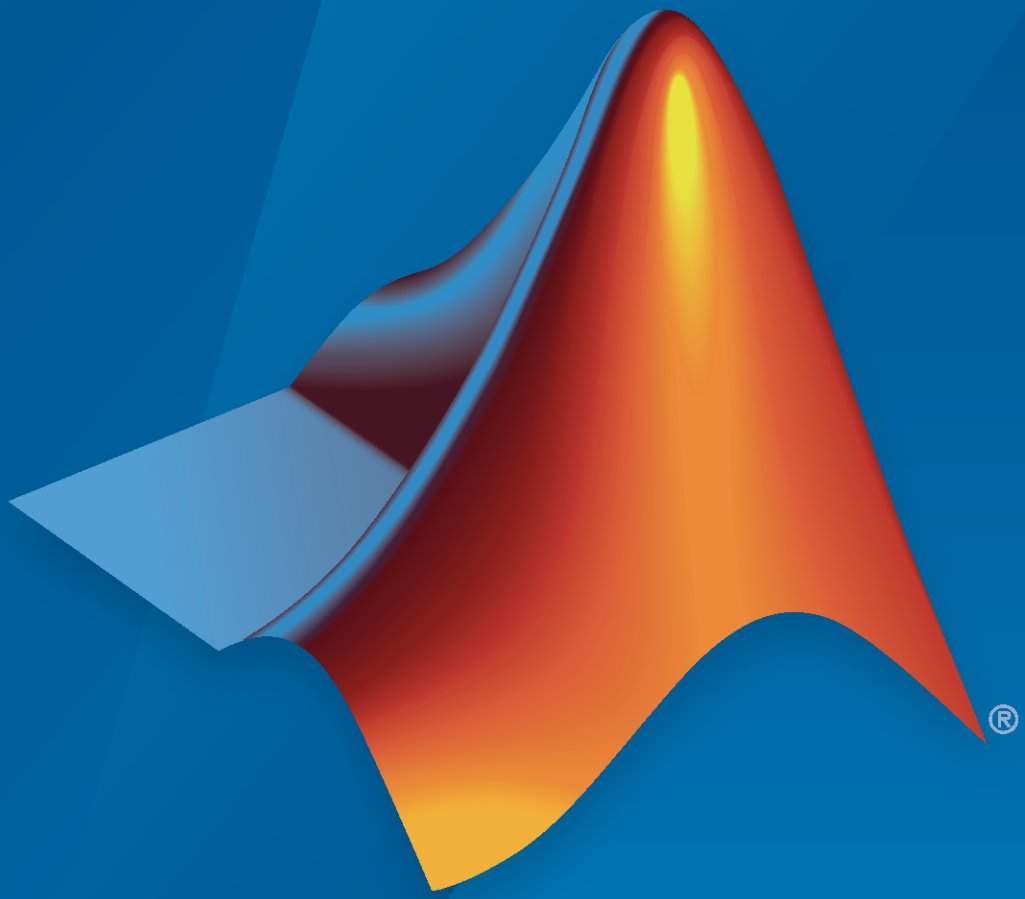
Audio Toolbox™

Reference



MATLAB® & SIMULINK®

R2023a

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.
1 Apple Hill Drive
Natick, MA 01760-2098

Audio 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)
September 2017	Online only	Revised for Version 1.3 (Release 2017b)
March 2018	Online only	Revised for Version 1.4 (Release 2018a)
September 2018	Online only	Revised for Version 1.5 (Release 2018b)
March 2019	Online only	Revised for Version 2.0 (Release 2019a)
September 2019	Online only	Revised for Version 2.1 (Release 2019b)
March 2020	Online only	Revised for Version 2.2 (Release 2020a)
September 2020	Online only	Revised for Version 2.3 (Release 2020b)
March 2021	Online only	Revised for Version 3.0 (Release 2021a)
September 2021	Online only	Revised for Version 3.1 (Release 2021b)
March 2022	Online only	Revised for Version 3.2 (Release 2022a)
September 2022	Online only	Revised for Version 3.3 (Release 2022b)
March 2023	Online only	Revised for Version 3.4 (Release 2023a)

Contents

Apps

Audio Labeler

(Removed) Define and visualize ground-truth labels

Note **Audio Labeler** has been removed. Use **Signal Labeler** instead. For more information, see "Compatibility Considerations".

Description

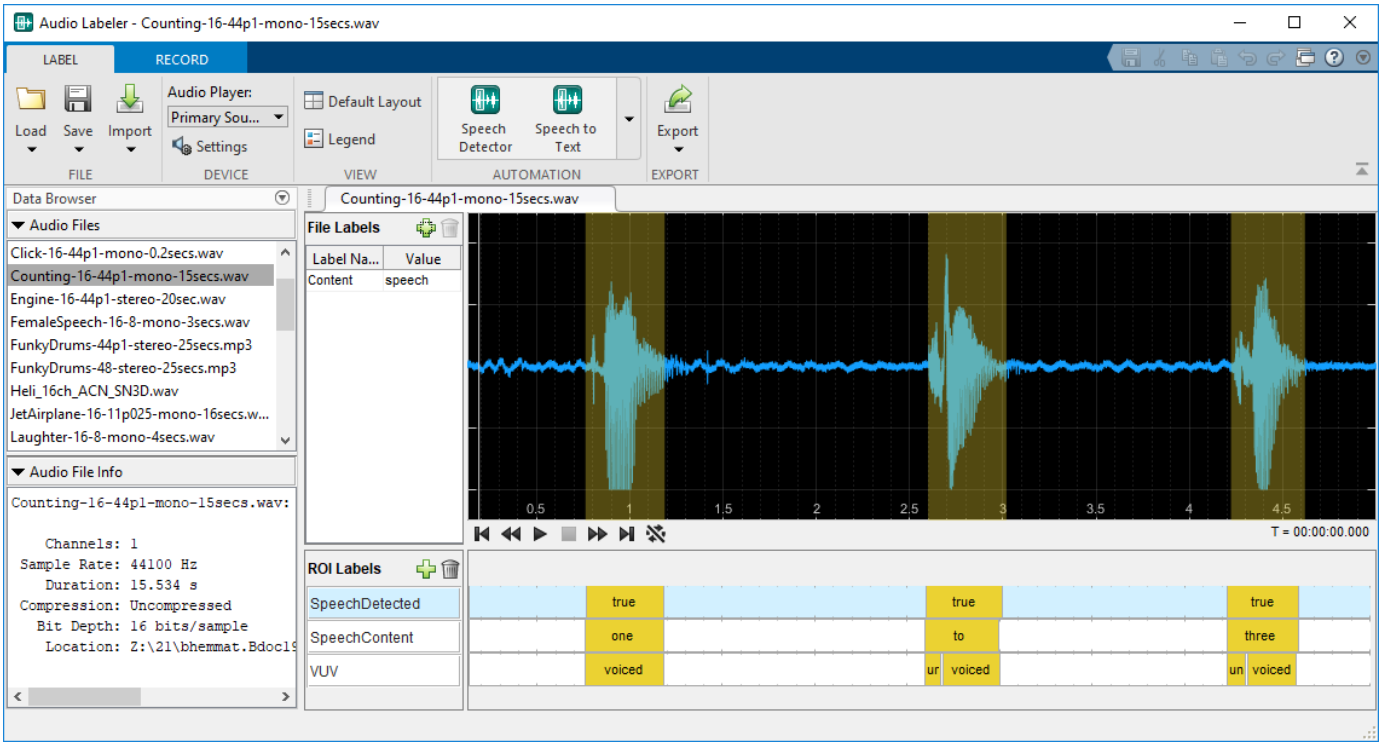
The **Audio Labeler** app enables you to label ground-truth data at both the region level and file level.

Using the app, you can:

- Create label definitions for consistent and fast labeling.
- Visualize the time-domain waveform during playback.
- Interactively specify labels at the file level and region level. You can specify regions by drawing directly on the time-domain waveform.
- Record new audio to add to your dataset.
- Apply automatic labeling of detected speech regions.
- Apply automatic word labeling using third-party speech-to-text transcription services. See "Speech-to-Text Transcription" for more information.

The app exports data as a `labeledSignalSet` object. You can use `labeledSignalSet` to train a network, classifier, or analyze data and report statistics.

- For advanced sublabeling and custom automated labeling functions, see "Import and Play Audio File Data in Signal Labeler".
- Audio playback and recording are not supported in MATLAB Online.



Open the Audio Labeler App

- MATLAB toolstrip: On the **Apps** tab, under **Signal Processing and Communications**, click the app icon.
- MATLAB command prompt: Enter `audioLabeler`.

Examples

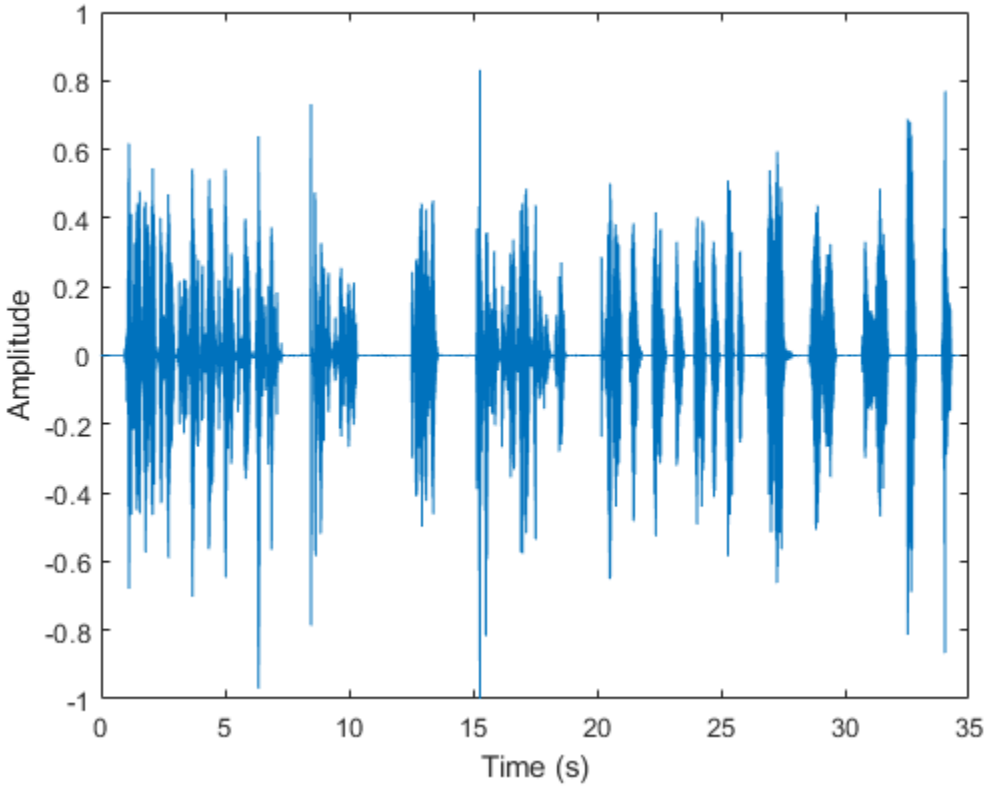
Create Keyword Spotting Mask Using Audio Labeler

In this example, you create a logical mask for an audio signal where ones correspond to the utterance "yes" and zeros correspond to the absence of the utterance "yes". To create the mask, you use the IBM™ speech-to-text API through the **Audio Labeler** app.

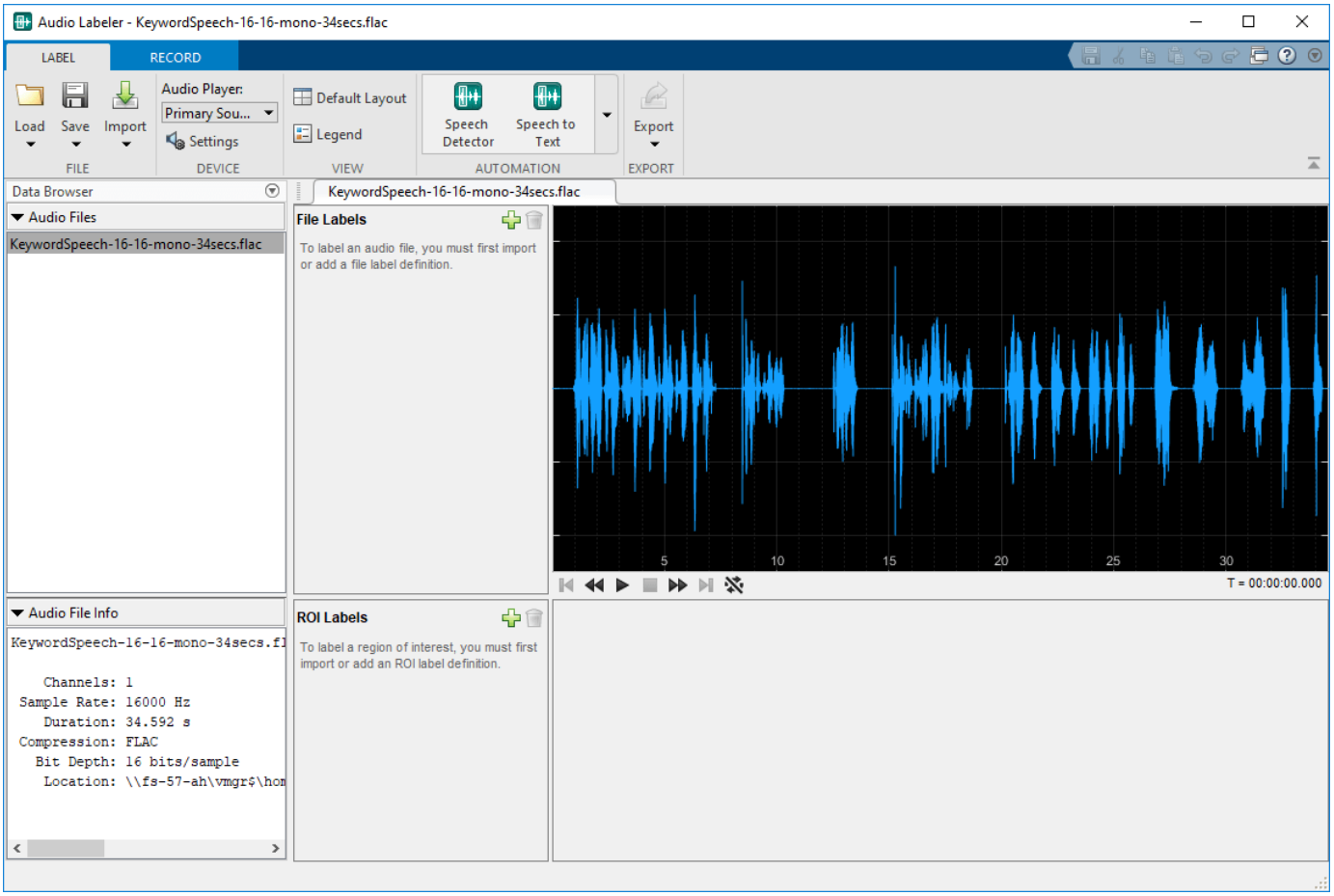
This example requires that you install the “Speech-to-Text Transcription” functionality.

Listen to the audio file that you want to label and then visualize it in the time domain.

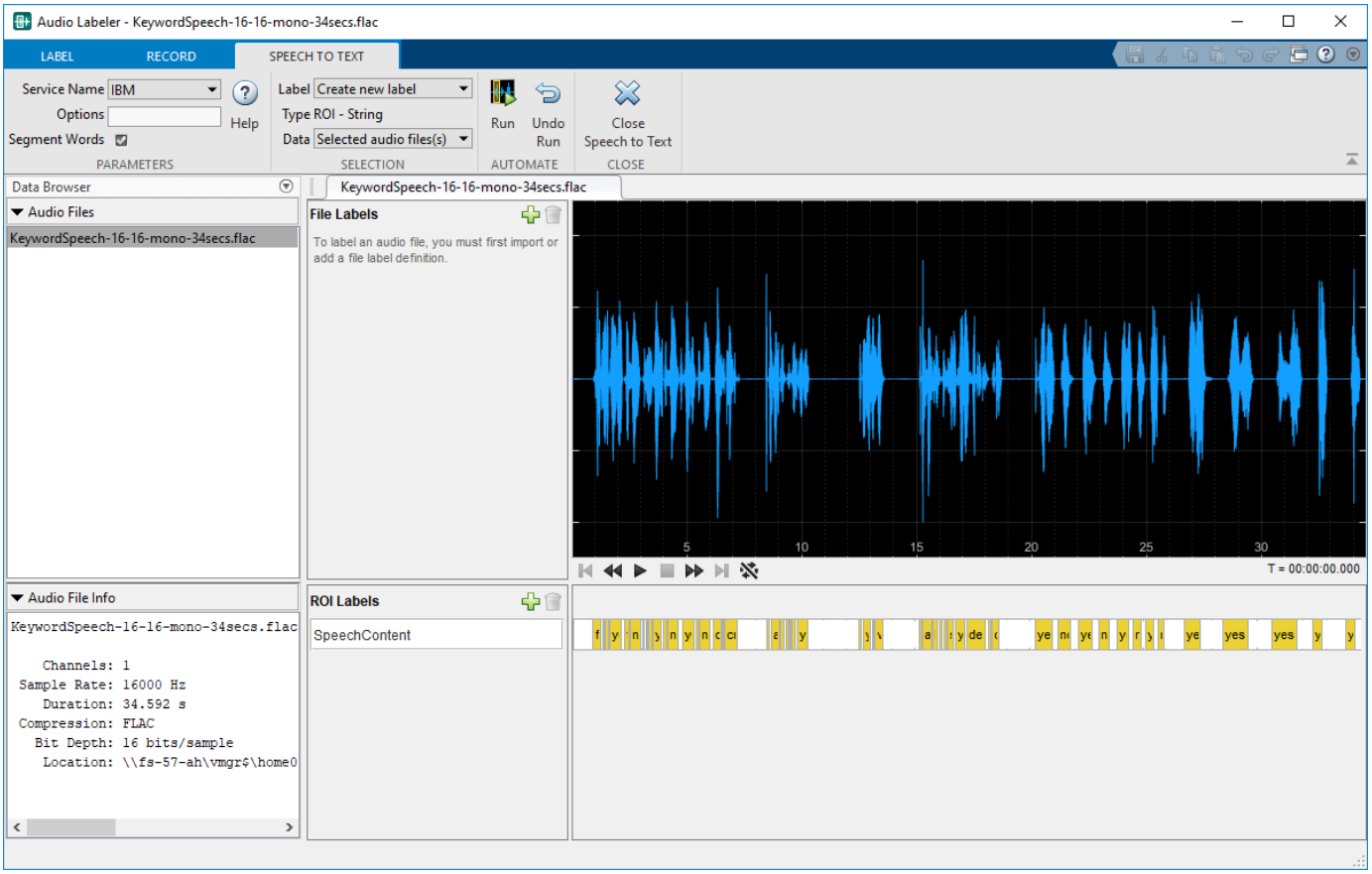
```
[audioIn,fs] = audioread("KeywordSpeech-16-16-mono-34secs.flac");

sound(audioIn,fs)

t = (0:numel(audioIn)-1)/fs;
plot(t,audioIn)
xlabel('Time (s)')
ylabel('Amplitude')
```



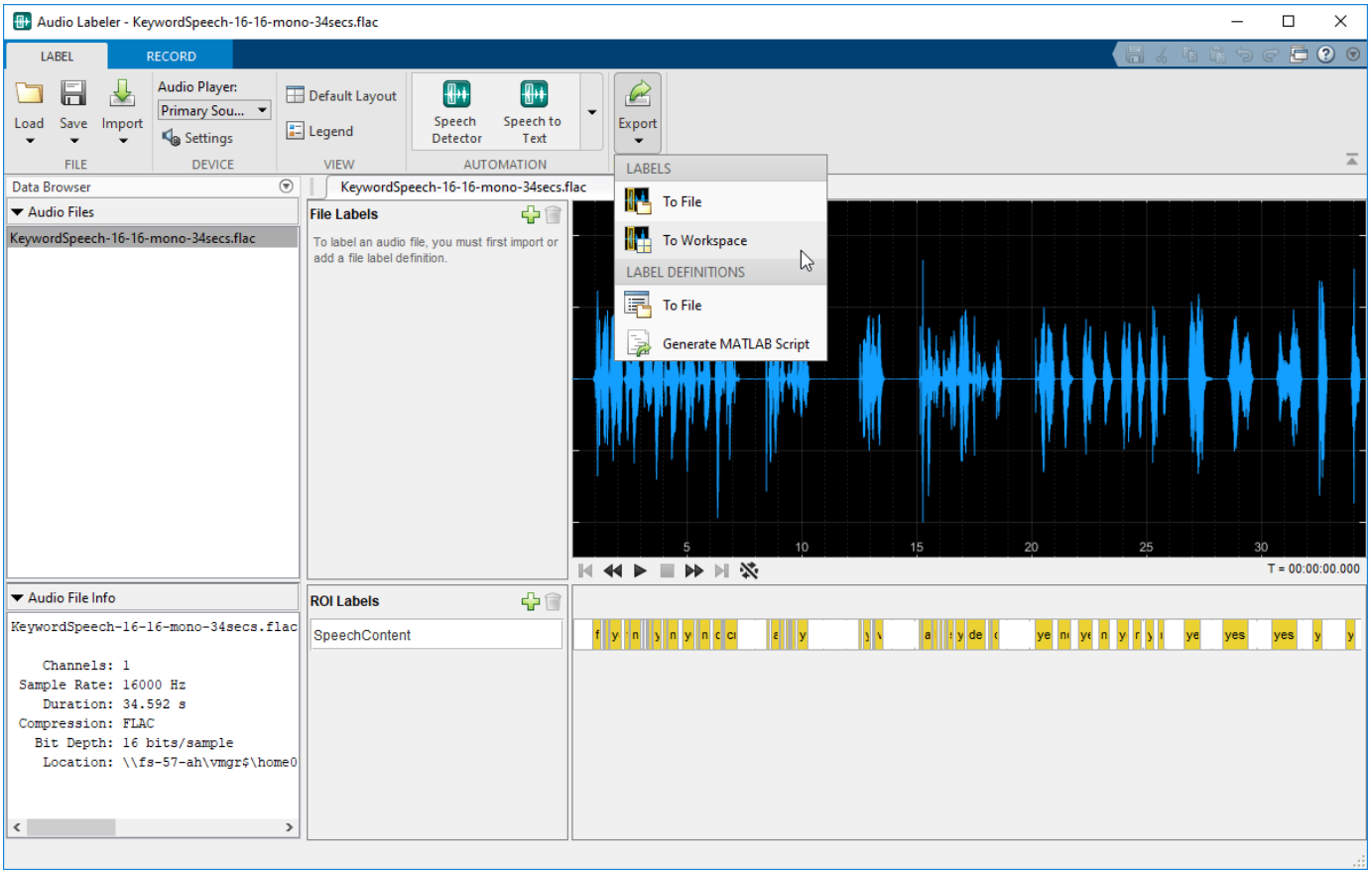
Open the **Audio Labeler** app and load the `KeywordSpeech-16-16-mono-34secs.flac` file into the **Data Browser**.



Under **Automation**, click **Speech to Text**. On the **Speech to Text** tab, select your preferred speech-to-text API. This example uses the IBM speech-to-text API. Select **Segment Words** so that the text labels are divided into individual words instead of sentences. Click **Run** to interface with the speech-to-text API and create a new region of interest (ROI) label. The ROI label contains words detected and labeled by IBM's speech-to-text API.



Close the **Speech to Text** tab and then export the labeled signal set to the workspace.



The labels are exported to the workspace as `labeledSignalSet` object with a time stamp. Set the variable `labeledSet` to the time-stamped `labeledSignalSet` object.

```
labeledSet = myLabeledSet;
```

Inspect the `SpeechContent` label.

```
speechContent = labeledSet.Labels.SpeechContent{1}
```

```
speechContent=52×2 table
     ROILimits        Value
    ____________    _________

    0.87    1.31    "first"
    1.31    1.41    "you"
    1.41    1.63    "said"
    1.63    2.22    "yes"
    2.25    2.52    "then"
    2.52    3.03    "no"
    3.09    3.22    "and"
    3.22    3.32    "you"
    3.32    3.52    "said"
    3.52    3.94    "yes"
    3.94    4.16    "then"
    4.16    4.66    "no"
    4.83    5.39    "yes"
    5.42    5.57    "the"
```

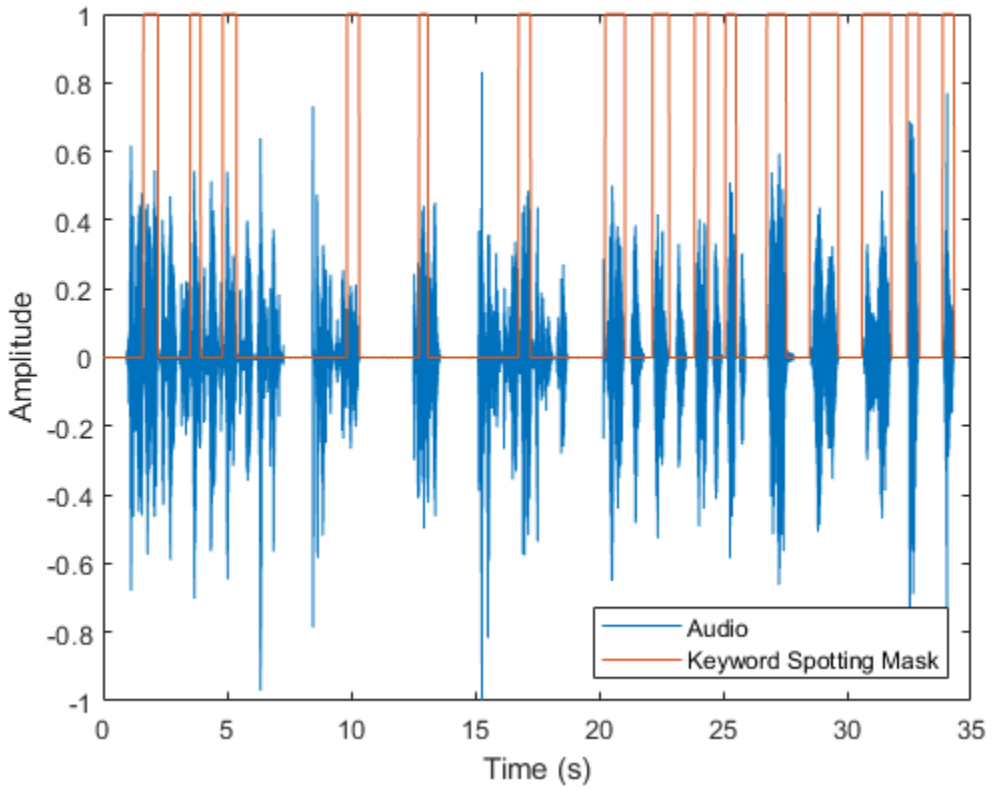
```
    5.57    6.07    "no"
    6.15    6.56    "driving"
      ⋮
```

The speech-to-text API returns the limits of the ROI labels in seconds. Use the `SpeechContent` table to create a logical vector.

```
keywordLabels = speechContent(speechContent.Value == "yes",:);
keywordROILimitsInSamples = round(keywordLabels.ROILimits*fs);

mask = zeros(size(audioIn),"logical");
for i = 1:size(keywordROILimitsInSamples)
    mask(keywordROILimitsInSamples(i,1):keywordROILimitsInSamples(i,2)) = true;
end
```

Plot the speech signal and the keyword spotting mask.

```
plot(t,audioIn, ...
     t,mask)
xlabel('Time (s)')
ylabel('Amplitude')
legend('Audio','Keyword Spotting Mask','Location','southeast')
```



Programmatic Use

`audioLabeler` opens the app, enabling you to label ground-truth data about audio.

Version History

Introduced in R2018b

R2023a: Audio Labeler has been removed

Errors starting in R2023a

The **Audio Labeler** app has been removed. Use **Signal Labeler** instead.

The **Signal Labeler** app:

- Replaces file-level labels with attribute labels to define full-signal characteristics.
- Uses different line colors for channels of an audio file by default.
- Does not expose label tags of label definitions. You cannot interactively see, add, or edit label tags for label definitions.
- Provides two new workflows when running autolabeling algorithms:
  - Label and inspect plotted audio files.
  - Label all audio files or a subset of audio files without inspection.
- Does not automatically create label definitions when running automation algorithms. To automate the detection of speech content, you must first add logical region-of-interest (ROI) label definitions. To perform speech-to-text transcription, you must first add string ROI label definitions.
- Requires automation algorithms be run on one channel at a time. For multichannel audio files, you can choose which channel to use as input.
- Does not provide audio recording.

R2022a: Warns

Warns starting in R2022a

The **Audio Labeler** app issues a warning that it will be removed in a future release.

See Also

Apps
Signal Labeler

Objects
`signalLabelDefinition` | `labeledSignalSet` | `audioDatastore` | `audioDeviceReader` | `audioDeviceWriter`

Impulse Response Measurer

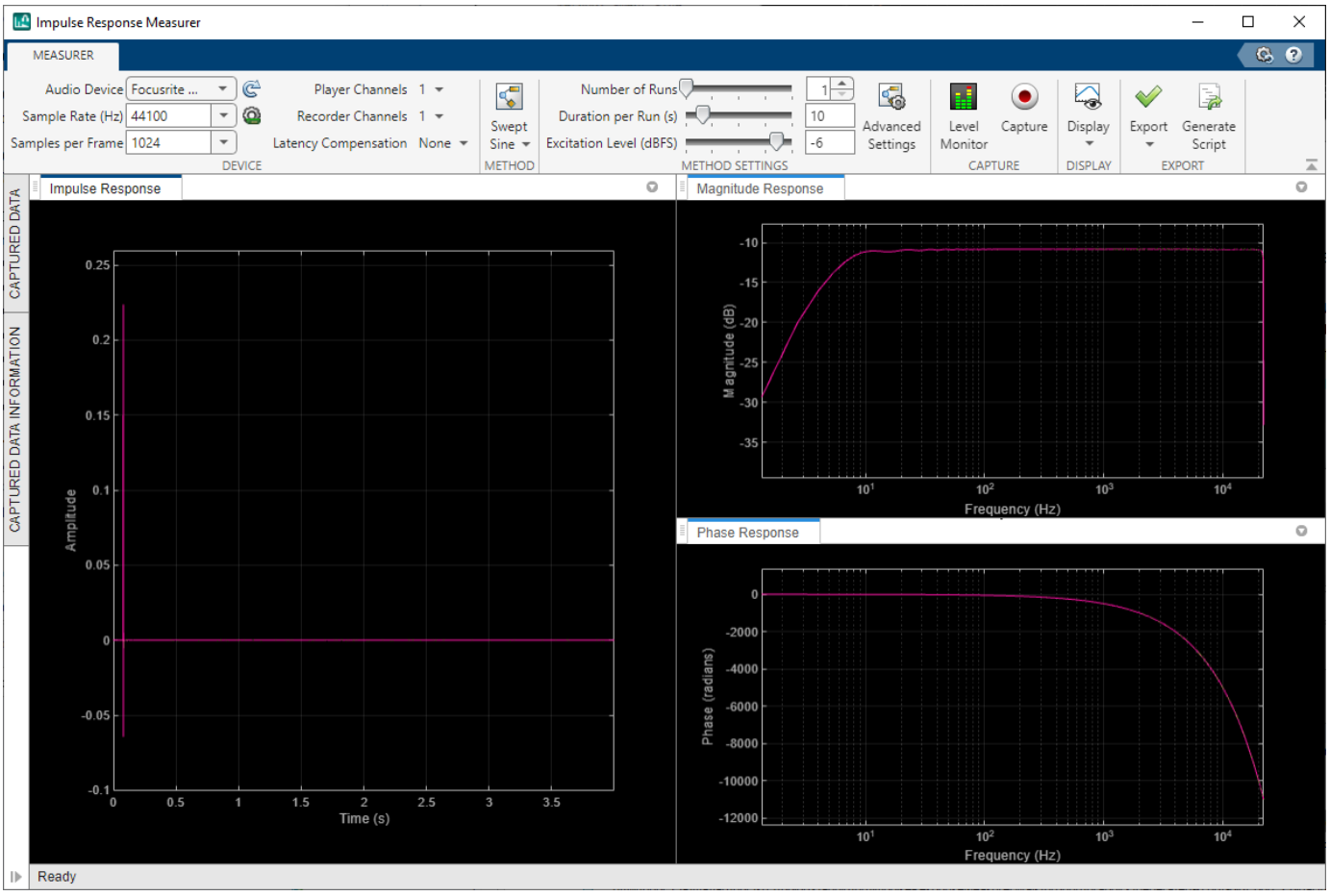
Measure impulse response of audio system

Description

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

Using this app, you can:

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



Open the Impulse Response Measurer App

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

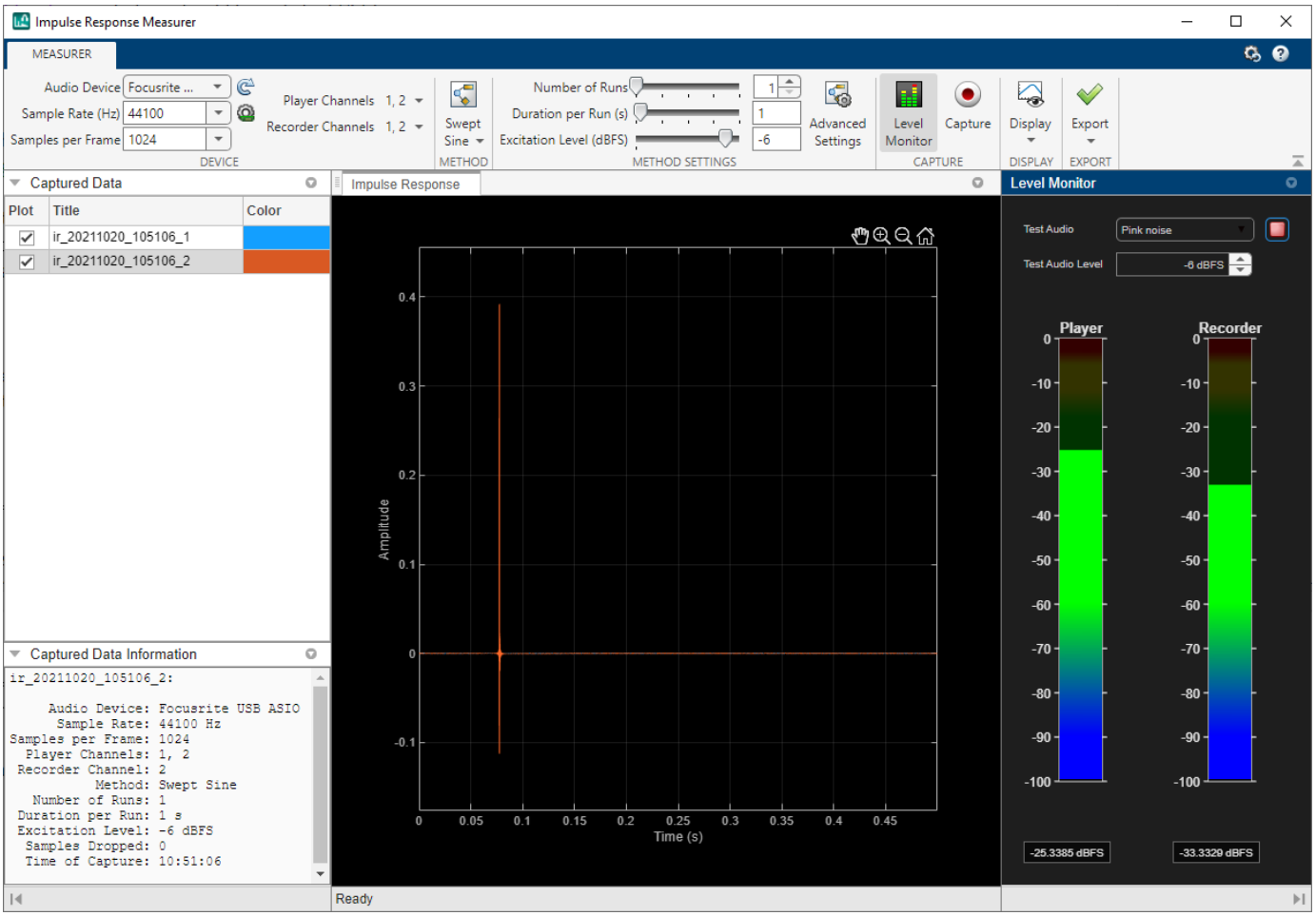
MATLAB Command prompt: Enter `impulseResponseMeasurer`.

Examples

Verify Input/Output Configuration

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

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



Choose player and recorder channels in the **Device** section of the toolstrip. Choose the test signal and the test audio level in the level monitor. Verify that the level reported by the recorder reacts

appropriately to level changes output by the player. Once you are satisfied that your system is configured correctly, close the level monitor and begin the impulse response capture.

- “Measure and Manage Impulse Responses”

Parameters

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

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

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

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

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

  The maximum total duration of both the sweep signal and the end silence is 60 seconds.

Version History

Introduced in R2018a

R2022b: Automatic latency compensation

Use a loopback cable to measure the audio device latency that delays the measured impulse response. You can optionally remove this latency from the captured measurements. Set **Latency Compensation** to `Loopback Measurement` to enable this feature.

R2022b: Exponential swept sine supports longer duration
Behavior changed in R2022b

When the **Method** is `Exponential Swept Sine`, the **Duration per Run (s)** must be less than 60 seconds. Previously, it had to be less than 15 seconds.

R2023a: Generate MATLAB code to measure impulse responses

Click the **Generate Script** button to generate MATLAB code that measures the impulse response according to the current settings of the app.

References

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

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

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

See Also

`audioPlayerRecorder` | `splMeter` | `reverberator`

Topics

“Measure and Manage Impulse Responses”

Audio Test Bench

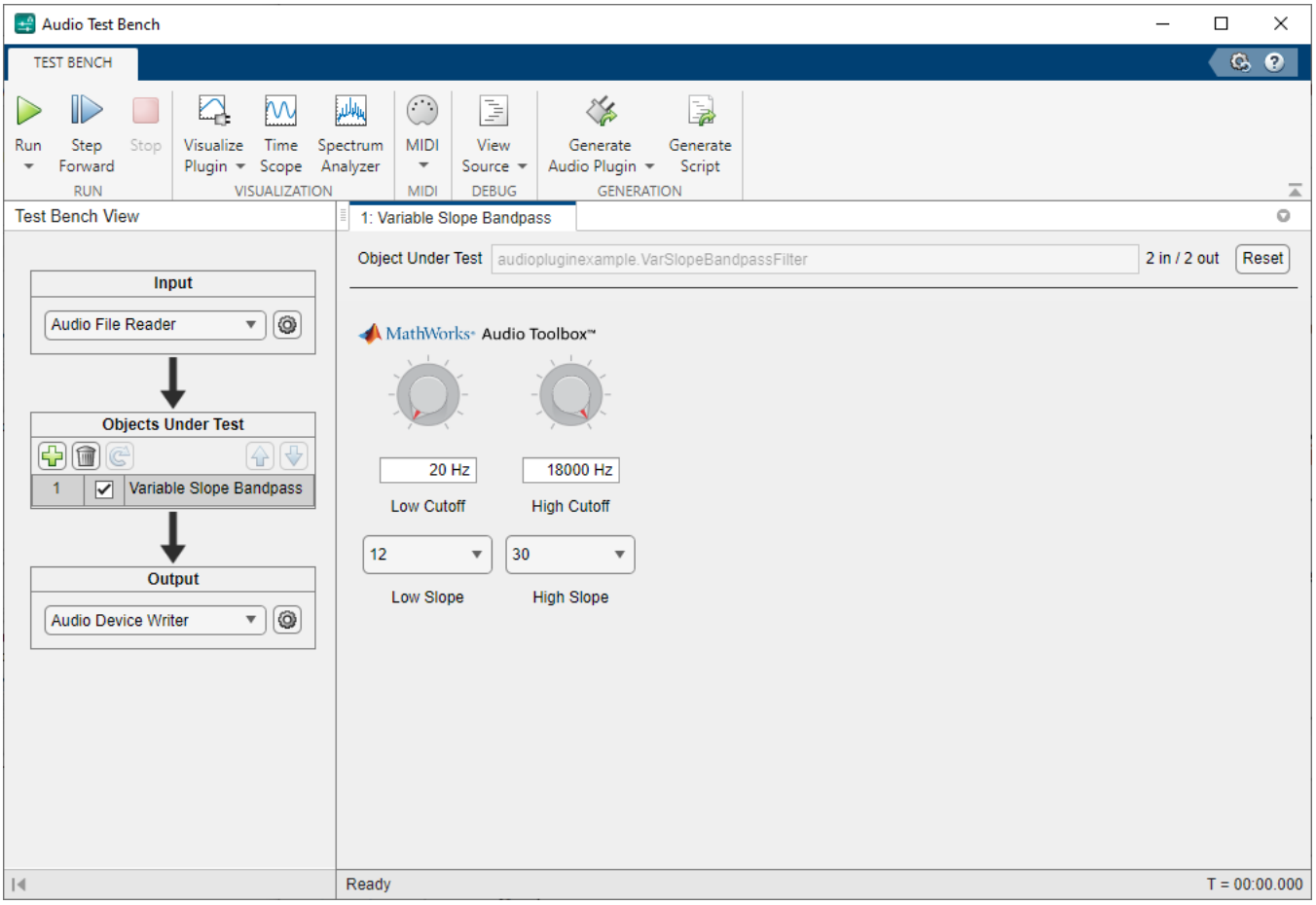
Debug, test, and tune audio plugins

Description

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

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

- Debug your audio plugins.
- Simulate your audio plugins 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 audio plugin binaries.



Open the Audio Test Bench App

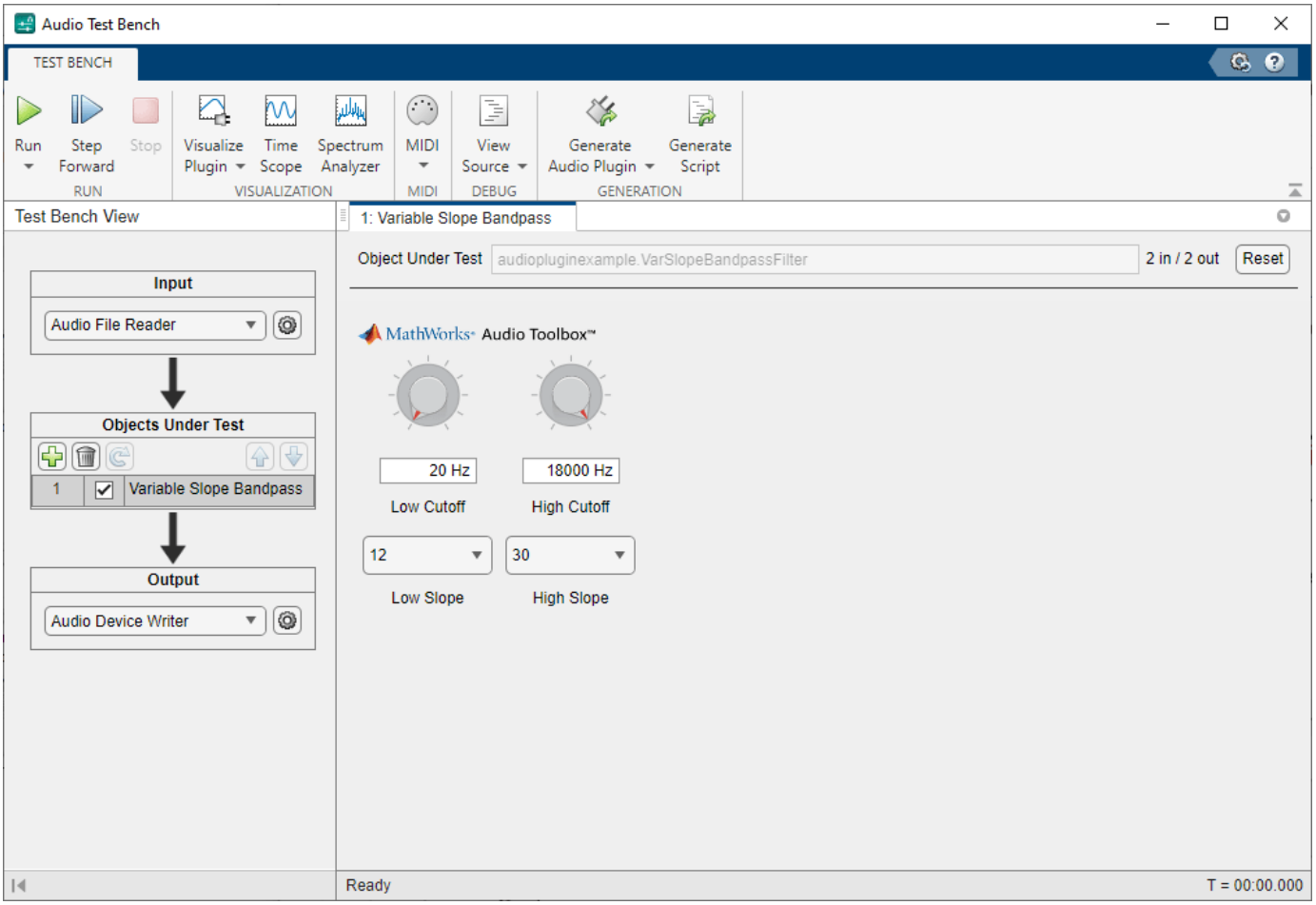
- MATLAB Toolstrip: On the **Apps** tab, under **Signal Processing and Communications**, click the app icon.
- MATLAB command prompt: Enter `audioTestBench`.

Examples

Open Audio Test Bench

Open the **Audio Test Bench** for an audio plugin class.

```
audioTestBench(audiopluginexample.VarSlopeBandpassFilter)
```



- “Develop, Analyze, and Debug Plugins In Audio Test Bench”

Programmatic Use

`audioTestBench(aClass)` opens the **Audio Test Bench** for an instance of `aClass`. Valid classes include:

- An audio plugin class that derives from `audioPlugin`, the base class for audio plugins.
- A compatible Audio Toolbox System object™.

`audioTestBench(aObject)` opens the **Audio Test Bench** for `aObject`. Valid objects include:

- An instance of an audio plugin class, where the class derives from `audioPlugin`, the base class for audio plugins.
- An instance of a compatible Audio Toolbox System object.
- A hosted plugin object, as 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.

`audioTestBench(plugin1,plugin2, ___ ,pluginN)` opens the **Audio Test Bench** with the sequence of plugins in a cascade.

`audioTestBench("-close")` closes the **Audio Test Bench**.

Tips

- The **Audio Test Bench** provides persistent input and output settings across sessions. You can reset all the app settings to their defaults by clicking the  button in the top right corner of the app.

Version History

Introduced in R2016a

R2023a: Support for cascaded objects under test

You can work with multiple audio plugins or System objects connected in a cascade configuration.

R2023a: Improvements to app settings and UI

The input and output settings are available in an easy-to-use format within the app. Some of the setting names have been changed slightly for clarity and simplicity.

Reset all the app settings to their defaults by clicking the reset button in the top right corner of the app.

R2022b: Choose from multiple plugin formats and specify coder configuration when generating audio plugin binaries

The **Audio Test Bench** has the same functionality as `generateAudioPlugin` and `audioPluginConfig` for generating audio plugin binaries. You can choose from multiple plugin binary formats, and you can specify the coder configuration for deep learning and code replacement libraries.

R2022b: Time Scope and Spectrum Analyzer can not be in the same window
Behavior changed in R2022b

Due to a change in interfaces for **Time Scope** and **Spectrum Analyzer**, you can only open each scope in its own separate window.

See Also

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

Classes
`audioPlugin` | `audioPluginSource`

Topics
“Develop, Analyze, and Debug Plugins In Audio Test Bench”
“What Are DAWs, Audio Plugins, and MIDI Controllers?”
“Design an Audio Plugin”
“Audio Plugins in MATLAB”
“Audio Plugin Example Gallery”

Extract Audio Features

Streamline audio feature extraction in the Live Editor

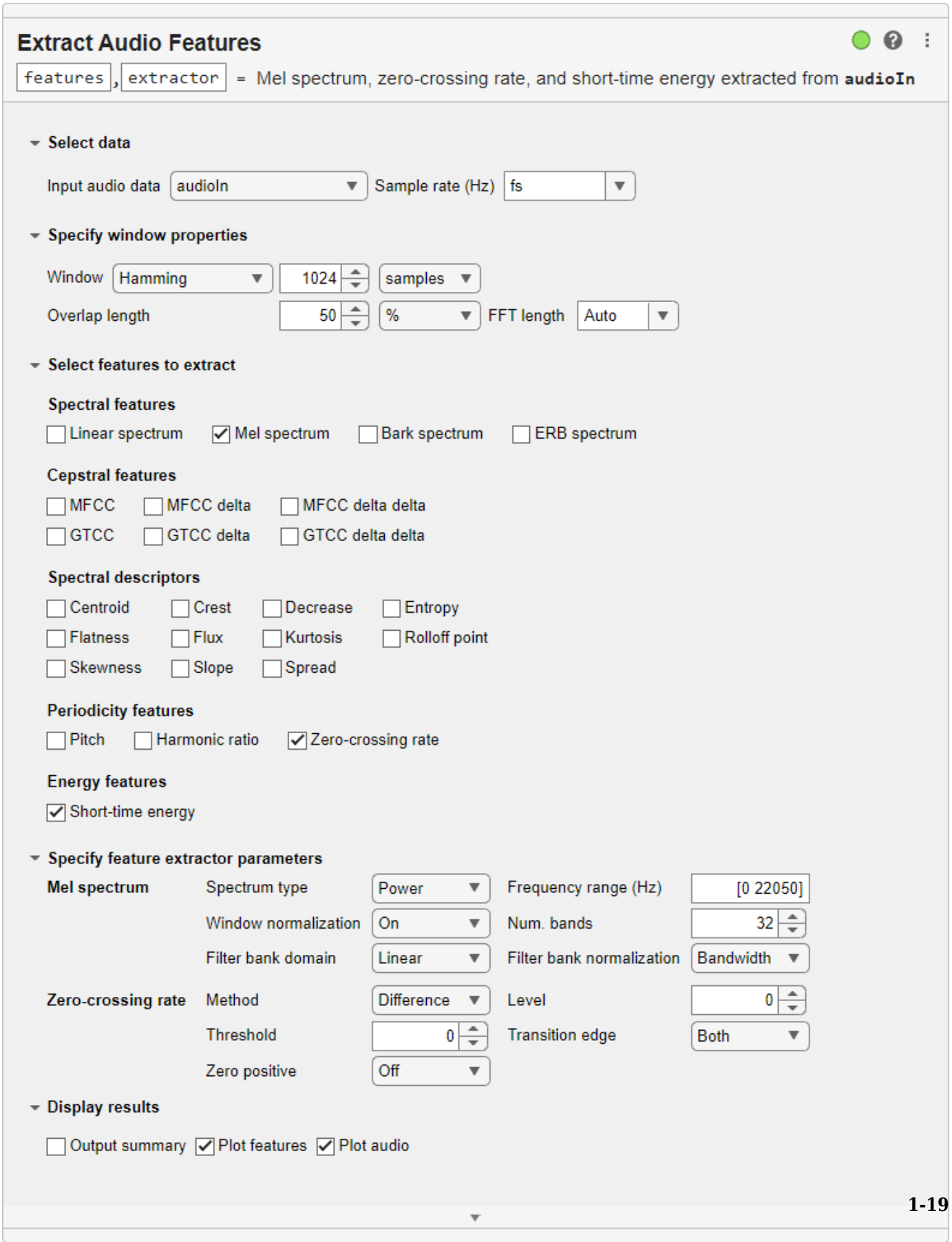
Description

The **Extract Audio Features** task enables you to configure an optimized feature extraction pipeline by selecting features and parameters graphically. You can reuse the output from **Extract Audio Features** to apply feature extraction to entire data sets. The task automatically generates MATLAB code for your live script.

Using this task, you can:

- Extract features of audio signals common to machine learning and deep learning workflows.
- Create a feature extraction object for use with large data sets.
- Visualize extracted audio features.

To learn more about interactive tasks in live scripts, see “Add Interactive Tasks to a Live Script”.


Extract Audio Features
features, extractor = Mel spectrum, zero-crossing rate, and short-time energy extracted from audioIn
Select data
Input audio data
audioIn
Sample rate (Hz)
fs
Specify window properties
Window
Hamming
1024
samples
Overlap length
50
%
FFT length
Auto
Select features to extract
Spectral features
Linear spectrum
Mel spectrum
Bark spectrum
ERB spectrum
Cepstral features
MFCC
MFCC delta
MFCC delta delta
GTCC
GTCC delta
GTCC delta delta
Spectral descriptors
Centroid
Crest
Decrease
Entropy
Flatness
Flux
Kurtosis
Rolloff point
Skewness
Slope
Spread
Periodicity features
Pitch
Harmonic ratio
Zero-crossing rate
Energy features
Short-time energy
Specify feature extractor parameters
Mel spectrum
Spectrum type
Power
Frequency range (Hz)
[0 22050]
Window normalization
On
Num. bands
32
Filter bank domain
Linear
Filter bank normalization
Bandwidth
Zero-crossing rate
Method
Difference
Level
0
Threshold
0
Transition edge
Both
Zero positive
Off
Display results
Output summary
Plot features
Plot audio

Open the Task

- On the **Live Editor** tab, select **Task > Extract Audio Features**.
- In a code block in the script, type a relevant keyword, such as `extract`, `audio`, or `feature`. Select **Extract Audio Features** from the suggested command completions.

Tips

The **Extract Audio Features** task provides a graphical user interface to configure an `audioFeatureExtractor` object. For details on the configuration parameters, see `audioFeatureExtractor`. The task accepts audio data in the same format as the input to the `extract` object function of `audioFeatureExtractor`.

Version History

Introduced in R2020a

R2022b: Visualize Audio Signal and Extracted Features

Select **Plot features** and **Plot audio** under the **Display results** section to visualize the extracted features and input audio signal, respectively.

See Also

`audioDataAugmenter` | `audioFeatureExtractor` | `audioDatastore`

Functions

speech2text

Transcribe speech signal to text

Syntax

```
transcript = speech2text(clientObj,audioIn,fs)
transcript = speech2text( ___ ,HTTPTimeout=timeout)
[transcript,rawOutput] = speech2text( ___ )
```

Description

`transcript = speech2text(clientObj,audioIn,fs)` transcribes speech in the input audio signal to text using a wav2vec 2.0 pretrained deep learning model or a third-party speech service.

Note To use `speech2text` with the third-party speech services, you must download the extended Audio Toolbox functionality from File Exchange. The File Exchange submission includes a tutorial to get started with the third-party services.

Using wav2vec 2.0 requires Deep Learning Toolbox™ and installing the pretrained model.

`transcript = speech2text( ___ ,HTTPTimeout=timeout)` specifies the time in seconds to wait for the initial server connection to the third-party speech service.

`[transcript,rawOutput] = speech2text( ___ )` also returns the unprocessed server output from the third-party speech service.

Examples

Download wav2vec 2.0 Network

Download and install the pretrained wav2vec 2.0 model for speech-to-text transcription.

Type `speechClient("wav2vec2.0")` into the command line. If the pretrained model for wav2vec 2.0 is not installed, the function provides a download link. To install the model, click the link to download the file and unzip it to a location on the MATLAB path.

Alternatively, execute the following commands to download the wav2vec 2.0 model, unzip it to your temporary directory, and then add it to your MATLAB path.

```
downloadFile = matlab.internal.examples.downloadSupportFile("audio","wav2vec2/wav2vec2-base-960.z
wav2vecLocation = fullfile(tempdir,"wav2vec");
unzip(downloadFile,wav2vecLocation)
addpath(wav2vecLocation)
```

Check that the installation is successful by typing `speechClient("wav2vec2.0")` into the command line. If the model is installed, then the function returns a `Wav2VecSpeechClient` object.

```
speechClient("wav2vec2.0")
```

```
ans =
  Wav2VecSpeechClient with properties:

    Segmentation: 'word'
      TimeStamps: 0
```

Perform Speech-to-Text Transcription

Read in an audio file containing speech and listen to it.

```
[y,fs] = audioread("speech_dft.wav");
soundsc(y,fs)
```

Create a `speechClient` object that uses the wav2vec 2.0 pretrained network. This requires installing the pretrained network. If the network is not installed, the function provides a link with instructions to download and install the pretrained model.

```
transcriber = speechClient("wav2vec2.0");
```

Use `speech2text` to obtain a transcription of the audio signal.

```
transcript = speech2text(transcriber,y,fs)
```

```
transcript=12×2 table
    Transcript     Confidence
    ___________    __________

    "the"           0.97605
    "discreet"      0.91927
    "fourier"       0.84546
    "transform"     0.89922
    "of"            0.66676
    "a"             0.50026
    "real"          0.88796
    "valued"        0.89913
    "signal"         0.8041
    "is"            0.53891
    "conjugate"     0.98438
    "symmetric"     0.89333
```

Input Arguments

`clientObj` — Client object

`speechClient` object

Client object, specified as an object returned by `speechClient`. The object is an interface to a pretrained wav2vec 2.0 model or to a third-party speech service.

Using `speech2text` with wav2vec 2.0 requires Deep Learning Toolbox and installing the pretrained wav2vec 2.0 model. If the model is not installed, calling `speechClient` with `"wav2vec2.0"` provides a link to download and install the model.

To use any of the third-party speech services, you must download the extended Audio Toolbox functionality from File Exchange. The File Exchange submission includes a tutorial to get started with the third-party services.

Example: `speechClient("wav2vec2.0")`

`audioIn` — Audio input
column vector

Audio input signal, specified as a column vector (single channel).

Data Types: `single` | `double`

`fs` — Sample rate (Hz)
positive scalar

Sample rate in Hz, specified as a positive scalar.

Data Types: `single` | `double`

`timeout` — Time to wait for server connection in seconds
positive scalar

Time to wait for initial server connection in seconds, specified as a positive scalar. This sets the `TimeOut` property of `clientObj`.

This argument applies only when `clientObj` interfaces with one of the third-party speech services.

Output Arguments

`transcript` — Speech transcript
table | string

Speech transcript of the input audio signal, returned as a table with a column containing the transcript and another column containing the associated confidence metrics. If the `Segmentation` property of `clientObj` is `"none"`, `speech2text` returns the transcript as a string.

The returned table can have additional columns depending on the `speechClient` properties and server options.

Data Types: `table` | `string`

`rawOutput` — Unprocessed server output
`ResponseMessage` | structure

Unprocessed server output, returned as a `matlab.net.http.ResponseMessage` object containing the HTTP response from the third-party speech service. If the third-party speech service is Amazon®, `speech2text` returns the server output as a structure.

