available devices.

`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.

This property is available when you set `ChannelMappingSource` to `'Auto'`.

`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 when calling `step` or `record` on `audioDeviceReader`.

`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.

`BitDepth` — Data type used by device to acquire audio data
`'16-bit integer'` (default) | `'8-bit integer'` | `'32-bit float'` | `'24-bit float'`

Data type used by device to acquire audio data, specified as a character vector.

`ChannelMappingSource` — Source of mapping between channels of input device and columns of 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 `step` or `record` is a six-column matrix. Column 1 corresponds with channel 1, column 2 corresponds with channel 2, and so on.
- `'Property'` — The `ChannelMapping` property determines mapping between channels of your audio device and columns of the output matrix.

`ChannelMapping` — Nondefault mapping between channels of input device and columns of 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 the “Specify Channel Mapping for `audioDeviceReader`” on page 3-22 example for more information.

This property is available when you set `ChannelMappingSource` to `'Property'`.

`OutputDataType` — Data type of the output
`'double'` (default) | `'single'` | `'int32'` | `'int16'` | `'uint8'`

Data type of the output, specified as a character vector.

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.

Methods

getAudioDevices	List available audio input devices
info	Get information about selected device
record	Stream audio data from input device
step	Stream audio data from input device

Common to All System Objects	
`clone`	Create System object with same property values
`getNumInpu`	Expected number of inputs to a System object
`getNumOutp`	Expected number of outputs of a System object
`isLocked`	Check locked states of a System object (logical)
`release`	Allow System object property value changes

Examples

Read from Microphone and Write to Audio File

Record ten 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.');
```

```
Speak into microphone now.
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
```

```
deviceReader = 

  audioDeviceReader with properties:

             Driver: 'DirectSound'
             Device: 'No audio input device detected'
        NumChannels: 1
    SamplesPerFrame: 1024
         SampleRate: 44100

  Use get to show all properties
```

Calculate the latency due to your device buffer.

```
fprintf('Latency due to device buffer: %f seconds.\n',...
    deviceReader.SamplesPerFrame/deviceReader.SampleRate)
```

```
Latency due to device buffer: 0.023220 seconds.
```

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)
```

```
Latency due to device buffer: 0.001451 seconds.
```

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)
```

```
Latency due to device buffer: 0.000667 seconds.
```

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 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: 0.
Total seconds overrun: 0.
```

Release your `audioDeviceReader` and `dsp.AudioDeviceWriter` System objects and zero your counter variable.

```
release(fileWriter);
release(deviceReader);
totalOverrun = 0;
```

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: 93440.
Total seconds overrun: 2.118821e+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: 0.
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 = 
  System: audioDeviceReader

  Properties:
                Driver: 'DirectSound'
                Device: 'Default'
       SamplesPerFrame: 1024
            SampleRate: 44100


  Advanced properties:
              BitDepth: '16-bit integer'
    ChannelMappingSource: 'Property'
        ChannelMapping: [1 2]
        OutputDataType: 'double'
```

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];
```

If 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.

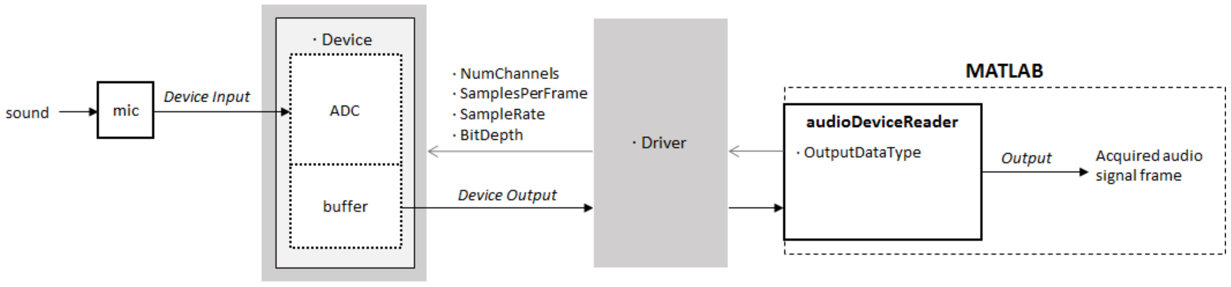
```
deviceReader.ChannelMapping = 2;
```

If you call your `audioDeviceReader`, input channel 2 of your device maps to an output vector.

More About

Audio Device Reader System Interaction

The audio device reader specifies the driver, the device and its attributes, and the data type and size output from your System object.



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

See Also

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

getAudioDevices

System object: audioDeviceReader

List available audio input devices

Syntax

```
devices = getAudioDevices(aDR)
```

Description

`devices = getAudioDevices(aDR)` returns a cell array listing available audio input devices. The list of available input devices depends on the specified `Driver` property of your `audioDeviceReader` object.

Introduced in R2016a

info

System object: audioDeviceReader

Get information about selected device

Syntax

```
aDRInfo = info(aDR)
```

Description

`aDRInfo = info(aDR)` returns a structure containing information about your `audioDeviceReader` System object. The structure contains information about the driver, device, and maximum number of input channels for your `audioDeviceReader` System object.

Introduced in R2016a

record

System object: audioDeviceReader

Stream audio data from input device

Syntax

```
[x,numOverrun] = record(aDR)
```

Description

`[x,numOverrun] = record(aDR)` reads one frame of audio samples from the selected audio input device and returns the number of samples by which the audio device reader's queue was overrun since the last call to `record`.

When you call the `record` method of an `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. Call the `release` method of the `audioDeviceReader` System object to release the audio device.

Note: The System object performs an internal initialization the first time you execute `record`. This initialization locks nontunable properties and input specifications, such as the dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

Introduced in R2016a

step

System object: audioDeviceReader

Stream audio data from input device

Syntax

```
[x,numOverrun] = step(aDR)
```

Description

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj)` and `y = obj()` perform equivalent operations.

`[x,numOverrun] = step(aDR)` reads one frame of audio samples from the selected audio input device and returns the number of samples by which the audio device reader's queue was overrun since the last call to `step`.

When you call the `step` method of an `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. Call the `release` method of the `audioDeviceReader` System object to release the audio device.

Note: The System object performs an internal initialization the first time you execute `step`. This initialization locks nontunable properties and input specifications, such as the dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

Introduced in R2016a

audioDeviceWriter System object

Play to sound card

Description

The `audioDeviceWriter` System object writes audio samples to an audio output device. See “Audio Device Writer System Interaction” on page 3-40 for a visualization of how the `audioDeviceWriter` System object plays audio samples.

To stream data to an audio device:

1. Define and set up your audio device writer. See “Construction” on page 3-31.
2. Call step or play to stream data to an audio device.

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`aDW = audioDeviceWriter` returns a System object, `aDW` that writes audio samples to an audio output device in real time.

`aDW = audioDeviceWriter(sampleRateValue)` sets the `SampleRate` property to `sampleRateValue`.

`aDW = audioDeviceWriter(Name,Value)` sets each property `Name` to the specified `Value`. Unspecified properties have default values.

Example: `aDW = audioDeviceWriter(12000,'BitDepth','8-bit integer')` creates a System object, `aDW`, with the `SampleRate` property set to `12000` and the `BitDepth` property set to `'8-bit integer'`.

Properties

`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'`, open the ASIO UI outside of MATLAB to set the sound card buffer size to the `BufferSize` value of your `audioDeviceWriter` System object. 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 `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 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) | character vector

Device used to play audio samples, specified as a character vector. Use the `getAudioDevices` method to list available devices.

`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.

`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. 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 install Audio System Toolbox. If the toolbox is not installed, specifying a nondefault `BitDepth` returns an error.

`SupportVariableSizeInput` — Option to 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 to `step` or `play`. 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 to `step` or `play`. The buffer size of your audio device is specified through the `BufferSize` property.

`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.

This property is available when you set `SupportVariableSizeInput` to `true`.

`ChannelMappingSource` — Source of mapping between columns of input matrix and channels of output device
`'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.

`ChannelMapping` — Nondefault mapping between columns of input matrix and channels of output device

`[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 3-38 example for more information.

This property is available when you set `ChannelMappingSource` to `'Property'`.

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 a nondefault `ChannelMapping` returns an error.

Methods

getAudioDevices	List available audio input devices
info	Get information about selected device
play	Stream audio data to output device
step	Stream audio data to output device

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

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
```

```
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 a 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: 70656.
Total seconds underrun: 3.204354e+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:

- Increase 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.
- Decrease the sample rate. Decreasing the sample rate reduces both latency and underrun at the cost of signal resolution.
- Choose an optimal driver and device for your system.

Specify Channel Mapping for `audioDeviceWriter`

Specify non-default 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 max number of channels available, corresponding to the columns of the input matrix. Use `info` to get the max 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 one corresponds to the sine tone and column two 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];
```

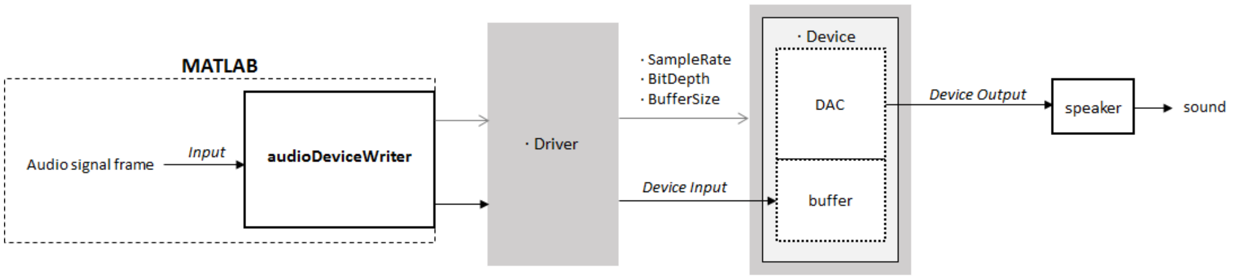
Play your audio signals with reversed mapping. If you are using headphones, notice that the tone and static have switched speakers.

```
count = 0;
while count < 500
    sine = sineGenerator();
    static = randn(length(sine),1);
    deviceWriter([sine,static]);
    count = count + 1;
end
```

More About

Audio Device Writer System Interaction

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.

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 .

See Also

See Also

Blocks
`Audio Device Writer`

System Objects
audioDeviceReader | audioPlayerRecorder | dsp.AudioFileWriter | dsp.AudioFileReader

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

getAudioDevices

System object: audioDeviceWriter

List available audio input devices

Syntax

```
devices = getAudioDevices(aDW)
```

Description

`devices = getAudioDevices(aDW)` returns a cell array listing available audio output devices. The list of available output devices depends on the specified `Driver` property of your `audioDeviceWriter` object.

Introduced in R2016a

info

System object: audioDeviceWriter

Get information about selected device

Syntax

```
aDWInfo = info(aDW)
```

Description

`aDWInfo = info(aDW)` returns a structure containing information about your `audioDeviceWriter` System object. The structure contains information about the driver, device, and maximum number of input channels for your `audioDeviceWriter` System object.

Introduced in R2016a

play

System object: audioDeviceWriter

Stream audio data to output device

Syntax

```
numUnderrun = play(aDW,x)
```

Description

`numUnderrun = play(aDW,x)` writes one frame of audio samples, `x`, to the audio output device specified by the audio device writer System object, `aDW`. The number of samples underrun since the last call to `play` is returned.

If `x` 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.

When you call the `play` method of an `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. Call the `release` method of the `audioDeviceWriter` System object to release the audio device.

Note: The System object performs an internal initialization the first time you execute `play`. This initialization locks nontunable properties and input specifications, such as the dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

Introduced in R2016a

step

System object: audioDeviceWriter

Stream audio data to output device

Syntax

```
numUnderrun = step(aDW,x)
```

Description

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`numUnderrun = step(aDW,x)` writes one frame of audio samples, `x`, to the audio output device specified by the audio device writer System object, `aDW`. The number of samples underrun since the last call to `step` is returned.

If `x` 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.

When you call the `step` method of an `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. Call the `release` method of the `audioDeviceWriter` System object to release the audio device.

Note: The System object performs an internal initialization the first time you execute `step`. This initialization locks nontunable properties and input specifications, such as the dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

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. Define and set up your audio oscillator. See "Construction" on page 3-47.
2. Call step to generate a waveform according to the properties of your `audioOscillator` object. The object has internal memory suited to frame-based processing.

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`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 sine waveforms with `Frequency` set to `8000` and `DCOffset` set to 1.

Properties

If a property is listed as tunable, then you can change its value even when the object is locked.

`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

`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

`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 DC offset is applied.

Tunable: Yes

`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 to 1. The range is a normalized 2π radians 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

`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

`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, specified as a positive integer. This property applies when you set the `SignalType` property to `'sine'`.

Individual tones are generated based on values specified by `Frequency`, `Amplitude`, `PhaseOffset`, and `DCOffset`.

Tunable: No

`DutyCycle` — Square waveform duty cycle

`0.5` (default) | scalar in the range 0 to 1

Square waveform duty cycle, specified as a scalar in the range 0 to 1. This property applies when you set the `SignalType` property to `'square'`.

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

`Width` — Sawtooth width

`1` (default) | real positive scalar

Sawtooth width, specified as a scalar in the range 0 to 1. This property applies when you set the `SignalType` property to `'sawtooth'`.

Sawtooth width determines the point in a sawtooth waveform period at which the maximum occurs.

Tunable: Yes

`SamplesPerFrame` — Number of samples per frame

`512` (default) | positive integer

Number of samples per frame, specified as a positive integer in the range 1 to 192,000.

This property determines the vector length that the `step` method of your `audioOscillator` object outputs.

Tunable: Yes

`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

`OutputDataType` — Data type of generated waveform

`'double'` (default) | `'single'`

Data type of generated waveform, specify as `'double'` or `'single'`.

Tunable: No

Methods

configureMIDI	Configure MIDI connections between System object and MIDI controller
createAudioPluginClass	Create audio plugin class that implements functionality of System object
disconnectMIDI	Disconnect MIDI controls from System object
getMIDIConnections	Get MIDI connection information
reset	Reset internal states of System object
step	Generate tunable waveforms

Common to All System Objects	
`clone`	Create System object with same property values
`getNumInpu`	Expected number of inputs to a System object
`getNumOutp`	Expected number of outputs of a System object
`isLocked`	Check locked states of a System object (logical)
`release`	Allow System object property value changes

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.

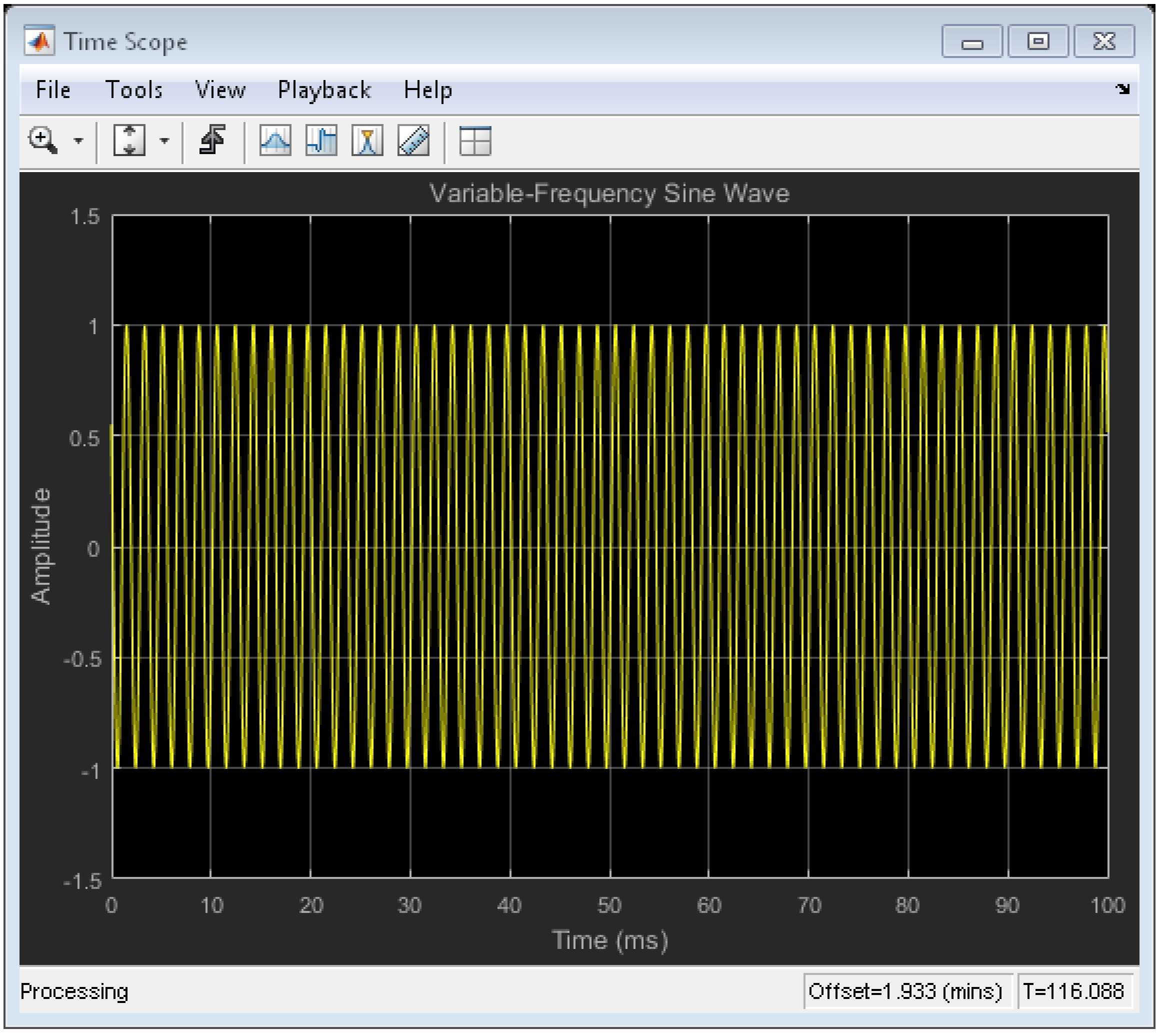
```
osc = audioOscillator;
```

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 sinewave 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 an audio oscillator and audio device writer 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` 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

See Also

System Objects
wavetableSynthesizer

Introduced in R2016a

configureMIDI

System object: audioOscillator

Configure MIDI connections between System object and MIDI controller

Syntax

```
configureMIDI(osc)
configureMIDI(osc,propName)
configureMIDI(osc,propName,controlNumber)
configureMIDI(osc,propName,controlNumber,'DeviceName',deviceName)
```

Description

`configureMIDI(osc)` starts a MIDI configuration user interface (UI). Use the UI to synchronize tunable properties of the audio oscillator System object, `osc`, to MIDI controls of your choice.

`configureMIDI(osc,propName)` makes the System object property, `propName`, respond to any control on the default MIDI device.

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

`configureMIDI(osc,propName,controlNumber,'DeviceName',deviceName)` makes the property respond to the MIDI control specified by `controlNumber` on the device specified by `deviceName`.

Each tunable property of the `audioOscillator` System object maps to MIDI controls with a specified range.

Property	Range	Mapping
`Frequency`	0.1 Hz to 20 kHz	log
`Amplitude`	0 to 10	linear
`DCOffset`	–10 to 10	linear

Property	Range	Mapping
`DutyCycle` (available when you set `SignalType` to `'square'`)	0 to 1	linear
`Width` (available when you set `SignalType` to `'sawtooth'`)	0 to 1	linear

Introduced in R2016a

createAudioPluginClass

System object: audioOscillator

Create audio plugin class that implements functionality of System object

Syntax

```
createAudioPluginClass(osc)
createAudioPluginClass(osc,pluginName)
```

Description

`createAudioPluginClass(osc)` creates a System object source plugin that implements the functionality of the `audioOscillator` System object, `osc`. The name of the created class is the `audioOscillator` System object variable name followed by `'Plugin'`, for example, `oscPlugin`. By default, the created class outputs a one-channel (column) matrix.

`createAudioPluginClass(osc,pluginName)` specifies the name of your created System object source plugin class.

Example: `createAudioPluginClass(osc,'myOscillator')` creates a System object source plugin with class name `myOscillator`.

Each tunable property of the `audioOscillator` System object maps to a plugin parameter with a default range.

Property	Plugin Parameter Range	Mapping
`Frequency`	0.1 Hz to 20 kHz	log
`Amplitude`	0 to 10	linear
`DCOffset`	–10 to 10	linear
`DutyCycle` (available when you set `SignalType` to `'square'`)	0 to 1	linear

Property	Plugin Parameter Range	Mapping
`Width` (available when you set `SignalType` to `'sawtooth'`)	0 to 1	linear

Introduced in R2016a

disconnectMIDI

System object: audioOscillator

Disconnect MIDI controls from System object

Syntax

```
disconnectMIDI(osc)
```

Description

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

Introduced in R2016a

getMIDIConnections

System object: audioOscillator

Get MIDI connection information

Syntax

```
connectionInfo = getMIDIConnections(osc)
```

Description

`connectionInfo = getMIDIConnections(osc)` returns a structure, `connectionInfo`, containing information about the MIDI connections for your audio oscillator, `osc`. Only those MIDI connections established using `configureMIDI` are returned. The `connectionInfo` structure contains a substructure for each tunable property of `osc` 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).

Introduced in R2016a

reset

System object: audioOscillator

Reset internal states of System object

Syntax

```
reset(osc)
```

Description

`reset(osc)` resets internal states of the audio oscillator, `osc`, to their initial values.

Introduced in R2016a

step

System object: audioOscillator

Generate tunable waveforms

Syntax

```
y = step(osc)
```

Description

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj)` and `y = obj()` perform equivalent operations.

`y = step(osc)` generates a waveform output, `y`. The type of waveform is specified by the algorithm and properties of the System object, `osc`.

Note: The System object performs an internal initialization the first time you execute `step`. This initialization locks nontunable properties and input specifications, such as the dimensions, complexity, and data type. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

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. Define and set up your crossover filter. See "Construction" on page 3-65.
2. Call step to implement a crossover filter on each channel of the input signal according to the properties of your `crossoverFilter` object. The input must be a real-valued, double-precision or single-precision matrix. The `crossoverFilter` object treats each column of the input as an independent channel.

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`crossFilt = crossoverFilter` creates a System object, `crossFilt`, that implements an audio crossover filter.

`crossFilt = crossoverFilter(numCrossoversValue)` sets the `NumCrossovers` property to `numCrossoversValue`.

`crossFilt = crossoverFilter(numCrossoversValue, crossoverFrequenciesValue)` sets the `CrossoverFrequencies` property to `crossoverFrequenciesValue`.

`crossFilt = crossoverFilter(numCrossoversValue, crossoverFrequenciesValue,crossoverSlopesValue)` sets the `CrossoverSlopes` property to `crossoverSlopesValue`.

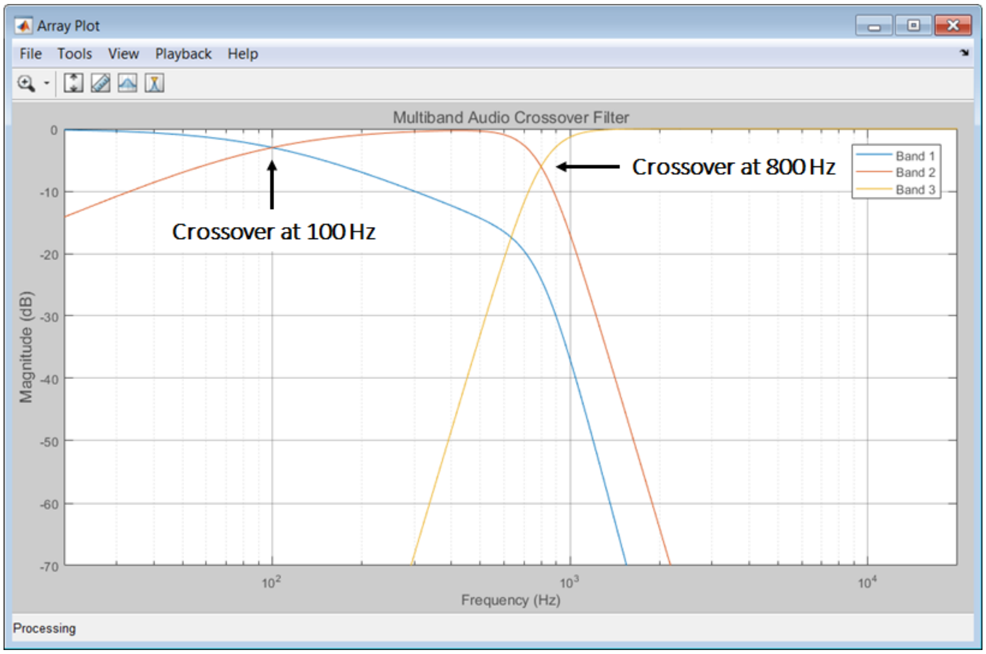
`crossFilt = crossoverFilter(numCrossoversValue, crossoverFrequenciesValue,crossoverSlopesValue,Fs)` sets the `SampleRate` property to `Fs`.

`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 the `NumCrossovers` property set to `2`, the `CrossoverFrequencies` property set to `[100,800]`, and the `CrossoverSlopes` property set to `[6,48]`.

To visualize the crossover bands of the `crossFilt` System object, use the visualize method of the object.

```
visualize(crossFilt)
```



Properties

If a property is listed as tunable, then you can change its value even when the object is locked.

`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 `crossoverFilter` is called by `step` 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

`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

`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 crossover slope is not specified inside the range, it is rounded to the nearest allowed value.

- If `CrossoverSlopes` is a scalar, all two-band component crossovers 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

`SampleRate` — Input sample rate (Hz)
`44100` (default) | positive scalar

Input sample rate in Hz, specified as a positive scalar.

Tunable: Yes

Methods

configureMIDI	Configure MIDI connections between System object and MIDI controller
cost	Implementation cost of System object
createAudioPluginClass	Create audio plugin class that implements functionality of System object
disconnectMIDI	Disconnect MIDI controls from System object
getMIDIConnections	Get MIDI connection information
reset	Reset internal states of System object
step	Implement audio crossover filter
visualize	Visualize magnitude response of System object

Common to All System Objects	
`clone`	Create System object with same property values

Common to All System Objects	
`getNumInpu`	Expected number of inputs to a System object
`getNumOutp`	Expected number of outputs of a System object
`isLocked`	Check locked states of a System object (logical)
`release`	Allow System object property value changes

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.

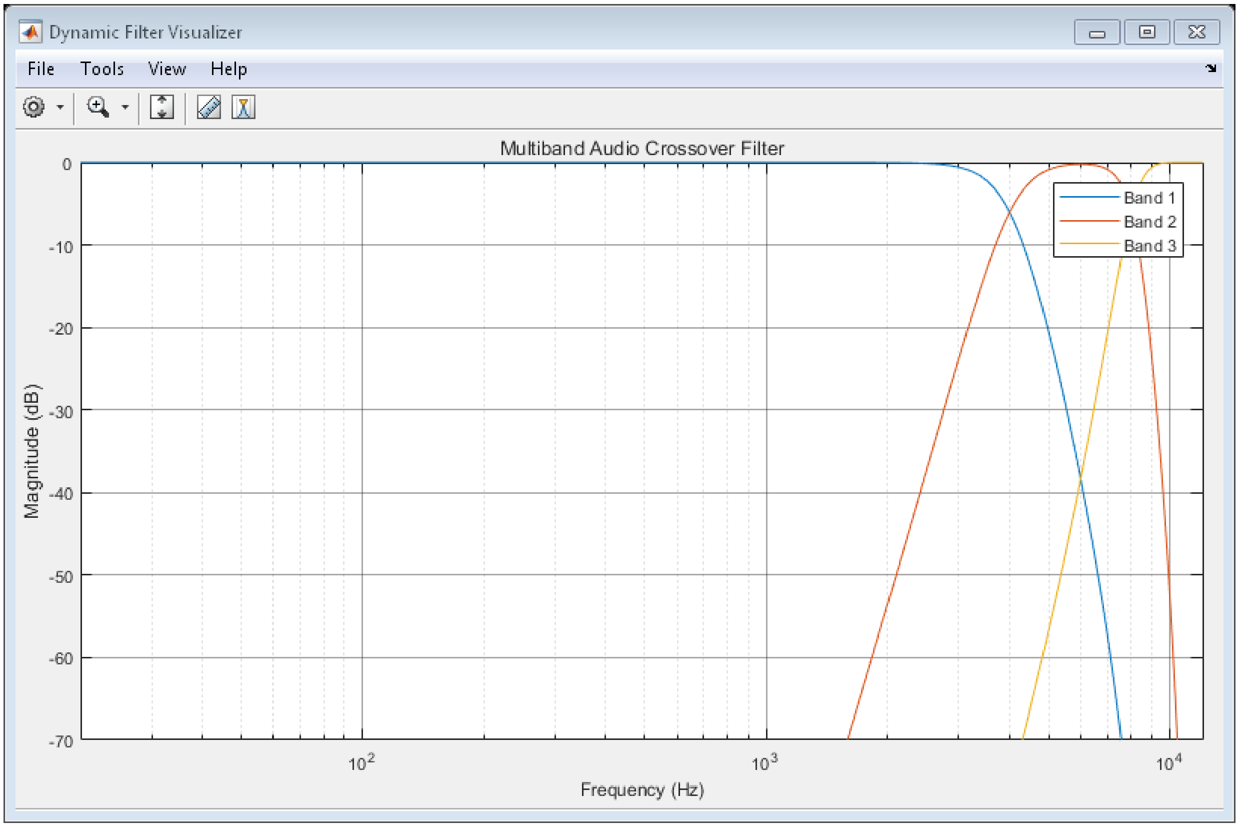
```
noise = randn(12000*5,1);
```

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);
```



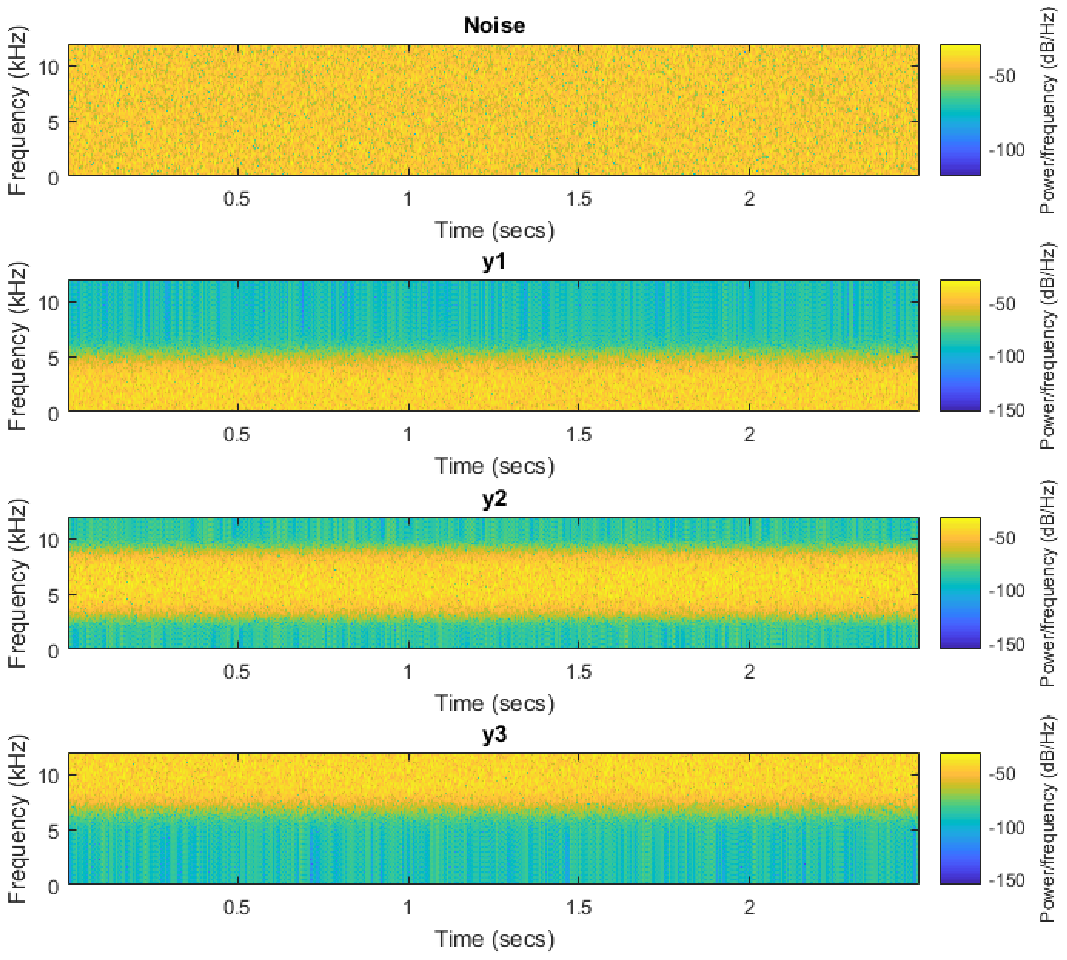
Call your crossover filter like a function with the noise signal as the argument.

```
[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.

Construct the 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 computation load of initialization in an audio stream loop.

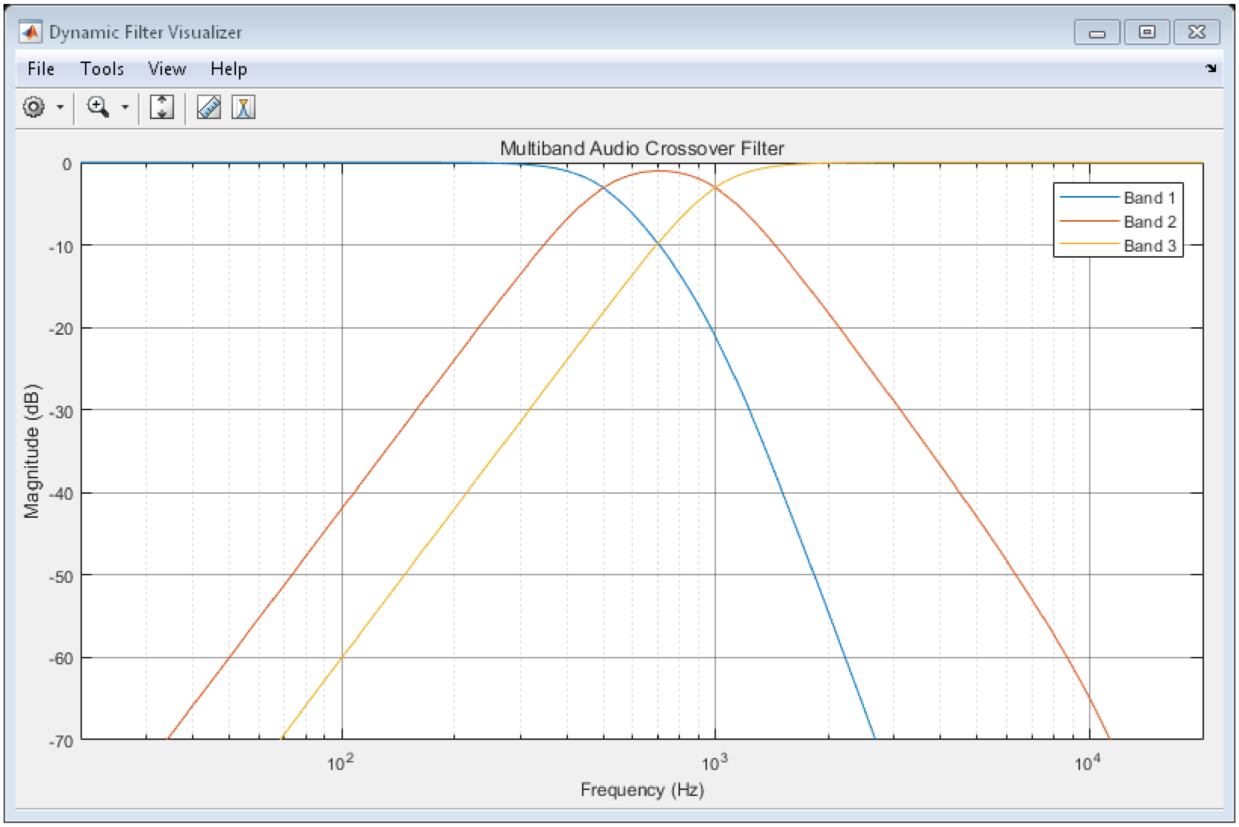
```
samplesPerFrame = 256;

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 crossover filter 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.