This output argument does not apply if `clientObj` interfaces with the wav2vec 2.0 pretrained model.

Version History

Introduced in R2022b

References

[1] Baevski, Alexei, Henry Zhou, Abdelrahman Mohamed, and Michael Auli. “Wav2vec 2.0: A Framework for Self-Supervised Learning of Speech Representations,” 2020. https://doi.org/10.48550/ARXIV.2006.11477.

See Also

`speechClient` | **Signal Labeler**

yamnetPreprocess

Preprocess audio for YAMNet classification

Syntax

```
features = yamnetPreprocess(audioIn,fs)
features = yamnetPreprocess(audioIn,fs,'OverlapPercentage',OP)
```

Description

`features = yamnetPreprocess(audioIn,fs)` generates mel spectrograms from `audioIn` that can be fed to the YAMNet pretrained network.

`features = yamnetPreprocess(audioIn,fs,'OverlapPercentage',OP)` specifies the overlap percentage between consecutive audio frames.

For example, `features = yamnetPreprocess(audioIn,fs,'OverlapPercentage',75)` applies a 75% overlap between consecutive frames used to generate the spectrograms.

Examples

Download YAMNet

Download and unzip the Audio Toolbox™ model for YAMNet.

Type `yamnet` at the Command Window. If the Audio Toolbox model for YAMNet is not installed, then the function provides a link to the location of the network weights. To download the model, click the link. Unzip the file to a location on the MATLAB path.

Alternatively, execute the following commands to download and unzip the YAMNet model to your temporary directory.

```
downloadFolder = fullfile(tempdir,'YAMNetDownload');
loc = websave(downloadFolder,'https://ssd.mathworks.com/supportfiles/audio/yamnet.zip');
YAMNetLocation = tempdir;
unzip(loc,YAMNetLocation)
addpath(fullfile(YAMNetLocation,'yamnet'))
```

Check that the installation is successful by typing `yamnet` at the Command Window. If the network is installed, then the function returns a `SeriesNetwork` (Deep Learning Toolbox) object.

```
yamnet
```

```
ans =
  SeriesNetwork with properties:

         Layers: [86×1 nnet.cnn.layer.Layer]
     InputNames: {'input_1'}
    OutputNames: {'Sound'}
```

Load Pretrained YAMNet

Load a pretrained YAMNet convolutional neural network and examine the layers and classes.

Use `yamnet` to load the pretrained YAMNet network. The output net is a `SeriesNetwork` (Deep Learning Toolbox) object.

```
net = yamnet
```

```
net =
  SeriesNetwork with properties:

         Layers: [86×1 nnet.cnn.layer.Layer]
     InputNames: {'input_1'}
    OutputNames: {'Sound'}
```

View the network architecture using the `Layers` property. The network has 86 layers. There are 28 layers with learnable weights: 27 convolutional layers, and 1 fully connected layer.

```
net.Layers
```

```
ans =
  86×1 Layer array with layers:

     1   'input_1'                   Image Input             96×64×1 images
     2   'conv2d'                    Convolution             32 3×3×1 convolutions with stride
     3   'b'                         Batch Normalization     Batch normalization with 32 channe
     4   'activation'                ReLU                    ReLU
     5   'depthwise_conv2d'          Grouped Convolution     32 groups of 1 3×3×1 convolutions
     6   'L11'                       Batch Normalization     Batch normalization with 32 channe
     7   'activation_1'              ReLU                    ReLU
     8   'conv2d_1'                  Convolution             64 1×1×32 convolutions with stride
     9   'L12'                       Batch Normalization     Batch normalization with 64 channe
    10   'activation_2'              ReLU                    ReLU
    11   'depthwise_conv2d_1'        Grouped Convolution     64 groups of 1 3×3×1 convolutions
    12   'L21'                       Batch Normalization     Batch normalization with 64 channe
    13   'activation_3'              ReLU                    ReLU
    14   'conv2d_2'                  Convolution             128 1×1×64 convolutions with strid
    15   'L22'                       Batch Normalization     Batch normalization with 128 chann
    16   'activation_4'              ReLU                    ReLU
    17   'depthwise_conv2d_2'        Grouped Convolution     128 groups of 1 3×3×1 convolutions
    18   'L31'                       Batch Normalization     Batch normalization with 128 chann
    19   'activation_5'              ReLU                    ReLU
    20   'conv2d_3'                  Convolution             128 1×1×128 convolutions with stri
    21   'L32'                       Batch Normalization     Batch normalization with 128 chann
    22   'activation_6'              ReLU                    ReLU
    23   'depthwise_conv2d_3'        Grouped Convolution     128 groups of 1 3×3×1 convolutions
    24   'L41'                       Batch Normalization     Batch normalization with 128 chann
    25   'activation_7'              ReLU                    ReLU
    26   'conv2d_4'                  Convolution             256 1×1×128 convolutions with stri
    27   'L42'                       Batch Normalization     Batch normalization with 256 chann
    28   'activation_8'              ReLU                    ReLU
    29   'depthwise_conv2d_4'        Grouped Convolution     256 groups of 1 3×3×1 convolutions
    30   'L51'                       Batch Normalization     Batch normalization with 256 chann
```

```
31   'activation_9'              ReLU                    ReLU
32   'conv2d_5'                  Convolution             256 1×1×256 convolutions with stri
33   'L52'                       Batch Normalization     Batch normalization with 256 chann
34   'activation_10'             ReLU                    ReLU
35   'depthwise_conv2d_5'        Grouped Convolution     256 groups of 1 3×3×1 convolutions
36   'L61'                       Batch Normalization     Batch normalization with 256 chann
37   'activation_11'             ReLU                    ReLU
38   'conv2d_6'                  Convolution             512 1×1×256 convolutions with stri
39   'L62'                       Batch Normalization     Batch normalization with 512 chann
40   'activation_12'             ReLU                    ReLU
41   'depthwise_conv2d_6'        Grouped Convolution     512 groups of 1 3×3×1 convolutions
42   'L71'                       Batch Normalization     Batch normalization with 512 chann
43   'activation_13'             ReLU                    ReLU
44   'conv2d_7'                  Convolution             512 1×1×512 convolutions with stri
45   'L72'                       Batch Normalization     Batch normalization with 512 chann
46   'activation_14'             ReLU                    ReLU
47   'depthwise_conv2d_7'        Grouped Convolution     512 groups of 1 3×3×1 convolutions
48   'L81'                       Batch Normalization     Batch normalization with 512 chann
49   'activation_15'             ReLU                    ReLU
50   'conv2d_8'                  Convolution             512 1×1×512 convolutions with stri
51   'L82'                       Batch Normalization     Batch normalization with 512 chann
52   'activation_16'             ReLU                    ReLU
53   'depthwise_conv2d_8'        Grouped Convolution     512 groups of 1 3×3×1 convolutions
54   'L91'                       Batch Normalization     Batch normalization with 512 chann
55   'activation_17'             ReLU                    ReLU
56   'conv2d_9'                  Convolution             512 1×1×512 convolutions with stri
57   'L92'                       Batch Normalization     Batch normalization with 512 chann
58   'activation_18'             ReLU                    ReLU
59   'depthwise_conv2d_9'        Grouped Convolution     512 groups of 1 3×3×1 convolutions
60   'L101'                      Batch Normalization     Batch normalization with 512 chann
61   'activation_19'             ReLU                    ReLU
62   'conv2d_10'                 Convolution             512 1×1×512 convolutions with stri
63   'L102'                      Batch Normalization     Batch normalization with 512 chann
64   'activation_20'             ReLU                    ReLU
65   'depthwise_conv2d_10'       Grouped Convolution     512 groups of 1 3×3×1 convolutions
66   'L111'                      Batch Normalization     Batch normalization with 512 chann
67   'activation_21'             ReLU                    ReLU
68   'conv2d_11'                 Convolution             512 1×1×512 convolutions with stri
69   'L112'                      Batch Normalization     Batch normalization with 512 chann
70   'activation_22'             ReLU                    ReLU
71   'depthwise_conv2d_11'       Grouped Convolution     512 groups of 1 3×3×1 convolutions
72   'L121'                      Batch Normalization     Batch normalization with 512 chann
73   'activation_23'             ReLU                    ReLU
74   'conv2d_12'                 Convolution             1024 1×1×512 convolutions with str
75   'L122'                      Batch Normalization     Batch normalization with 1024 chan
76   'activation_24'             ReLU                    ReLU
77   'depthwise_conv2d_12'       Grouped Convolution     1024 groups of 1 3×3×1 convolution
78   'L131'                      Batch Normalization     Batch normalization with 1024 chan
79   'activation_25'             ReLU                    ReLU
80   'conv2d_13'                 Convolution             1024 1×1×1024 convolutions with st
81   'L132'                      Batch Normalization     Batch normalization with 1024 chan
82   'activation_26'             ReLU                    ReLU
83   'global_average_pooling2d'  Global Average Pooling  Global average pooling
84   'dense'                     Fully Connected         521 fully connected layer
85   'softmax'                   Softmax                 softmax
86   'Sound'                     Classification Output   crossentropyex with 'Speech' and 5
```

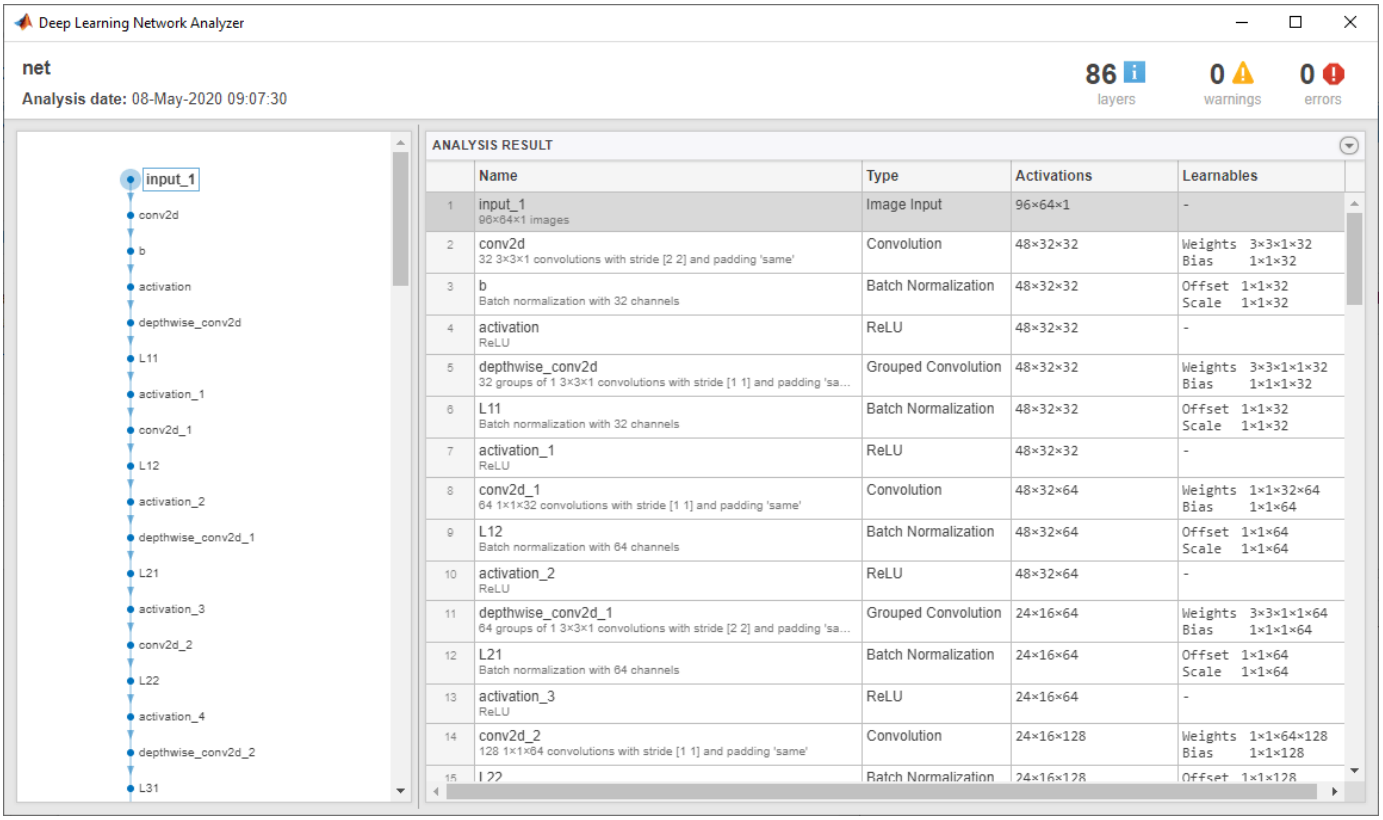
To view the names of the classes learned by the network, you can view the `Classes` property of the classification output layer (the final layer). View the first 10 classes by specifying the first 10 elements.

```
net.Layers(end).Classes(1:10)
```

```
ans = 10×1 categorical
     Speech
     Child speech, kid speaking
     Conversation
     Narration, monologue
     Babbling
     Speech synthesizer
     Shout
     Bellow
     Whoop
     Yell
```

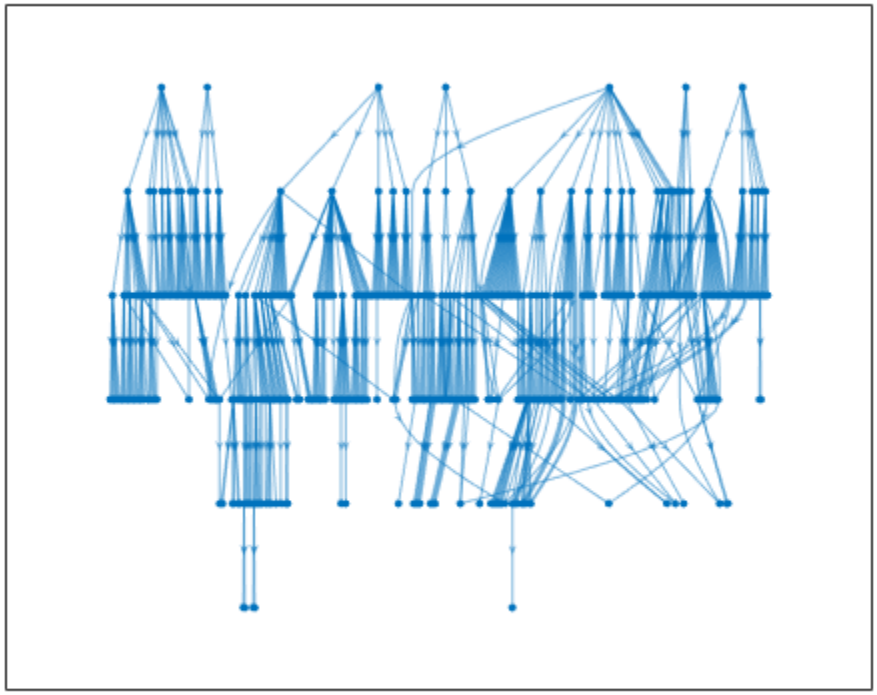
Use `analyzeNetwork` (Deep Learning Toolbox) to visually explore the network.

```
analyzeNetwork(net)
```



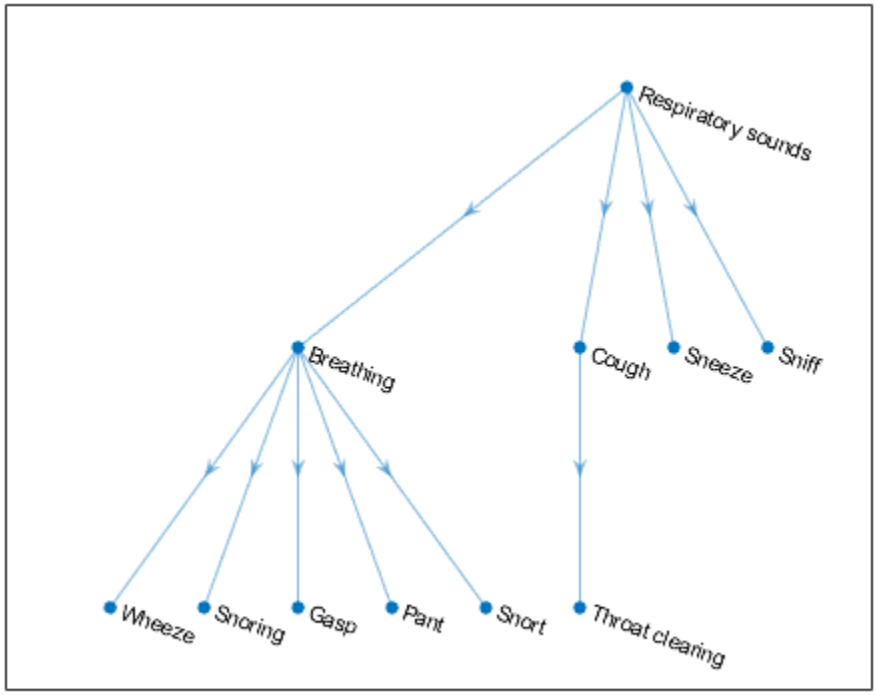
YAMNet was released with a corresponding sound class ontology, which you can explore using the `yamnetGraph` object.

```
ygraph = yamnetGraph;
p = plot(ygraph);
layout(p,'layered')
```



The ontology graph plots all 521 possible sound classes. Plot a subgraph of the sounds related to respiratory sounds.

```
allRespiratorySounds = dfsearch(ygraph,"Respiratory sounds");
ygraphSpeech = subgraph(ygraph,allRespiratorySounds);
plot(ygraphSpeech)
```



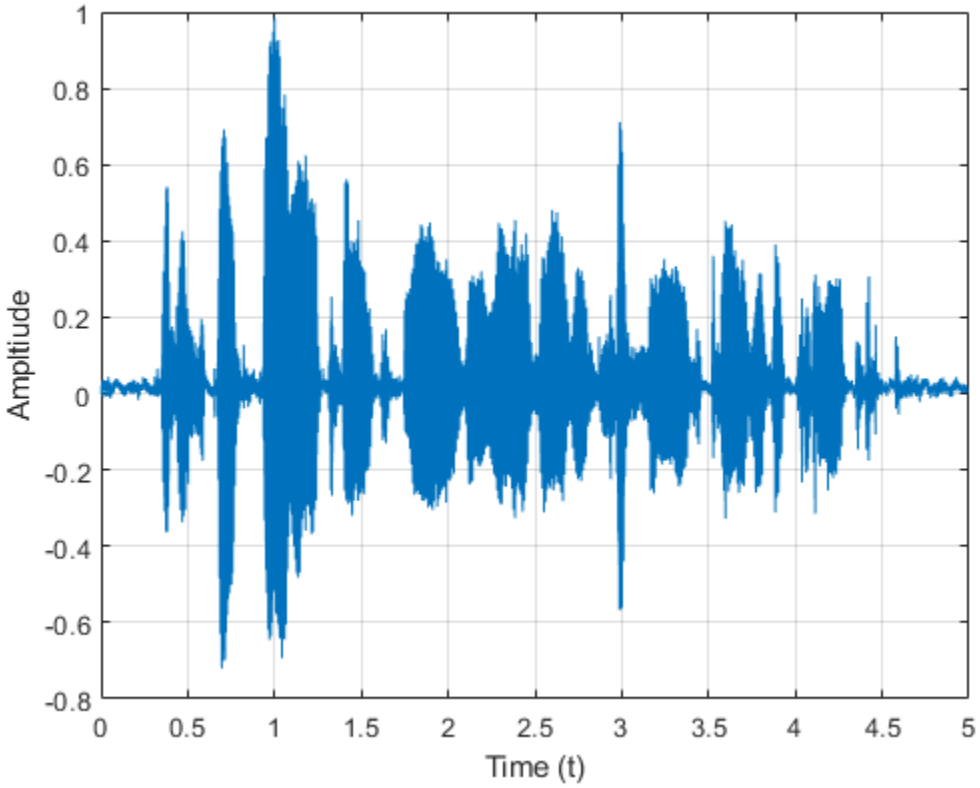
Preprocess Audio and Classify Sounds with YAMNet

Read in an audio signal.

```
[audioIn,fs] = audioread('SpeechDFT-16-8-mono-5secs.wav');
```

Plot and listen to the audio signal.

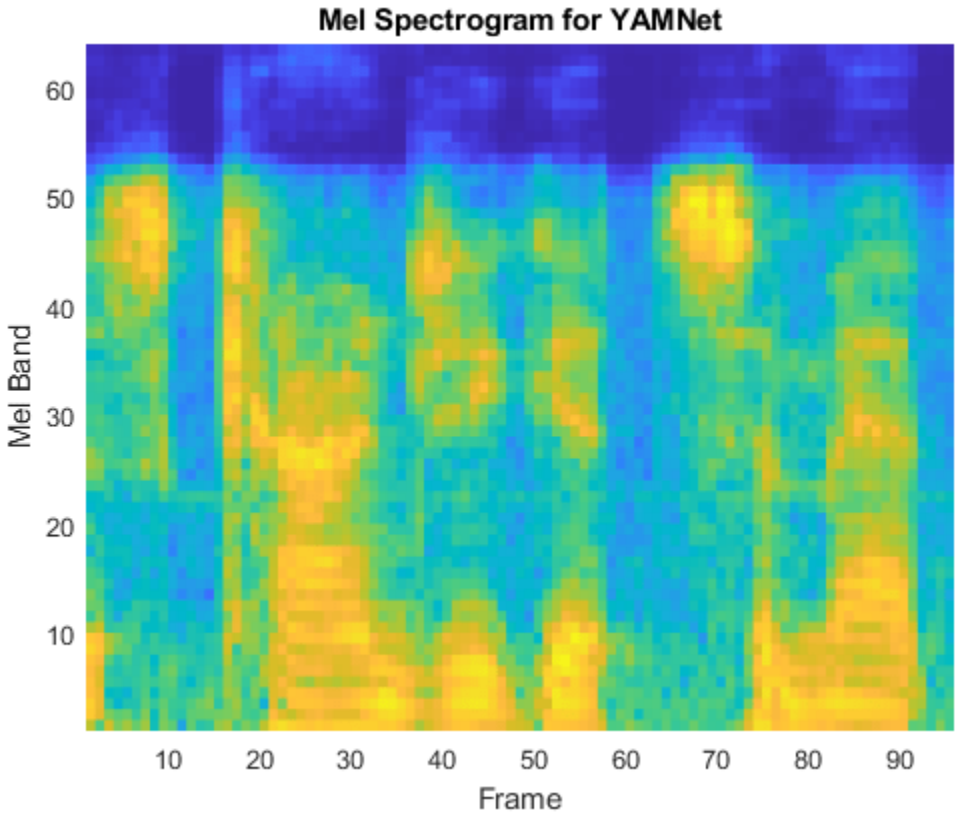
```
T = 1/fs;
t = 0:T:(length(audioIn)*T) - T;
plot(t,audioIn);
grid on
xlabel('Time (t)')
ylabel('Ampltiude')
```




```
soundsc(audioIn,fs)
```

Use `yamnetPreprocess` to extract mel spectrograms from the audio signal. Visualize an arbitrary spectrogram from the array.

```
melSpectYam = yamnetPreprocess(audioIn,fs);

arbSpect = melSpectYam(:,:,1,randi(size(melSpectYam,4)));
surf(arbSpect,'EdgeColor','none')
view([90,-90])
axis([1 size(arbSpect,1) 1 size(arbSpect,2)])
xlabel('Mel Band')
ylabel('Frame')
title('Mel Spectrogram for YAMNet')
axis tight
```



Create a YAMNet neural network (This requires Deep Learning Toolbox). Call `classify` with your YAMNet network and the preprocessed mel spectrogram images.

```
net = yamnet;
classes = classify(net,melSpectYam);
```

Classify the audio signal as the most frequently occurring sound.

```
mySound = mode(classes)
```

```
mySound = categorical
     Speech
```

Input Arguments

`audioIn` — Input signal

column vector | matrix

Input signal, specified as a column vector or matrix. If you specify a matrix, `yamnetPreprocess` treats the columns of the matrix as individual audio channels.

Data Types: `single` | `double`

`fs` — Sample rate (Hz)

positive scalar

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

Data Types: `single` | `double`

`OP` — Overlap percentage between consecutive mel spectrograms
`50` (default) | scalar in the range [0,100)

Percentage overlap between consecutive mel spectrograms, specified as a scalar in the range [0,100).

Data Types: `single` | `double`

Output Arguments

`features` — Mel spectrograms that can be fed to YAMNet pretrained network
96-by-64-by-1-by-*K* array

Mel spectrograms generated from `audioIn`, returned as a 96-by-64-by-1-by-*K* array, where:

- 96 –– Represents the number of 25 ms frames in each mel spectrogram
- 64 –– Represents the number of mel bands spanning 125 Hz to 7.5 kHz
- *K* –– Represents the number of mel spectrograms and depends on the length of `audioIn`, the number of channels in `audioIn`, as well as `OverlapPercentage`

Note Each 96-by-64-by-1 patch represents a single mel spectrogram image. For multichannel inputs, mel spectrograms are stacked along the fourth dimension.

Data Types: `single`

Version History

Introduced in R2021a

References

[1] Gemmeke, Jort F., et al. "Audio Set: An Ontology and Human-Labeled Dataset for Audio Events." *2017 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, IEEE, 2017, pp. 776–80. *DOI.org (Crossref)*, doi:10.1109/ICASSP.2017.7952261.

[2] Hershey, Shawn, et al. "CNN Architectures for Large-Scale Audio Classification." *2017 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, IEEE, 2017, pp. 131–35. *DOI.org (Crossref)*, doi:10.1109/ICASSP.2017.7952132.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

GPU Arrays
Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see "Run MATLAB Functions on a GPU" (Parallel Computing Toolbox).

See Also

Apps
Signal Labeler

Blocks
Sound Classifier | VGGish Embeddings | VGGish Preprocess | VGGish | YAMNet | YAMNet Preprocess

Functions
`classifySound` | `vggish` | `vggishEmbeddings` | `vggishPreprocess` | `yamnet` | `yamnetGraph`

vggishPreprocess

Preprocess audio for VGGish feature extraction

Syntax

```
features = vggishPreprocess(audioIn,fs)
features = vggishPreprocess(audioIn,fs,'OverlapPercentage',OP)
```

Description

`features = vggishPreprocess(audioIn,fs)` generates mel spectrograms from `audioIn` that can be fed to the VGGish pretrained network.

`features = vggishPreprocess(audioIn,fs,'OverlapPercentage',OP)` specifies the overlap percentage between consecutive audio frames.

For example, `vggishPreprocess(audioIn,fs,'OverlapPercentage',75)` applies a 75% overlap between consecutive frames used to generate the spectrograms.

Examples

Download VGGish Network

Download and unzip the Audio Toolbox™ model for VGGish.

Type `vggish` at the Command Window. If the Audio Toolbox model for VGGish is not installed, then the function provides a link to the location of the network weights. To download the model, click the link. Unzip the file to a location on the MATLAB path.

Alternatively, execute these commands to download and unzip the VGGish model to your temporary directory.

```
downloadFolder = fullfile(tempdir,'VGGishDownload');
loc = websave(downloadFolder,'https://ssd.mathworks.com/supportfiles/audio/vggish.zip');
VGGishLocation = tempdir;
unzip(loc,VGGishLocation)
addpath(fullfile(VGGishLocation,'vggish'))
```

Check that the installation is successful by typing `vggish` at the Command Window. If the network is installed, then the function returns a `SeriesNetwork` (Deep Learning Toolbox) object.

```
vggish
```

```
ans =
  SeriesNetwork with properties:

         Layers: [24×1 nnet.cnn.layer.Layer]
     InputNames: {'InputBatch'}
    OutputNames: {'regressionoutput'}
```

Load Pretrained VGGish Network

Load a pretrained VGGish convolutional neural network and examine the layers and classes.

Use `vggish` to load the pretrained VGGish network. The output `net` is a `SeriesNetwork` (Deep Learning Toolbox) object.

```
net = vggish
```

```
net =
  SeriesNetwork with properties:

         Layers: [24×1 nnet.cnn.layer.Layer]
     InputNames: {'InputBatch'}
    OutputNames: {'regressionoutput'}
```

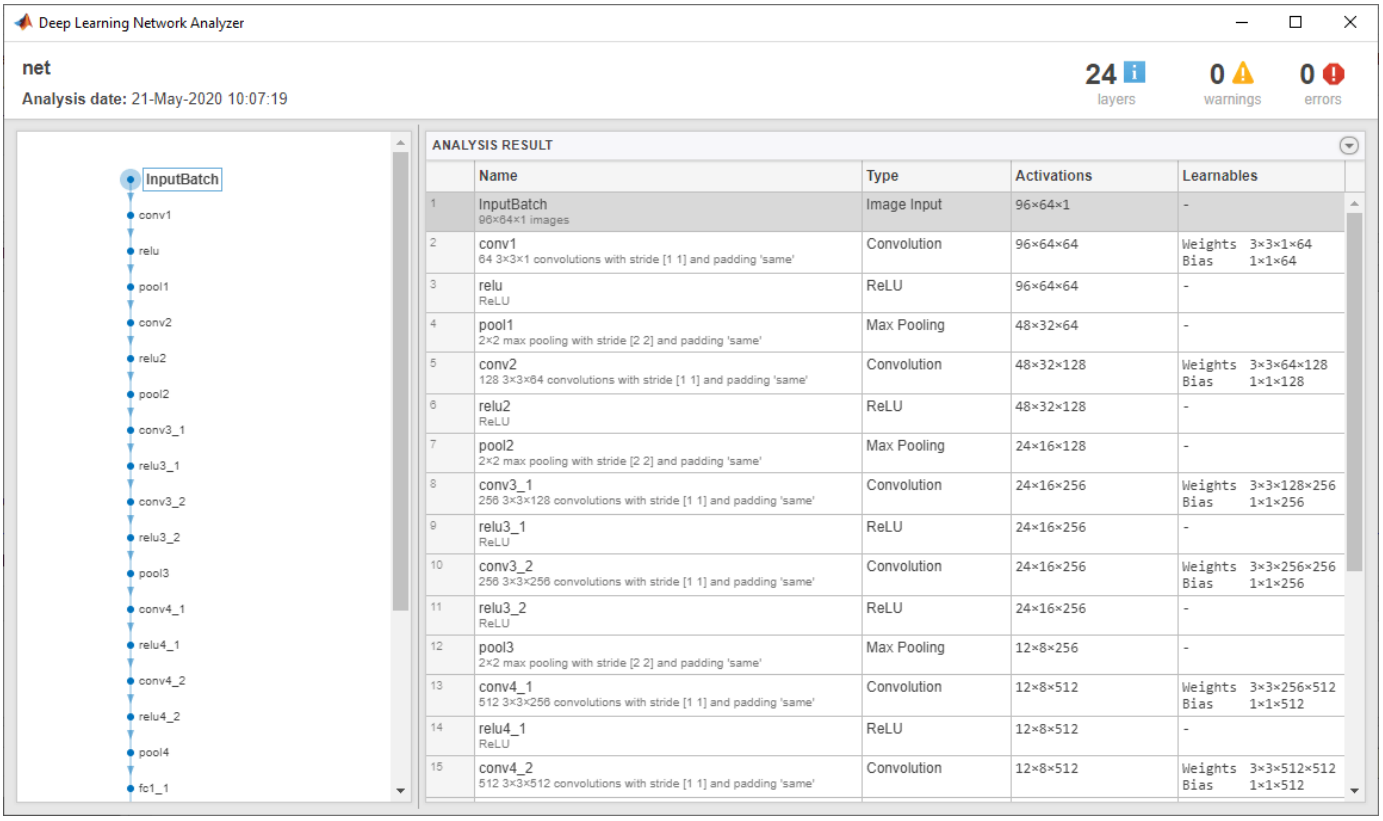
View the network architecture using the `Layers` property. The network has 24 layers. There are nine layers with learnable weights, of which six are convolutional layers and three are fully connected layers.

```
net.Layers
```

```
ans =
  24×1 Layer array with layers:

     1   'InputBatch'         Image Input         96×64×1 images
     2   'conv1'              Convolution         64 3×3×1 convolutions with stride [1  1] and pa
     3   'relu'               ReLU                ReLU
     4   'pool1'              Max Pooling         2×2 max pooling with stride [2  2] and padding
     5   'conv2'              Convolution         128 3×3×64 convolutions with stride [1  1] and
     6   'relu2'              ReLU                ReLU
     7   'pool2'              Max Pooling         2×2 max pooling with stride [2  2] and padding
     8   'conv3_1'            Convolution         256 3×3×128 convolutions with stride [1  1] and
     9   'relu3_1'            ReLU                ReLU
    10   'conv3_2'            Convolution         256 3×3×256 convolutions with stride [1  1] and
    11   'relu3_2'            ReLU                ReLU
    12   'pool3'              Max Pooling         2×2 max pooling with stride [2  2] and padding
    13   'conv4_1'            Convolution         512 3×3×256 convolutions with stride [1  1] and
    14   'relu4_1'            ReLU                ReLU
    15   'conv4_2'            Convolution         512 3×3×512 convolutions with stride [1  1] and
    16   'relu4_2'            ReLU                ReLU
    17   'pool4'              Max Pooling         2×2 max pooling with stride [2  2] and padding
    18   'fc1_1'              Fully Connected     4096 fully connected layer
    19   'relu5_1'            ReLU                ReLU
    20   'fc1_2'              Fully Connected     4096 fully connected layer
    21   'relu5_2'            ReLU                ReLU
    22   'fc2'                Fully Connected     128 fully connected layer
    23   'EmbeddingBatch'     ReLU                ReLU
    24   'regressionoutput'   Regression Output   mean-squared-error
```

Use `analyzeNetwork` (Deep Learning Toolbox) to visually explore the network.

```
analyzeNetwork(net)
```



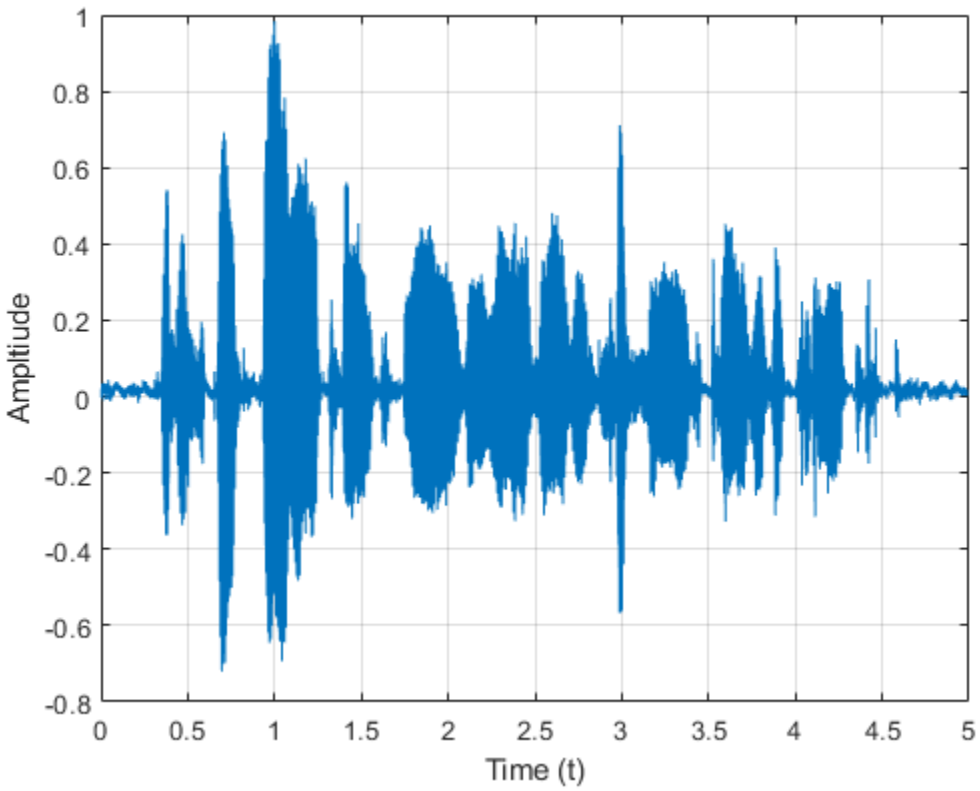
Extract Audio Embeddings with VGGish

Read in an audio signal.

```
[audioIn,fs] = audioread('SpeechDFT-16-8-mono-5secs.wav');
```

Plot and listen to the audio signal.

```
T = 1/fs;
t = 0:T:(length(audioIn)*T) - T;
plot(t,audioIn);
grid on
xlabel('Time (t)')
ylabel('Ampltiude')
```




```
soundsc(audioIn,fs)
```

Use `vggishPreprocess` to extract mel spectrograms from the audio signal.

```
melSpectVgg = vggishPreprocess(audioIn,fs);
```

Create a VGGish network (This requires Deep Learning Toolbox). Call `predict` to use your VGGish network for audio feature embedding extraction from the preprocessed mel spectrogram images. The feature embeddings are returned as a `numFrames`-by-128 matrix, where `numFrames` is the number of individual spectrograms, and 128 is the number of elements in each feature vector.

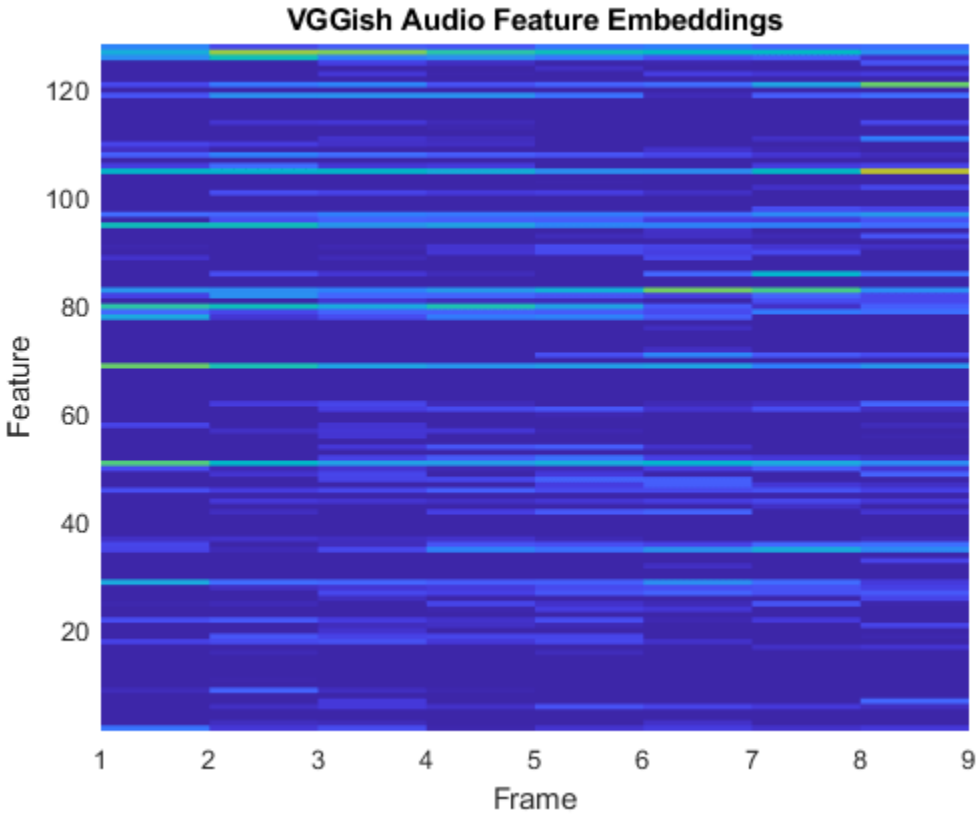
```
net = vggish;
embeddings = predict(net,melSpectVgg);
[numFrames,numFeatures] = size(embeddings)
```

```
numFrames = 9
```

```
numFeatures = 128
```

Visualize the VGGish feature embeddings.

```
surf(embeddings,'EdgeColor','none')
view([90,-90])
axis([1 numFeatures 1 numFrames])
xlabel('Feature')
ylabel('Frame')
title('VGGish Audio Feature Embeddings')
```



Input Arguments

`audioIn` — Input signal
column vector | matrix

Input signal, specified as a column vector or matrix. If you specify a matrix, `vggishPreprocess` treats the columns of the matrix as individual audio channels.

Data Types: `single` | `double`

`fs` — Sample rate (Hz)
positive scalar

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

Data Types: `single` | `double`

`OP` — Overlap percentage between consecutive mel spectrograms
`50` (default) | scalar in the range [0,100)

Percentage overlap between consecutive mel spectrograms, specified as a scalar in the range [0,100).

Data Types: `single` | `double`

Output Arguments

`features` — Mel spectrograms that can be fed to the VGGish pretrained network

96-by-64-by-1-by-*K* array

Mel spectrograms generated from `audioIn`, returned as a 96-by-64-by-1-by-*K* array, where:

- 96 –– Represents the number of 25 ms frames in each mel spectrogram.
- 64 –– Represents the number of mel bands spanning 125 Hz to 7.5 kHz.
- *K* –– Represents the number of mel spectrograms and depends on the length of `audioIn`, the number of channels in `audioIn`, as well as `OverlapPercentage`.

> **Note** Each 96-by-64-by-1 patch represents a single mel spectrogram image. For multichannel inputs, mel spectrograms are stacked along the 4th dimension.

Data Types: `single`

Version History

Introduced in R2021a

References

[1] Gemmeke, Jort F., et al. “Audio Set: An Ontology and Human-Labeled Dataset for Audio Events.” *2017 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, IEEE, 2017, pp. 776–80. *DOI.org (Crossref)*,doi:10.1109/ICASSP.2017.7952261.

[2] Hershey, Shawn, et al. “CNN Architectures for Large-Scale Audio Classification.” *2017 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, IEEE, 2017, pp. 131–35. *DOI.org (Crossref)*, doi:10.1109/ICASSP.2017.7952132.

Extended Capabilities

C/C++ Code Generation

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

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

See Also

Apps

Signal Labeler

Blocks

Sound Classifier | VGGish Embeddings | VGGish Preprocess | VGGish | YAMNet | YAMNet Preprocess

Functions

`classifySound` | `vggish` | `vggishEmbeddings` | `yamnet` | `yamnetGraph` | `yamnetPreprocess`

pitchnn

Estimate pitch with deep learning neural network

Syntax

```
f0 = pitchnn(audioIn,fs)
f0 = pitchnn(audioIn,fs,Name,Value)

[f0,loc] = pitchnn( ___ )
[f0,loc,activations] = pitchnn( ___ )

pitchnn( ___ )
```

Description

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

`f0 = pitchnn(audioIn,fs,Name,Value)` specifies options using one or more `Name,Value` arguments. For example, `f0 = pitchnn(audioIn,fs,'ConfidenceThreshold',0.5)` sets the confidence threshold for each value of `f0` to `0.5`.

`[f0,loc] = pitchnn( ___ )` returns the time values, `loc`, associated with each fundamental frequency estimate.

`[f0,loc,activations] = pitchnn( ___ )` returns the activations of a `crepe` pretrained network.

`pitchnn( ___ )` with no output arguments plots the estimated fundamental frequency over time.

Examples

Download CREPE Network

Download and unzip the Audio Toolbox™ model for CREPE.

Type `crepe` at the Command Window. If the Audio Toolbox model for CREPE is not installed, then the function provides a link to the location of the network weights. To download the model, click the link and unzip the file to a location on the MATLAB path.

Alternatively, execute these commands to download and unzip the CREPE model to your temporary directory.

```
downloadFolder = fullfile(tempdir,'crepeDownload');
loc = websave(downloadFolder,'https://ssd.mathworks.com/supportfiles/audio/crepe.zip');
crepeLocation = tempdir;
unzip(loc,crepeLocation)
addpath(fullfile(crepeLocation,'crepe'))
```

Check that the installation is successful by typing `crepe` at the Command Window. If the network is installed, then the function returns a `DAGNetwork` (Deep Learning Toolbox) object.

```
crepe
```

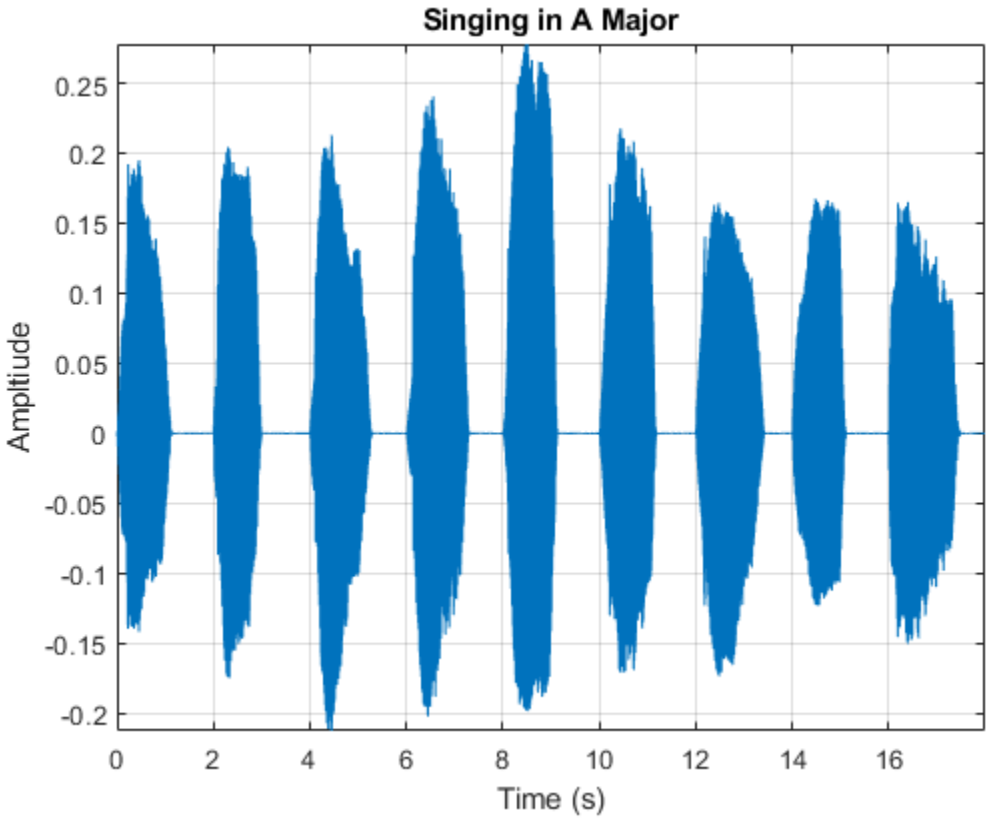
```
ans =
  DAGNetwork with properties:

         Layers: [34×1 nnet.cnn.layer.Layer]
    Connections: [33×2 table]
     InputNames: {'input'}
    OutputNames: {'pitch'}
```

Pitch Estimation with `pitchnn`

The CREPE network requires you to preprocess your audio signals to generate buffered, overlapped, and normalized audio frames that can be used as input to the network. This example demonstrates the `pitchnn` function performing all of these steps for you.

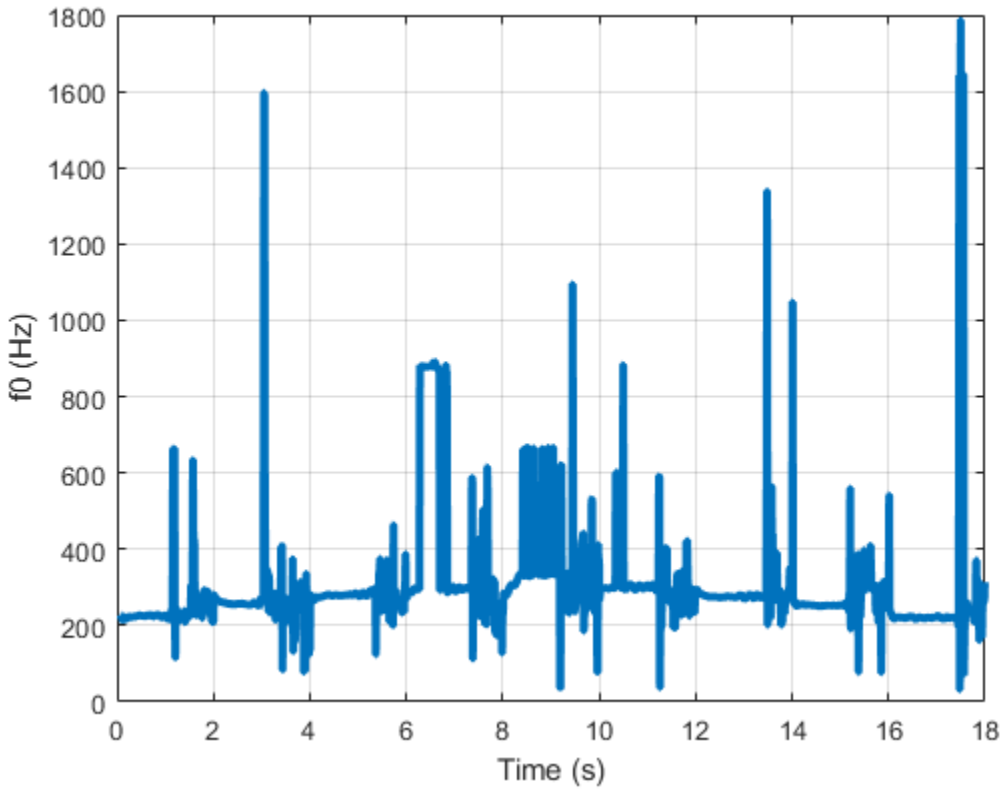
Read in an audio signal for pitch estimation. Visualize and listen to the audio. There are nine vocal utterances in the audio clip.

```
[audioIn,fs] = audioread('SingingAMajor-16-mono-18secs.ogg');
soundsc(audioIn,fs)
T = 1/fs;
t = 0:T:(length(audioIn)*T) - T;
plot(t,audioIn);
grid on
axis tight
xlabel('Time (s)')
ylabel('Ampltiude')
title('Singing in A Major')
```



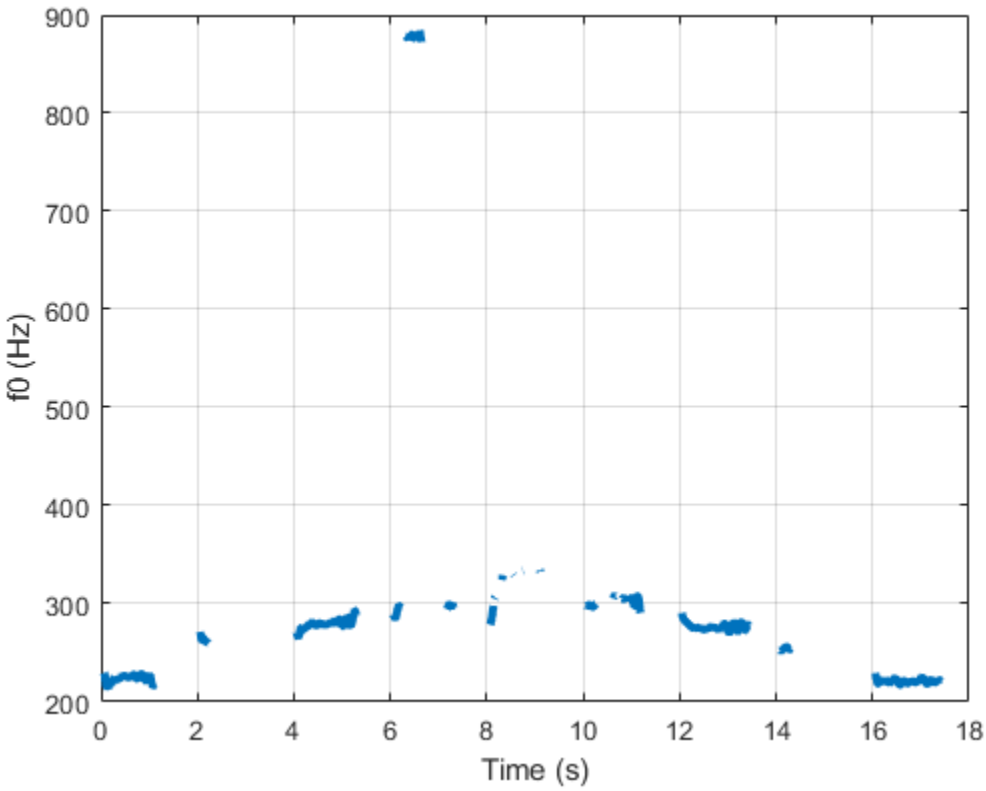
Use the `pitchnn` function to produce the pitch estimate using a CREPE network with `ModelCapacity` set to `tiny` and `ConfidenceThreshold` disabled. Calling `pitchnn` with no output arguments plots the pitch estimation over time. If you call `pitchnn` before downloading the model, an error is printed to the Command Window with a download link.

```
pitchnn(audioIn,fs,'ModelCapacity','tiny','ConfidenceThreshold',0)
```



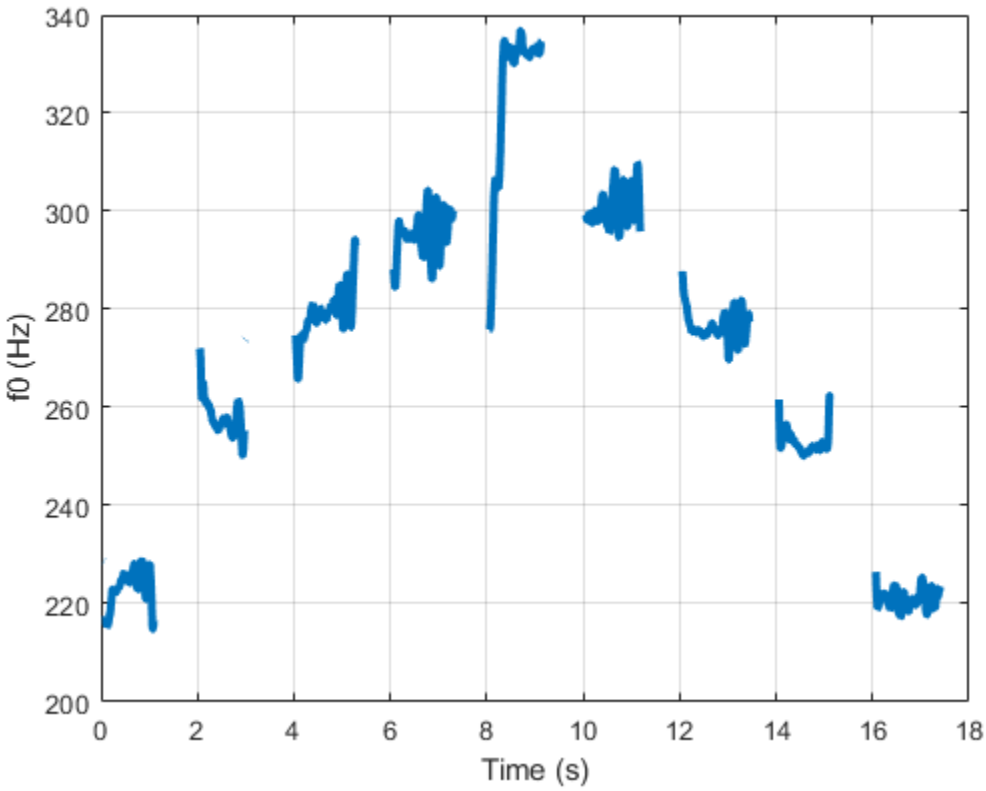
With confidence thresholding disabled, `pitchnn` provides a pitch estimate for every frame. Increase the `ConfidenceThreshold` to `0.8`.

```
pitchnn(audioIn,fs,'ModelCapacity','tiny','ConfidenceThreshold',0.8)
```



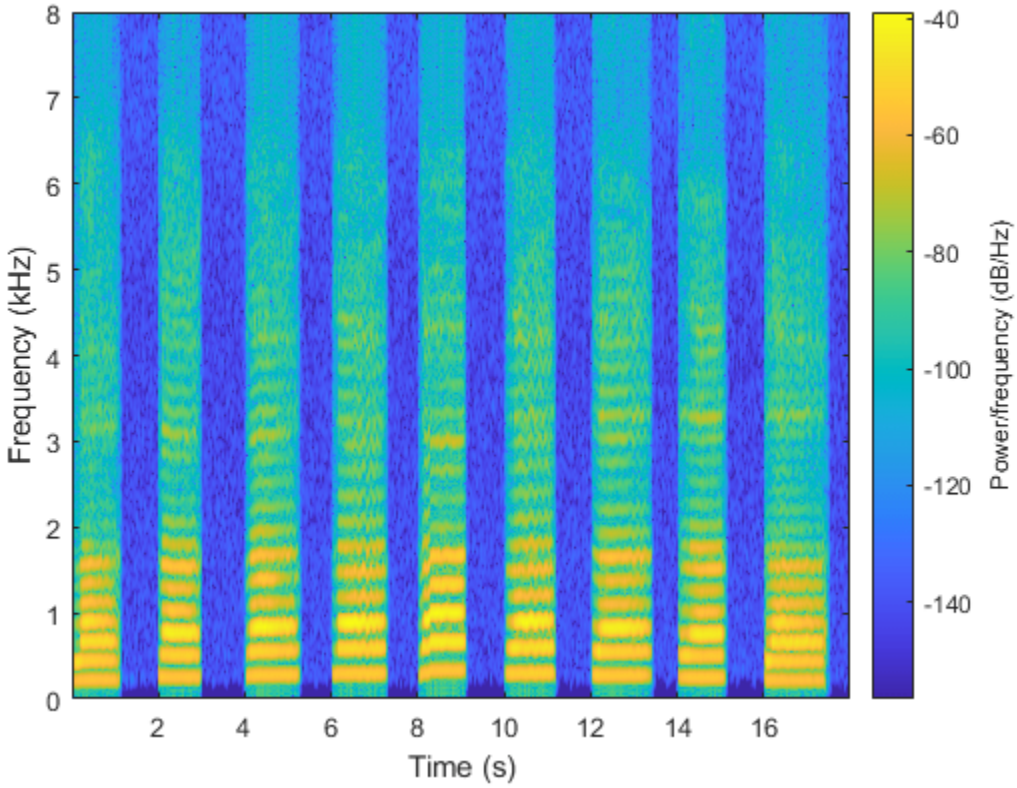
Call `pitchnn` with `ModelCapacity` set to `full`. There are nine primary pitch estimation groupings, each group corresponding with one of the nine vocal utterances.

```
pitchnn(audioIn,fs,'ModelCapacity','full','ConfidenceThreshold',0.8)
```



Call `spectrogram` and compare the frequency content of the signal with the pitch estimates from `pitchnn`. Use a frame size of `250` samples and an overlap of `225` samples or 90%. Use `4096` DFT points for the transform.

```
spectrogram(audioIn,250,225,4096,fs,'yaxis')
```



Input Arguments

`audioIn` — Input signal

column vector | matrix

Input signal, specified as a column vector or matrix. If you specify a matrix, `pitchnn` treats the columns of the matrix as individual audio channels.