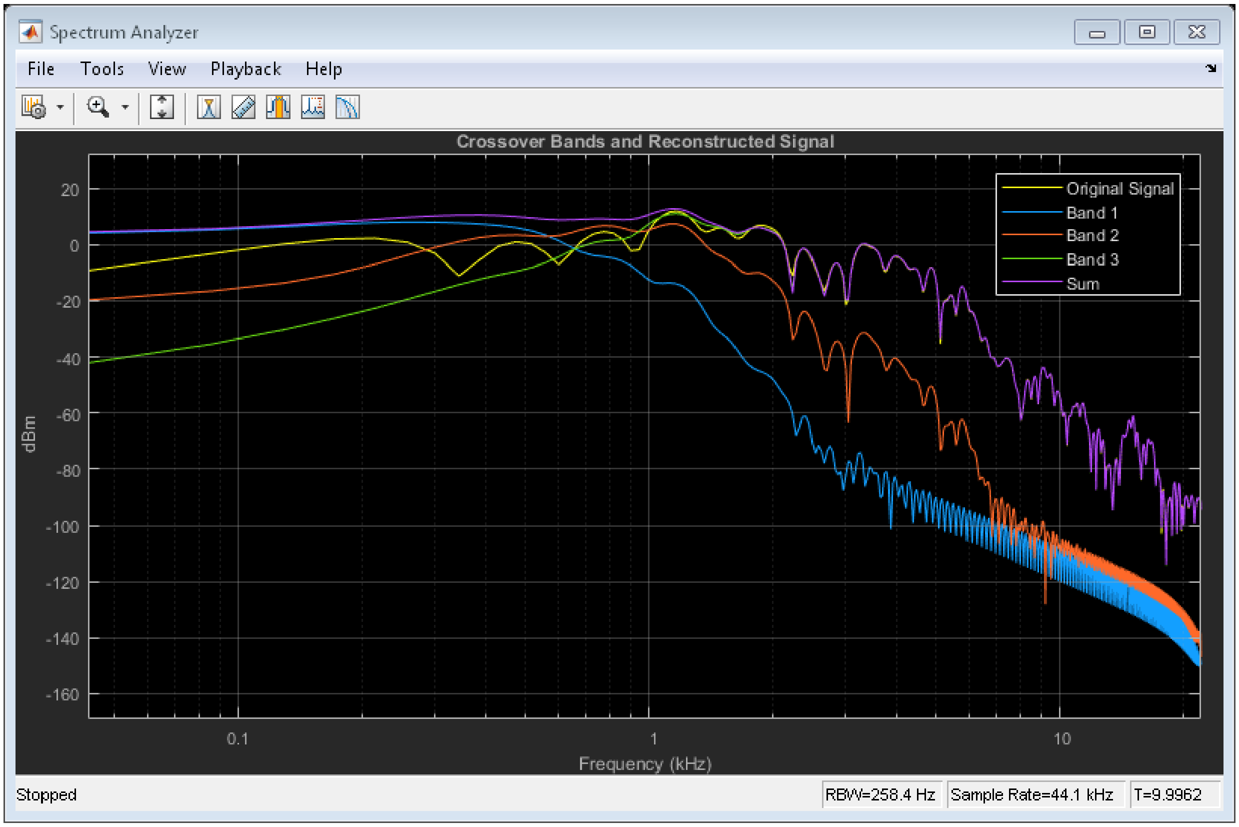
```
cost(crossFilt)


ans = 

  struct with fields:

                  NumCoefficients: 48
                        NumStates: 18
    MultiplicationsPerInputSample: 48
          AdditionsPerInputSample: 37
```

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;
    step(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

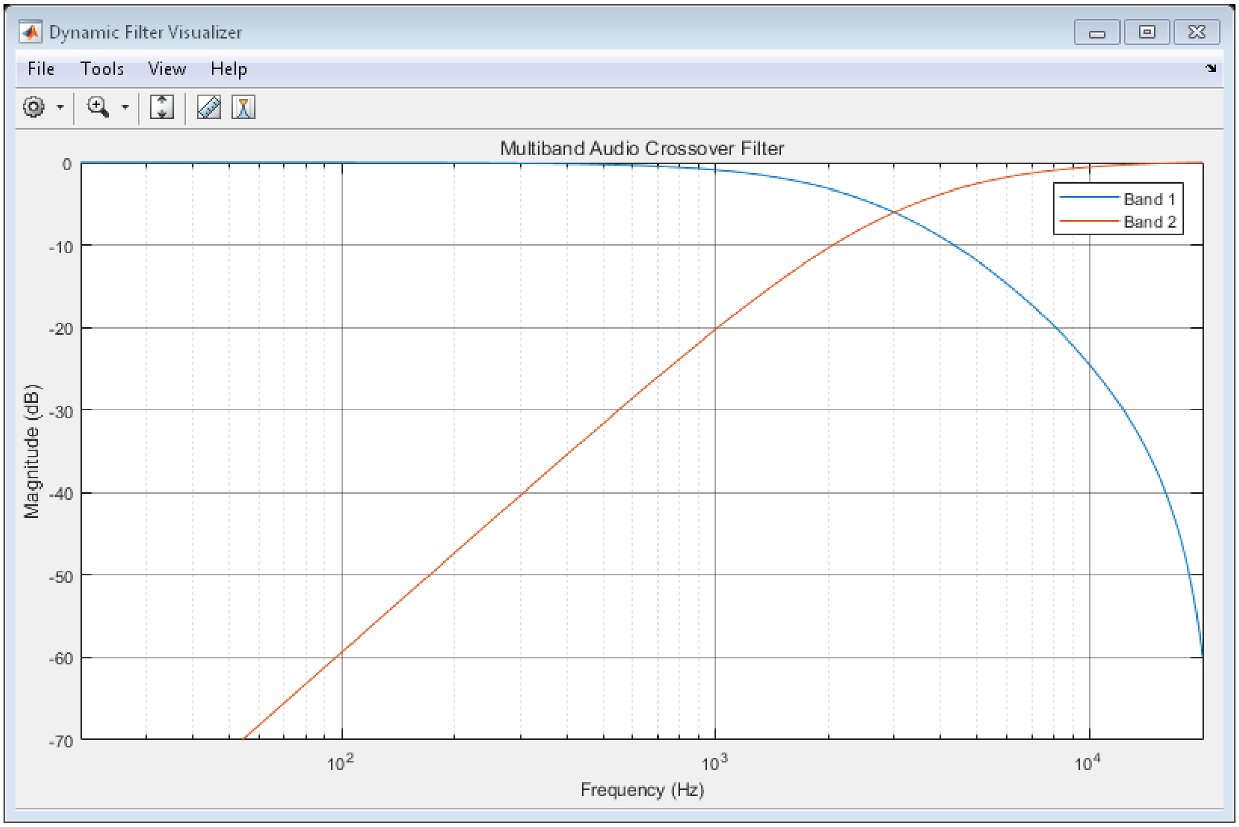
De-essing is the process of diminishing sibilant sounds in an audio signal. Sibilance refers to the *s*, *z*, and *sh* sounds in speech, which can be disproportionately emphasized during recording. *es* sounds fall under the category of unvoiced speech with all consonants, and have a higher frequency 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` object and `audioDeviceWriter` 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` 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'});
```

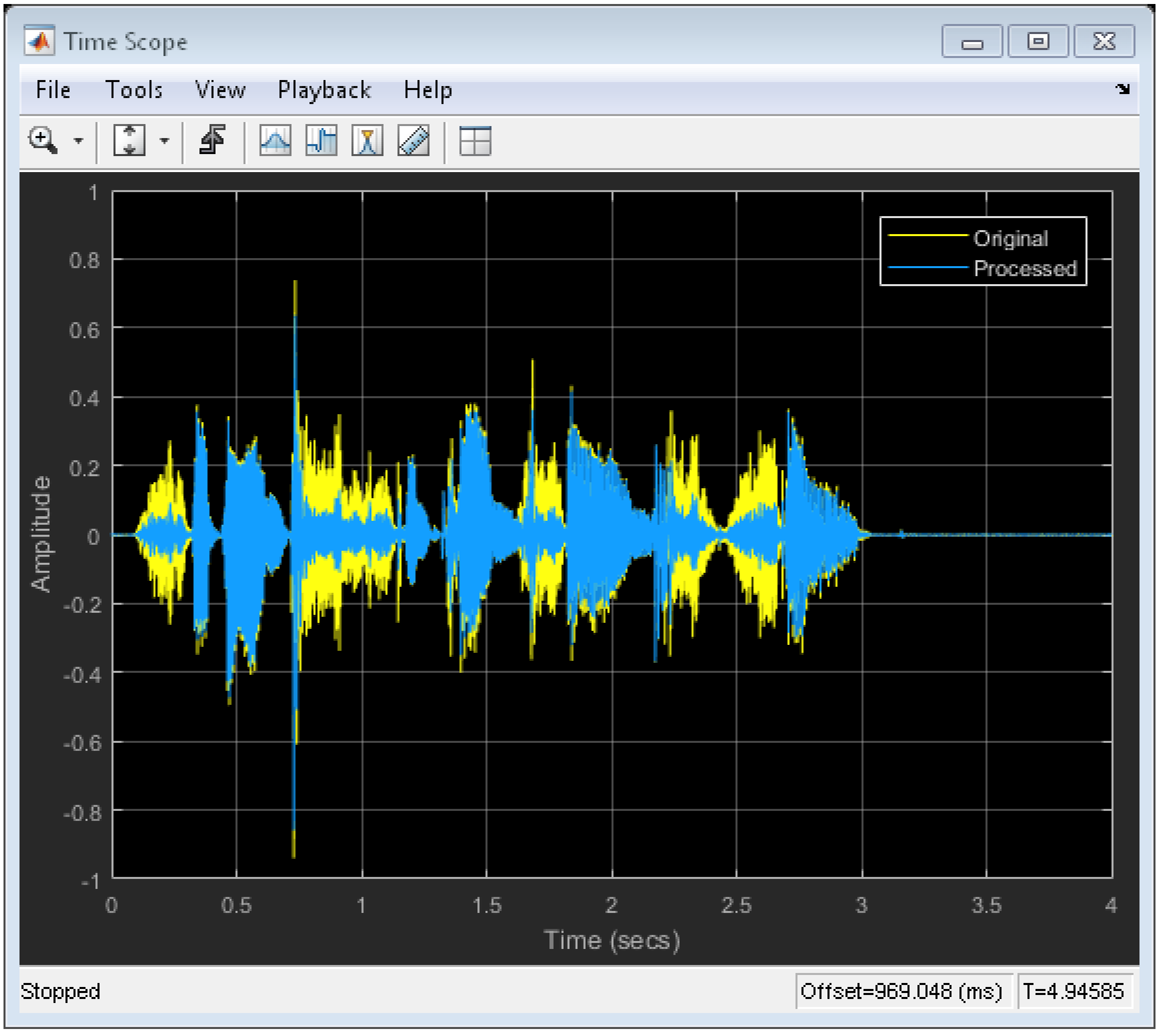
In an audio stream loop:

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 *p*, *d*, and *g* words. 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 `audioDeviceWriter` System object™ to read an audio signal from a file and write an audio signal to a device. Listen to 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.

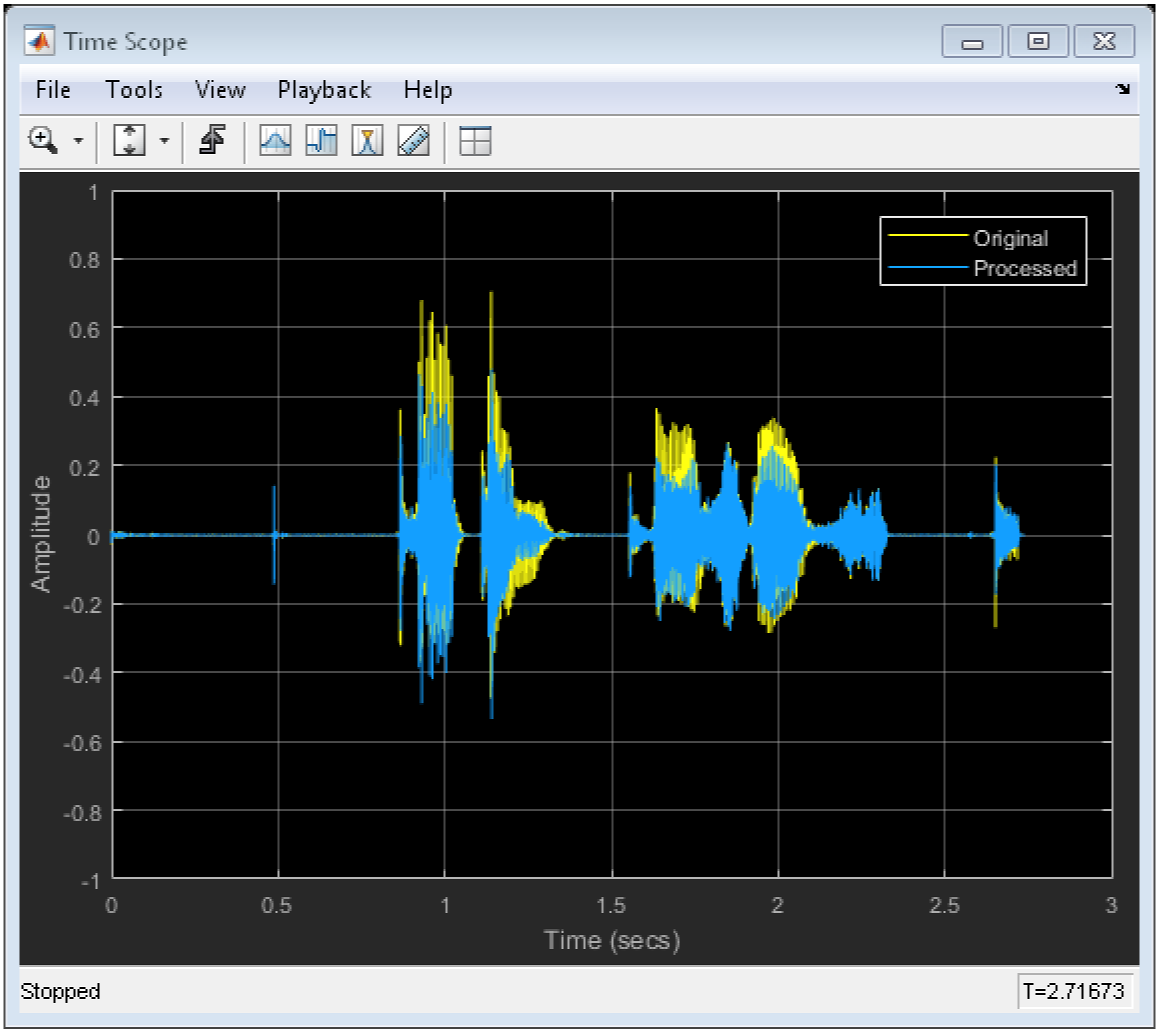
```
[B,A] = designVarSlopeFilter(48,120/(44100/2),'hi');
biquadFilter = dsp.BiquadFilter( ...
    'SOSMatrixSource','Input port', ...
    'ScaleValuesInputPort',false);

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)
```

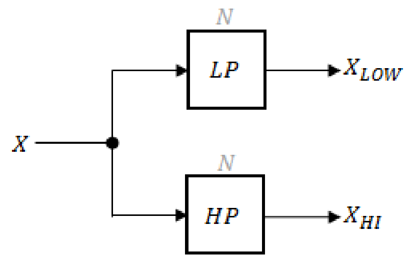


Algorithms

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

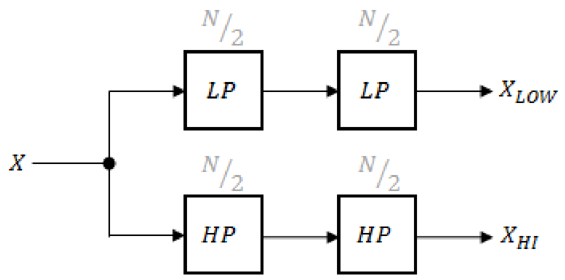
Odd-order two-band (one crossover) filters are implemented as parallel complementary highpass and lowpass filters.



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.

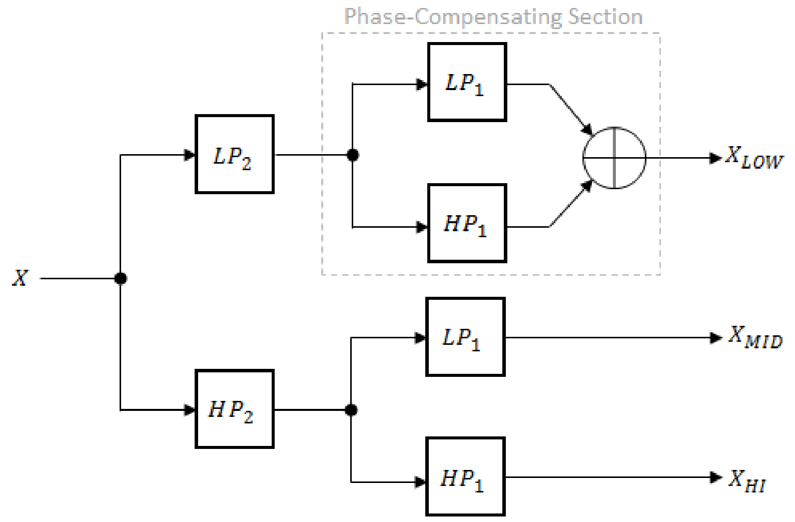


LP and *HP* are Butterworth filters of order *N/2*, where *N* is the order of the overall filter. The filters are implemented as direct-form II transposed second-order sections.

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, 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

See Also

Blocks
`Crossover Filter`

System Objects
multibandParametricEQ

Introduced in R2016a

configureMIDI

System object: crossoverFilter

Configure MIDI connections between System object and MIDI controller

Syntax

```
configureMIDI(crossFilt)
configureMIDI(crossFilt,propName)
configureMIDI(crossFilt,propName,controlNumber)
configureMIDI(crossFilt,propName,controlNumber,'DeviceName',
deviceName)
```

Description

`configureMIDI(crossFilt)` starts a MIDI configuration user interface (UI). Use the UI to synchronize tunable properties of the crossover filter System object, `crossFilt`, to MIDI controls of your choice.

`configureMIDI(crossFilt,propName)` makes the System object property, `propName`, respond to any control on the default MIDI device.

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

`configureMIDI(crossFilt,propName,controlNumber,'DeviceName', deviceName)` makes the property respond to the MIDI control specified by `controlNumber` on the device specified by `deviceName`.

Each tunable property of the `crossoverFilter` System object maps to MIDI controls with a specified range.

Property	Range	Unit
`CrossoverFrequencies`	20 to 20,000	Hz
`CrossoverSlopes`	6 to 48	dB/octave

Introduced in R2016a

cost

System object: crossoverFilter

Implementation cost of System object

Syntax

`C = cost(crossFilt)`

Description

`C = cost(crossFilt)` returns a structure, C, whose fields contain information about the computation cost of implementing the crossover filter, `crossFilt`.

Structure Field	Description
`NumCoefficients`	Number of filter coefficients (excluding coefficients with values 0, 1, or –1)
`NumStates`	Number of states
`MultiplicationsPerInputSample`	Number of multiplications per input sample
`AdditionsPerInputSample`	Number of additions per input sample

Introduced in R2016a

createAudioPluginClass

System object: crossoverFilter

Create audio plugin class that implements functionality of System object

Syntax

```
createAudioPluginClass(crossFilt)
createAudioPluginClass(crossFilt,pluginName)
```

Description

`createAudioPluginClass(crossFilt)` creates a System object plugin that implements the functionality of the `crossoverFilter` System object, `crossFilt`. The name of the created class is the `crossoverFilter` System object variable name followed by `'Plugin'`, for example, `crossFiltPlugin`.

Note: 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(crossFilt,pluginName)` specifies the name of your created System object plugin class.

Example: `createAudioPluginClass(crossFilt,'xOverFilter')` creates a System object plugin with class name `xOverFilter`.

Each tunable property of the `crossoverFilter` System object maps to a plugin parameter with a default range.

Property	Plugin Parameter Range	Unit
`CrossoverFrequencies`	20 to 20,000	Hz
`CrossoverSlopes`	6 to 48	dB/octave

Introduced in R2016a

disconnectMIDI

System object: crossoverFilter

Disconnect MIDI controls from System object

Syntax

```
disconnectMIDI(crossFilt)
```

Description

`disconnectMIDI(crossFilt)` disconnects MIDI controls from your crossover filter, `crossFilt`. Only those MIDI connections established using `configureMIDI` are disconnected.

Introduced in R2016a

getMIDIConnections

System object: crossoverFilter

Get MIDI connection information

Syntax

```
connectionInfo= getMIDIConnections(crossFilt)
```

Description

`connectionInfo= getMIDIConnections(crossFilt)` returns a structure, `connectionInfo`, containing information about the MIDI connections for your crossover filter, `crossFilt`. Only those MIDI connections established using `configureMIDI` are returned. The `connectionInfo` structure contains a substructure for each tunable property of `crossFilt` 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).

Introduced in R2016a

reset

System object: crossoverFilter

Reset internal states of System object

Syntax

`reset(crossFilter)`

Description

`reset(crossFilter)` resets internal states of the crossover filter, `crossFilt`, to their initial values.

Introduced in R2016a

step

System object: crossoverFilter

Implement audio crossover filter

Syntax

```
[band1,band2,...,bandN] = step(crossFilt,x)
```

Description

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`[band1,band2,...,bandN] = step(crossFilt,x)` applies a crossover filter on the input, x, and returns the filtered output bands, `[band1,band2,...,bandN]`, where `N = NumCrossovers + 1`.

`x` must be a real-valued, double-precision or single-precision matrix. The System object treats each column of the input as an independent channel.

Note: The System object performs an internal initialization the first time you execute `step`. This initialization locks nontunable properties and input specifications, such as the dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

Introduced in R2016a

visualize

System object: crossoverFilter

Visualize magnitude response of System object

Syntax

```
visualize(crossFilt)
visualize(crossFilt,N)
```

Description

`visualize(crossFilt)` plots the magnitude response of each individual filter band. The plot is updated automatically when properties of the object change.

`visualize(crossFilt,N)` specifies an `N`-point FFT used to calculate the magnitude response. The default is `2048`.

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 on your input:

1. Define and set up your dynamic range compressor. See "Construction" on page 3-97.
2. Call step to perform dynamic range compression on each channel of the input signal according to the properties of your `compressor` object. The input must be a real-valued, double-precision or single-precision matrix. The `compressor` object treats each column of the input as an independent channel.

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`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 the `AttackTime` property set to `0.01` and the `SampleRate` property set to `16000`.

Properties

If a property is listed as tunable, then you can change its value even when the object is locked.

`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

`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 > `thresholdValue`, 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

`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 $(2 \times |x - T|) \le W$.

- 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

`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

`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

`MakeUpGainMode` — Make-up gain mode

`'Auto'` (default) | `'Property'`

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.

`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. This property is available when you set `MakeUpGainMode` to `'Property'`.

Tunable: Yes

`SampleRate` — Input sample rate (Hz)

`44100` (default) | positive scalar

Input sample rate in Hz, specified as a positive scalar.

Tunable: Yes

Methods

configureMIDI	Configure MIDI connections between System object and MIDI controller
createAudioPluginClass	Create audio plugin class that implements functionality of System object
disconnectMIDI	Disconnect MIDI controls from System object

getMIDIConnections	Get MIDI connection information
reset	Reset internal states of System object
step	Perform dynamic range compression
visualize	Visualize static compression characteristics of System object

Common to All System Objects	
clone	Create System object with same property values
getNumInpu	Expected number of inputs to a System object
getNumOutp	Expected number of outputs of a System object
isLocked	Check locked states of a System object (logical)
release	Allow System object property value changes

Examples

Compress Audio Signal

Use dynamic range compression to attenuate the volume of loud sounds.

Set up the audio file reader and audio device writer System objects.

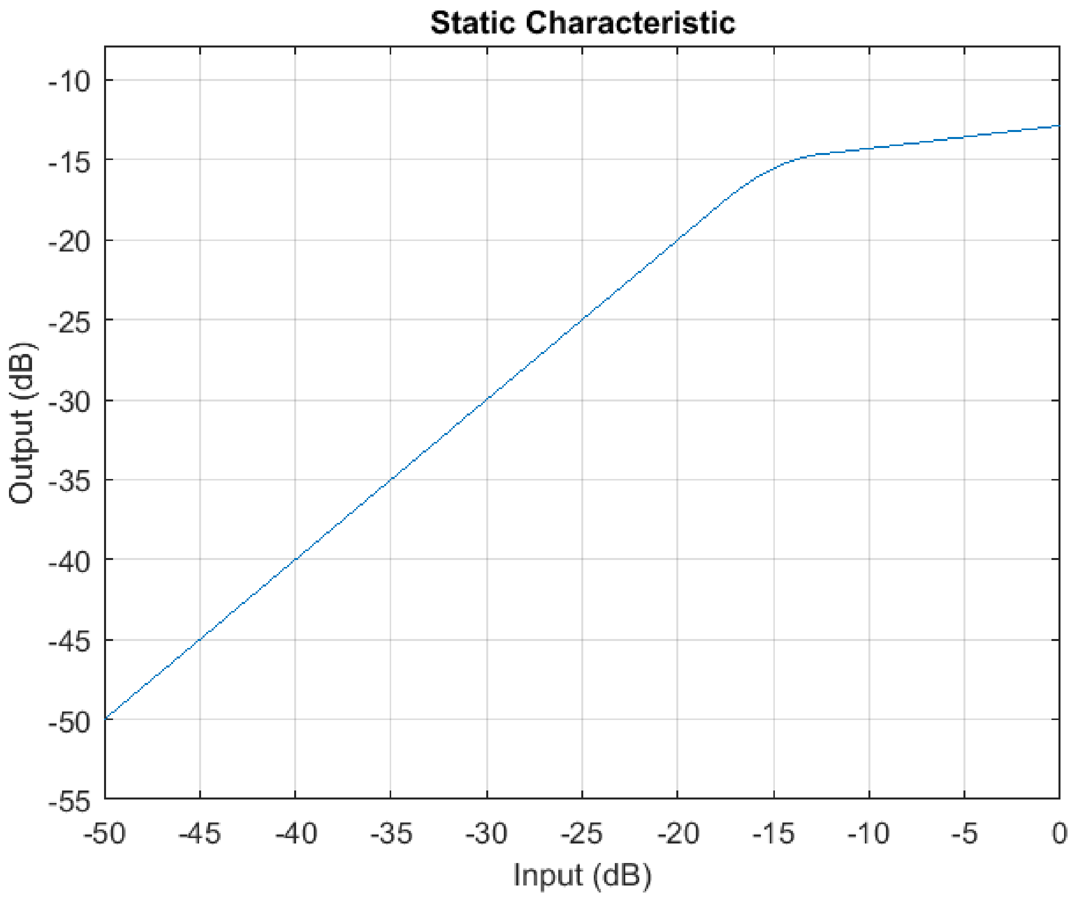
```
frameLength = 1024;
fileReader = dsp.AudioFileReader(...
    'Filename','RockDrums-44p1-stereo-11secs.mp3',...
    'SamplesPerFrame',frameLength);
deviceWriter = audioDeviceWriter(...
    'SampleRate',fileReader.SampleRate);
```

Set up the compressor to have a threshold of -15 dB, a ratio of 7, and a knee width of 5. 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)
```

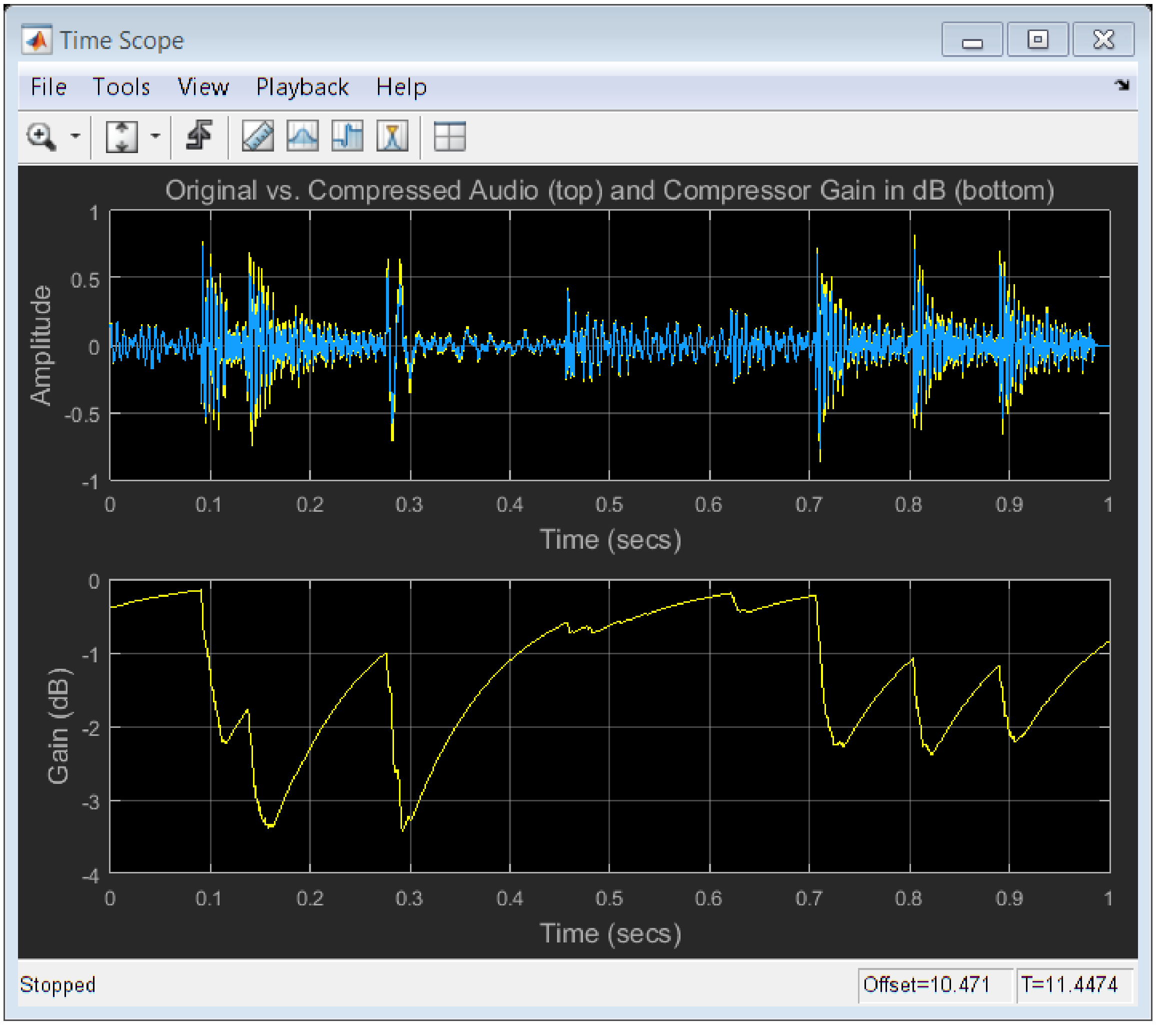


Set up the scope to visualize the original audio signal, the compressed audio signal, and the applied compressor gain.

```
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`.

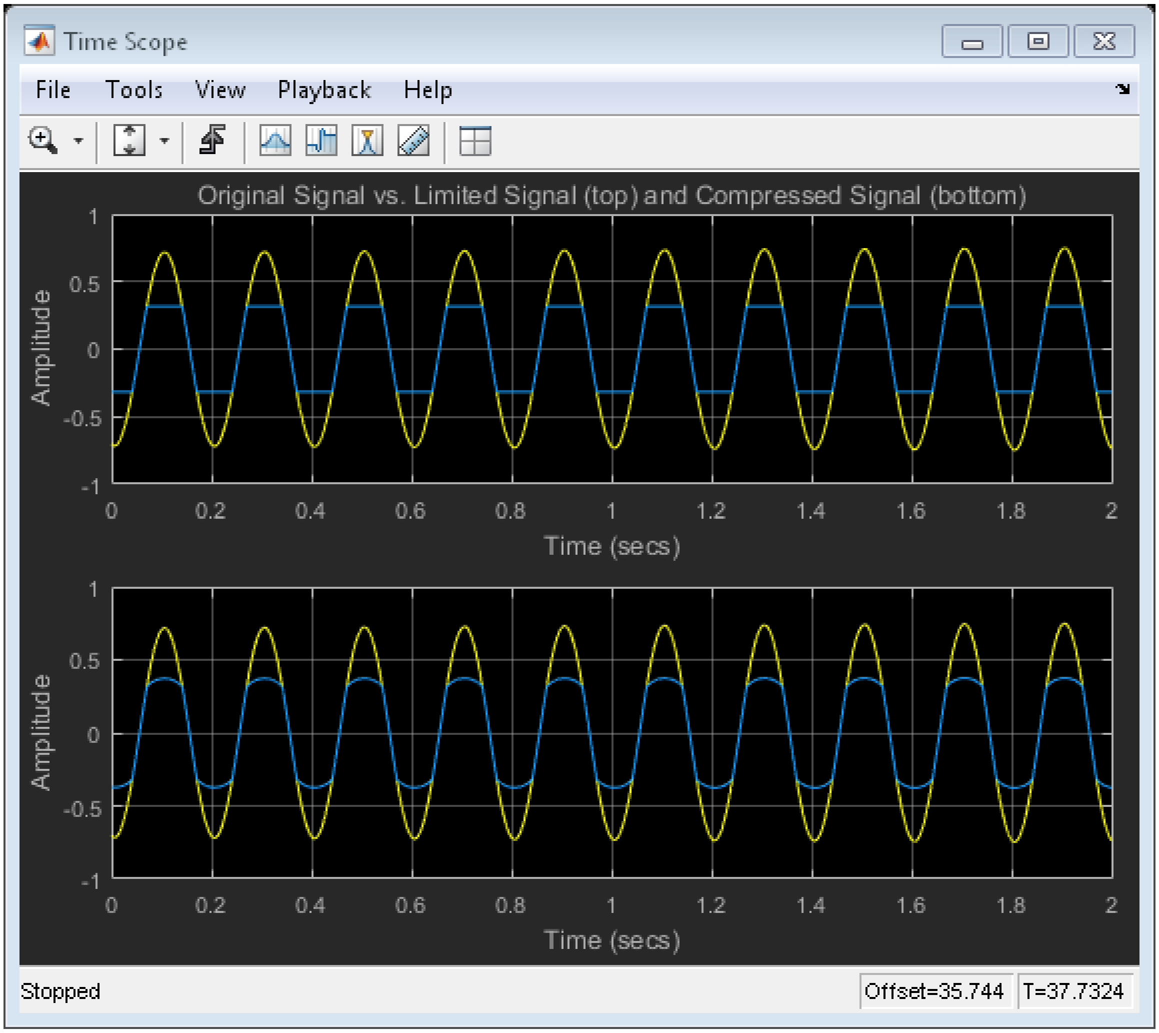
```
dRL = limiter('AttackTime',0,'ReleaseTime',0);
dRC = compressor('AttackTime',0,'ReleaseTime',0);

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` object and `audioDeviceWriter` 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)
```

```
Now playing original signal...
Now playing limited signal...
Now playing compressed signal...
```

Diminish Plosives from Speech Signal

Plosives are consonant sounds resulting from a sudden release of airflow. They are most pronounced in *p*, *d*, and *g* words. 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 `audioDeviceWriter` System object™ to read an audio signal from a file and write an audio signal to a device. Listen to 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);
```

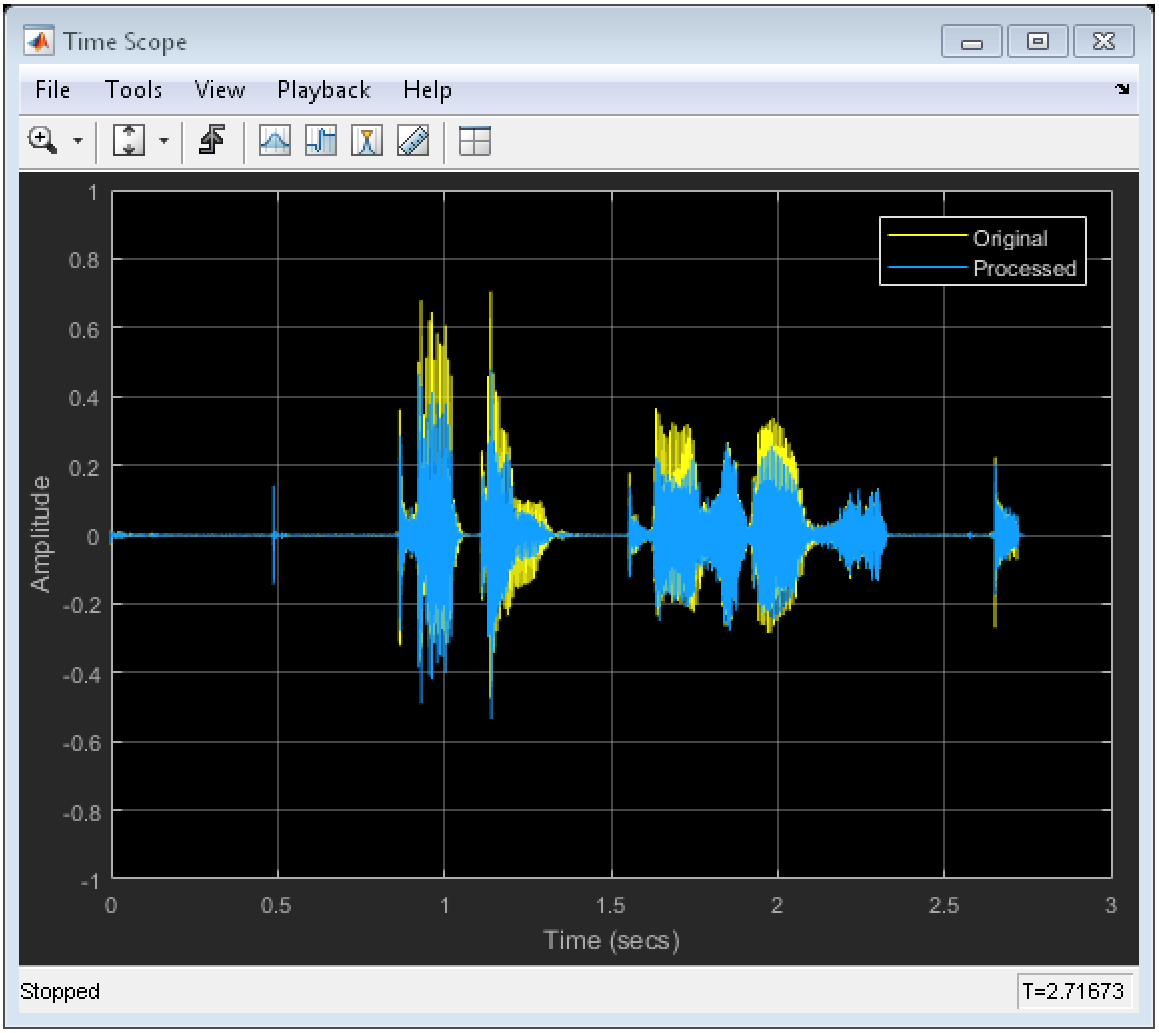
```
scope = dsp.TimeScope( ...
    'SampleRate',fileReader.SampleRate, ...
    'TimeSpan',3, ...
    'BufferLength',fileReader.SampleRate*3*2, ...
    'YLimits',[-1 1], ...
    'ShowGrid',true, ...
    'ShowLegend',true, ...
    'ChannelNames',{'Original','Processed'});
```

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.

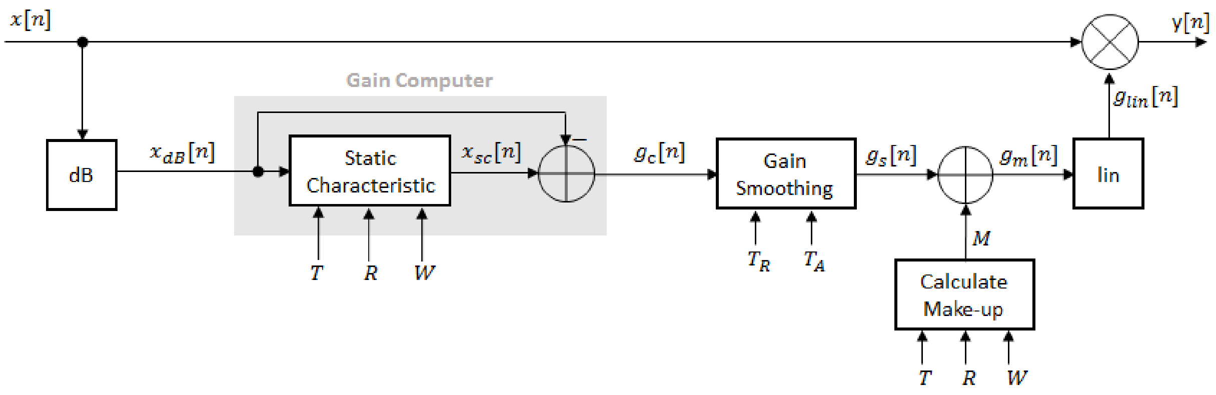
As a best practice, release your objects once done.

```
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.



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 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:

$$x_{sc}(x_{dB}) = \begin{cases} x_{dB} & x_{dB} < T \\ T + \dfrac{(x_{dB} - T)}{R} & x_{dB} \geq T \end{cases}$$

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 and release time properties:

$$g_s[n] = \begin{cases} \alpha_A g_s[n-1] + (1-\alpha_A) g_c[n], & g_c[n] > g_s[n-1] \\ \alpha_R g_s[n-1] + (1-\alpha_R) g_c[n], & g_c[n] \leq g_s[n-1] \end{cases}$$

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.

5 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,

$$M = -x_{sc}(x_{dB} = 0).$$

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.

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 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, 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

See Also

Blocks
`Compressor`

System Objects
expander | noiseGate | limiter

Topics
“Dynamic Range Control”

Introduced in R2016a

configureMIDI

System object: compressor

Configure MIDI connections between System object and MIDI controller

Syntax

```
configureMIDI(dRC)
configureMIDI(dRC,propName)
configureMIDI(dRC,propName,controlNumber)
configureMIDI(dRC,propName,controlNumber,'DeviceName',deviceName)
```

Description

`configureMIDI(dRC)` starts a MIDI configuration user interface (UI). Use the UI to synchronize tunable properties of the dynamic range compressor System object, `dRC`, to MIDI controls of your choice.

`configureMIDI(dRC,propName)` makes the System object property, `propName`, respond to any control on the default MIDI device.

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

`configureMIDI(dRC,propName,controlNumber,'DeviceName',deviceName)` makes the property respond to the MIDI control specified by `controlNumber` on the device specified by `deviceName`.

Each tunable property of the `compressor` System object maps to MIDI controls with a specified range.

Property	Range	Unit
`Threshold`	–50 to 0	dB
`Ratio`	1 to 50	none
`KneeWidth`	0 to 20	dB

Property	Range	Unit
`AttackTime`	0 to 4	seconds
`ReleaseTime`	0 to 4	seconds
`MakeUpGain` (available when you set `MakeUpGainMode` to `'Property'`)	–10 to 24	dB

Introduced in R2016a

createAudioPluginClass

System object: compressor

Create audio plugin class that implements functionality of System object

Syntax

```
createAudioPluginClass(dRC)
createAudioPluginClass(dRC,pluginName)
```

Description

`createAudioPluginClass(dRC)` creates a System object plugin that implements the functionality of the `compressor` System object, `dRC`. The name of the created class is the `compressor` System object variable name followed by `'Plugin'`, for example, `dRCPlugin`.

Note: 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(dRC,pluginName)` specifies the name of your created System object plugin class.

Example: `createAudioPluginClass(dRC,'myCompressor')` creates a System object plugin with class name `myCompressor`.

Each tunable property of the `compressor` System object maps to a plugin parameter with a default range.

Property	Plugin Parameter Range	Unit
`Threshold`	–50 to 0	dB
`Ratio`	1 to 50	none
`KneeWidth`	0 to 20	dB

Property	Plugin Parameter Range	Unit
`AttackTime`	0 to 4	seconds
`ReleaseTime`	0 to 4	seconds
`MakeUpGain` (available when you set `MakeUpGainMode` to `'Property'`)	–10 to 24	dB

Introduced in R2016a

disconnectMIDI

System object: compressor

Disconnect MIDI controls from System object

Syntax

`disconnectMIDI(dRC)`

Description

`disconnectMIDI(dRC)` disconnects MIDI controls from your dynamic range compressor, `dRC`. Only those MIDI connections established using `configureMIDI` are disconnected.

Introduced in R2016a

getMIDIConnections

System object: compressor

Get MIDI connection information

Syntax

```
connectionInfo = getMIDIConnections(dRC)
```

Description

`connectionInfo = getMIDIConnections(dRC)` returns a structure, `connectionInfo`, containing information about the MIDI connections for your dynamic range compressor, `dRC`. Only those MIDI connections established using `configureMIDI` are returned. The `connectionInfo` structure contains a substructure for each tunable property of `dRC` 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).

Introduced in R2016a

reset

System object: compressor

Reset internal states of System object

Syntax

```
reset(dRC)
```

Description

`reset(dRC)` resets internal states of the dynamic range compressor, `dRC`, to their initial values.

Introduced in R2016a

step

System object: compressor

Perform dynamic range compression

Syntax

```
y = step(dRC,x)
[y,g] = step(dRC,x)
```

Description

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`y = step(dRC,x)` performs dynamic range compression on the input signal, `x`, and returns the compressed signal, `y`. The type of dynamic range compression is specified by the algorithm and properties of the `compressor` System object, `dRC`.

`x` must be a real-valued, double-precision or single-precision matrix. The System object treats each column of the input as an independent channel.

Note: The System object performs an internal initialization the first time you execute `step`. This initialization locks nontunable properties and input specifications, such as the dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

`[y,g] = step(dRC,x)` also returns the gain, in dB, applied at each input sample.

Introduced in R2016a

visualize

System object: compressor

Visualize static compression characteristics of System object

Syntax