Data Types: `single` | `double`

`fs` — Sample rate (Hz)

positive scalar

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

Data Types: `single` | `double`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `pitchnn(audioIn,fs,'OverlapPercentage',50)` sets the percent overlap between consecutive audio frames to 50.

`OverlapPercentage` — Overlap percentage between consecutive audio frames
`85` (default) | nonnegative scalar in the range [0,100)

Percentage overlap between consecutive audio frames, specified as a scalar in the range [0,100).

Data Types: `single` | `double`

`ConfidenceThreshold` — Confidence threshold
`0.5` (default) | nonnegative scalar in the range [0,1)

Confidence threshold for each value of `f0`, specified as a scalar in the range [0,1).

To disable threshold, set this argument to `0`.

Note If the maximum value of the corresponding `activations` vector is less than `'ConfidenceThreshold'`, `f0` is `NaN`.

Data Types: `single` | `double`

`ModelCapacity` — Model Capacity
`'full'` (default) | `'tiny'` | `'small'` | `'medium'` | `'large'`

Model capacity, specified as `'tiny'`, `'small'`, `'medium'`, `'large'`, or `'full'`.

Tip `'ModelCapacity'` controls the complexity of the underlying deep learning neural network. The higher the model capacity, the greater the number of nodes and layers in the model.

Data Types: `string` | `char`

Output Arguments

`f0` — Estimated fundamental frequency
N-by-*C* array

Estimated fundamental frequency in Hertz, returned as an *N*-by-*C* array, where *N* is the number of fundamental frequency estimates and *C* is the number of channels in `audioIn`.

Data Types: `single`

`loc` — Time values
1-by-*N* vector

Time values associated with each `f0` estimate, returned as a 1-by-*N* vector, where *N* is the number of fundamental frequency estimates. The time values correspond to the most recent samples used to compute the estimates.

Data Types: `single` | `double`

`activations` — CREPE network activations
N-by-360-by-*C* matrix

Activations from the CREPE network, returned as an *N*-by-`360`-by-*C* matrix, where *N* is the number of generated frames from the network and *C* is the number of channels in `audioIn`.

Data Types: `single` | `double`

Version History

Introduced in R2021a

References

[1] Kim, Jong Wook, Justin Salamon, Peter Li, and Juan Pablo Bello. “Crepe: A Convolutional Representation for Pitch Estimation.” In *2018 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 161–65. Calgary, AB: IEEE, 2018. https://doi.org/10.1109/ICASSP.2018.8461329.

Extended Capabilities

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

See Also

`crepe` | `crepePostprocess` | `crepePreprocess`

crepePostprocess

Postprocess output of CREPE deep learning network

Syntax

```
f0 = crepePostprocess(activations)
f0 = crepePostprocess(activations,'ConfidenceThreshold',TH)
```

Description

`f0 = crepePostprocess(activations)` converts the output of a `crepe` pretrained network to pitch estimates in Hz.

`f0 = crepePostprocess(activations,'ConfidenceThreshold',TH)` specifies the confidence threshold as a nonnegative scalar value less than 1.

For example, `f0 = crepePostprocess(actiations,'ConfidenceThreshold',0.75)` specifies a confidence threshold of 0.75.

Examples

Download CREPE Network

Download and unzip the Audio Toolbox™ model for CREPE.

Type `crepe` at the Command Window. If the Audio Toolbox model for CREPE is not installed, then the function provides a link to the location of the network weights. To download the model, click the link and unzip the file to a location on the MATLAB path.

Alternatively, execute these commands to download and unzip the CREPE model to your temporary directory.

```
downloadFolder = fullfile(tempdir,'crepeDownload');
loc = websave(downloadFolder,'https://ssd.mathworks.com/supportfiles/audio/crepe.zip');
crepeLocation = tempdir;
unzip(loc,crepeLocation)
addpath(fullfile(crepeLocation,'crepe'))
```

Check that the installation is successful by typing `crepe` at the Command Window. If the network is installed, then the function returns a `DAGNetwork` (Deep Learning Toolbox) object.

```
crepe
```

```
ans =
  DAGNetwork with properties:

         Layers: [34×1 nnet.cnn.layer.Layer]
    Connections: [33×2 table]
     InputNames: {'input'}
    OutputNames: {'pitch'}
```

Load Pretrained CREPE Network

Load a pretrained CREPE convolutional neural network and examine the layers and classes.

Use `crepe` to load the pretrained CREPE network. The output `net` is a `DAGNetwork` (Deep Learning Toolbox) object.

```
net = crepe
```

```
net =
  DAGNetwork with properties:

         Layers: [34×1 nnet.cnn.layer.Layer]
    Connections: [33×2 table]
     InputNames: {'input'}
    OutputNames: {'pitch'}
```

View the network architecture using the `Layers` property. The network has 34 layers. There are 13 layers with learnable weights, of which six are convolutional layers, six are batch normalization layers, and one is a fully connected layer.

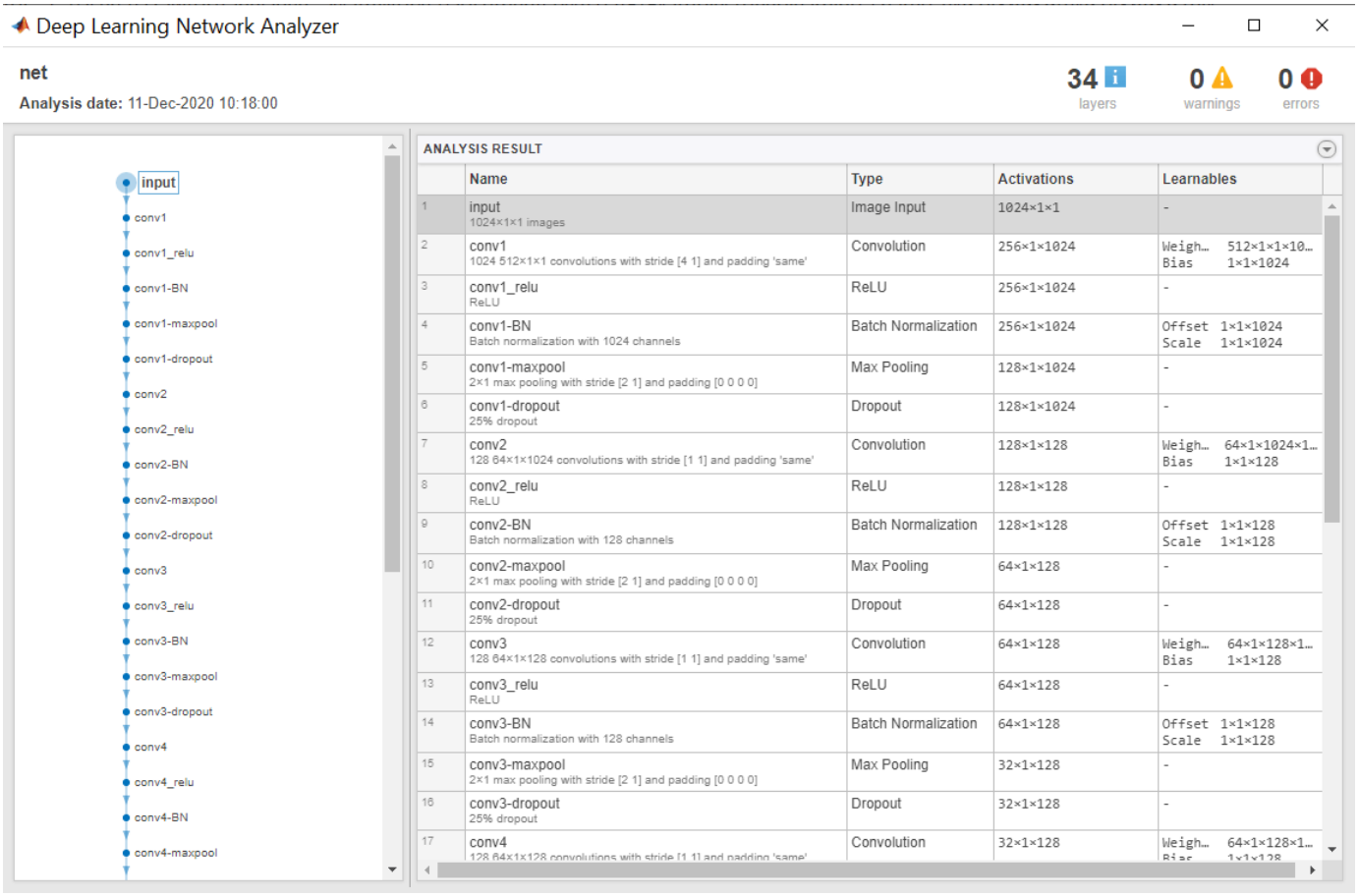
```
net.Layers
```

```
ans =
  34×1 Layer array with layers:

     1   'input'               Image Input           1024×1×1 images
     2   'conv1'               Convolution           1024 512×1×1 convolutions with stride [4  1
     3   'conv1_relu'          ReLU                  ReLU
     4   'conv1-BN'            Batch Normalization   Batch normalization with 1024 channels
     5   'conv1-maxpool'       Max Pooling           2×1 max pooling with stride [2  1] and pado
     6   'conv1-dropout'       Dropout               25% dropout
     7   'conv2'               Convolution           128 64×1×1024 convolutions with stride [1
     8   'conv2_relu'          ReLU                  ReLU
     9   'conv2-BN'            Batch Normalization   Batch normalization with 128 channels
    10   'conv2-maxpool'       Max Pooling           2×1 max pooling with stride [2  1] and pado
    11   'conv2-dropout'       Dropout               25% dropout
    12   'conv3'               Convolution           128 64×1×128 convolutions with stride [1  1
    13   'conv3_relu'          ReLU                  ReLU
    14   'conv3-BN'            Batch Normalization   Batch normalization with 128 channels
    15   'conv3-maxpool'       Max Pooling           2×1 max pooling with stride [2  1] and pado
    16   'conv3-dropout'       Dropout               25% dropout
    17   'conv4'               Convolution           128 64×1×128 convolutions with stride [1  1
    18   'conv4_relu'          ReLU                  ReLU
    19   'conv4-BN'            Batch Normalization   Batch normalization with 128 channels
    20   'conv4-maxpool'       Max Pooling           2×1 max pooling with stride [2  1] and pado
    21   'conv4-dropout'       Dropout               25% dropout
    22   'conv5'               Convolution           256 64×1×128 convolutions with stride [1  1
    23   'conv5_relu'          ReLU                  ReLU
    24   'conv5-BN'            Batch Normalization   Batch normalization with 256 channels
    25   'conv5-maxpool'       Max Pooling           2×1 max pooling with stride [2  1] and pado
    26   'conv5-dropout'       Dropout               25% dropout
    27   'conv6'               Convolution           512 64×1×256 convolutions with stride [1  1
    28   'conv6_relu'          ReLU                  ReLU
```

```
    29   'conv6-BN'              Batch Normalization   Batch normalization with 512 channels
    30   'conv6-maxpool'         Max Pooling           2×1 max pooling with stride [2  1] and pado
    31   'conv6-dropout'         Dropout               25% dropout
    32   'classifier'            Fully Connected       360 fully connected layer
    33   'classifier_sigmoid'    Sigmoid               sigmoid
    34   'pitch'                 Regression Output     mean-squared-error
```

Use `analyzeNetwork` (Deep Learning Toolbox) to visually explore the network.

```
analyzeNetwork(net)
```



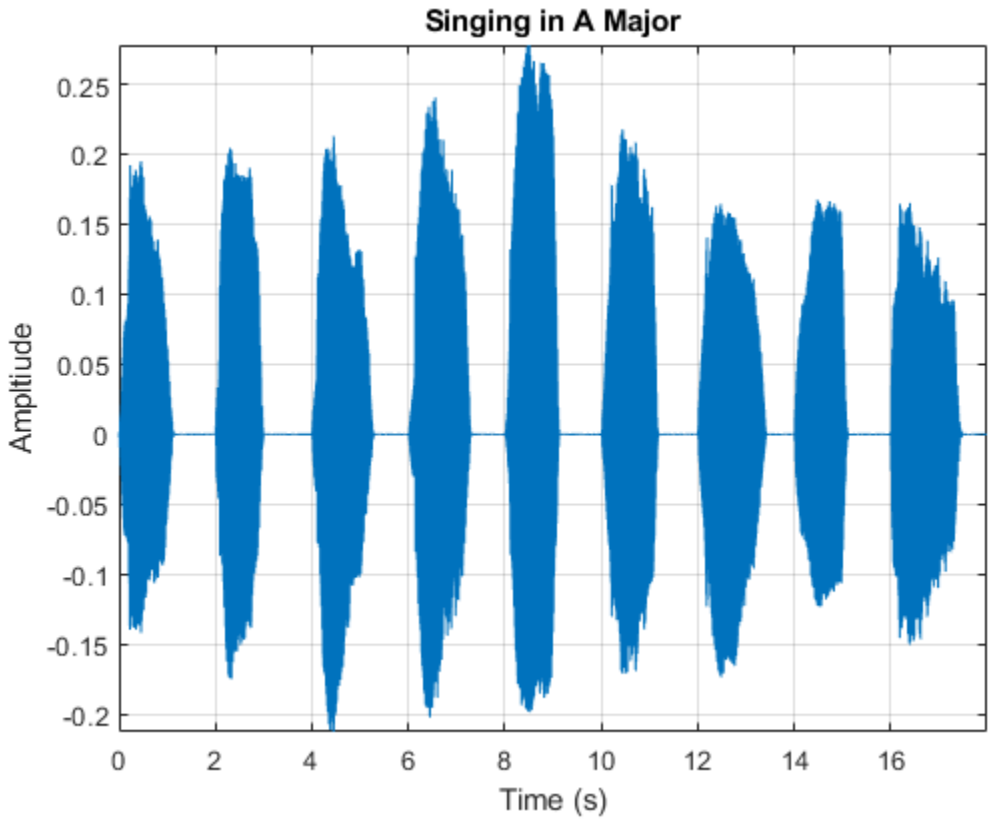
Estimate Pitch Using CREPE Network

The CREPE network requires you to preprocess your audio signals to generate buffered, overlapped, and normalized audio frames that can be used as input to the network. This example walks through audio preprocessing using `crepePreprocess` and audio postprocessing with pitch estimation using `crepePostprocess`. The `pitchnn` function performs these steps for you.

Read in an audio signal for pitch estimation. Visualize and listen to the audio. There are nine vocal utterances in the audio clip.

```
[audioIn,fs] = audioread('SingingAMajor-16-mono-18secs.ogg');
soundsc(audioIn,fs)
```

```
T = 1/fs;
t = 0:T:(length(audioIn)*T) - T;
plot(t,audioIn);
grid on
axis tight
xlabel('Time (s)')
ylabel('Ampltiude')
title('Singing in A Major')
```



Use `crepePreprocess` to partition the audio into frames of 1024 samples with an 85% overlap between consecutive mel spectrograms. Place the frames along the fourth dimension.

```
[frames,loc] = crepePreprocess(audioIn,fs);
```

Create a CREPE network with `ModelCapacity` set to `tiny`. If you call `crepe` before downloading the model, an error is printed to the Command Window with a download link.

```
netTiny = crepe('ModelCapacity','tiny');
```

Predict the network activations.

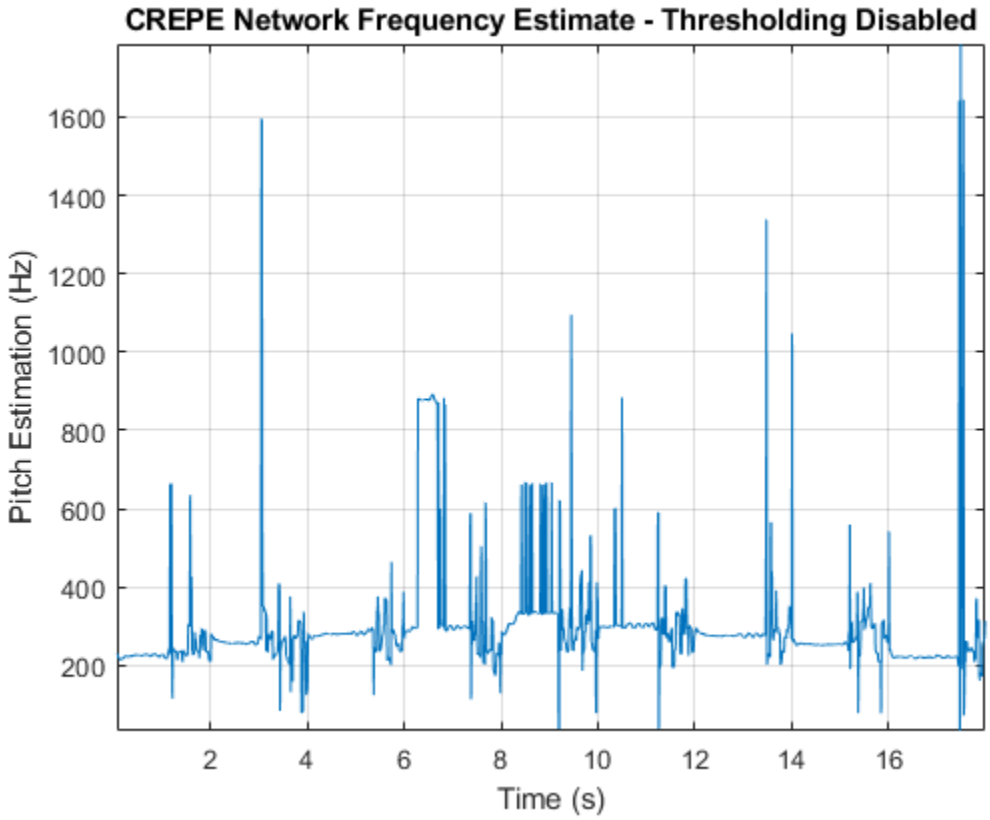
```
activationsTiny = predict(netTiny,frames);
```

Use `crepePostprocess` to produce the fundamental frequency pitch estimation in Hz. Disable confidence thresholding by setting `ConfidenceThreshold` to `0`.

```
f0Tiny = crepePostprocess(activationsTiny,'ConfidenceThreshold',0);
```

Visualize the pitch estimation over time.

```
plot(loc,f0Tiny)
grid on
axis tight
xlabel('Time (s)')
ylabel('Pitch Estimation (Hz)')
title('CREPE Network Frequency Estimate - Thresholding Disabled')
```

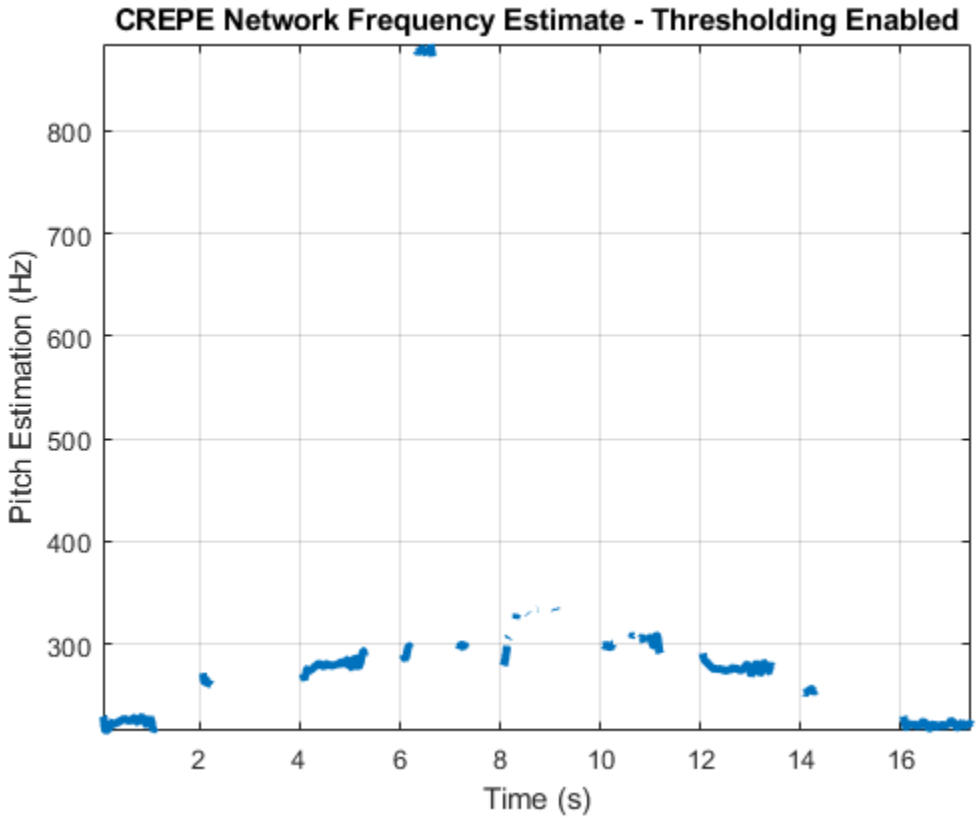


With confidence thresholding disabled, `crepePostprocess` provides a pitch estimate for every frame. Increase the `ConfidenceThreshold` to `0.8`.

```
f0Tiny = crepePostprocess(activationsTiny,'ConfidenceThreshold',0.8);
```

Visualize the pitch estimation over time.

```
plot(loc,f0Tiny,'LineWidth',3)
grid on
axis tight
xlabel('Time (s)')
ylabel('Pitch Estimation (Hz)')
title('CREPE Network Frequency Estimate - Thresholding Enabled')
```



Create a new CREPE network with `ModelCapacity` set to `full`.

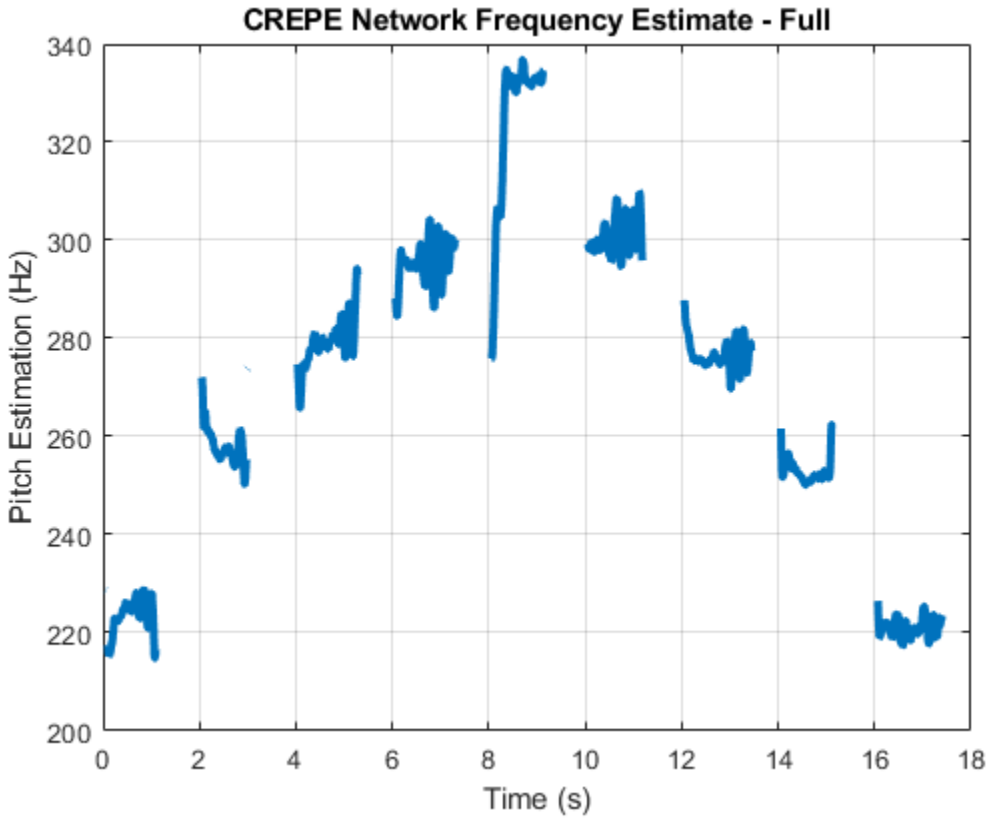
```
netFull = crepe('ModelCapacity','full');
```

Predict the network activations.

```
activationsFull = predict(netFull,frames);
f0Full = crepePostprocess(activationsFull,'ConfidenceThreshold',0.8);
```

Visualize the pitch estimation. There are nine primary pitch estimation groupings, each group corresponding with one of the nine vocal utterances.

```
plot(loc,f0Full,'LineWidth',3)
grid on
xlabel('Time (s)')
ylabel('Pitch Estimation (Hz)')
title('CREPE Network Frequency Estimate - Full')
```



Find the time elements corresponding to the last vocal utterance.

```
roundedLocVec = round(loc,2);
lastUtteranceBegin = find(roundedLocVec == 16);
lastUtteranceEnd = find(roundedLocVec == 18);
```

For simplicity, take the most frequently occurring pitch estimate within the utterance group as the fundamental frequency estimate for that timespan. Generate a pure tone with a frequency matching the pitch estimate for the last vocal utterance.

```
lastUtteranceEstimation = mode(f0Full(lastUtteranceBegin:lastUtteranceEnd))
```

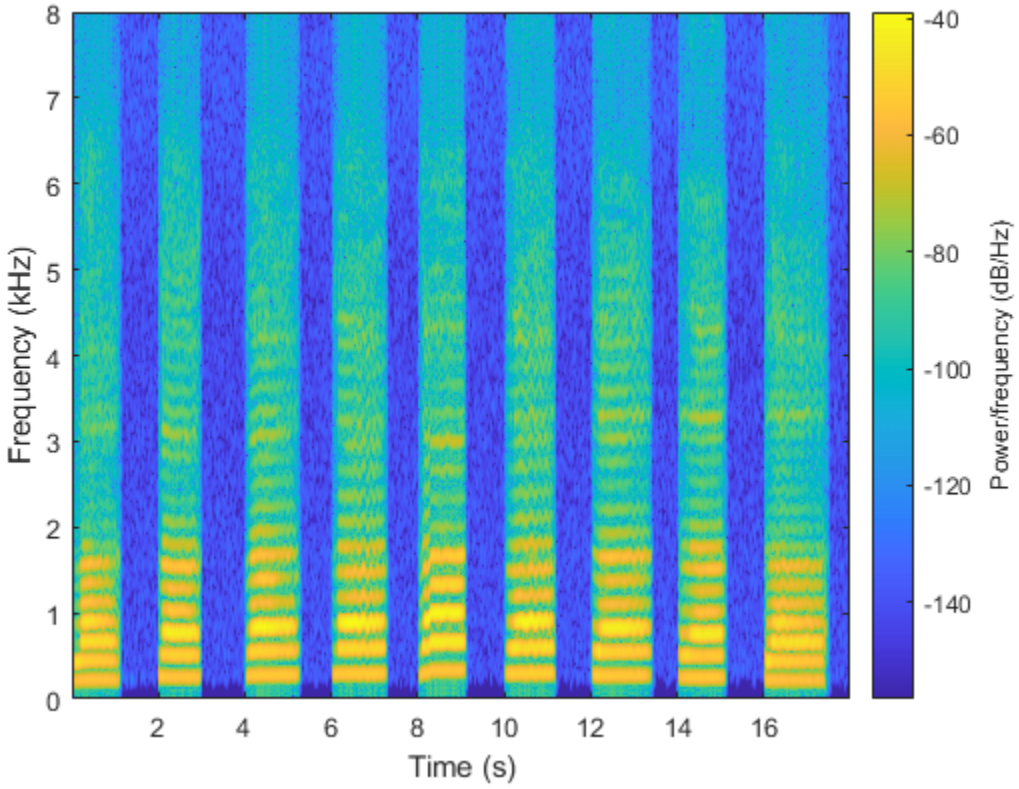
```
lastUtteranceEstimation = single
    217.2709
```

The value for `lastUtteranceEstimate` of `217.3` Hz. corresponds to the note A3. Overlay the synthesized tone on the last vocal utterance to audibly compare the two.

```
lastVocalUtterance = audioIn(fs*16:fs*18);
newTime = 0:T:2;
compareTone = cos(2*pi*lastUtteranceEstimation*newTime).';

soundsc(lastVocalUtterance + compareTone,fs);
```

Call `spectrogram` to more closely inspect the frequency content of the singing. Use a frame size of `250` samples and an overlap of `225` samples or 90%. Use `4096` DFT points for the transform. The `spectrogram` reveals that the vocal recording is actually a set of complex harmonic tones composed of multiple frequencies.

```
spectrogram(audioIn,250,225,4096,fs,'yaxis')
```



Input Arguments

`activations` — CREPE network output

N-by-`360` matrix

Audio frames generated from a `crepe` pretrained network, specified as an *N*-by-`360` matrix, where *N* is the number of generated frames.

Data Types: `single` | `double`

`TH` — Confidence threshold

`0.5` (default) | nonnegative scalar in the range [0,1)

Confidence threshold for each value of `f0`, specified as the comma-separated pair consisting of `'ConfidenceThreshold'` and a scalar in the range [0,1).

To disable thresholding, set `TH` to `0`.

Note If the maximum value of the corresponding `activations` vector is less than `TH`, `f0` is `NaN`.

Data Types: `single` | `double`

Output Arguments

`f0` — Estimated fundamental frequency

N-by-1 vector

Estimated fundamental frequency in Hertz, returned as an *N*-by-1 vector, where *N* is the number of generated frames.

Data Types: `single`

Version History

Introduced in R2021a

References

[1] Kim, Jong Wook, Justin Salamon, Peter Li, and Juan Pablo Bello. “Crepe: A Convolutional Representation for Pitch Estimation.” In *2018 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 161–65. Calgary, AB: IEEE, 2018. https://doi.org/10.1109/ICASSP.2018.8461329.

Extended Capabilities

C/C++ Code Generation

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

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

See Also

`crepe` | `pitchnn` | `crepePreprocess`

crepePreprocess

Preprocess audio for CREPE deep learning network

Syntax

```
frames = crepePreprocess(audioIn,fs)
frames = crepePreprocess(audioIn,fs,'OverlapPercentage',OP)

[frames,loc] = crepePreprocess( ___ )
```

Description

`frames = crepePreprocess(audioIn,fs)` generates frames from `audioIn` that can be fed to the CREPE pretrained deep learning network.

`frames = crepePreprocess(audioIn,fs,'OverlapPercentage',OP)` specifies the overlap percentage between consecutive audio frames.

For example, `frames = crepePreprocess(audioIn,fs,'OverlapPercentage',75)` applies a 75% overlap between consecutive frames used to generate the processed frames.

`[frames,loc] = crepePreprocess( ___ )` returns the time values, `loc`, associated with each frame.

Examples

Download CREPE Network

Download and unzip the Audio Toolbox™ model for CREPE.

Type `crepe` at the Command Window. If the Audio Toolbox model for CREPE is not installed, then the function provides a link to the location of the network weights. To download the model, click the link and unzip the file to a location on the MATLAB path.

Alternatively, execute these commands to download and unzip the CREPE model to your temporary directory.

```
downloadFolder = fullfile(tempdir,'crepeDownload');
loc = websave(downloadFolder,'https://ssd.mathworks.com/supportfiles/audio/crepe.zip');
crepeLocation = tempdir;
unzip(loc,crepeLocation)
addpath(fullfile(crepeLocation,'crepe'))
```

Check that the installation is successful by typing `crepe` at the Command Window. If the network is installed, then the function returns a `DAGNetwork` (Deep Learning Toolbox) object.

```
crepe
```

```
ans =
  DAGNetwork with properties:
```

```
         Layers: [34×1 nnet.cnn.layer.Layer]
    Connections: [33×2 table]
     InputNames: {'input'}
    OutputNames: {'pitch'}
```

Load Pretrained CREPE Network

Load a pretrained CREPE convolutional neural network and examine the layers and classes.

Use `crepe` to load the pretrained CREPE network. The output `net` is a `DAGNetwork` (Deep Learning Toolbox) object.

```
net = crepe
```

```
net =
  DAGNetwork with properties:

         Layers: [34×1 nnet.cnn.layer.Layer]
    Connections: [33×2 table]
     InputNames: {'input'}
    OutputNames: {'pitch'}
```

View the network architecture using the `Layers` property. The network has 34 layers. There are 13 layers with learnable weights, of which six are convolutional layers, six are batch normalization layers, and one is a fully connected layer.

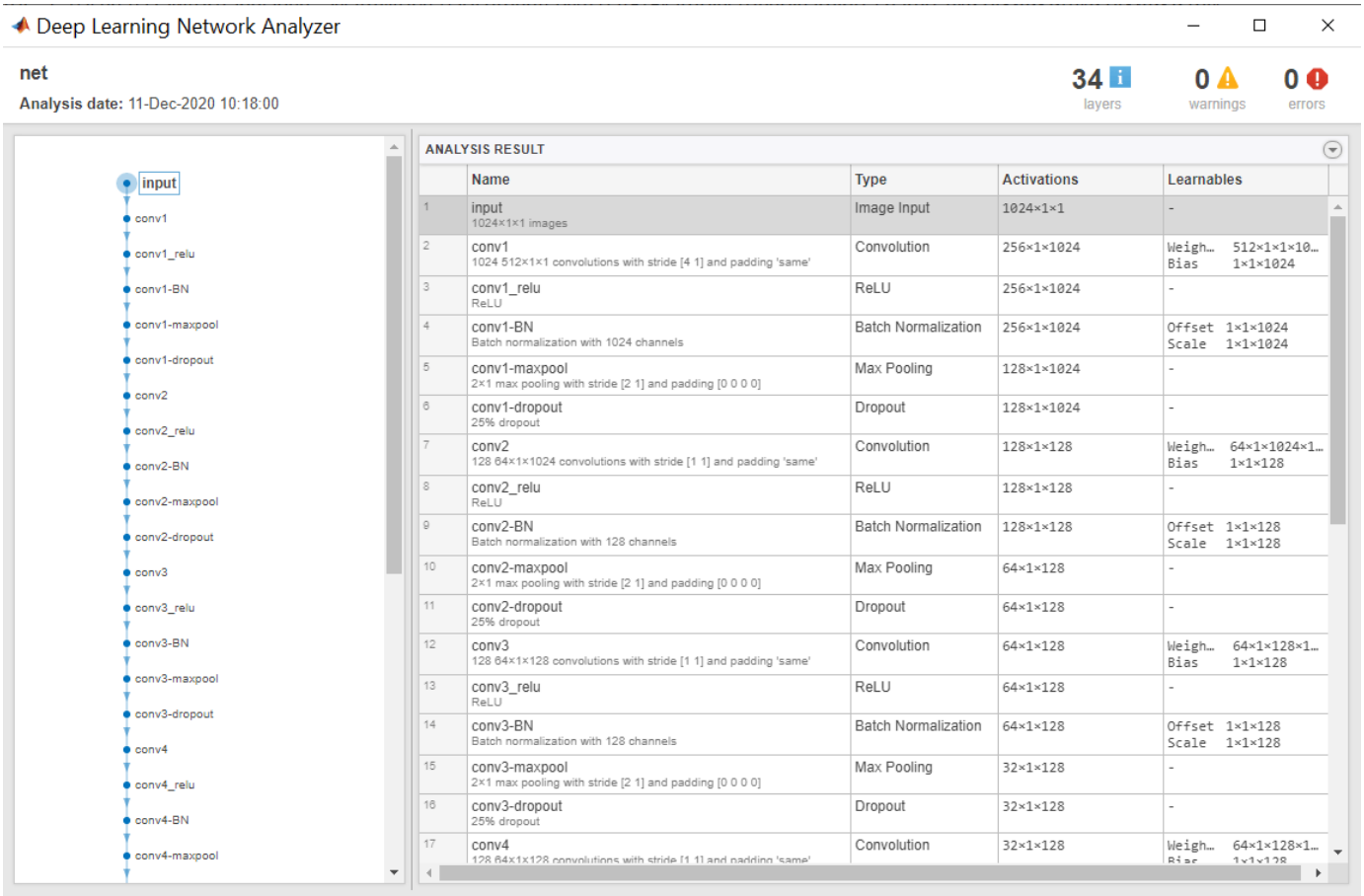
```
net.Layers
```

```
ans =
  34×1 Layer array with layers:

     1   'input'               Image Input           1024×1×1 images
     2   'conv1'               Convolution           1024 512×1×1 convolutions with stride [4  1
     3   'conv1_relu'          ReLU                  ReLU
     4   'conv1-BN'            Batch Normalization   Batch normalization with 1024 channels
     5   'conv1-maxpool'       Max Pooling           2×1 max pooling with stride [2  1] and pado
     6   'conv1-dropout'       Dropout               25% dropout
     7   'conv2'               Convolution           128 64×1×1024 convolutions with stride [1
     8   'conv2_relu'          ReLU                  ReLU
     9   'conv2-BN'            Batch Normalization   Batch normalization with 128 channels
    10   'conv2-maxpool'       Max Pooling           2×1 max pooling with stride [2  1] and pado
    11   'conv2-dropout'       Dropout               25% dropout
    12   'conv3'               Convolution           128 64×1×128 convolutions with stride [1  1
    13   'conv3_relu'          ReLU                  ReLU
    14   'conv3-BN'            Batch Normalization   Batch normalization with 128 channels
    15   'conv3-maxpool'       Max Pooling           2×1 max pooling with stride [2  1] and pado
    16   'conv3-dropout'       Dropout               25% dropout
    17   'conv4'               Convolution           128 64×1×128 convolutions with stride [1  1
    18   'conv4_relu'          ReLU                  ReLU
    19   'conv4-BN'            Batch Normalization   Batch normalization with 128 channels
    20   'conv4-maxpool'       Max Pooling           2×1 max pooling with stride [2  1] and pado
    21   'conv4-dropout'       Dropout               25% dropout
```

```
22   'conv5'                Convolution             256 64×1×128 convolutions with stride [1  1
23   'conv5_relu'           ReLU                    ReLU
24   'conv5-BN'             Batch Normalization     Batch normalization with 256 channels
25   'conv5-maxpool'        Max Pooling             2×1 max pooling with stride [2  1] and pado
26   'conv5-dropout'        Dropout                 25% dropout
27   'conv6'                Convolution             512 64×1×256 convolutions with stride [1  1
28   'conv6_relu'           ReLU                    ReLU
29   'conv6-BN'             Batch Normalization     Batch normalization with 512 channels
30   'conv6-maxpool'        Max Pooling             2×1 max pooling with stride [2  1] and pado
31   'conv6-dropout'        Dropout                 25% dropout
32   'classifier'           Fully Connected         360 fully connected layer
33   'classifier_sigmoid'   Sigmoid                 sigmoid
34   'pitch'                Regression Output       mean-squared-error
```

Use `analyzeNetwork` (Deep Learning Toolbox) to visually explore the network.

```
analyzeNetwork(net)
```



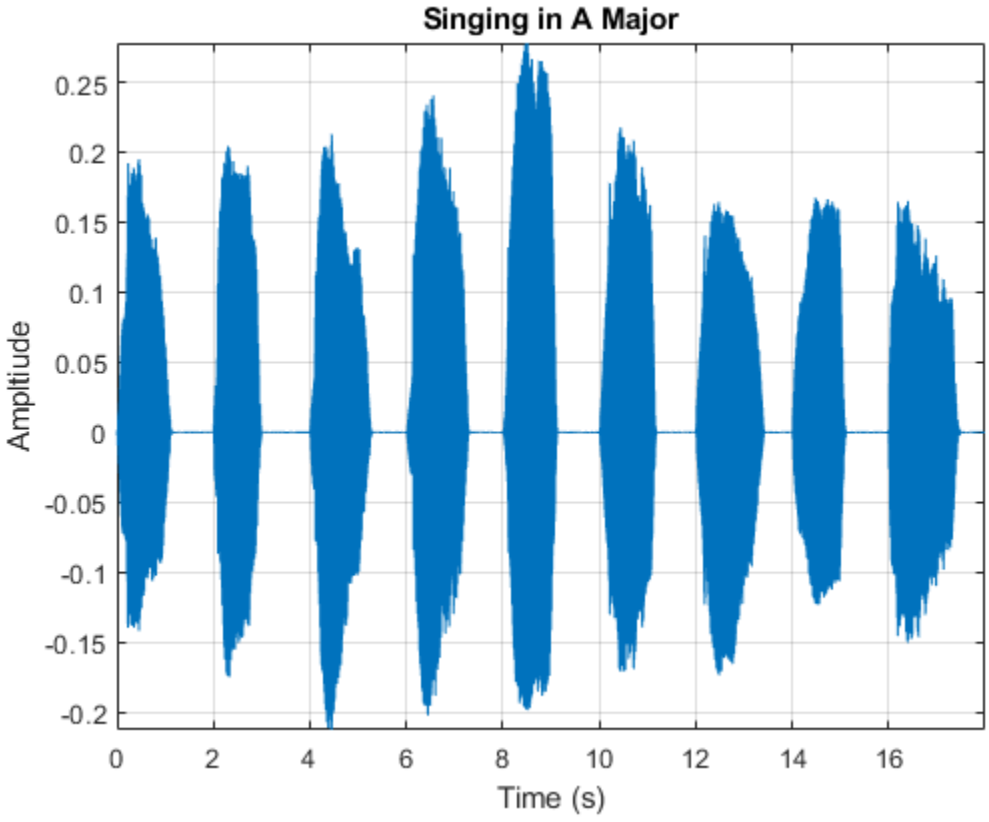
Estimate Pitch Using CREPE Network

The CREPE network requires you to preprocess your audio signals to generate buffered, overlapped, and normalized audio frames that can be used as input to the network. This example walks through

audio preprocessing using `crepePreprocess` and audio postprocessing with pitch estimation using `crepePostprocess`. The `pitchnn` function performs these steps for you.

Read in an audio signal for pitch estimation. Visualize and listen to the audio. There are nine vocal utterances in the audio clip.

```
[audioIn,fs] = audioread('SingingAMajor-16-mono-18secs.ogg');
soundsc(audioIn,fs)
T = 1/fs;
t = 0:T:(length(audioIn)*T) - T;
plot(t,audioIn);
grid on
axis tight
xlabel('Time (s)')
ylabel('Ampltiude')
title('Singing in A Major')
```



Use `crepePreprocess` to partition the audio into frames of 1024 samples with an 85% overlap between consecutive mel spectrograms. Place the frames along the fourth dimension.

```
[frames,loc] = crepePreprocess(audioIn,fs);
```

Create a CREPE network with `ModelCapacity` set to `tiny`. If you call `crepe` before downloading the model, an error is printed to the Command Window with a download link.

```
netTiny = crepe('ModelCapacity','tiny');
```

Predict the network activations.

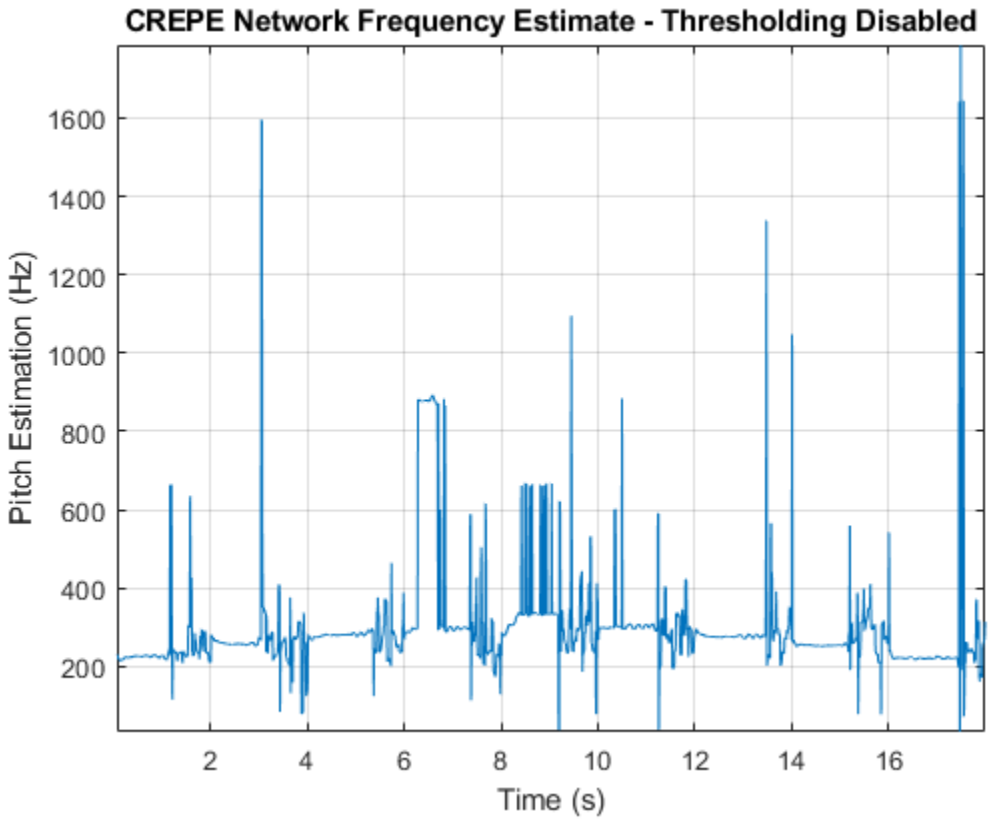
```
activationsTiny = predict(netTiny,frames);
```

Use `crepePostprocess` to produce the fundamental frequency pitch estimation in Hz. Disable confidence thresholding by setting `ConfidenceThreshold` to `0`.

```
f0Tiny = crepePostprocess(activationsTiny,'ConfidenceThreshold',0);
```

Visualize the pitch estimation over time.

```
plot(loc,f0Tiny)
grid on
axis tight
xlabel('Time (s)')
ylabel('Pitch Estimation (Hz)')
title('CREPE Network Frequency Estimate - Thresholding Disabled')
```



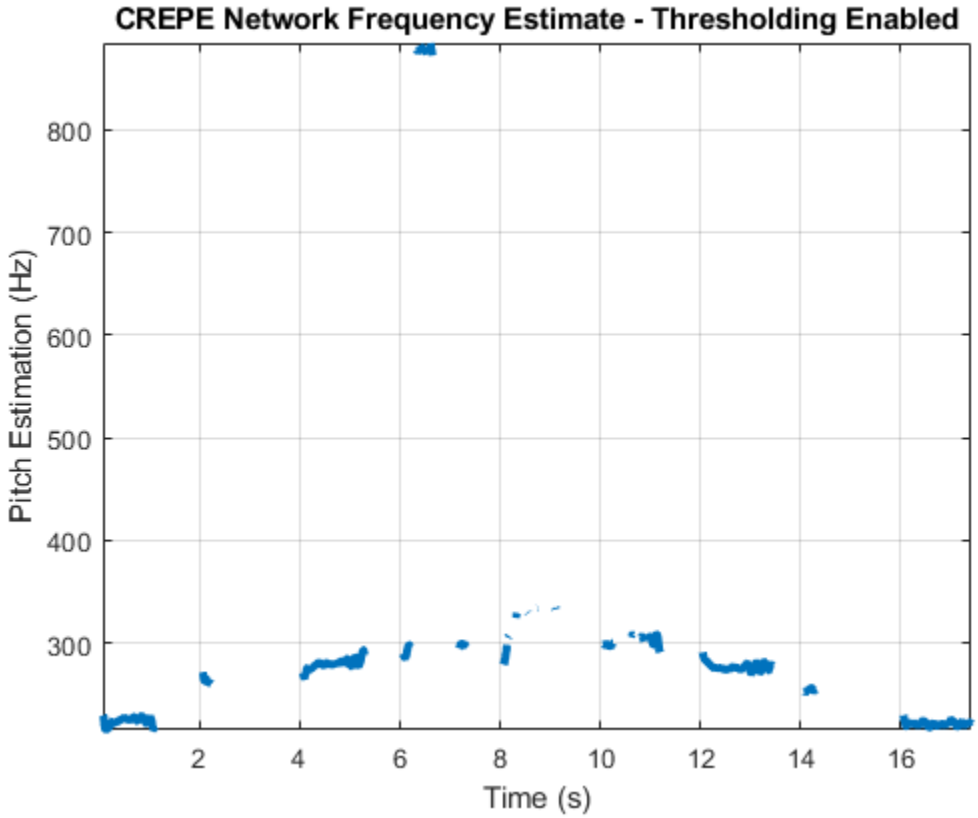
With confidence thresholding disabled, `crepePostprocess` provides a pitch estimate for every frame. Increase the `ConfidenceThreshold` to `0.8`.

```
f0Tiny = crepePostprocess(activationsTiny,'ConfidenceThreshold',0.8);
```

Visualize the pitch estimation over time.

```
plot(loc,f0Tiny,'LineWidth',3)
grid on
axis tight
xlabel('Time (s)')
```

```
ylabel('Pitch Estimation (Hz)')
title('CREPE Network Frequency Estimate - Thresholding Enabled')
```



Create a new CREPE network with `ModelCapacity` set to `full`.

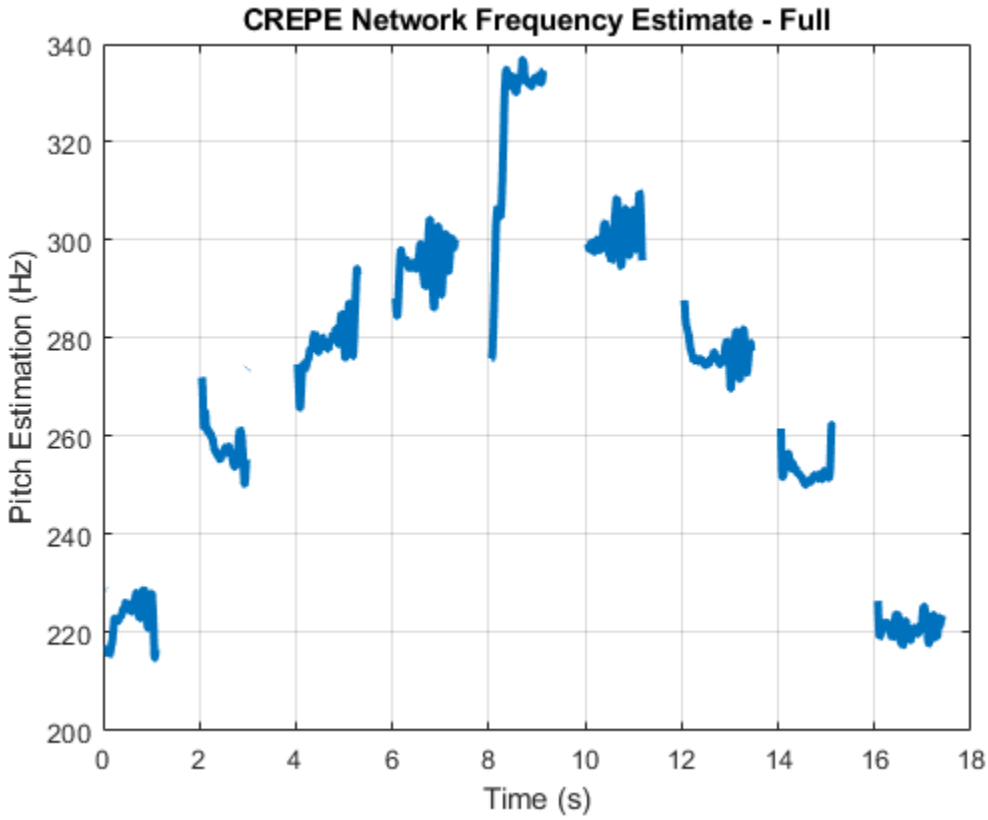
```
netFull = crepe('ModelCapacity','full');
```

Predict the network activations.

```
activationsFull = predict(netFull,frames);
f0Full = crepePostprocess(activationsFull,'ConfidenceThreshold',0.8);
```

Visualize the pitch estimation. There are nine primary pitch estimation groupings, each group corresponding with one of the nine vocal utterances.

```
plot(loc,f0Full,'LineWidth',3)
grid on
xlabel('Time (s)')
ylabel('Pitch Estimation (Hz)')
title('CREPE Network Frequency Estimate - Full')
```



Find the time elements corresponding to the last vocal utterance.

```
roundedLocVec = round(loc,2);
lastUtteranceBegin = find(roundedLocVec == 16);
lastUtteranceEnd = find(roundedLocVec == 18);
```

For simplicity, take the most frequently occurring pitch estimate within the utterance group as the fundamental frequency estimate for that timespan. Generate a pure tone with a frequency matching the pitch estimate for the last vocal utterance.

```
lastUtteranceEstimation = mode(f0Full(lastUtteranceBegin:lastUtteranceEnd))
```

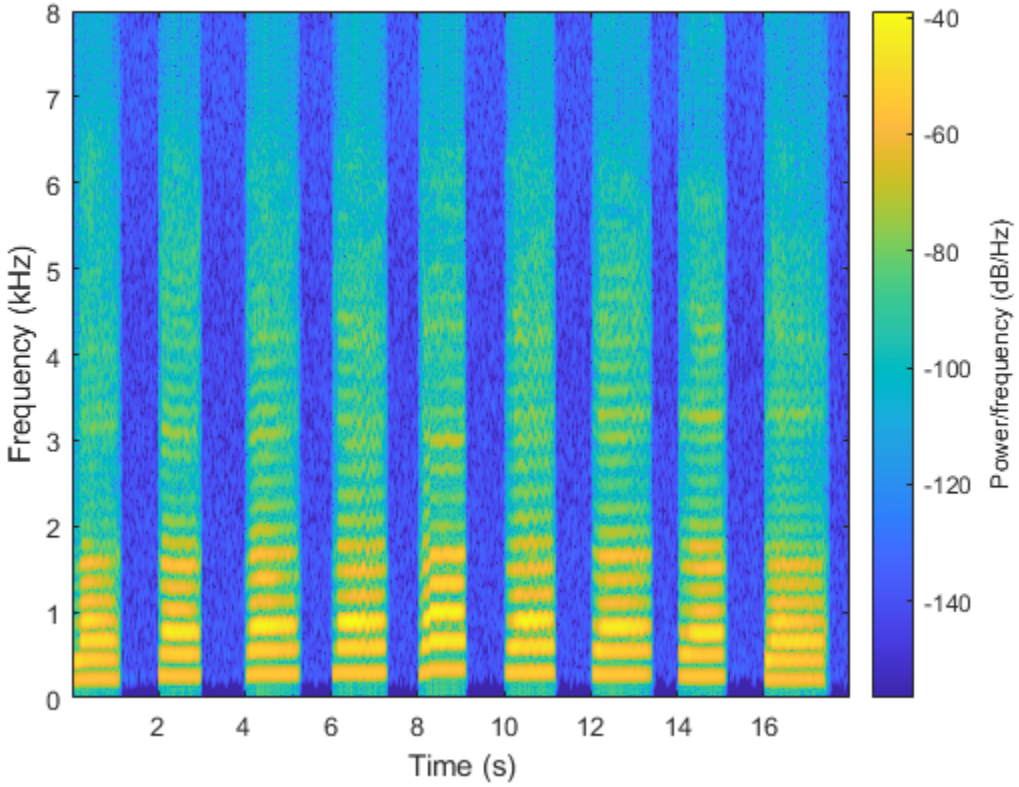
```
lastUtteranceEstimation = single
    217.2709
```

The value for `lastUtteranceEstimate` of `217.3` Hz. corresponds to the note A3. Overlay the synthesized tone on the last vocal utterance to audibly compare the two.

```
lastVocalUtterance = audioIn(fs*16:fs*18);
newTime = 0:T:2;
compareTone = cos(2*pi*lastUtteranceEstimation*newTime).';

soundsc(lastVocalUtterance + compareTone,fs);
```

Call `spectrogram` to more closely inspect the frequency content of the singing. Use a frame size of `250` samples and an overlap of `225` samples or 90%. Use `4096` DFT points for the transform. The `spectrogram` reveals that the vocal recording is actually a set of complex harmonic tones composed of multiple frequencies.

```
spectrogram(audioIn,250,225,4096,fs,'yaxis')
```



Input Arguments

`audioIn` — Input signal

column vector | matrix

Input signal, specified as a column vector or matrix. If you specify a matrix, `crepePreprocess` treats the columns of the matrix as individual audio channels.

Data Types: `single` | `double`

`fs` — Sample rate (Hz)

positive scalar

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

Data Types: `single` | `double`

`OP` — Overlap percentage between consecutive audio frames

85 (default) | nonnegative scalar in the range [0,100)

Percentage overlap between consecutive audio frames, specified as the comma-separated pair consisting of `'OverlapPercentage'` and a scalar in the range [0,100).

Data Types: `single` | `double`

Output Arguments

`frames` — Audio frames that can be fed to CREPE pretrained network

`1024`-by-`1`-by-`1`-by-*N* array

Processed audio frames, returned as a `1024`-by-`1`-by-`1`-by-*N* array, where *N* is the number of generated frames.

Note For multichannel inputs, generated `frames` are stacked along the 4th dimension according to channel. For example, if `audioIn` is a stereo signal, the number of generated `frames` for each channel is actually *N*`/2`. The first *N*`/2` `frames` correspond to channel 1 and the subsequent *N*`/2` `frames` correspond to channel 2.

Data Types: `single` | `double`

`loc` — Time values

1-by-*N* vector

Time values associated with each frame, returned as a `1`-by-*N* vector, where *N* is the number of generated frames. The time values correspond to the most recent samples used to compute the frames.

Data Types: `single` | `double`

Version History

Introduced in R2021a

References

[1] Kim, Jong Wook, Justin Salamon, Peter Li, and Juan Pablo Bello. “Crepe: A Convolutional Representation for Pitch Estimation.” In *2018 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 161–65. Calgary, AB: IEEE, 2018. https://doi.org/10.1109/ICASSP.2018.8461329.

Extended Capabilities

C/C++ Code Generation

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

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

See Also

`crepe` | `pitchnn` | `crepePostprocess`

crepe

CREPE neural network

Syntax

```
net = crepe
net = crepe('ModelCapacity',CAP)
```

Description

`net = crepe` returns a pretrained CREPE model.

This function requires both Audio Toolbox and Deep Learning Toolbox.

`net = crepe('ModelCapacity',CAP)` specifies the model capacity.

For example, `net = crepe('ModelCapacity','small')` specifies the model capacity as small.

Examples

Download CREPE Network

Download and unzip the Audio Toolbox™ model for CREPE.

Type `crepe` at the Command Window. If the Audio Toolbox model for CREPE is not installed, then the function provides a link to the location of the network weights. To download the model, click the link and unzip the file to a location on the MATLAB path.