```
visualize(dRC)
visualize(dRC,myInputRange)
Y = visualize( ___ )
```

Description

`visualize(dRC)` plots the static compression characteristic of the dynamic range compressor, `dRC`. The method computes the dB output level for the input range `[-50:0.01:0]` dB. The plot is updated automatically when properties of the object change.

`visualize(dRC,myInputRange)` enables you to specify the input range in dB. Specify `myInputRange` as a vector of ascending values.

`Y = visualize( ___ )` returns the dB output level, `Y`, corresponding to the input range. You can use any of the input arguments from previous syntaxes.

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 on your input:

1. Define and set up your dynamic range expander. See “Construction” on page 3-125.
2. Call step to perform dynamic range expansion on each channel of the input signal according to the properties of your `expander` object. The input must be a real-valued, double-precision or single-precision matrix. The `expander` object treats each column of the input as an independent channel.

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`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 the `AttackTime` property set to `0.01`, and the `SampleRate` property set to `16000`.

Properties

If a property is listed as tunable, then you can change its value even when the object is locked.

`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

`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

`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 $(2\times|x-T|) \leq W$.

- 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

`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

`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

`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

`SampleRate` — Input sample rate (Hz)

`44100` (default) | positive scalar

Input sample rate in Hz, specified as a positive scalar.

Tunable: Yes

Methods

configureMIDI	Configure MIDI connections between System object and MIDI controller
createAudioPluginClass	Create audio plugin class that implements functionality of System object
disconnectMIDI	Disconnect MIDI controls from System object
getMIDIConnections	Get MIDI connection information
reset	Reset internal states of System object
step	Perform dynamic range expansion

visualize	Visualize static expander characteristics of System object

Common to All System Objects	
`clone`	Create System object with same property values
`getNumInpu`	Expected number of inputs to a System object
`getNumOutp`	Expected number of outputs of a System object
`isLocked`	Check locked states of a System object (logical)
`release`	Allow System object property value changes

Examples

Expand Audio Signal

Use dynamic range expansion to attenuate background noise from an audio signal.

Set up the audio file reader and audio device writer 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 Gausian noise. Play the audio.

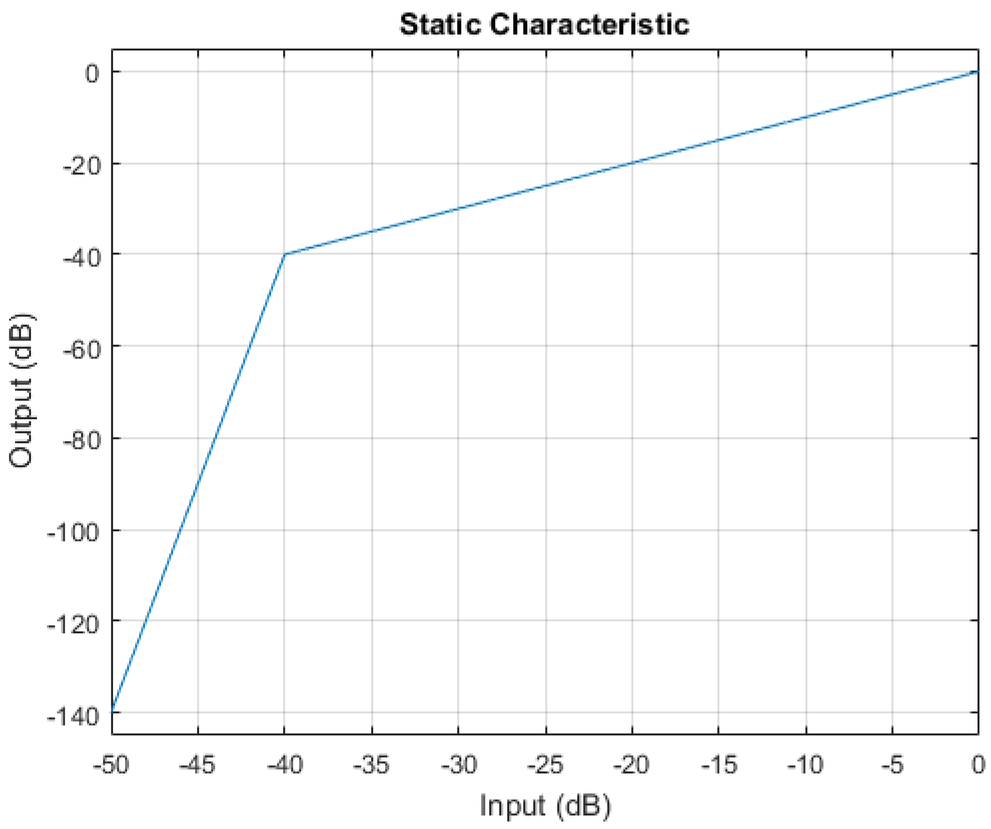
```
while ~isDone(fileReader)
    x = fileReader();
    xCorrupted = x + (1e-2/4)*randn(frameLength,1);
    deviceWriter(xCorrupted);
end

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);
```

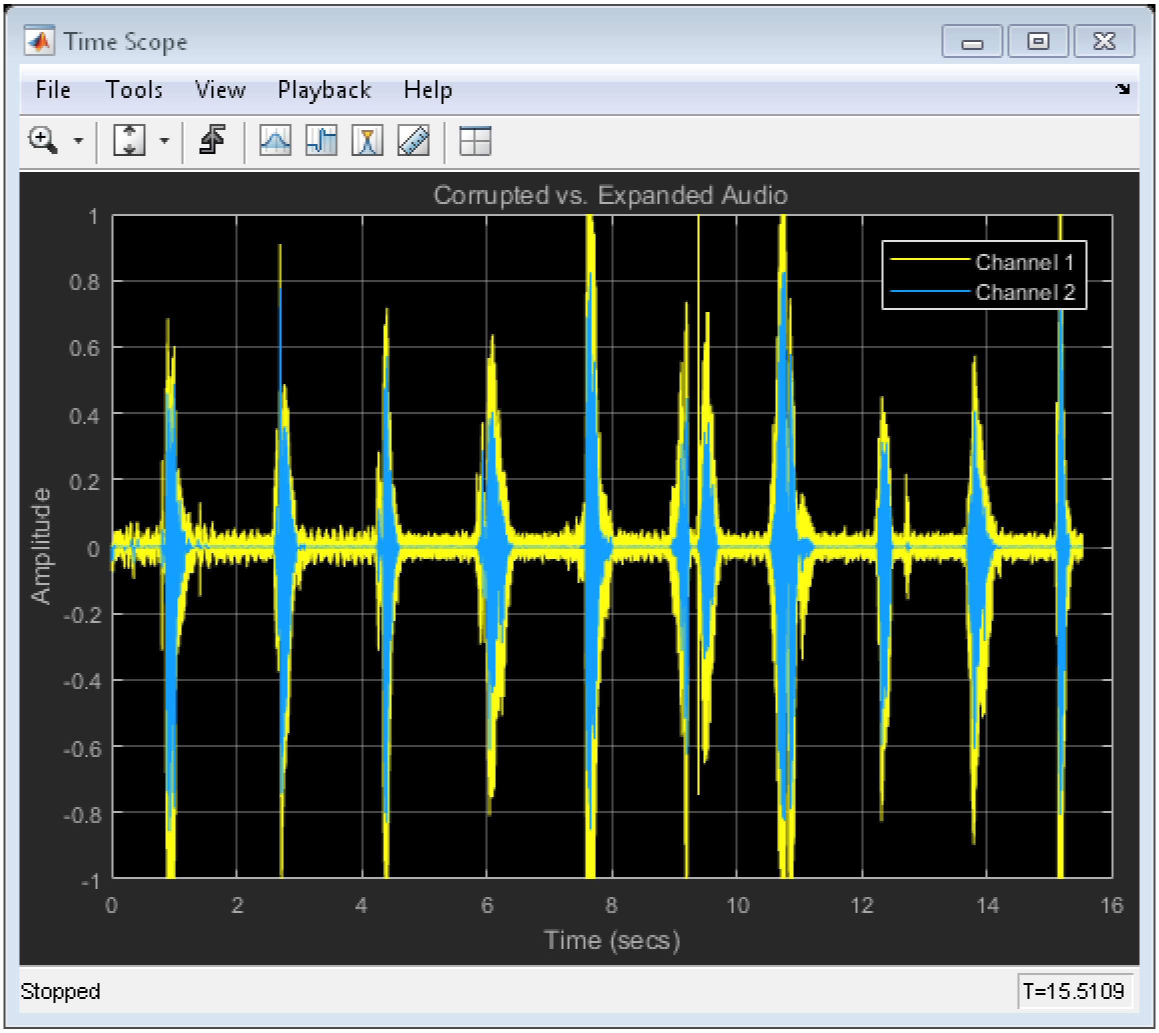


Set up the scope to visualize the signal before and after dynamic range expansion.

```
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

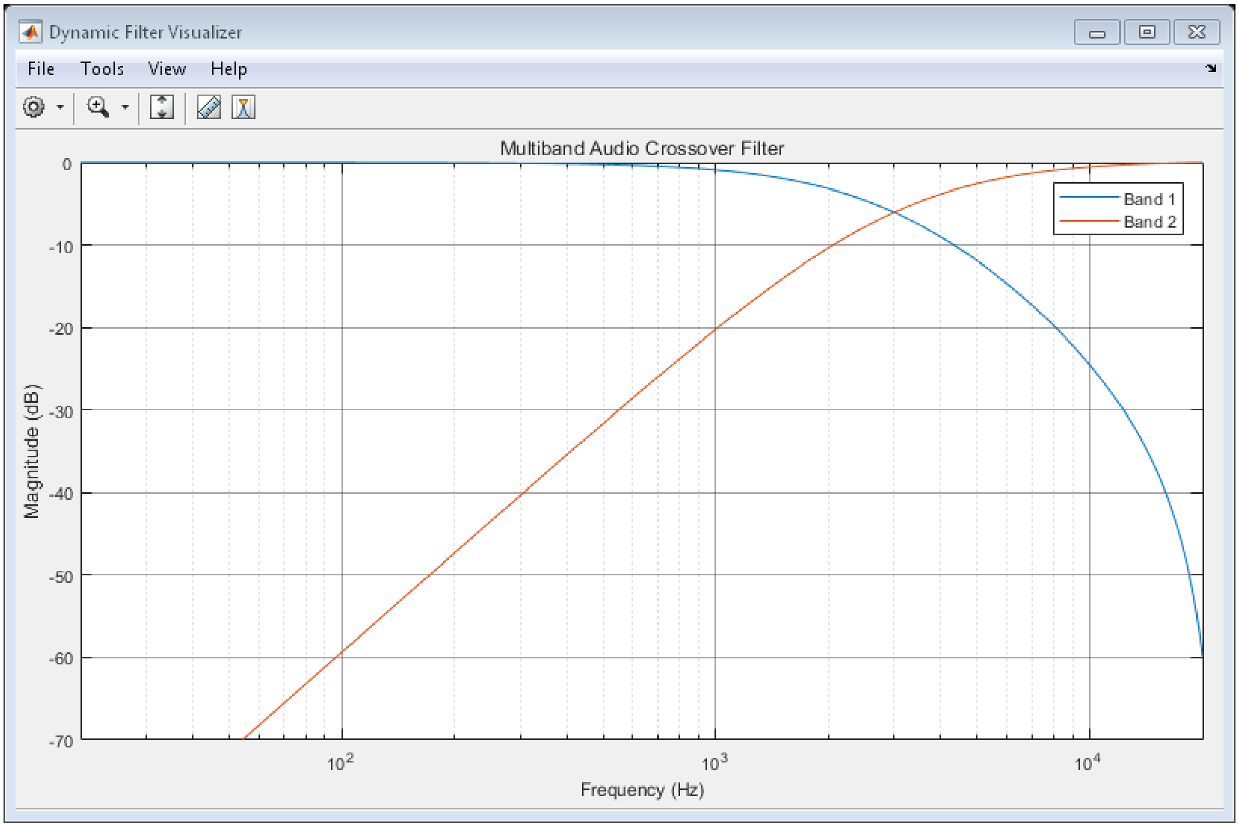
De-essing is the process of diminishing sibilant sounds in an audio signal. Sibilance refers to the *s*, *z*, and *sh* sounds in speech, which can be disproportionately emphasized during recording. *es* sounds fall under the category of unvoiced speech with all consonants, and have a higher frequency 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` object and `audioDeviceWriter` 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` 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'});
```

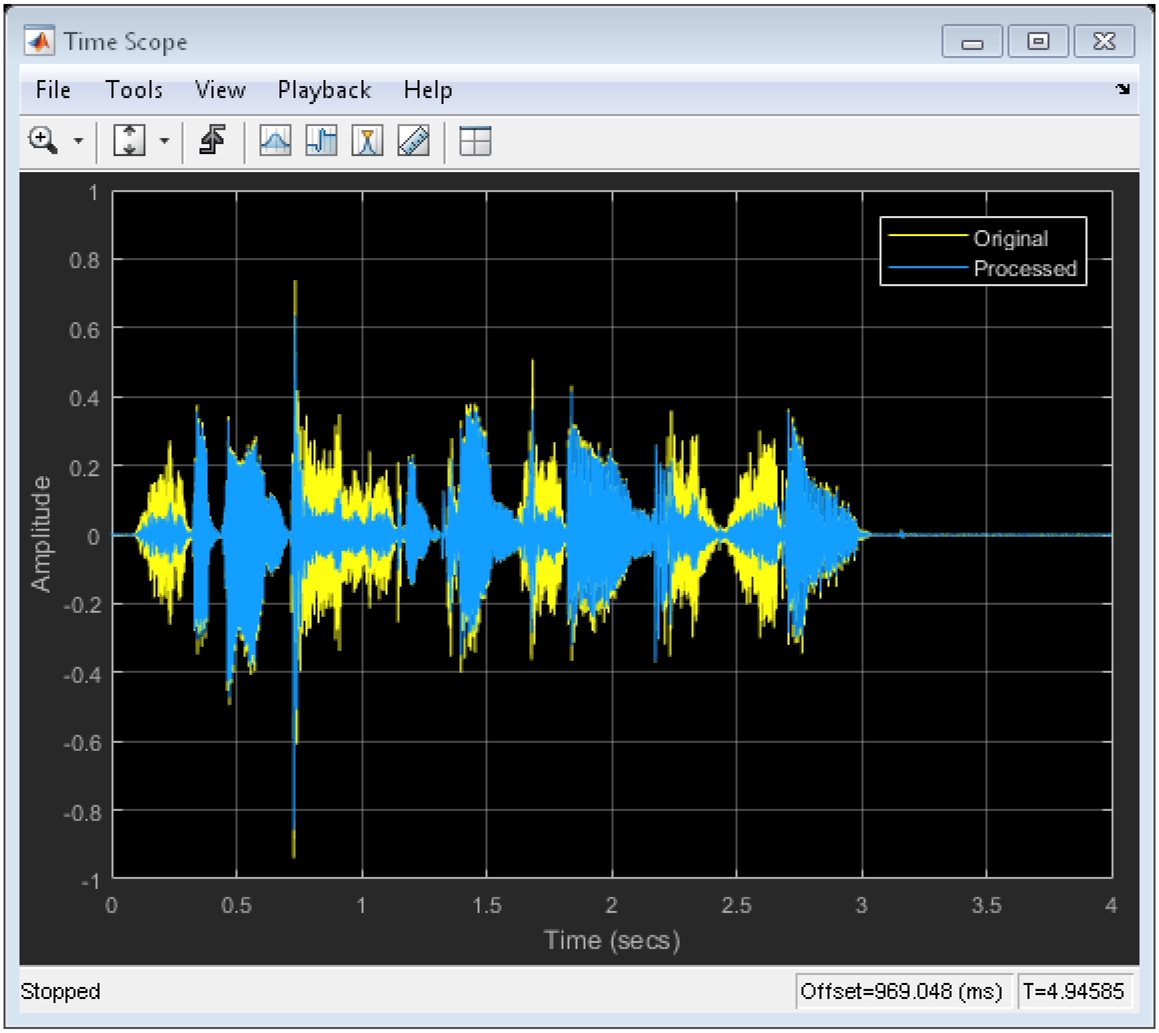
In an audio stream loop:

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.

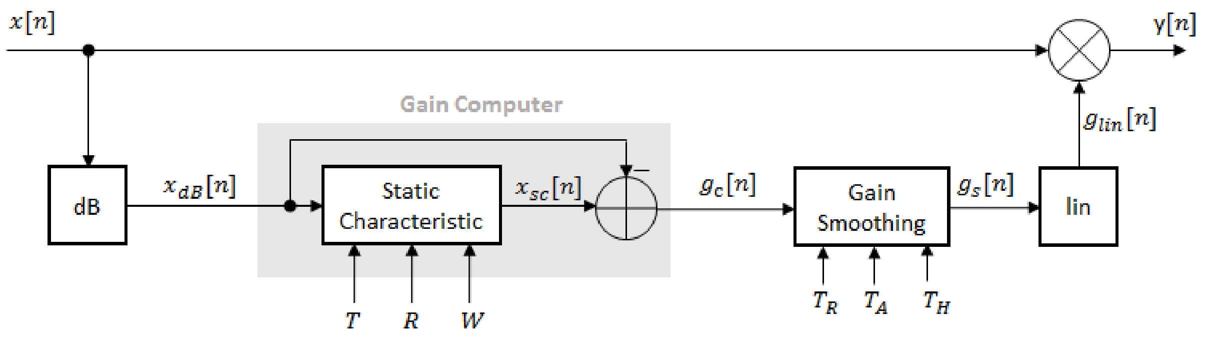
As a best practice, release your objects once done.

```
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.



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:

   $$x_{sc}(x_{dB}) = \begin{cases} T + (x_{dB} - T) \times R & x_{dB} < \left(T - \frac{W}{2}\right) \\ x_{dB} + \dfrac{(1-R)\left(x_{dB} - T - \frac{W}{2}\right)^2}{2W} & \left(T - \frac{W}{2}\right) \le x_{dB} \le \left(T + \frac{W}{2}\right) \\ x_{dB} & x_{dB} > \left(T + \frac{W}{2}\right) \end{cases},$$

   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:

$$x_{sc}(x_{dB}) = \begin{cases} T + (x_{dB} - T) \times R & x_{dB} < T \\ x_{dB} & x_{dB} \geq T \end{cases}$$

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 properties:

$$g_s[n] = \begin{cases} \alpha_A g_s[n-1] + (1-\alpha_A) g_c[n] & (C_A > T_H) \ \& \ (g_c[n] > g_s[n-1]) \\ g_s[n-1] & C_A \leq T_H \\ \alpha_R g_s[n-1] + (1-\alpha_R) g_c[n] & (C_R > T_H) \ \& \ (g_c[n] \leq g_s[n-1]) \\ g_s[n-1] & C_R \leq T_H \end{cases}$$

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.

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

$$y[n] = x[n] \times g_{lin}[n].$$

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, 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

See Also

Blocks
`Expander`

System Objects
noiseGate | compressor | limiter

Topics

“Dynamic Range Control”

Introduced in R2016a

configureMIDI

System object: expander

Configure MIDI connections between System object and MIDI controller

Syntax

```
configureMIDI(dRE)
configureMIDI(dRE,propName)
configureMIDI(dRE,propName,controlNumber)
configureMIDI(dRE,propName,controlNumber,'DeviceName',deviceName)
```

Description

`configureMIDI(dRE)` starts a MIDI configuration user interface (UI). Use the UI to synchronize tunable properties of the dynamic range expander System object, `dRE`, to MIDI controls of your choice.

`configureMIDI(dRE,propName)` makes the System object property, `propName`, respond to any control on the default MIDI device.

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

`configureMIDI(dRE,propName,controlNumber,'DeviceName',deviceName)` makes the property respond to the MIDI control specified by `controlNumber` on the device specified by `deviceName`.

Each tunable property of the `expander` System object maps to MIDI controls with a specified range.

Property	Range	Unit
`Threshold`	–140 to 0	dB
`Ratio`	1 to 50	none
`KneeWidth`	0 to 20	dB

Property	Range	Unit
`AttackTime`	0 to 4	seconds
`ReleaseTime`	0 to 4	seconds
`HoldTime`	0 to 4	seconds

Introduced in R2016a

createAudioPluginClass

System object: expander

Create audio plugin class that implements functionality of System object

Syntax

```
createAudioPluginClass(dRE)
createAudioPluginClass(dRE,pluginName)
```

Description

`createAudioPluginClass(dRE)` creates a System object plugin that implements the functionality of the `expander` System object, `dRE`. The name of the created class is the `expander` System object variable name followed by `'Plugin'`, for example, `dREPlugin`.

Note: 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(dRE,pluginName)` specifies the name of your created System object plugin class.

Example: `createAudioPluginClass(dRE,'myExpander')` creates a System object plugin with class name `myExpander`.

Each tunable property of the `expander` System object maps to a plugin parameter with a default range.

Property	Plugin Parameter Range	Unit
`Threshold`	–140 to 0	dB
`Ratio`	1 to 50	none
`KneeWidth`	0 to 20	dB

Property	Plugin Parameter Range	Unit
`AttackTime`	0 to 4	seconds
`ReleaseTime`	0 to 4	seconds
`HoldTime`	0 to 4	seconds

Introduced in R2016a

disconnectMIDI

System object: expander

Disconnect MIDI controls from System object

Syntax

```
disconnectMIDI(dRE)
```

Description

`disconnectMIDI(dRE)` disconnects MIDI controls from your dynamic range expander, `dRE`. Only those MIDI connections established using `configureMIDI` are disconnected.

Introduced in R2016a

getMIDIConnections

System object: expander

Get MIDI connection information

Syntax

```
connectionInfo = getMIDIConnections(dRE)
```

Description

`connectionInfo = getMIDIConnections(dRE)` returns a structure, `connectionInfo`, containing information about the MIDI connections for your dynamic range expander, `dRE`. Only those MIDI connections established using `configureMIDI` are returned. The `connectionInfo` structure contains a substructure for each tunable property of `dRE` 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).

Introduced in R2016a

reset

System object: expander

Reset internal states of System object

Syntax

```
reset(dRE)
```

Description

`reset(dRE)` resets internal states of the dynamic range expander, `dRE`, to their initial values.

Introduced in R2016a

step

System object: expander

Perform dynamic range expansion

Syntax

```
y = step(dRE,x)
[y,g] = step(dRE,x)
```

Description

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`y = step(dRE,x)` performs dynamic range expansion on the input signal, `x`, and returns the expanded signal, `y`. The type of dynamic range expansion is specified by the algorithm and properties of the `expander` System object, `dRE`.

`x` must be a real-valued, double-precision or single-precision matrix. The System object treats each column of the input as an independent channel.

Note: The System object performs an internal initialization the first time you execute `step`. This initialization locks nontunable properties and input specifications, such as the dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

`[y,g] = step(dRE,x)` also returns the gain, in dB, applied at each input sample.

Introduced in R2016a

visualize

System object: expander

Visualize static expander characteristics of System object

Syntax

```
visualize(dRE)
visualize(dRE,myInputRange)
Y = visualize( ___ )
```

Description

`visualize(dRE)` plots the static expansion characteristic of the dynamic range expander, `dRE`. The method computes the dB output level for the input range `[-20:0.01:0]` dB. The plot is updated automatically when properties of the object change.

`visualize(dRE,myInputRange)` enables you to specify the input range in dB. Specify `myInputRange` as a vector of ascending values.

`Y = visualize( ___ )` returns the dB output level, `Y`, corresponding to the input range. You can use any of the input arguments from previous syntaxes.

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 on your input:

1. Define and set up your dynamic range limiter. See "Construction" on page 3-150.
2. Call step to perform dynamic range limiting on each channel of the input signal according to the properties of your `limiter` object. The input must be a real-valued, double-precision or single-precision matrix. The `limiter` object treats each column of the input as an independent channel.

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`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 `AttackTime` property set to `0.01` and `SampleRate` property set to `16000`.

Properties

If a property is listed as tunable, then you can change its value even when the object is locked.

`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

`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 $\left(2 \times |x - T|\right) \leq W$.

- 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

`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

`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

`MakeUpGainMode` — Make-up gain mode
`'Auto'` (default) | `'Property'`

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.

`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. This property is available when you set `MakeUpGainMode` to `'Property'`.

Tunable: Yes

`SampleRate` — Input sample rate (Hz)
44100 (default) | positive scalar

Input sample rate in Hz, specified as a positive scalar.

Tunable: Yes

Methods

configureMIDI	Configure MIDI connections between System object and MIDI controller
createAudioPluginClass	Create audio plugin class that implements functionality of System object
disconnectMIDI	Disconnect MIDI controls from System object
getMIDIConnections	Get MIDI connection information
reset	Reset internal states of System object
step	Perform dynamic range limiting
visualize	Visualize static limiter characteristics of System object

Common to All System Objects	
`clone`	Create System object with same property values
`getNumInpu`	Expected number of inputs to a System object
`getNumOutp`	Expected number of outputs of a System object
`isLocked`	Check locked states of a System object (logical)
`release`	Allow System object property value changes

Examples

Limit Audio Signal

Use dynamic range limiting to suppress the volume of loud sounds.

Set up the audio file reader and audio device writer System objects.

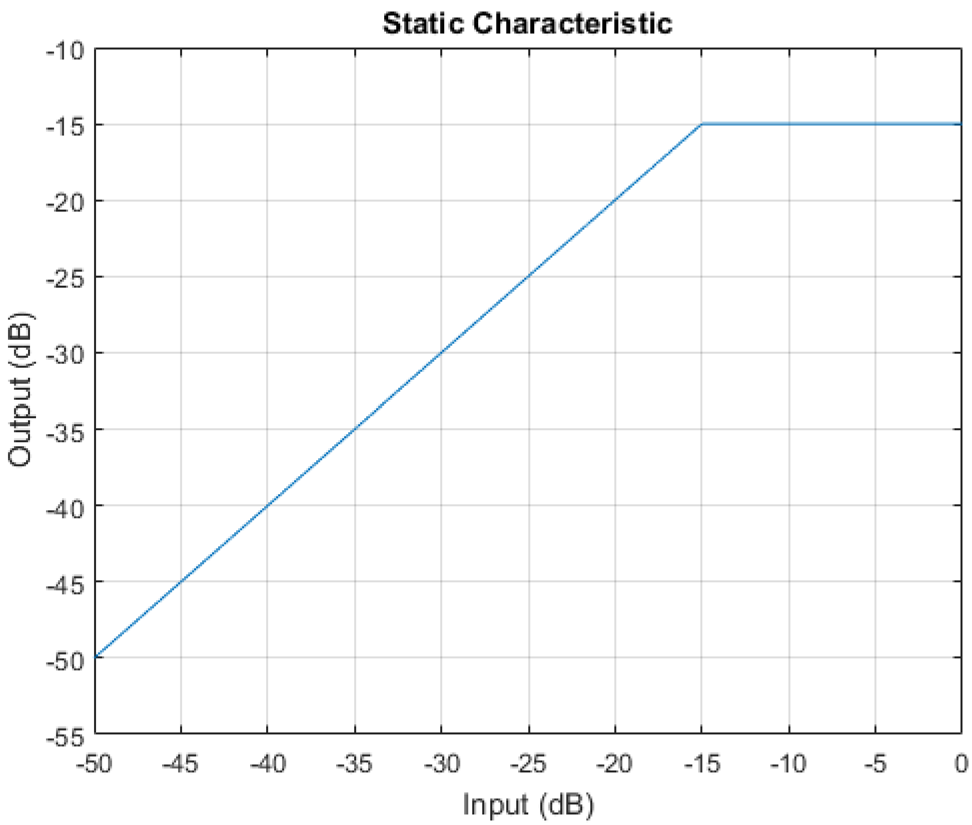
```
frameLength = 1024;
fileReader = dsp.AudioFileReader(...
    'Filename','RockDrums-44p1-stereo-11secs.mp3',...
    'SamplesPerFrame',frameLength);
deviceWriter = audioDeviceWriter(...
    'SampleRate',fileReader.SampleRate);
```

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);
```

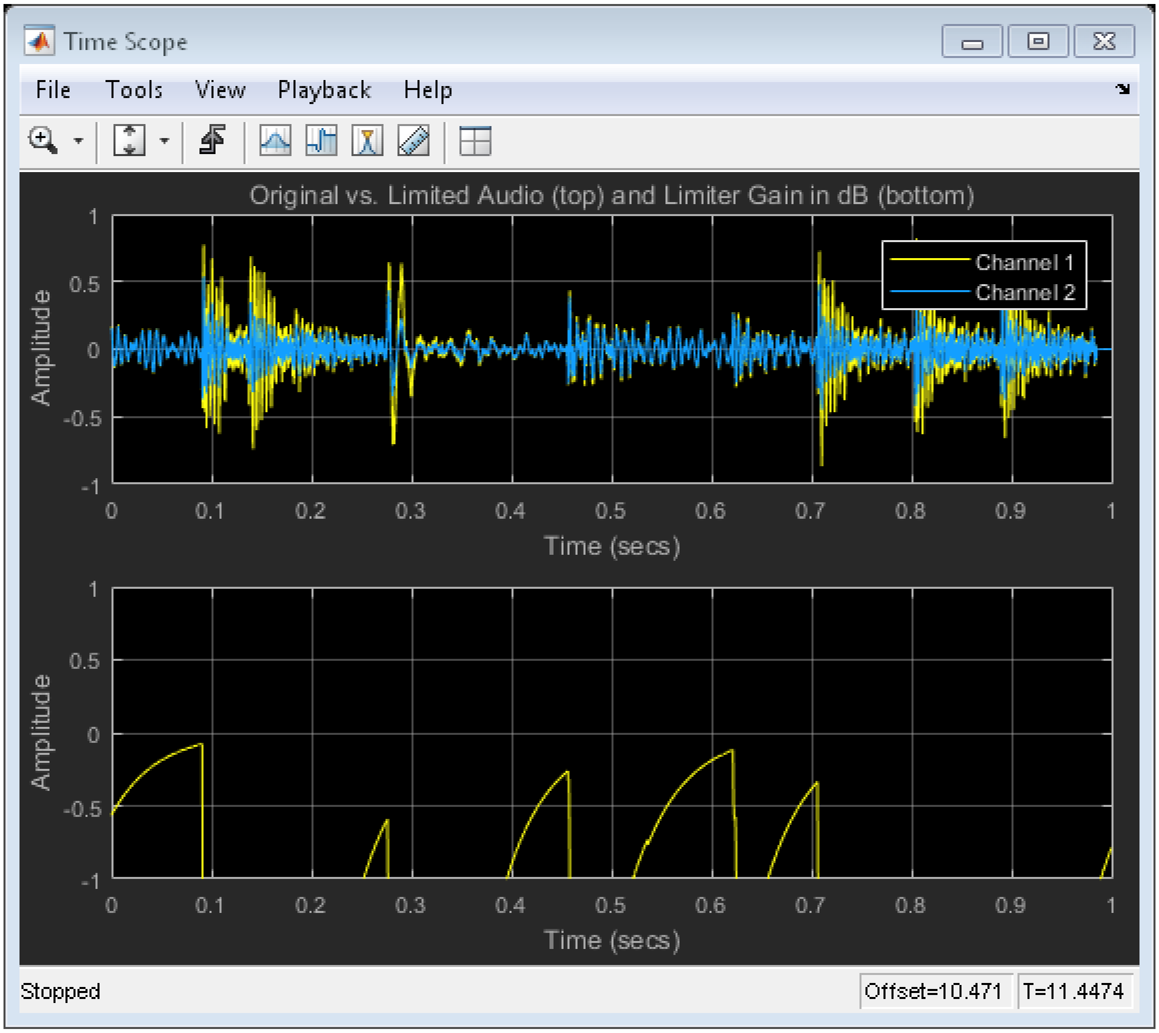


Set up a time scope to visualize the original signal and the limited signal.

```
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`.

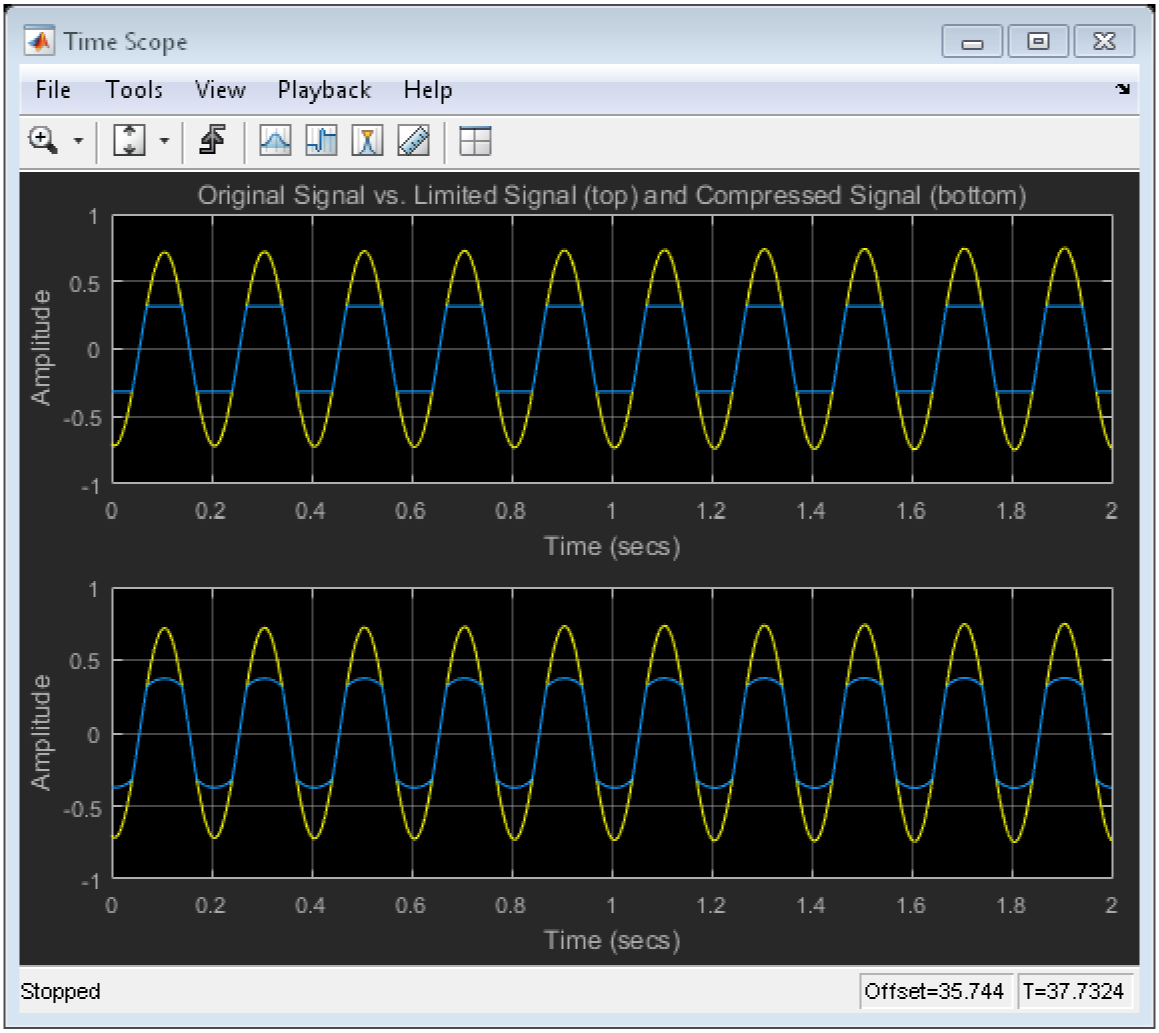
```
dRL = limiter('AttackTime',0,'ReleaseTime',0);
dRC = compressor('AttackTime',0,'ReleaseTime',0);

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` object and `audioDeviceWriter` 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);
```

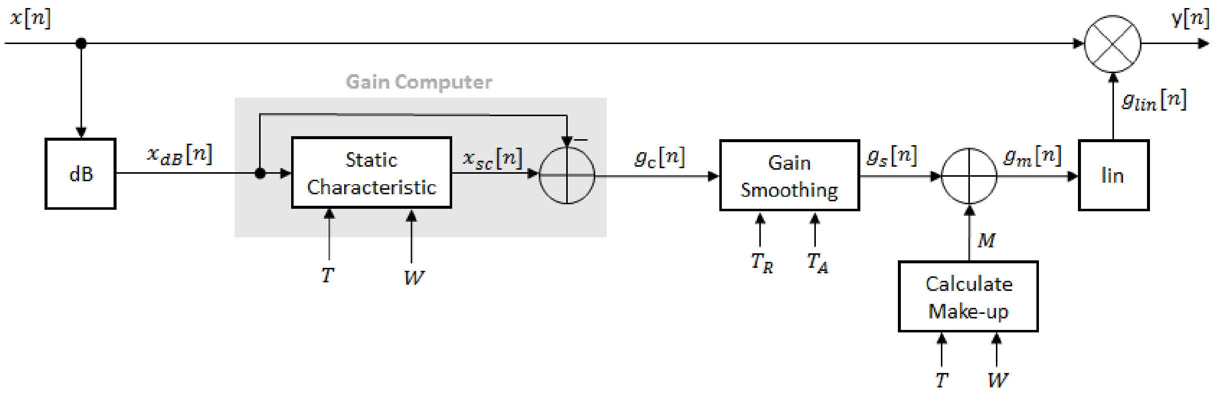
Read successive frames from an audio file in a loop. Listen to and compare the effect of dynamic range limiting and dynamic range compression on an audio signal.

```
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)
```

```
Now playing original signal...
Now playing limited signal...
Now playing compressed signal...
```

Algorithms

The `limiter` System object 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 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:

$$x_{sc}(x_{dB}) = \begin{cases} x_{dB} & x_{dB} < T \\ T & x_{dB} \geq T \end{cases}$$

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 and release time properties:

$$g_s[n] = \begin{cases} \alpha_A g_s[n-1] + (1-\alpha_A) g_c[n], & g_c[n] > g_s[n-1] \\ \alpha_R g_s[n-1] + (1-\alpha_R) g_c[n], & g_c[n] \leq g_s[n-1] \end{cases}$$

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.

5 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:

$$M = -x_{sc}(x_{dB} = 0)$$

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.

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, 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

See Also

Blocks
`Limiter`

System Objects
noiseGate | compressor | expander

Topics

“Dynamic Range Control”

Introduced in R2016a

configureMIDI

System object: limiter

Configure MIDI connections between System object and MIDI controller

Syntax

```
configureMIDI(dRL)
configureMIDI(dRL,propName)
configureMIDI(dRL,propName,controlNumber)
configureMIDI(dRL,propName,controlNumber,'DeviceName',deviceName)
```

Description

`configureMIDI(dRL)` starts a MIDI configuration user interface (UI). Use the UI to synchronize tunable properties of the dynamic range limiter System object, `dRL`, to MIDI controls of your choice.

`configureMIDI(dRL,propName)` makes the System object property, `propName`, respond to any control on the default MIDI device.

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

`configureMIDI(dRL,propName,controlNumber,'DeviceName',deviceName)` makes the property respond to the MIDI control specified by `controlNumber` on the device specified by `deviceName`.

Each tunable property of the `lLimiter` System object maps to a MIDI control with a specified range.

Property	Range	Unit
`Threshold`	–50 to 0	dB
`KneeWidth`	0 to 20	dB
`AttackTime`	0 to 4	seconds

Property	Range	Unit
`ReleaseTime`	0 to 4	seconds
`MakeUpGain` (available when you set `MakeUpGainMode` to `'Property'`)	–10 to 24	dB

Introduced in R2016a

createAudioPluginClass

System object: limiter

Create audio plugin class that implements functionality of System object

Syntax

```
createAudioPluginClass(dRL)
createAudioPluginClass(dRL,pluginName)
```

Description

`createAudioPluginClass(dRL)` creates a System object plugin that implements the functionality of the `dynamicRangeLimiter` System object, `dRL`. The name of the created class is the `limiter` System object variable name followed by `'Plugin'`, for example, `dRLPlugin`.

Note: 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(dRL,pluginName)` specifies the name of your created System object plugin class.

Example: `createAudioPluginClass(dRL,'myLimiter')` creates a System object plugin with class name `myLimiter`.

Each tunable property of the `limiter` System object maps to a plugin parameter with a default range.

Property	Plugin Parameter Range	Unit
`Threshold`	–50 to 0	dB
`KneeWidth`	0 to 20	dB
`AttackTime`	0 to 4	s

Property	Plugin Parameter Range	Unit
`ReleaseTime`	0 to 4	s
`MakeUpGain` (available when you set `MakeUpGainMode` to `'Property'`)	–10 to 24	dB

Introduced in R2016a

disconnectMIDI

System object: limiter

Disconnect MIDI controls from System object

Syntax

```
disconnectMIDI(dRL)
```

Description

`disconnectMIDI(dRL)` disconnects MIDI controls from your dynamic range limiter, `dRL`. Only those MIDI connections established using `configureMIDI` are disconnected.

Introduced in R2016a

getMIDIConnections

System object: limiter

Get MIDI connection information

Syntax

```
connectionInfo = getMIDIConnections(dRL)
```

Description

`connectionInfo = getMIDIConnections(dRL)` returns a structure, `connectionInfo`, containing information about the MIDI connections for your dynamic range limiter, `dRL`. Only those MIDI connections established using `configureMIDI` are returned. The `connectionInfo` structure contains a substructure for each tunable property of `dRL` 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).

Introduced in R2016a

reset

System object: limiter

Reset internal states of System object

Syntax

```
reset(dRL)
```

Description

`reset(dRL)` resets internal states of the dynamic range limiter, `dRL`, to their initial values.

Introduced in R2016a

step

System object: limiter

Perform dynamic range limiting

Syntax

```
y = step(dRL,x)
[y,g] = step(dRL,x)
```

Description

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`y = step(dRL,x)` performs dynamic range limiting on the input signal, `x`, and returns the limited signal, `y`. The type of dynamic range limiting is specified by the algorithm and properties of the `limiter` System object, `dRL`.

`x` must be a real-valued, double-precision or single-precision matrix. The System object treats each column of the input as an independent channel.

Note: The System object performs an internal initialization the first time you execute `step`. This initialization locks nontunable properties and input specifications, such as the dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

`[y,g] = step(dRL,x)` also returns the gain, in dB, applied at each input sample.

Introduced in R2016a

visualize

System object: limiter

Visualize static limiter characteristics of System object

Syntax

```
visualize(dRL)
visualize(dRL,myInputRange)
Y = visualize( ___ )
```

Description

`visualize(dRL)` plots the static compression characteristic of the dynamic range limiter, `dRL`. The method computes the dB output level for the input range `[-50:0.01:0]` dB. The plot is updated automatically when properties of the object change.

`visualize(dRL,myInputRange)` enables you to specify the input range in dB. Specify `myInputRange` as a vector of ascending values.

`Y = visualize( ___ )` returns the dB output level, `Y`, corresponding to the input range. You can use any of the input arguments from previous syntaxes.

Introduced in R2016a

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 on your input:

1. Define and set up your loudness meter. See "Construction" on page 3-175.
2. Call step to perform loudness metering on the input signal according to the properties of your `loudnessMeter` object. The input must be a real-valued, double-precision or single-precision matrix. The `loudnessMeter` object treats each column of the input as an independent channel. If you use the default `ChannelWeights`, specify the input channels in order: [Left, Right, Center, Left surround, Right surround].

Note: Alternatively, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`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 12,000 Hz.

Properties

If a property is listed as tunable, then you can change its value even when the object is locked.

`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].

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

Tunable: Yes

`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

`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`.

`SampleRate` — Input sample rate (Hz)
`44100` (default) | positive scalar

Input sample rate in Hz, specified as a positive scalar.

Tunable: Yes

Methods

reset	Reset internal states of System object
step	Compute loudness in accordance with EBU R 128 and ITU-R BS.1770-4
visualize	Open 'EBU Mode' meter display

Common to All System Objects	
`clone`	Create System object with same property values
`getNumInpu`	Expected number of inputs to a System object
`getNumOutp`	Expected number of outputs of a System object
`isLocked`	Check locked states of a System object (logical)
`release`	Allow System object property value changes

Examples

Loudness of Audio Signal

Create a file reader 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`.

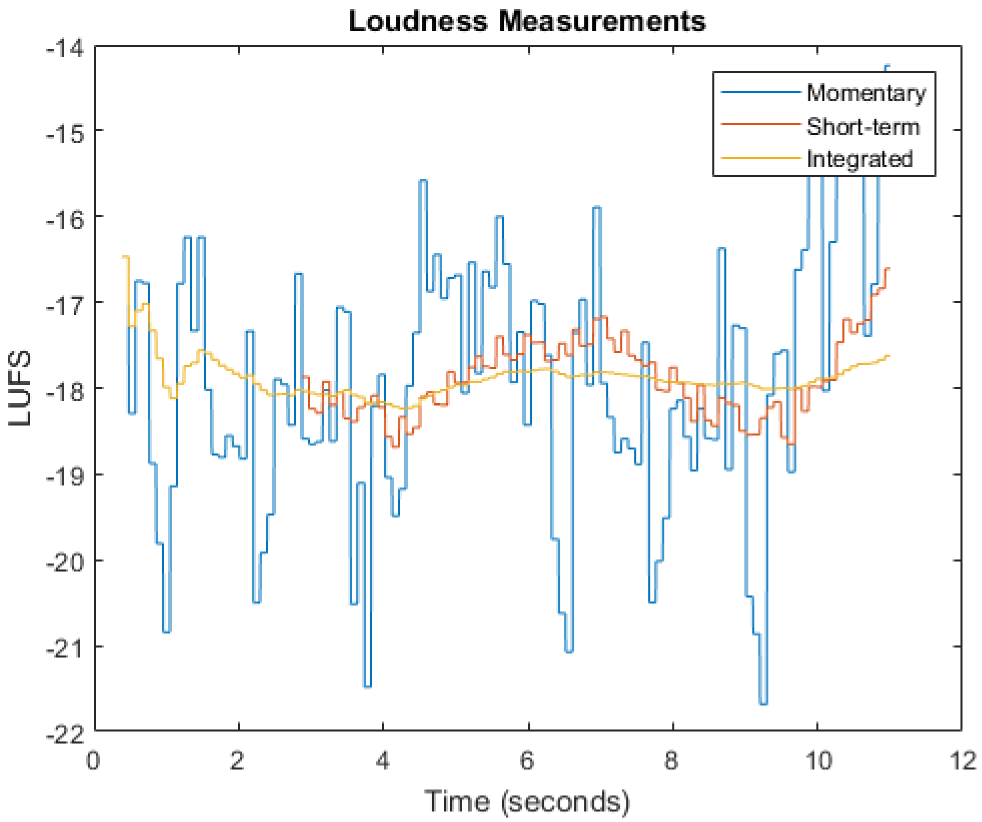
```
fileReader = dsp.AudioFileReader('RockDrums-44p1-stereo-11secs.mp3');
loudMtr = loudnessMeter('SampleRate',fileReader.SampleRate);
```

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.

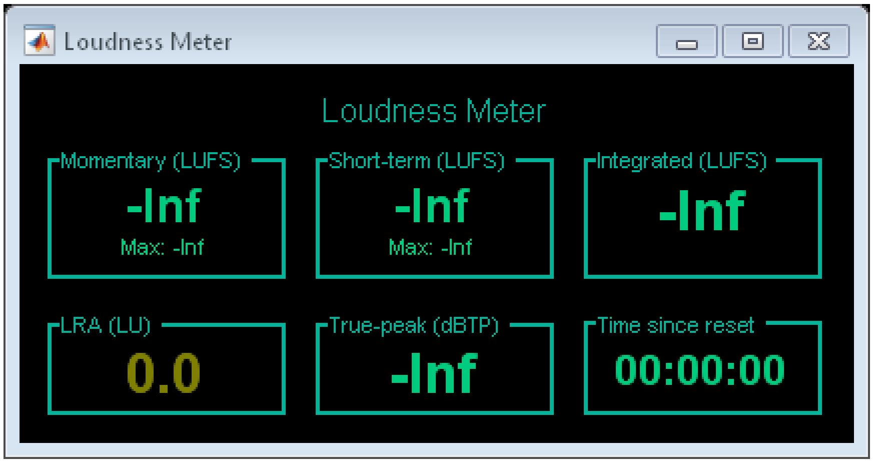
```
fileReader = dsp.AudioFileReader('FunkyDrums-44p1-stereo-25secs.mp3', ...
    'SamplesPerFrame',1024);
fs = fileReader.SampleRate;
deviceWriter = audioDeviceWriter('SampleRate',fs);
```

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);
```



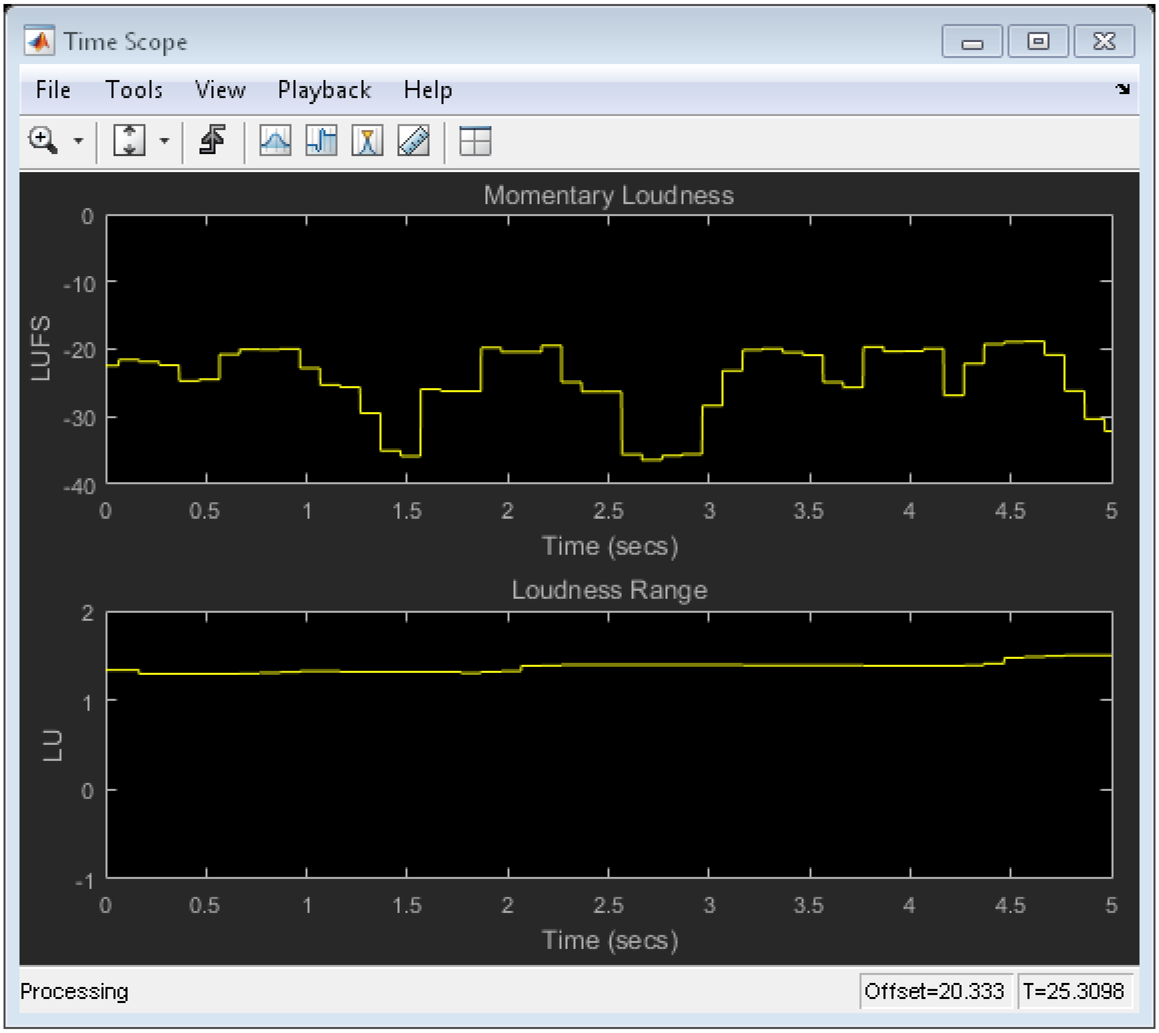
In an audio stream loop:

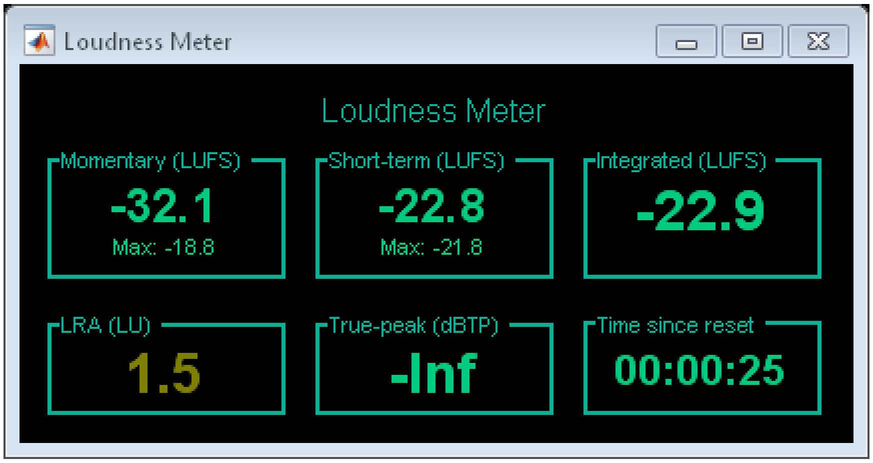
- Read in your audio file.
- Compute the momentary loudness and loudness range.

- Visualize the momentary loudness and loudness range on your time scope.
- Listen to 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)
```


Time Scope
File
Tools
View
Playback
Help
Momentary Loudness
LUFS
0
-10
-20
-30
-40
0
0.5
1
1.5
2
2.5
3
3.5
4
4.5
5
Time (secs)
Loudness Range
LU
2
1
0
-1
Time (secs)
Processing
Offset=20.333
T=25.3098



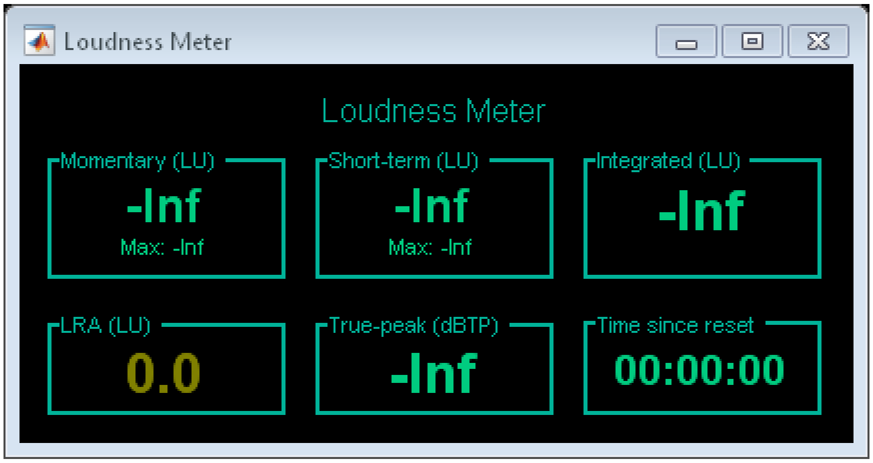
Relative Scale for Loudness Measurements

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)
```

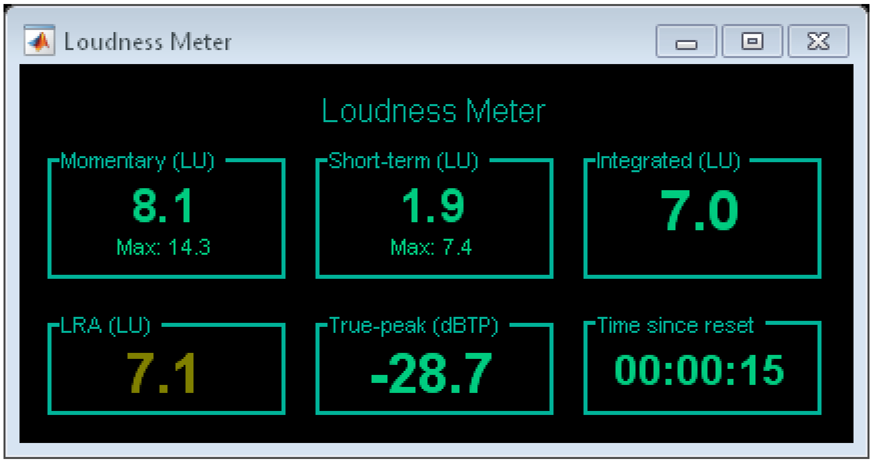


Create a time scope to visualize your audio signal and its measured relative momentary and short-term loudness.

```
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

Momentary (LU)
8.1
Max: 14.3

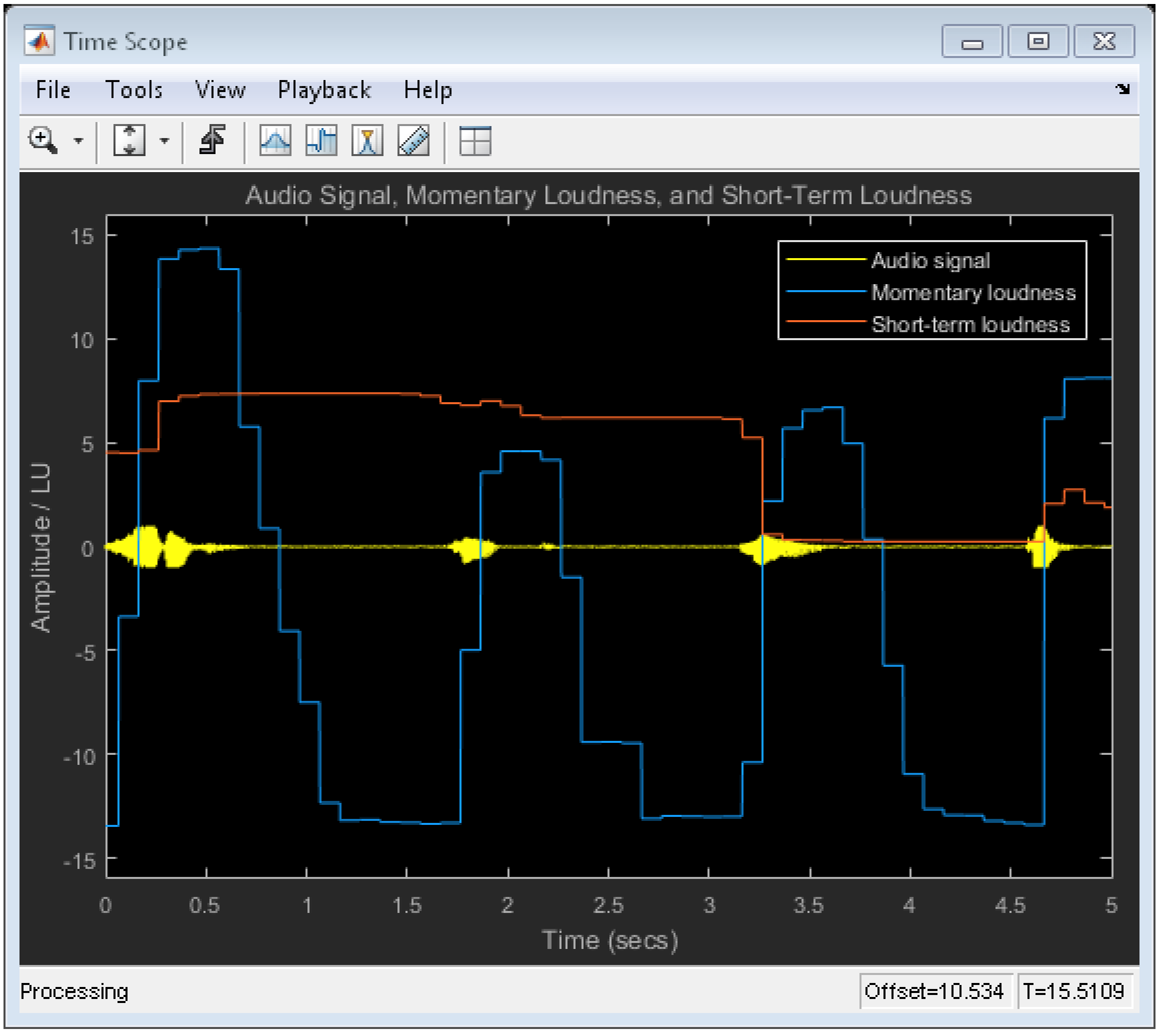
Short-term (LU)
1.9
Max: 7.4

Integrated (LU)
7.0

LRA (LU)
7.1

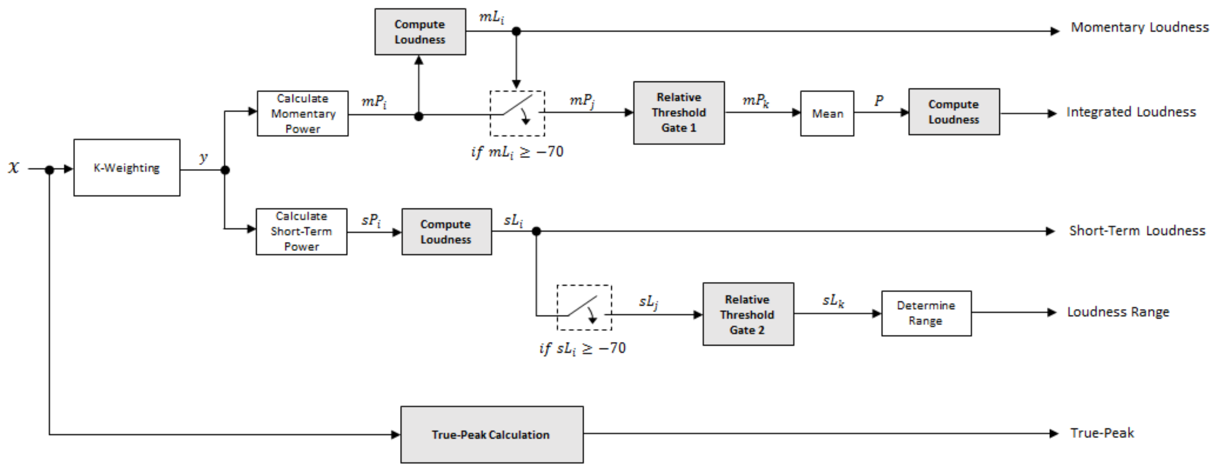
True-peak (dBTP)
-28.7

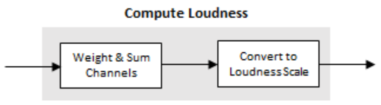
Time since reset
00:00:15



Algorithm

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.

Momentary Loudness and Integrated 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` 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 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 `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.

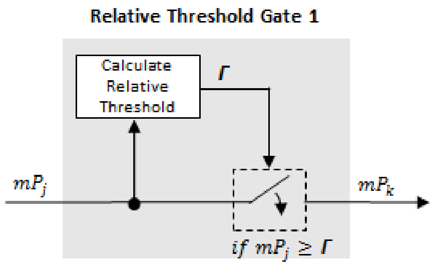
The momentary power segment is gated using the corresponding momentary loudness segment calculation:

$$mP_i \rightarrow mP_j$$

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

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:

$$mP_j \to mP_k$$

$$k = \{\, j \mid mP_j \geq \Gamma \,\}$$

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 by passing the mean momentary power, P, through the Compute Loudness system:

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

Short-Term Loudness and Loudness Range

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` 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 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 `loudnessMeter` System object. It is also used internally to calculate the loudness range (steps 3–5).

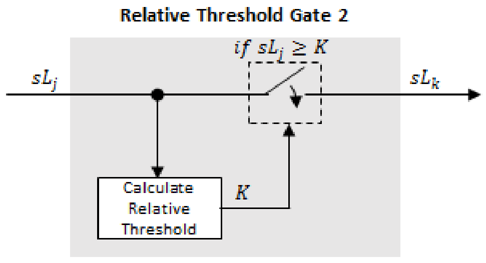
3 The short-term loudness is gated using an absolute threshold:

$$sL_i \rightarrow sL_j$$

$$j = \{\ i \mid sL_i \geq -70\ \}$$

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:

$$sL_j \rightarrow sL_k$$

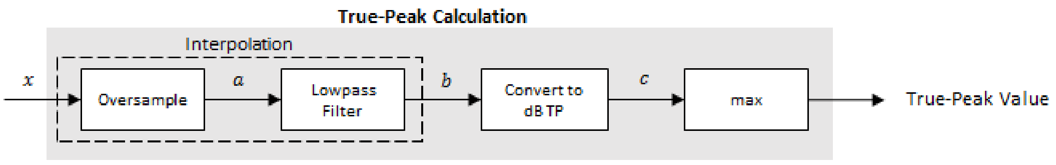
$$k = \left\{ j \mid sL_j \geq K \right\}$$

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
[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.

[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

See Also

System Objects
weightingFilter | octaveFilter

Blocks
`Loudness Meter`

Functions
`integratedLoudness`

Introduced in R2016b

reset

System object: loudnessMeter

Reset internal states of System object

Syntax

```
reset(loudMtr)
```

Description

`reset(loudMtr)` resets internal states of the loudness meter, `loudMtr`, to their initial values.

Introduced in R2016b

step

System object: loudnessMeter

Compute loudness in accordance with EBU R 128 and ITU-R BS.1770-4

Syntax

`[momentary,shortTerm,integrated,range,peak] = step(loudMtr,x)`

Description

Note: Alternatively, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`[momentary,shortTerm,integrated,range,peak] = step(loudMtr,x)` 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, `x`. It also returns the integrated loudness and loudness range of the input to your loudness meter since the last time `reset` was called. By default, loudness measurements are returned in loudness units full scale (LUFS). If you set the `UseRelativeScale` property to `true`, loudness measurements are returned in loudness units (LU). The true-peak value is returned in dB TP.

`x` must be a real-valued, double-precision or single-precision matrix. The System object treats each column of the input as an independent channel. 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].

`momentary`, `shortTerm`, `integrated`, and `range` are column vectors with the same number of rows as `x`. `peak` is a scalar value.

Note: The System object performs an internal initialization the first time you execute `step`. This initialization locks nontunable properties and input specifications, such as the dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

Introduced in R2016b

visualize

System object: loudnessMeter

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`.

As per the EBU R 128 standard, the loudness range value is displayed in yellow for the first 60 seconds after reset. The yellow indicates that the value is not yet considered stable.

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. Define and set up your multiband parametric equalizer. See “Construction” on page 3-198.
2. Call step to perform multiband parametric equalization on each channel of the input signal according to the properties of your `multibandParametricEQ` object. The input must be a real-valued, double-precision or single-precision matrix. The `multibandParametricEQ` object treats each column of the input as an independent channel.

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`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 construction 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 construction, the first property you specify locks the value.

Construction Arguments

`NumEQBands` — Number of equalizer bands

3 (default) | integer in the range 1 to 10

Number of equalizer bands, specified as an integer in the range 1 to 10. The number of equalizer bands does not include shelving filters, highpass filters, or lowpass filters.

`NumEQBands` must be set during construction. It cannot be modified after construction.

Example: `mPEQ = multibandParametricEQ('NumEQBands',5)` constructs a multiband parametric equalizer with 5 bands.

`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` must be set during construction. It cannot be modified after construction.

Example: `mPEQ = multibandParametricEQ('EQOrder',6)` constructs a multiband parametric equalizer with the default 3 bands, all of order 6.

`HasLowShelfFilter` — Low-shelf filter toggle

`false` (default) | `true`

Low-shelf filter toggle, specified as `false` or `true`.

- `false` — Do not include low-shelf filter in multiband parametric equalizer implementation.
- `true` — Include low-shelf filter in multiband parametric equalizer implementation.

`HasLowpassFilter` must be set during construction. It cannot be modified after construction.

Example: `mPEQ = multibandParametricEQ('HasLowShelfFilter',true)` constructs a default multiband parametric equalizer with low-shelf filtering enabled.

`HasHighShelfFilter` — High-shelf filter toggle

`false` (default) | `true`

High-shelf filter toggle, specified as `false` or `true`.

- `false` — Do not include high-shelf filter in multiband parametric equalizer implementation.
- `true` — Include high-shelf filter in multiband parametric equalizer implementation.

`HasHighShelfFilter` must be set during construction. It is cannot be modified after construction.

Example: `mPEQ = multibandParametricEQ('HasHighShelfFilter',true)` constructs a default multiband parametric equalizer with high-shelf filtering enabled.

`HasLowpassFilter` — Lowpass filter toggle

`false` (default) | `true`

Lowpass filter toggle, specified as `false` or `true`.

- `false` — Do not include lowpass filter in multiband parametric equalizer implementation.
- `true` — Include lowpass filter in multiband parametric equalizer implementation.

`HasLowpassFilter` must be set during construction. It cannot be modified after construction.

Example: `mPEQ = multibandParametricEQ('HasLowpassFilter',true)` constructs a default multiband parametric equalizer with lowpass filtering enabled.

`HasHighpassFilter` — Highpass filter toggle

`false` (default) | `true`

Highpass filter toggle, specified as `false` or `true`.

- `false` — Do not include highpass filter in multiband parametric equalizer implementation.
- `true` — Include highpass filter in multiband parametric equalizer implementation.

`HasHighpassFilter` must be set during construction. It cannot be modified after construction.

Example: `mPEQ = multibandParametricEQ('HasHighpassFilter',true)` constructs a default multiband parametric equalizer with highpass filtering enabled.

`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` must be set during construction. It cannot be modified after construction.

Example: `mPEQ = multibandParametricEQ('Oversample',true)` constructs a default multiband parametric equalizer with oversampling enabled.

Properties

If a property is listed as tunable, then you can change its value even when the object is locked.

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

`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 to 700. Any values outside the range are saturated.

Tunable: Yes

`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` to 20. Values above 20 are saturated.

Tunable: Yes

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.

This property is available when you set `HasLowShelfFilter` to `true` during construction.

Tunable: Yes

`LowShelfSlope` — Low-shelf filter slope coefficient

`1.5` (default) | real scalar in the range 0.1 to 5

Low-shelf filter slope coefficient, specified as a real scalar in the range 0.1 to 5. Values outside the range are saturated.

This property is available when you set `HasLowShelfFilter` to `true` during construction.

Tunable: Yes

`LowShelfGain` — Low-shelf filter gain (dB)

`0` (default) | real scalar in the range –12 to 12

Low-shelf filter gain in dB, specified as a real scalar in the range –12 to 12. Values outside the range are saturated.

This property is available when you set `HasLowShelfFilter` to `true` during construction.

Tunable: Yes

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.

This property is available when you set `HasHighShelfFilter` to `true` during construction.

Tunable: Yes

`HighShelfSlope` — High-shelf slope coefficient

`1.5` (default) | real scalar in the range 0.1 to 5

High-shelf filter slope coefficient, specified as a real scalar in the range 0.1 to 5. Values outside the range are saturated.

This property is available when you set `HasHighShelfFilter` to `true` during construction.

Tunable: Yes

`HighShelfGain` — High-shelf filter gain (dB)

`0` (default) | real scalar in the range –12 to 12

High-shelf filter gain in dB, specified as a real scalar in the range –12 to 12. Values outside the range are saturated.

This property is available when you set `HasHighShelfFilter` to `true` during construction.

Tunable: Yes

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.

This property is available when you set `HasLowpassFilter` to `true` during construction.

Tunable: Yes

`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.

This property is available when you set `HasLowpassFilter` to `true` during construction.

Tunable: Yes

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.

This property is available when you set `HasHighpassFilter` to `true` during construction.

Tunable: Yes

`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.

This property is available when you set `HasHighpassFilter` to `true` during construction.

Tunable: Yes

Sampling

`SampleRate` — Input sample rate (Hz)

`44100` (default) | positive scalar

Input sample rate in Hz, specified as a positive scalar.

Tunable: Yes

Methods

configureMIDI	Configure MIDI connections between System object and MIDI controller
createAudioPluginClass	Create audio plugin class that implements functionality of System object
reset	Reset internal states of System object
step	Perform multiband parametric equalization
visualize	Visualize magnitude response of System object

Common to All System Objects	
`clone`	Create System object with same property values
`getNumInpu`	Expected number of inputs to a System object
`getNumOutp`	Expected number of outputs of a System object
`isLocked`	Check locked states of a System object (logical)
`release`	Allow System object property value changes

Examples

Multiband Parametric Equalization

Create 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.

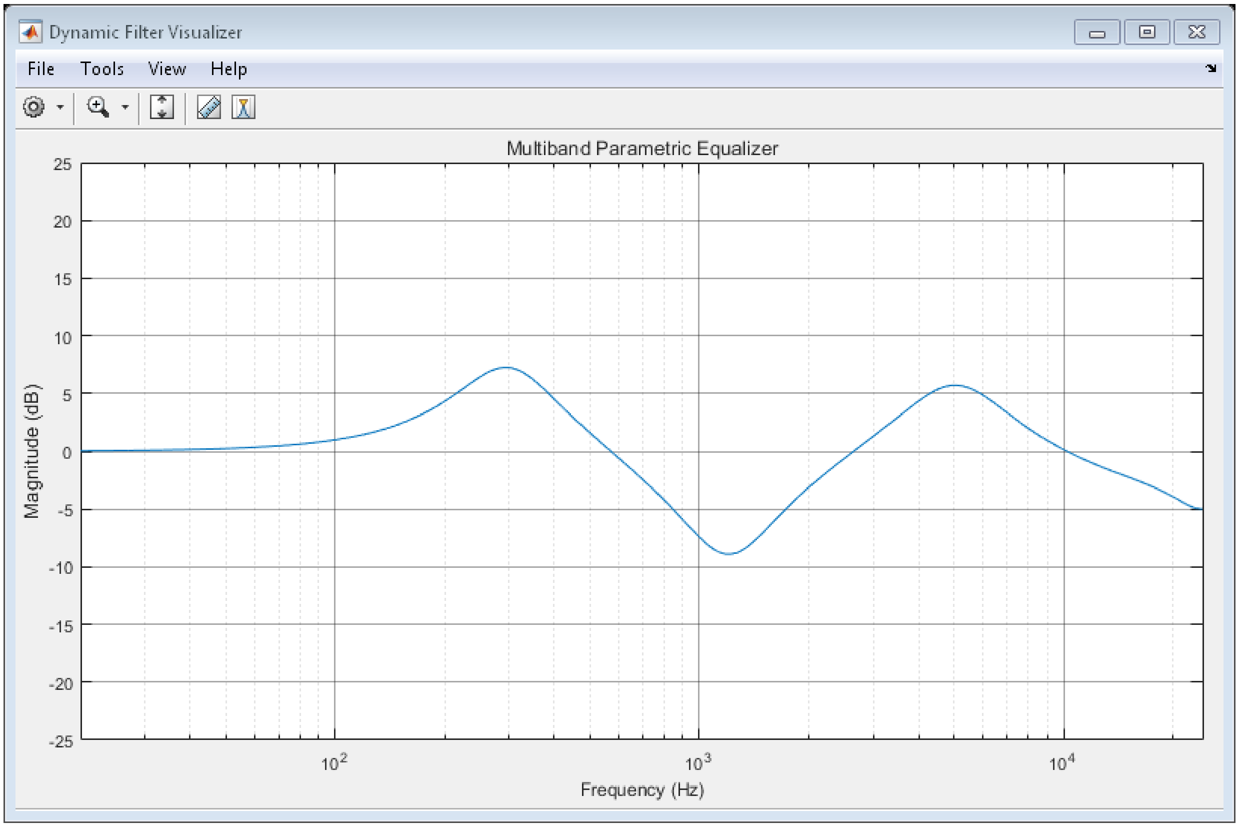
```
frameLength = 512;

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);
```

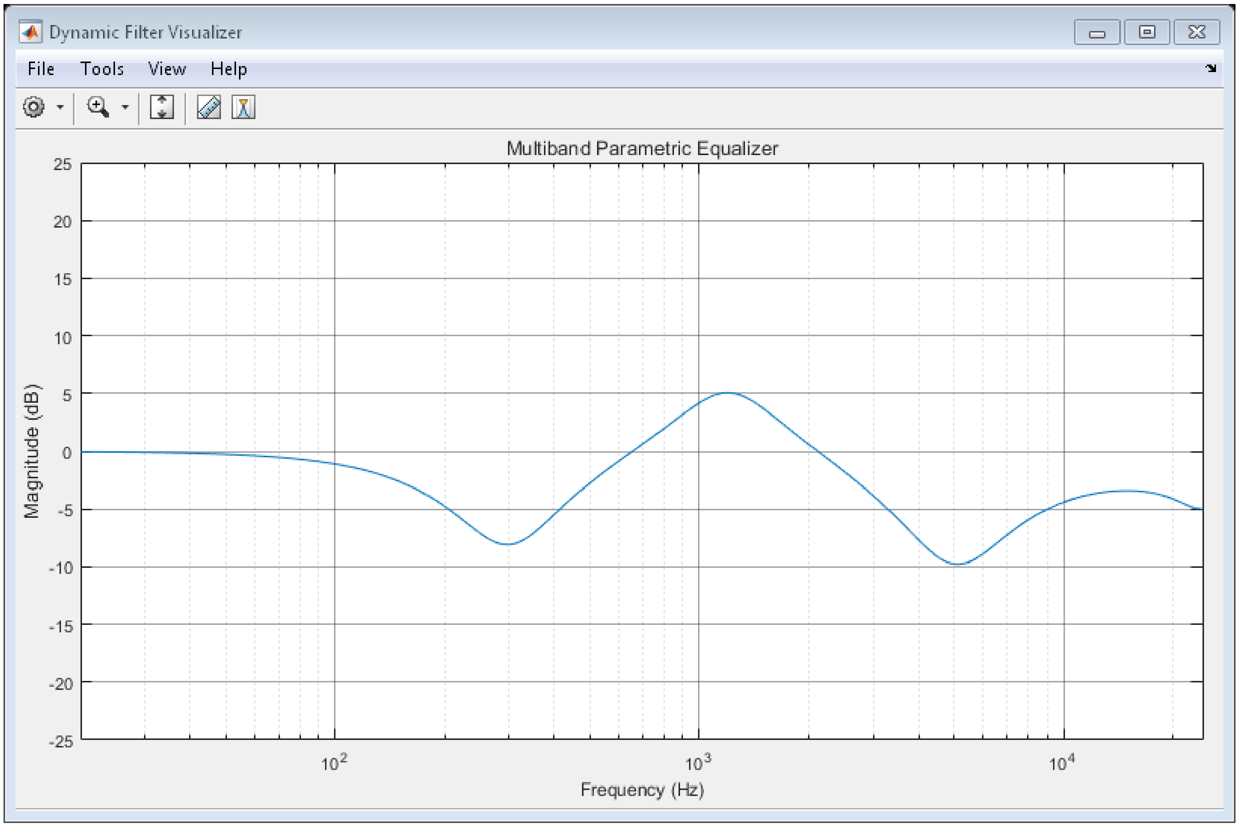
Visualize the magnitude frequency response of your multiband parametric equalizer.

```
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;
    visualize(mPEQ)
end

release(fileReader)
release(mPEQ)
release(deviceWriter)
```

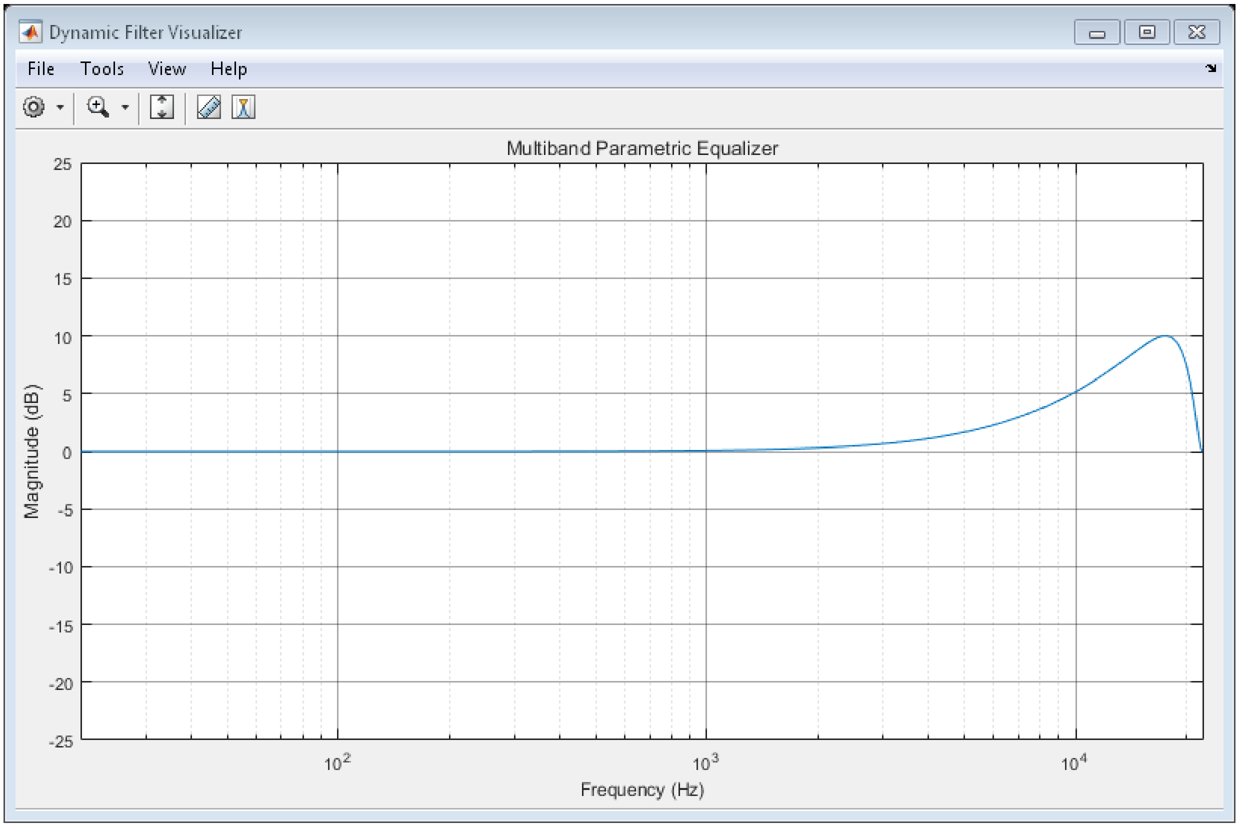


Oversample Audio Signal

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.