Alternatively, execute these commands to download and unzip the CREPE model to your temporary directory.

```
downloadFolder = fullfile(tempdir,'crepeDownload');
loc = websave(downloadFolder,'https://ssd.mathworks.com/supportfiles/audio/crepe.zip');
crepeLocation = tempdir;
unzip(loc,crepeLocation)
addpath(fullfile(crepeLocation,'crepe'))
```

Check that the installation is successful by typing `crepe` at the Command Window. If the network is installed, then the function returns a `DAGNetwork` (Deep Learning Toolbox) object.

```
crepe
```

```
ans =
  DAGNetwork with properties:

         Layers: [34×1 nnet.cnn.layer.Layer]
    Connections: [33×2 table]
     InputNames: {'input'}
    OutputNames: {'pitch'}
```

Load Pretrained CREPE Network

Load a pretrained CREPE convolutional neural network and examine the layers and classes.

Use `crepe` to load the pretrained CREPE network. The output `net` is a `DAGNetwork` (Deep Learning Toolbox) object.

```
net = crepe
```

```
net =
  DAGNetwork with properties:

         Layers: [34×1 nnet.cnn.layer.Layer]
    Connections: [33×2 table]
     InputNames: {'input'}
    OutputNames: {'pitch'}
```

View the network architecture using the `Layers` property. The network has 34 layers. There are 13 layers with learnable weights, of which six are convolutional layers, six are batch normalization layers, and one is a fully connected layer.

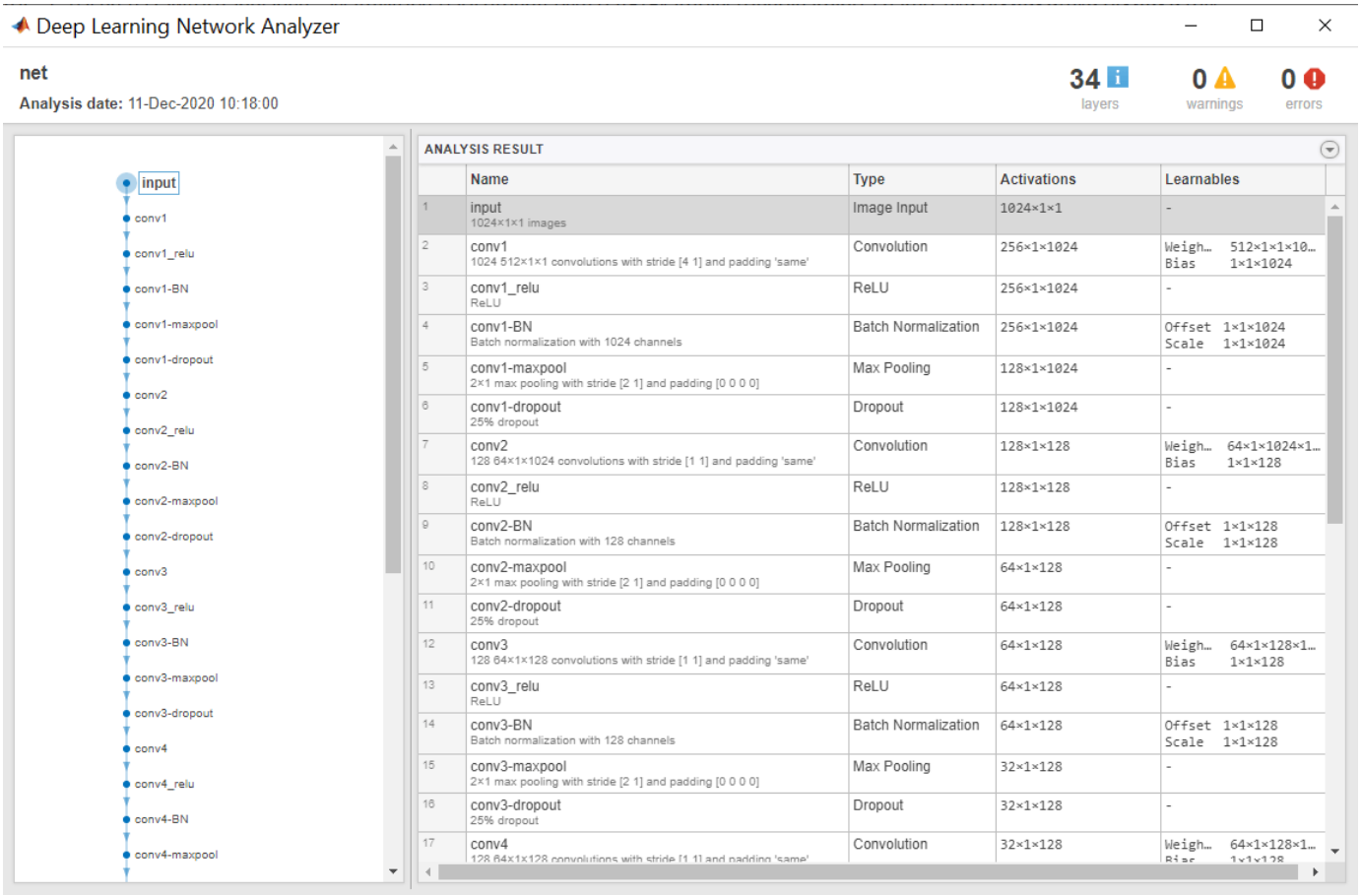
```
net.Layers
```

```
ans =
  34×1 Layer array with layers:

     1   'input'               Image Input           1024×1×1 images
     2   'conv1'               Convolution           1024 512×1×1 convolutions with stride [4  1
     3   'conv1_relu'          ReLU                  ReLU
     4   'conv1-BN'            Batch Normalization   Batch normalization with 1024 channels
     5   'conv1-maxpool'       Max Pooling           2×1 max pooling with stride [2  1] and pado
     6   'conv1-dropout'       Dropout               25% dropout
     7   'conv2'               Convolution           128 64×1×1024 convolutions with stride [1
     8   'conv2_relu'          ReLU                  ReLU
     9   'conv2-BN'            Batch Normalization   Batch normalization with 128 channels
    10   'conv2-maxpool'       Max Pooling           2×1 max pooling with stride [2  1] and pado
    11   'conv2-dropout'       Dropout               25% dropout
    12   'conv3'               Convolution           128 64×1×128 convolutions with stride [1  1
    13   'conv3_relu'          ReLU                  ReLU
    14   'conv3-BN'            Batch Normalization   Batch normalization with 128 channels
    15   'conv3-maxpool'       Max Pooling           2×1 max pooling with stride [2  1] and pado
    16   'conv3-dropout'       Dropout               25% dropout
    17   'conv4'               Convolution           128 64×1×128 convolutions with stride [1  1
    18   'conv4_relu'          ReLU                  ReLU
    19   'conv4-BN'            Batch Normalization   Batch normalization with 128 channels
    20   'conv4-maxpool'       Max Pooling           2×1 max pooling with stride [2  1] and pado
    21   'conv4-dropout'       Dropout               25% dropout
    22   'conv5'               Convolution           256 64×1×128 convolutions with stride [1  1
    23   'conv5_relu'          ReLU                  ReLU
    24   'conv5-BN'            Batch Normalization   Batch normalization with 256 channels
    25   'conv5-maxpool'       Max Pooling           2×1 max pooling with stride [2  1] and pado
    26   'conv5-dropout'       Dropout               25% dropout
    27   'conv6'               Convolution           512 64×1×256 convolutions with stride [1  1
    28   'conv6_relu'          ReLU                  ReLU
    29   'conv6-BN'            Batch Normalization   Batch normalization with 512 channels
```

```
30   'conv6-maxpool'        Max Pooling           2×1 max pooling with stride [2  1] and pado
31   'conv6-dropout'        Dropout               25% dropout
32   'classifier'           Fully Connected       360 fully connected layer
33   'classifier_sigmoid'   Sigmoid               sigmoid
34   'pitch'                Regression Output     mean-squared-error
```

Use `analyzeNetwork` (Deep Learning Toolbox) to visually explore the network.

```
analyzeNetwork(net)
```



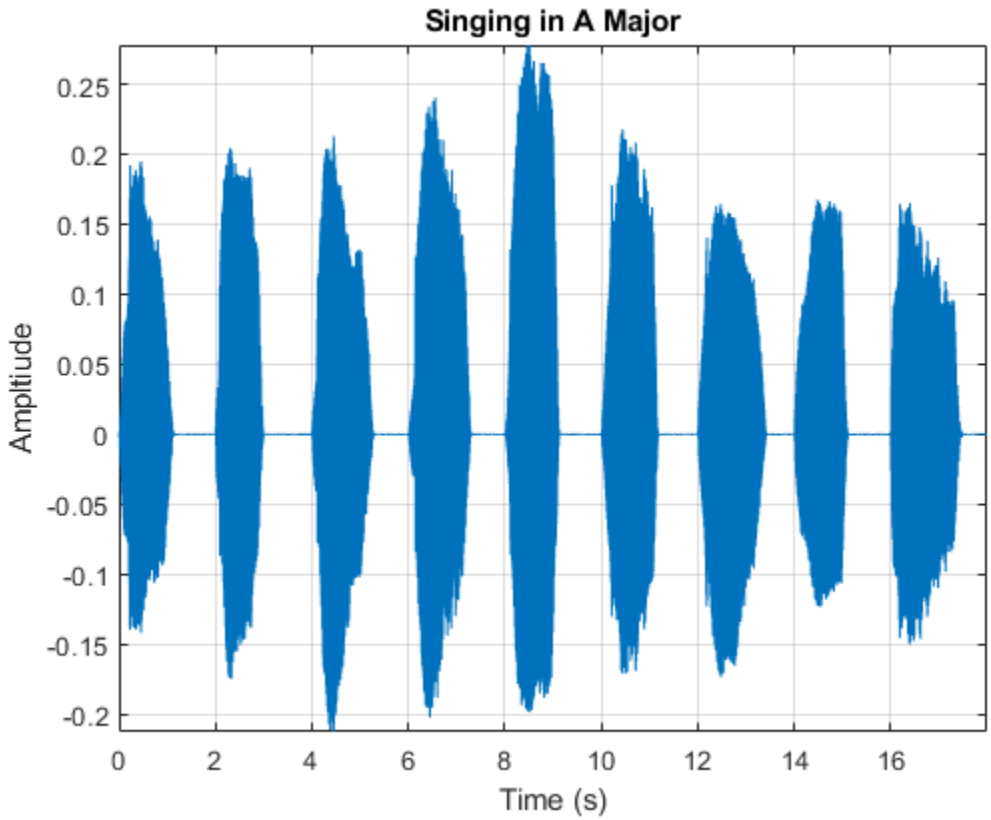
Estimate Pitch Using CREPE Network

The CREPE network requires you to preprocess your audio signals to generate buffered, overlapped, and normalized audio frames that can be used as input to the network. This example walks through audio preprocessing using `crepePreprocess` and audio postprocessing with pitch estimation using `crepePostprocess`. The `pitchnn` function performs these steps for you.

Read in an audio signal for pitch estimation. Visualize and listen to the audio. There are nine vocal utterances in the audio clip.

```
[audioIn,fs] = audioread('SingingAMajor-16-mono-18secs.ogg');
soundsc(audioIn,fs)
```

```
T = 1/fs;
t = 0:T:(length(audioIn)*T) - T;
plot(t,audioIn);
grid on
axis tight
xlabel('Time (s)')
ylabel('Ampltiude')
title('Singing in A Major')
```



Use `crepePreprocess` to partition the audio into frames of 1024 samples with an 85% overlap between consecutive mel spectrograms. Place the frames along the fourth dimension.

```
[frames,loc] = crepePreprocess(audioIn,fs);
```

Create a CREPE network with `ModelCapacity` set to `tiny`. If you call `crepe` before downloading the model, an error is printed to the Command Window with a download link.

```
netTiny = crepe('ModelCapacity','tiny');
```

Predict the network activations.

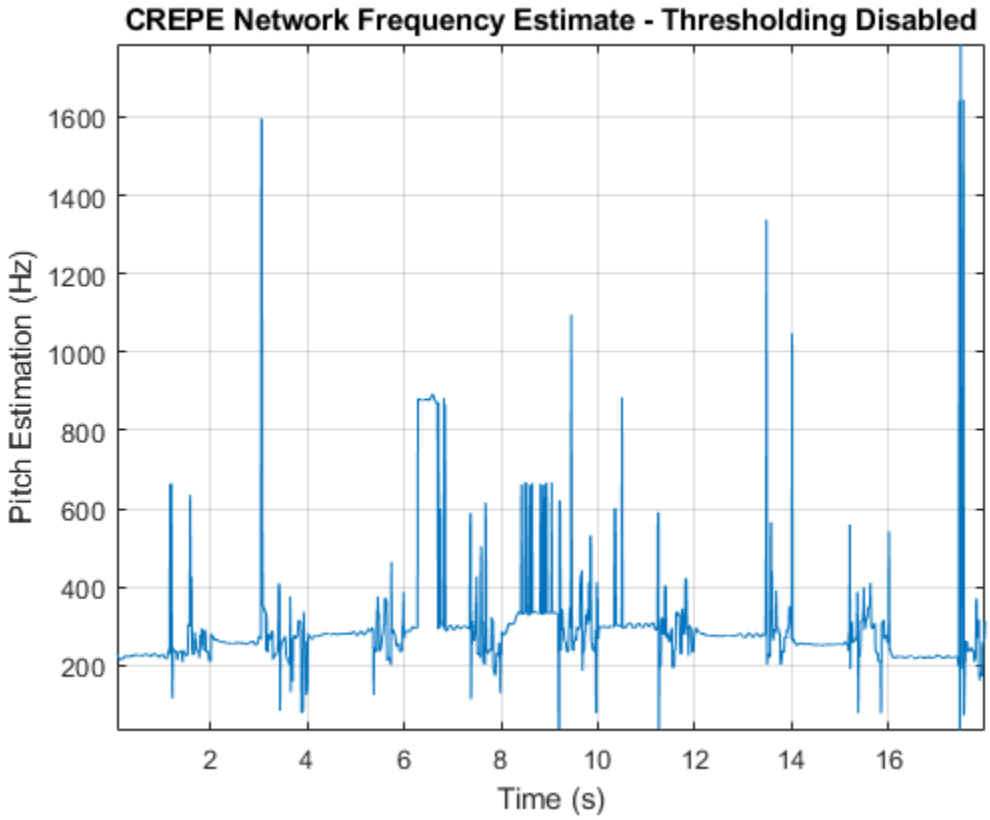
```
activationsTiny = predict(netTiny,frames);
```

Use `crepePostprocess` to produce the fundamental frequency pitch estimation in Hz. Disable confidence thresholding by setting `ConfidenceThreshold` to `0`.

```
f0Tiny = crepePostprocess(activationsTiny,'ConfidenceThreshold',0);
```

Visualize the pitch estimation over time.

```
plot(loc,f0Tiny)
grid on
axis tight
xlabel('Time (s)')
ylabel('Pitch Estimation (Hz)')
title('CREPE Network Frequency Estimate - Thresholding Disabled')
```

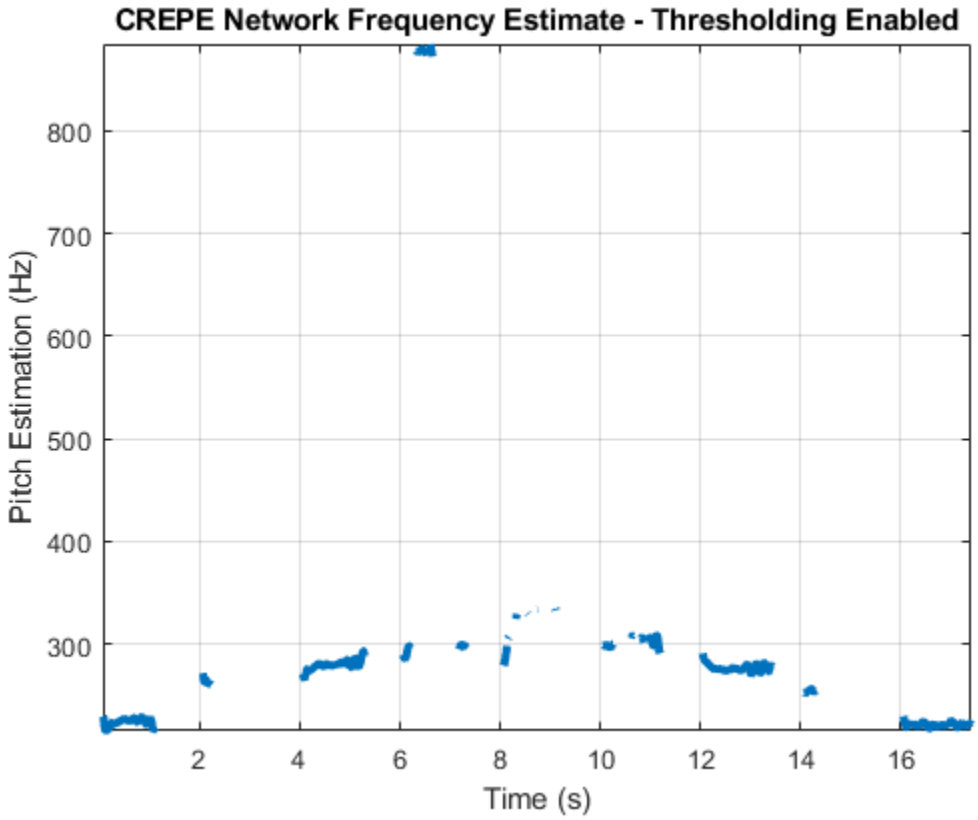


With confidence thresholding disabled, `crepePostprocess` provides a pitch estimate for every frame. Increase the `ConfidenceThreshold` to `0.8`.

```
f0Tiny = crepePostprocess(activationsTiny,'ConfidenceThreshold',0.8);
```

Visualize the pitch estimation over time.

```
plot(loc,f0Tiny,'LineWidth',3)
grid on
axis tight
xlabel('Time (s)')
ylabel('Pitch Estimation (Hz)')
title('CREPE Network Frequency Estimate - Thresholding Enabled')
```



Create a new CREPE network with `ModelCapacity` set to `full`.

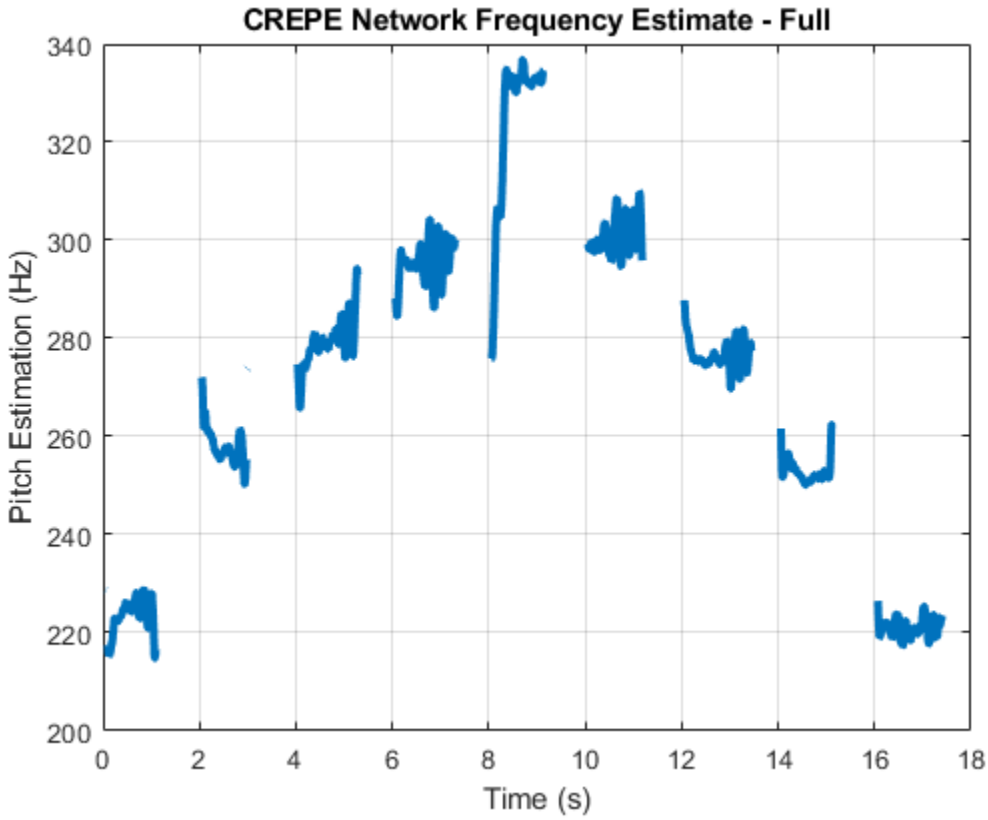
```
netFull = crepe('ModelCapacity','full');
```

Predict the network activations.

```
activationsFull = predict(netFull,frames);
f0Full = crepePostprocess(activationsFull,'ConfidenceThreshold',0.8);
```

Visualize the pitch estimation. There are nine primary pitch estimation groupings, each group corresponding with one of the nine vocal utterances.

```
plot(loc,f0Full,'LineWidth',3)
grid on
xlabel('Time (s)')
ylabel('Pitch Estimation (Hz)')
title('CREPE Network Frequency Estimate - Full')
```



Find the time elements corresponding to the last vocal utterance.

```
roundedLocVec = round(loc,2);
lastUtteranceBegin = find(roundedLocVec == 16);
lastUtteranceEnd = find(roundedLocVec == 18);
```

For simplicity, take the most frequently occurring pitch estimate within the utterance group as the fundamental frequency estimate for that timespan. Generate a pure tone with a frequency matching the pitch estimate for the last vocal utterance.

```
lastUtteranceEstimation = mode(f0Full(lastUtteranceBegin:lastUtteranceEnd))
```

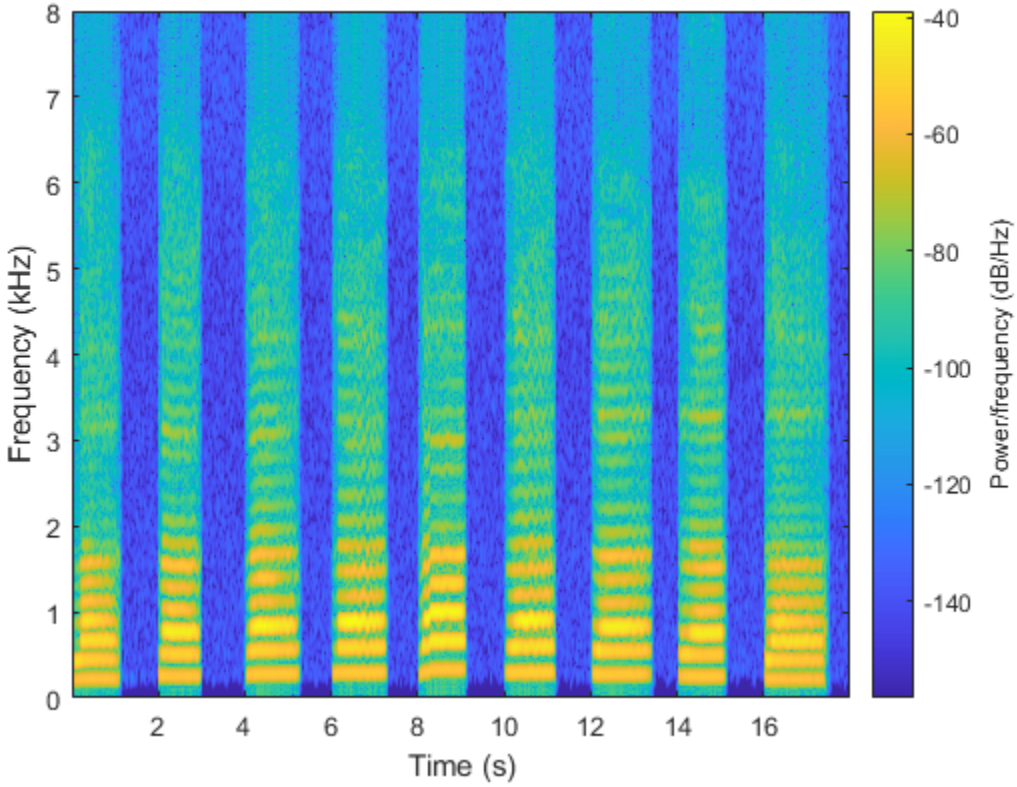
```
lastUtteranceEstimation = single
    217.2709
```

The value for `lastUtteranceEstimate` of `217.3` Hz. corresponds to the note A3. Overlay the synthesized tone on the last vocal utterance to audibly compare the two.

```
lastVocalUtterance = audioIn(fs*16:fs*18);
newTime = 0:T:2;
compareTone = cos(2*pi*lastUtteranceEstimation*newTime).';

soundsc(lastVocalUtterance + compareTone,fs);
```

Call `spectrogram` to more closely inspect the frequency content of the singing. Use a frame size of `250` samples and an overlap of `225` samples or 90%. Use `4096` DFT points for the transform. The `spectrogram` reveals that the vocal recording is actually a set of complex harmonic tones composed of multiple frequencies.

```
spectrogram(audioIn,250,225,4096,fs,'yaxis')
```



Input Arguments

CAP — Model Capacity

`'full'` (default) | `'tiny'` | `'small'` | `'medium'` | `'large'`

Model capacity, specified as the comma-separated pair consisting of `'ModelCapacity'` and `'tiny'`, `'small'`, `'medium'`, `'large'`, or `'full'`.

Tip `'ModelCapacity'` controls the complexity of the underlying deep learning neural network. The higher the model capacity, the greater the number of nodes and layers in the model. Selecting the right model capacity for your data will help prevent under or overfitting.

Data Types: `string` | `char`

Output Arguments

net — Pretrained CREPE neural network

`DAGNetwork` object

Pretrained CREPE neural network, returned as a `DAGNetwork` object.

Version History

Introduced in R2021a

References

[1] Kim, Jong Wook, Justin Salamon, Peter Li, and Juan Pablo Bello. “Crepe: A Convolutional Representation for Pitch Estimation.” In *2018 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 161–65. Calgary, AB: IEEE, 2018. https://doi.org/10.1109/ICASSP.2018.8461329.

Extended Capabilities

C/C++ Code Generation

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

Usage notes and limitations:

- Only the `activations` and `predict` object functions are supported.
- To create a `SeriesNetwork` object for code generation, see “Load Pretrained Networks for Code Generation” (MATLAB Coder).

GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

Usage notes and limitations:

- Only the `activations`, `classify`, `predict`, `predictAndUpdateState`, and `resetState` object functions are supported.
- To create a `SeriesNetwork` object for code generation, see “Load Pretrained Networks for Code Generation” (GPU Coder).

See Also

`vggish` | `crepePreprocess` | `pitchnn` | `crepePostprocess`

openl3Embeddings

Extract OpenL3 feature embeddings

Syntax

```
embeddings = openl3Embeddings(audioIn,fs)

embeddings = openl3Embeddings(audioIn,fs,Name=Value)
```

Description

`embeddings = openl3Embeddings(audioIn,fs)` returns OpenL3 feature embeddings over time for audio input `audioIn` with sample rate `fs`. Columns of the input are treated as individual channels.

`embeddings = openl3Embeddings(audioIn,fs,Name=Value)` specifies options using one or more name-value arguments. For example, `embeddings = openl3Embeddings(audioIn,fs,OverlapPercentage=75)` applies a 75% overlap between consecutive frames used to create the audio embeddings.

This function requires both Audio Toolbox and Deep Learning Toolbox.

Examples

Download openl3Embeddings Functionality

Download and unzip the Audio Toolbox™ model for OpenL3.

Type `openl3Embeddings` at the command line. If the Audio Toolbox model for OpenL3 is not installed, the function provides a link to the location of the network weights. To download the model, click the link. Unzip the file to a location on the MATLAB path.

Alternatively, execute the following commands to download and unzip the OpenL3 model to your temporary directory.

```
downloadFolder = fullfile(tempdir,"OpenL3Download");
loc = websave(downloadFolder,"https://ssd.mathworks.com/supportfiles/audio/openl3.zip");
OpenL3Location = tempdir;
unzip(loc,OpenL3Location)
addpath(fullfile(OpenL3Location,"openl3"))
```

Extract OpenL3 Embeddings

Read in an audio file.

```
[audioIn,fs] = audioread('MainStreetOne-16-16-mono-12secs.wav');
```

Call the `openl3Embeddings` function with the audio and sample rate to extract OpenL3 feature embeddings from the audio. Using the openl3Embeddings function requires installing the pretrained OpenL3 network. If the network is not installed, the function provides a link to download the pretrained model.

```
embeddings = openl3Embeddings(audioIn,fs);
```

The `openl3Embeddings` function returns a matrix of 512-element feature vectors over time.

```
[numHops,numElementsPerHop,numChannels] = size(embeddings)
```

```
numHops = 111
```

```
numElementsPerHop = 512
```

```
numChannels = 1
```

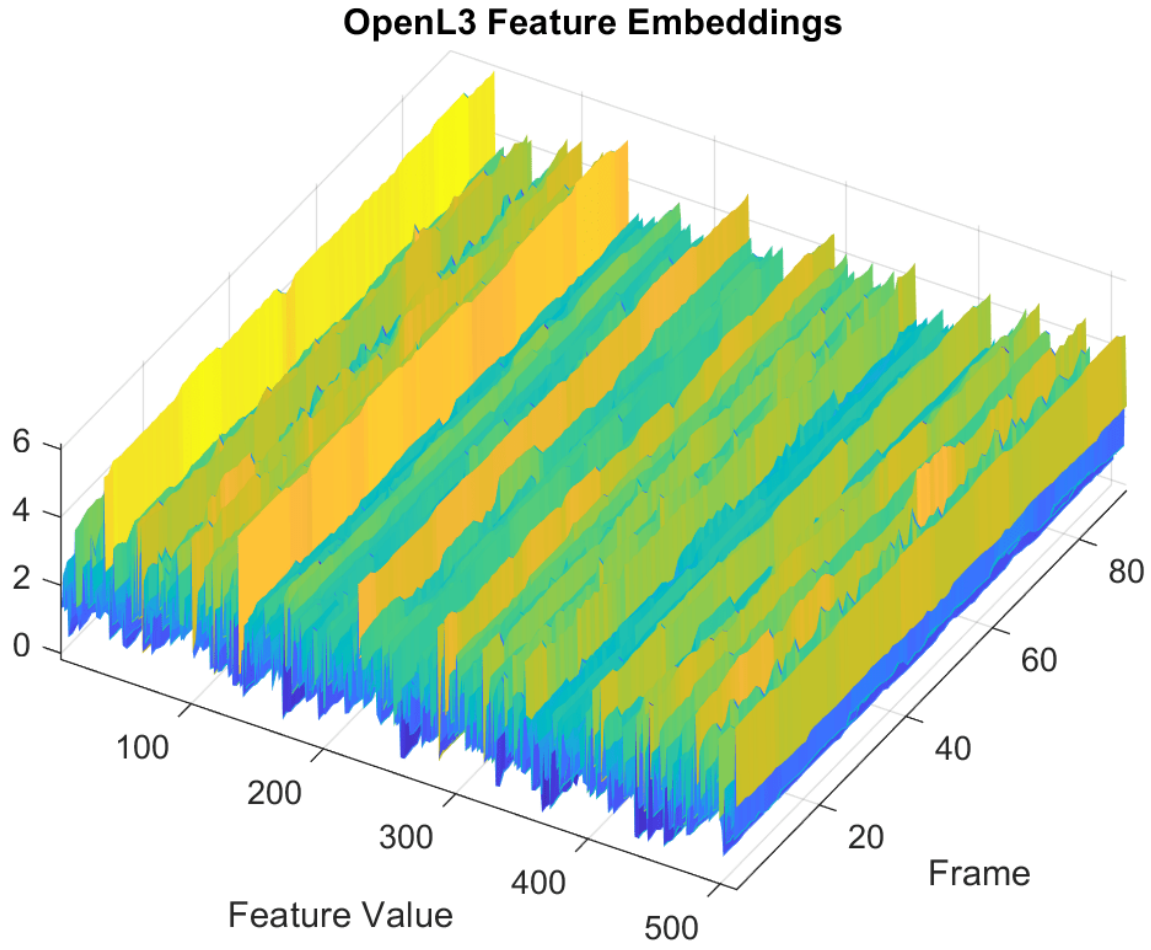
Decrease Time Resolution of OpenL3 Embeddings

Create a 10-second pink noise signal and then extract OpenL3 embeddings. The `openl3Embeddings` function extracts feature embeddings from mel spectrograms with 90% overlap. Using the `openl3Embeddings` function requires installing the pretrained OpenL3 network. If the network is not installed, the function provides a link to download the pretrained model.

```
fs = 16e3;
dur = 10;
audioIn = pinknoise(dur*fs,1,"single");
embeddings = openl3Embeddings(audioIn,fs);
```

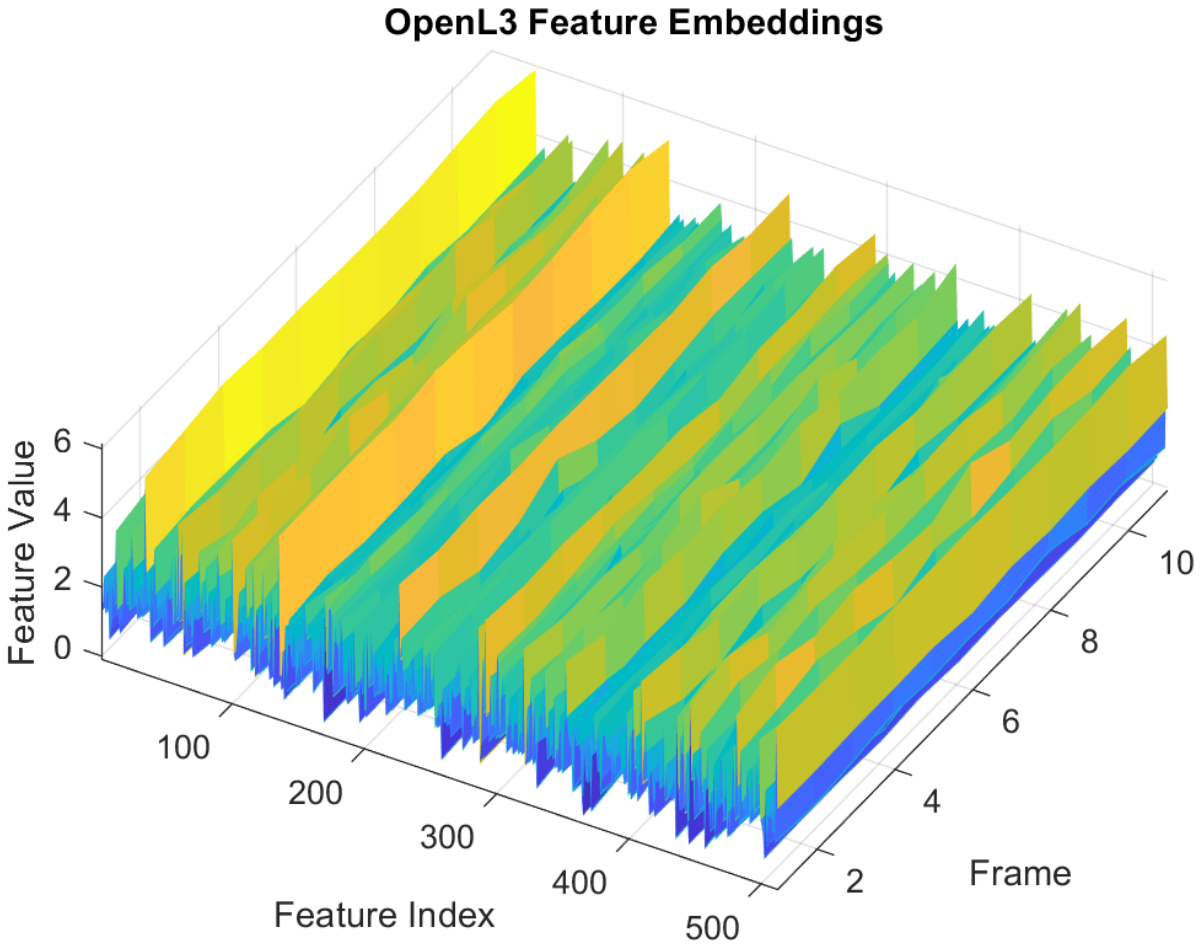
Plot the OpenL3 feature embeddings over time.

```
surf(embeddings,EdgeColor="none")
view([30 65])
axis tight
xlabel("Feature Index")
ylabel("Frame")
xlabel("Feature Value")
title("OpenL3 Feature Embeddings")
```



To decrease the resolution of OpenL3 feature embeddings over time, specify the percent overlap between mel spectrograms. Plot the results.

```
overlapPercentage = 10;
embeddings = openl3Embeddings(audioIn,fs,OverlapPercentage=overlapPercentage);
surf(embeddings,EdgeColor="none")
view([30 65])
axis tight
xlabel("Feature Index")
ylabel("Frame")
zlabel("Feature Value")
title("OpenL3 Feature Embeddings")
```



Input Arguments

`audioIn` — Input signal
column vector | matrix

Input signal, specified as a column vector or matrix. If you specify a matrix, `openl3Embeddings` treats the columns of the matrix as individual audio channels.

Data Types: `single` | `double`

`fs` — Sample rate (Hz)
positive scalar

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

Data Types: `single` | `double`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `openl3Embeddings(audioIn,fs,SpectrumType="mel256")`

`OverlapPercentage` — Percentage overlap between consecutive spectrograms
`90` (default) | scalar in the range [0,100)

Percentage overlap between consecutive spectrograms, specified as a scalar in the range [0,100).

Data Types: `single` | `double`

`SpectrumType` — Spectrum type
`"mel128"` (default) | `"mel256"` | `"linear"`

Spectrum type generated from audio and used as input to the neural network, specified as `"mel128"`, `"mel256"`, or `"linear"`.

Note The `SpectrumType` that you select controls the spectrogram used in the network. See `openl3` or `openl3Preprocess` for more details.

Data Types: `char` | `string`

`EmbeddingLength` — Embedding length
`512` (default) | `6144`

Length of the output audio embedding, specified as `512` or `6144`.

Data Types: `single` | `double`

`ContentType` — Audio content type
`"env"` (default) | `"music"`

Audio content type the neural network is trained on, specified as `"env"` or `"music"`.

Set `ContentType` to:

- `"env"` when you want to use a model trained on environmental data.
- `"music"` when you want to use a model trained on musical data.

Data Types: `char` | `string`

Output Arguments

`embeddings` — Compact representation of audio data
N-by-*L*-by-*C* array

Compact representation of audio data, returned as an *N*-by-*L*-by-*C* array, where:

- *N* –– Represents the number of buffered frames the audio signal is partitioned into and depends on the length of `audioIn` and the `OverlapPercentage`.
- *L* –– Represents the audio embedding length.
- *C* –– Represents the number of input channels.

Data Types: `single`

Version History

Introduced in R2022a

References

[1] Cramer, Jason, et al. "Look, Listen, and Learn More: Design Choices for Deep Audio Embeddings." In *ICASSP 2019 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, IEEE, 2019, pp. 3852-56. *DOI.org (Crossref)*, doi:/10.1109/ICASSP.2019.8682475.

Extended Capabilities

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

See Also

`openl3Preprocess` | `openl3` | `vggish` | `classifySound` | `vggishEmbeddings` | `audioFeatureExtractor`

openl3Features

(To be removed) Extract OpenL3 features

Note The `openl3Features` function will be removed in a future release. Use `openl3Embeddings` instead. For more information, see “Compatibility Considerations”.

Syntax

```
embeddings = openl3Features(audioIn,fs)

embeddings = openl3Features(audioIn,fs,Name,Value)
```

Description

`embeddings = openl3Features(audioIn,fs)` returns OpenL3 feature embeddings over time for audio input `audioIn` with sample rate `fs`. Columns of the input are treated as individual channels.

`embeddings = openl3Features(audioIn,fs,Name,Value)` specifies options using one or more `Name,Value` arguments. For example, `embeddings = openl3Features(audioIn,fs,'OverlapPercentage',75)` applies a 75% overlap between consecutive frames used to create the audio embeddings.

This function requires both Audio Toolbox and Deep Learning Toolbox.

Examples

Download `openl3Features` Functionality

Download and unzip the Audio Toolbox™ model for OpenL3.

Type `openl3Features` at the command line. If the Audio Toolbox model for OpenL3 is not installed, the function provides a link to the location of the network weights. To download the model, click the link. Unzip the file to a location on the MATLAB path.

Alternatively, execute the following commands to download and unzip the OpenL3 model to your temporary directory.

```
downloadFolder = fullfile(tempdir,'OpenL3Download');
loc = websave(downloadFolder,'https://ssd.mathworks.com/supportfiles/audio/openl3.zip');
OpenL3Location = tempdir;
unzip(loc,OpenL3Location)
addpath(fullfile(OpenL3Location,'openl3'))
```

Extract OpenL3 Embeddings

Read in an audio file.

```
[audioIn,fs] = audioread('MainStreetOne-16-16-mono-12secs.wav');
```

Call the `openl3Features` function with the audio and sample rate to extract OpenL3 feature embeddings from the audio.

```
featureVectors = openl3Features(audioIn,fs);
```

The `openl3Features` function returns a matrix of 512-element feature vectors over time.

```
[numHops,numElementsPerHop,numChannels] = size(featureVectors)
```

```
numHops = 111
```

```
numElementsPerHop = 512
```

```
numChannels = 1
```

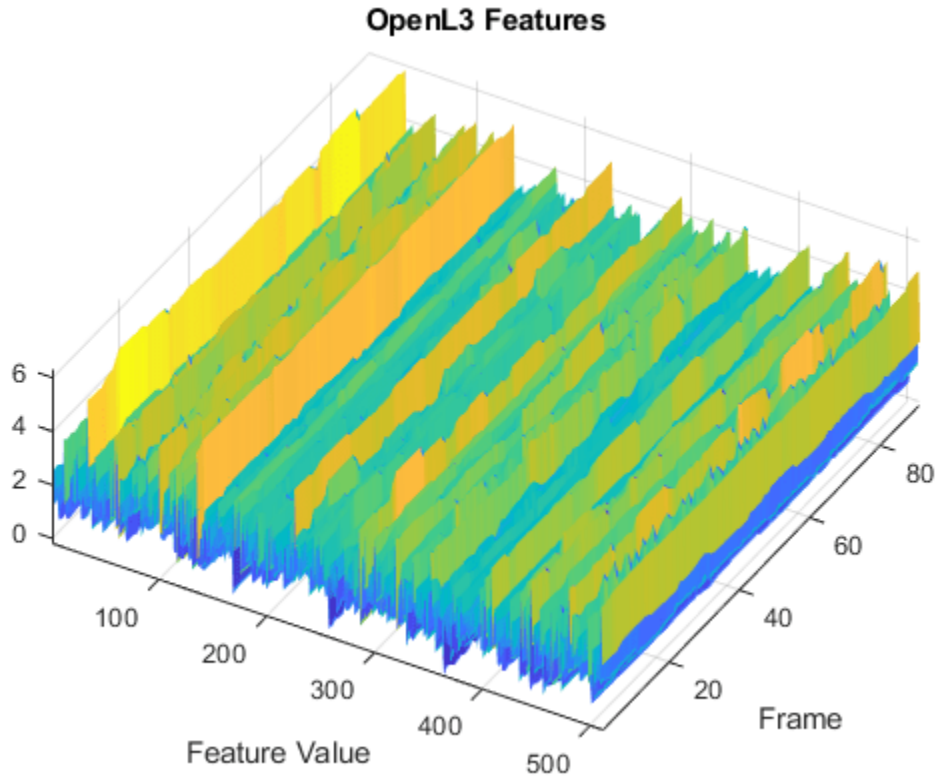
Decrease Time Resolution of OpenL3 Features

Create a 10-second pink noise signal and then extract OpenL3 features. The `openl3Features` function extracts features from mel spectrograms with 90% overlap.

```
fs = 16e3;
dur = 10;
audioIn = pinknoise(dur*fs,1,'single');
features = openl3Features(audioIn,fs);
```

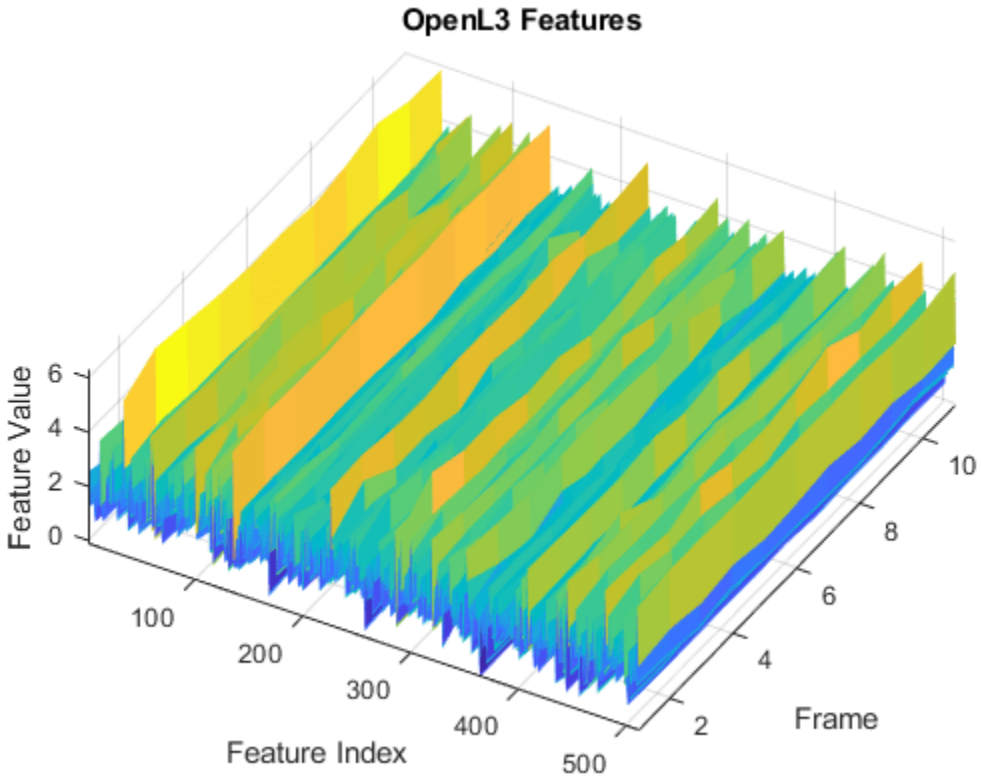
Plot the OpenL3 features over time.

```
surf(features,'EdgeColor','none')
view([30 65])
axis tight
xlabel('Feature Index')
ylabel('Frame')
xlabel('Feature Value')
title('OpenL3 Features')
```



To decrease the resolution of OpenL3 features over time, specify the percent overlap between mel spectrograms. Plot the results.

```
overlapPercentage = 10 ;
features = openl3Features(audioIn,fs,'OverlapPercentage',overlapPercentage);
surf(features,'EdgeColor','none')
view([30 65])
axis tight
xlabel('Feature Index')
ylabel('Frame')
zlabel('Feature Value')
title('OpenL3 Features')
```



Input Arguments

`audioIn` — Input signal

column vector | matrix

Input signal, specified as a column vector or matrix. If you specify a matrix, `openl3Features` treats the columns of the matrix as individual audio channels.

Data Types: `single` | `double`

`fs` — Sample rate (Hz)

positive scalar

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

Data Types: `single` | `double`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `openl3Features(audioIn,fs,'SpectrumType','mel256')`

`OverlapPercentage` — Percentage overlap between consecutive spectrograms
`90` (default) | scalar in the range [0,100)

Percentage overlap between consecutive spectrograms, specified as a scalar in the range [0,100).

Data Types: `single` | `double`

`SpectrumType` — Spectrum type
`'mel128'` (default) | `'mel256'` | `'linear'`

Spectrum type generated from audio and used as input to the neural network, specified as `'mel128'`, `'mel256'`, or `'linear'`.

Note The `SpectrumType` that you select controls the spectrogram used in the network. See `openl3` or `openl3Preprocess` for more details.

Data Types: `char` | `string`

`EmbeddingLength` — Embedding length
`512` (default) | `6144`

Length of the output audio embedding, specified as `'512'` or `'6144'`.

Data Types: `single` | `double`

`ContentType` — Audio content type
`'env'` (default) | `'music'`

Audio content type the neural network is trained on, specified as `'env'` or `'music'`.

Set `ContentType` to:

- `'env'` when you want to use a model trained on environmental data.
- `'music'` when you want to use a model trained on musical data.

Data Types: `char` | `string`

Output Arguments

`embeddings` — Compact representation of audio data
N-by-*L*-by-*C* array

Compact representation of audio data, returned as an *N*-by-*L*-by-*C* array, where:

- *N* –- Represents the number of buffered frames the audio signal is partitioned into and depends on the length of `audioIn` and the `'OverlapPercentage'`.
- *L* –- Represents the audio embedding length.
- *C* –- Represents the number of input channels.

Data Types: `single`

Version History

Introduced in R2021a

R2022a: `openl3Features` will be removed
Not recommended starting in R2022a

The `openl3Features` function will be removed in a future release. Use `openl3Embeddings` instead. Existing calls to `openl3Features` continue to run.

References

[1] Cramer, Jason, et al. "Look, Listen, and Learn More: Design Choices for Deep Audio Embeddings." In *ICASSP 2019 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, IEEE, 2019, pp. 3852-56. *DOI.org (Crossref)*, doi:/10.1109/ICASSP.2019.8682475.

Extended Capabilities

GPU Arrays
Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

See Also

`openl3Preprocess` | `openl3` | `vggish` | `classifySound` | `vggishEmbeddings` | `audioFeatureExtractor`

openl3Preprocess

Preprocess audio for OpenL3 feature extraction

Syntax

```
features = openl3Preprocess(audioIn,fs)

features = openl3Preprocess(audioIn,fs,Name,Value)
```

Description

`features = openl3Preprocess(audioIn,fs)` generates spectrograms from `audioIn` that can be fed to the OpenL3 pretrained network.

`features = openl3Preprocess(audioIn,fs,Name,Value)` specifies options using one or more `Name,Value` arguments. For example, `features = openl3Preprocess(audioIn,fs,'OverlapPercentage',75)` applies a 75% overlap between consecutive frames used to generate the spectrograms.

Examples

Download OpenL3 Network

Download and unzip the Audio Toolbox™ model for OpenL3.

Type `openl3` at the Command Window. If the Audio Toolbox model for OpenL3 is not installed, the function provides a link to the location of the network weights. To download the model, click the link. Unzip the file to a location on the MATLAB path.

Alternatively, execute these commands to download and unzip the OpenL3 model to your temporary directory.

```
downloadFolder = fullfile(tempdir,'OpenL3Download');
loc = websave(downloadFolder,'https://ssd.mathworks.com/supportfiles/audio/openl3.zip');
OpenL3Location = tempdir;
unzip(loc,OpenL3Location)
addpath(fullfile(OpenL3Location,'openl3'))
```

Check that the installation is successful by typing `openl3` at the Command Window. If the network is installed, then the function returns a `DAGNetwork` (Deep Learning Toolbox) object.

```
openl3
```

```
ans =
  DAGNetwork with properties:

         Layers: [30×1 nnet.cnn.layer.Layer]
    Connections: [29×2 table]
     InputNames: {'in'}
    OutputNames: {'out'}
```

Extract OpenL3 Embeddings from Audio Signal

Use `openl3Preprocess` to extract embeddings from an audio signal.

Read in an audio signal.

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

To extract spectrograms from the audio, call the `openl3Preprocess` function with the audio and sample rate. Use 50% overlap and set the spectrum type to linear. The `openl3Preprocess` function returns an array of 30 spectrograms produced using an FFT length of 512.

```
features = openl3Preprocess(audioIn,fs,'OverlapPercentage',50,'SpectrumType','linear');
[posFFTbinsOvLap50,numHopsOvLap50,~,numSpectOvLap50] = size(features)
```

```
posFFTbinsOvLap50 = 257

numHopsOvLap50 = 197

numSpectOvLap50 = 30
```

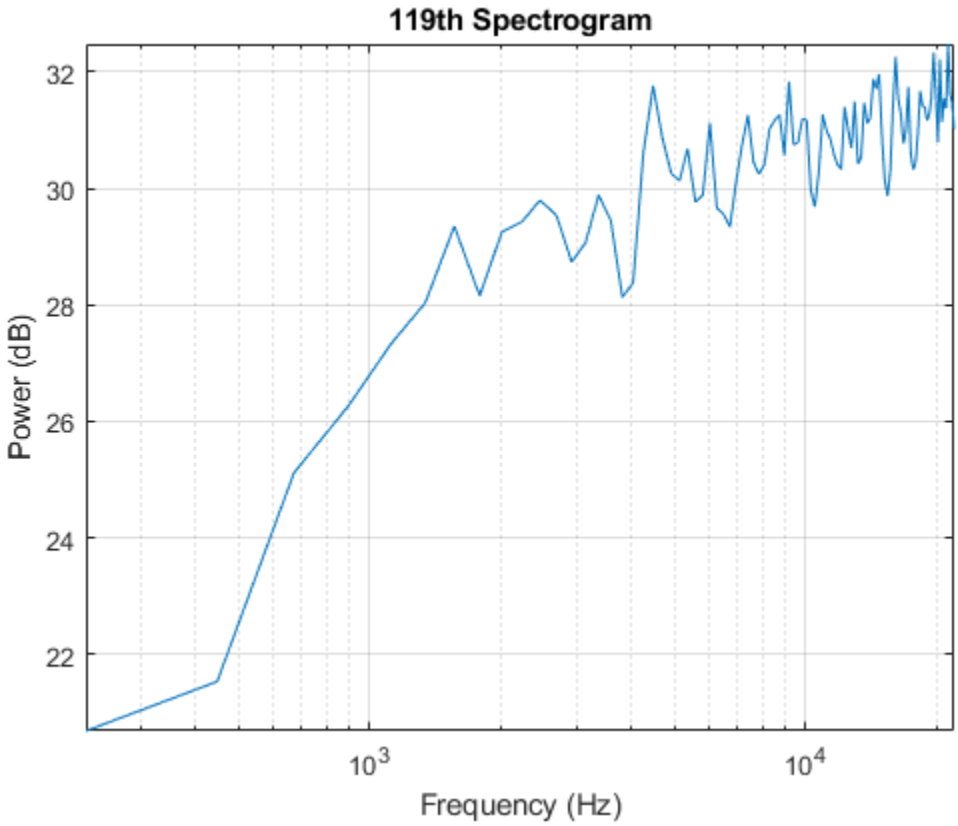
Call `openl3Preprocess` again, this time using the default overlap of 90%. The `openl3Preprocess` function now returns an array of 146 spectrograms.

```
features = openl3Preprocess(audioIn,fs,'SpectrumType','linear');
[posFFTbinsOvLap90,numHopsOvLap90,~,numSpectOvLap90] = size(features)
```

```
posFFTbinsOvLap90 = 257

numHopsOvLap90 = 197

numSpectOvLap90 = 146
```

Visualize one of the spectrograms at random.

```
randSpect = randi(numSpectOvLap90);
viewRandSpect = features(:,:,:,randSpect);
N = size(viewRandSpect,2);
binsToHz = (0:N-1)*fs/N;
nyquistBin = round(N/2);
semilogx(binsToHz(1:nyquistBin),mag2db(abs(viewRandSpect(1:nyquistBin))))
xlabel('Frequency (Hz)')
ylabel('Power (dB)');
title([num2str(randSpect),'th Spectrogram'])
axis tight
grid on
```

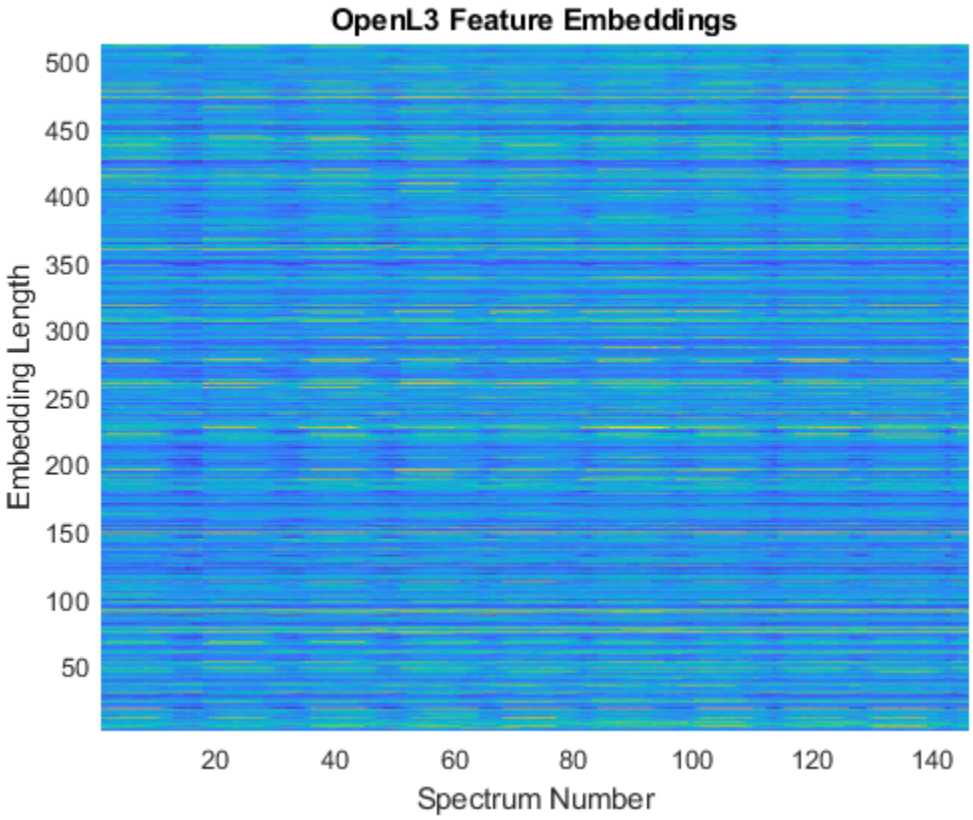


Create an OpenL3 network (this requires Deep Learning Toolbox) using the same `'SpectrumType'`.

```
net = openl3('SpectrumType','linear');
```

Extract and visualize the audio embeddings.

```
embeddings = predict(net,features);
surf(embeddings,'EdgeColor','none')
view([90,-90])
axis([1 numSpectOvLap90 1 numSpectOvLap90])
xlabel('Embedding Length')
ylabel('Spectrum Number')
title('OpenL3 Feature Embeddings')
axis tight
```



Input Arguments

`audioIn` — Input signal

column vector | matrix

Input signal, specified as a column vector or matrix. If you specify a matrix, `openl3Preprocess` treats the columns of the matrix as individual audio channels.

Data Types: `single` | `double`

`fs` — Sample rate (Hz)

positive scalar

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

Data Types: `single` | `double`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `openl3Preprocess(audioIn,fs,'SpectrumType','mel256')`

`OverlapPercentage` — Percentage overlap between consecutive spectrograms
`90` (default) | scalar in the range [0,100)

Percentage overlap between consecutive spectrograms, specified as a scalar in the range [0,100).

Data Types: `single` | `double`

`SpectrumType` — Spectrum type
`'mel128'` (default) | `'mel256'` | `'linear'`

Spectrum type generated from audio and used as input to the neural network, specified as one of these:

- `'mel128'` –– Generates mel spectrograms using 128 mel bands.
- `'mel256'` –– Generates mel spectrograms using 256 mel bands.
- `'linear'` –– Generates positive one-sided spectrograms using an FFT length of 512.

Data Types: `char` | `string`

Output Arguments

`features` — Spectrograms that can be fed to OpenL3 pretrained network
N-by-*M*-by-1-by-*K* array

Spectrograms generated from `audioIn`, returned as an *N*-by-*M*-by-1-by-*K* array.

When you specify `'SpectrumType'` as one of these:

- `'mel128'` –– The dimensions are 128-by-199-by-1-by-*K*, where 128 is the number of mel bands and 199 is the number of time hops.
- `'mel256'` –– The dimensions are 256-by-199-by-1-by-*K*, where 256 is the number of mel bands and 199 is the number of time hops.
- `'linear'` –– The dimensions are 257-by-197-by-1-by-*K*, where 257 is the positive one-sided FFT length and 197 is the number of time hops.
- *K* represents the number of spectrograms and depends on the length of `audioIn`, the number of channels in `audioIn`, as well as `OverlapPercentage`.

Data Types: `single`

Version History

Introduced in R2021a

References

[1] Cramer, Jason, et al. "Look, Listen, and Learn More: Design Choices for Deep Audio Embeddings." In *ICASSP 2019 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, IEEE, 2019, pp. 3852-56. *DOI.org (Crossref)*, doi:/10.1109/ICASSP.2019.8682475.

Extended Capabilities

C/C++ Code Generation

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

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

See Also

`openl3` | `vggish` | `vggishEmbeddings` | `openl3Embeddings` | `classifySound` | `audioFeatureExtractor`

openl3

OpenL3 neural network

Syntax

```
net = openl3
net = openl3(Name,Value)
```

Description

`net = openl3` returns a pretrained OpenL3 model.

This function requires both Audio Toolbox and Deep Learning Toolbox.

`net = openl3(Name,Value)` specifies options using one or more Name, Value arguments. For example, `net = openl3('EmbeddingLength',6144)` specifies the output embedding length as 6144.

Examples

Download OpenL3 Network

Download and unzip the Audio Toolbox™ model for OpenL3.

Type `openl3` at the Command Window. If the Audio Toolbox model for OpenL3 is not installed, the function provides a link to the location of the network weights. To download the model, click the link. Unzip the file to a location on the MATLAB path.

Alternatively, execute these commands to download and unzip the OpenL3 model to your temporary directory.

```
downloadFolder = fullfile(tempdir,'OpenL3Download');
loc = websave(downloadFolder,'https://ssd.mathworks.com/supportfiles/audio/openl3.zip');
OpenL3Location = tempdir;
unzip(loc,OpenL3Location)
addpath(fullfile(OpenL3Location,'openl3'))
```

Check that the installation is successful by typing `openl3` at the Command Window. If the network is installed, then the function returns a `DAGNetwork` (Deep Learning Toolbox) object.

```
openl3
```

```
ans =
  DAGNetwork with properties:

         Layers: [30×1 nnet.cnn.layer.Layer]
    Connections: [29×2 table]
     InputNames: {'in'}
    OutputNames: {'out'}
```

Load Pretrained OpenL3 Network

Load a pretrained OpenL3 convolutional neural network and examine the layers and classes.

Use `openl3` to load the pretrained OpenL3 network. The output `net` is a `DAGNetwork` (Deep Learning Toolbox) object.

```
net = openl3
```

```
net =
  DAGNetwork with properties:

         Layers: [30×1 nnet.cnn.layer.Layer]
    Connections: [29×2 table]
     InputNames: {'in'}
    OutputNames: {'out'}
```

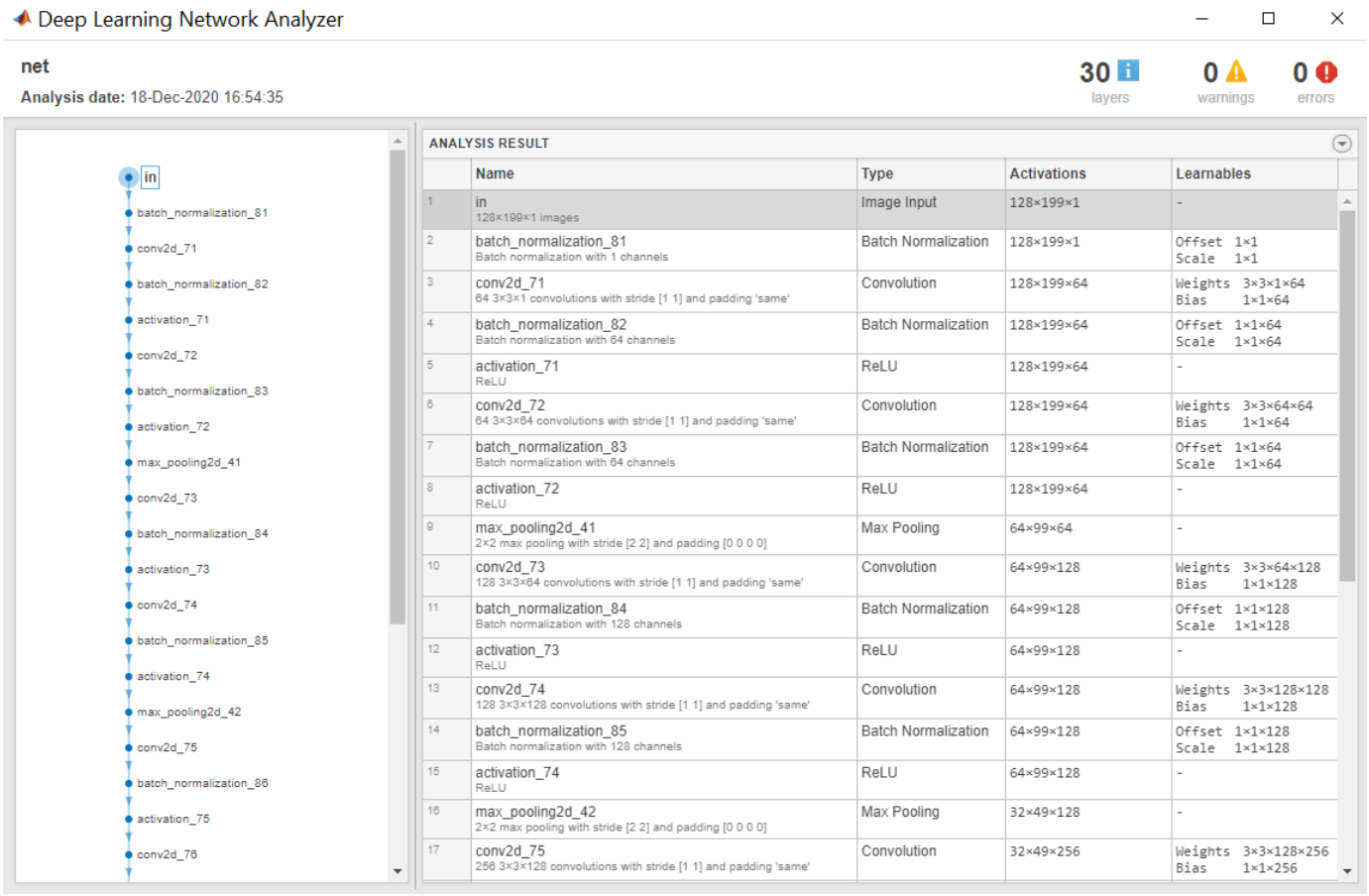
View the network architecture using the `Layers` property. The network has 30 layers. There are 16 layers with learnable weights, of which eight are batch normalization layers and eight are convolutional layers.

```
net.Layers
```

```
ans =
  30×1 Layer array with layers:

     1   'in'                      Image Input           128×199×1 images
     2   'batch_normalization_81'  Batch Normalization   Batch normalization with 1 channels
     3   'conv2d_71'               Convolution           64 3×3×1 convolutions with stride [1  1
     4   'batch_normalization_82'  Batch Normalization   Batch normalization with 64 channels
     5   'activation_71'           ReLU                  ReLU
     6   'conv2d_72'               Convolution           64 3×3×64 convolutions with stride [1
     7   'batch_normalization_83'  Batch Normalization   Batch normalization with 64 channels
     8   'activation_72'           ReLU                  ReLU
     9   'max_pooling2d_41'        Max Pooling           2×2 max pooling with stride [2  2] and
    10   'conv2d_73'               Convolution           128 3×3×64 convolutions with stride [1
    11   'batch_normalization_84'  Batch Normalization   Batch normalization with 128 channels
    12   'activation_73'           ReLU                  ReLU
    13   'conv2d_74'               Convolution           128 3×3×128 convolutions with stride [1
    14   'batch_normalization_85'  Batch Normalization   Batch normalization with 128 channels
    15   'activation_74'           ReLU                  ReLU
    16   'max_pooling2d_42'        Max Pooling           2×2 max pooling with stride [2  2] and
    17   'conv2d_75'               Convolution           256 3×3×128 convolutions with stride [1
    18   'batch_normalization_86'  Batch Normalization   Batch normalization with 256 channels
    19   'activation_75'           ReLU                  ReLU
    20   'conv2d_76'               Convolution           256 3×3×256 convolutions with stride [1
    21   'batch_normalization_87'  Batch Normalization   Batch normalization with 256 channels
    22   'activation_76'           ReLU                  ReLU
    23   'max_pooling2d_43'        Max Pooling           2×2 max pooling with stride [2  2] and
    24   'conv2d_77'               Convolution           512 3×3×256 convolutions with stride [1
    25   'batch_normalization_88'  Batch Normalization   Batch normalization with 512 channels
    26   'activation_77'           ReLU                  ReLU
    27   'audio_embedding_layer'   Convolution           512 3×3×512 convolutions with stride [1
    28   'max_pooling2d_44'        Max Pooling           16×24 max pooling with stride [16  24]
    29   'flatten'                 Keras Flatten         Flatten activations into 1-D assuming C
    30   'out'                     Regression Output     mean-squared-error
```

Use `analyzeNetwork` (Deep Learning Toolbox) to visually explore the network.

```
analyzeNetwork(net)
```



Extract OpenL3 Embeddings from Audio Signal

Use `openl3Preprocess` to extract embeddings from an audio signal.

Read in an audio signal.

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

To extract spectrograms from the audio, call the `openl3Preprocess` function with the audio and sample rate. Use 50% overlap and set the spectrum type to linear. The `openl3Preprocess` function returns an array of 30 spectrograms produced using an FFT length of 512.

```
features = openl3Preprocess(audioIn,fs,'OverlapPercentage',50,'SpectrumType','linear');
[posFFTbinsOvLap50,numHopsOvLap50,~,numSpectOvLap50] = size(features)
```

```
posFFTbinsOvLap50 = 257
```

```
numHopsOvLap50 = 197
```

```
numSpectOvLap50 = 30
```

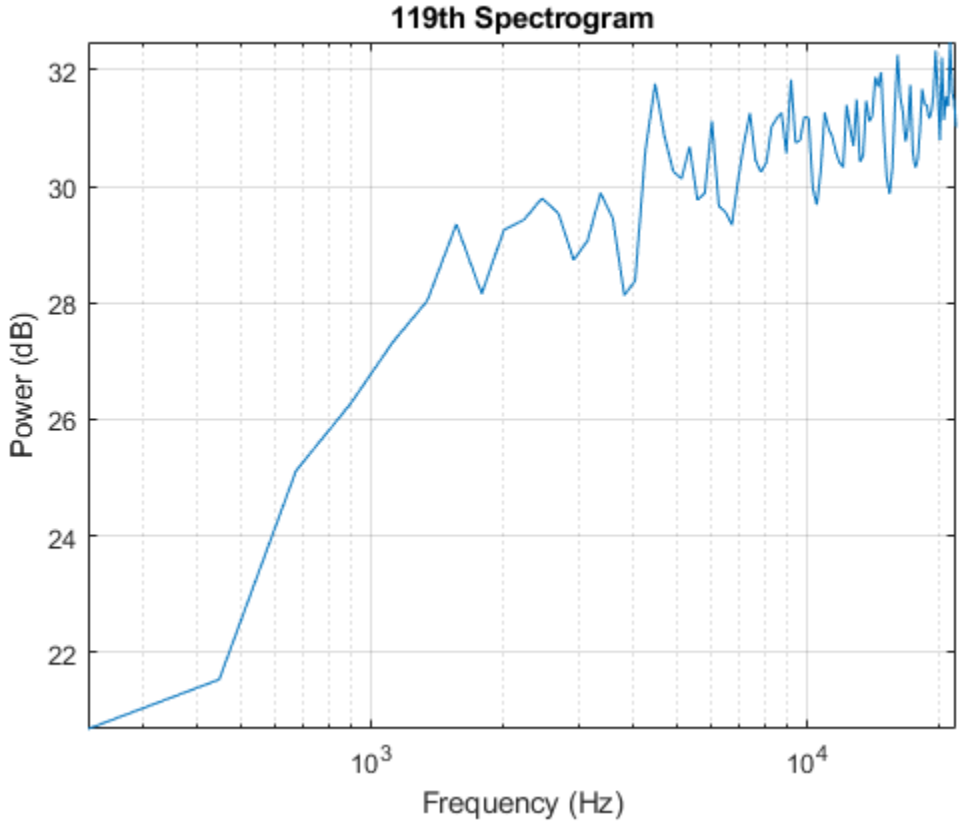
Call `openl3Preprocess` again, this time using the default overlap of 90%. The `openl3Preprocess` function now returns an array of 146 spectrograms.

```
features = openl3Preprocess(audioIn,fs,'SpectrumType','linear');
[posFFTbinsOvLap90,numHopsOvLap90,~,numSpectOvLap90] = size(features)
```

```
posFFTbinsOvLap90 = 257

numHopsOvLap90 = 197

numSpectOvLap90 = 146
```

Visualize one of the spectrograms at random.

```
randSpect = randi(numSpectOvLap90);
viewRandSpect = features(:,:,:,randSpect);
N = size(viewRandSpect,2);
binsToHz = (0:N-1)*fs/N;
nyquistBin = round(N/2);
semilogx(binsToHz(1:nyquistBin),mag2db(abs(viewRandSpect(1:nyquistBin))))
xlabel('Frequency (Hz)')
ylabel('Power (dB)');
title([num2str(randSpect),'th Spectrogram'])
axis tight
grid on
```

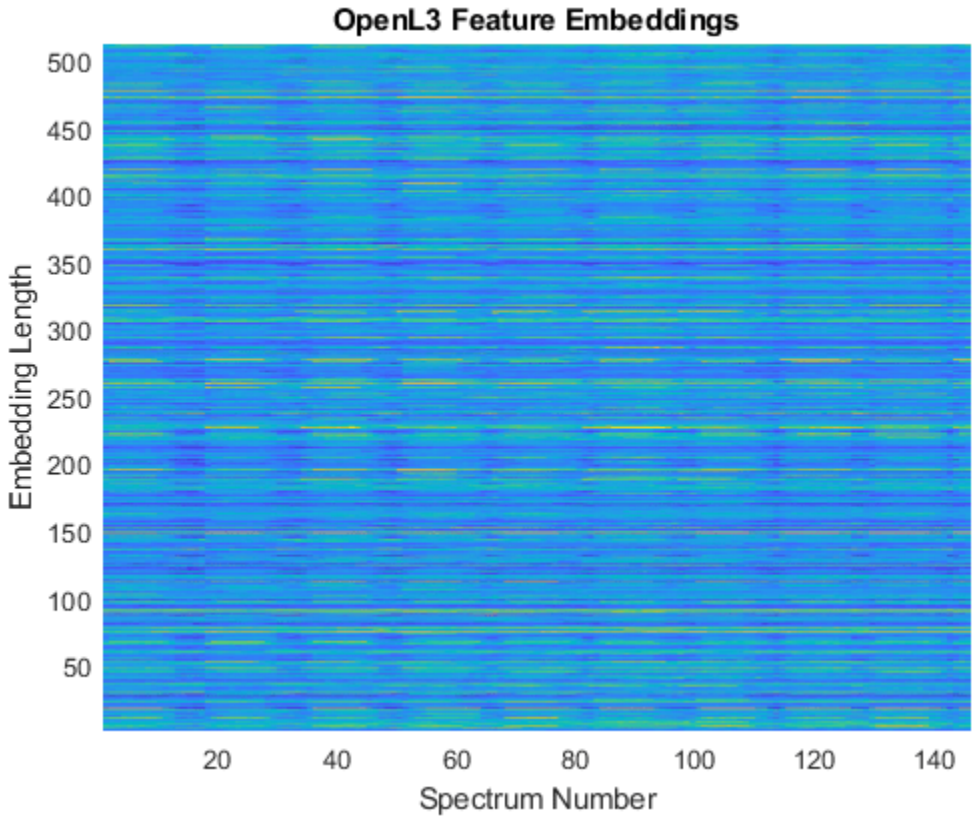


Create an OpenL3 network (this requires Deep Learning Toolbox) using the same `'SpectrumType'`.

```
net = openl3('SpectrumType','linear');
```

Extract and visualize the audio embeddings.

```
embeddings = predict(net,features);
surf(embeddings,'EdgeColor','none')
view([90,-90])
axis([1 numSpectOvLap90 1 numSpectOvLap90])
xlabel('Embedding Length')
ylabel('Spectrum Number')
title('OpenL3 Feature Embeddings')
axis tight
```



Input Arguments

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `openl3('EmbeddingLength',6144)`

`SpectrumType` — Spectrum type
`'mel128'` (default) | `'mel256'` | `'linear'`

Spectrum type generated from audio and used as input to the neural network, specified as `'mel128'`, `'mel256'`, or `'linear'`.

When using `'SpectrumType'` and:

- `'mel128'` –– The network accepts mel spectrograms with 128 mel bands as input. The input dimensions to the network are 128-by-199-by-1-by-*K*, where 128 is the number of mel bands and 199 is the number of time hops.
- `'mel256'` –– The network accepts mel spectrograms with 256 mel bands as input. The input dimensions to the network are 256-by-199-by-1-by-*K*, where 256 is the number of mel bands and 199 is the number of time hops.
- `'linear'` –– The network accepts positive one-sided spectrograms with an FFT length of 257. The input dimensions to the network are 257-by-197-by-1-by-*K*, where 257 is the positive one-sided FFT length and 197 is the number of time hops.

K represents the number of spectrograms. When preprocessing your data with `openl3Preprocess`, you must use the same `'SpectrumType'`.

Data Types: `char` | `string`

`EmbeddingLength` — Embedding length
512 (default) | 6144

Length of the output audio embedding, specified as 512 or 6144.

Data Types: `single` | `double`

`ContentType` — Audio content type
`'env'` (default) | `'music'`

Audio content type the neural network is trained on, specified as `'env'` or `'music'`.

Set `ContentType` to:

- `'env'` when you want to use a model trained on environmental data.
- `'music'` when you want to use a model trained on musical data.

Data Types: `char` | `string`

Output Arguments

`net` — Pretrained OpenL3 neural network
`DAGNetwork` object

Pretrained OpenL3 neural network, returned as a `DAGNetwork` object.

Version History

Introduced in R2021a

References

[1] Cramer, Jason, et al. "Look, Listen, and Learn More: Design Choices for Deep Audio Embeddings." In *ICASSP 2019 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, IEEE, 2019, pp. 3852-56. *DOI.org (Crossref)*, doi:/10.1109/ICASSP.2019.8682475.

Extended Capabilities

C/C++ Code Generation

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

Usage notes and limitations:

- Only the `activations` and `predict` object functions are supported.
- To create a `SeriesNetwork` object for code generation, see “Load Pretrained Networks for Code Generation” (MATLAB Coder).

GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

Usage notes and limitations:

- Only the `activations`, `classify`, `predict`, `predictAndUpdateState`, and `resetState` object functions are supported.
- To create a `SeriesNetwork` object for code generation, see “Load Pretrained Networks for Code Generation” (GPU Coder).

See Also

`openl3Preprocess` | `openl3Embeddings` | `vggish` | `classifySound` | `vggishEmbeddings` | `audioFeatureExtractor`

speakerRecognition

Pretrained speaker recognition system

Syntax

```
sr = speakerRecognition
```

Description

`sr = speakerRecognition` returns a pretrained speaker recognition system, `'ivec-english-16kHz'`. The `'ivec-english-16kHz'` system is an instance of an object of type `ivectorSystem` trained on the LibriSpeech data set.

Examples

Perform Speaker Verification

This example uses a pretrained speaker recognition system, `'ivec-english-16kHz'`. The `'ivec-english-16kHz'` system is an instance of `ivectorSystem` trained on the LibriSpeech data set.

Download the pretrained speaker recognition system into your temporary directory, whose location is specified by the MATLAB® `tempdir` command. If you want to place the data files in a folder different from `tempdir`, change the directory name. Add the temporary directory to the search path. Create an i-vector system.

```
fname = "ivec-english-16kHz.zip";
URL = "https://ssd.mathworks.com/supportfiles/audio/speakerRecognition/" + fname;

unzip(URL,tempdir);

addpath(tempdir)

sr = speakerRecognition;
```

Read two speech signals, each of which contains the phrase "volume up" spoken out loud several times with different intonations. In one of the signals, the speaker is male. In the other signal, the speaker is female.

Read each signal and split it into two parts. One of the parts is used to enroll the speaker. The other part is used for speaker verification and identification.

```
[bf,fs] = audioread("MaleVolumeUp-16-mono-6secs.ogg");
enrollBF = bf(1:3*fs);
testBF = bf(3*fs+1:end);
bfLabel = "BF";

[rd,fs] = audioread("FemaleVolumeUp-16-mono-11secs.ogg");
enrollRD = rd(1:5*fs);
testRD = rd(5*fs+1:end);
rdLabel = "RD";
```

Enroll the speakers into the speaker recognition system. This creates a template of the speaker that can be used for verification or identification.

```
 enroll(sr,{enrollBF,enrollRD},[bfLabel,rdLabel])
```

```
Extracting i-vectors ...done.
Enrolling i-vectors .....done.
Enrollment complete.
```

Call the `identify` function on the test data.

```
candidates = identify(sr,testBF)
```

```
candidates=2×2 table
    Label      Score
    _____    _________

     BF        0.99474
     RD      0.0017846
```

```
candidates = identify(sr,testRD)
```

```
candidates=2×2 table
    Label      Score
    _____    __________

     RD         0.24113
     BF      3.2741e-05
```

Call the `verify` function with the test data to confirm that the system correctly accepts or rejects speakers.

```
isVerified = verify(sr,testBF,bfLabel)
```

```
isVerified = logical
   1
```

```
isVerified = verify(sr,testBF,rdLabel)
```

```
isVerified = logical
   0
```

```
isVerified = verify(sr,testRD,rdLabel)
```

```
isVerified = logical
   1
```

```
isVerified = verify(sr,testRD,bfLabel)
```

```
isVerified = logical
   0
```

Call the `info` function to get information about how the model was trained.

```
info(sr)

Header
  - This system was trained using the LibriSpeech train and development sets.
  LibriSpeech is an approximately 1000-hour corpus of read English speech sampled at 16 kHz.
  - The detection error tradeoff was determined by enrolling one file from each speaker in the
  LibriSpeech test set, and then evaluating exhaustive pairs of the enrolled and remaining data.
  - The system was calibrated using the train-clean-100 and dev-clean data of LibriSpeech.

i-vector system input
  Input feature vector length: 60
  Input data type: double

trainExtractor
  Train signals: 286808
  UBMNumComponents: 2048
  UBMNumIterations: 10
  TVSRank: 512
  TVSNumIterations: 5

trainClassifier
  Train signals: 286807
  Train labels: 1 (91), 100043 (31) ... and 5652 more
  NumEigenvectors: 200
  PLDANumDimensions: 200
  PLDANumIterations: 5

calibrate
  Calibration signals: 31242
  Calibration labels: 103 (102), 1034 (96) ... and 289 more

detectionErrorTradeoff
  Evaluation signals: 5382
  Evaluation labels: 102255 (46), 1066 (24) ... and 175 more
```

Remove the temporary directory from the search path.

```
rmpath(tempdir)
```

Output Arguments

`sr` — Pretrained speaker recognition system

`ivectorSystem` object

Pretrained speaker recognition system, returned as an object of type `ivectorSystem`.

Version History

Introduced in R2021b

References

[1] Panayotov, Vassil, Guoguo Chen, Daniel Povey, and Sanjeev Khudanpur. “Librispeech: An ASR Corpus Based on Public Domain Audio Books.” In *2015 IEEE International Conference on*

Acoustics, Speech and Signal Processing (ICASSP), 5206–10. South Brisbane, Queensland, Australia: IEEE, 2015. https://doi.org/10.1109/ICASSP.2015.7178964.

See Also

`classifySound` | `ivectorSystem` | `pitchnn`

Topics

“i-vector Score Normalization”
“i-vector Score Calibration”

External Websites

https://www.openslr.org/12

acousticRoughness

Perceived roughness of acoustic signal

Syntax

```
roughness = acousticRoughness(audioIn,fs)
roughness = acousticRoughness(audioIn,fs,calibrationFactor)
roughness = acousticRoughness(specificLoudnessIn)
roughness = acousticRoughness(___,Name,Value)

[roughness,specificRoughness] = acousticRoughness(___)
[roughness,specificRoughness,fMod] = acousticRoughness(___)

acousticRoughness(___)
```

Description

`roughness = acousticRoughness(audioIn,fs)` returns roughness strength in aspers based on Zwicker [1] and ISO 532-1 time-varying loudness [2].

`roughness = acousticRoughness(audioIn,fs,calibrationFactor)` specifies a nondefault microphone calibration factor used to compute loudness.

`roughness = acousticRoughness(specificLoudnessIn)` computes roughness using high-resolution time-varying specific loudness.

`roughness = acousticRoughness(___,Name,Value)` specifies options using one or more Name, Value pair arguments. For example, `roughness = acousticRoughness(audioIn,fs,'SoundField','diffuse')` returns roughness assuming a diffuse sound field.

`[roughness,specificRoughness] = acousticRoughness(___)` also returns specific roughness strength.

`[roughness,specificRoughness,fMod] = acousticRoughness(___)` also returns the dominant modulation frequency detected by the algorithm.

`acousticRoughness(___)` with no output arguments plots roughness strength and specific roughness strength and displays the modulation frequency textually. If `audioIn` is stereo, the 3-D plot shows the sum of both channels.

Examples

Measure Acoustic Roughness

Measure acoustic roughness based on [1] on page 2-89 and ISO 532-1 [2] on page 2-89. Assume a free-field reference pressure of 20 micropascals and a recording level calibration such that a 1 kHz tone registers as 100 dB on a SPL meter.

Read in a stereo audio file and convert to mono.

```
[audioInStereo,fs] = audioread('Engine-16-44p1-stereo-20sec.wav');
audioIn = (audioInStereo(:,1) + audioInStereo(:,2)) / 2;
```

Compute acoustic roughness on the mono audio signal and display the average value.

```
roughness = acousticRoughness(audioIn,fs);
meanRoughness = mean(roughness);
displayOutput = ['Average computed value of acoustic roughness is ',num2str(meanRoughness),' aspe
disp(displayOutput)
```

```
Average computed value of acoustic roughness is 0.17901 aspers.
```

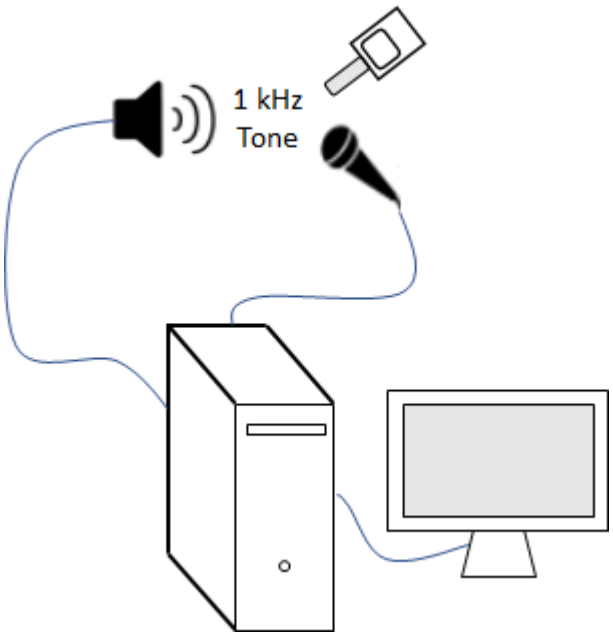
References

[1] Zwicker, Eberhard, and Hugo Fastl. *Psychoacoustics: Facts and Models*. Vol. 22. Springer Science & Business Media, 2013.

[2] ISO 532-1:2017(E). "Acoustics – Methods for calculating loudness – Part 1: Zwicker method." *International Organization for Standardization*.

Roughness Measurements Using Calibrated Microphone

Set up an experiment as indicated by the diagram.



Create an `audioDeviceReader` object to read from the microphone and an `audioDeviceWriter` object to write to your speaker.

```
fs = 48e3;
deviceReader = audioDeviceReader(fs,"SamplesPerFrame",2048);
deviceWriter = audioDeviceWriter(fs);
```

Create an `audioOscillator` object to generate a 1 kHz sinusoid.

```
osc = audioOscillator("sine",1e3,"SampleRate",fs,"SamplesPerFrame",2048);
```

Create a `dsp.AsyncBuffer` object to buffer data acquired from the microphone.

```
dur = 5;
buff = dsp.AsyncBuffer(dur*fs);
```

For 5 seconds, play the sinusoid through your speaker and record using your microphone. While the audio streams, note the loudness as reported by your SPL meter. Once complete, read the contents of the buffer object.

```
numFrames = dur*(fs/osc.SamplesPerFrame);
for ii = 1:numFrames
    audioOut = osc();
    deviceWriter(audioOut);

    audioIn = deviceReader();
    write(buff,audioIn);
end

SPLreading = 69.7;

micRecording = read(buff);
```

To compute the calibration factor for the microphone, use the `calibrateMicrophone` function.

```
calibrationFactor = calibrateMicrophone(micRecording(fs+1:end),deviceReader.SampleRate,SPLreading
```

You can now use the calibration factor you determined to measure the roughness of any sound that is acquired through the same microphone recording chain.

Perform the experiment again by adding 100% amplitude modulation at 40 Hz. To create the modulation signal, use the `audioOscillator` object and specify the amplitude as `0.5` and the DC offset as `0.5` to oscillate between 0 and 1.

```
mod = audioOscillator("sine",40,"SampleRate",fs, ...
    "Amplitude",0.5,"DCOffset",0.5,"SamplesPerFrame",2048);

dur = 5;
buff = dsp.AsyncBuffer(dur*fs);
numFrames = dur*(fs/osc.SamplesPerFrame);
for ii = 1:numFrames
    audioOut = osc().*mod();
    deviceWriter(audioOut);

    audioIn = deviceReader();
    write(buff,audioIn);
end

micRecording = read(buff);
```

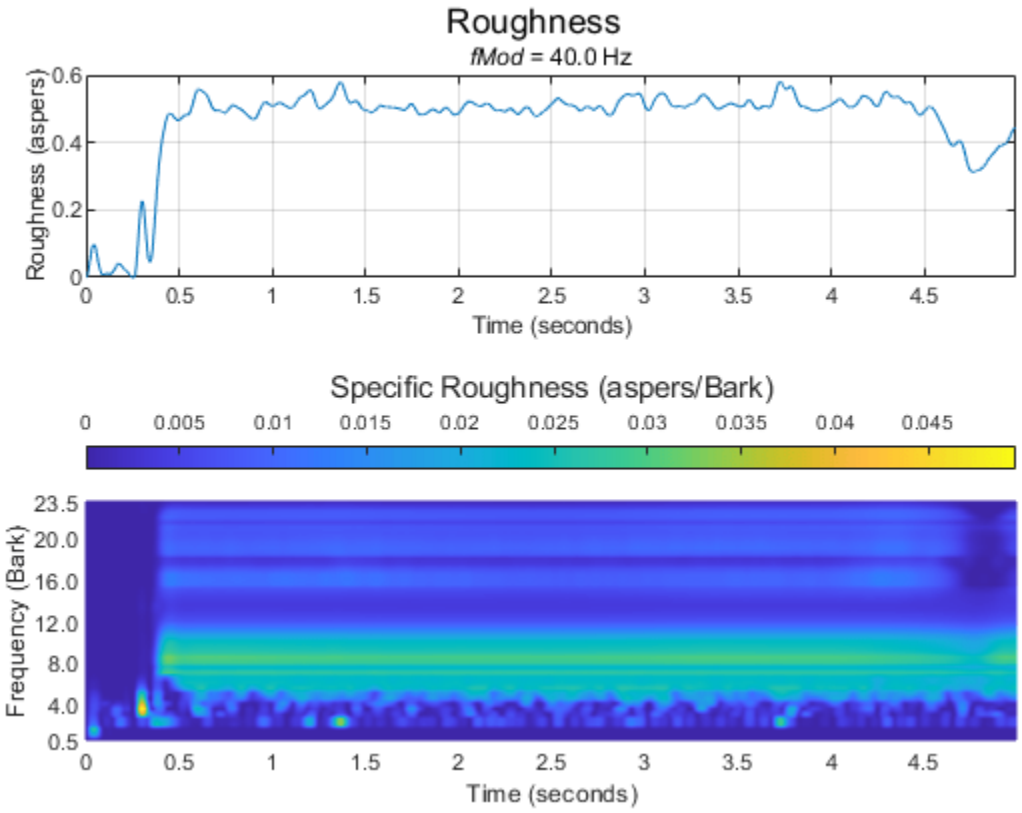
Call the `acousticRoughness function` with the microphone recording, sample rate, and calibration factor. The roughness reported from `acousticRoughness` uses the true acoustic

loudness measurement as specified by ISO 532-1. Display the average roughness strength over the 5 seconds and plot roughness and specific roughness.

```
roughness = acousticRoughness(micRecording,deviceReader.SampleRate,calibrationFactor);
fprintf('Average roughness = %d (asper)',mean(roughness(2001:end,:)))
```

```
Average roughness = 4.992723e-01 (asper)
```

```
acousticRoughness(micRecording,deviceReader.SampleRate,calibrationFactor)
```



Measure Roughness from Specific Loudness

Read in an audio file.

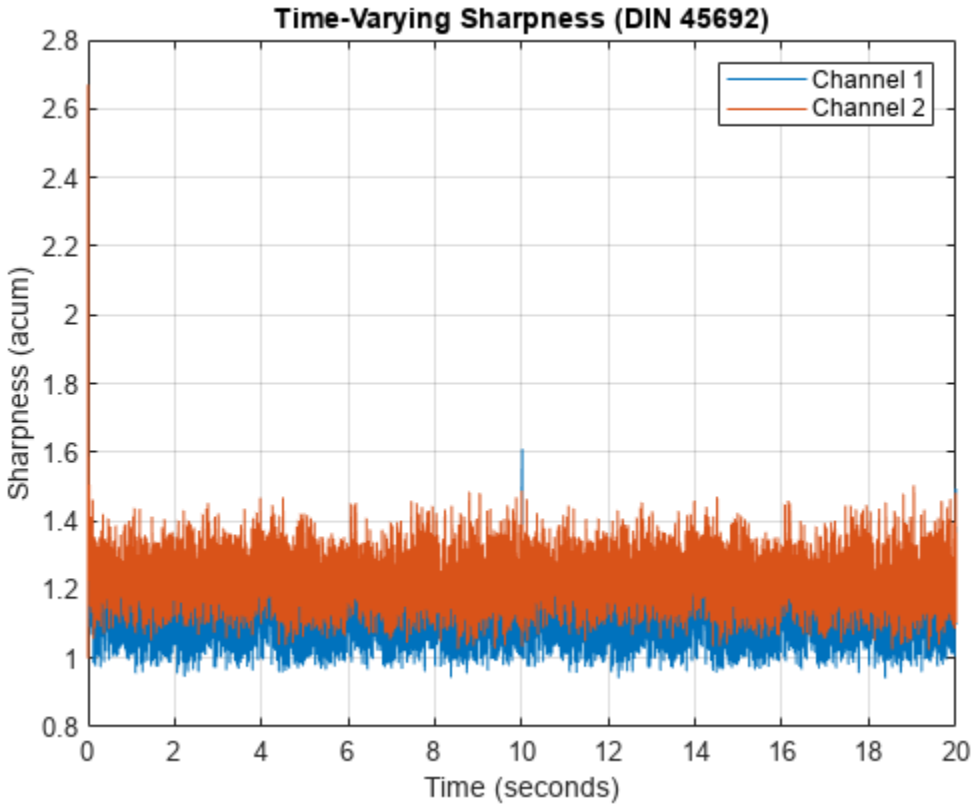
```
[audioIn,fs] = audioread("Engine-16-44p1-stereo-20sec.wav");
```

Call the `acousticLoudness` function using high time resolution to calculate the specific loudness.

```
[~,specificLoudnessHD] = acousticLoudness(audioIn,fs,'TimeVarying',true,'TimeResolution','high');
```

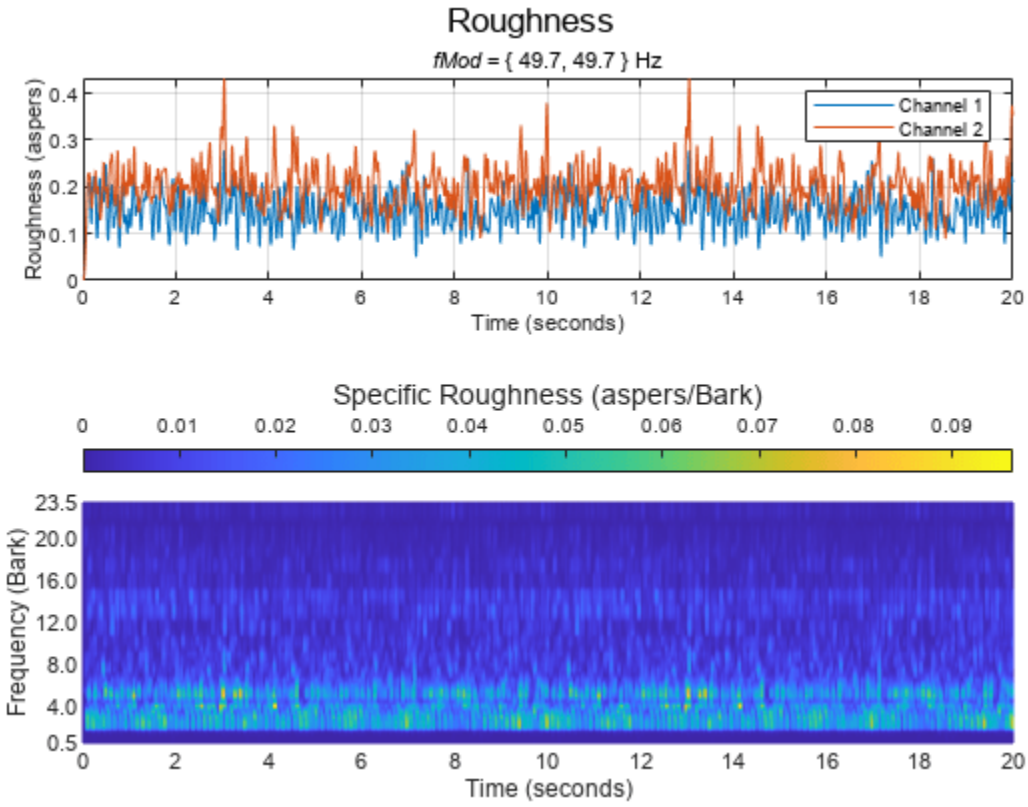
Call the `acousticSharpness` function using standard resolution specific loudness without any output arguments to plot the acoustic sharpness.

```
specificLoudness = specificLoudnessHD(1:4:end,:,:);
acousticSharpness(specificLoudness,'TimeVarying',true)
```



Call `acousticRoughness` without any outputs to plot the acoustic roughness.

```
acousticRoughness(specificLoudnessHD)
```



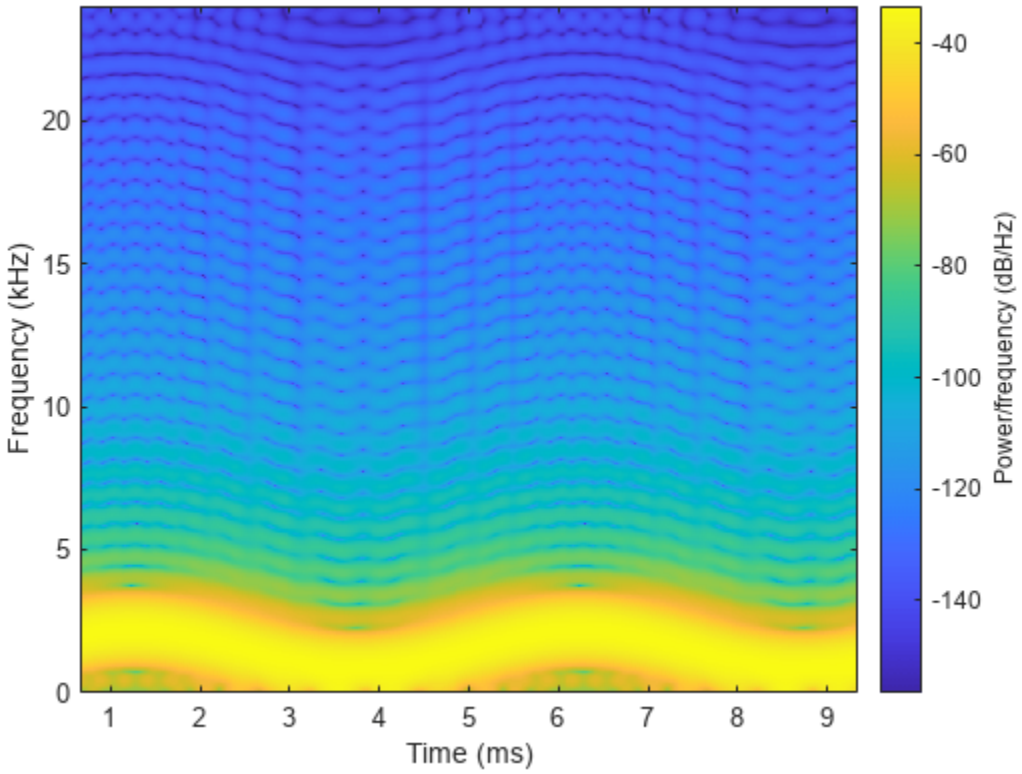
Effect of Frequency Modulation on Acoustic Roughness

Generate a pure tone with a 1500 Hz center frequency and approximately 700 Hz frequency deviation at a modulation frequency of 200 Hz.

```
fs = 48e3;

fMod = 200 ;
dur = 5 ;

numSamples = dur*fs;
t = (0:numSamples-1)/fs;

tone = sin(2*pi*t*fMod)';

fc = 1500 ;
excursionRatio = 0.47 ;

excursion = 2*pi*(fc*excursionRatio/fs);
audioIn = modulate(tone,fc,fs,'fm',excursion);
```

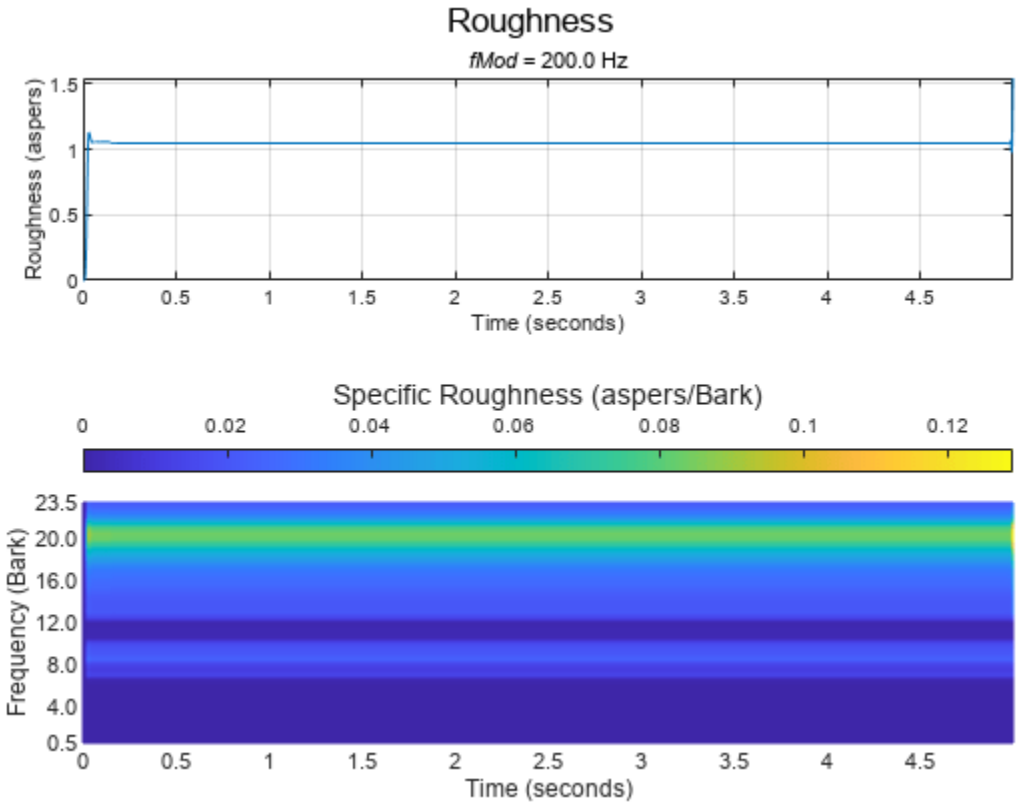
Listen to the audio and plot a spectrogram of the first 10 ms.

```
sound(audioIn,fs)
spectrogram(audioIn(1:0.01*fs),hann(64,'periodic'),63,1024,fs,'yaxis')
```



Call the `acousticRoughness` function with no output arguments to plot the acoustic roughness strength.

```
acousticRoughness(audioIn,fs);
```



Specify Known Roughness Modulation Frequency

The `acousticRoughness` function enables you to specify a known roughness frequency. If you do not specify a known roughness frequency, the function auto detects the roughness.

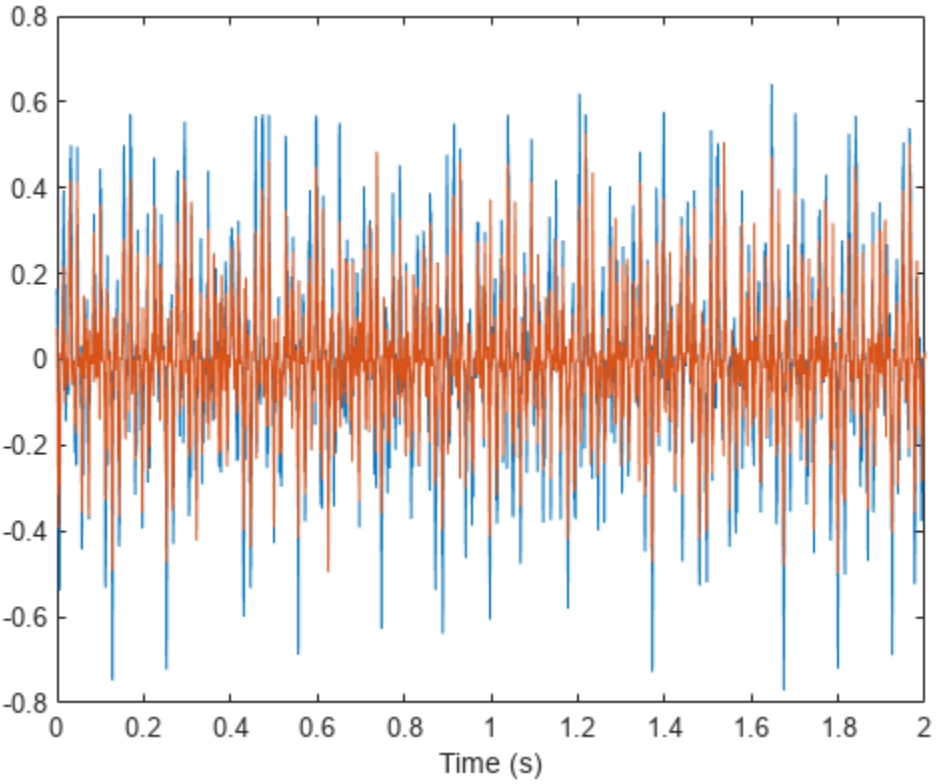
Create a `dsp.AudioFileReader` object to read in an audio signal frame-by-frame. Create an `audioOscillator` object to create a modulation wave. Apply the modulation wave to the audio file.

```
fileReader = dsp.AudioFileReader('Engine-16-44p1-stereo-20sec.wav');

fMod = 72.41 ;

amplitude = 0.5 ;

osc = audioOscillator('sine',fMod, ...
    "DCOffset",0.5, ...
    "Amplitude",amplitude, ...
    "SampleRate",fileReader.SampleRate, ...
    "SamplesPerFrame",fileReader.SamplesPerFrame);

testSignal = [];
while ~isDone(fileReader)
    x = fileReader();
```

```
    testSignal = [testSignal;osc().*fileReader()];
end
```

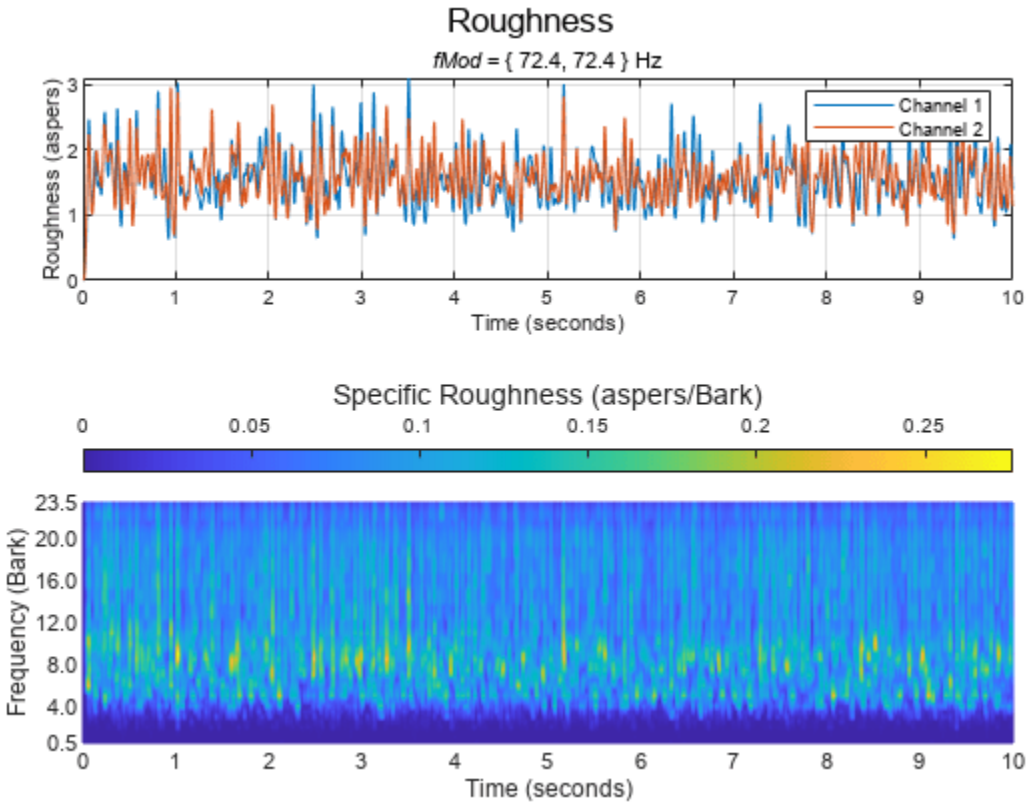
Listen to 2 seconds of the test signal and plot its waveform.

```
samplesToView = 1:2*fileReader.SampleRate;
sound(testSignal(samplesToView,:),fileReader.SampleRate);

plot(samplesToView/fileReader.SampleRate,testSignal(samplesToView,:))
xlabel('Time (s)')
```



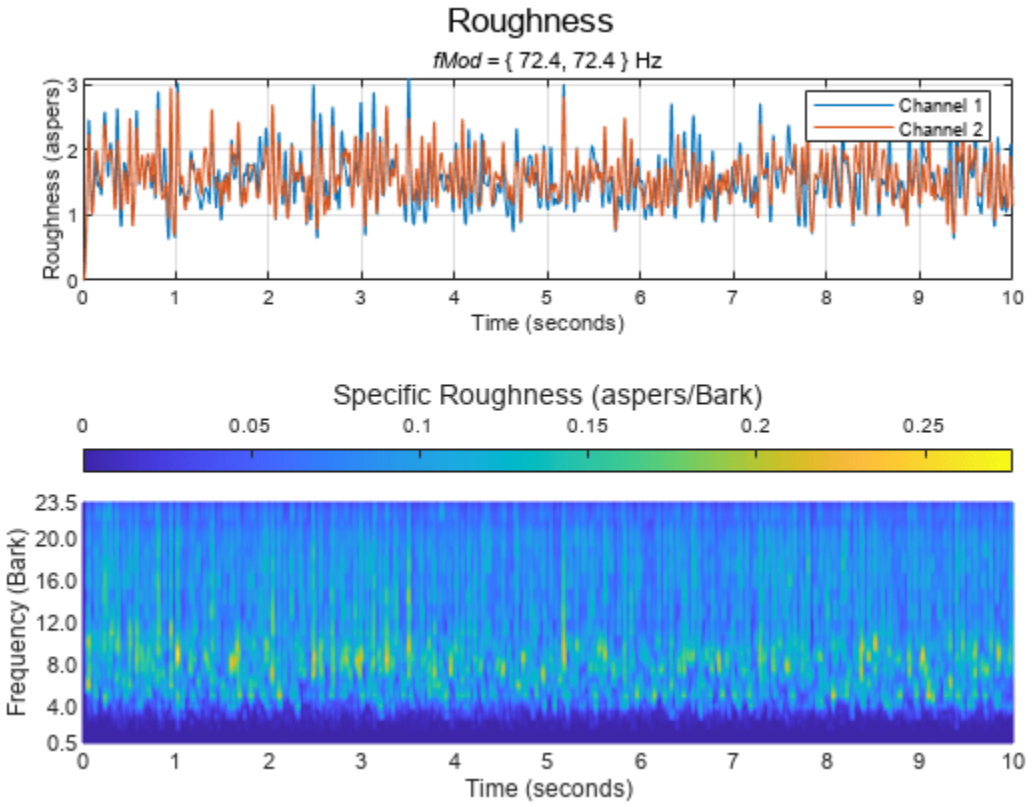
Plot the acoustic roughness. The detected frequency of the modulation is displayed textually.

```
acousticRoughness(testSignal,fileReader.SampleRate);
```



Specify the known modulation frequency and then plot the acoustic roughness again.

```
acousticRoughness(testSignal,fileReader.SampleRate,'ModulationFrequency',fMod)
```



Input Arguments

`audioIn` — Audio input

column vector | two-column matrix

Audio input, specified as a column vector (mono) or matrix with two columns (stereo).

Tip To measure roughness strength given any modulation frequency, the recommended minimum signal duration is 0.5 seconds.

Data Types: `single` | `double`

`fs` — Sample rate (Hz)

positive scalar

Sample rate in Hz, specified as a positive scalar. The recommended sample rate for new recordings is 48 kHz.

Note The minimum acceptable sample rate is 8 kHz.

Data Types: `single` | `double`

`calibrationFactor` — Microphone calibration factor

`sqrt(8)` | positive scalar

Microphone calibration factor, specified as a positive scalar. The default calibration factor corresponds to a full-scale 1 kHz sine wave with a sound pressure level of 100 dB (SPL). To compute the calibration factor specific to your system, use the `calibrateMicrophone` function.

Data Types: `single` | `double`

`specificLoudnessIn` — Specific loudness (sones/Bark)

T-by-240-by-*C*

Specific loudness in sones/Bark, specified as a *T*-by-240-by-*C* array, where:

- *T* is 1 per 0.5 ms of the time-varying signal (high resolution loudness).
- 240 is the number of Bark bins in the domain for specific loudness. The Bark bins are `0.1:0.1:24`.
- *C* is the number of channels.

You can use the `acousticLoudness` function to calculate time varying specific loudness using this syntax.

```
[~,specificLoudness] = acousticLoudness(audioIn,fs,'TimeVarying',true,'TimeResolution','high');
```

Data Types: `single` | `double`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `acousticRoughness(audioIn,fs,'ModulationFrequency',100)`

`ModulationFrequency` — Known roughness modulation frequency (Hz)

`'auto-detect'` (default) | scalar or two-element vector with values in the range `[1,1000]`

Known modulation frequency in Hz, specified as the comma-separated pair consisting of `'ModulationFrequency'` and a scalar or two-element vector with values in the range `[1,1000]`.

Set `ModulationFrequency` to:

- `'auto-detect'` when you want the function to detect the modulation frequency automatically. When you select this option the function limits the search range to between 10 and 400 Hz.
- a scalar if the input is mono.
- a scalar or two-element vector if the input is stereo.

Data Types: `single` | `double` | `char` | `string`

`SoundField` — Sound field

`'free'` (default) | `'diffuse'`

Sound field of audio recording, specified as the comma-separated pair of `'SoundField'` and `'free'` or `'diffuse'`.

Data Types: `char` | `string`

`PressureReference` — Reference pressure (Pa)
`20e-6` (default) | positive scalar

Reference pressure for dB calculation in pascals, specified as the comma-separated pair of `'PressureReference'` and a positive scalar. The default value, 20 micropascals, is the common value of air.

Data Types: `single` | `double`

Output Arguments

`roughness` — Roughness strength (asper)
K-by-1 | *K*-by-2

Roughness strength in asper, returned as a *K*-by-1 column vector or *K*-by-2 matrix of independent channels. *K* corresponds to the time dimension.

Note Roughness is reported for each channel independently at every 0.5 ms interval.

Data Types: `single` | `double`

`specificRoughness` — Specific roughness strength (asper/Bark)
K-by-47 matrix | *K*-by-47-by-2 array

Specific roughness strength in asper/Bark, returned as a *K*-by-47 matrix or a *K*-by-47-by-2 array. The first dimension of `specificRoughness`, *K*, corresponds to the time dimension and matches the first dimension of `roughness`. The second dimension of `specificRoughness`, 47, corresponds to bands on the Bark scale, with centers in the range of [0.5, 23.5], in increments of 0.5. The third dimension of `specificRoughness` corresponds to the number of channels and matches the second dimension of `roughness`.