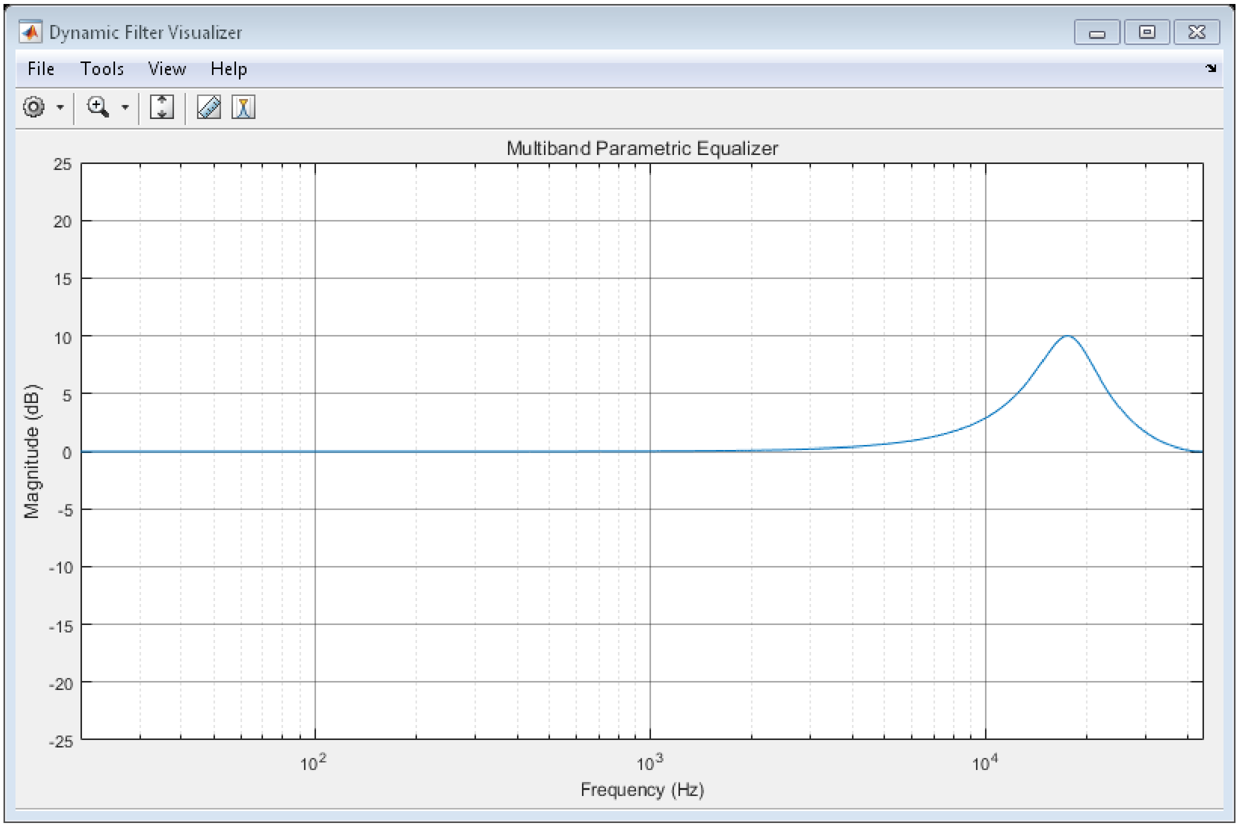
```
mPEQ = multibandParametricEQ(...
    'NumEQBands',1,...
    'Frequencies',9.5e3,...
    'PeakGains',10);
visualize(mPEQ)
for i = 1:1000
    mPEQ.Frequencies = mPEQ.Frequencies + 8;
end
```



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

See Also

Blocks

`Parametric EQ Filter`

Functions

`designParamEQ` | `designShelvingEQ` | `designVarSlopeFilter`

Topics

“Parametric Equalizer Design”
“Equalization”

Introduced in R2016a

configureMIDI

System object: multibandParametricEQ

Configure MIDI connections between System object and MIDI controller

Syntax

```
configureMIDI(mPEQ)
configureMIDI(mPEQ,propName)
configureMIDI(mPEQ,propName,controlNumber)
configureMIDI(mPEQ,propName,controlNumber,'DeviceName',deviceName)
```

Description

`configureMIDI(mPEQ)` starts a MIDI configuration user interface (UI). Use the UI to synchronize tunable properties of the multiband parametric equalizer System object, `mPEQ`, to MIDI controls of your choice.

`configureMIDI(mPEQ,propName)` makes the System object property, `propName`, respond to any control on the default MIDI device.

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

`configureMIDI(mPEQ,propName,controlNumber,'DeviceName',deviceName)` makes the property respond to the MIDI control specified by `controlNumber` on the device specified by `deviceName`.

Each tunable property of the `multibandParametricEQ` System object maps to MIDI controls with a specified range.

Property	Range	Unit
`Frequencies`	20 to 20,000 (log scale)	Hz
`QualityFactors`	0.2 to 700	none
`PeakGains`	–50 to 20	dB

Property	Range	Unit
`LowShelfCutoff`	20 to 20,000 (log scale)	Hz
`LowShelfSlope`	0.1 to 5	none
`LowShelfGain`	–12 to 12	dB
`HighShelfCutoff`	20 to 20,000 (log scale)	Hz
`HighShelfSlope`	0.1 to 5	none
`HighShelfGain`	–12 to 12	dB
`LowpassCutoff`	20 to 20,000 (log scale)	Hz
`LowpassSlope`	0 to 48	dB/octave
`HighpassCutoff`	20 to 20,000 (log scale)	Hz
`HighpassSlope`	0 to 48	dB/octave

Introduced in R2016b

createAudioPluginClass

System object: multibandParametricEQ

Create audio plugin class that implements functionality of System object

Syntax

```
createAudioPluginClass(mPEQ)
createAudioPluginClass(mPEQ,pluginName)
```

Description

`createAudioPluginClass(mPEQ)` creates a System object plugin that implements the functionality of the `multibandParametricEQ` System object, `mPEQ`. The name of the created class is the `multibandParametricEQ` System object variable name followed by `'Plugin'`, for example, `mPEQPlugin`.

Note: 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(mPEQ,pluginName)` specifies the name of your created System object plugin class.

Example: `createAudioPluginClass(mPEQ,'Equalizer')` creates a System object plugin with class name `Equalizer`.

Each tunable property of the `multibandParametricEQ` System object maps to a plugin parameter with a default range. For vector properties such as `Frequencies`, each element of the vector maps to a plugin parameter.

Property	Plugin Parameter Range	Unit
`Frequencies`	20 to 20,000 (log mapping)	Hz
`QualityFactors`	0.2 to 700	none

Property	Plugin Parameter Range	Unit
`PeakGains`	–50 to 20	dB
`LowShelfCutoff`	20 to 20,000 (log mapping)	Hz
`LowShelfSlope`	0.1 to 5	none
`LowShelfGain`	–12 to 12	dB
`HighShelfCutoff`	20 to 20,000 (log mapping)	Hz
`HighShelfSlope`	0.1 to 5	none
`HighShelfGain`	–12 to 12	dB
`LowpassCutoff`	20 to 20,000 (log mapping)	Hz
`LowpassSlope`	0 to 48	dB/octave
`HighpassCutoff`	20 to 20,000 (log mapping)	Hz
`HighpassSlope`	0 to 48	dB/octave

Introduced in R2016b

reset

System object: multibandParametricEQ

Reset internal states of System object

Syntax

`reset(mPEQ)`

Description

`reset(mPEQ)` resets internal states of the multiband parametric equalizer, `mPEQ`, to their initial values.

Introduced in R2016a

step

System object: multibandParametricEQ

Perform multiband parametric equalization

Syntax

```
y = step(mPEQ,x)
```

Description

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`y = step(mPEQ,x)` performs multiband parametric equalization on the input signal, `x`, and returns the filtered signal, `y`. The type of equalization is specified by the algorithm and properties of the `multibandParametricEQ` System object, `mPEQ`.

`x` must be a real-valued, double-precision or single-precision matrix. The System object treats each column of the input as an independent channel.

Note: The System object performs an internal initialization the first time you execute `step`. This initialization locks nontunable properties and input specifications, such as the dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

Introduced in R2016a

visualize

System object: multibandParametricEQ

Visualize magnitude response of System object

Syntax

```
visualize(mPEQ)
visualize(mPEQ,N)
```

Description

`visualize(mPEQ)` plots the magnitude response of the multiband parametric equalizer. The plot includes any enabled shelving filters, lowpass filters, or highpass filters. The plot is updated automatically when properties of the object change.

`visualize(mPEQ,N)` specifies an `N`-point FFT used to calculate the magnitude response. The default is `2048`.

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 on your input:

1. Define and set up your dynamic range gate. See “Construction” on page 3-219.
2. Call step to perform dynamic range gating on each channel of the input signal according to the properties of your `noiseGate` object. The input must be a real-valued, double-precision or single-precision matrix. The `noiseGate` object treats each column of the input as an independent channel.

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`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 the `AttackTime` property set to `0.01`, and the `SampleRate` property set to `16000`.

Properties

If a property is listed as tunable, then you can change its value even when the object is locked.

`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

`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

`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

`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

`SampleRate` — Input sample rate (Hz)

`44100` (default) | positive scalar

Input sample rate in Hz, specified as a positive scalar.

Tunable: Yes

Methods

configureMIDI	Configure MIDI connections between System object and MIDI controller
createAudioPluginClass	Create audio plugin class that implements functionality of System object
disconnectMIDI	Disconnect MIDI controls from System object
getMIDIConnections	Get MIDI connection information
reset	Reset internal states of System object
step	Perform dynamic range gating
visualize	Visualize static gate characteristics of System object

Common to All System Objects	
`clone`	Create System object with same property values
`getNumInpu`	Expected number of inputs to a System object
`getNumOutp`	Expected number of outputs of a System object
`isLocked`	Check locked states of a System object (logical)

Common to All System Objects	
`release`	Allow System object property value changes

Examples

Gate Audio Signal

Use dynamic range gating to attenuate background noise from an audio signal.

Set up the audio file reader and audio device writer 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.

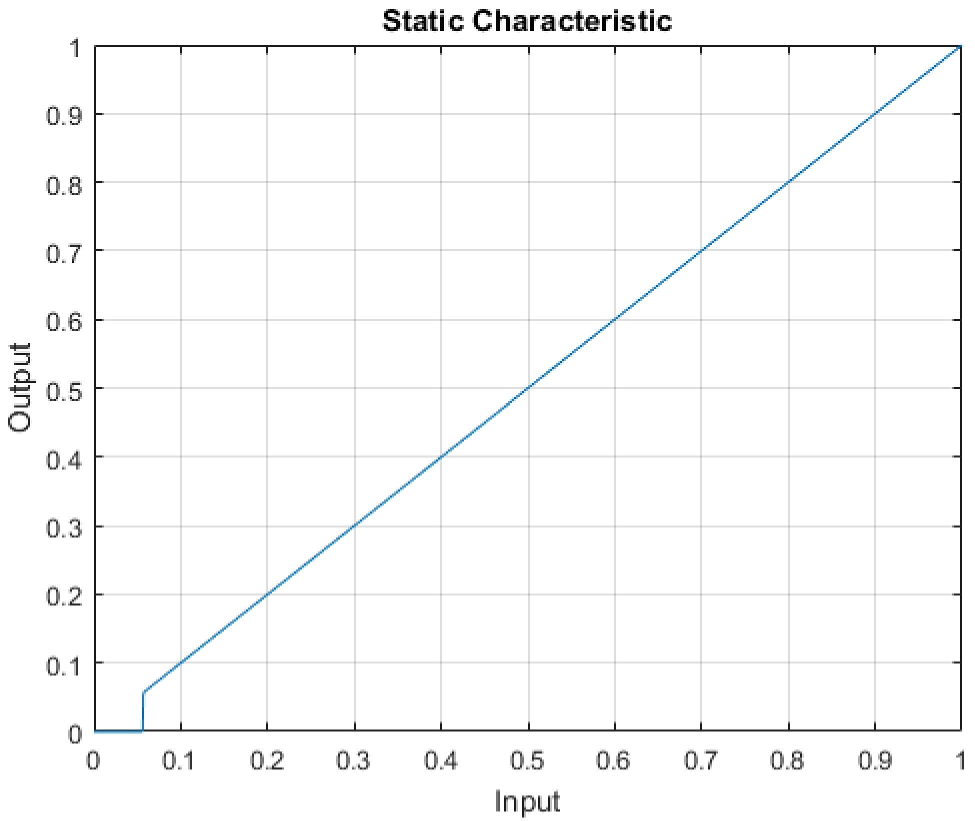
```
while ~isDone(fileReader)
    x = fileReader();
    xCorrupted = x + (1e-2/4)*randn(frameLength,1);
    deviceWriter(xCorrupted);
end

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);
```



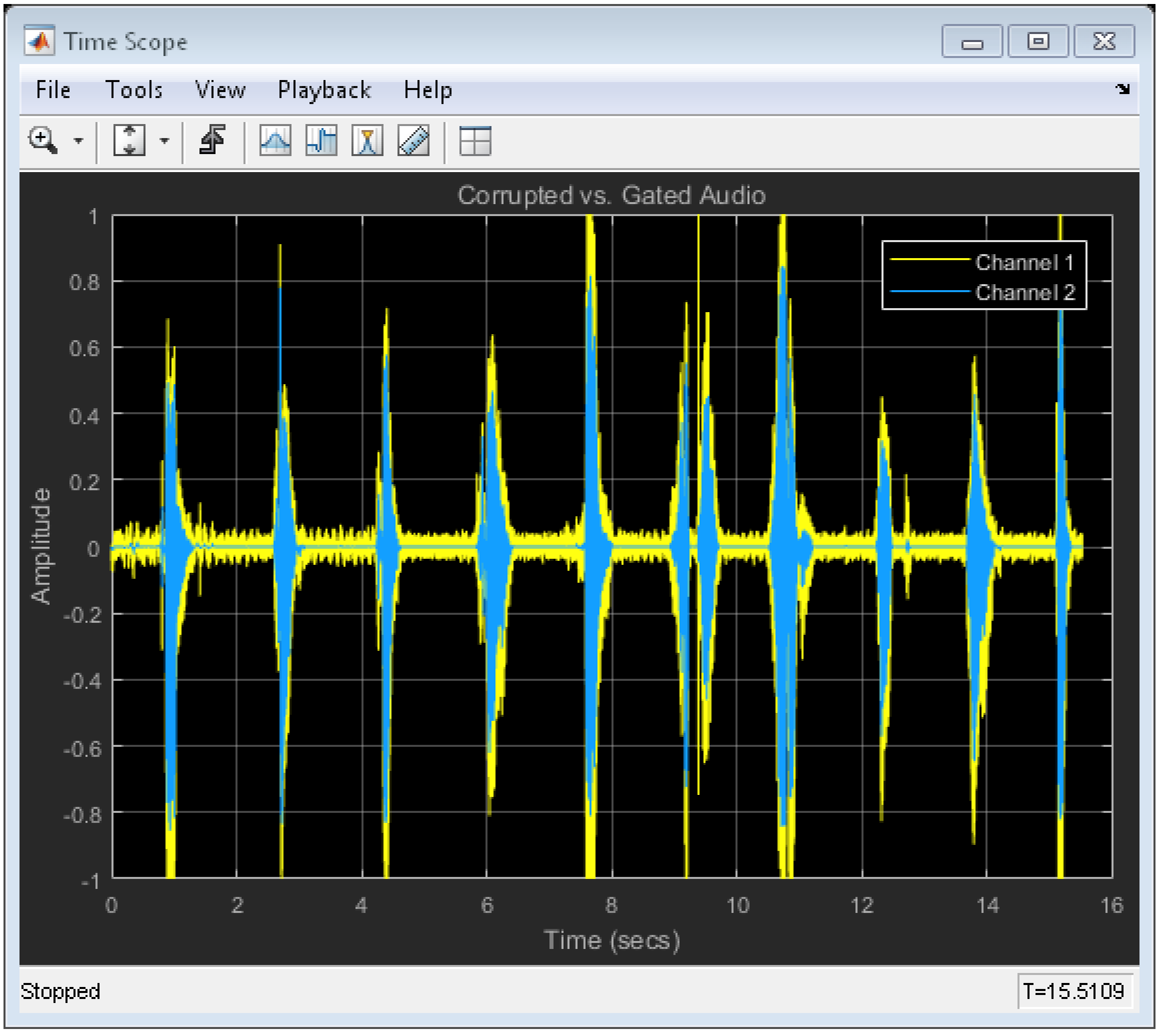
Set up a time scope to visualize the signal before and after dynamic range gating.

```
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.

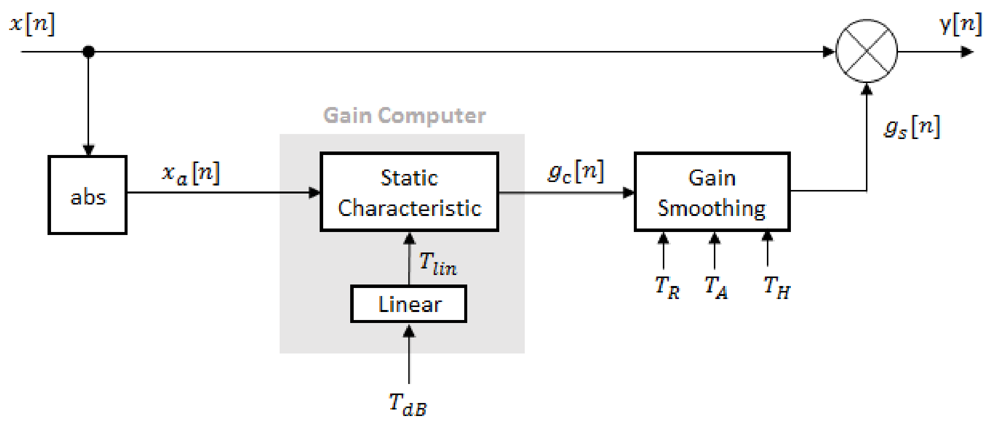
```
while ~isDone(fileReader)
```

```
    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.



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 determine 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 properties:

$$g_s[n] = \begin{cases} \alpha_A g_s[n-1] + (1-\alpha_A) g_c[n] & \text{if } (C_A > T_H)\ \&\ (g_c[n] > g_s[n-1]) \\ g_s[n-1] & \text{if } C_A \leq T_H \\ \alpha_R g_s[n-1] + (1-\alpha_R) g_c[n] & \text{if } (C_R > T_H)\ \&\ (g_c[n] \leq g_s[n-1]) \\ g_s[n-1] & \text{if } C_R \leq T_H \end{cases}$$

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.

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, 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

See Also

Blocks
`Noise Gate`

System Objects
expander | compressor | limiter

Topics
“Dynamic Range Control”

Introduced in R2016a

configureMIDI

System object: noiseGate

Configure MIDI connections between System object and MIDI controller

Syntax

```
configureMIDI(dRG)
configureMIDI(dRG,propName)
configureMIDI(dRG,propName,controlNumber)
configureMIDI(dRG,propName,controlNumber,'DeviceName',deviceName)
```

Description

`configureMIDI(dRG)` starts a MIDI configuration user interface (UI). Use the UI to synchronize tunable properties of the dynamic range gate System object, `dRG`, to MIDI controls of your choice.

`configureMIDI(dRG,propName)` makes the System object property, `propName`, respond to any control on the default MIDI device.

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

`configureMIDI(dRG,propName,controlNumber,'DeviceName',deviceName)` makes the property respond to the MIDI control specified by `controlNumber` on the device specified by `deviceName`.

Each tunable property of the `noiseGate` System object maps to MIDI controls with a specified range.

Property	Range	Unit
`Threshold`	–140 to 0	dB
`AttackTime`	0 to 4	seconds
`ReleaseTime`	0 to 4	seconds

Property	Range	Unit
`HoldTime`	0 to 4	seconds

Introduced in R2016a

createAudioPluginClass

System object: noiseGate

Create audio plugin class that implements functionality of System object

Syntax

```
createAudioPluginClass(dRG)
createAudioPluginClass(dRG,pluginName)
```

Description

`createAudioPluginClass(dRG)` creates a System object plugin that implements the functionality of the `noiseGate` System object, `dRG`. The name of the created class is the `noiseGate` System object variable name followed by `'Plugin'`, for example, `dRGPlugin`.

Note: 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(dRG,pluginName)` specifies the name of your created System object plugin class.

Example: `createAudioPluginClass(dRG,'myGate')` creates a System object plugin with class name `myGate`.

Each tunable property of the `noiseGate` System object maps to a plugin parameter with a default range.

Property	Plugin Parameter Range	Unit
`Threshold`	–140 to 0	dB
`AttackTime`	0 to 4	s
`ReleaseTime`	0 to 4	s

Property	Plugin Parameter Range	Unit
`HoldTime`	0 to 4	s

Introduced in R2016a

disconnectMIDI

System object: noiseGate

Disconnect MIDI controls from System object

Syntax

```
disconnectMIDI(dRG)
```

Description

`disconnectMIDI(dRG)` disconnects MIDI controls from your dynamic range gate, `dRG`. Only those MIDI connections established using `configureMIDI` are disconnected.

Introduced in R2016a

getMIDIConnections

System object: noiseGate

Get MIDI connection information

Syntax

```
connectionInfo = getMIDIConnections(dRG)
```

Description

`connectionInfo = getMIDIConnections(dRG)` returns a structure, `connectionInfo`, containing information about the MIDI connections for your dynamic range gate, `dRG`. Only those MIDI connections established using `configureMIDI` are returned. The `connectionInfo` structure contains a substructure for each tunable property of `dRG` 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).

Introduced in R2016a

reset

System object: noiseGate

Reset internal states of System object

Syntax

`reset(dRG)`

Description

`reset(dRG)` resets internal states of the dynamic range gate, `dRG`, to their initial values.

Introduced in R2016a

step

System object: noiseGate

Perform dynamic range gating

Syntax

```
y = step(dRG,x)
[y,g] = step(dRG,x)
```

Description

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`y = step(dRG,x)` performs dynamic range gating on the input signal, `x`, and returns the gated signal, `y`. The type of dynamic range gating is specified by the algorithm and properties of the `noiseGate` System object, `dRG`.

`x` must be a real-valued, double-precision or single-precision matrix. The System object treats each column of the input as an independent channel.

Note: The System object performs an internal initialization the first time you execute `step`. This initialization locks nontunable properties and input specifications, such as the dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

`[y,g] = step(dRG,x)` also returns the gain, in dB, applied at each input sample.

Introduced in R2016a

visualize

System object: noiseGate

Visualize static gate characteristics of System object

Syntax

```
visualize(dRG)
visualize(dRG,myInputRange)
Y = visualize( ___ )
```

Description

`visualize(dRG)` plots the static gate characteristics of the dynamic range gate, `dRG`. The method computes the dB output level for the input range `[0:0.001:1]` dB. The plot is updated automatically when properties of the object change.

`visualize(dRG,myInputRange)` enables you to specify the input range in dB. Specify `myInputRange` as a vector of ascending values.

`Y = visualize( ___ )` returns the dB output level, `Y`, corresponding to the input range. You can use any of the input arguments from previous syntaxes.

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 filtering on your input:

1. Define and set up your octave-band filter. See "Construction" on page 3-239.
2. Call step to perform octave-band filtering on each channel of the input signal according to the properties of your `octaveFilter` object. The input must be a real-valued, double-precision or single-precision matrix. The `octaveFilter` object treats each column of the input as an independent channel.

Note: Alternatively, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`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

If a property is listed as tunable, then you can change its value even when the object is locked.

`FilterOrder` — Order of octave filter

`6` (default) | even integer

Order of the octave filter, specified as an even integer.

Tunable: No

`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

`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

`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

`SampleRate` — Input sample rate (Hz)

`44100` (default) | positive scalar

Input sample rate in Hz, specified as a positive scalar.

Tunable: Yes

Methods

configureMIDI	Configure MIDI connections between System object and MIDI controller
createAudioPluginClass	Create audio plugin class that implements functionality of System object
disconnectMIDI	Disconnect MIDI controls from System object
getANSICenterFrequencies	Get the list of valid ANSI S1.11-2004 center frequencies
getFilter	Return biquad filter object using octave filter design
getMIDIConnections	Get MIDI connection information
isStandardCompliant	Verify filter design is ANSI S1.11-2004 compliant
reset	Reset internal states of System object

step	Apply octave-band filtering
visualize	Visualize and validate filter response

Common to All System Objects	
`clone`	Create System object with same property values
`getNumInpu`	Expected number of inputs to a System object
`getNumOutp`	Expected number of outputs of a System object
`isLocked`	Check locked states of a System object (logical)
`release`	Allow System object property value changes

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 an audio file reader System object.

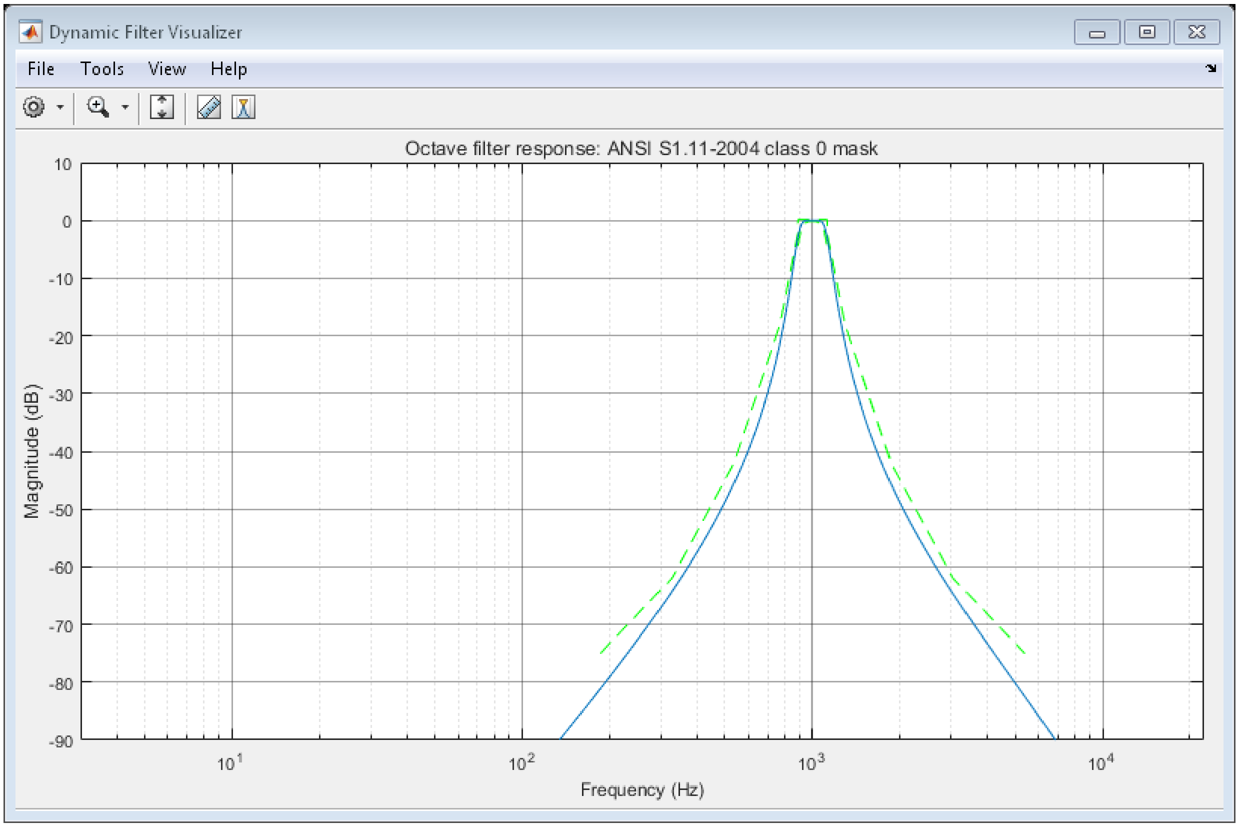
```
samplesPerFrame = 1024;
reader = dsp.AudioFileReader('Filename',...
    'RockGuitar-16-44p1-stereo-72secs.wav',...
    'SamplesPerFrame',samplesPerFrame,...
    'PlayCount',Inf);
```

Create an octave filter 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');
```

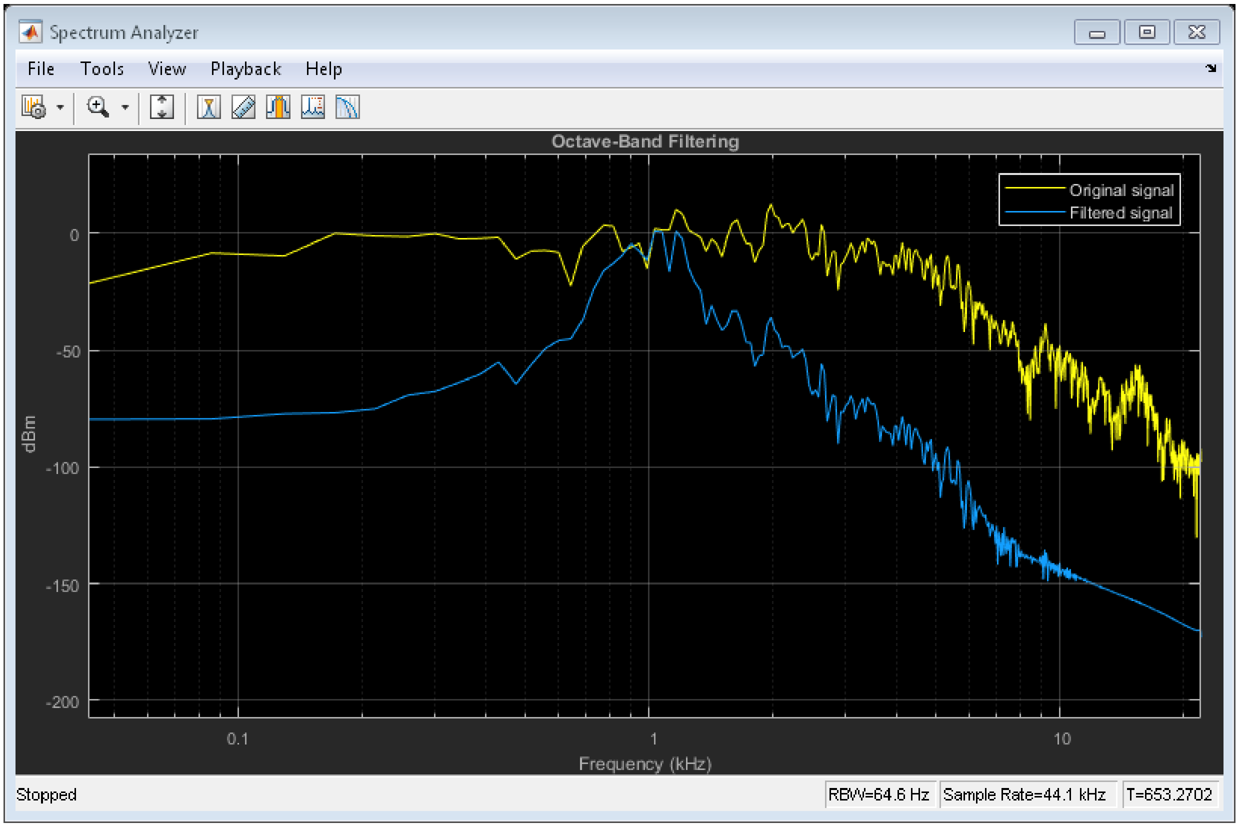


Create a spectrum analyzer to visualize the original audio signal and the audio signal after octave-band filtering.

```
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);
```

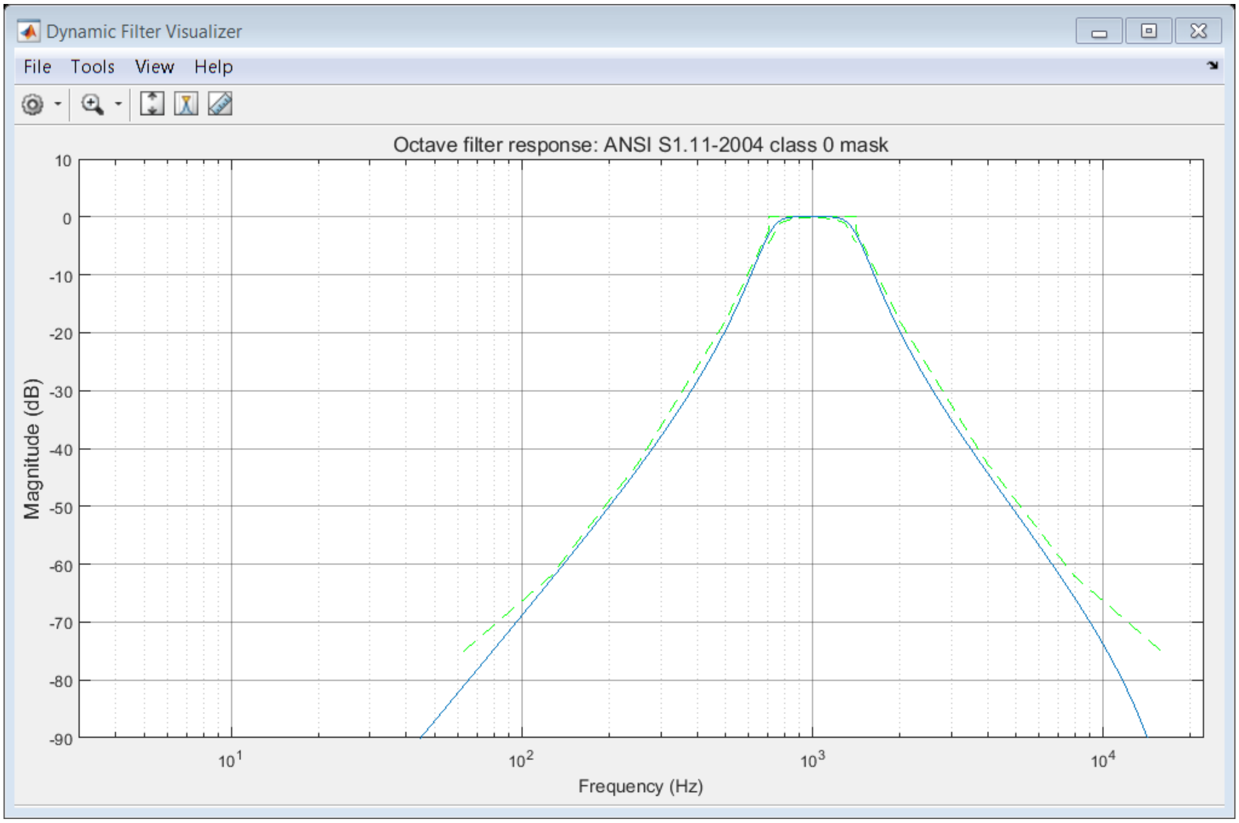


Create Octave-Band Filter Bank

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');
```

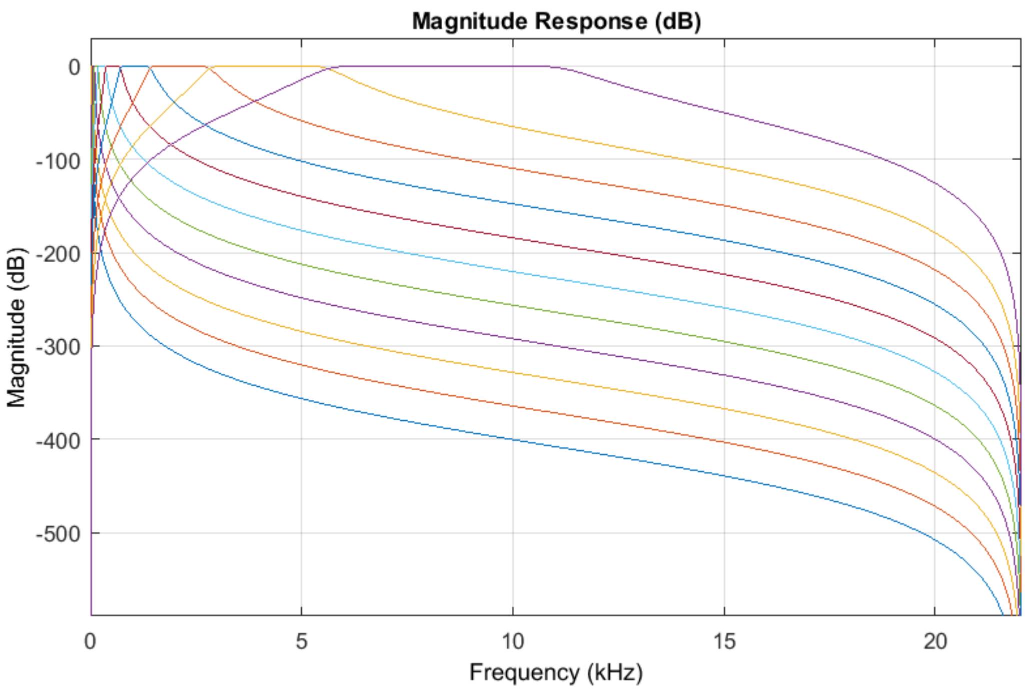


Get a vector of valid center frequencies, given the center frequency of `octFilt`. Create an octave filter bank using the valid center frequencies.

```
centerFrequencies = getANSICenterFrequencies(octFilt);
for i = 1:11
    octaveFilterBank{i} = octaveFilter(centerFrequencies(i),'FilterOrder',12);
end
```

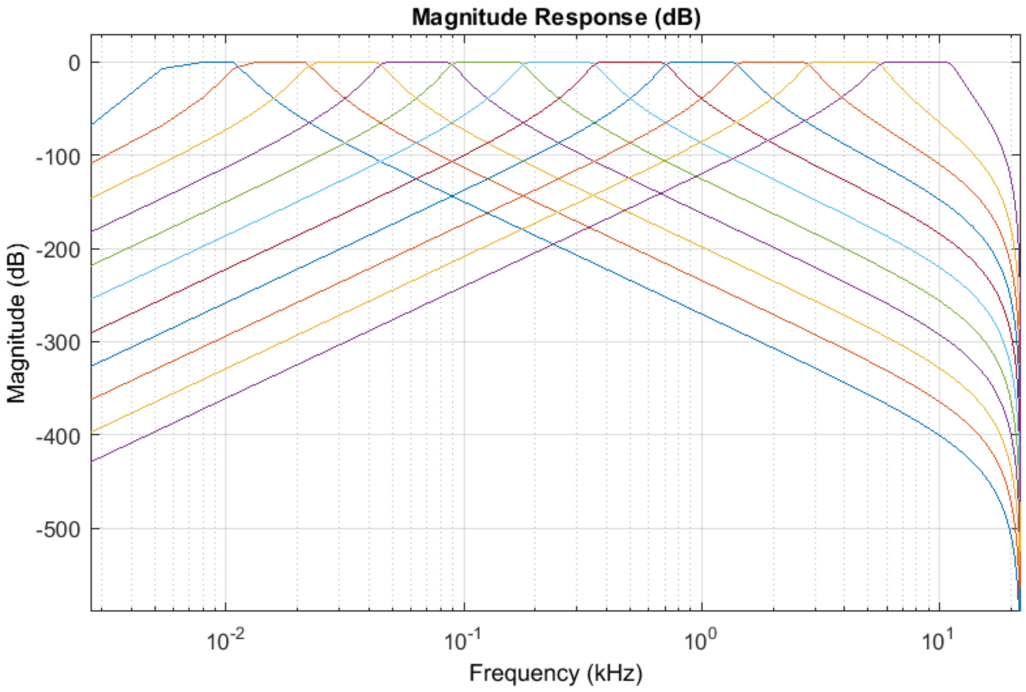
Use `getFilter` to return biquad filter objects for each filter in your octave filter bank. Visualize the octave filter bank with a linear frequency scale.

```
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();
```

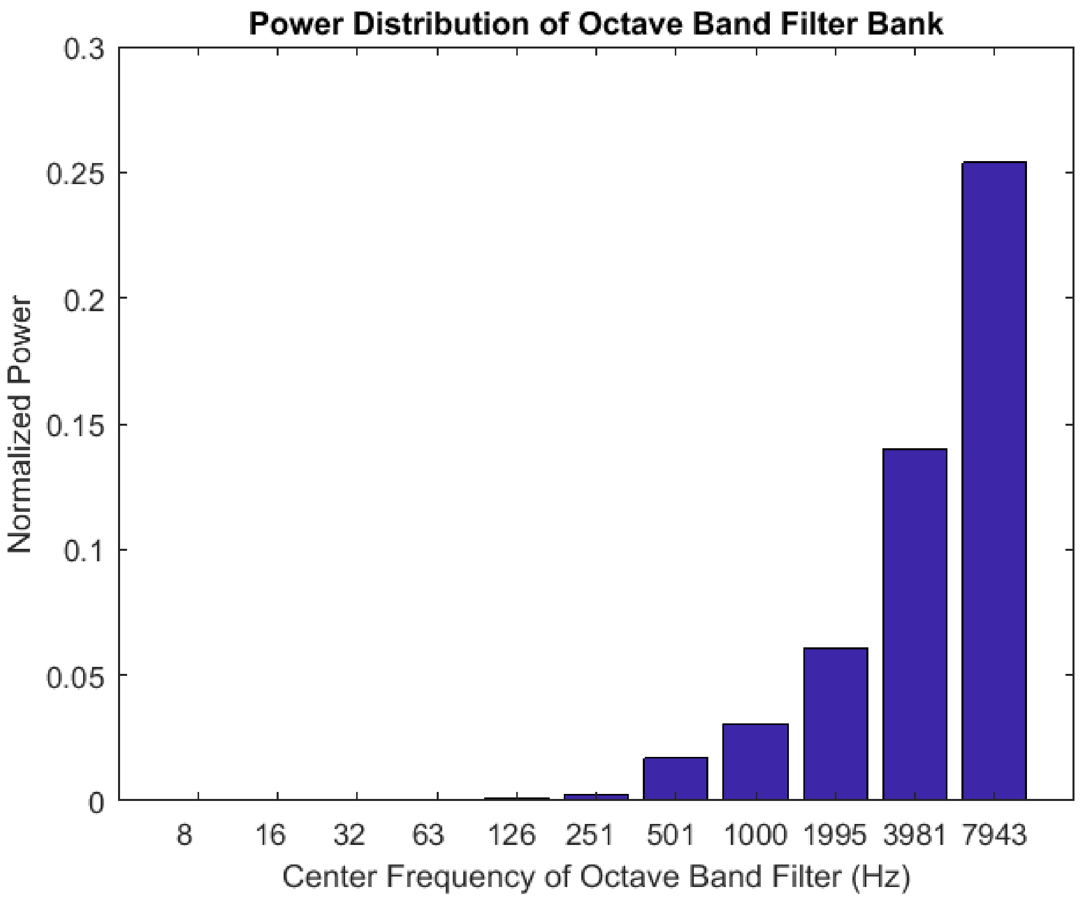
Pass the white noise signal through the octave-band filter bank.

```
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 bandpower increases by approximately a factor of two, because the octave bandwidth increases by a factor of two. The power distribution of an octave filter bank mimics how higher frequencies are percieved louder in white noise. You can use octave filter banks to weight a spectrum for percieved 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.

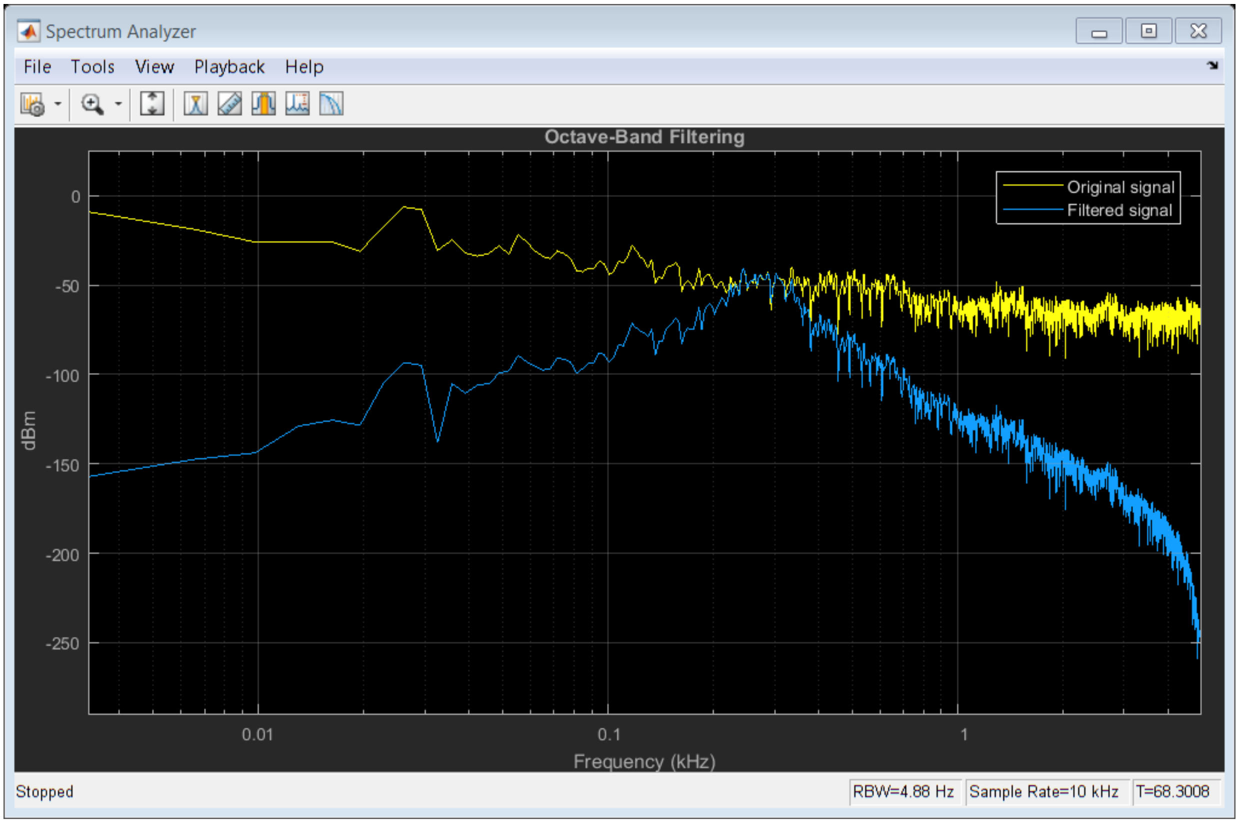
```
fileReader = dsp.AudioFileReader('Counting-16-44p1-mono-15secs.wav');
deviceWriter = audioDeviceWriter('SampleRate',fileReader.SampleRate);
```

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` objects with default properties. Create an `octaveFilter` 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.

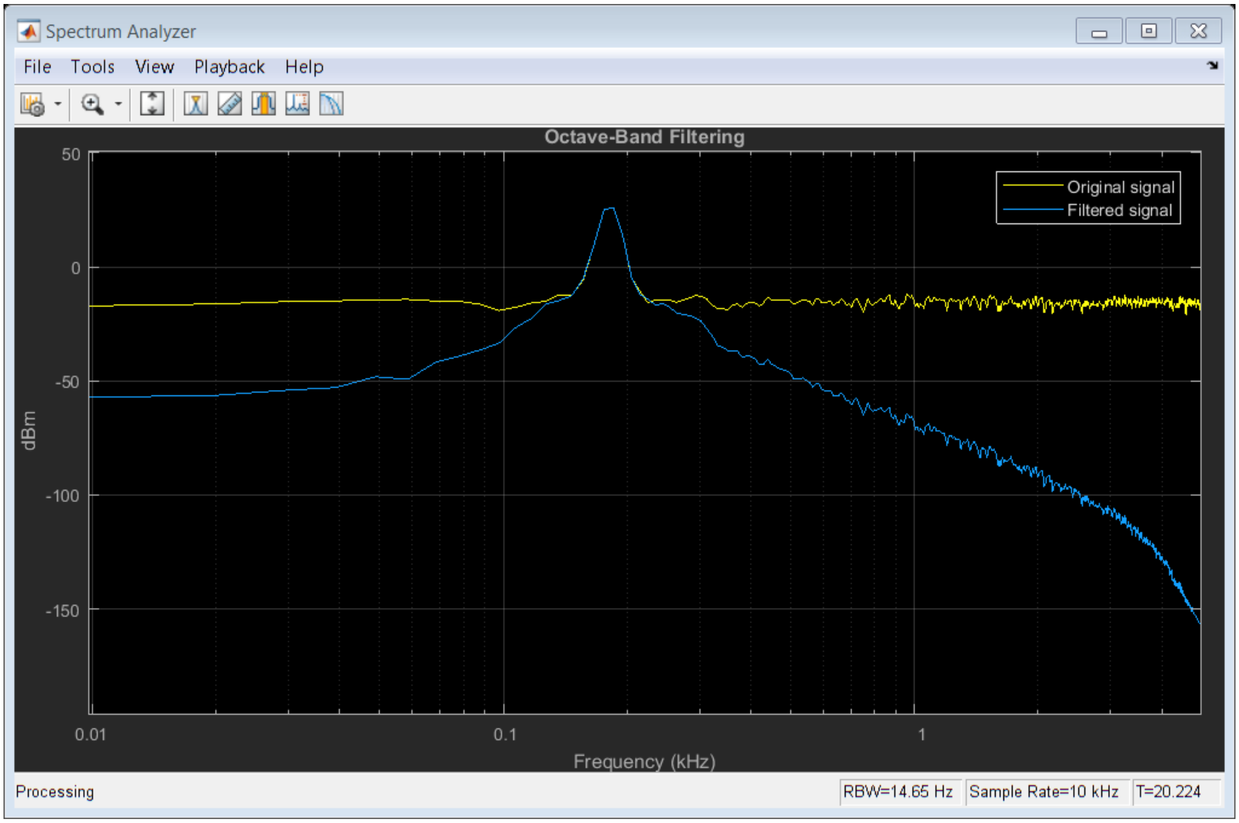
```
scope = dsp.SpectrumAnalyzer(...
    'PlotAsTwoSidedSpectrum',false,...
    'FrequencyScale','Log',...
    'FrequencyResolutionMethod','WindowLength',...
    'Title','Octave-Band Filtering',...
    'ShowLegend',true,...
    'SpectralAverages',10,...
    'ChannelNames',{'Original signal','Filtered 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);
```



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.

Algorithm

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].

First the object normalizes the specified center frequency:

$$f_c = \frac{2 \times CenterFrequency}{SampleRate}$$

Then the object computes the band edge frequencies:

$$f_{pa} = f_c \times G^{-1/2b}$$

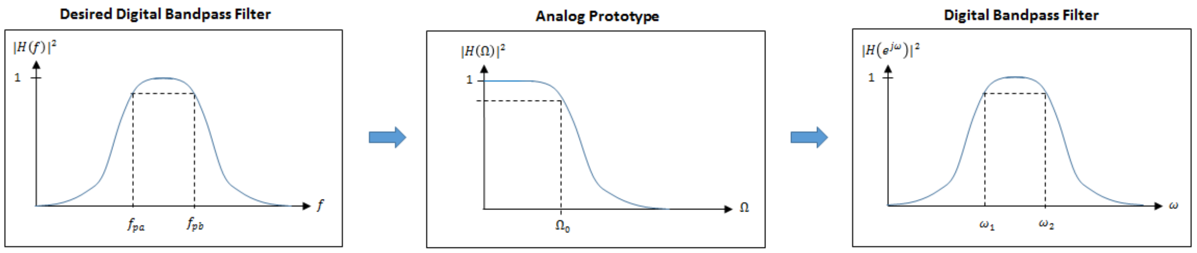
$$f_{pb} = f_c \times G^{1/2b}$$

- 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:

$$H(s) = H_0(s)H_1(s)H_2(s)\cdots H_K(s)\text{, where:}$$

- $$H_0(s) = \begin{cases} 1, & \text{if} \quad N = 2K \\ \dfrac{1}{1+\dfrac{1}{\Omega_0}}, & \text{if} \quad N = 2K+1 \end{cases}$$

- $$H_i(s) = \frac{1}{1-2\dfrac{s}{\Omega_0}\cos\theta_i + \dfrac{s^2}{\Omega_0^2}}, \quad i = 1,2,\ldots,K$$

- $$\theta_i = \frac{\pi}{2N}(N-1+2i), \quad i = 1,2,\ldots,N,\ldots,2N$$

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:

$$\Omega_0 = \frac{c - \cos\omega_{pb}}{\sin\omega_{pb}}$$

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, 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

See Also

Blocks
`Octave Filter`

System Objects
multibandParametricEQ | weightingFilter | dsp.BiquadFilter

Topics
“Octave-Band and Fractional Octave-Band Filters”

Introduced in R2016b

configureMIDI

System object: octaveFilter

Configure MIDI connections between System object and MIDI controller

Syntax

```
configureMIDI(octFilt)
configureMIDI(octFilt,propName)
configureMIDI(octFilt,propName,controlNumber)
configureMIDI(octFilt,propName,controlNumber,'DeviceName',
deviceName)
```

Description

`configureMIDI(octFilt)` starts a MIDI configuration user interface (UI). Use the UI to synchronize tunable properties of the octave filter System object, `octFilt`, to MIDI control of your choice.

`configureMIDI(octFilt,propName)` makes the System object property, `propName`, respond to any control on the default MIDI device.

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

`configureMIDI(octFilt,propName,controlNumber,'DeviceName',deviceName)` makes the property respond to the MIDI control specified by `controlNumber` on the device specified by `deviceName`.

Each tunable property of the `octaveFilter` System object maps to MIDI controls with a specified range.

Property	Range	Mapping
`CenterFrequency`	3 Hz to 22 kHz	log

Property	Range	Mapping
`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.

Introduced in R2016b

createAudioPluginClass

System object: octaveFilter

Create audio plugin class that implements functionality of System object

Syntax

```
createAudioPluginClass(octFilt)
createAudioPluginClass(octFilt,pluginName)
```

Description

`createAudioPluginClass(octFilt)` creates a System object plugin that implements the functionality of the `octaveFilter` System object, `octFilt`. The name of the created class is the `octaveFilter` System object variable name followed by `'Plugin'`, for example, `octFiltPlugin`.

Note: 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(octFilt,pluginName)` specifies the name of your created System object plugin class.

Example: `createAudioPluginClass(octFilt,'octaveBandFilter')` creates a System object plugin with class name `octaveBandFilter`.

Each tunable property of the `octaveFilter` System object maps to a plugin parameter with a default range.

Property	Range	Mapping
`CenterFrequency`	3 Hz to 22 kHz	log

Property	Range	Mapping
`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'`	enum

Introduced in R2016b

disconnectMIDI

System object: octaveFilter

Disconnect MIDI controls from System object

Syntax

```
disconnectMIDI(octFilt)
```

Description

`disconnectMIDI(octFilt)` disconnects MIDI controls from your octave filter, `octFilt`. Only those MIDI connections established using `configureMIDI` are disconnected.

Introduced in R2016b

getANSICenterFrequencies

System object: octaveFilter

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.

The range for computing valid center frequencies is 3 Hz to (Fs/2) Hz, where the `SampleRate` property of your octave filter defines Fs.

Introduced in R2016b

getFilter

System object: octaveFilter

Return biquad filter object using octave filter design

Syntax

```
biquad = getFilter(octFilt)
```

Description

`biquad = getFilter(octFilt)` returns a `dsp.BiquadFilter` object, `biquad`. The `SOSMatrix` and `ScaleValues` properties of the biquad filter object are set as specified by the `octaveFilter` object, `octFilt`.

Use `getFilter` for the design capabilities of the `octaveFilter` System object and the processing capabilities of the `dsp.BiquadFilter` System object.

Introduced in R2016b

getMIDIConnections

System object: octaveFilter

Get MIDI connection information

Syntax

```
connectionInfo = getMIDIConnections(octFilt)
```

Description

`connectionInfo = getMIDIConnections(octFilt)` returns a structure, `connectionInfo`, containing information about the MIDI connections for your octave filter, `octFilt`. Only those MIDI connections established using `configureMIDI` are returned. The `connectionInfo` structure contains a substructure for each tunable property of `octFilt` 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).

Introduced in R2016b

isStandardCompliant

System object: octaveFilter

Verify 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 `octFilt` filter design is compliant with the ANSI S1.11-2004 standard for `classType`. Specify `classType` as `'class 0'`, `'class 1'`, or `'class 2'`.

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.

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

Introduced in R2016b

reset

System object: octaveFilter

Reset internal states of System object

Syntax

```
reset(octFilt)
```

Description

`reset(octFilt)` resets internal states of the octave filter, `octFilt`, to their initial values.

Introduced in R2016b

step

System object: octaveFilter

Apply octave-band filtering

Syntax

```
y = step(octFilt,x)
```

Description

Note: Alternatively, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`y = step(octFilt,x)` applies octave-band filtering to the input signal, `x`, and returns the filtered signal, `y`. The type of filtering is specified by the algorithm and properties of the `octaveFilter` System object, `octFilt`.

`x` must be a real-valued, double-precision or single-precision matrix. The System object treats each column of the input as an independent channel.

Note: The System object performs an internal initialization the first time you execute `step`. This initialization locks nontunable properties and input specifications, such as the dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

Introduced in R2016b

visualize

System object: octaveFilter

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. The default is `2048`.

`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.

Introduced in R2016b

reverberator System object

Add reverberation to audio signal

Description

The `reverberator` System object adds reverberation to mono or stereo audio signals. Properties of the `reverberator` System object specify the “Reverberation Model” on page 3-275 used.

To add reverberation to your input:

1. Define and set up your reverberator. See “Construction” on page 3-270.
2. Call step to add reverberation to the input signal according to the properties of your `reverberator` object. The input must be a real-valued, double-precision or single-precision matrix. The input matrix must have one or two columns, corresponding to a mono or stereo signal, respectively. The output matrix always has two columns (stereo).

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`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 the `PreDelay` property set to `0.5` and the `WetDryMix` property set to `1`.

Properties

If a property is listed as tunable, then you can change its value even when the object is locked.

`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 to 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

`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

`Diffusion` — Density of reverb tail

`0.5` (default) | real positive scalar

Density of reverb tail, specified as a real positive 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

`DecayFactor` — Decay factor of reverb tail

`0.5` (default) | real positive scalar

Decay factor of reverb tail, specified as a real positive scalar in the range 0 to 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

`HighFrequencyDamping` — High-frequency damping

`0.0005` (default) | real scalar

High-frequency damping, specified as a real positive scalar in the range 0 to 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

`WetDryMix` — Wet-dry mix

`0.3` (default) | real scalar

Wet-dry mix, specified as a real positive scalar in the range 0 to 1.

Wet-dry mix is the ratio of wet (reverberated) to dry (original) signal that your `reverberator` System object outputs.

Tunable: Yes

`SampleRate` — Input sample rate (Hz)

`44100` (default) | positive scalar

Input sample rate in Hz, specified as a positive scalar.

Tunable: Yes

Methods

configureMIDI	Configure MIDI connections between System object and MIDI controller
createAudioPluginClass	Create audio plugin class that implements functionality of System object

disconnectMIDI	Disconnect MIDI controls from System object
getMIDIConnections	Get MIDI connection information
reset	Reset internal states of System object
step	Add artificial reverberation

Common to All System Objects	
`clone`	Create System object with same property values
`getNumInpu`	Expected number of inputs to a System object
`getNumOutp`	Expected number of outputs of a System object
`isLocked`	Check locked states of a System object (logical)
`release`	Allow System object property value changes

Examples

Add Reverberation to Audio Signal

Use the `reverberator` System object™ to add artificial reverberation to an audio signal read from a file.

Construct the audio file reader and audio device writer 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
```

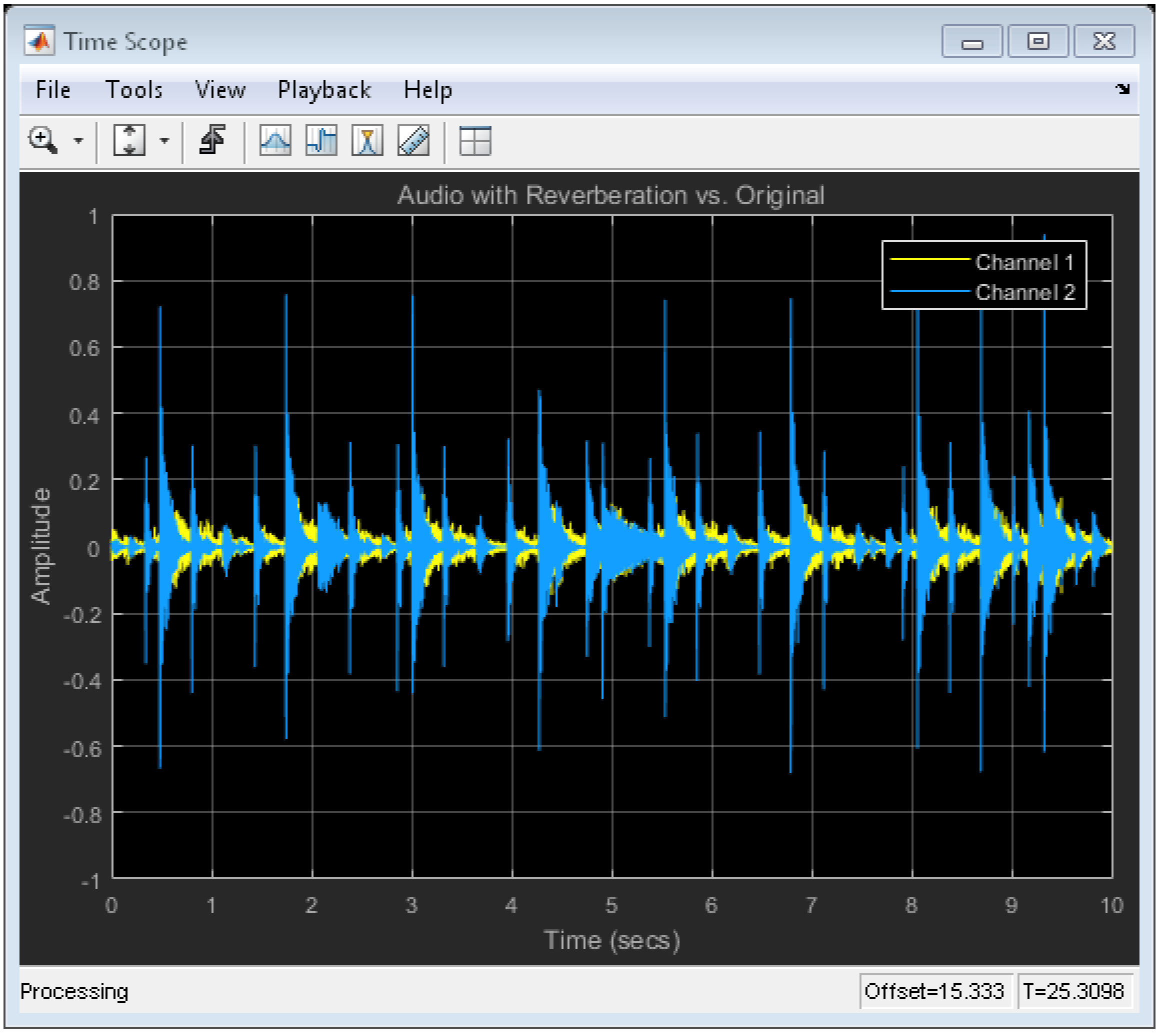
```
reverb = 

  reverberator with properties:

                PreDelay: 0
        HighCutFrequency: 20000
               Diffusion: 0.5000
             DecayFactor: 0.5000
    HighFrequencyDamping: 5.0000e-04
               WetDryMix: 0.3000
              SampleRate: 44100
```

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)
```



Definitions

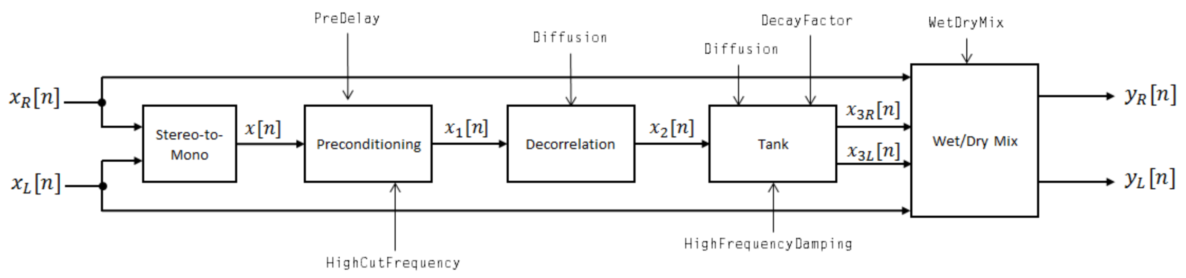
Reverberation Model

Reverberation refers to the buildup and decay of reflected audio waves in a given space. Reverberation models are used in digital environments to mimic the physical effect of reverberation.

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.



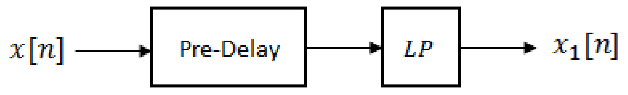
The description for the algorithm that follows is for a stereo input. A mono input is a simplified case.

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 `PreDelay` property determines the value of k.
- The signal is fed through a single-pole lowpass filter with transfer function

$$LP(z) = \frac{1-\alpha}{1-\alpha z^{-1}},$$

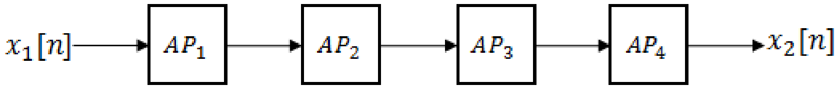
where

$$\alpha = \exp\left(-2\pi \times \frac{f_c}{f_s}\right).$$

- 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

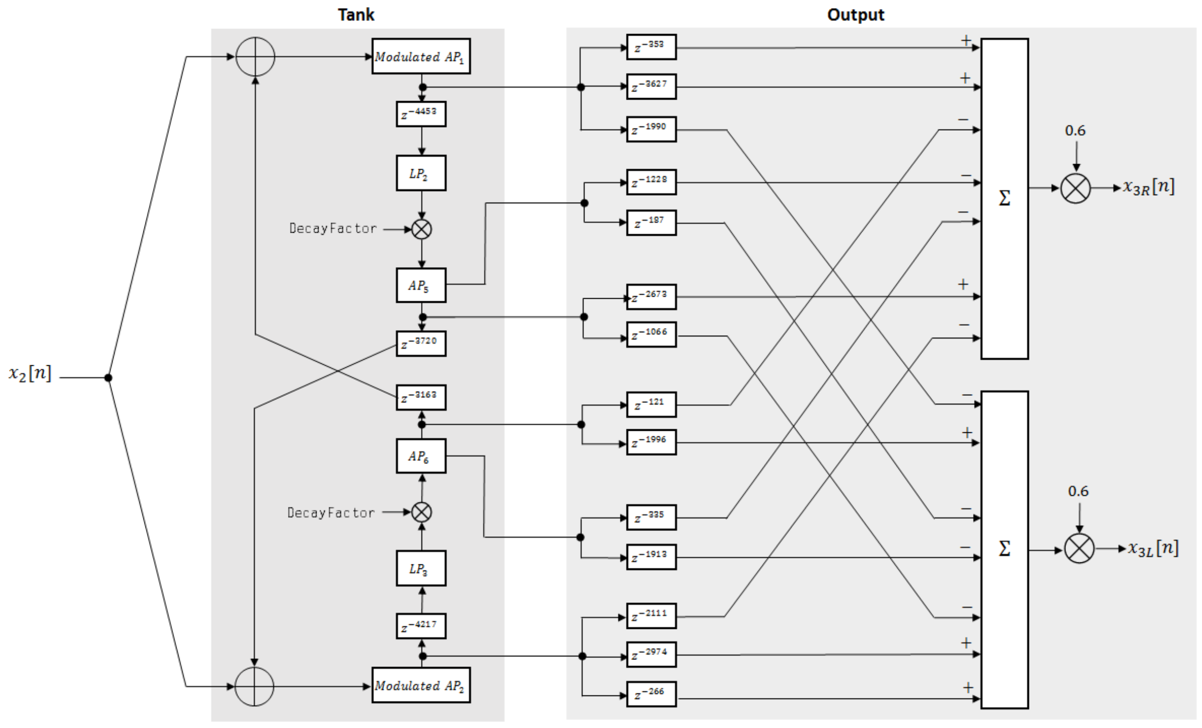
$$AP(z) = \frac{\beta + z^{-k}}{1+\beta z^{-k}},$$

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, 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, 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

See Also

Blocks
`Reverberator`

Introduced in R2016a

configureMIDI

System object: reverberator

Configure MIDI connections between System object and MIDI controller

Syntax

```
configureMIDI(reverb)
configureMIDI(reverb,propName)
configureMIDI(reverb,propName,controlNumber)
configureMIDI(reverb,propName,controlNumber,'DeviceName',deviceName)
```

Description

`configureMIDI(reverb)` starts a MIDI configuration user interface (UI). Use the UI to synchronize tunable properties of the `reverberator` System object, `reverb`, to MIDI controls of your choice.

`configureMIDI(reverb,propName)` makes the System object property, `propName`, respond to any control on the default MIDI device.

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

`configureMIDI(reverb,propName,controlNumber,'DeviceName',deviceName)` makes the property respond to the MIDI control specified by `controlNumber` on the device specified by `deviceName`.

Each tunable property of the `reverberator` System object maps to MIDI controls with a specified range.

Property	Range	Unit
`PreDelay`	0 to 1	s
`HighCutFrequency`	20 to 20,000 (log scale)	Hz
`Diffusion`	0 to 1	none

Property	Range	Unit
`DecayFactor`	0 to 1	none
`HighFrequencyDamping`	0 to 1	none
`WetDryMix`	0 to 1	none

Introduced in R2016a

createAudioPluginClass

System object: reverberator

Create audio plugin class that implements functionality of System object

Syntax

```
createAudioPluginClass(reverb)
createAudioPluginClass(reverb,pluginName)
```

Description

`createAudioPluginClass(reverb)` creates a System object plugin that implements the functionality of the `reverberator` System object, `reverb`. The name of the created class is the `reverberator` System object variable name followed by `'Plugin'`, for example, `reverbPlugin`.

Note: If the object is locked, the number of input channels of the plugin is equal to the number of channels of the object. Otherwise, the number of channels is equal to 2. The number of output channels of the plugin is always equal to 2.

`createAudioPluginClass(reverb,pluginName)` specifies the name of your created System object plugin class.

Example: `createAudioPluginClass(reverb,'concertHall')` creates a System object plugin with class name `concertHall`.

Each tunable property of the `reverberator` System object maps to a plugin parameter with a default range.

Property	Plugin Parameter Range	Unit
`PreDelay`	0 to 1	s
`HighCutFrequency`	20 to 20,000 (log scale)	Hz
`Diffusion`	0 to 1	none

Property	Plugin Parameter Range	Unit
`DecayFactor`	0 to 1	none
`HighFrequencyDamping`	0 to 1	none
`WetDryMix`	0 to 1	none

Introduced in R2016a

disconnectMIDI

System object: reverberator

Disconnect MIDI controls from System object

Syntax

```
disconnectMIDI(reverb)
```

Description

`disconnectMIDI(reverb)` disconnects MIDI controls from your reverberator, `reverb`. Only those MIDI connections established using `configureMIDI` are disconnected.

Introduced in R2016a

getMIDIConnections

System object: reverberator

Get MIDI connection information

Syntax

```
connectionInfo = getMIDIConnections(reverb)
```

Description

`connectionInfo = getMIDIConnections(reverb)` returns a structure, `connectionInfo`, containing information about the MIDI connections for your reverberator, `reverb`. Only those MIDI connections established using `configureMIDI` are returned. The `connectionInfo` structure contains a substructure for each tunable property of `reverb` 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).

Introduced in R2016a

reset

System object: reverberator

Reset internal states of System object

Syntax

```
reset(reverb)
```

Description

`reset(reverb)` resets internal states of the reverberator, `reverb`, to their initial values.

Introduced in R2016a

step

System object: reverberator

Add artificial reverberation

Syntax

```
y = step(reverb,x)
```

Description

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`y = step(reverb,x)` adds artificial reverberation to the input signal, `x`, and returns the mixed signal, `y`. The type of reverberation is specified by the algorithm and properties of the `reverberator` System object, `reverb`.

`x` must be a real-valued, double-precision or single-precision matrix with one or two columns. The output is always a stereo signal (two columns).

Note: The System object performs an internal initialization the first time you execute `step`. This initialization locks nontunable properties and input specifications, such as the dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

Introduced in R2016a

wavetableSynthesizer System object

Generate periodic signals 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. Define and set up your wavetable synthesizer. See "Construction" on page 3-289.
2. Call step to generate a signal according to the properties of your `wavetableSynthesizer` object. The object has internal memory suited to frame-based processing.

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`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 sine waveforms using the default `Wavetable` (a single-cycle sinusoid), with `Amplitude` set to `2` and `DCOffset` set to 2.5.

Properties

If a property is listed as tunable, then you can change its value even when the object is locked.

`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.

Tunable: 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.

`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

`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 DC offset is applied.

Tunable: Yes

`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 to 1. The range is a normalized 2π radians interval.

Tunable: No

`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

`SamplesPerFrame` — Number of samples per frame

`512` (default) | positive integer

Number of samples per frame, specified as a positive integer in the range 1 to 192,000.

This property determines the vector length that the `step` method of your `wavetableSynthesizer` object outputs.

Tunable: Yes

`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

Methods

configureMIDI	Configure MIDI connections between System object and MIDI controller

createAudioPluginClass	Create audio plugin class that implements functionality of System object
disconnectMIDI	Disconnect MIDI controls from System object
getMIDIConnections	Get MIDI connection information
reset	Reset internal states of System object
step	Generate periodic signals from single-cycle waveforms

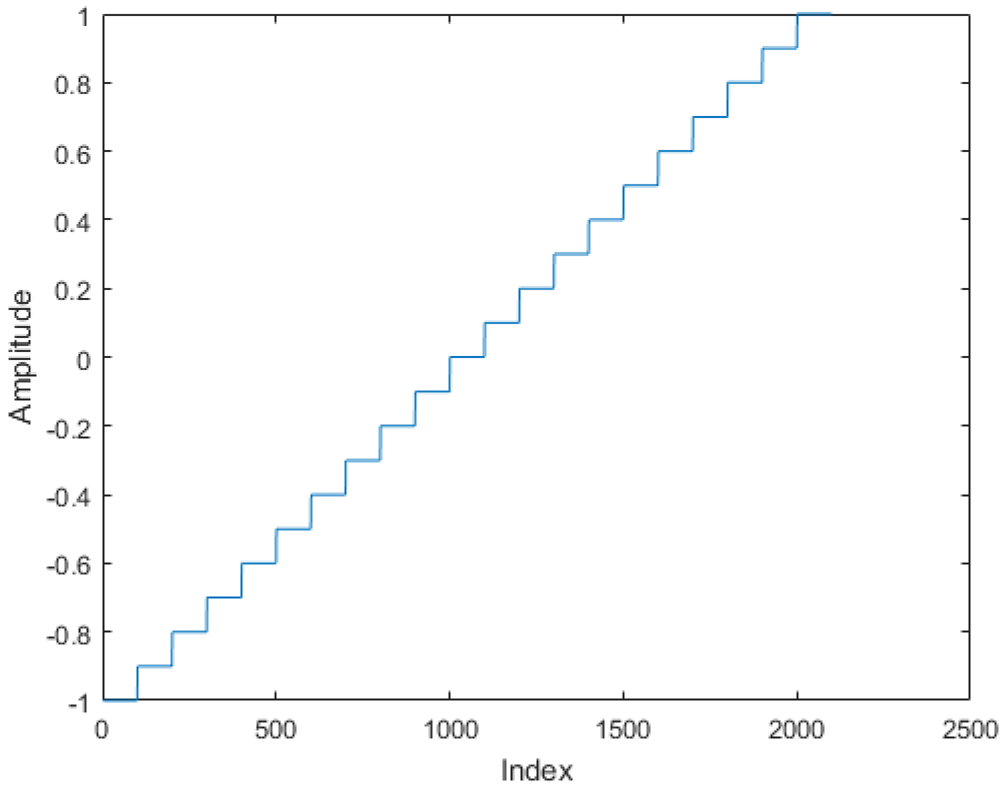
Common to All System Objects	
`clone`	Create System object with same property values
`getNumInpu`	Expected number of inputs to a System object
`getNumOutp`	Expected number of outputs of a System object
`isLocked`	Check locked states of a System object (logical)
`release`	Allow System object property value changes

Examples

Generate Variable-Frequency Staircase Wave

Define and plot a single-cycle waveform.

```
values = -1:.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.

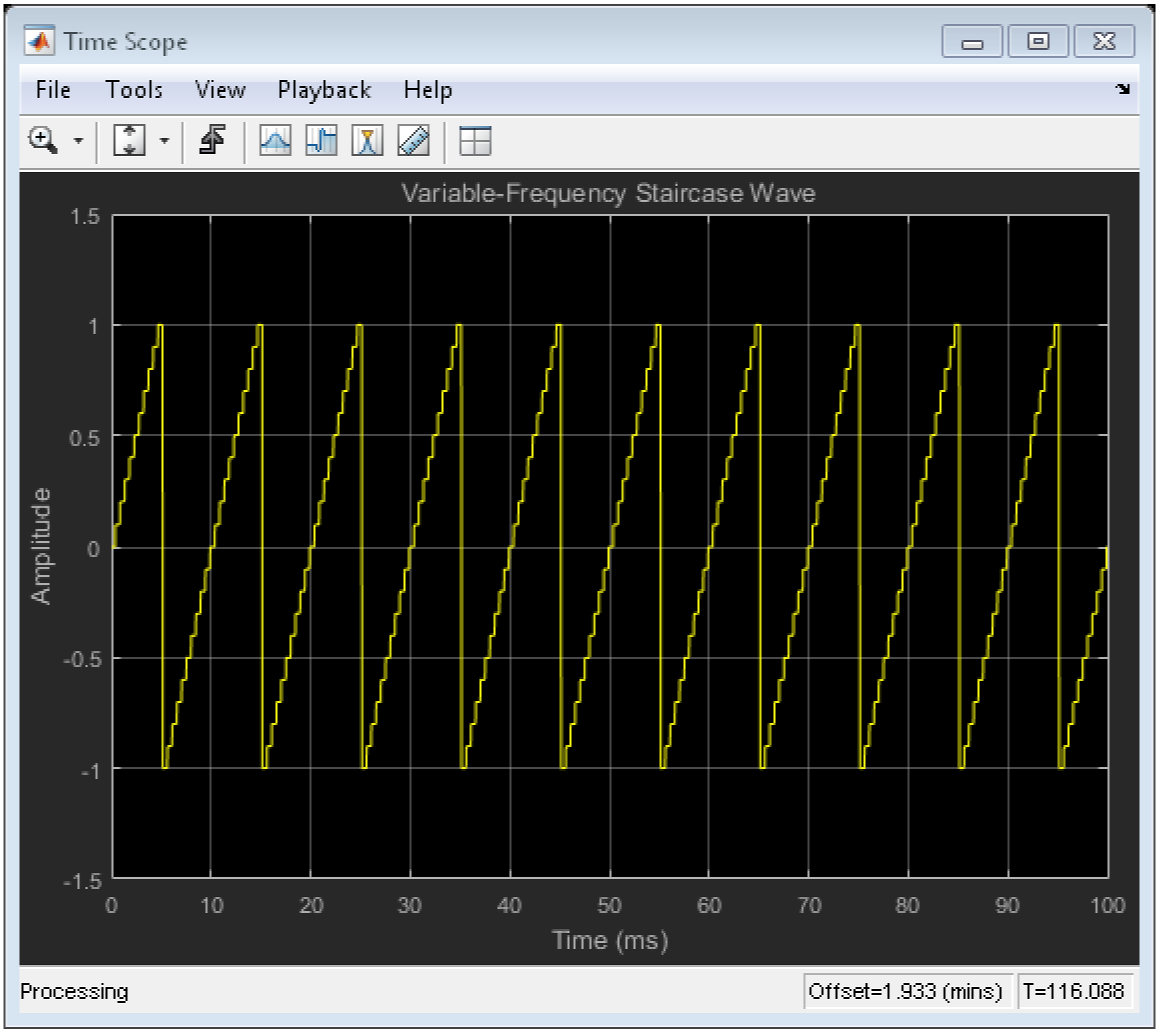
```
waveSynth = wavetableSynthesizer(singleCycleWave,10);
```

Create a time scope to visualize the staircase wave generated by `waveSynth`.

```
scope = dsp.TimeScope(...
    'SampleRate',waveSynth.SampleRate,...
    'TimeSpan',.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 listen to 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 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 listen to the resulting audio.

Create a single-cycle waveform for the `wavetableSynthesizer` to index into. Create a wavetable synthesizer object.

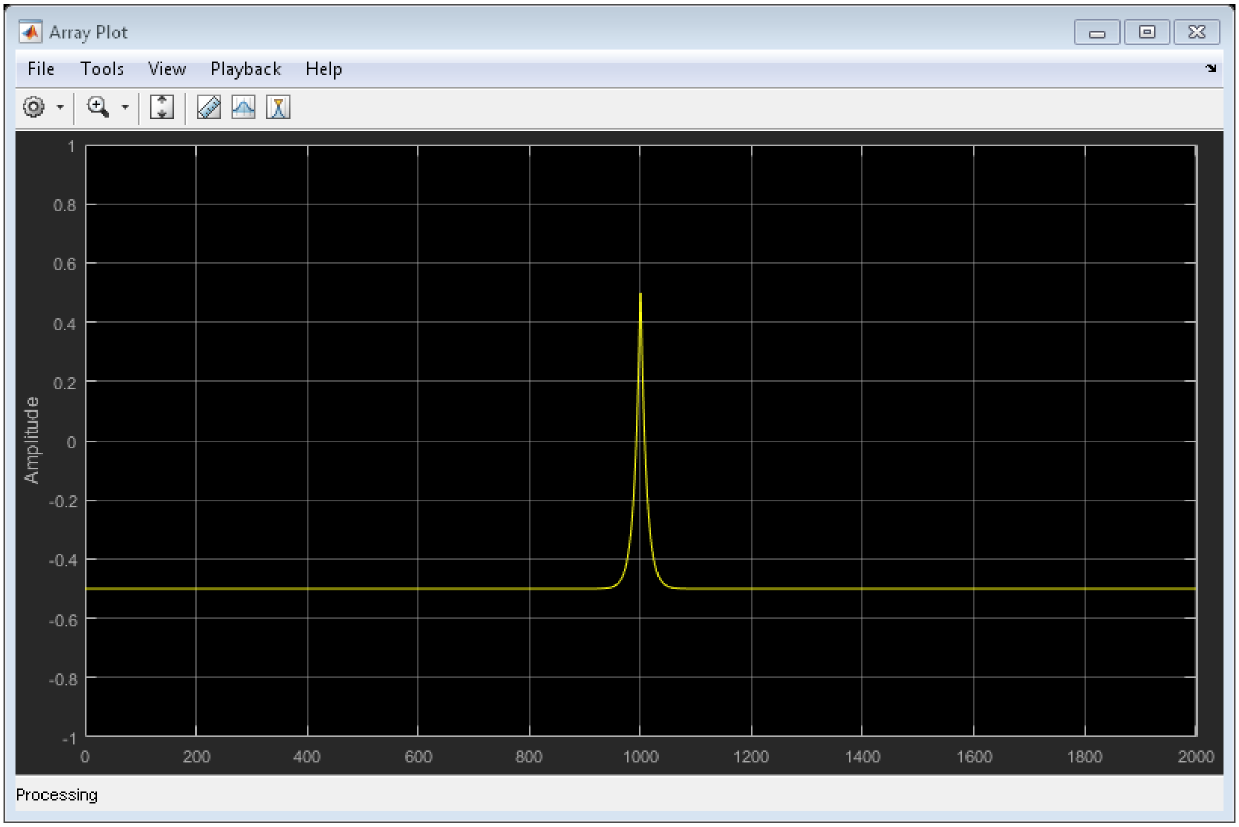
```
t = 0:0.001:1;
exponent = 5;
waveTable = [t.^exponent,fliplr(t.^exponent)] - 0.5;

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;
```

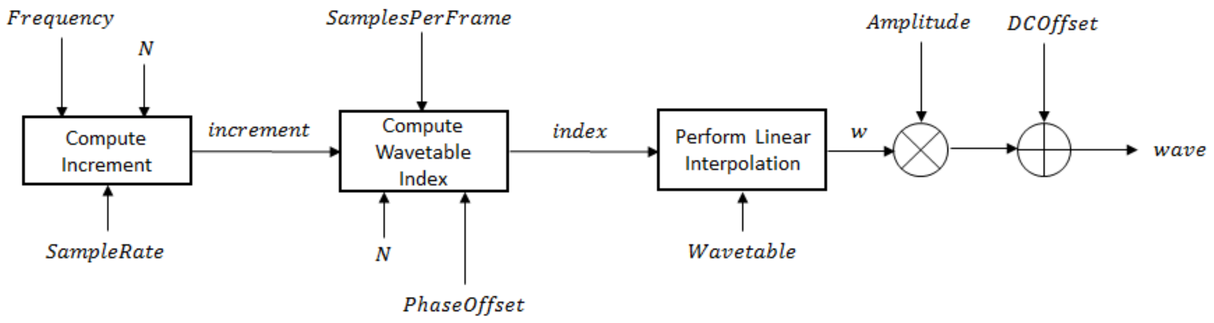
In an audio stream loop, incrementally modify the `Wavetable` property of the wavetable synthesizer and plot it. Call the synthesizer to output a waveform and play the waveform to your audio device.