Data Types: `single` | `double`

`fMod` — Dominant modulation frequency (Hz)
scalar (mono input) | 1-by-2 vector (stereo input)

Dominant modulation frequency in Hz, returned as a scalar for mono input or a 1-by-2 vector for stereo input.

Data Types: `single` | `double`

Algorithms

Acoustic roughness strength is a perceptual measurement of modulations in amplitude or frequency that are too high to discern separately. The acoustic loudness algorithm is described in [2] and implemented in the `acousticLoudness` function. The acoustic roughness calculation is described in [1]. The algorithm for acoustic roughness defines the roughness of 1 asper as a 1 kHz tone at 60 dB with a 100% amplitude modulation at 70 Hz [3]. The algorithm is outlined as follows:

$$roughness = cal \int_{z=0}^{24} f_{\mathrm{mod}}\, \Delta L\, dz$$

Where f_{mod} is the detected or known modulation frequency, *cal* is a constant ensuring unity roughness of the reference signal, and ΔL is the perceived modulation depth. If the modulation frequency is not specified when calling `acousticRoughness`, it is auto-detected by peak-picking a frequency-domain representation of the acoustic loudness. The perceived modulation depth ΔL is calculated by passing rectified specific loudness bands through ½ octave filters centered around f_{mod}, followed by a lowpass filter to determine the envelope.

Version History

Introduced in R2021a

References

[1] Zwicker, Eberhard, and Hugo Fastl. *Psychoacoustics: Facts and Models*. Vol. 22. Springer Science & Business Media, 2013.

[2] ISO 532-1:2017(E). "Acoustics – Methods for calculating loudness – Part 1: Zwicker method." *International Organization for Standardization*.

[3] Kalafata, Stamatina. "Sound Levels, Noise Source Identification and Perceptual Analysis in an Intensive Care Unit." Master's thesis, University of Gothenburg, 2014.

Extended Capabilities

C/C++ Code Generation

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

Usage notes and limitations:

- Sample rate must be known (constant) at compile-time.

See Also

`acousticFluctuation` | `acousticLoudness` | `acousticSharpness` | `calibrateMicrophone`

Topics

“Effect of Soundproofing on Perceived Noise Levels”

vggish

VGGish neural network

Syntax

```
net = vggish
```

Description

`net = vggish` returns a pretrained VGGish model.

This function requires both Audio Toolbox and Deep Learning Toolbox.

Examples

Download VGGish Network

Download and unzip the Audio Toolbox™ model for VGGish.

Type `vggish` at the Command Window. If the Audio Toolbox model for VGGish is not installed, then the function provides a link to the location of the network weights. To download the model, click the link. Unzip the file to a location on the MATLAB path.

Alternatively, execute these commands to download and unzip the VGGish model to your temporary directory.

```
downloadFolder = fullfile(tempdir,'VGGishDownload');
loc = websave(downloadFolder,'https://ssd.mathworks.com/supportfiles/audio/vggish.zip');
VGGishLocation = tempdir;
unzip(loc,VGGishLocation)
addpath(fullfile(VGGishLocation,'vggish'))
```

Check that the installation is successful by typing `vggish` at the Command Window. If the network is installed, then the function returns a `SeriesNetwork` (Deep Learning Toolbox) object.

```
vggish
```

```
ans =
  SeriesNetwork with properties:

         Layers: [24×1 nnet.cnn.layer.Layer]
     InputNames: {'InputBatch'}
    OutputNames: {'regressionoutput'}
```

Load Pretrained VGGish Network

Load a pretrained VGGish convolutional neural network and examine the layers and classes.

Use `vggish` to load the pretrained VGGish network. The output `net` is a `SeriesNetwork` (Deep Learning Toolbox) object.

```
net = vggish
```

```
net =
  SeriesNetwork with properties:

         Layers: [24×1 nnet.cnn.layer.Layer]
     InputNames: {'InputBatch'}
    OutputNames: {'regressionoutput'}
```

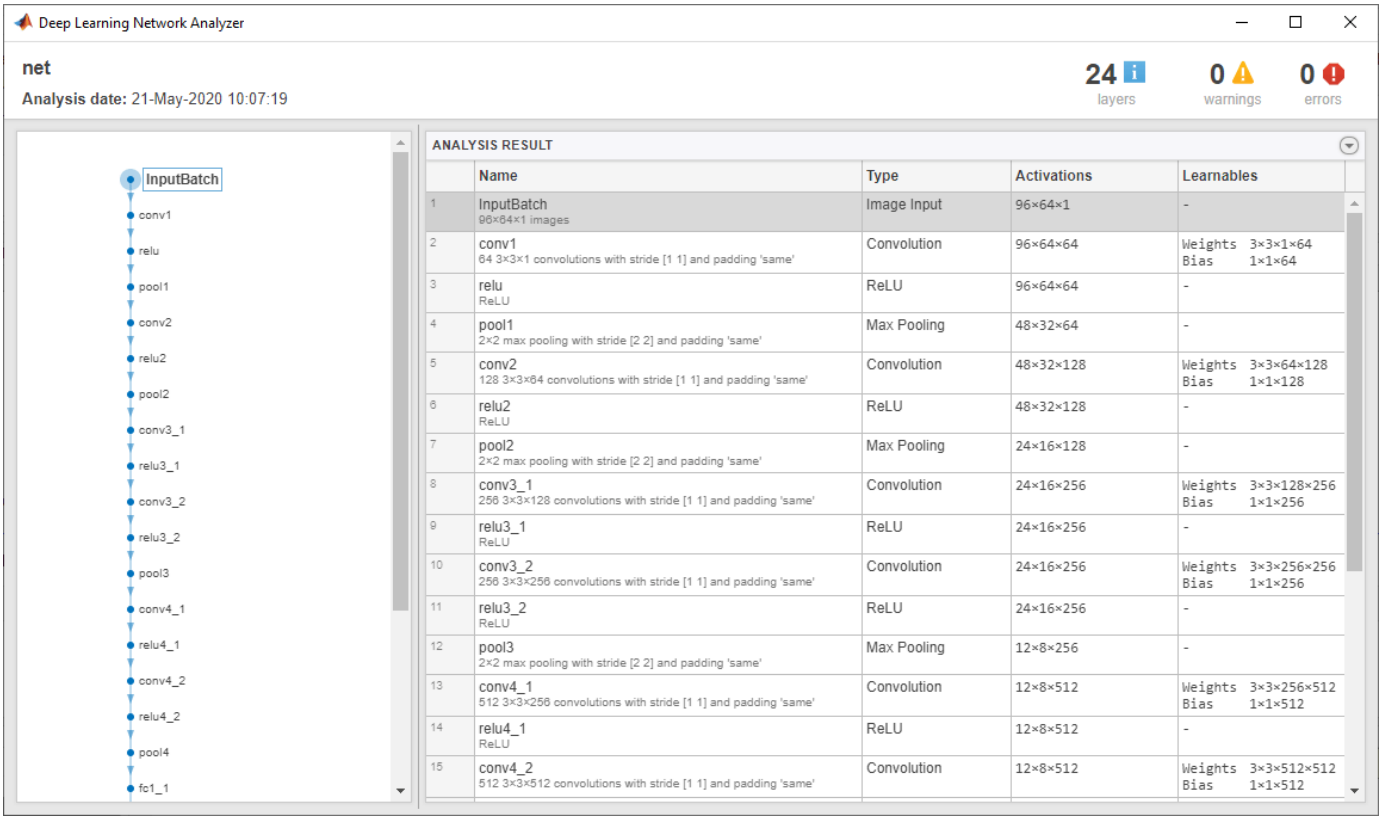
View the network architecture using the `Layers` property. The network has 24 layers. There are nine layers with learnable weights, of which six are convolutional layers and three are fully connected layers.

```
net.Layers
```

```
ans =
  24×1 Layer array with layers:

     1   'InputBatch'         Image Input         96×64×1 images
     2   'conv1'              Convolution         64 3×3×1 convolutions with stride [1  1] and pa
     3   'relu'               ReLU                ReLU
     4   'pool1'              Max Pooling         2×2 max pooling with stride [2  2] and padding
     5   'conv2'              Convolution         128 3×3×64 convolutions with stride [1  1] and
     6   'relu2'              ReLU                ReLU
     7   'pool2'              Max Pooling         2×2 max pooling with stride [2  2] and padding
     8   'conv3_1'            Convolution         256 3×3×128 convolutions with stride [1  1] and
     9   'relu3_1'            ReLU                ReLU
    10   'conv3_2'            Convolution         256 3×3×256 convolutions with stride [1  1] and
    11   'relu3_2'            ReLU                ReLU
    12   'pool3'              Max Pooling         2×2 max pooling with stride [2  2] and padding
    13   'conv4_1'            Convolution         512 3×3×256 convolutions with stride [1  1] and
    14   'relu4_1'            ReLU                ReLU
    15   'conv4_2'            Convolution         512 3×3×512 convolutions with stride [1  1] and
    16   'relu4_2'            ReLU                ReLU
    17   'pool4'              Max Pooling         2×2 max pooling with stride [2  2] and padding
    18   'fc1_1'              Fully Connected     4096 fully connected layer
    19   'relu5_1'            ReLU                ReLU
    20   'fc1_2'              Fully Connected     4096 fully connected layer
    21   'relu5_2'            ReLU                ReLU
    22   'fc2'                Fully Connected     128 fully connected layer
    23   'EmbeddingBatch'     ReLU                ReLU
    24   'regressionoutput'   Regression Output   mean-squared-error
```

Use `analyzeNetwork` (Deep Learning Toolbox) to visually explore the network.

```
analyzeNetwork(net)
```



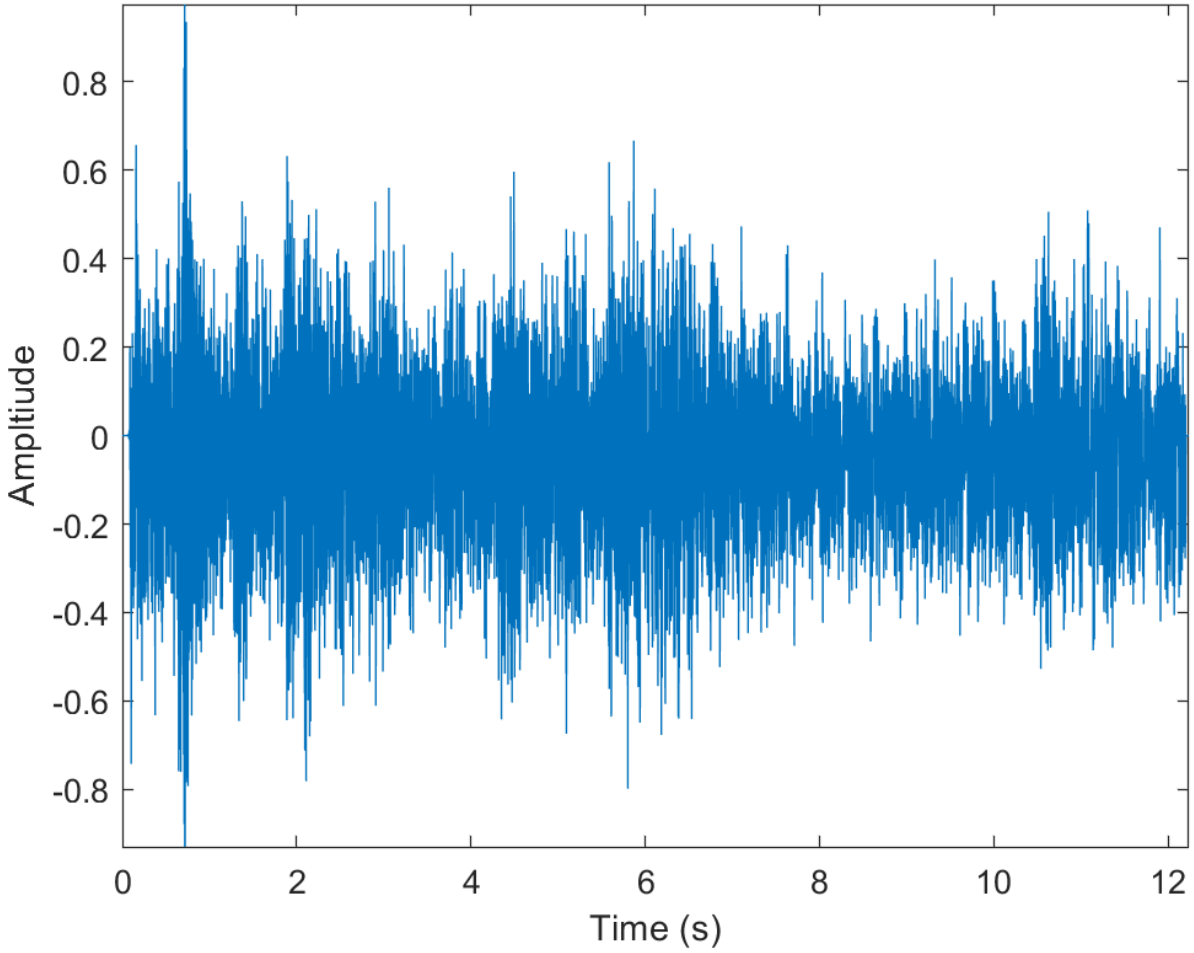
Extract Features Using VGGish

Read in an audio signal to extract feature embeddings from it.

```
[audioIn,fs] = audioread(Ambiance-16-44p1-...);
```

Plot and listen to the audio signal.

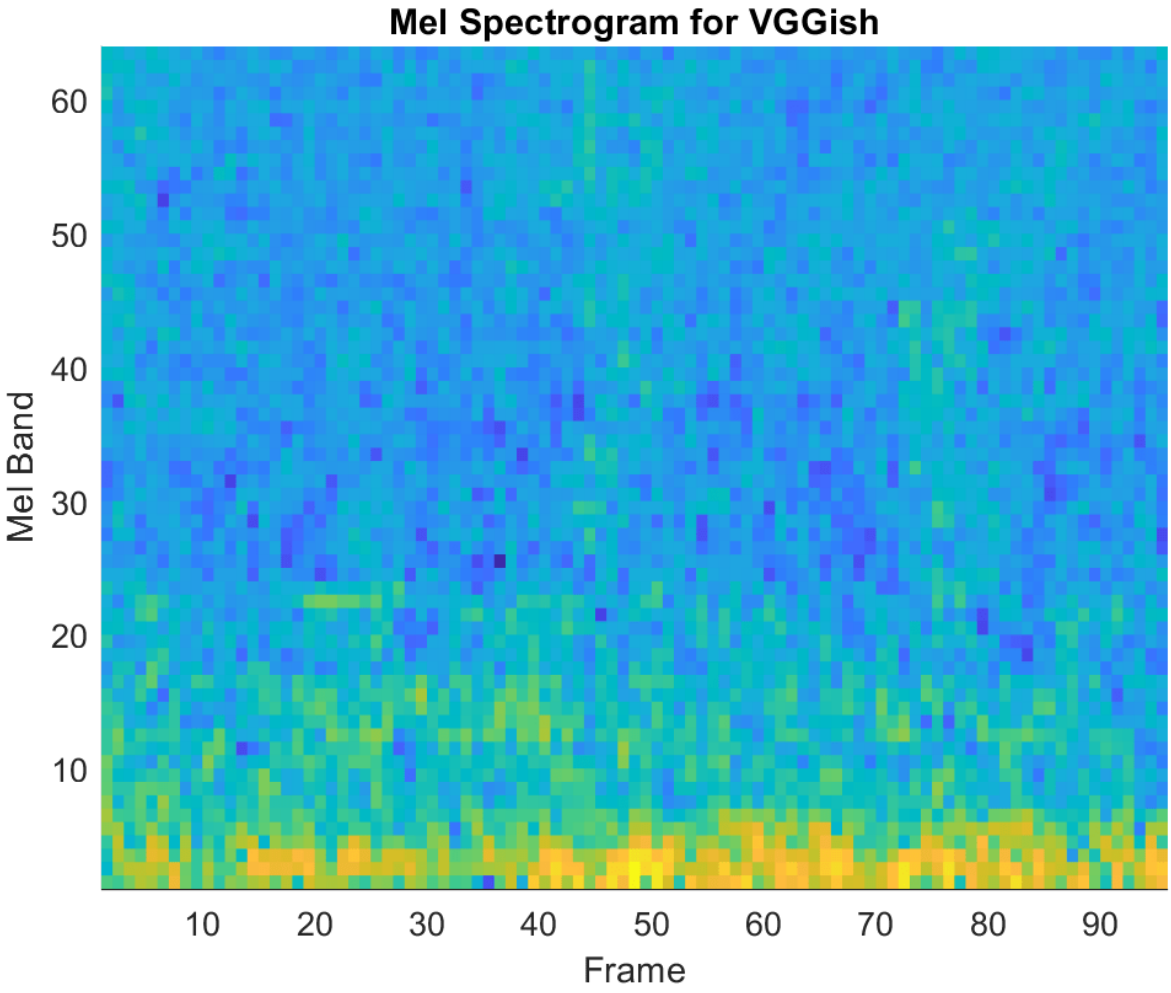
```
t = (0:numel(audioIn)-1)/fs;
plot(t,audioIn)
xlabel("Time (s)")
ylabel("Ampltiude")
axis tight
```




```
% To play the sound, call soundsc(audioIn,fs)
```

VGGish requires you to preprocess the audio signal to match the input format used to train the network. The preprocesssing steps include resampling the audio signal and computing an array of mel spectrograms. To learn more about mel spectrograms, see `melSpectrogram`. Use vggish`Preprocess` to preprocess the signal and extract the mel spectrograms to be passed to VGGish. Visualize one of these spectrograms chosen at random.

```
spectrograms = vggishPreprocess(audioIn,fs);

arbitrarySpect = spectrograms(:,:,1,randi(size(spectrograms,4)));
surf(arbitrarySpect,EdgeColor="none")
view(90,-90)
xlabel("Mel Band")
ylabel("Frame")
title("Mel Spectrogram for VGGish")
axis tight
```



Create a VGGish neural network. Using the `vggish` function requires installing the pretrained VGGish network. If the network is not installed, the function provides a link to download the pretrained model.

```
net = vggish;
```

Call `predict` with the network on the preprocessed mel spectrogram images to extract feature embeddings. The feature embeddings are returned as a `numFrames`-by-128 matrix, where `numFrames` is the number of individual spectrograms and 128 is the number of elements in each feature vector.

```
features = predict(net,spectrograms);
[numFrames,numFeatures] = size(features)
```

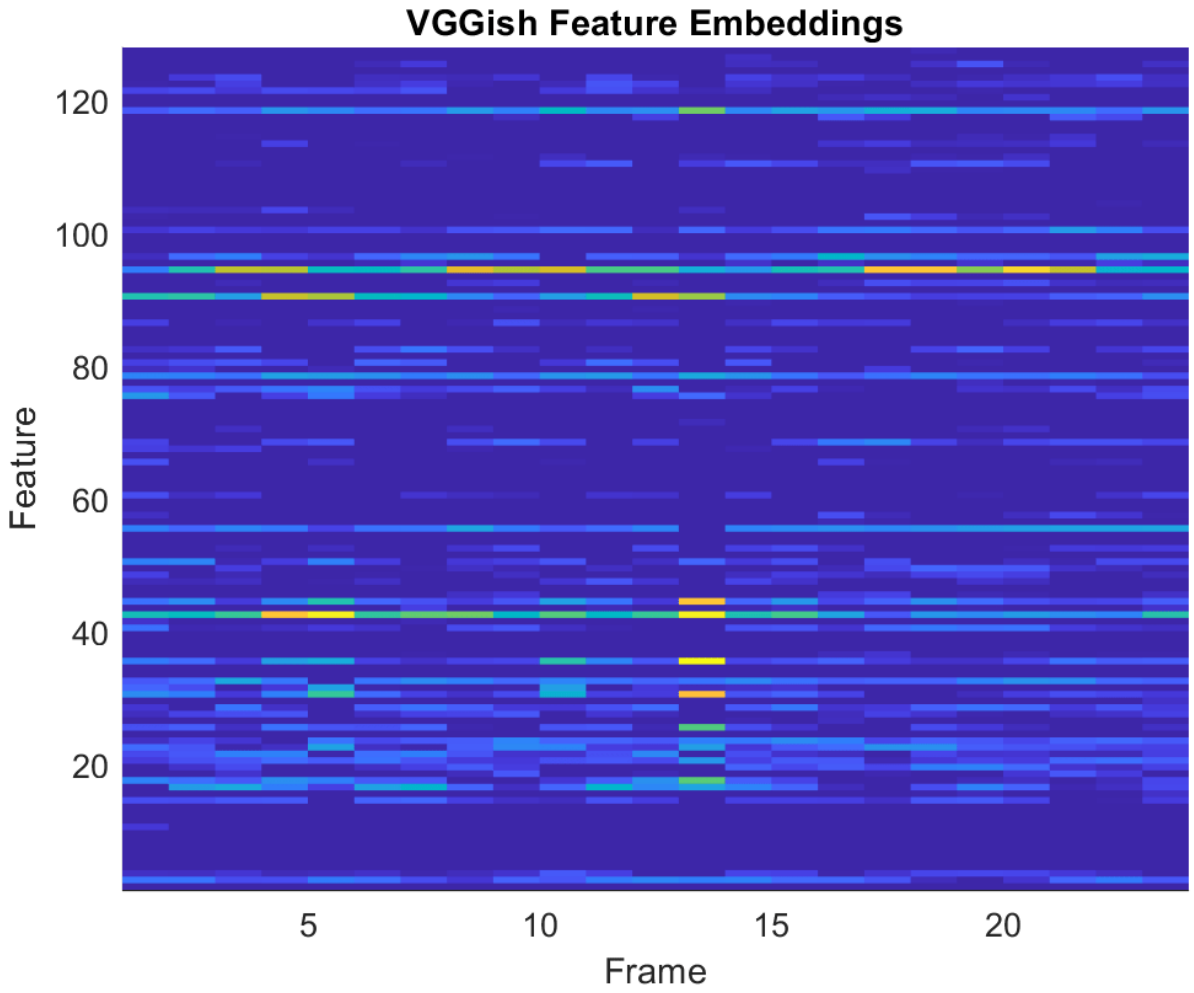
```
numFrames = 24
```

```
numFeatures = 128
```

Visualize the VGGish feature embeddings.

```
surf(features,EdgeColor="none")
view([90 -90])
```

```
xlabel("Feature")
ylabel("Frame")
title("VGGish Feature Embeddings")
axis tight
```



Transfer Learning Using VGGish

In this example, you transfer the learning in the VGGish regression model to an audio classification task.

Download and unzip the environmental sound classification data set. This data set consists of recordings labeled as one of 10 different audio sound classes (ESC-10).

```
downloadFolder = matlab.internal.examples.downloadSupportFile("audio","ESC-10.zip");
unzip(downloadFolder,tempdir)
dataLocation = fullfile(tempdir,"ESC-10");
```

Create an `audioDatastore` object to manage the data and split it into train and validation sets. Call `countEachLabel` to display the distribution of sound classes and the number of unique labels.

```
ads = audioDatastore(dataLocation,IncludeSubfolders=true,LabelSource="foldernames");
labelTable = countEachLabel(ads)
```

```
labelTable=10×2 table
        Label         Count
    ______________    _____

    chainsaw           40
    clock_tick         40
    crackling_fire     40
    crying_baby        40
    dog                40
    helicopter         40
    rain               40
    rooster            38
    sea_waves          40
    sneezing           40
```

Determine the total number of classes.

```
numClasses = height(labelTable);
```

Call `splitEachLabel` to split the data set into train and validation sets. Inspect the distribution of labels in the training and validation sets.

```
[adsTrain, adsValidation] = splitEachLabel(ads,0.8);

countEachLabel(adsTrain)
```

```
ans=10×2 table
        Label         Count
    ______________    _____

    chainsaw           32
    clock_tick         32
    crackling_fire     32
    crying_baby        32
    dog                32
    helicopter         32
    rain               32
    rooster            30
    sea_waves          32
    sneezing           32
```

```
countEachLabel(adsValidation)
```

```
ans=10×2 table
        Label         Count
    ______________    _____

    chainsaw            8
    clock_tick          8
    crackling_fire      8
    crying_baby         8
```

```
    dog                 8
    helicopter          8
    rain                8
    rooster             8
    sea_waves           8
    sneezing            8
```

The VGGish network expects audio to be preprocessed into log mel spectrograms. Use `vggishPreprocess` to extract the spectrograms from the train set. There are multiple spectrograms for each audio signal. Replicate the labels so that they are in one-to-one correspondence with the spectrograms.

```
overlapPercentage = 75;

trainFeatures = [];
trainLabels = [];
while hasdata(adsTrain)
    [audioIn,fileInfo] = read(adsTrain);
    features = vggishPreprocess(audioIn,fileInfo.SampleRate,OverlapPercentage=overlapPercentage)
    numSpectrograms = size(features,4);
    trainFeatures = cat(4,trainFeatures,features);
    trainLabels = cat(2,trainLabels,repelem(fileInfo.Label,numSpectrograms));
end
```

Extract spectrograms from the validation set and replicate the labels.

```
validationFeatures = [];
validationLabels = [];
segmentsPerFile = zeros(numel(adsValidation.Files), 1);
idx = 1;
while hasdata(adsValidation)
    [audioIn,fileInfo] = read(adsValidation);
    features = vggishPreprocess(audioIn,fileInfo.SampleRate,OverlapPercentage=overlapPercentage)
    numSpectrograms = size(features,4);
    validationFeatures = cat(4,validationFeatures,features);
    validationLabels = cat(2,validationLabels,repelem(fileInfo.Label,numSpectrograms));

    segmentsPerFile(idx) = numSpectrograms;
    idx = idx + 1;
end
```

Load the VGGish model and convert it to a `layerGraph` (Deep Learning Toolbox) object.

```
net = vggish;

lgraph = layerGraph(net.Layers);
```

Use `removeLayers` (Deep Learning Toolbox) to remove the final regression output layer from the graph. After you remove the regression layer, the new final layer of the graph is a ReLU layer named `'EmbeddingBatch'`.

```
lgraph = removeLayers(lgraph,"regressionoutput");
lgraph.Layers(end)

ans =
  ReLULayer with properties:
```

```
    Name: 'EmbeddingBatch'
```

Use `addLayers` (Deep Learning Toolbox) to add a `fullyConnectedLayer` (Deep Learning Toolbox), a `softmaxLayer` (Deep Learning Toolbox), and a `classificationLayer` (Deep Learning Toolbox) to the graph. Set the `WeightLearnRateFactor` and `BiasLearnRateFactor` of the new fully connected layer to 10 so that learning is faster in the new layer than in the transferred layers.

```
lgraph = addLayers(lgraph,[ ...
    fullyConnectedLayer(numClasses,Name="FCFinal",WeightLearnRateFactor=10,BiasLearnRateFactor=10
    softmaxLayer(Name="softmax")
    classificationLayer(Name="classOut")]);
```

Use `connectLayers` (Deep Learning Toolbox) to append the fully connected, softmax, and classification layers to the layer graph.

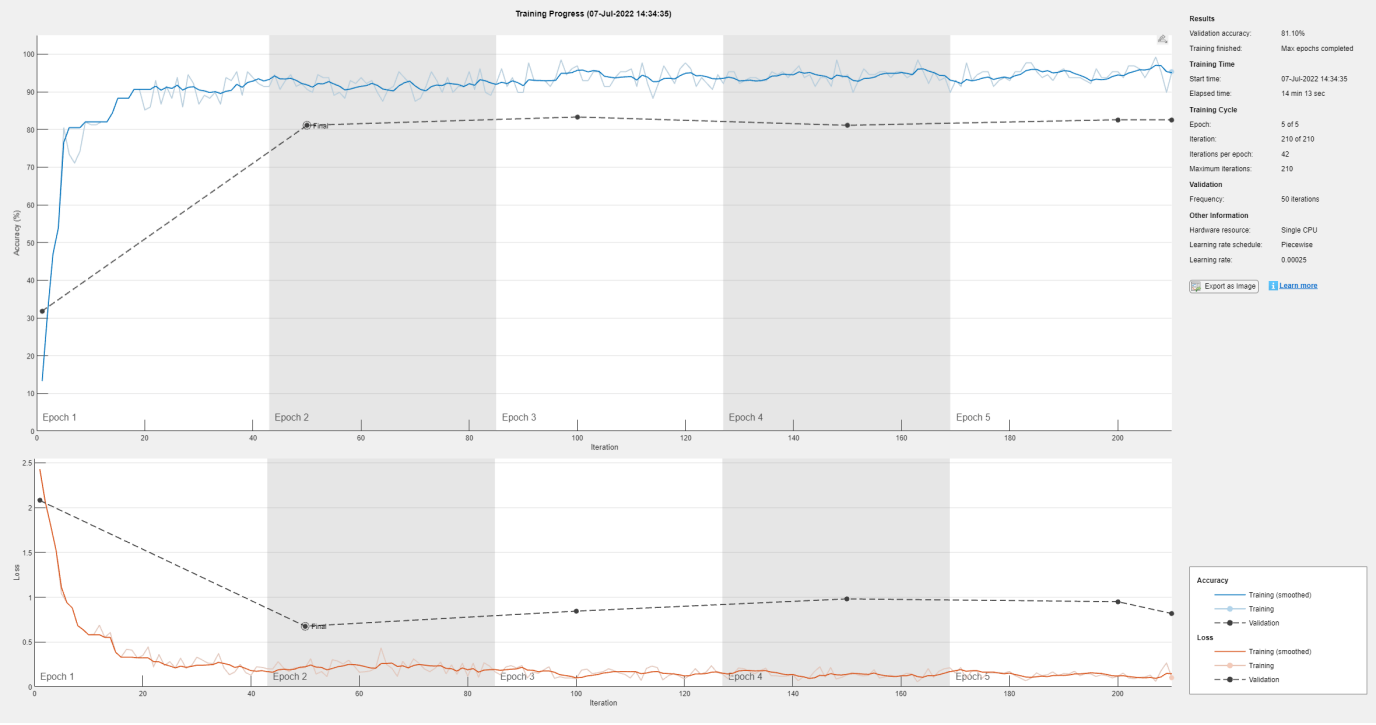
```
lgraph = connectLayers(lgraph,"EmbeddingBatch","FCFinal");
```

To define training options, use `trainingOptions` (Deep Learning Toolbox).

```
miniBatchSize = 128;
options = trainingOptions("adam", ...
    MaxEpochs=5, ...
    MiniBatchSize=miniBatchSize, ...
    Shuffle="every-epoch", ...
    ValidationData={validationFeatures,validationLabels}, ...
    ValidationFrequency=50, ...
    LearnRateSchedule="piecewise", ...
    LearnRateDropFactor=0.5, ...
    LearnRateDropPeriod=2, ...
    OutputNetwork="best-validation-loss", ...
    Verbose=false, ...
    Plots="training-progress");
```

To train the network, use `trainNetwork` (Deep Learning Toolbox).

```
[trainedNet, netInfo] = trainNetwork(trainFeatures,trainLabels,lgraph,options);
```

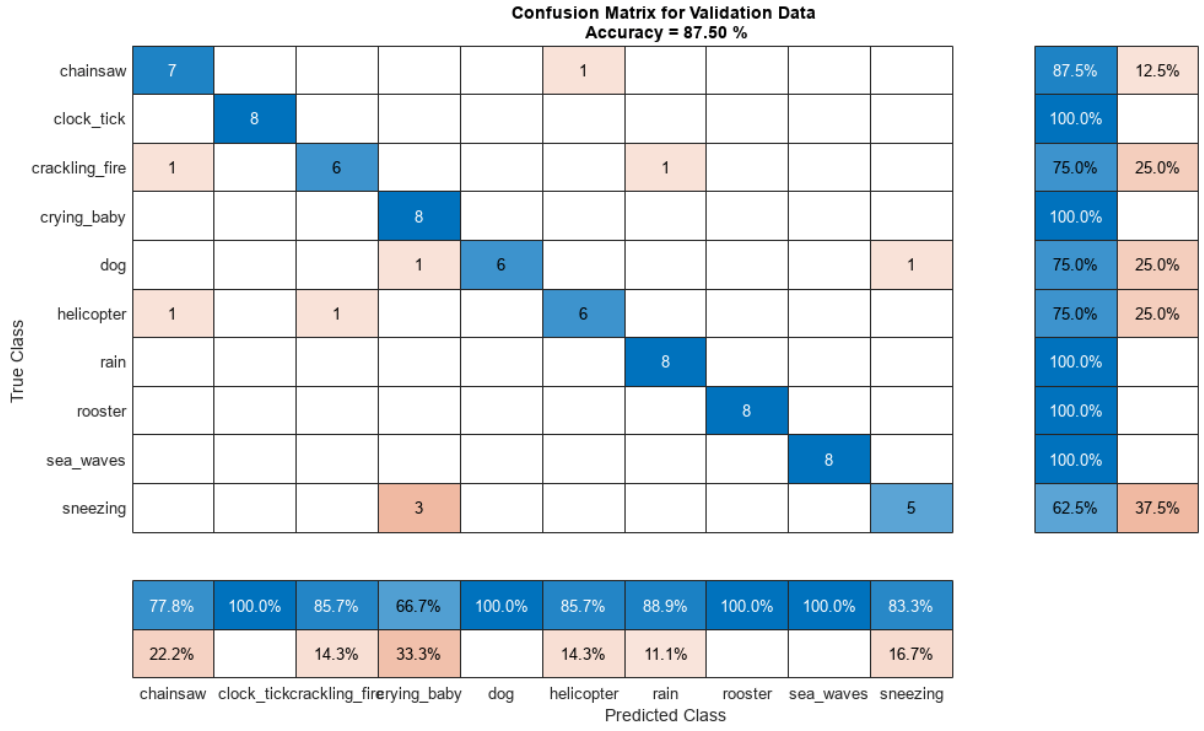


Each audio file was split into several segments to feed into the VGGish network. Combine the predictions for each file in the validation set using a majority-rule decision.

```
validationPredictions = classify(trainedNet,validationFeatures);

idx = 1;
validationPredictionsPerFile = categorical;
for ii = 1:numel(adsValidation.Files)
    validationPredictionsPerFile(ii,1) = mode(validationPredictions(idx:idx+segmentsPerFile(ii)-1
    idx = idx + segmentsPerFile(ii);
end
```

Use `confusionchart` (Deep Learning Toolbox) to evaluate the performance of the network on the validation set.

```
figure(Units="normalized",Position=[0.2 0.2 0.5 0.5]);
confusionchart(adsValidation.Labels,validationPredictionsPerFile, ...
    Title=sprintf("Confusion Matrix for Validation Data \nAccuracy = %0.2f %%",mean(validationPre
    ColumnSummary="column-normalized", ...
    RowSummary="row-normalized")
```



Output Arguments

net — Pretrained VGGish neural network

`SeriesNetwork` object

Pretrained VGGish neural network, returned as a `SeriesNetwork` object.

Version History

Introduced in R2020b

References

[1] Gemmeke, Jort F., Daniel P. W. Ellis, Dylan Freedman, Aren Jansen, Wade Lawrence, R. Channing Moore, Manoj Plakal, and Marvin Ritter. 2017. “Audio Set: An Ontology and Human-Labeled Dataset for Audio Events.” In *2017 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 776–80. New Orleans, LA: IEEE. https://doi.org/10.1109/ICASSP.2017.7952261.

[2] Hershey, Shawn, Sourish Chaudhuri, Daniel P. W. Ellis, Jort F. Gemmeke, Aren Jansen, R. Channing Moore, Manoj Plakal, et al. 2017. “CNN Architectures for Large-Scale Audio Classification.” In *2017 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 131–35. New Orleans, LA: IEEE. https://doi.org/10.1109/ICASSP.2017.7952132.

Extended Capabilities

C/C++ Code Generation

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

Usage notes and limitations:

- Only the `activations` and `predict` object functions are supported.
- To create a `SeriesNetwork` object for code generation, see “Load Pretrained Networks for Code Generation” (MATLAB Coder).

GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

Usage notes and limitations:

- Only the `activations`, `classify`, `predict`, `predictAndUpdateState`, and `resetState` object functions are supported.
- To create a `SeriesNetwork` object for code generation, see “Load Pretrained Networks for Code Generation” (GPU Coder).

See Also

Apps

Signal Labeler

Blocks

Sound Classifier | VGGish Embeddings | VGGish Preprocess | VGGish | YAMNet | YAMNet Preprocess

Functions

`audioFeatureExtractor` | `classifySound` | `melSpectrogram` | `vggishEmbeddings` | `vggishPreprocess` | `yamnet` | `yamnetGraph` | `yamnetPreprocess`

yamnet

YAMNet neural network

Syntax

```
net = yamnet
```

Description

`net = yamnet` returns a pretrained YAMNet model.

This function requires both Audio Toolbox and Deep Learning Toolbox.

Examples

Download YAMNet

Download and unzip the Audio Toolbox™ model for YAMNet.

Type `yamnet` at the Command Window. If the Audio Toolbox model for YAMNet is not installed, then the function provides a link to the location of the network weights. To download the model, click the link. Unzip the file to a location on the MATLAB path.

Alternatively, execute the following commands to download and unzip the YAMNet model to your temporary directory.

```
downloadFolder = fullfile(tempdir,'YAMNetDownload');
loc = websave(downloadFolder,'https://ssd.mathworks.com/supportfiles/audio/yamnet.zip');
YAMNetLocation = tempdir;
unzip(loc,YAMNetLocation)
addpath(fullfile(YAMNetLocation,'yamnet'))
```

Check that the installation is successful by typing `yamnet` at the Command Window. If the network is installed, then the function returns a `SeriesNetwork` (Deep Learning Toolbox) object.

```
yamnet
```

```
ans =
  SeriesNetwork with properties:

         Layers: [86×1 nnet.cnn.layer.Layer]
     InputNames: {'input_1'}
    OutputNames: {'Sound'}
```

Load Pretrained YAMNet

Load a pretrained YAMNet convolutional neural network and examine the layers and classes.

Use `yamnet` to load the pretrained YAMNet network. The output net is a `SeriesNetwork` (Deep Learning Toolbox) object.

```
net = yamnet
```

```
net =
  SeriesNetwork with properties:

         Layers: [86×1 nnet.cnn.layer.Layer]
     InputNames: {'input_1'}
    OutputNames: {'Sound'}
```

View the network architecture using the `Layers` property. The network has 86 layers. There are 28 layers with learnable weights: 27 convolutional layers, and 1 fully connected layer.

```
net.Layers
```

```
ans =
  86x1 Layer array with layers:

     1   'input_1'                 Image Input             96×64×1 images
     2   'conv2d'                  Convolution             32 3×3×1 convolutions with stride
     3   'b'                       Batch Normalization     Batch normalization with 32 channe
     4   'activation'              ReLU                    ReLU
     5   'depthwise_conv2d'        Grouped Convolution     32 groups of 1 3×3×1 convolutions
     6   'L11'                     Batch Normalization     Batch normalization with 32 channe
     7   'activation_1'            ReLU                    ReLU
     8   'conv2d_1'                Convolution             64 1×1×32 convolutions with stride
     9   'L12'                     Batch Normalization     Batch normalization with 64 channe
    10   'activation_2'            ReLU                    ReLU
    11   'depthwise_conv2d_1'      Grouped Convolution     64 groups of 1 3×3×1 convolutions
    12   'L21'                     Batch Normalization     Batch normalization with 64 channe
    13   'activation_3'            ReLU                    ReLU
    14   'conv2d_2'                Convolution             128 1×1×64 convolutions with strid
    15   'L22'                     Batch Normalization     Batch normalization with 128 chann
    16   'activation_4'            ReLU                    ReLU
    17   'depthwise_conv2d_2'      Grouped Convolution     128 groups of 1 3×3×1 convolutions
    18   'L31'                     Batch Normalization     Batch normalization with 128 chann
    19   'activation_5'            ReLU                    ReLU
    20   'conv2d_3'                Convolution             128 1×1×128 convolutions with stri
    21   'L32'                     Batch Normalization     Batch normalization with 128 chann
    22   'activation_6'            ReLU                    ReLU
    23   'depthwise_conv2d_3'      Grouped Convolution     128 groups of 1 3×3×1 convolutions
    24   'L41'                     Batch Normalization     Batch normalization with 128 chann
    25   'activation_7'            ReLU                    ReLU
    26   'conv2d_4'                Convolution             256 1×1×128 convolutions with stri
    27   'L42'                     Batch Normalization     Batch normalization with 256 chann
    28   'activation_8'            ReLU                    ReLU
    29   'depthwise_conv2d_4'      Grouped Convolution     256 groups of 1 3×3×1 convolutions
    30   'L51'                     Batch Normalization     Batch normalization with 256 chann
    31   'activation_9'            ReLU                    ReLU
    32   'conv2d_5'                Convolution             256 1×1×256 convolutions with stri
    33   'L52'                     Batch Normalization     Batch normalization with 256 chann
    34   'activation_10'           ReLU                    ReLU
    35   'depthwise_conv2d_5'      Grouped Convolution     256 groups of 1 3×3×1 convolutions
    36   'L61'                     Batch Normalization     Batch normalization with 256 chann
    37   'activation_11'           ReLU                    ReLU
```

```
    38   'conv2d_6'                  Convolution              512 1×1×256 convolutions with stri
    39   'L62'                       Batch Normalization      Batch normalization with 512 chann
    40   'activation_12'             ReLU                     ReLU
    41   'depthwise_conv2d_6'        Grouped Convolution      512 groups of 1 3×3×1 convolutions
    42   'L71'                       Batch Normalization      Batch normalization with 512 chann
    43   'activation_13'             ReLU                     ReLU
    44   'conv2d_7'                  Convolution              512 1×1×512 convolutions with stri
    45   'L72'                       Batch Normalization      Batch normalization with 512 chann
    46   'activation_14'             ReLU                     ReLU
    47   'depthwise_conv2d_7'        Grouped Convolution      512 groups of 1 3×3×1 convolutions
    48   'L81'                       Batch Normalization      Batch normalization with 512 chann
    49   'activation_15'             ReLU                     ReLU
    50   'conv2d_8'                  Convolution              512 1×1×512 convolutions with stri
    51   'L82'                       Batch Normalization      Batch normalization with 512 chann
    52   'activation_16'             ReLU                     ReLU
    53   'depthwise_conv2d_8'        Grouped Convolution      512 groups of 1 3×3×1 convolutions
    54   'L91'                       Batch Normalization      Batch normalization with 512 chann
    55   'activation_17'             ReLU                     ReLU
    56   'conv2d_9'                  Convolution              512 1×1×512 convolutions with stri
    57   'L92'                       Batch Normalization      Batch normalization with 512 chann
    58   'activation_18'             ReLU                     ReLU
    59   'depthwise_conv2d_9'        Grouped Convolution      512 groups of 1 3×3×1 convolutions
    60   'L101'                      Batch Normalization      Batch normalization with 512 chann
    61   'activation_19'             ReLU                     ReLU
    62   'conv2d_10'                 Convolution              512 1×1×512 convolutions with stri
    63   'L102'                      Batch Normalization      Batch normalization with 512 chann
    64   'activation_20'             ReLU                     ReLU
    65   'depthwise_conv2d_10'       Grouped Convolution      512 groups of 1 3×3×1 convolutions
    66   'L111'                      Batch Normalization      Batch normalization with 512 chann
    67   'activation_21'             ReLU                     ReLU
    68   'conv2d_11'                 Convolution              512 1×1×512 convolutions with stri
    69   'L112'                      Batch Normalization      Batch normalization with 512 chann
    70   'activation_22'             ReLU                     ReLU
    71   'depthwise_conv2d_11'       Grouped Convolution      512 groups of 1 3×3×1 convolutions
    72   'L121'                      Batch Normalization      Batch normalization with 512 chann
    73   'activation_23'             ReLU                     ReLU
    74   'conv2d_12'                 Convolution              1024 1×1×512 convolutions with str
    75   'L122'                      Batch Normalization      Batch normalization with 1024 chan
    76   'activation_24'             ReLU                     ReLU
    77   'depthwise_conv2d_12'       Grouped Convolution      1024 groups of 1 3×3×1 convolution
    78   'L131'                      Batch Normalization      Batch normalization with 1024 chan
    79   'activation_25'             ReLU                     ReLU
    80   'conv2d_13'                 Convolution              1024 1×1×1024 convolutions with st
    81   'L132'                      Batch Normalization      Batch normalization with 1024 chan
    82   'activation_26'             ReLU                     ReLU
    83   'global_average_pooling2d'  Global Average Pooling   Global average pooling
    84   'dense'                     Fully Connected          521 fully connected layer
    85   'softmax'                   Softmax                  softmax
    86   'Sound'                     Classification Output    crossentropyex with 'Speech' and 5
```

To view the names of the classes learned by the network, you can view the `Classes` property of the classification output layer (the final layer). View the first 10 classes by specifying the first 10 elements.

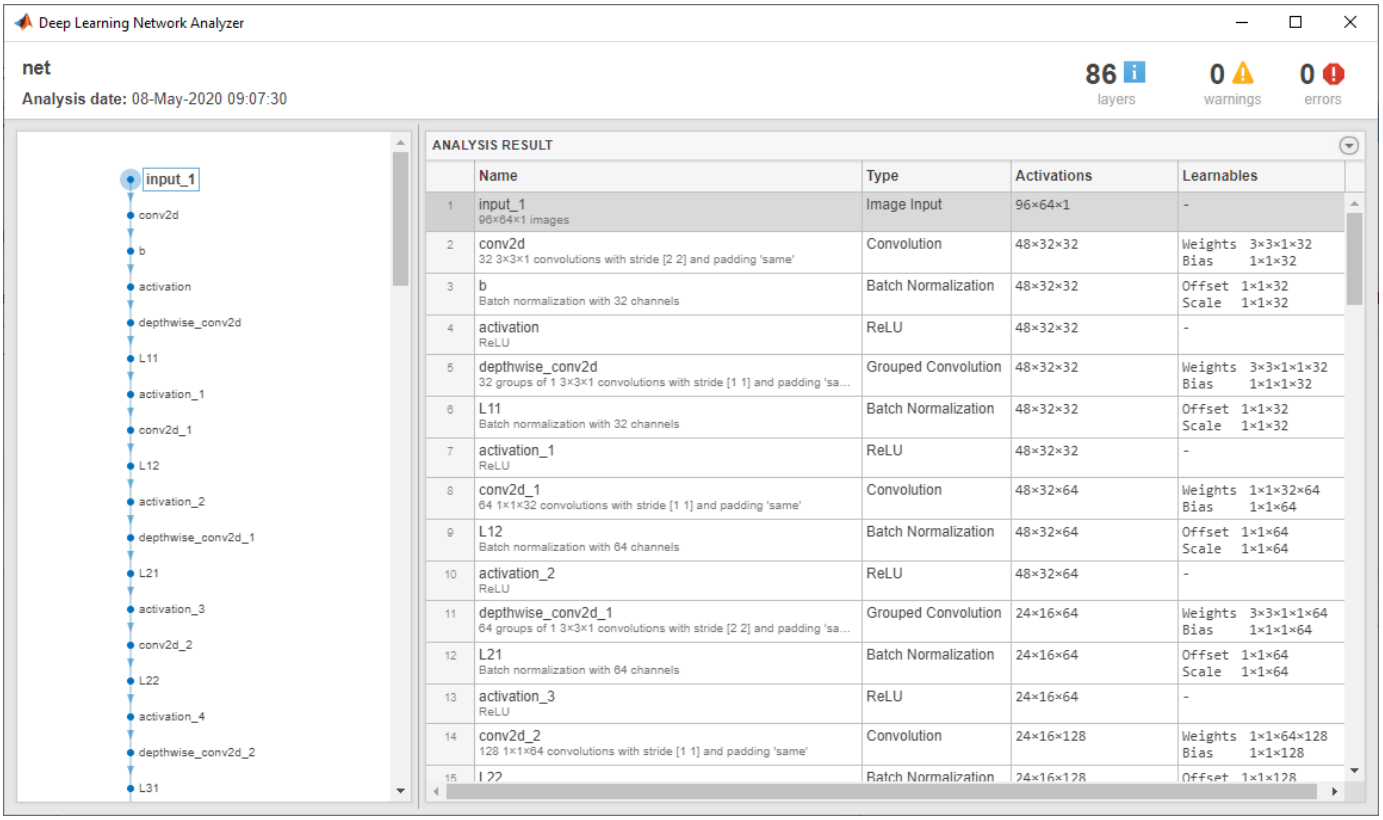
```
net.Layers(end).Classes(1:10)
```

```
ans = 10×1 categorical
     Speech
```

```
Child speech, kid speaking
Conversation
Narration, monologue
Babbling
Speech synthesizer
Shout
Bellow
Whoop
Yell
```

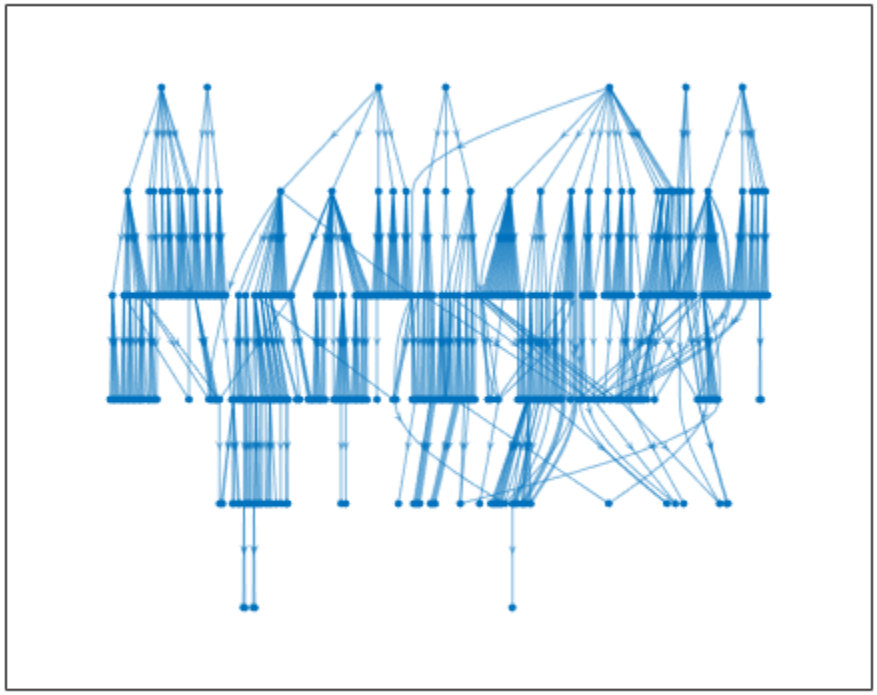
Use `analyzeNetwork` (Deep Learning Toolbox) to visually explore the network.

```
analyzeNetwork(net)
```



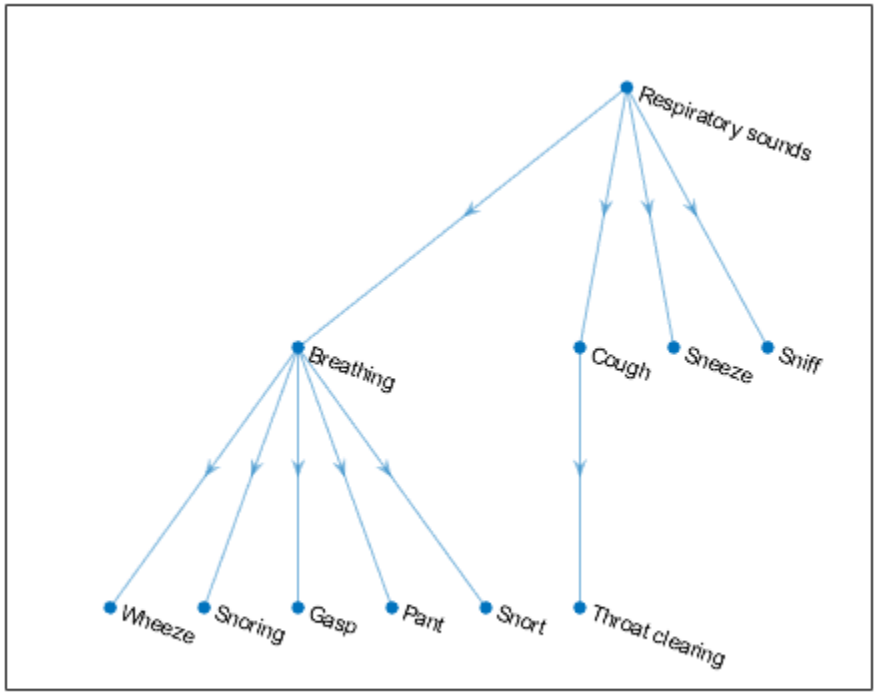
YAMNet was released with a corresponding sound class ontology, which you can explore using the `yamnetGraph` object.

```
ygraph = yamnetGraph;
p = plot(ygraph);
layout(p,'layered')
```



The ontology graph plots all 521 possible sound classes. Plot a subgraph of the sounds related to respiratory sounds.

```
allRespiratorySounds = dfsearch(ygraph,"Respiratory sounds");
ygraphSpeech = subgraph(ygraph,allRespiratorySounds);
plot(ygraphSpeech)
```



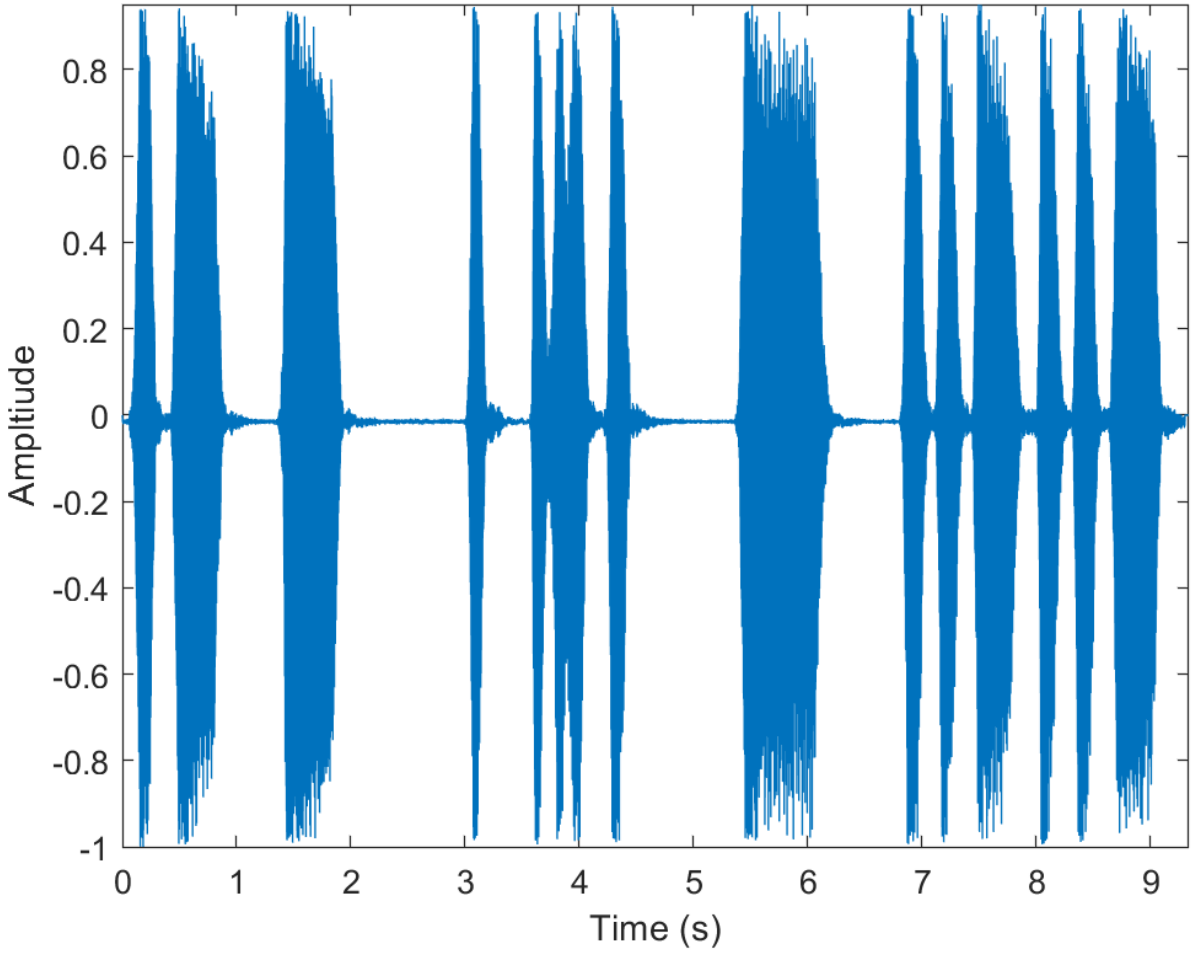
Classify Sounds Using YAMNet

Read in an audio signal to classify it.

```
[audioIn,fs] = audioread( TrainWhistle-16-44... );
```

Plot and listen to the audio signal.