```
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 wavetable synthesizer System object synthesizes periodic signals using a cached single-cycle waveform, specified waveform properties, and phase memory.



1 Compute the increment step size,

$$increment = \frac{Frequency}{SampleRate} \times N\,,$$

where N is the number of elements in your wavetable.

2 Compute `Wavetable` index,

$$index[n] = \begin{cases} index[n-1] + increment & if \;\; index[n-1] < N \\ index[n-1] + increment - N & else \end{cases},$$

for $2 \le n \le SamplesPerFrame$. The `PhaseOffset` property determines *index*[*n*=*1*].

3 Index into the `Wavetable` and perform linear interpolation:

$$w = \begin{cases} \left(Wavetable[1] - Wavetable[index_{low}]\right) \times fraction + Wavetable[index_{low}] & if \;\; index_{high} \\ \left(Wavetable[index_{high}] - Wavetable[index_{low}]\right) \times fraction + Wavetable[index_{low}] & else \end{cases}$$

- $index_{low} = floor(index[n] + 1)$
- $index_{high} = index_{low} + 1$
- $fraction = index - floor(index)$

4 Multiply by `Amplitude` and add `DCOffset`.

$$wave = w \times Amplitude + DCOffset$$

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

See Also

System Objects
audioOscillator

Introduced in R2016a

configureMIDI

System object: wavetableSynthesizer

Configure MIDI connections between System object and MIDI controller

Syntax

```
configureMIDI(waveSynth)
configureMIDI(waveSynth,propName)
configureMIDI(waveSynth,propName,controlNumber)
configureMIDI(waveSynth,propName,controlNumber,'DeviceName',
deviceName)
```

Description

`configureMIDI(waveSynth)` starts a MIDI configuration user interface (UI). Use the UI to synchronize tunable properties of the `wavetableSynthesizer` System object, `waveSynth`, to MIDI controls of your choice.

`configureMIDI(waveSynth,propName)` makes the System object property, `propName`, respond to any control on the default MIDI device.

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

`configureMIDI(waveSynth,propName,controlNumber,'DeviceName',deviceName)` makes the property respond to the MIDI control specified by `controlNumber` on the device specified by `deviceName`.

Each tunable property of the `wavetableSynthesizer` System object maps to MIDI controls with a specified range.

Property	Range	Mapping
`Frequency`	0.1 Hz to 20 kHz	log
`Amplitude`	0 to 10	linear
`DCOffset`	–10 to 10	linear

Introduced in R2016a

createAudioPluginClass

System object: wavetableSynthesizer

Create audio plugin class that implements functionality of System object

Syntax

```
createAudioPluginClass(waveSynth)
createAudioPluginClass(waveSynth,pluginName)
```

Description

`createAudioPluginClass(waveSynth)` creates a System object source plugin that implements the functionality of the `wavetableSynthesizer` System object, `waveSynth`. The name of the created class is the `wavetableSynthesizer` System object variable name followed by `'Plugin'`, for example, `waveSynthPlugin`. By default, the created class outputs a one-channel (column) matrix.

`createAudioPluginClass(waveSynth,pluginName)` specifies the name of your created System object source plugin class.

Example: `createAudioPluginClass(waveSynth,'myWavetableSynthesizer')` creates a System object source plugin with class name `myWavetableSynthesizer`.

Each tunable property of the `wavetableSynthesizer` System object maps to a plugin parameter with a default range.

Property	Plugin Parameter Range	Mapping
`Frequency`	0.1 Hz to 20 kHz	log
`Amplitude`	0 to 10	linear
`DCOffset`	–10 to 10	linear

Introduced in R2016a

disconnectMIDI

System object: wavetableSynthesizer

Disconnect MIDI controls from System object

Syntax

```
disconnectMIDI(waveSynth)
```

Description

`disconnectMIDI(waveSynth)` disconnects MIDI controls from your wavetable synthesizer, `waveSynth`. Only those MIDI connections established using `configureMIDI` are disconnected.

Introduced in R2016a

getMIDIConnections

System object: wavetableSynthesizer

Get MIDI connection information

Syntax

```
connectionInfo= getMIDIConnections(waveSynth)
```

Description

`connectionInfo= getMIDIConnections(waveSynth)` returns a structure, `connectionInfo`, containing information about the MIDI connections for your wavetable synthesizer, `waveSynth`. Only those MIDI connections established using `configureMIDI` are returned. The `connectionInfo` structure contains a substructure for each tunable property of `waveSynth` 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).

Introduced in R2016a

reset

System object: wavetableSynthesizer

Reset internal states of System object

Syntax

`reset(waveSynth)`

Description

`reset(waveSynth)` resets internal states of the wavetable synthesizer, `waveSynth`, to their initial values.

Introduced in R2016a

step

System object: wavetableSynthesizer

Generate periodic signals from single-cycle waveforms

Syntax

```
y = step(waveSynth)
```

Description

Note: Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj)` and `y = obj()` perform equivalent operations.

`y = step(waveSynth)` generates a periodic signal, `y`. The type of signal is specified by the algorithm and properties of the `wavetableSynthesizer` System object, `waveSynth`.

Note: The System object performs an internal initialization the first time you execute `step`. This initialization locks nontunable properties and input specifications, such as the dimensions, complexity, and data type. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

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 on your input:

1. Define and set up your frequency-weighted filter. See “Construction” on page 3-308.
2. Call step to perform frequency-weighted filtering on each channel of the input signal according to the properties of your `weightingFilter` object. The input must be a real-valued, double-precision or single-precision matrix. The `weightingFilter` object treats each column of the input as an independent channel.

Note: Alternatively, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`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

If a property is listed as tunable, then you can change its value even when the object is locked.

`Method` — Type of weighting

`'A-weighting'` (default) | `'C-weighting'` | `'K-weighting'`

Type of weighting, specified as `'A-weighting'`, `'C-weighting'`, or `'K-weighting'`. See "Weighting Types" on page 3-321 for more information.

Tunable: No

`SampleRate` — Input sample rate (Hz)

`44100` (default) | positive scalar

Input sample rate in Hz, specified as a positive scalar.

Tunable: Yes

Methods

createAudioPluginClass	Create audio plugin class that implements functionality of System object
getFilter	Return biquad filter object using weighting filter design
isStandardCompliant	Verify filter design is IEC 61672-1:2002 compliant
reset	Reset internal states of System object
step	Apply frequency-weighted filtering
visualize	Visualize and validate filter response

Common to All System Objects	
`clone`	Create System object with same property values
`getNumInpu`	Expected number of inputs to a System object
`getNumOutp`	Expected number of outputs of a System object
`isLocked`	Check locked states of a System object (logical)
`release`	Allow System object property value changes

Examples

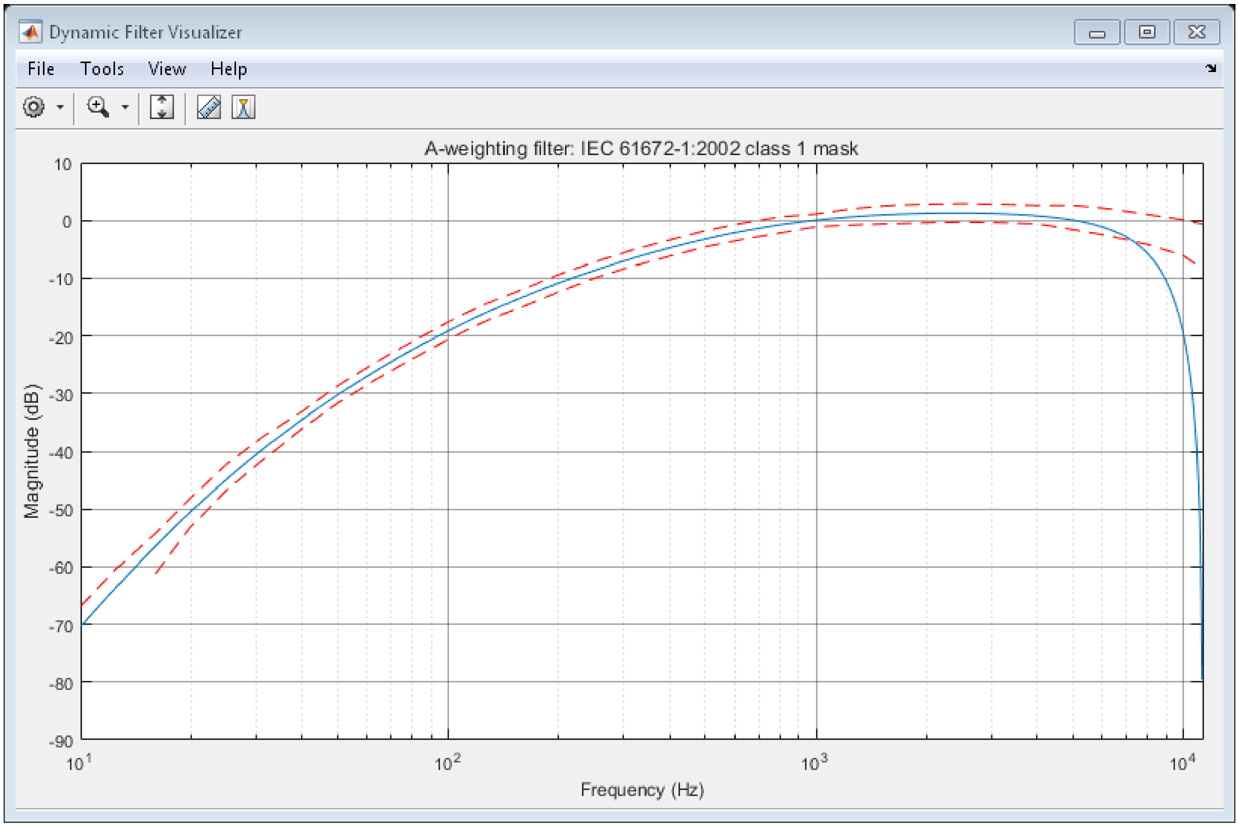
Validate Filter Compliance

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')
```

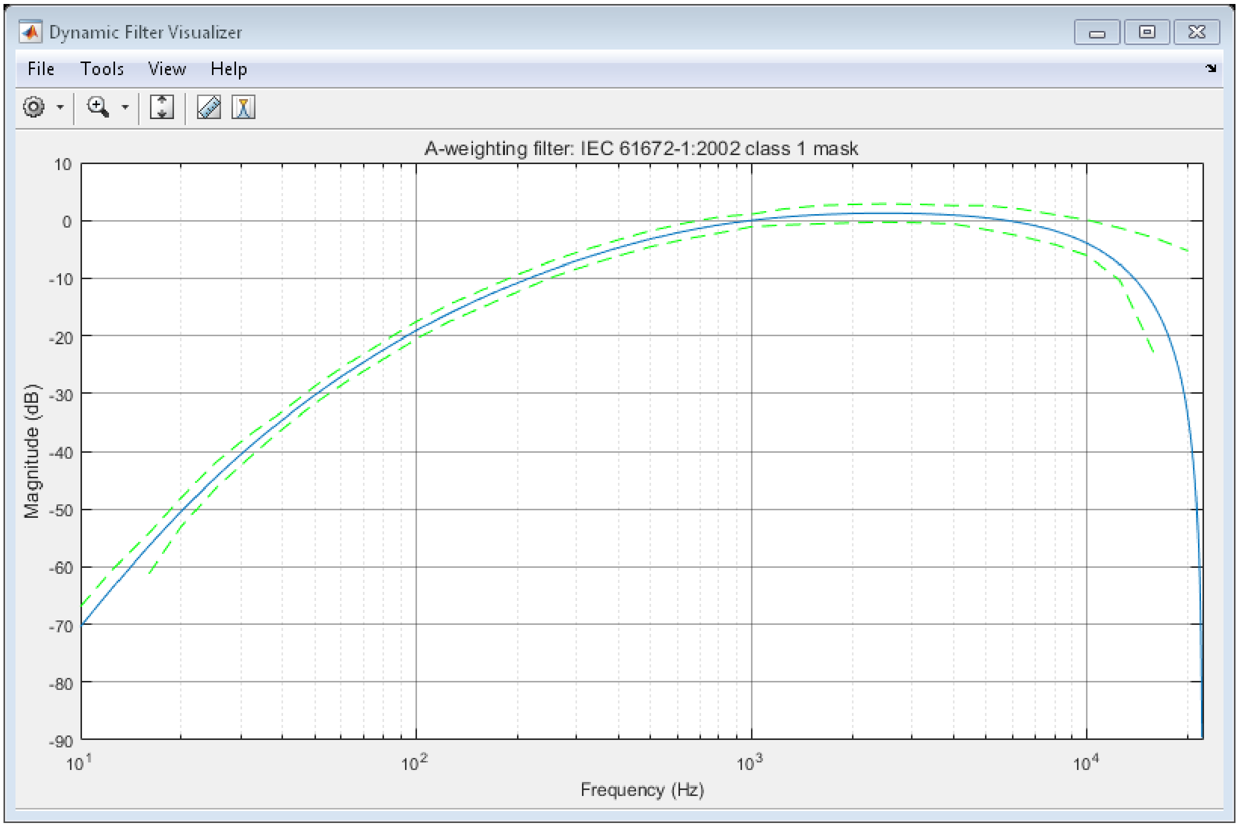
```
complianceStatus =

  logical

   0
```



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')
```

```
complianceStatus =

  logical

   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 an audio file reader System object.

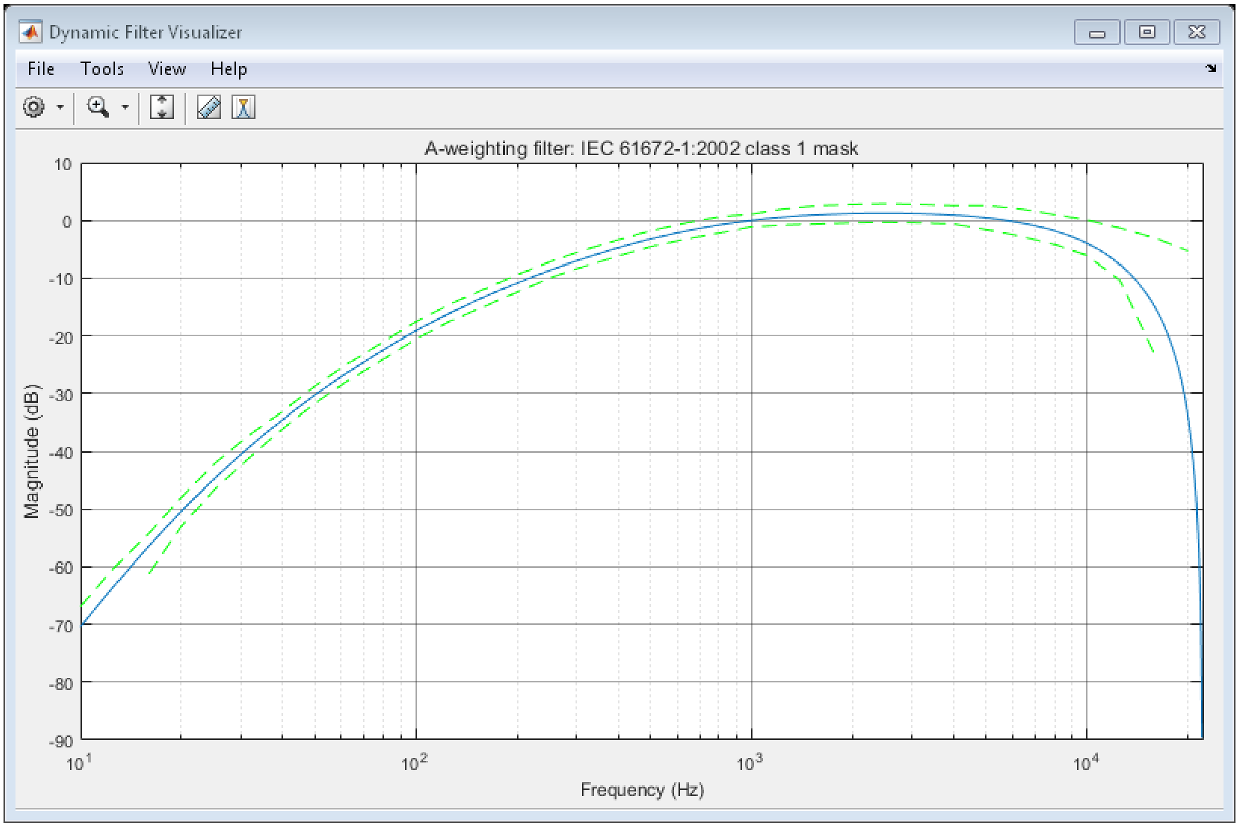
```
samplesPerFrame = 1024;
reader = dsp.AudioFileReader('Filename',...
    'RockGuitar-16-44p1-stereo-72secs.wav',...
    'SamplesPerFrame',samplesPerFrame,...
    'PlayCount',Inf);
```

Create a weighting filter 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');
```

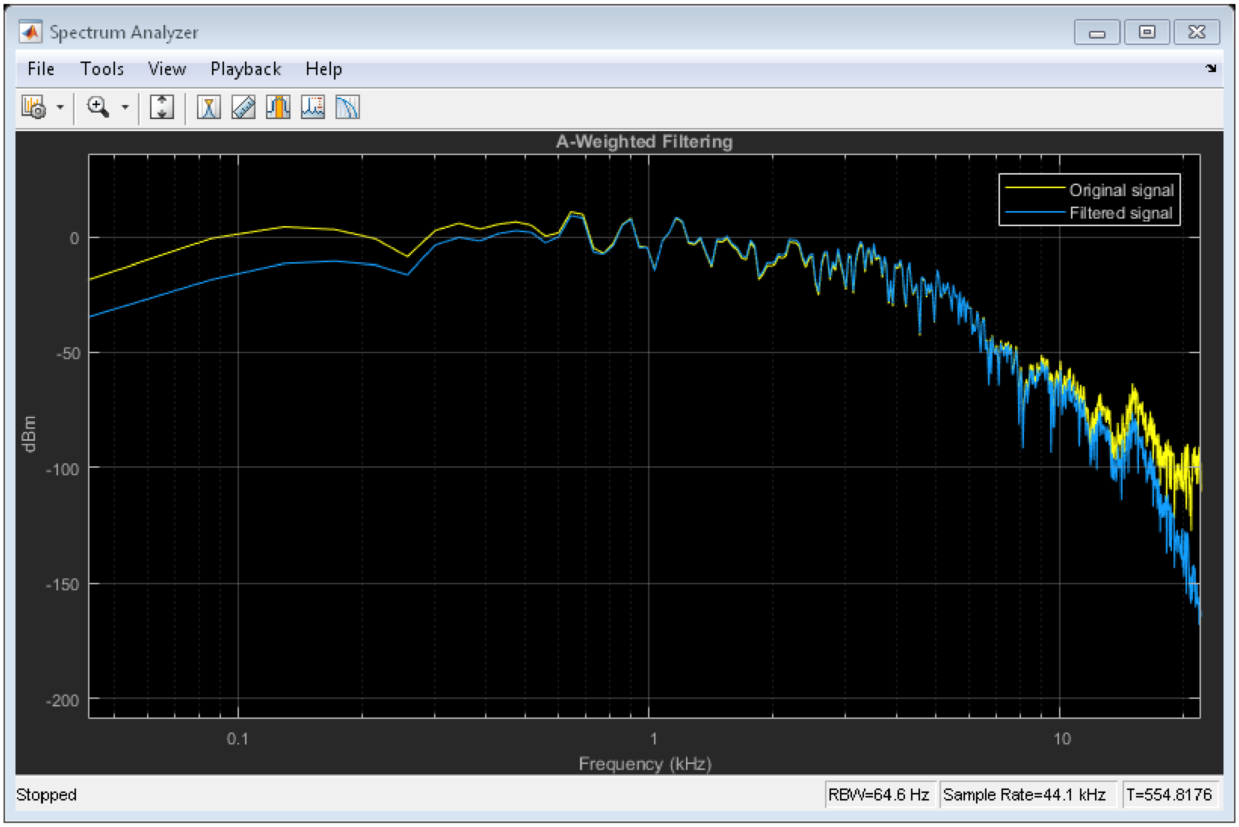


Create a spectrum analyzer to visualize the original audio signal and the audio signal after frequency-weighted filtering.

```
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);
```

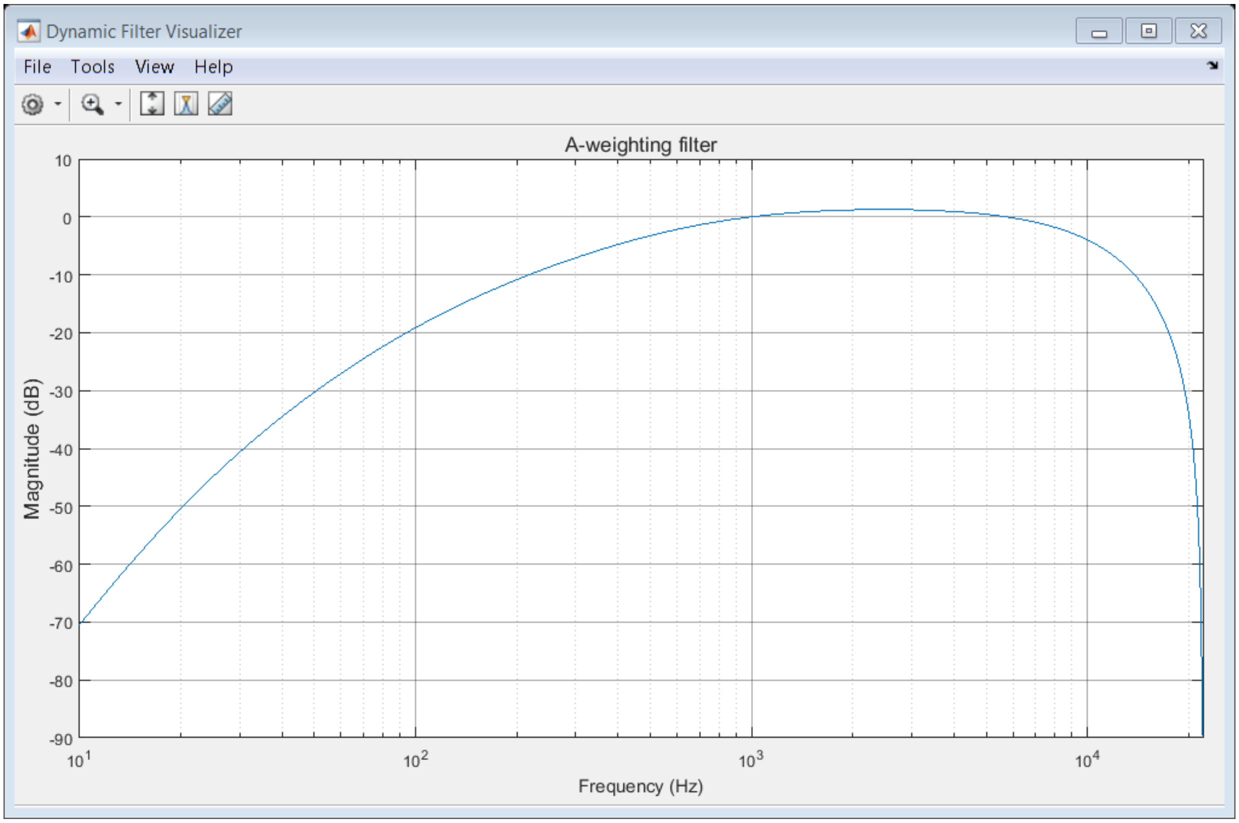


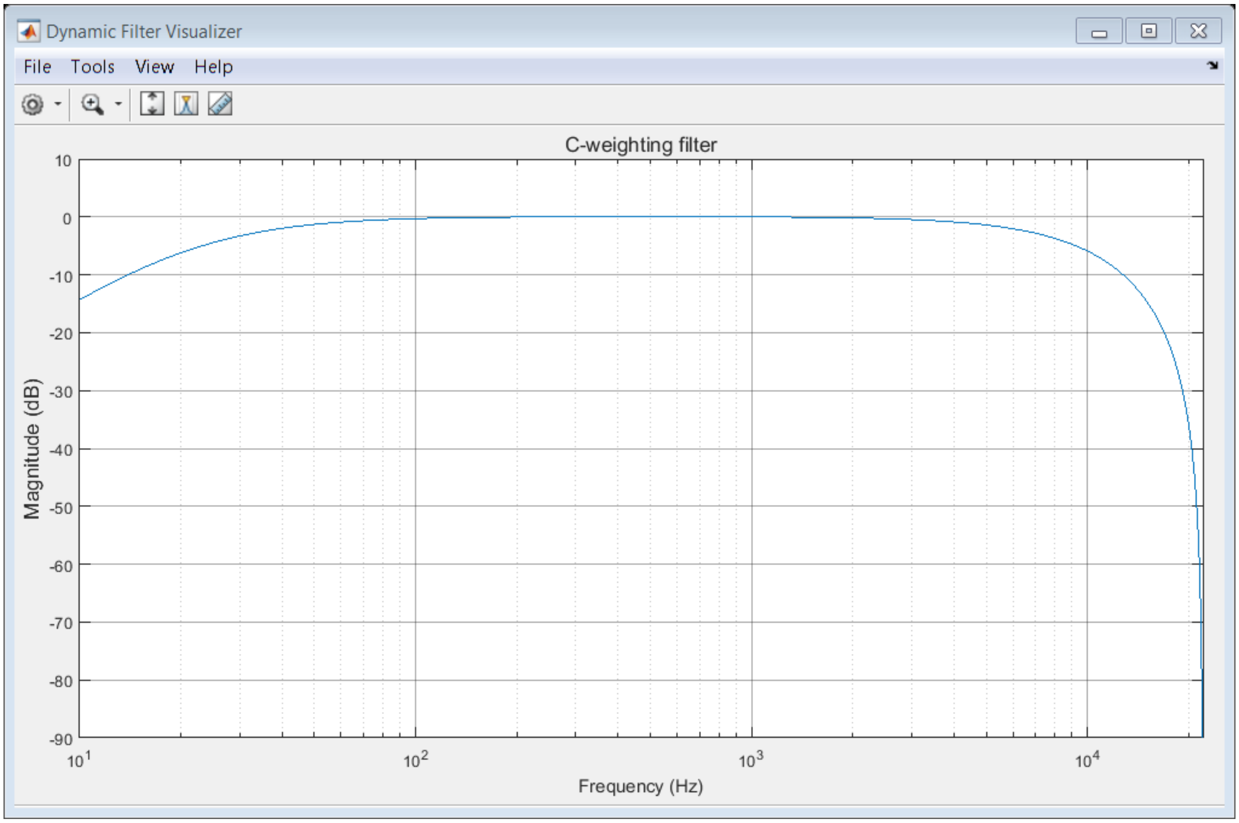
Compare Weighting Types

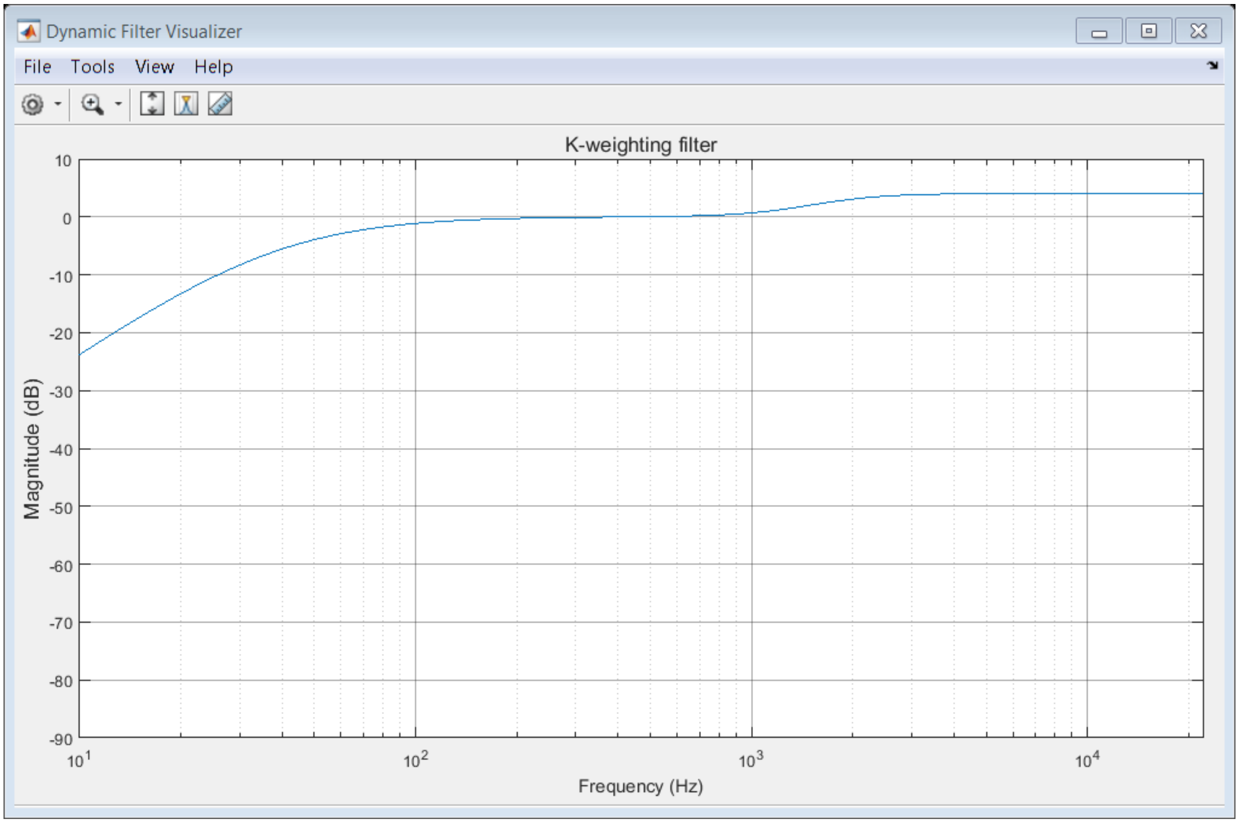
Compare the A-weighted, C-weighted, and K-weighted filtering of an engine sound.

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)
Frequency (Hz)
10
0
-10
-20
-30
-40
-50
-60
-70
-80
-90
10^1
10^2
10^3
10^4


Dynamic Filter Visualizer
File
Tools
View
Help
C-weighting filter
10
0
-10
-20
-30
-40
-50
-60
-70
-80
-90
Magnitude (dB)
10^1
10^2
10^3
10^4
Frequency (Hz)



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
```

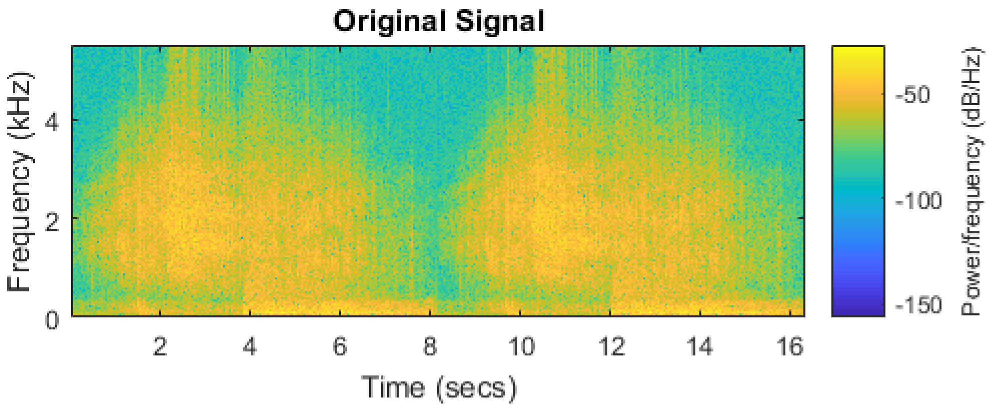
```
while toc<8
    x = fileReader();
    deviceWriter(x);
end
release(fileReader)
```

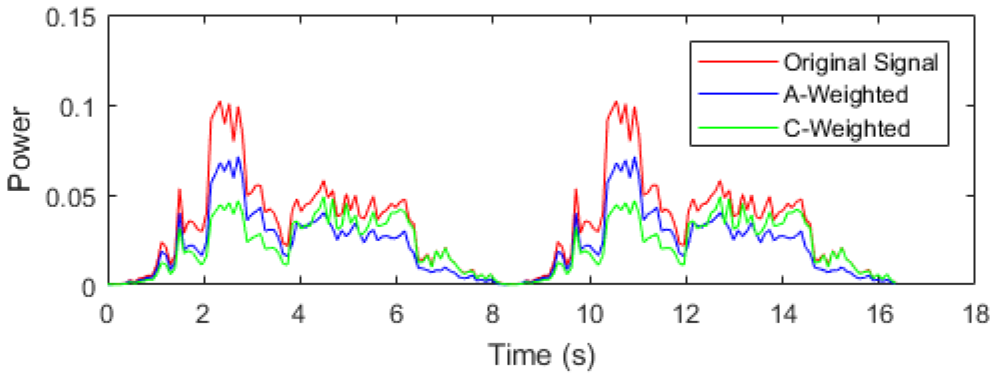
Play the signal processed by the A-weighted filter. Then play the signal processed by the C-weighted filter. Cache the power in each frame of the original and filtered signals for analysis. As a best practice, release your file reader and device writer once complete.

```
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');
```





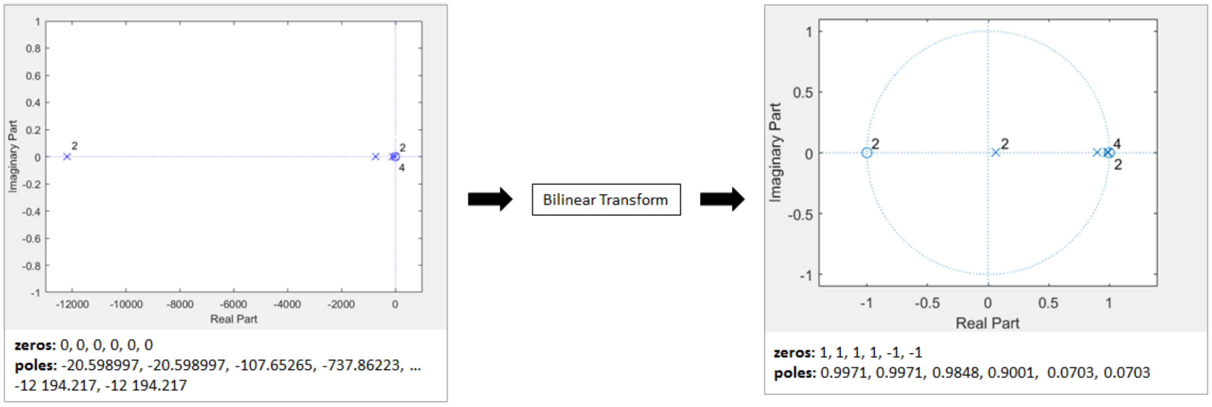
Weighting Types

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:



zeros: 0, 0, 0, 0, 0, 0
poles: -20.598997, -20.598997, -107.65265, -737.86223, ... -12 194.217, -12 194.217

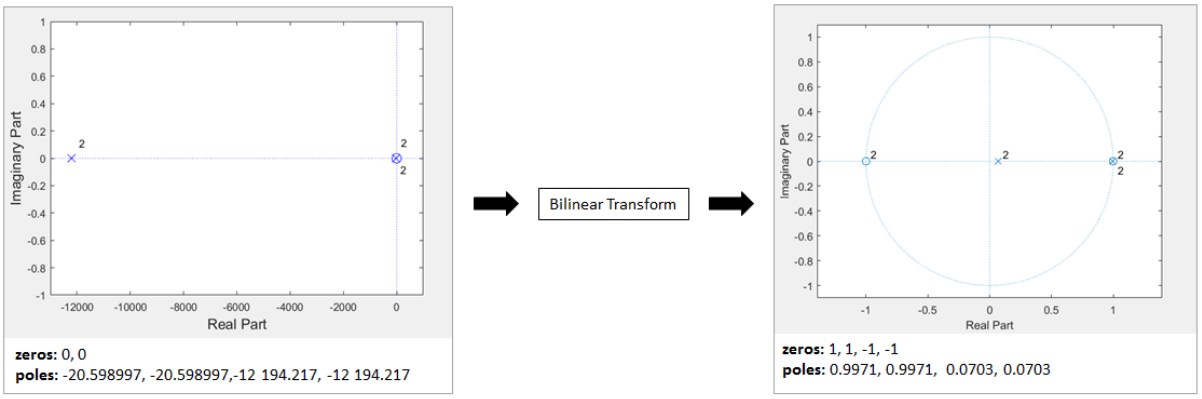
zeros: 1, 1, 1, 1, -1, -1
poles: 0.9971, 0.9971, 0.9848, 0.9001, 0.0703, 0.0703

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:



zeros: 0, 0
poles: -20.598997, -20.598997,-12 194.217, -12 194.217

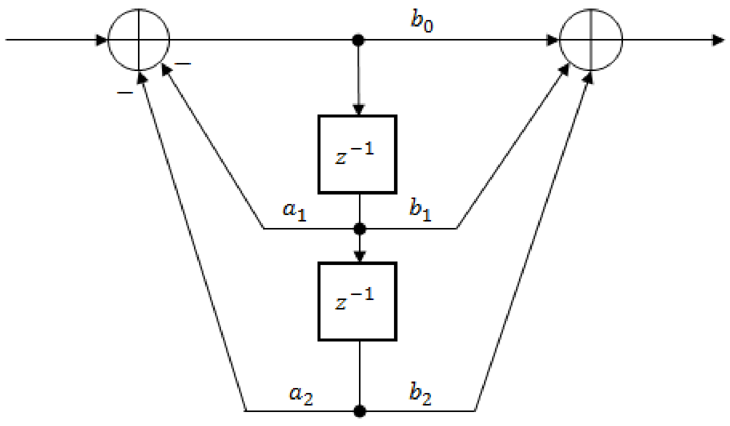
zeros: 1, 1, -1, -1
poles: 0.9971, 0.9971, 0.0703, 0.0703

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 MATLAB® Coder™.

Usage notes and limitations:

"System Objects in MATLAB Code Generation" (MATLAB Coder)

See Also

See Also

Blocks
`Weighting Filter`

System Objects
multibandParametricEQ | octaveFilter | dsp.BiquadFilter

Topics
"Audio Weighting Filters"
"Sound Pressure Measurement Using Weighting Filters"

Introduced in R2016b

createAudioPluginClass

System object: weightingFilter

Create audio plugin class that implements functionality of System object

Syntax

```
createAudioPluginClass(weightFilt)
createAudioPluginClass(weightFilt,pluginName)
```

Description

`createAudioPluginClass(weightFilt)` creates a System object plugin that implements the functionality of the `weightingFilter` System object, `weightFilt`. The name of the created class is the `weightingFilter` System object variable name followed by `'Plugin'`, for example, `weightFiltPlugin`.

Note: 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(weightFilt,pluginName)` specifies the name of your created System object plugin class.

Example: `createAudioPluginClass(weightFilt,'AweightedFilter')` creates a System object plugin with class name `AweightedFilter`.

Introduced in R2016b

getFilter

System object: weightingFilter

Return biquad filter object using weighting filter design

Syntax

```
biquad = getFilter(weightFilt)
```

Description

`biquad = getFilter(weightFilt)` returns a `dsp.BiquadFilter` object, `biquad`. The `SOSMatrix` and `ScaleValues` properties of the biquad filter object are set as specified by the `weightingFilter` object, `weightFilt`.

Use `getFilter` for the design capabilities of the `weightingFilter` System object and the processing capabilities of the `dsp.BiquadFilter` System object.

Introduced in R2016b

isStandardCompliant

System object: weightingFilter

Verify filter design is IEC 61672-1:2002 compliant

Syntax

```
complianceStatus = isStandardCompliant(weightFilt,classType)
complianceStatus =
isStandardCompliant(weightFilt,classType,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. Specify `classType` as `'class 1'` or `'class 2'`. You can check compliance for A-weighting and C-weighting filters only.

`complianceStatus = isStandardCompliant(weightFilt,classType,freqRange)` specifies the range of frequencies checked for compliance. Specify `freqRange` as a two-element vector of increasing values: `[minFreq, maxFreq]`.

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. Increase the sample rate to meet compliance.

Introduced in R2016b

reset

System object: weightingFilter

Reset internal states of System object

Syntax

```
reset(weightFilt)
```

Description

`reset(weightFilt)` resets internal states of the octave filter, `weightFilt`, to their initial values.

Introduced in R2016b

step

System object: weightingFilter

Apply frequency-weighted filtering

Syntax

```
y = step(weightFilt,x)
```

Description

Note: Alternatively, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`y = step(weightFilt,x)` applies frequency-weighted filtering to the input signal, `x`, and returns the filtered signal, `y`. The type of filtering is specified by the algorithm and properties of the `weightingFilter` System object, `weightFilt`.

`x` must be a real-valued, double-precision or single-precision matrix. The System object treats each column of the input as an independent channel.

Note: The System object performs an internal initialization the first time you execute `step`. This initialization locks nontunable properties and input specifications, such as the dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

Introduced in R2016b

visualize

System object: weightingFilter

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. The default is `2048`.

`visualize(___,mType)` creates a mask based on the class of filter specified by `mType`, using either of the previous syntaxes. Specify `mType` 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.

Introduced in R2016b

Classes in Audio System Toolbox

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

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

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 normalized value of the `Medium Peak Gain` parameter value to zero.

```
parameterValue = 0;
setParameter(hostedPlugin,'Medium Peak Gain',parameterValue)
```

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 parameterValue < 0.995
    parameterValue = parameterValue + 0.001;
    setParameter(hostedPlugin,'Medium Peak Gain',parameterValue);
    x = fileReader();
    y = process(hostedPlugin,x);
    deviceWriter(y);
end

release(fileReader)
release(deviceWriter)
```

See Also

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, 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 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).
`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.

Field	Description
`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 and VST3.
`InputChannels`	Number of channels passed to the processing function of the plugin.
`OutputChannel`	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 normalized value of the `Amplitude` parameter to 0.05. Set the normalized value of the `Frequency` parameter to 0.8.

```
setParameter(hostedSourcePlugin,'Amplitude',0.05);
frequencyParameterValue = 0.8;
setParameter(hostedSourcePlugin,'Frequency',frequencyParameterValue);
```

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
    frequencyParameterValue = frequencyParameterValue - 0.0004*k;
    setParameter(hostedSourcePlugin,'Frequency',frequencyParameterValue);
    y = process(hostedSourcePlugin);
    deviceWriter(y);
    if i == 500
        k = -1;
    end
end

release(deviceWriter)
```

See Also

See Also

Functions
`loadAudioPlugin`

Classes
audioPlugin | audioPluginSource | externalAudioPlugin

Topics
"Host External Audio Plugins"
"Hierarchies of Classes — Concepts" (MATLAB)

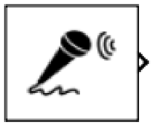
Introduced in R2016b

Blocks in Audio System Toolbox

Audio Device Reader

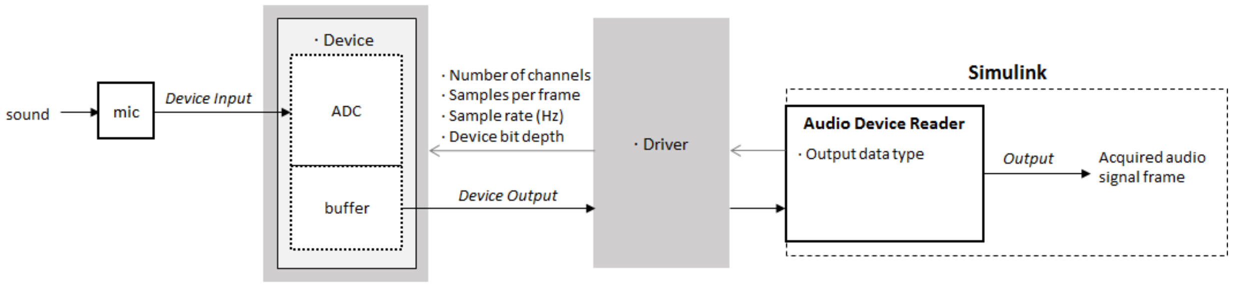
Record from sound card

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.

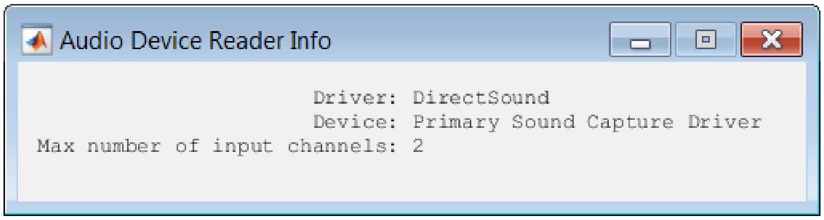
This parameter applies only on Windows machines. Linux machines always use the ALSA driver. Mac machines always use the CoreAudio driver.

`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:



```
                        Driver: DirectSound
                        Device: Primary Sound Capture Driver
Max number of input channels: 2
```

`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`

Model Examples

See Also

See Also

System Objects
audioDeviceReader | audioDeviceWriter

Blocks
`Audio Device Writer`

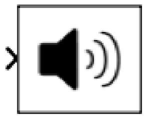
Topics
"Run Audio I/O Features Outside MATLAB and Simulink"
"Audio I/O: Buffering, Latency, and Throughput"

Introduced in R2016a

Audio Device Writer

Play to sound card

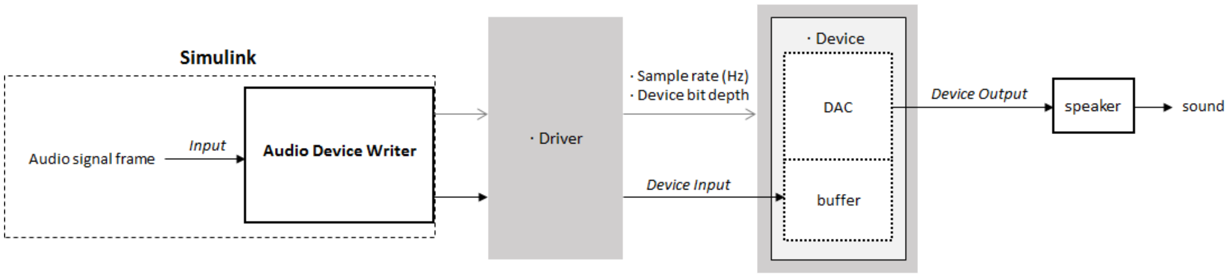
Library: Audio System Toolbox / Sinks



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 underrrun 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.

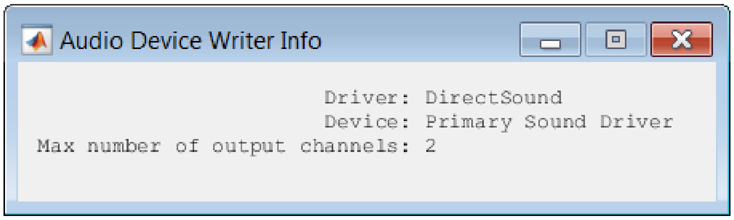
To specify nondefault **Driver** values, you must install Audio System Toolbox. If the toolbox is not installed, specifying nondefault **Driver** values returns an error.

`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:



```
                     Driver: DirectSound
                     Device: Primary Sound Driver
Max number of output channels: 2
```

`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 underrrun while writing a frame of data (one input matrix).

Model Examples

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

Usage notes and limitations:

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 block 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 .

See Also

See Also

Blocks
`Audio Device Reader` | Binary File Reader

System Objects
audioDeviceWriter | audioDeviceReader

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. It uses specified attack and release times to achieve a smooth applied gain curve. You can tune parameters of the Compressor block to meet your processing needs.

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.

This port is unnamed until you select the **Output gain (dB)** 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

Operation threshold is the level above which gain is applied to the input signal.

Tunable: Yes

`Ratio` — Compression ratio
5 (default) | scalar in the range 1 to 50

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 $R = \frac{(x[n]-T)}{(y[n]-T)}$.

- 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

`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

$$y = x + \frac{\left(\frac{1}{R} - 1\right) \times \left(x - T + \frac{W}{2}\right)^2}{(2 \times W)}$$

for the range $(2 \times |x - T|) \le W$.

- 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

`View static characteristic` — Open static characteristic plot of the dynamic range compressor

button

The plot is updated automatically when parameters of the Compressor block change.

`Attack time (s)` — Time it takes applied gain to ramp up

`0.05` (default) | scalar in the range 0 to 4

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. The **Attack time (s)** parameter smooths the applied gain curve.

Tunable: Yes

`Release time (s)` — Time it takes applied gain to ramp down
`0.2` (default) | scalar in the range 0 to 4

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. The **Release time (s)** parameter smooths the applied gain curve.

Tunable: Yes

`Make-up gain mode` — Make-up gain mode
`Property` (default) | `Auto`

- `Property` — Make-up gain is set to the value specified by **Make-up gain (dB)**.
- `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.

`Make-up gain (dB)` — Applied make-up gain
`0` (default) | scalar in the range –10 to 24

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 **Make-up gain mode** 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, you specify the sample rate in **Input sample rate (Hz)**.

`Input sample rate (Hz)` — Sample rate of input
`44100` (default) | positive scalar

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.

`Simulate using` — Specify type of simulation to run

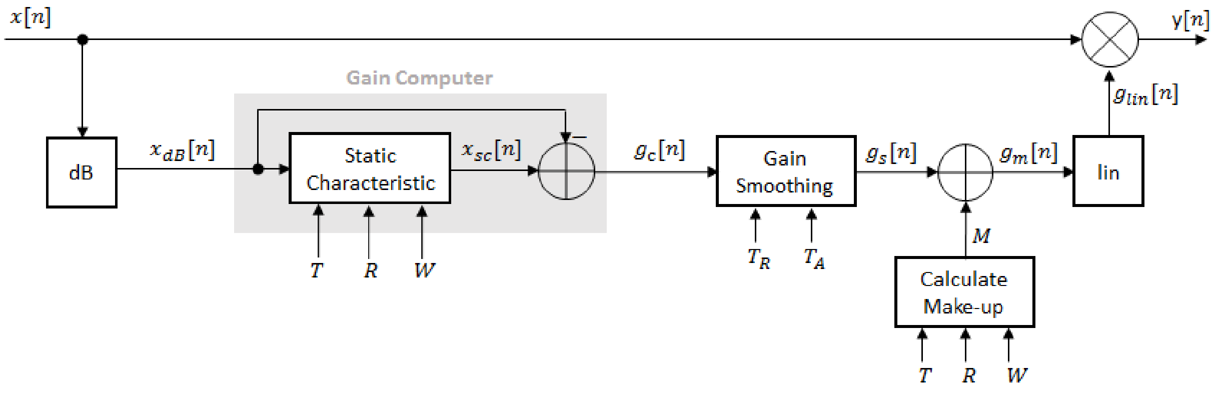
`Interpreted execution` (default) | `Code generation`

- `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`.

Model Examples

Algorithm

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:

$$x_{sc}(x_{dB}) = \begin{cases} x_{dB} & x_{dB} < \left(T - \frac{W}{2}\right) \\ x_{dB} + \dfrac{\left(\frac{1}{R} - 1\right)\left(x_{dB} - T + \frac{W}{2}\right)^2}{2W} & \left(T - \frac{W}{2}\right) \le x_{dB} \le \left(T + \frac{W}{2}\right) \\ T + \dfrac{(x_{dB} - T)}{R} & x_{dB} > \left(T + \frac{W}{2}\right) \end{cases},$$

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:

$$x_{sc}(x_{dB}) = \begin{cases} x_{dB} & x_{dB} < T \\ T + \dfrac{(x_{dB} - T)}{R} & x_{dB} \ge T \end{cases}$$

3 The computed gain, $g_c[n]$, is calculated as

$$g_c[n] = x_{sc}[n] - x_{dB}[n].$$

4 $g_\Delta[n]$ 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,

$$M = -x_{sc}(x_{dB} = 0).$$

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, pp. 399–408.

See Also

See Also

Blocks
`Expander` | `Limiter` | `Noise Gate`

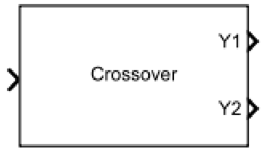
System Objects
compressor

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.

`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)**.

`Input sample rate (Hz)` — Sample rate of input
44100 (default) | positive scalar

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`.

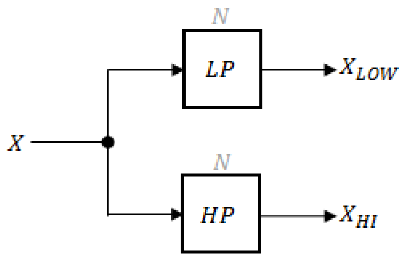
Model Examples

Algorithm

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

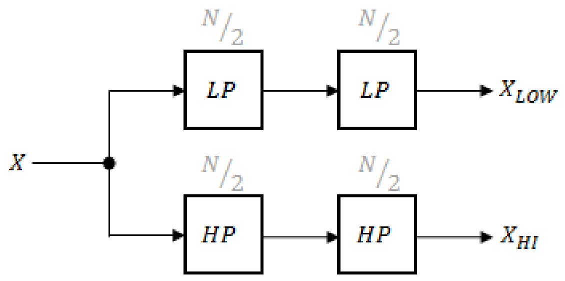
Odd-order two-band (one crossover) filters are implemented as parallel complementary highpass and lowpass filters.



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.

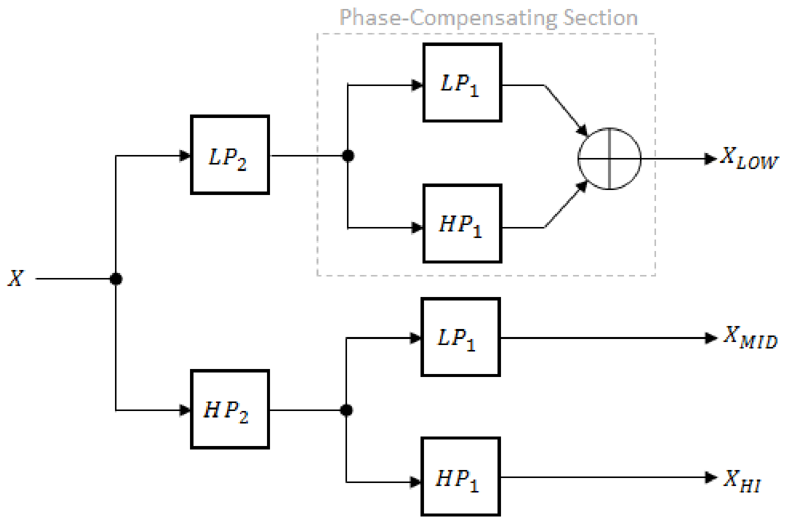


LP and *HP* are Butterworth filters of order *N/2*, where *N* is the order of the overall filter. The filters are implemented as direct-form II transposed second-order sections.

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, pp. 239–245.

See Also

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. It uses specified attack, release, and hold times to achieve a smooth applied gain curve. You can tune parameters of the Expander block to meet your processing needs.

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

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

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 $R = \frac{(y[n]-T)}{(x[n]-T)}$.

- 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

`Threshold (dB)` — Operation threshold
`–10` (default) | scalar in the range –140 to 0

Operation threshold is the level below which gain is applied to the input signal.

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 $(2\times|x-T|) \le W$.

- 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

`View static characteristic` — Open static characteristic plot of the dynamic range expander
button

The plot is updated automatically when parameters of the Expander block change.

`Attack time (s)` — Time it takes applied gain to ramp up
`0.05` (default) | scalar in the range 0 to 4

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. The **Attack time (s)** parameter smooths the applied gain curve.

Tunable: Yes

`Release time (s)` — Time it takes applied gain to ramp down
`0.2` (default) | scalar in the range 0 to 4

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. The **Release time (s)** parameter smooths the applied gain curve.

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.

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)**.

`Input sample rate (Hz)` — Sample rate of input
`44100` (default) | positive scalar

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. By default, this parameter is cleared.

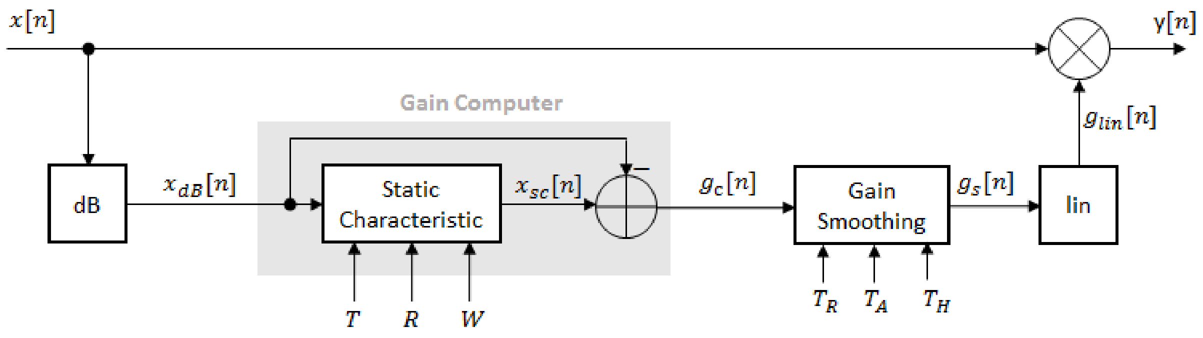
`Simulate using` — Specify type of simulation to run
`Interpreted execution` (default) | `Code generation`

- `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`.

Model Examples

Algorithm

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:

$$x_{sc}(x_{dB}) = \begin{cases} T + (x_{dB} - T) \times R & x_{dB} < \left(T - \frac{W}{2}\right) \\ x_{dB} + \frac{(1-R)\left(x_{dB} - T - \frac{W}{2}\right)^2}{2W} & \left(T - \frac{W}{2}\right) \le x_{dB} \le \left(T + \frac{W}{2}\right) \\ x_{dB} & x_{dB} > \left(T + \frac{W}{2}\right) \end{cases} ,$$

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:

$$x_{sc}(x_{dB}) = \begin{cases} T + (x_{dB} - T) \times R & x_{dB} < T \\ x_{dB} & x_{dB} \ge T \end{cases}$$

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:

$$g_s[n] = \begin{cases} \alpha_A g_s[n-1] + (1-\alpha_A) g_c[n] & (C_A > T_H)\ \&\ (g_c[n] > g_s[n-1]) \\ g_s[n-1] & C_A \le T_H \\ \alpha_R g_s[n-1] + (1-\alpha_R) g_c[n] & (C_R > T_H)\ \&\ (g_c[n] \le g_s[n-1]) \\ g_s[n-1] & C_R \le T_H \end{cases}$$

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 **Attack time (s)** parameter. T_R is the release time period, specified by the **Realease time (s)** parameter. Fs is the input sampling rate, specified by the **Inherit sample rate from input** or **Input sample rate (Hz)** parameter.

C_A and C_R are hold counters for attack and release, respectively. The limit, T_H, determined by the **Hold time (s)** 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

$$y[n] = x[n] \times g_{lin}[n].$$

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, pp. 399–408.

See Also

See Also

Blocks
`Compressor` | `Limiter` | `Noise Gate`

System Objects
expander

Topics
“Dynamic Range Control”

Introduced in R2016a

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. It uses specified attack and release times to achieve a smooth applied gain curve. You can tune parameters of the Limiter block to meet your processing needs.

Ports

Input

`Port_1` — 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.

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

Operation threshold is the level above which gain is applied to the input signal.

Tunable: Yes

`Knee width (dB)` — Transition area in the limiter characteristic
`0` (default) | scalar in the range 0 to 20

For soft knee characteristics, the transition area is defined by the relation

$$y = x - \frac{\left(x - T + \frac{W}{2}\right)^2}{(2 \times W)}$$

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.
- T is the threshold in dB.
- W is the knee width in dB.

Tunable: Yes

`View static characteristic` — Open static characteristic plot of the dynamic range limiter
button

The plot is updated automatically when parameters of the Limiter block change.

`Attack time (s)` — Time it takes applied gain to ramp up
`0` (default) | scalar in the range 0 to 4

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. The **Attack time (s)** parameter smooths the applied gain curve.

Tunable: Yes

`Release time (s)` — Time it takes applied gain to ramp down
`0.2` (default) | scalar in the range 0 to 4

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. The **Release time (s)** parameter smooths the applied gain curve.

Tunable: Yes

`Make-up gain mode` — Make-up gain mode
`Property` (default) | `Auto`

- `Property` — Make-up gain is set to the value specified by **Make-up gain (dB)**.
- `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.

`Make-up gain (dB)` — Applied make-up gain
`0` (default) | scalar in the range –10 to 24

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 **Make-up gain mode** 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, you specify the sample rate in **Input sample rate (Hz)**.

`Input sample rate (Hz)` — Specify input sample rate
`44100` (default) | positive scalar

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. By default, this parameter is cleared.

`Simulate using` — Specify type of simulation to run
`Interpreted execution` (default) | `Code generation`

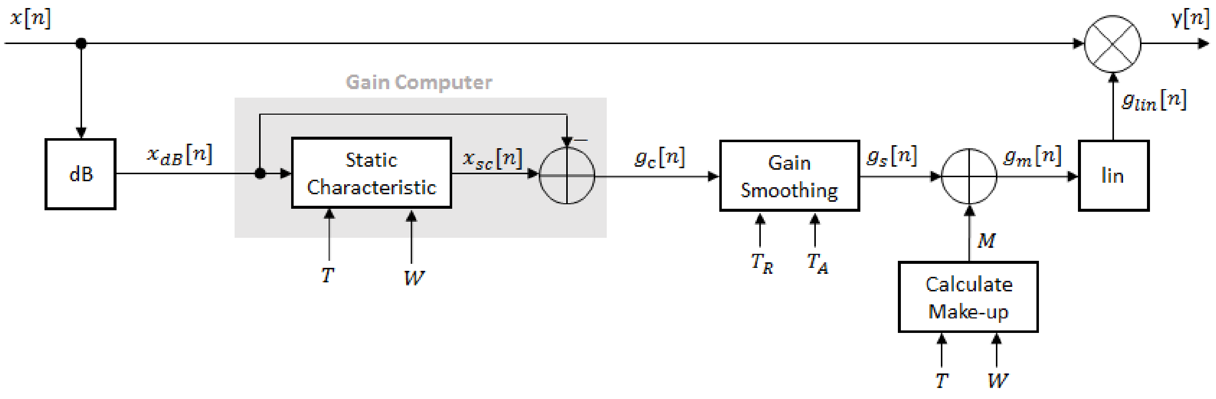
- `Interpreted execution` — Simulate model using the MATLAB interpreter. This option shortens startup time and has simulation speed comparable to `Code generation`. In this mode, you can debug the source code of the block.

- `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`.

Model Examples

Algorithm

The Limiter block processes a signal frame by frame and element by element.



1 The N-point signal, $x[n]$, is converted to decibels:

$$x_{dB}[n] = 20 \times \log_{10} |x[n]|$$

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 **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:

$$M = -x_{sc}(x_{dB} = 0)$$

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, pp. 399–408.

See Also

See Also

Blocks
`Compressor` | `Expander` | `Noise Gate`

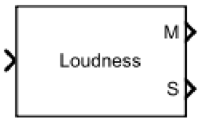
System Objects
limiter

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 consistently 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: No

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`

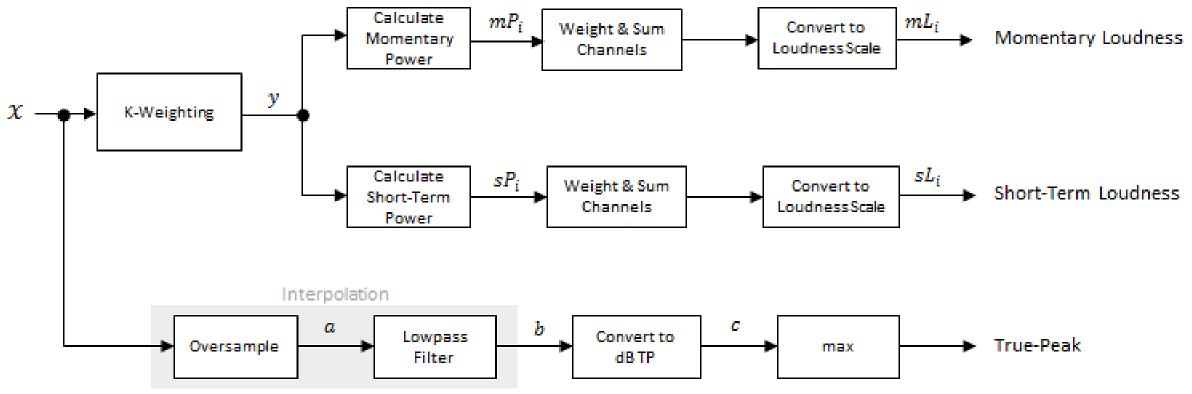
- `Code generation` — Simulate model using generated C code. The first time you run a simulation, Simulink generates C code for the block. The C code is reused for subsequent simulations, as long as the model does not change. This option requires additional startup time but the speed of the subsequent 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

Model Examples

Algorithm

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.

Momentary 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 *i*th 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.

Short-Term Loudness

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.

See Also

See Also

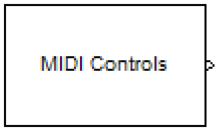
Functions
`integratedLoudness`

System Objects
loudnessMeter

Introduced in R2016b

MIDI Controls

Output values from controls on MIDI control surface
Library: Audio System Toolbox / Sources



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)`

Model Examples

See Also

See Also

Functions
`midicontrols` | `midiid` | `midiread` | `midisync`

Topics
"Musical Instrument Digital Interface (MIDI)"

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. It uses specified attack, release, and hold times to achieve a smooth applied gain curve. You can tune parameters of the Noise Gate block to meet your processing needs.

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

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

Operation threshold is the level below which gain is applied to the input signal.

Tunable: Yes

`View static characteristic` — Open static characteristic plot of the dynamic range gate
button

The plot is updated automatically when parameters of the Noise Gate block change.

`Attack time (s)` — Time it takes applied gain to ramp up
`0.05` (default) | scalar in the range 0 to 4

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. The **Attack time (s)** parameter smooths the applied gain curve.

Tunable: Yes

`Release time (s)` — Time it takes applied gain to ramp down
`0.2` (default) | scalar in the range 0 to 4

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. The **Release time (s)** parameter smooths the applied gain curve.

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.

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)**.

`Input sample rate (Hz)` — Specify input sample rate
`44100` (default) | scalar

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.

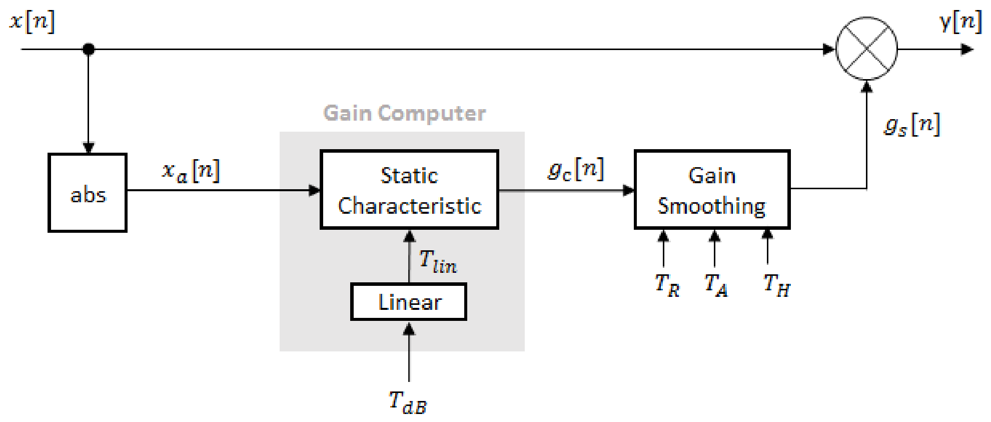
`Simulate using` — Specify type of simulation to run
`Interpreted execution` (default) | `Code generation`

- `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`.

Model Examples

Algorithm

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:

$$g_s[n] = \begin{cases} \alpha_A g_s[n-1] + (1-\alpha_A) g_c[n] & if\ (C_A > T_H)\ \&\ (g_c[n] > g_s[n-1]) \\ g_s[n-1] & if\ \ C_A \leq T_H \\ \alpha_R g_s[n-1] + (1-\alpha_R) g_c[n] & if\ (C_R > T_H)\ \&\ (g_c[n] \leq g_s[n-1]) \\ g_s[n-1] & if\ \ C_R \leq T_H \end{cases}$$

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 **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.

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.

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, pp. 399–408.

See Also

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

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

`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 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 the octave filter
`6` (default) | even integer

Tunable: No

`Center frequency (Hz)` — Center frequency of the octave filter
`1000` (default) | scalar in the range `3` to `22000`

- 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 one Hz. Frequencies below this value are quantized to one Hz.

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)**.

`Input sample rate (Hz)` — Sample rate of input
`44100` (default) | scalar

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.

`Mask for attenuation limits` — Creates 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.

`Visualize filter response` — Open plot to visualize magnitude response and compliance mask
button

A 2048-point FFT is used to calculate the magnitude response.

Model Examples

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.

Algorithm

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 object normalizes the specified center frequency:

$$f_c = \frac{2 \times CenterFrequency}{SampleRate}$$

Then the object computes the band edge frequencies:

$$f_{pa} = f_c \times G^{-1/2b}$$

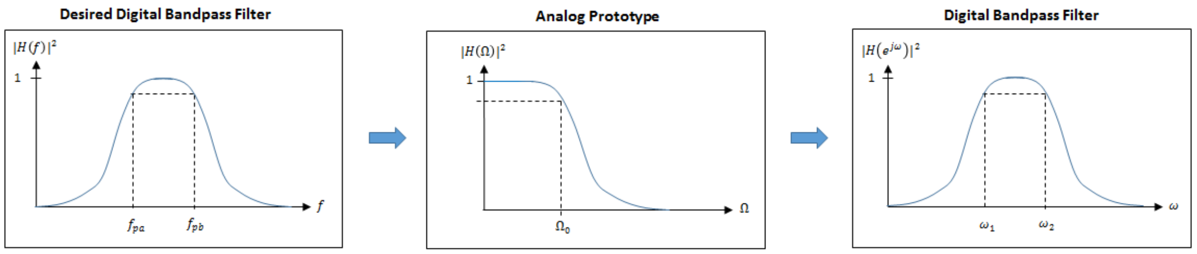
$$f_{pb} = f_c \times G^{1/2b}$$

- 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:

$$H(s) = H_0(s)H_1(s)H_2(s)\cdots H_K(s)\text{, where:}$$

- $$H_0(s) = \begin{cases} 1, & \text{if} \quad N = 2K \\ \dfrac{1}{1+\dfrac{1}{\Omega_0}}, & \text{if} \quad N = 2K+1 \end{cases}$$

- $$H_i(s) = \frac{1}{1 - 2\dfrac{s}{\Omega_0}\cos\theta_i + \dfrac{s^2}{\Omega_0^2}}, \quad i = 1,2,\ldots,K$$

- $$\theta_i = \frac{\pi}{2N}(N-1+2i), \quad i = 1,2,\ldots,N,\ldots,2N$$

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:

$$\Omega_0 = \frac{c - \cos\omega_{pb}}{\sin\omega_{pb}}$$

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, 2009.

See Also

See Also

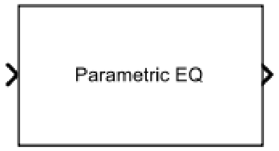
System Objects
octaveFilter | weightingFilter

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)**.

`Input sample rate (Hz)` — Sample rate of input
`44100` (default) | scalar

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`.

Model Examples

References

[1] Orfanidis, Sophocles J. *Introduction to Signal Processing*. Upper Saddle River, NJ: Prentice-Hall, 1996.

See Also

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)**.

`Input sample rate (Hz)` — Sample rate of input
`44100` (default) | positive scalar

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

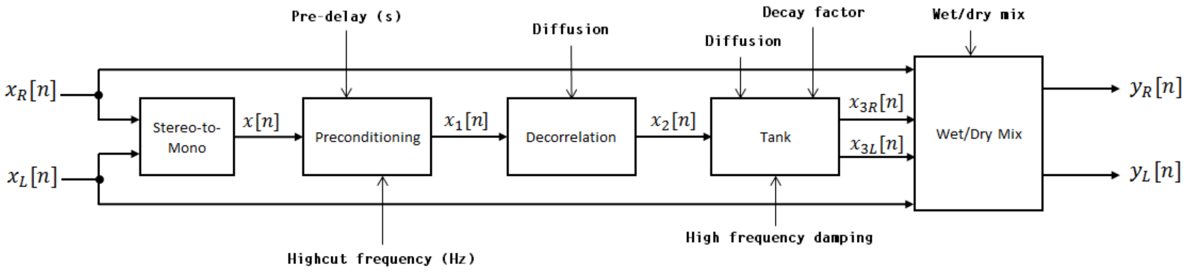
additional startup time but the speed of the subsequent simulations is comparable to `Interpreted execution`.

Model Examples

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.



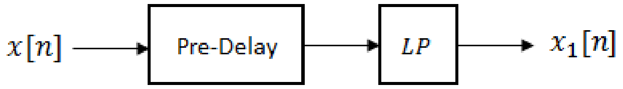
The description for the algorithm that follows is for a stereo input. A mono input is a simplified case.

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

  $$LP(z) = \frac{1-\alpha}{1-\alpha z^{-1}},$$

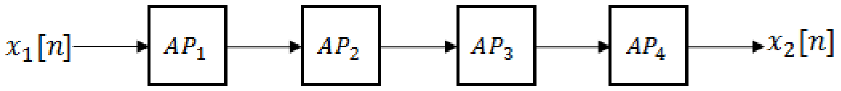
  where

  $$\alpha = \exp\left(-2\pi \times \frac{f_c}{f_s}\right).$$

  - 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

$$AP(z) = \frac{\beta + z^{-k}}{1+\beta z^{-k}},$$

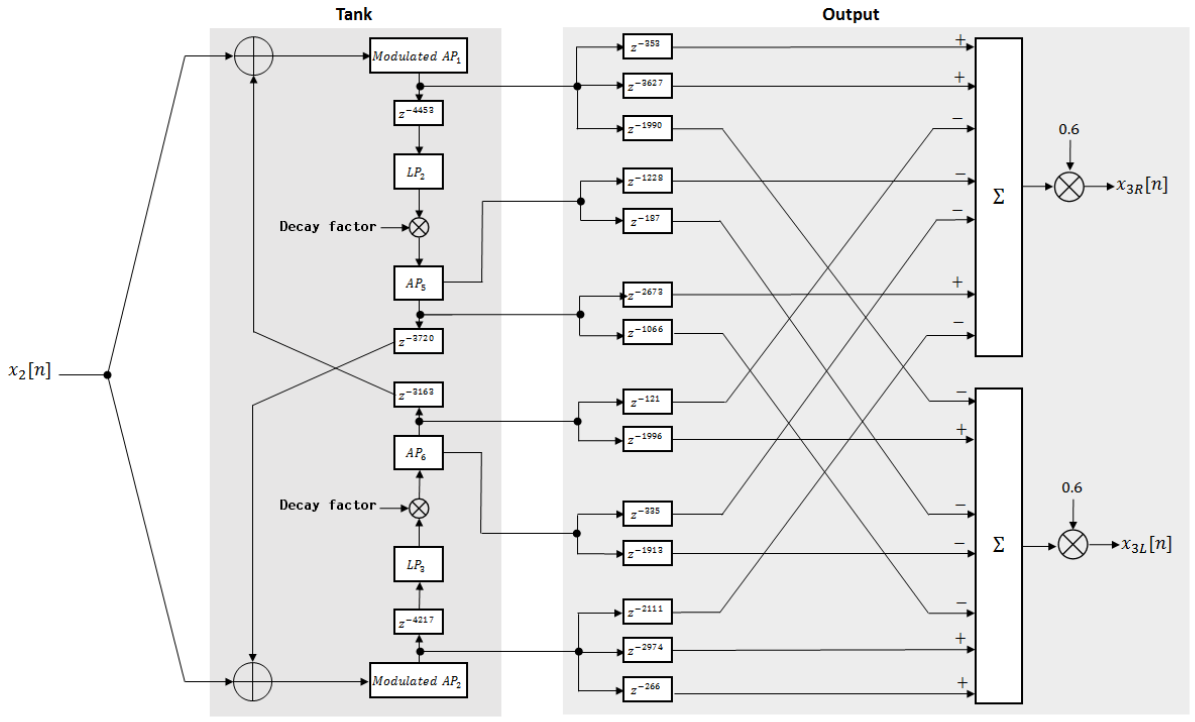
where β is the coefficient specified by the `Diffusion` property and k is the delay as follows:

- 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, 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, pp. 764–788.

See Also

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 5-83, “C-Weighting” on page 5-84, and “K-Weighting” on page 5-85 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)**.

`Input sample rate (Hz)` — Sample rate of input
`44100` (default) | `positive scalar`

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.

`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.

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.

Model Examples

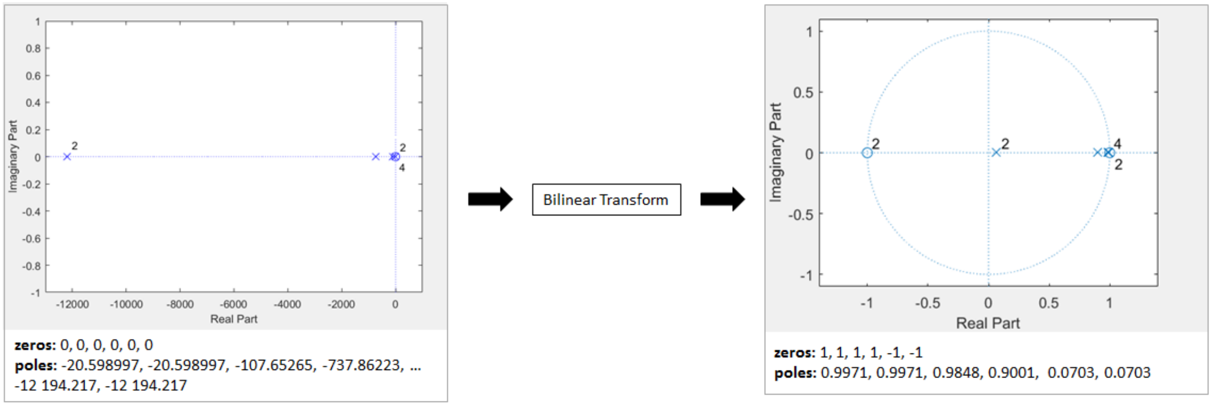
Definitions

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:



zeros: 0, 0, 0, 0, 0, 0
poles: -20.598997, -20.598997, -107.65265, -737.86223, ...
-12 194.217, -12 194.217

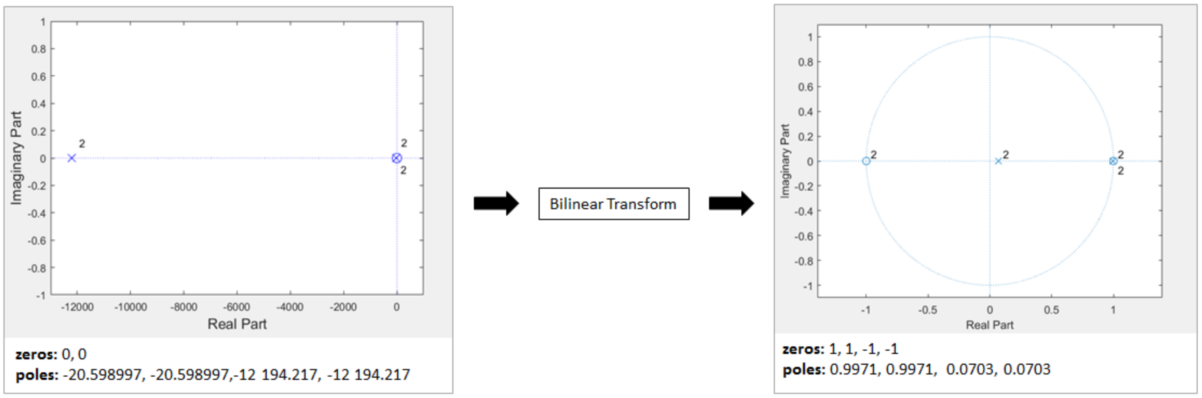
zeros: 1, 1, 1, 1, -1, -1
poles: 0.9971, 0.9971, 0.9848, 0.9001, 0.0703, 0.0703

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:



zeros: 0, 0
poles: -20.598997, -20.598997,-12 194.217, -12 194.217

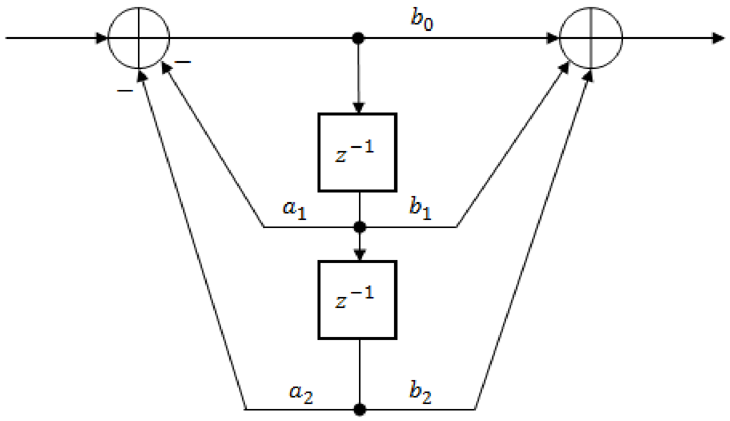
zeros: 1, 1, -1, -1
poles: 0.9971, 0.9971, 0.0703, 0.0703

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
$a_1 = -1.69065929318241$	$a_1 = -1.99004745483398$
$a_2 = 0.73248077421585$	$a_2 = 0.99007225036621$
$b_0 = 1.53512485958697$	$b_0 = 1.0$
$b_1 = -2.69169189406 38$	$b_1 = -2.0$
$b_2 = 1.19839281085285$	$b_2 = 1.0$

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.

See Also

See Also

Blocks

`Loudness Meter` | `Octave Filter`

System Objects

weightingFilter | octaveFilter | loudnessMeter