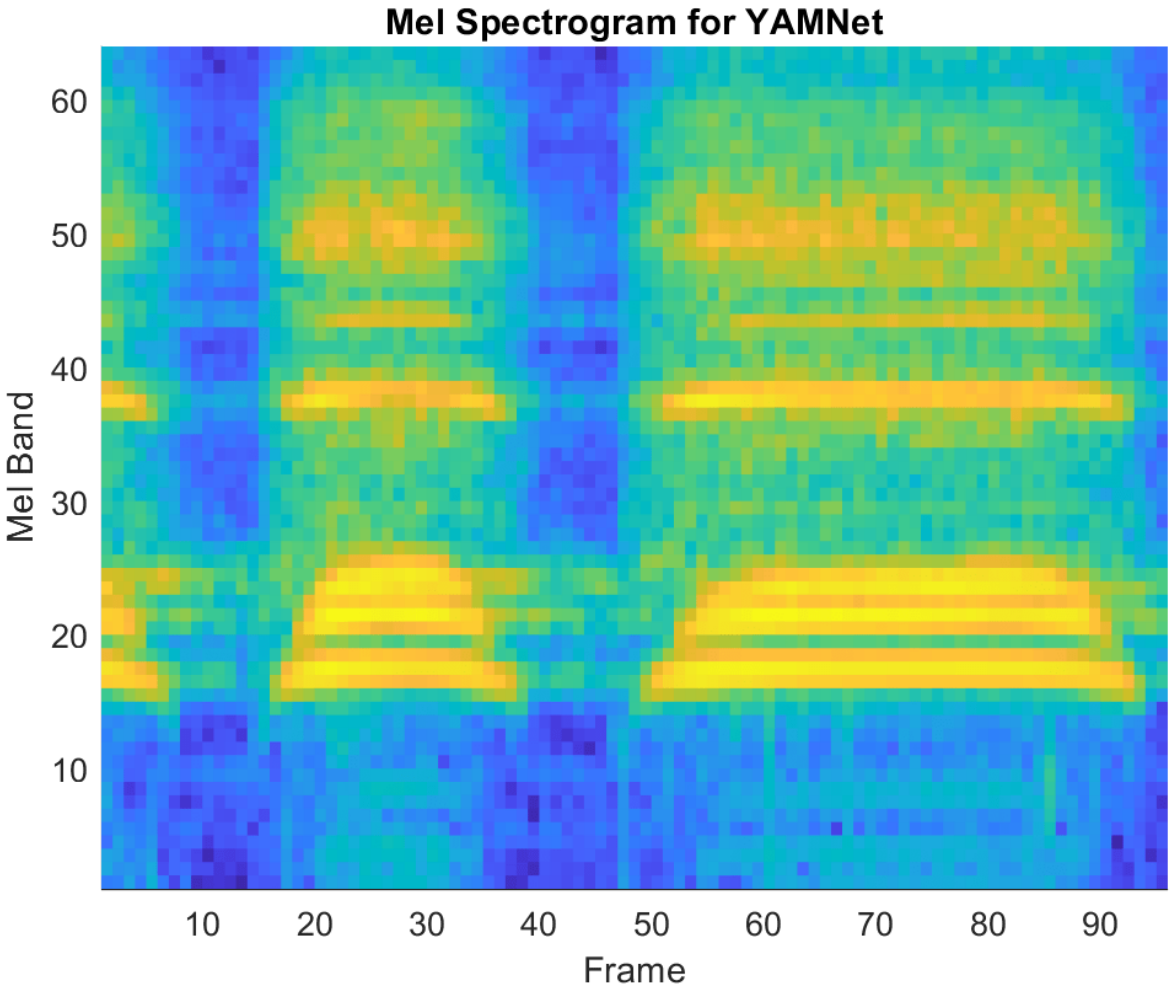
```
t = (0:numel(audioIn)-1)/fs;
plot(t,audioIn)
xlabel("Time (s)")
ylabel("Ampltiude")
axis tight
```




```
% To play the sound, call soundsc(audioIn,fs)
```

YAMNet requires you to preprocess the audio signal to match the input format used to train the network. The preprocesssing steps include resampling the audio signal and computing an array of mel spectrograms. To learn more about mel spectrograms, see `melSpectrogram`. Use `yamnetPreprocess` to preprocess the signal and extract the mel spectrograms to be passed to YAMNet. Visualize one of these spectrograms chosen at random.

```
spectrograms = yamnetPreprocess(audioIn,fs);

arbitrarySpect = spectrograms(:,:,1,randi(size(spectrograms,4)));
surf(arbitrarySpect,EdgeColor="none")
view([90 -90])
xlabel("Mel Band")
ylabel("Frame")
title("Mel Spectrogram for YAMNet")
axis tight
```



Create a YAMNet neural network. Using the `yamnet` function requires installing the pretrained YAMNet network. If the network is not installed, the function provides a link to download the pretrained model. Call `classify` with the network on the preprocessed mel spectrogram images.

```
net = yamnet;
classes = classify(net,spectrograms);
```

Calling `classify` returns a label for each of the spectrogram images in the input. Classify the sound as the most frequently occurring label in the output of `classify`.

```
mySound = mode(classes)
```

```
mySound = categorical
     Whistle
```

Transfer Learning Using YAMNet

Download and unzip the air compressor data set [1] on page 2-125. This data set consists of recordings from air compressors in a healthy state or one of 7 faulty states.

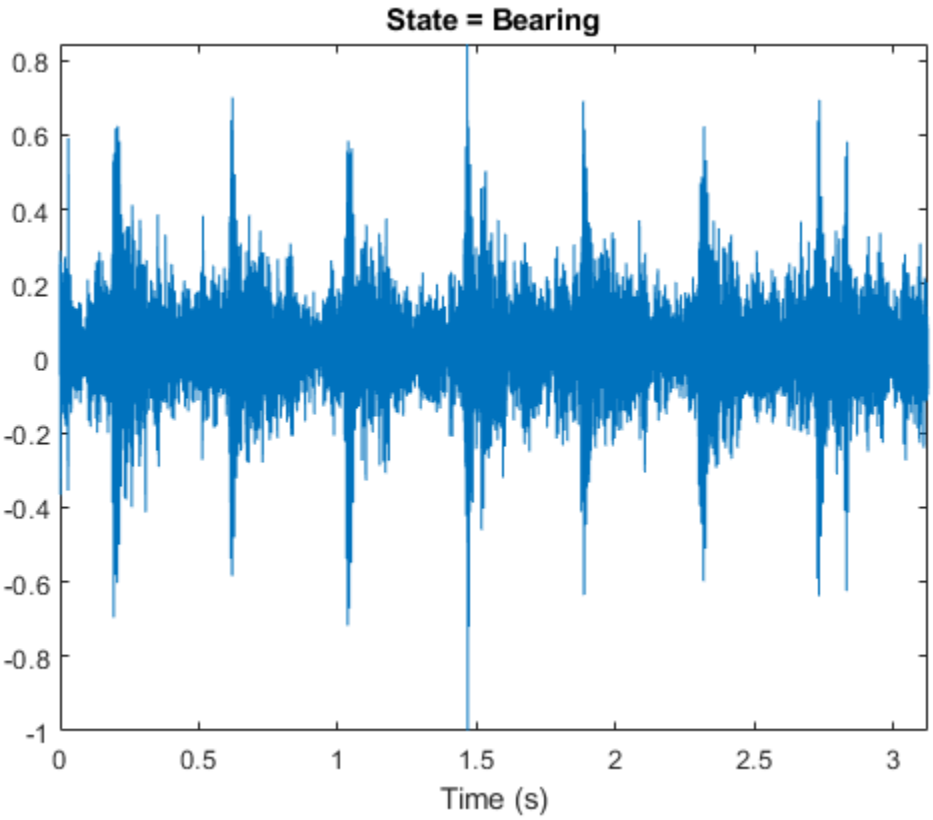
```
url = 'https://www.mathworks.com/supportfiles/audio/AirCompressorDataset/AirCompressorDataset.zip
downloadFolder = fullfile(tempdir,'aircompressordataset');
datasetLocation = tempdir;

if ~exist(fullfile(tempdir,'AirCompressorDataSet'),'dir')
    loc = websave(downloadFolder,url);
    unzip(loc,fullfile(tempdir,'AirCompressorDataSet'))
end
```

Create an `audioDatastore` object to manage the data and split it into train and validation sets.

```
ads = audioDatastore(downloadFolder,'IncludeSubfolders',true,'LabelSource','foldernames');

[adsTrain,adsValidation] = splitEachLabel(ads,0.8,0.2);
```

Read an audio file from the datastore and save the sample rate for later use. Reset the datastore to return the read pointer to the beginning of the data set. Listen to the audio signal and plot the signal in the time domain.

```
[x,fileInfo] = read(adsTrain);
fs = fileInfo.SampleRate;

reset(adsTrain)

sound(x,fs)

figure
t = (0:size(x,1)-1)/fs;
plot(t,x)
xlabel('Time (s)')
title('State = ' + string(fileInfo.Label))
axis tight
```



Extract Mel spectrograms from the train set using `yamnetPreprocess`. There are multiple spectrograms for each audio signal. Replicate the labels so that they are in one-to-one correspondence with the spectrograms.

```
emptyLabelVector = adsTrain.Labels;
emptyLabelVector(:) = [];

trainFeatures = [];
trainLabels = emptyLabelVector;
while hasdata(adsTrain)
    [audioIn,fileInfo] = read(adsTrain);
    features = yamnetPreprocess(audioIn,fileInfo.SampleRate);
    numSpectrums = size(features,4);
    trainFeatures = cat(4,trainFeatures,features);
    trainLabels = cat(2,trainLabels,repmat(fileInfo.Label,1,numSpectrums));
end
```

Extract features from the validation set and replicate the labels.

```
validationFeatures = [];
validationLabels = emptyLabelVector;
while hasdata(adsValidation)
    [audioIn,fileInfo] = read(adsValidation);
    features = yamnetPreprocess(audioIn,fileInfo.SampleRate);
    numSpectrums = size(features,4);
    validationFeatures = cat(4,validationFeatures,features);
    validationLabels = cat(2,validationLabels,repmat(fileInfo.Label,1,numSpectrums));
end
```

The air compressor data set has only eight classes.

Read in YAMNet and convert it to a `layerGraph` (Deep Learning Toolbox).

If YAMNet pretrained network is not installed on your machine, execute the following commands to download and unzip the YAMNet model to your temporary directory.

```
downloadFolder = fullfile(tempdir,'YAMNetDownload');
loc = websave(downloadFolder,'https://ssd.mathworks.com/supportfiles/audio/yamnet.zip');
YAMNetLocation = tempdir;
unzip(loc,YAMNetLocation)
addpath(fullfile(YAMNetLocation,'yamnet'))
```

After you read in YAMNet and convert it to a `layerGraph` (Deep Learning Toolbox), replace the final `fullyConnectedLayer` (Deep Learning Toolbox) and the final `classificationLayer` (Deep Learning Toolbox) to reflect the new task.

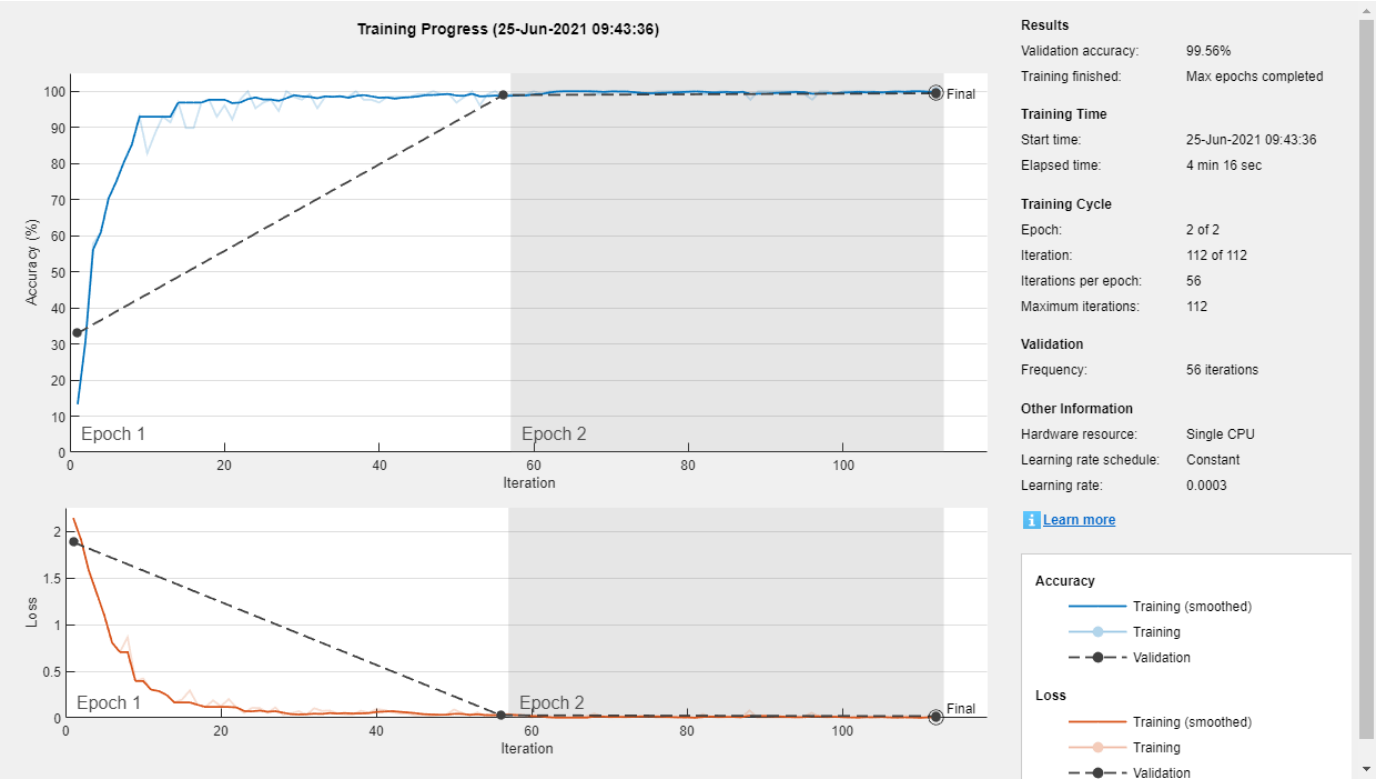
```
uniqueLabels = unique(adsTrain.Labels);
numLabels = numel(uniqueLabels);

net = yamnet;

lgraph = layerGraph(net.Layers);

newDenseLayer = fullyConnectedLayer(numLabels,"Name","dense");
lgraph = replaceLayer(lgraph,"dense",newDenseLayer);

newClassificationLayer = classificationLayer("Name","Sounds","Classes",uniqueLabels);
lgraph = replaceLayer(lgraph,"Sound",newClassificationLayer);
```

To define training options, use `trainingOptions` (Deep Learning Toolbox).

```
miniBatchSize = 128;
validationFrequency = floor(numel(trainLabels)/miniBatchSize);
options = trainingOptions('adam', ...
    'InitialLearnRate',3e-4, ...
    'MaxEpochs',2, ...
    'MiniBatchSize',miniBatchSize, ...
    'Shuffle','every-epoch', ...
    'Plots','training-progress', ...
    'Verbose',false, ...
    'ValidationData',{single(validationFeatures),validationLabels}, ...
    'ValidationFrequency',validationFrequency);
```

To train the network, use `trainNetwork` (Deep Learning Toolbox).

```
airCompressorNet = trainNetwork(trainFeatures,trainLabels,lgraph,options);
```



Save the trained network to `airCompressorNet.mat`. You can now use this pre-trained network by loading the `airCompressorNet.mat` file.

```
save airCompressorNet.mat airCompressorNet
```

References

[1] Verma, Nishchal K., et al. “Intelligent Condition Based Monitoring Using Acoustic Signals for Air Compressors.” *IEEE Transactions on Reliability*, vol. 65, no. 1, Mar. 2016, pp. 291–309. *DOI.org (Crossref)*, doi:10.1109/TR.2015.2459684.

Output Arguments

net — Pretrained YAMNet neural network

`SeriesNetwork` object

Pretrained YAMNet neural network, returned as a `SeriesNetwork` object.

Version History

Introduced in R2020b

References

[1] Gemmeke, Jort F., et al. “Audio Set: An Ontology and Human-Labeled Dataset for Audio Events.” *2017 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, IEEE, 2017, pp. 776–80. *DOI.org (Crossref)*, doi:10.1109/ICASSP.2017.7952261.

[2] Hershey, Shawn, et al. “CNN Architectures for Large-Scale Audio Classification.” *2017 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, IEEE, 2017, pp. 131–35. *DOI.org (Crossref)*, doi:10.1109/ICASSP.2017.7952132.

Extended Capabilities

C/C++ Code Generation

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

Usage notes and limitations:

- Only the `activations` and `predict` object functions are supported.
- To create a `SeriesNetwork` object for code generation, see “Load Pretrained Networks for Code Generation” (MATLAB Coder).

GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

Usage notes and limitations:

- Only the `activations`, `classify`, `predict`, `predictAndUpdateState`, and `resetState` object functions are supported.
- To create a `SeriesNetwork` object for code generation, see “Load Pretrained Networks for Code Generation” (GPU Coder).

See Also

Apps

Signal Labeler

Blocks

Sound Classifier | VGGish Embeddings | VGGish Preprocess | VGGish | YAMNet | YAMNet Preprocess

Functions

`audioFeatureExtractor` | `classifySound` | `designAuditoryFilterBank` | `melSpectrogram` | `vggish` | `vggishPreprocess` | `yamnetGraph` | `yamnetPreprocess`

vggishEmbeddings

Extract VGGish feature embeddings

Syntax

```
embeddings = vggishEmbeddings(audioIn,fs)
embeddings = vggishEmbeddings(audioIn,fs,Name=Value)
```

Description

`embeddings = vggishEmbeddings(audioIn,fs)` returns VGGish feature embeddings over time for the audio input `audioIn` with sample rate `fs`. Columns of the input are treated as individual channels.

`embeddings = vggishEmbeddings(audioIn,fs,Name=Value)` specifies options using one or more name-value arguments. For example, `embeddings = vggishEmbeddings(audioIn,fs,ApplyPCA=true)` applies a principal component analysis (PCA) transformation to the audio embeddings.

This function requires both Audio Toolbox and Deep Learning Toolbox.

Examples

Download vggishEmbeddings Functionality

Download and unzip the Audio Toolbox™ model for VGGish.

Type `vggishEmbeddings` at the command line. If the Audio Toolbox model for VGGish is not installed, then the function provides a link to the location of the network weights. To download the model, click the link. Unzip the file to a location on the MATLAB path.

Alternatively, execute the following commands to download and unzip the VGGish model to your temporary directory.

```
downloadFolder = fullfile(tempdir,"VGGishDownload");
loc = websave(downloadFolder,"https://ssd.mathworks.com/supportfiles/audio/vggish.zip");
VGGishLocation = tempdir;
unzip(loc,VGGishLocation)
addpath(fullfile(VGGishLocation,"vggish"))
```

Extract VGGish Embeddings

Read in an audio file.

```
[audioIn,fs] = audioread("MainStreetOne-16-16-mono-12secs.wav");
```

Call the `vggishEmbeddings` function with the audio and sample rate to extract VGGish feature embeddings from the audio. Using the `vggishEmbeddings` function requires installing the

pretrained VGGish network. If the network is not installed, the function provides a link to download the pretrained model.

```
embeddings = vggishEmbeddings(audioIn,fs);
```

The `vggishEmbeddings` function returns a matrix of 128-element feature vectors over time.

```
[numHops,numElementsPerHop,numChannels] = size(embeddings)
```

```
numHops = 23
```

```
numElementsPerHop = 128
```

```
numChannels = 1
```

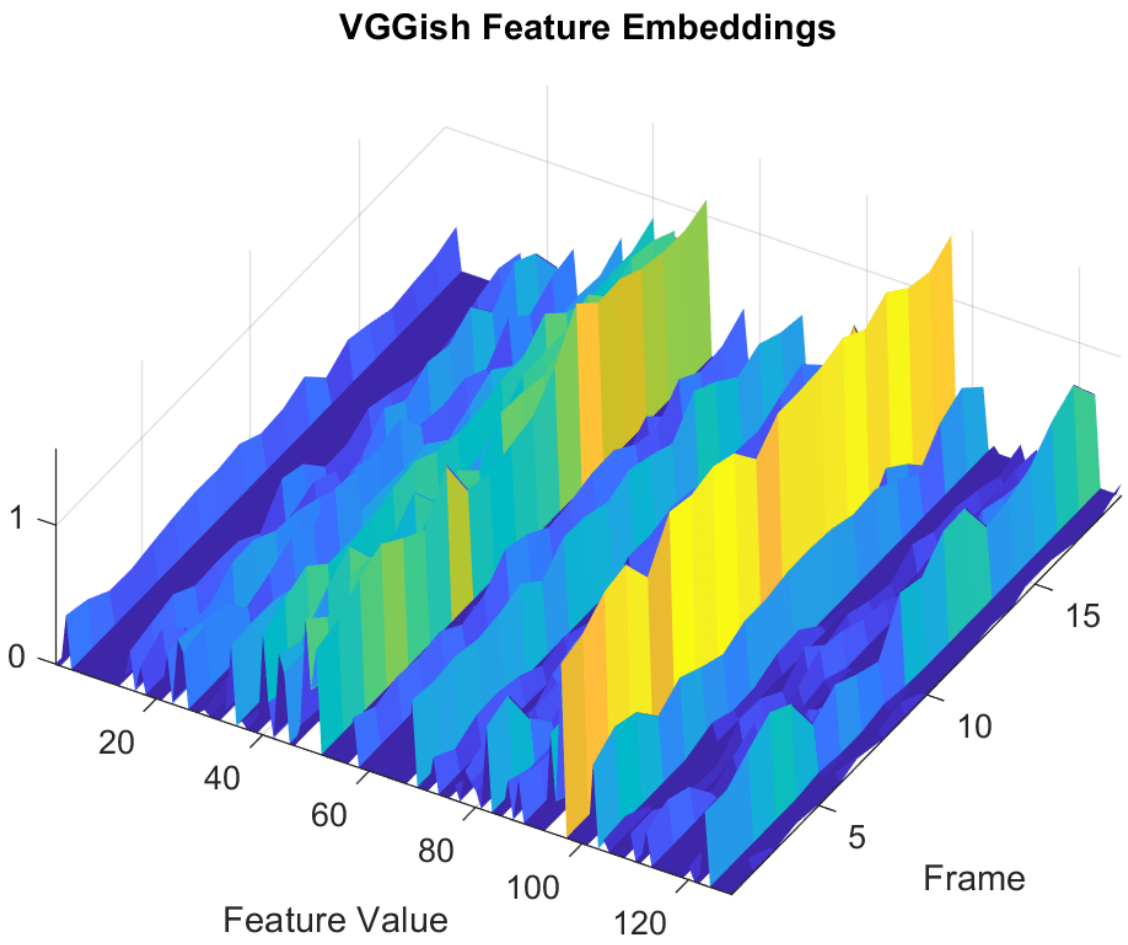
Increase Time Resolution of VGGish Embeddings

Create a 10-second pink noise signal and then extract VGGish embeddings. The `vggishEmbeddings` function extracts feature embeddings from mel spectrograms with 50% overlap. Using the `vggishEmbeddings` function requires installing the pretrained VGGish network. If the network is not installed, the function provides a link to download the pretrained model.

```
fs = 16e3;
dur = 10;
audioIn = pinknoise(dur*fs,1,"single");
embeddings = vggishEmbeddings(audioIn,fs);
```

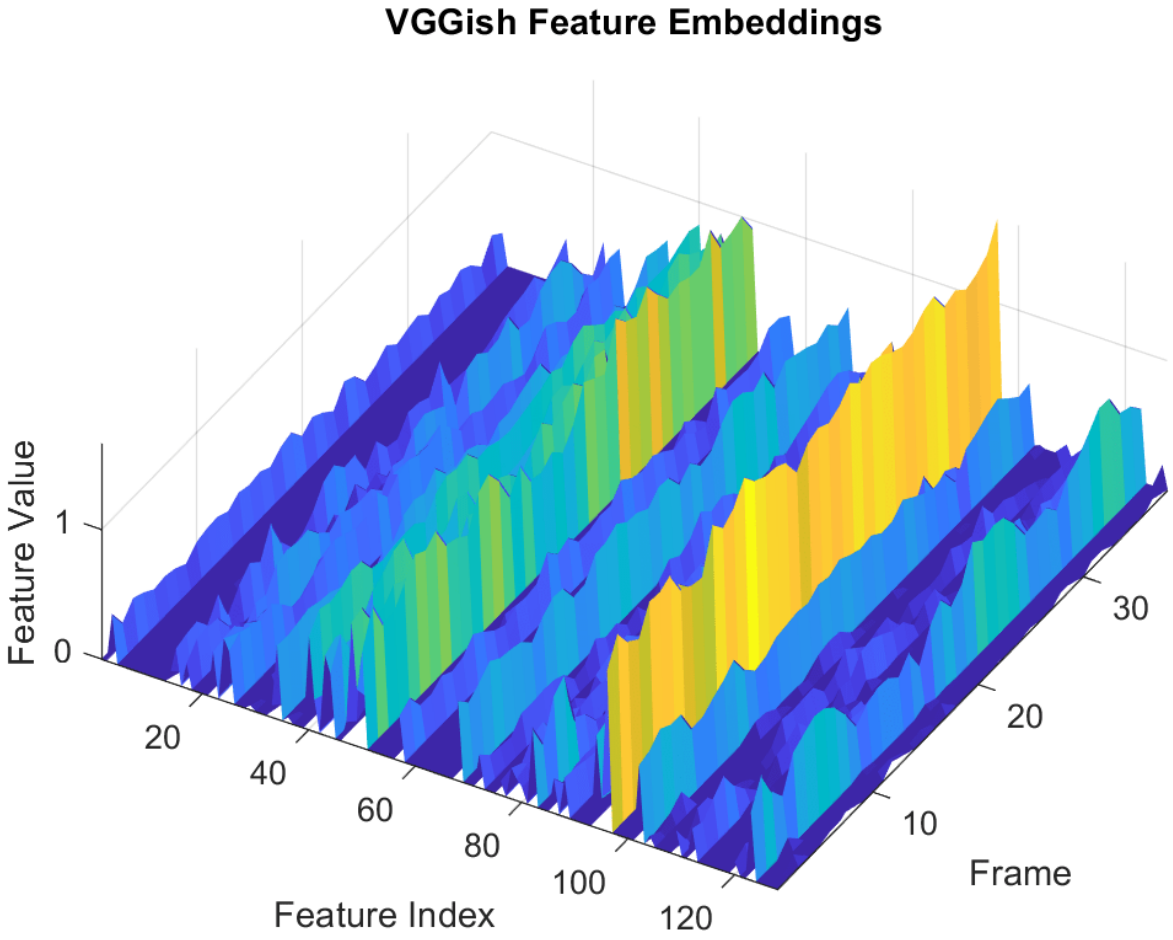
Plot the VGGish feature embeddings over time.

```
surf(embeddings,EdgeColor="none")
view([30 65])
axis tight
xlabel("Feature Index")
ylabel("Frame")
xlabel("Feature Value")
title("VGGish Feature Embeddings")
```



To increase the resolution of VGGish feature embeddings over time, specify the percent overlap between mel spectrograms. Plot the results.

```
overlapPercentage = 75;
embeddings = vggishEmbeddings(audioIn,fs,OverlapPercentage=overlapPercentage);

surf(embeddings,EdgeColor="none")
view([30 65])
axis tight
xlabel("Feature Index")
ylabel("Frame")
zlabel("Feature Value")
title("VGGish Feature Embeddings")
```



Apply Principal Component Analysis to VGGish Embeddings

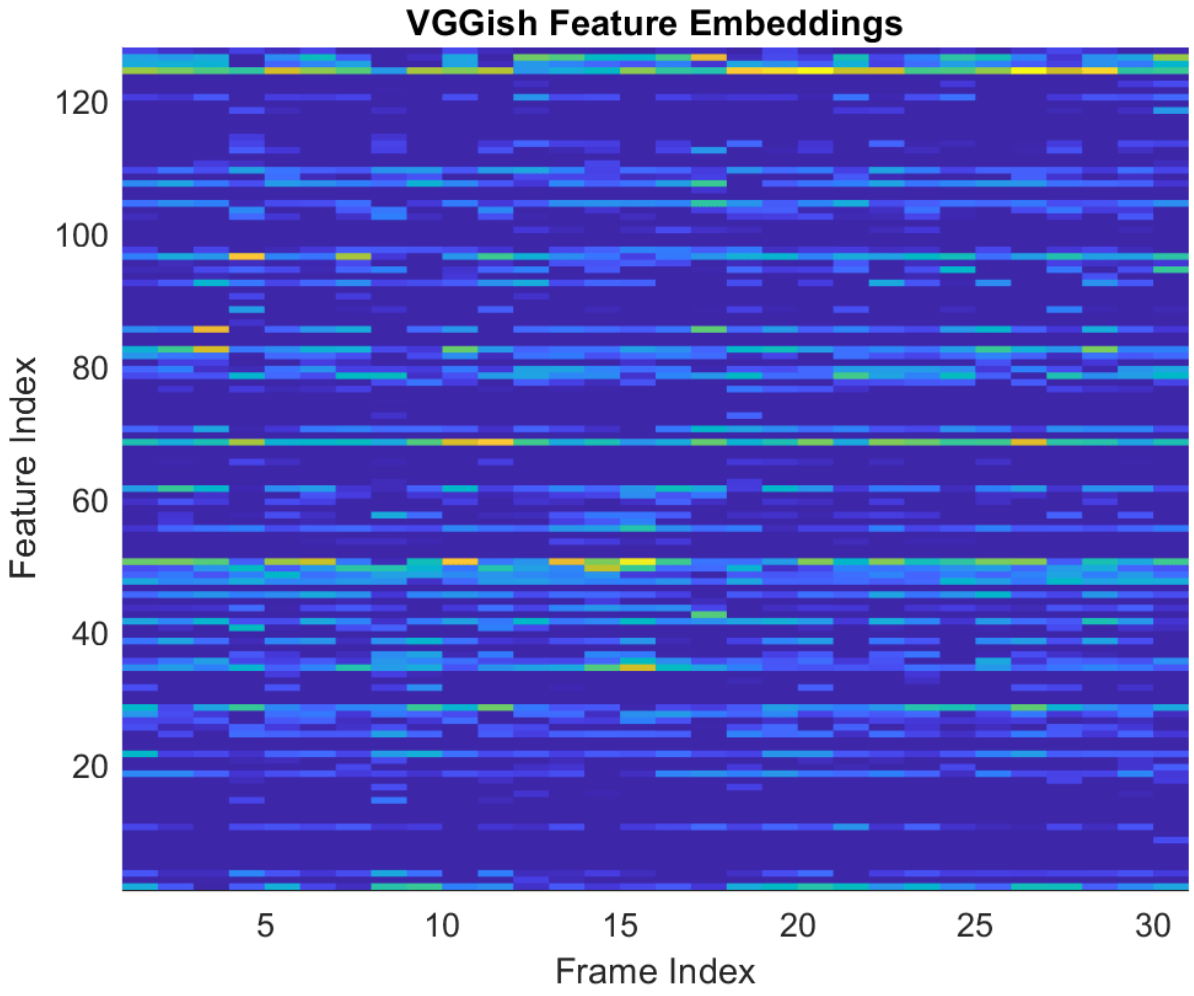
Read in an audio file, listen to it, and then extract VGGish feature embeddings from the audio. Using the `vggishEmbeddings` function requires installing the pretrained VGGish network. If the network is not installed, the function provides a link to download the pretrained model.

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

sound(audioIn,fs)

embeddings = vggishEmbeddings(audioIn,fs);
```

Visualize the VGGish feature embeddings over time. Many of the individual features are zero-valued and contain no useful information.

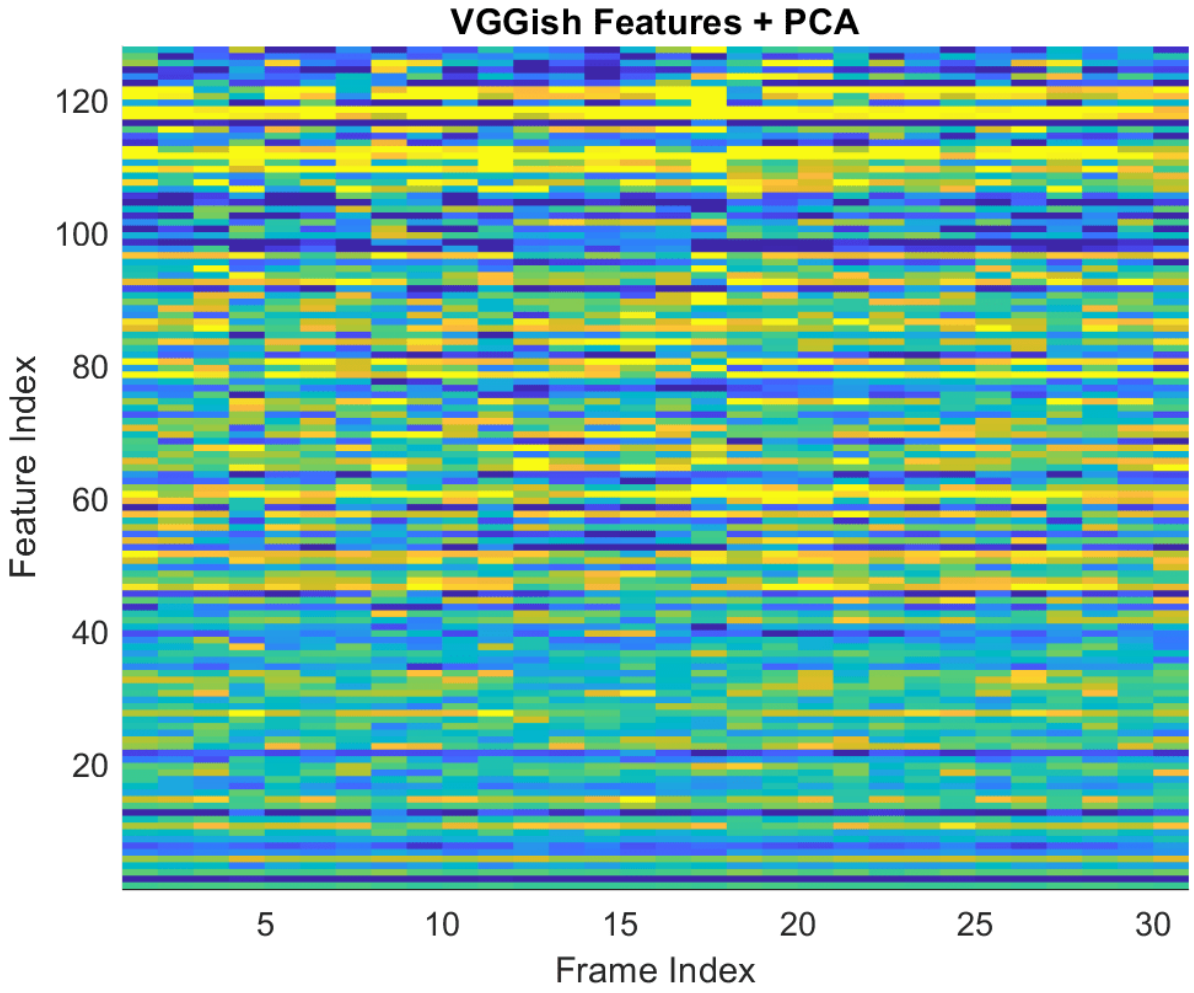
```
surf(embeddings,EdgeColor="none")
view([90,-90])
axis tight
```

```
xlabel("Feature Index")
ylabel("Frame Index")
title("VGGish Feature Embeddings")
```



You can apply principal component analysis (PCA) to map the feature vectors into a space that emphasizes variation between the embeddings. Call the `vggishEmbeddings` function again and specify `ApplyPCA` as `true`. Visualize the VGGish feature embeddings after PCA.

```
embeddings = vggishEmbeddings(audioIn,fs,ApplyPCA=true);

surf(embeddings,EdgeColor="none")
view([90,-90])
axis tight
xlabel("Feature Index")
ylabel("Frame Index")
title("VGGish Features + PCA")
```



Use VGGish Embeddings for Deep Learning

Download and unzip the air compressor data set. This data set consists of recordings from air compressors in a healthy state or in one of seven faulty states.

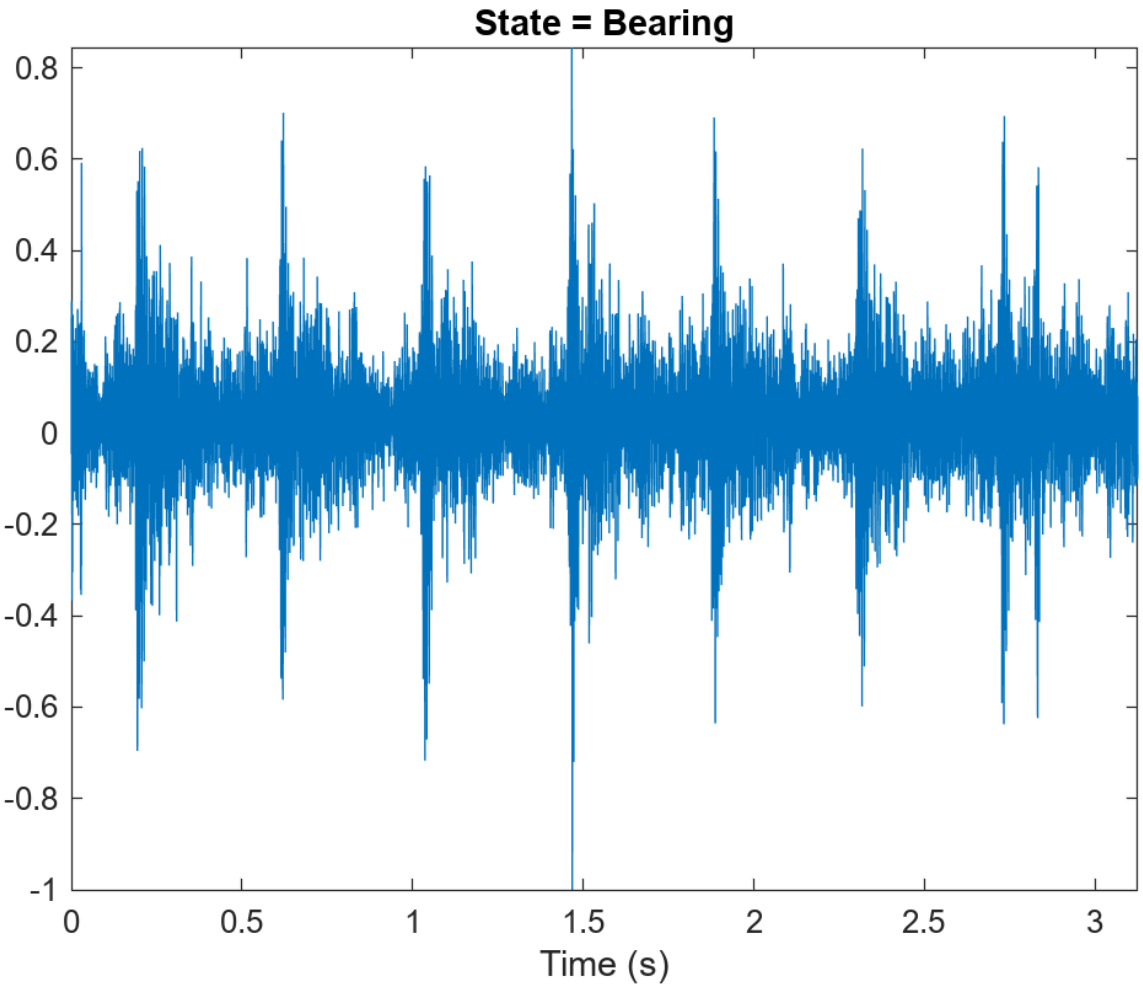
```
zipFile = matlab.internal.examples.downloadSupportFile("audio", ...
    "AirCompressorDataset/AirCompressorDataset.zip");
unzip(zipFile,tempdir)
dataLocation = fullfile(tempdir,"AirCompressorDataset");
```

Create an `audioDatastore` object to manage the data and split it into training and validation sets.

```
ads = audioDatastore(dataLocation,IncludeSubfolders=true, ...
    LabelSource="foldernames");

[adsTrain,adsValidation] = splitEachLabel(ads,0.8);
```

Read an audio file from the datastore. Reset the datastore to return the read pointer to the beginning of the data set. Listen to the audio signal and plot the signal in the time domain.

```
[x,fileInfo] = read(adsTrain);
fs = fileInfo.SampleRate;

reset(adsTrain)

sound(x,fs)

figure
t = (0:size(x,1)-1)/fs;
plot(t,x)
xlabel("Time (s)")
title("State = " + string(fileInfo.Label))
axis tight
```



Extract VGGish feature embeddings from the training and validation sets. Using the `vggishEmbeddings` function requires installing the pretrained VGGish network. If the network is not installed, the function provides a link to download the pretrained model. There are multiple embeddings vectors for each audio file. Replicate the labels so that they are in one-to-one correspondence with the embeddings vectors.

```
trainFeatures = [];
trainLabels = [];
while hasdata(adsTrain)
    [audioIn,fileInfo] = read(adsTrain);
    features = vggishEmbeddings(audioIn,fileInfo.SampleRate, ...
        OverlapPercentage=75);
    numFeatureVecs = size(features,1);
    trainFeatures = cat(1,trainFeatures,features);
    trainLabels = cat(1,trainLabels,repelem(fileInfo.Label,numFeatureVecs)');
end

validationFeatures = [];
validationLabels = [];
segmentsPerFile = zeros(numel(adsValidation.Files), 1);
idx = 1;
while hasdata(adsValidation)
    [audioIn,fileInfo] = read(adsValidation);
    features = vggishEmbeddings(audioIn,fileInfo.SampleRate, ...
        OverlapPercentage=75);
    numFeatureVecs = size(features,1);
    validationFeatures = cat(1,validationFeatures,features);
    validationLabels = cat(1,validationLabels, ...
        repelem(fileInfo.Label,numFeatureVecs)');

    segmentsPerFile(idx) = numFeatureVecs;
    idx = idx + 1;
end
```

Define a simple network with two fully connected layers.

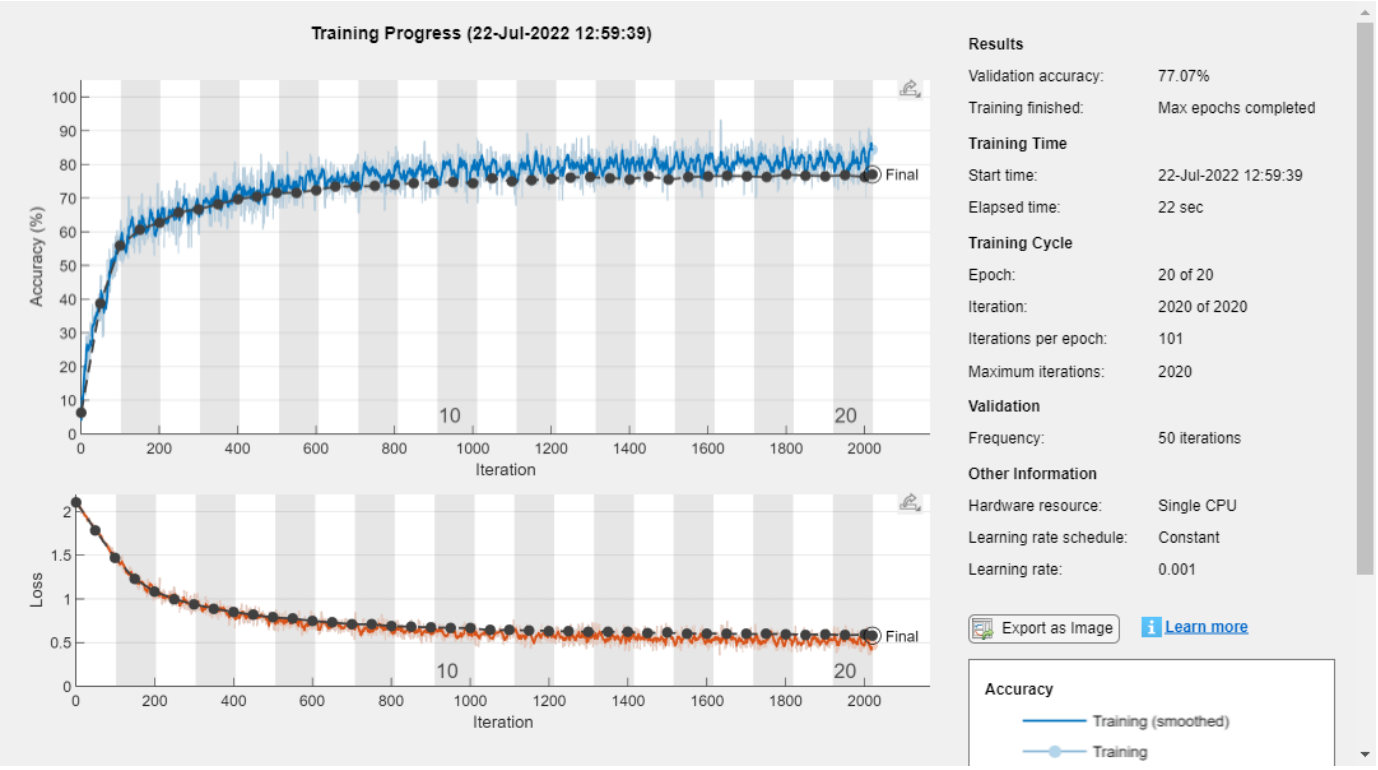
```
layers = [
    featureInputLayer(128)
    fullyConnectedLayer(32)
    reluLayer
    fullyConnectedLayer(8)
    softmaxLayer
    classificationLayer];
```

To define training options, use `trainingOptions` (Deep Learning Toolbox).

```
miniBatchSize = 128;
options = trainingOptions("adam", ...
    MaxEpochs=20, ...
    MiniBatchSize=miniBatchSize, ...
    Shuffle="every-epoch", ...
    ValidationData={validationFeatures,validationLabels}, ...
    ValidationFrequency=50, ...
    Plots="training-progress", ...
    Verbose=false);
```

To train the network, use `trainNetwork` (Deep Learning Toolbox).

```
net = trainNetwork(trainFeatures,trainLabels,layers,options)
```



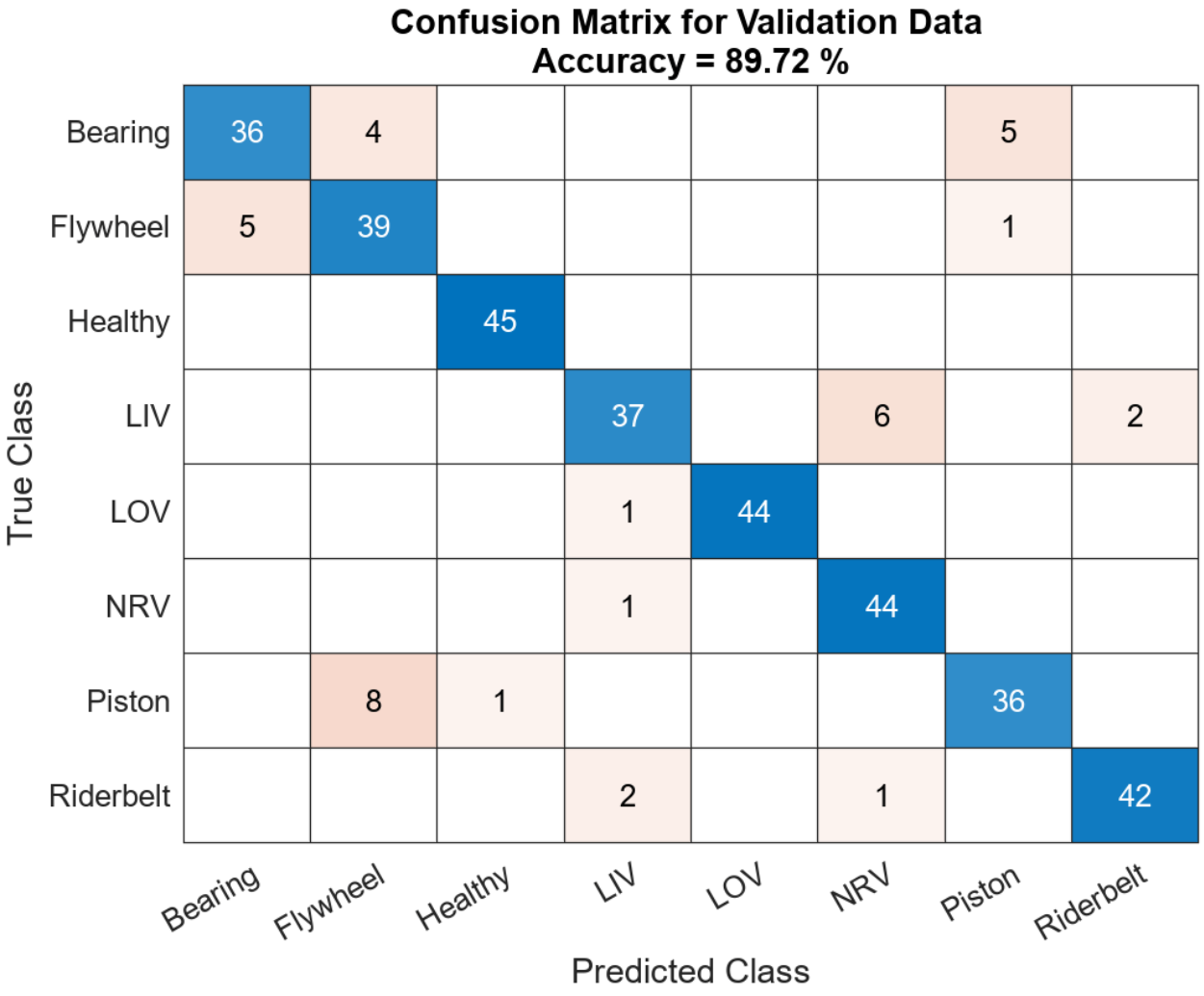

```
net =
  SeriesNetwork with properties:

         Layers: [6×1 nnet.cnn.layer.Layer]
     InputNames: {'input'}
    OutputNames: {'classoutput'}
```

Each audio file was split into several segments to feed into the network. Combine the predictions for each file in the validation set using a majority-rule decision.

```
validationPredictions = classify(net,validationFeatures);

idx = 1;
validationPredictionsPerFile = categorical;
for ii = 1:numel(adsValidation.Files)
    validationPredictionsPerFile(ii,1) = ...
        mode(validationPredictions(idx:idx+segmentsPerFile(ii)-1));
    idx = idx + segmentsPerFile(ii);
end
```

Visualize the confusion matrix for the validation set.

```
figure
confusionchart(adsValidation.Labels,validationPredictionsPerFile, ...
    Title=sprintf("Confusion Matrix for Validation Data \nAccuracy = %0.2f %%", ...
    mean(validationPredictionsPerFile==adsValidation.Labels)*100))
```



Use VGGish Embeddings for Machine Learning

Download and unzip the air compressor data set [1] on page 2-138. This data set consists of recordings from air compressors in a healthy state or in one of seven faulty states.

```
datasetZipFile = matlab.internal.examples.downloadSupportFile("audio","AirCompressorDataset/AirC
datasetFolder = fullfile(fileparts(datasetZipFile),"AirCompressorDataset");
if ~exist(datasetFolder,"dir")
    unzip(datasetZipFile,fileparts(datasetZipFile));
end
```

Create an `audioDatastore` object to manage the data and split it into training and validation sets.

```
ads = audioDatastore(datasetFolder,IncludeSubfolders=true,LabelSource="foldernames");
```

In this example, you classify signals as either healthy or faulty. Combine all of the faulty labels into a single label. Split the datastore into training and validation sets.

```
labels = ads.Labels;
labels(labels~=categorical("Healthy")) = categorical("Faulty");
ads.Labels = removecats(labels);

[adsTrain,adsValidation] = splitEachLabel(ads,0.8,0.2);
```

Extract VGGish feature embeddings from the training set. Each audio file corresponds to multiple VGGish features. Replicate the labels so that they are in one-to-one correspondence with the features. Using the `vggishEmbeddings` function requires installing the pretrained VGGish network. If the network is not installed, the function provides a link to download the pretrained model.

```
trainFeatures = [];
trainLabels = [];
for idx = 1:numel(adsTrain.Files)
    [audioIn,fileInfo] = read(adsTrain);
    embeddings = vggishEmbeddings(audioIn,fileInfo.SampleRate);
    trainFeatures = [trainFeatures;embeddings];
    trainLabels = [trainLabels;repelem(fileInfo.Label,size(embeddings,1))'];
end
```

Train a cubic support vector machine (SVM) using `fitcsvm` (Statistics and Machine Learning Toolbox). To explore other classifiers and their performances, use Classification Learner (Statistics and Machine Learning Toolbox).

```
faultDetector = fitcsvm( ...
    trainFeatures, ...
    trainLabels, ...
    KernelFunction="polynomial", ...
    PolynomialOrder=3, ...
    KernelScale="auto", ...
    BoxConstraint=1, ...
    Standardize=true, ...
    ClassNames=categories(trainLabels));
```

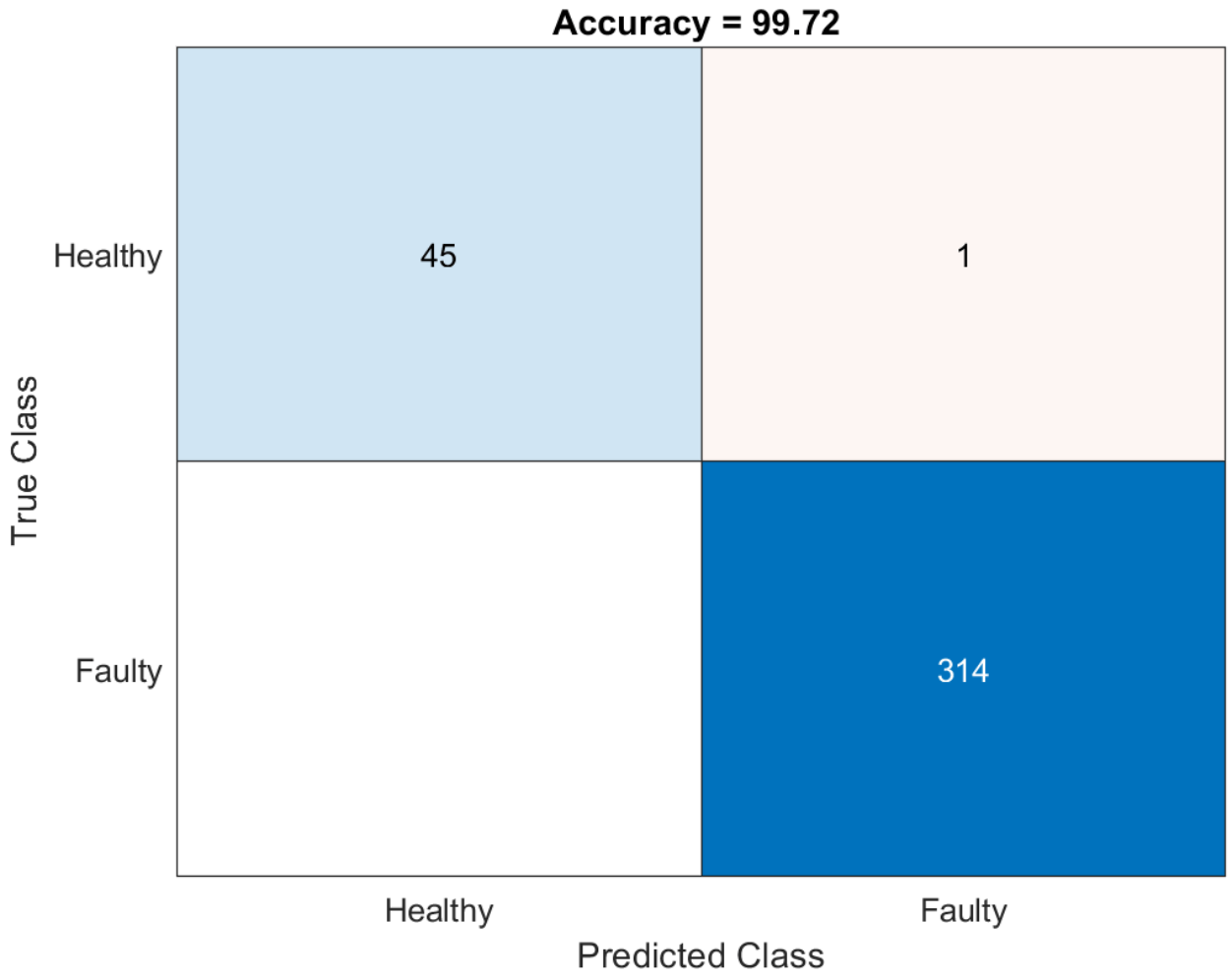
For each file in the validation set:

1. Extract VGGish feature embeddings.
2. For each VGGish feature vector in a file, use the trained classifier to predict whether the machine is healthy or faulty.
3. Take the mode of the predictions for each file.

```
predictions = [];
for idx = 1:numel(adsValidation.Files)
    [audioIn,fileInfo] = read(adsValidation);

    embeddings = vggishEmbeddings(audioIn,fileInfo.SampleRate);

    predictionsPerFile = categorical(predict(faultDetector,embeddings));

    predictions = [predictions;mode(predictionsPerFile)];
end
```

Use `confusionchart` (Statistics and Machine Learning Toolbox) to display the performance of the classifier.

```
accuracy = sum(predictions==adsValidation.Labels)/numel(adsValidation.Labels);
cc = confusionchart(predictions,adsValidation.Labels);
cc.Title = sprintf("Accuracy = %0.2f %",accuracy*100);
```



References

[1] Verma, Nishchal K., Rahul Kumar Sevakula, Sonal Dixit, and Al Salour. 2016. “Intelligent Condition Based Monitoring Using Acoustic Signals for Air Compressors.” *IEEE Transactions on Reliability* 65 (1): 291–309. https://doi.org/10.1109/TR.2015.2459684.

Input Arguments

`audioIn` — Input signal

column vector | matrix

Input signal, specified as a column vector or matrix. If you specify a matrix, `vggishEmbeddings` treats the columns of the matrix as individual audio channels.

The duration of `audioIn` must be equal to or greater than 0.975 seconds.

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`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `OverlapPercentage=75`

`OverlapPercentage` — Percentage overlap between consecutive audio frames
`50` (default) | scalar in the range [0,100)

Percentage overlap between consecutive audio frames, specified as a scalar in the range [0,100).

Data Types: `single` | `double`

`ApplyPCA` — Flag to apply PCA transformation to audio embeddings
`false` (default) | `true`

Flag to apply PCA transformation to audio embeddings, specified as either `true` or `false`.

Data Types: `logical`

Output Arguments

`embeddings` — Compact representation of audio data
L-by-128-by-*N* array

Compact representation of audio data, returned as an *L*-by-128-by-*N* array, where:

- *L* –- Represents the number of frames the audio signal is partitioned into. This is determined by the `OverlapPercentage`.
- 128 –- Represents the audio embedding length.
- *N* –- Represents the number of channels.

Algorithms

The `vggishEmbeddings` function uses VGGish to extract feature embeddings from audio. The `vggishEmbeddings` function preprocesses the audio so that it is in the format required by VGGish and optionally postprocesses the embeddings.

Preprocess

1. Resample `audioIn` to 16 kHz and cast to single precision.
2. Compute a one-sided short time Fourier transform using a 25 ms periodic Hann window with a 10 ms hop, and a 512-point DFT. The audio is now represented by a 257-by-*L* array, where 257 is the number of bins in the one-sided spectra, and *L* depends on the length of the input.
3. Convert the complex spectral values to magnitude and discard phase information.
4. Pass the one-sided magnitude spectrum through a 64-band mel-spaced filter bank, then sum the magnitudes in each band. The audio is now represented by a single 64-by-*L* mel spectrogram.
5. Convert the mel spectrogram to a log scale.
6. Buffer the mel spectrogram into overlapped segments consisting of 96 spectra each. The audio is now represented by a 96-by-64-by-1-by-*K* array, where 96 is the number of spectra in the individual mel spectrograms, 64 is the number of mel bands, and the spectrograms are spaced along the fourth dimension for compatibility with the VGGish model. The number of mel spectrograms, *K*, depends on the input length and `OverlapPercentage`.

Feature Extraction

Pass the 96-by-64-by-1-by-*K* array of mel spectrograms through VGGish to return a *K*-by-128 matrix. The output from VGGish are the feature embeddings corresponding to each 0.975 s frame of audio data.

Postprocess

If `ApplyPCA` is set to `true`, the feature embeddings are postprocessed to match the postprocessing of the released AudioSet embeddings. The VGGish model was released with a precomputed principal component analysis (PCA) matrix and mean vector to apply a PCA transformation and whitening during inference. The postprocessing includes applying PCA, whitening, and quantization.

1. Subtract the precomputed 1-by-128 PCA mean vector from the *K*-by-128 feature matrix, and then premultiply the result by the precomputed 128-by-128 PCA matrix.
2. Clip the transformed and whitened embeddings to between –2 and 2, then quantize the result to values that can be represented by `uint8`.

Version History

Introduced in R2022a

References

[1] Gemmeke, Jort F., Daniel P. W. Ellis, Dylan Freedman, Aren Jansen, Wade Lawrence, R. Channing Moore, Manoj Plakal, and Marvin Ritter. 2017. “Audio Set: An Ontology and Human-Labeled Dataset for Audio Events.” In *2017 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 776–80. New Orleans, LA: IEEE. https://doi.org/10.1109/ICASSP.2017.7952261.

[2] Hershey, Shawn, Sourish Chaudhuri, Daniel P. W. Ellis, Jort F. Gemmeke, Aren Jansen, R. Channing Moore, Manoj Plakal, et al. 2017. “CNN Architectures for Large-Scale Audio Classification.” In *2017 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 131–35. New Orleans, LA: IEEE. https://doi.org/10.1109/ICASSP.2017.7952132.

Extended Capabilities

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

See Also

Apps

Signal Labeler

Blocks

Sound Classifier | VGGish Embeddings | VGGish Preprocess | VGGish | YAMNet | YAMNet Preprocess

Functions

`audioFeatureExtractor` | `classifySound` | `vggish` | `vggishPreprocess` | `yamnet` | `yamnetGraph` | `yamnetPreprocess`

vggishFeatures

(To be removed) Extract VGGish features

Note The `vggishFeatures` function will be removed in a future release. Use `vggishEmbeddings` instead. For more information, see “Compatibility Considerations”.

Syntax

```
embeddings = vggishFeatures(audioIn,fs)
embeddings = vggishFeatures(audioIn,fs,Name,Value)
```

Description

`embeddings = vggishFeatures(audioIn,fs)` returns VGGish feature embeddings over time for the audio input `audioIn` with sample rate `fs`. Columns of the input are treated as individual channels.

`embeddings = vggishFeatures(audioIn,fs,Name,Value)` specifies options using one or more `Name,Value` arguments. For example, `embeddings = vggishFeatures(audioIn,fs,'ApplyPCA',true)` applies a principal component analysis (PCA) transformation to the audio embeddings.

This function requires both Audio Toolbox and Deep Learning Toolbox.

Examples

Download `vggishFeatures` Functionality

Download and unzip the Audio Toolbox™ model for VGGish.

Type `vggishFeatures` at the command line. If the Audio Toolbox model for VGGish is not installed, then the function provides a link to the location of the network weights. To download the model, click the link. Unzip the file to a location on the MATLAB path.

Alternatively, execute the following commands to download and unzip the VGGish model to your temporary directory.

```
downloadFolder = fullfile(tempdir,'VGGishDownload');
loc = websave(downloadFolder,'https://ssd.mathworks.com/supportfiles/audio/vggish.zip');
VGGishLocation = tempdir;
unzip(loc,VGGishLocation)
addpath(fullfile(VGGishLocation,'vggish'))
```

Extract VGGish Embeddings

Read in an audio file.

```
[audioIn,fs] = audioread('MainStreetOne-16-16-mono-12secs.wav');
```

Call the `vggishFeatures` function with the audio and sample rate to extract VGGish feature embeddings from the audio.

```
featureVectors = vggishFeatures(audioIn,fs);
```

The `vggishFeatures` function returns a matrix of 128-element feature vectors over time.

```
[numHops,numElementsPerHop,numChannels] = size(featureVectors)
```

```
numHops = 23
```

```
numElementsPerHop = 128
```

```
numChannels = 1
```

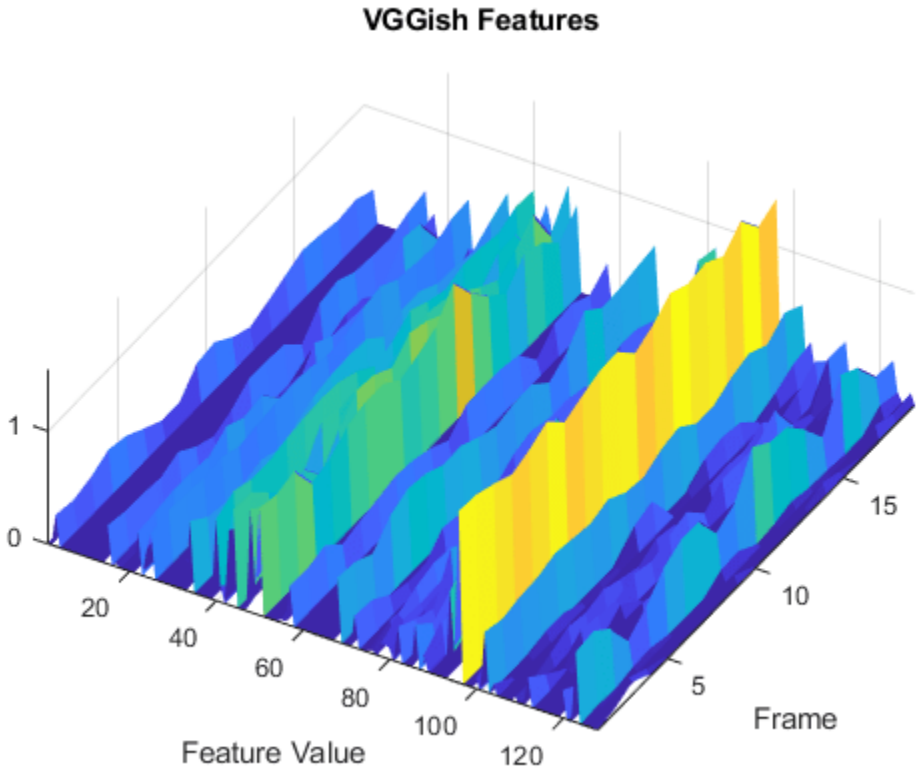
Increase Time Resolution of VGGish Features

Create a 10-second pink noise signal and then extract VGGish features. The `vggishFeatures` function extracts features from mel spectrograms with 50% overlap.

```
fs = 16e3;
dur = 10;

audioIn = pinknoise(dur*fs,1,'single');

features = vggishFeatures(audioIn,fs);
```

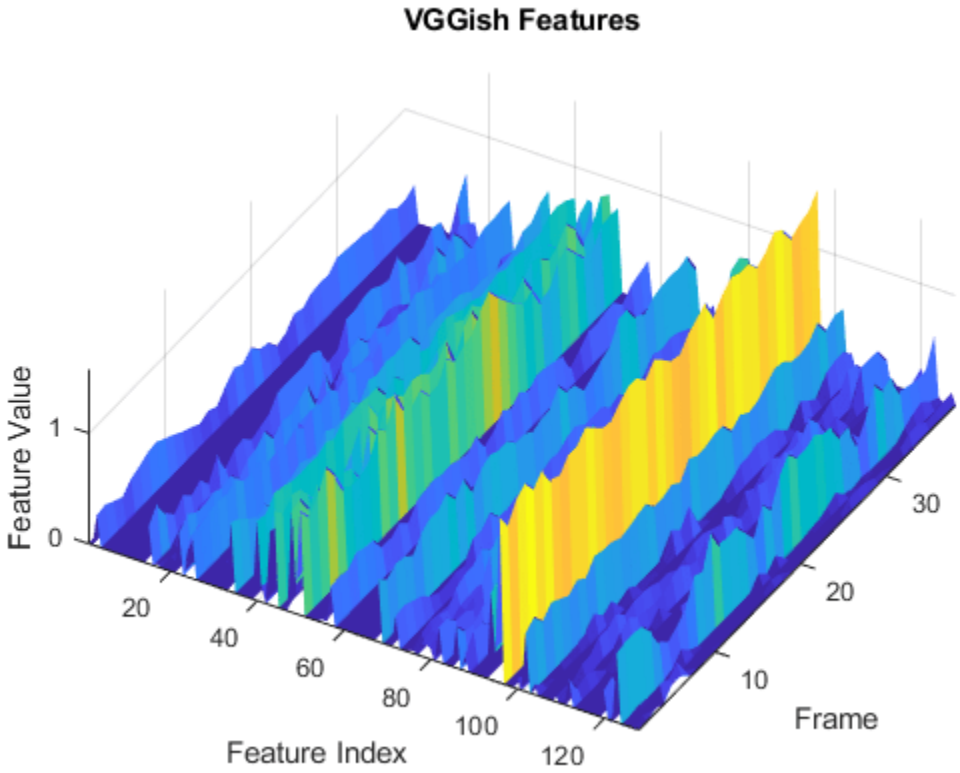
Plot the VGGish features over time.

```
surf(features,'EdgeColor','none')
view([30 65])
axis tight
xlabel('Feature Index')
ylabel('Frame')
xlabel('Feature Value')
title('VGGish Features')
```



To increase the resolution of VGGish features over time, specify the percent overlap between mel spectrograms. Plot the results.

```
overlapPercentage = 75 ;
features = vggishFeatures(audioIn,fs,'OverlapPercentage',overlapPercentage);

surf(features,'EdgeColor','none')
view([30 65])
axis tight
xlabel('Feature Index')
ylabel('Frame')
zlabel('Feature Value')
title('VGGish Features')
```



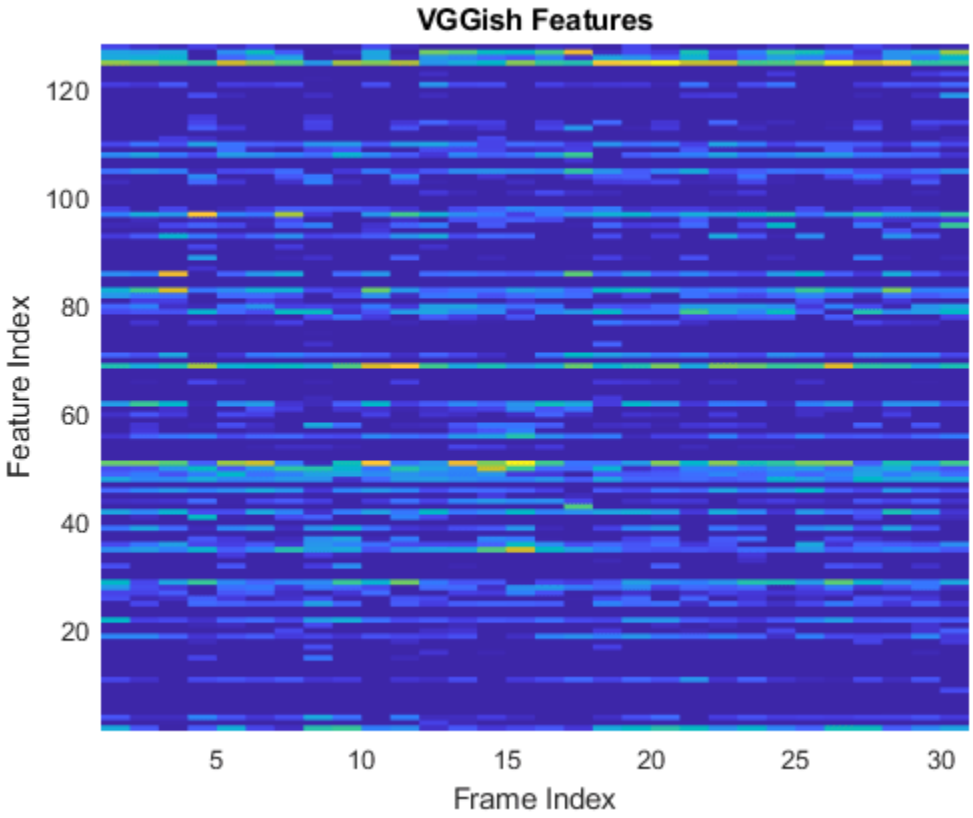
Apply Principal Component Analysis to VGGish Embeddings

Read in an audio file, listen to it, and then extract VGGish features from the audio.

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

sound(audioIn,fs)

features = vggishFeatures(audioIn,fs);
```

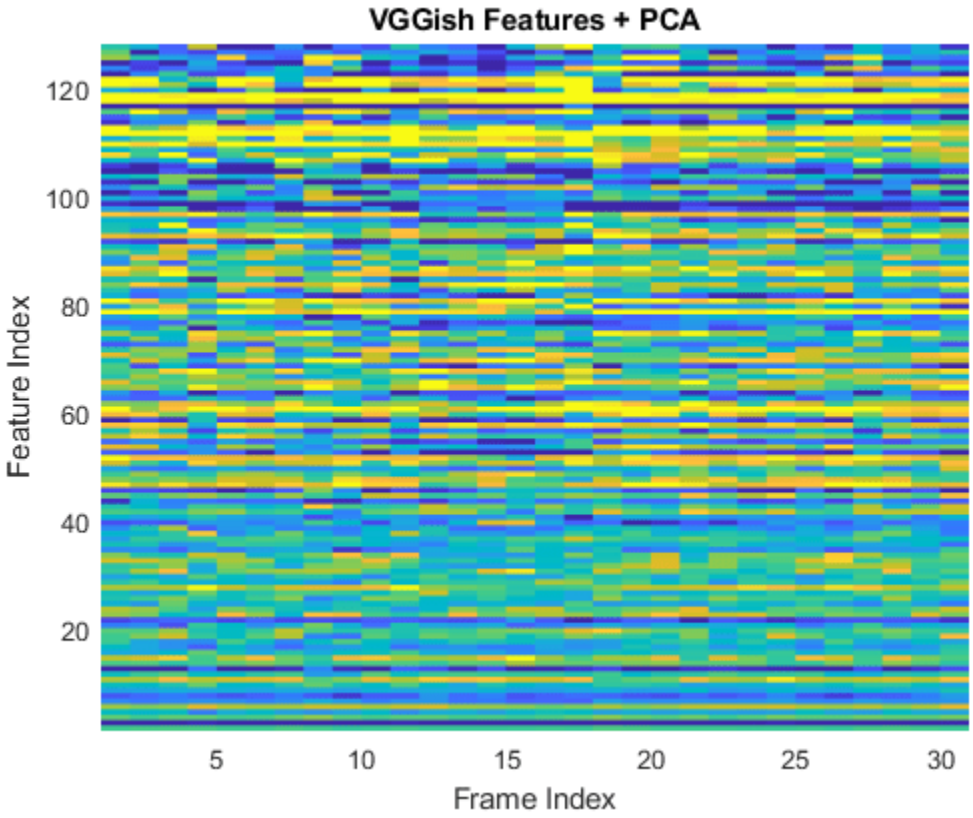
Visualize the VGGish features over time. Many of the individual features are zero-valued and contain no useful information.

```
surf(features,'EdgeColor','none')
view([90,-90])
axis tight
xlabel('Feature Index')
ylabel('Frame Index')
title('VGGish Features')
```



You can apply principal component analysis (PCA) to map the feature vectors into a space that emphasizes variation between the embeddings. Call the `vggishFeatures` function again and specify `ApplyPCA` as `true`. Visualize the VGGish features after PCA.

```
features = vggishFeatures(audioIn,fs,'ApplyPCA',true);

surf(features,'EdgeColor','none')
view([90,-90])
axis tight
xlabel('Feature Index')
ylabel('Frame Index')
title('VGGish Features + PCA')
```



Use VGGish Embeddings for Deep Learning

Download and unzip the air compressor data set. This data set consists of recordings from air compressors in a healthy state or in one of seven faulty states.

```
url = 'https://www.mathworks.com/supportfiles/audio/AirCompressorDataset/AirCompressorDataset.zip
downloadFolder = fullfile(tempdir,'aircompressordataset');
datasetLocation = tempdir;

if ~exist(fullfile(tempdir,'AirCompressorDataSet'),'dir')
    loc = websave(downloadFolder,url);
    unzip(loc,fullfile(tempdir,'AirCompressorDataSet'))
end
```

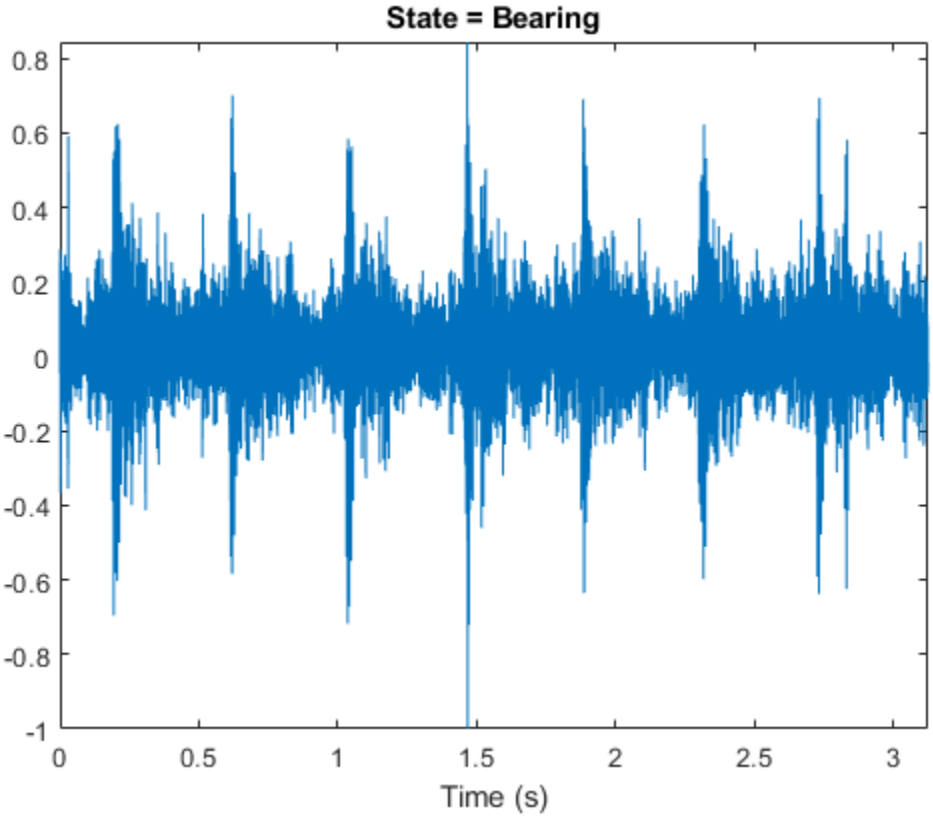
Create an `audioDatastore` object to manage the data and split it into training and validation sets.

```
ads = audioDatastore(downloadFolder,'IncludeSubfolders',true,'LabelSource','foldernames');

[adsTrain,adsValidation] = splitEachLabel(ads,0.8,0.2);
```

Read an audio file from the datastore and save the sample rate for later use. Reset the datastore to return the read pointer to the beginning of the data set. Listen to the audio signal and plot the signal in the time domain.

```
[x,fileInfo] = read(adsTrain);
fs = fileInfo.SampleRate;
```

```
reset(adsTrain)

sound(x,fs)

figure
t = (0:size(x,1)-1)/fs;
plot(t,x)
xlabel('Time (s)')
title('State = ' + string(fileInfo.Label))
axis tight
```



Extract VGGish features from the training and validation sets. Transpose the features so that time is along rows.

```
trainFeatures = cell(1,numel(adsTrain.Files));
for idx = 1:numel(adsTrain.Files)
    [audioIn,fileInfo] = read(adsTrain);
    features = vggishFeatures(audioIn,fileInfo.SampleRate);
    trainFeatures{idx} = features';
end

validationFeatures = cell(1,numel(adsValidation.Files));
for idx = 1:numel(adsValidation.Files)
    [audioIn,fileInfo] = read(adsValidation);
    features = vggishFeatures(audioIn,fileInfo.SampleRate);
    validationFeatures{idx} = features';
end
```

Define a “Long Short-Term Memory Neural Networks” (Deep Learning Toolbox) network.

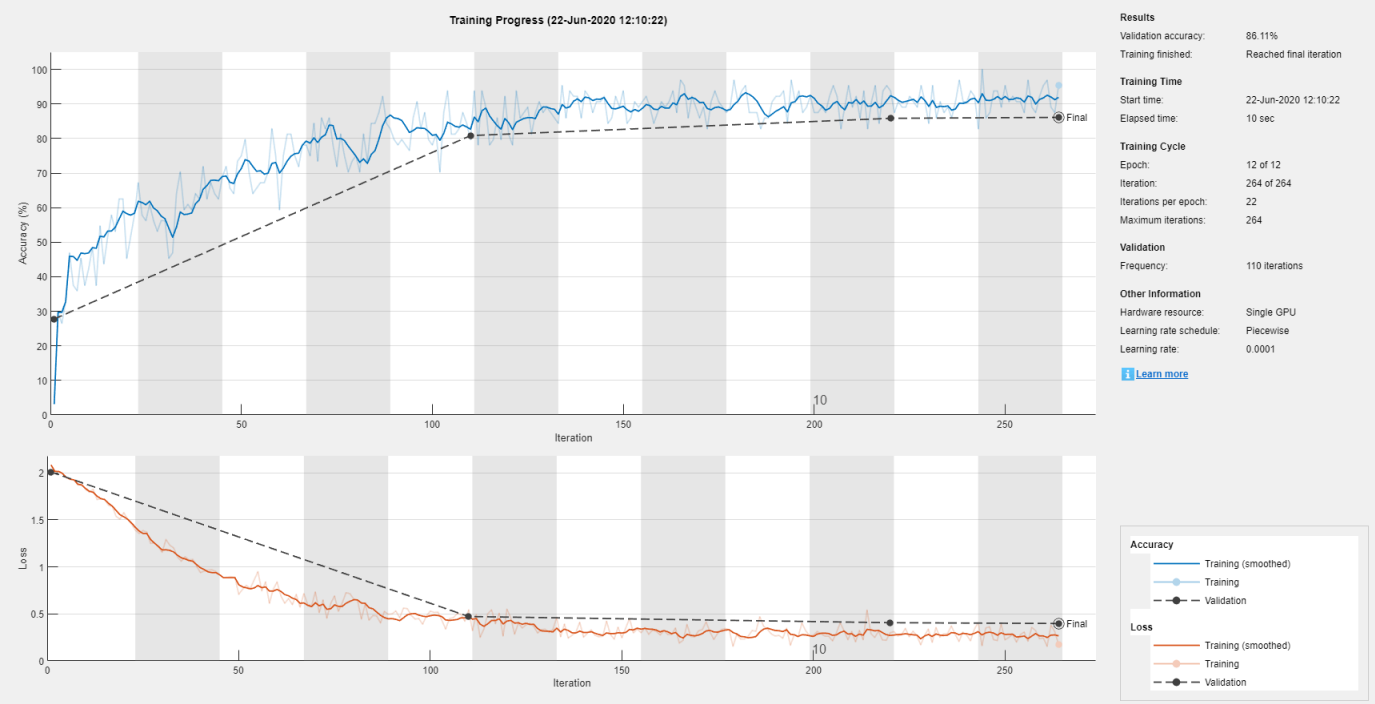
```
layers = [
    sequenceInputLayer(128)
    lstmLayer(100,'OutputMode','last')
    fullyConnectedLayer(8)
    softmaxLayer
    classificationLayer];
```

To define training options, use `trainingOptions` (Deep Learning Toolbox).

```
miniBatchSize = 64;
validationFrequency = 5*floor(numel(trainFeatures)/miniBatchSize);
options = trainingOptions("adam", ...
    "MaxEpochs",12, ...
    "MiniBatchSize",miniBatchSize, ...
    "Plots","training-progress", ...
    "Shuffle","every-epoch", ...
    "LearnRateSchedule","piecewise", ...
    "LearnRateDropPeriod",6, ...
    "LearnRateDropFactor",0.1, ...
    "ValidationData",{validationFeatures,adsValidation.Labels}, ...
    "ValidationFrequency",validationFrequency, ...
    'Verbose',false);
```

To train the network, use `trainNetwork` (Deep Learning Toolbox).

```
net = trainNetwork(trainFeatures,adsTrain.Labels,layers,options)
```

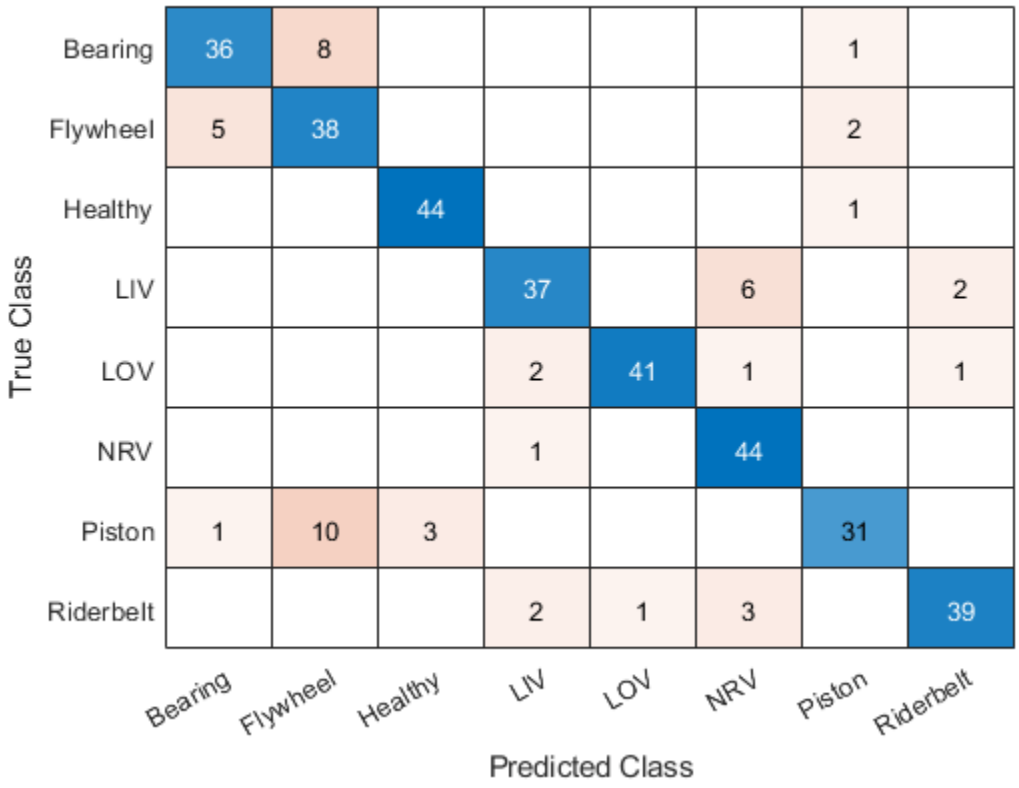



```
net =
  SeriesNetwork with properties:

         Layers: [5×1 nnet.cnn.layer.Layer]
```

```
     InputNames: {'sequenceinput'}
    OutputNames: {'classoutput'}
```

Visualize the confusion matrix for the validation set.

```
predictedClass = classify(net,validationFeatures);
confusionchart(adsValidation.Labels,predictedClass)
```



Use VGGish Embeddings for Machine Learning

Download and unzip the air compressor data set [1] on page 2-152. This data set consists of recordings from air compressors in a healthy state or in one of seven faulty states.

```
url = 'https://www.mathworks.com/supportfiles/audio/AirCompressorDataset/AirCompressorDataset.zip
downloadFolder = fullfile(tempdir,'aircompressordataset');
datasetLocation = tempdir;

if ~exist(fullfile(tempdir,'AirCompressorDataSet'),'dir')
    loc = websave(downloadFolder,url);
    unzip(loc,fullfile(tempdir,'AirCompressorDataSet'))
end
```

Create an `audioDatastore` object to manage the data and split it into training and validation sets.

```
ads = audioDatastore(downloadFolder,'IncludeSubfolders',true,'LabelSource','foldernames');
```

In this example, you classify signals as either healthy or faulty. Combine all of the faulty labels into a single label. Split the datastore into training and validation sets.

```
labels = ads.Labels;
labels(labels~=categorical("Healthy")) = categorical("Faulty");
ads.Labels = removecats(labels);

[adsTrain,adsValidation] = splitEachLabel(ads,0.8,0.2);
```

Extract VGGish features from the training set. Each audio file corresponds to multiple VGGish features. Replicate the labels so that they are in one-to-one correspondence with the features.

```
trainFeatures = [];
trainLabels = [];
for idx = 1:numel(adsTrain.Files)
    [audioIn,fileInfo] = read(adsTrain);
    features = vggishFeatures(audioIn,fileInfo.SampleRate);
    trainFeatures = [trainFeatures;features];
    trainLabels = [trainLabels;repelem(fileInfo.Label,size(features,1))'];
end
```

Train a cubic support vector machine (SVM) using `fitcsvm` (Statistics and Machine Learning Toolbox). To explore other classifiers and their performances, use Classification Learner (Statistics and Machine Learning Toolbox).

```
faultDetector = fitcsvm( ...
    trainFeatures, ...
    trainLabels, ...
    'KernelFunction','polynomial', ...
    'PolynomialOrder',3, ...
    'KernelScale','auto', ...
    'BoxConstraint',1, ...
    'Standardize',true, ...
    'ClassNames',categories(trainLabels));
```

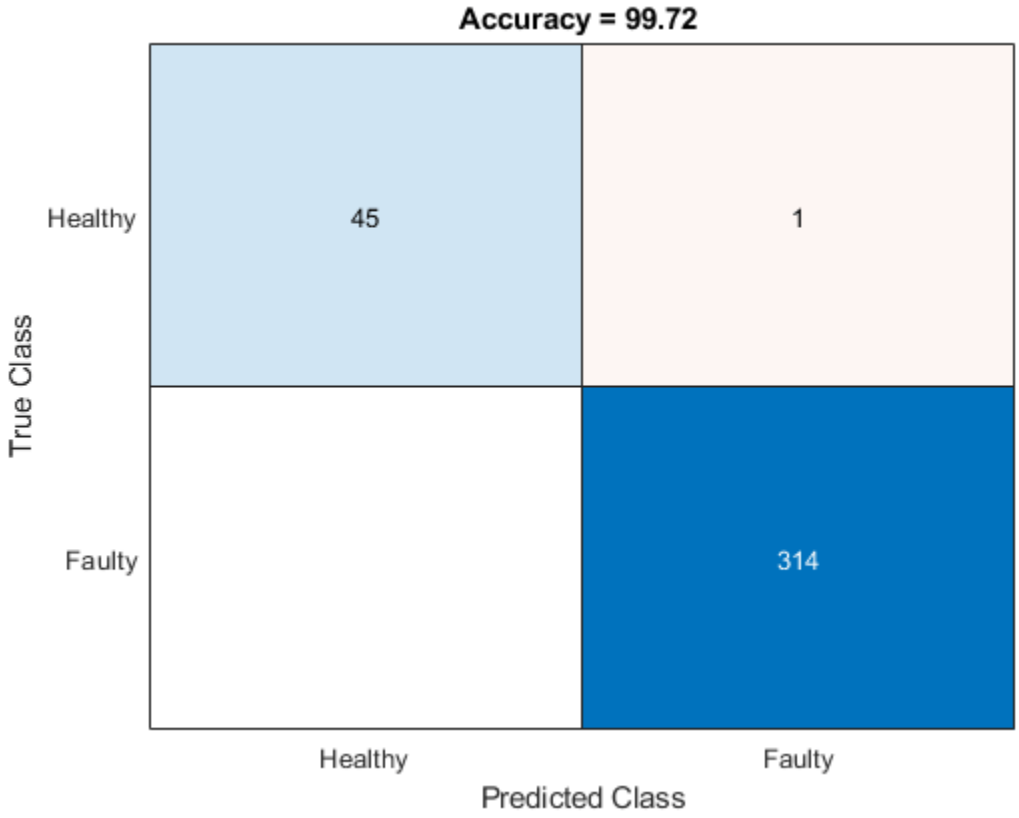
For each file in the validation set:

1. Extract VGGish features.
2. For each VGGish feature vector in a file, use the trained classifier to predict whether the machine is healthy or faulty.
3. Take the mode of the predictions for each file.

```
predictions = [];
for idx = 1:numel(adsValidation.Files)
    [audioIn,fileInfo] = read(adsValidation);

    features = vggishFeatures(audioIn,fileInfo.SampleRate);

    predictionsPerFile = categorical(predict(faultDetector,features));

    predictions = [predictions;mode(predictionsPerFile)];
end
```

Use `confusionchart` (Statistics and Machine Learning Toolbox) to display the performance of the classifier.

```
accuracy = sum(predictions==adsValidation.Labels)/numel(adsValidation.Labels);
cc = confusionchart(predictions,adsValidation.Labels);
cc.Title = sprintf('Accuracy = %0.2f %',accuracy*100);
```



References

[1] Verma, Nishchal K., Rahul Kumar Sevakula, Sonal Dixit, and Al Salour. 2016. “Intelligent Condition Based Monitoring Using Acoustic Signals for Air Compressors.” *IEEE Transactions on Reliability* 65 (1): 291–309. https://doi.org/10.1109/TR.2015.2459684.

Input Arguments

`audioIn` — Input signal
column vector | matrix

Input signal, specified as a column vector or matrix. If you specify a matrix, `vggishFeatures` treats the columns of the matrix as individual audio channels.

The duration of `audioIn` must be equal to or greater than 0.975 seconds.

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`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `'OverlapPercentage',75`

`OverlapPercentage` — Percentage overlap between consecutive audio frames
`50` (default) | scalar in the range [0,100)

Percentage overlap between consecutive audio frames, specified as a scalar in the range [0,100).

Data Types: `single` | `double`

`ApplyPCA` — Flag to apply PCA transformation to audio embeddings
`false` (default) | `true`

Flag to apply PCA transformation to audio embeddings, specified as either `true` or `false`.

Data Types: `logical`

Output Arguments

`embeddings` — Compact representation of audio data
L-by-128-by-*N* array

Compact representation of audio data, returned as an *L*-by-128-by-*N* array, where:

- *L* –- Represents the number of frames the audio signal is partitioned into. This is determined by the `OverlapPercentage`.
- 128 –- Represents the audio embedding length.
- *N* –- Represents the number of channels.

Algorithms

The `vggishFeatures` function uses VGGish to extract feature embeddings from audio. The `vggishFeatures` function preprocesses the audio so that it is in the format required by VGGish and optionally postprocesses the embeddings.

Preprocess

1. Resample `audioIn` to 16 kHz and cast to single precision.
2. Compute a one-sided short time Fourier transform using a 25 ms periodic Hann window with a 10 ms hop, and a 512-point DFT. The audio is now represented by a 257-by-*L* array, where 257 is the number of bins in the one-sided spectra, and *L* depends on the length of the input.
3. Convert the complex spectral values to magnitude and discard phase information.

4. Pass the one-sided magnitude spectrum through a 64-band mel-spaced filter bank, then sum the magnitudes in each band. The audio is now represented by a single 64-by-*L* mel spectrogram.
5. Convert the mel spectrogram to a log scale.
6. Buffer the mel spectrogram into overlapped segments consisting of 96 spectra each. The audio is now represented by a 96-by-64-by-1-by-*K* array, where 96 is the number of spectra in the individual mel spectrograms, 64 is the number of mel bands, and the spectrograms are spaced along the fourth dimension for compatibility with the VGGish model. The number of mel spectrograms, *K*, depends on the input length and `OverlapPercentage`.

Feature Extraction

Pass the 96-by-64-by-1-by-*K* array of mel spectrograms through VGGish to return a *K*-by-128 matrix. The output from VGGish are the feature embeddings corresponding to each 0.975 s frame of audio data.

Postprocess

If `ApplyPCA` is set to `true`, the feature embeddings are postprocessed to match the postprocessing of the released AudioSet embeddings. The VGGish model was released with a precomputed principal component analysis (PCA) matrix and mean vector to apply a PCA transformation and whitening during inference. The postprocessing includes applying PCA, whitening, and quantization.

1. Subtract the precomputed 1-by-128 PCA mean vector from the *K*-by-128 feature matrix, and then premultiply the result by the precomputed 128-by-128 PCA matrix.
2. Clip the transformed and whitened embeddings to between –2 and 2, then quantize the result to values that can be represented by `uint8`.

Version History

Introduced in R2020b

R2022a: `vggishFeatures` will be removed

Not recommended starting in R2022a

The `vggishFeatures` function will be removed in a future release. Use `vggishEmbeddings` instead. Existing calls to `vggishFeatures` continue to run.

References

[1] Gemmeke, Jort F., Daniel P. W. Ellis, Dylan Freedman, Aren Jansen, Wade Lawrence, R. Channing Moore, Manoj Plakal, and Marvin Ritter. 2017. “Audio Set: An Ontology and Human-Labeled Dataset for Audio Events.” In *2017 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 776–80. New Orleans, LA: IEEE. https://doi.org/10.1109/ICASSP.2017.7952261.

[2] Hershey, Shawn, Sourish Chaudhuri, Daniel P. W. Ellis, Jort F. Gemmeke, Aren Jansen, R. Channing Moore, Manoj Plakal, et al. 2017. “CNN Architectures for Large-Scale Audio Classification.” In *2017 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 131–35. New Orleans, LA: IEEE. https://doi.org/10.1109/ICASSP.2017.7952132.

Extended Capabilities

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

See Also

Apps

Signal Labeler

Blocks

Sound Classifier | VGGish Embeddings | VGGish Preprocess | VGGish | YAMNet | YAMNet Preprocess

Functions

`audioFeatureExtractor` | `classifySound` | `vggish` | `vggishPreprocess` | `yamnet` | `yamnetGraph` | `yamnetPreprocess`

yamnetGraph

Graph of YAMNet AudioSet ontology

Syntax

```
ygraph = yamnetGraph
[ygraph,classes] = yamnetGraph
```

Description

`ygraph = yamnetGraph` returns a directed graph of the AudioSet ontology.

`[ygraph,classes] = yamnetGraph` also returns a string array of classes supported by YAMNet.

This function requires both Audio Toolbox and Deep Learning Toolbox.

Examples

Download `yamnetGraph`

Download and unzip the Audio Toolbox™ support for YAMNet.

Type `yamnetGraph` at the Command Window. If the Audio Toolbox support for YAMNet is not installed, then the function provides a link to the download location. To download the model, click the link. Unzip the file to a location on the MATLAB path.

Alternatively, execute the following commands to download and unzip the YAMNet model to your temporary directory.

```
downloadFolder = fullfile(tempdir,'YAMNetDownload');
loc = websave(downloadFolder,'https://ssd.mathworks.com/supportfiles/audio/yamnet.zip');
YAMNetLocation = tempdir;
unzip(loc,YAMNetLocation)
addpath(fullfile(YAMNetLocation,'yamnet'))
```

Check that the installation is successful by typing `yamnetGraph` at the Command Window. If the network is installed, then the function returns a `digraph` object.

```
yamnetGraph
```

Identify Major Categories of Ontology

Create a `digraph` object that describes the AudioSet ontology.

```
ygraph = yamnetGraph
```

```
ygraph =
  digraph with properties:
```

```
    Edges: [670×1 table]
    Nodes: [632×1 table]
```

Visualize the ontology. The ontology consists of 632 separate classes with 670 connections.

```
p = plot(ygraph);
layout(p,'layered')
```

Get the name of each sound class. If the sound class has no predecessors, identify it as a major category of the ontology.

```
nodeNames = ygraph.Nodes.Name;
topCategories = {};
for index = 1:numel(nodeNames)
    pre = predecessors(ygraph,nodeNames{index});
    if isempty(pre)
        topCategories{end+1} = nodeNames{index};
    end
end
```

Display the categories as an array of strings.

```
topCategories = string(topCategories)
```

```
topCategories = 1×7 string
    "Human sounds"    "Animal"    "Music"    "Natural sounds"    "Sounds of things"    "Source-ar
```

Highlight and label the top categories on the `digraph` plot.

```
highlight(p,topCategories,"NodeColor","red","MarkerSize",8)
labelnode(p,topCategories,topCategories)
```

Plot Subgraph of Animal Sounds

Create a `digraph` object that represents the AudioSet ontology.

```
ygraph = yamnetGraph;
```

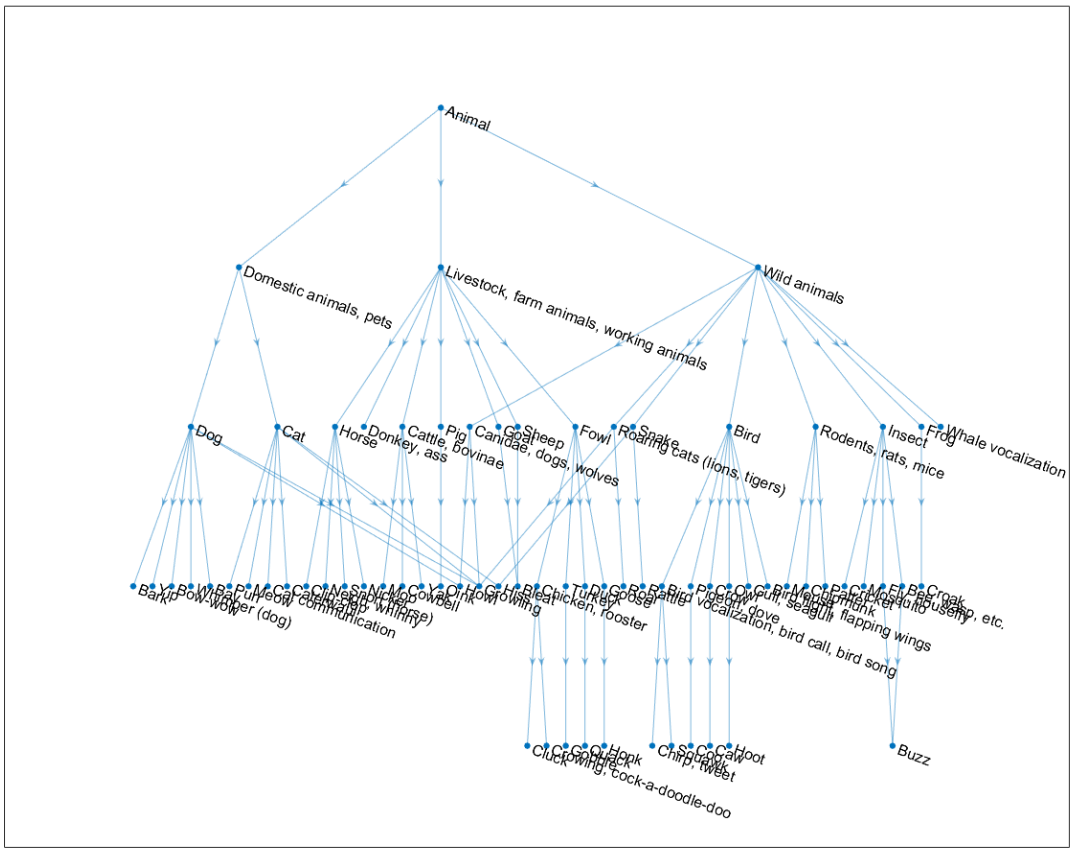
Use `dfsearch` to perform a depth-first graph search to identify all audio classes under the class `Animal`.

```
animalNodes = dfsearch(ygraph,"Animal");
```

Use `subgraph` to create a new `digraph` object that only includes the identified audio classes. Plot the resulting directed edges graph.

```
animalGraph = subgraph(ygraph,animalNodes);

p = plot(animalGraph);

p.NodeFontSize = 12;
graphFigure = gcf;
```

```
old = graphFigure.Position;
set(graphFigure,'position',[old(1),old(2),old(3)*3,old(4)*3])
```



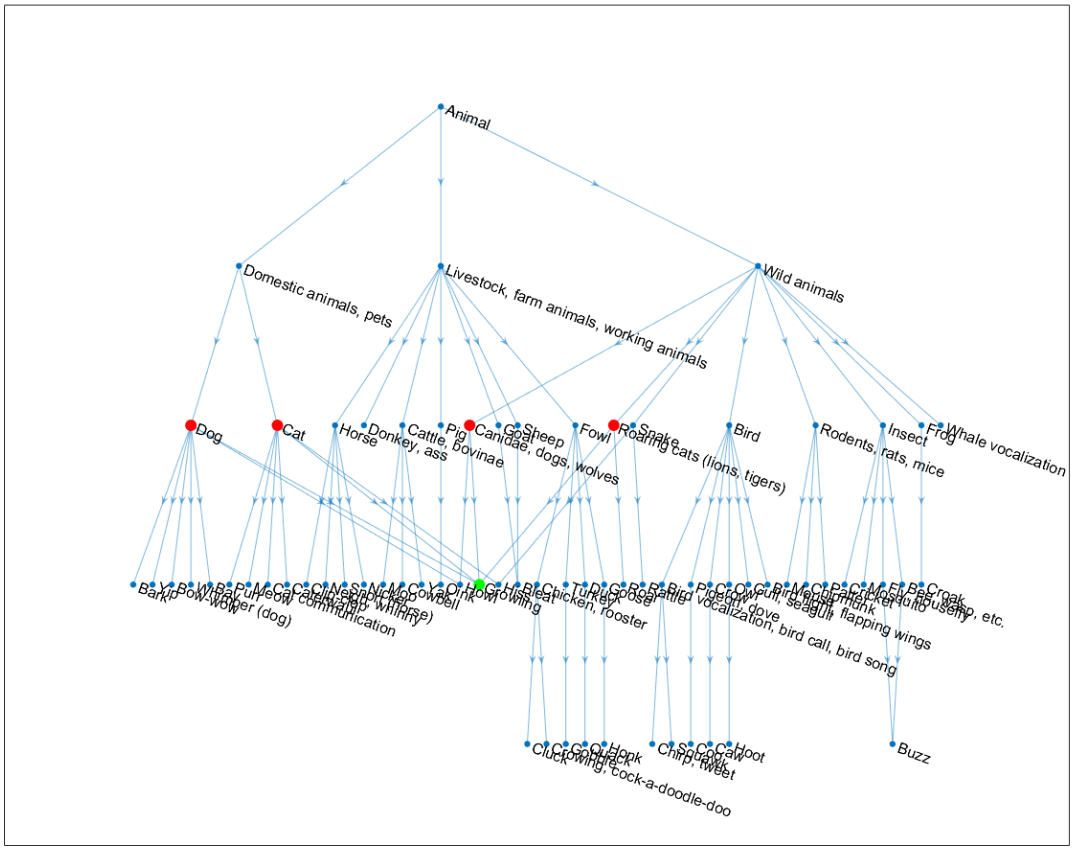
Use `predecessors` to determine all predecessors to the `Growling` sound. Highlight the predecessors on the plot.

```
preIDs = predecessors(animalGraph,"Growling")
```

```
preIDs = 4×1 string
    "Dog"
    "Cat"
    "Roaring cats (lions, tigers)"
    "Canidae, dogs, wolves"
```

Use `highlight` to highlight the `Growling` node and the predecessors on the plot.

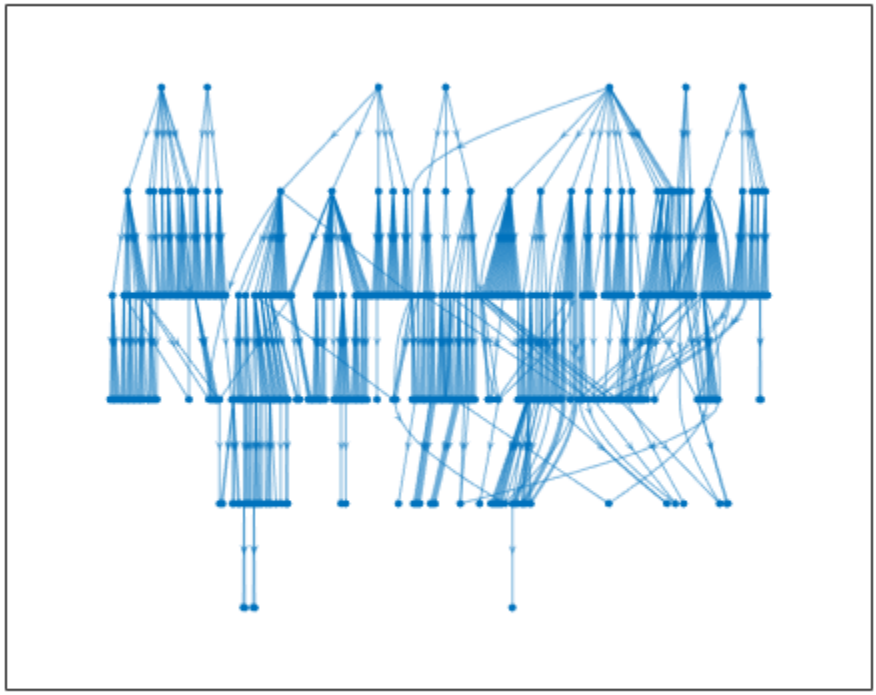
```
highlight(p,"Growling",'NodeColor','g','MarkerSize',8)
highlight(p,preIDs,'NodeColor','r','MarkerSize',8)
```



Visualize Sounds Supported by YAMNet

Create a digraph object that describes the AudioSet ontology. Also return the classes supported by YAMNet. Plot the directed graph.

```
[ygraph,classes] = yamnetGraph;
p = plot(ygraph);
layout(p,'layered')
```



YAMNet predicts a subset of the full AudioSet ontology. Display the sound classes that are in the AudioSet ontology but are not possible outputs from the YAMNet network.

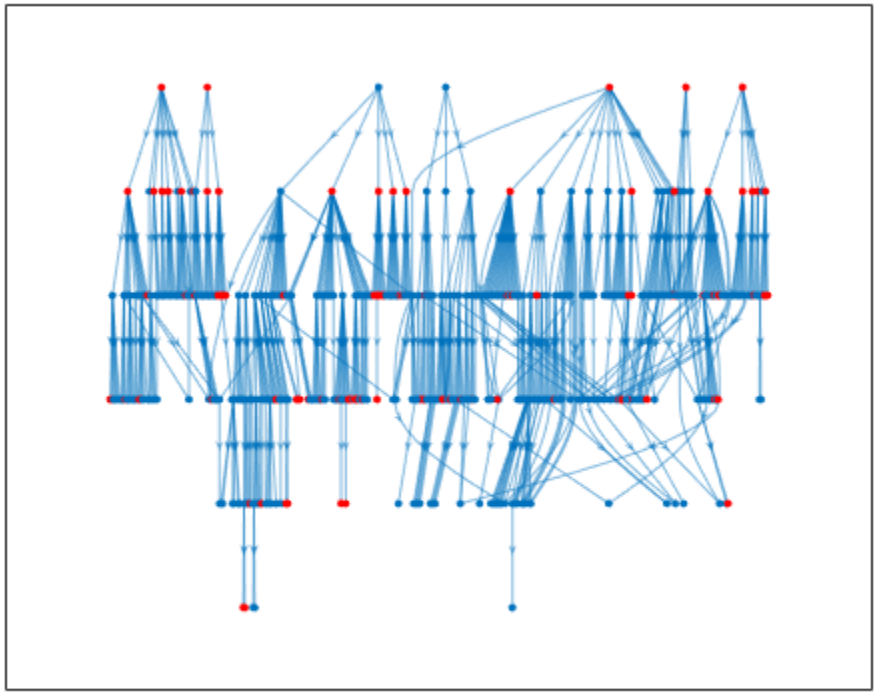
```
audiosetClasses = ygraph.Nodes.Name;
classDiff = setdiff(audiosetClasses,classes)
```

```
classDiff = 111×1 string
    "Acoustic environment"
    "Alto saxophone"
    "Background noise"
    "Bass (frequency range)"
    "Bass (instrument role)"
    "Bassline"
    "Bassoon"
    "Battle cry"
    "Bay"
    "Beat"
    "Birthday music"
    "Blare"
    "Booing"
    "Brief tone"
    "Bugle"
    "Cat communication"
    "Cellphone buzz, vibrating alert"
    "Channel, environment and background"
    "Chipmunk"
    "Chord"
    "Clavinet"
```

```
"Clunk"
"Compact disc"
"Cornet"
"Crash cymbal"
"Cumbia"
"Deformable shell"
"Digestive"
"Domestic sounds, home sounds"
"Donkey, ass"
  ⋮
```

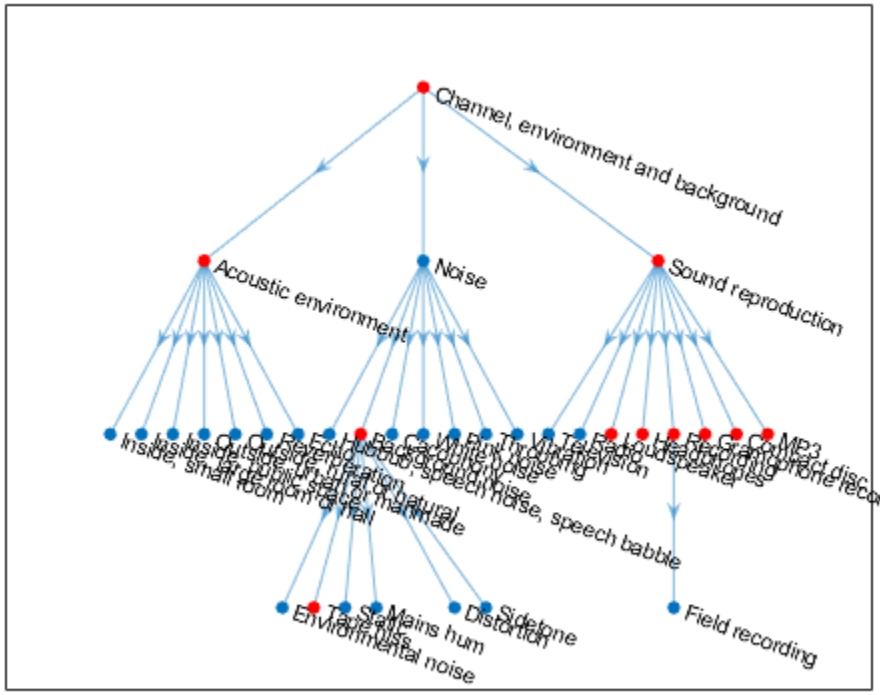
Highlight the classes that are not possible outputs from YAMNet.

```
highlight(p,classDiff,'NodeColor','r')
```



Analyze one of the major categories.

```
categoryToAnalyze = Channel, environm...;
subsetNodes = dfsearch(ygraph,categoryToAnalyze);
ygraphSubset = subgraph(ygraph,subsetNodes);
classToHighlight = intersect(classDiff,ygraphSubset.Nodes.Name);
pSub = plot(ygraphSubset);
layout(pSub,'layered')
highlight(pSub,classToHighlight,'NodeColor','r')
```



Visualize Specificity of Sound Classes

Create a `digraph` object that describes the AudioSet ontology.

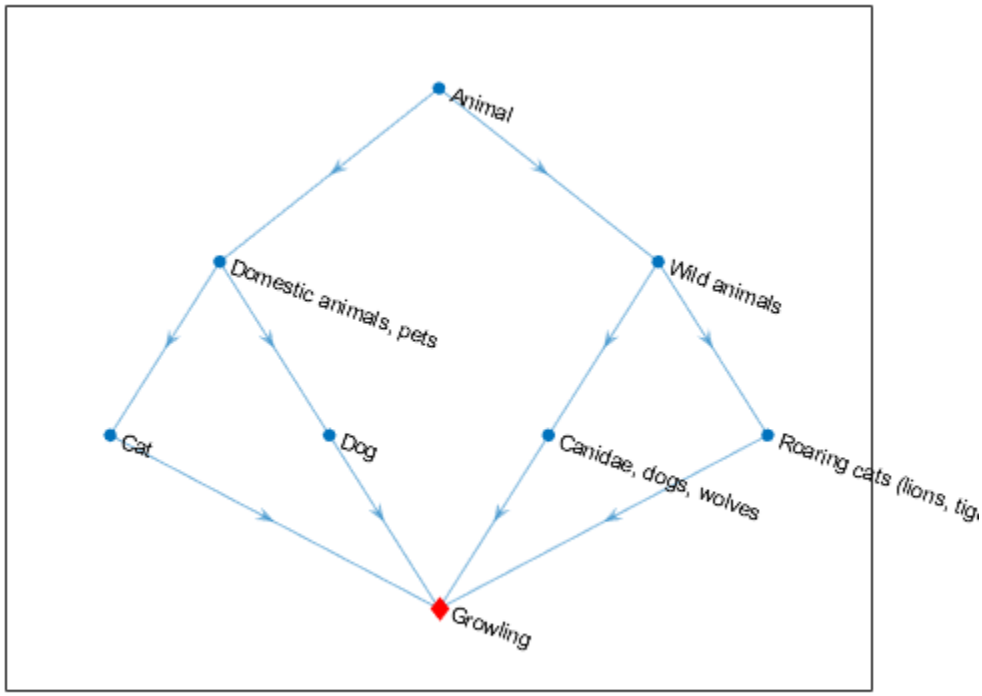
```
ygraph = yamnetGraph;
```

Specify a sound class to visualize, and specify the number of predecessors and successors. The available sound classes are only those that are supported as outputs from YAMNet. If you specify more predecessors or successors than those in the ontology, only the predecessors and successors in the ontology are shown.

```
soundClass = Growling;
numPredecessors = 3;
numSuccessors = 0;
```

```
pred = nearest(ygraph,soundClass,numPredecessors,'Direction','incoming');
suc = nearest(ygraph,soundClass,numSuccessors,'Direction','outgoing');
subClasses = [soundClass;pred;suc];

ygraphSub = subgraph(ygraph,unique(subClasses));
p = plot(ygraphSub);
```

```
layout(p,'layered')
highlight(p,soundClass,'Marker','d','NodeColor','red','MarkerSize',6)
```



Output Arguments

`ygraph` — AudioSet ontology graph with directed edges

`digraph` object

AudioSet ontology graph with directed edges, returned as a `digraph` object.

`classes` — Classes supported by YAMNet

string array

Classes supported by YAMNet, returned as a string array. The classes supported by YAMNet are a subset of the AudioSet ontology.

Tips

Google® provides a website where you can explore the AudioSet ontology and the corresponding data set: https://research.google.com/audioset/ontology/index.html.

Version History

Introduced in R2020b

References

[1] Gemmeke, Jort F., et al. “Audio Set: An Ontology and Human-Labeled Dataset for Audio Events.” *2017 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, IEEE, 2017, pp. 776–80. *DOI.org (Crossref)*, doi:10.1109/ICASSP.2017.7952261.

[2] Hershey, Shawn, et al. “CNN Architectures for Large-Scale Audio Classification.” *2017 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, IEEE, 2017, pp. 131–35. *DOI.org (Crossref)*, doi:10.1109/ICASSP.2017.7952132.

See Also

Apps
Signal Labeler

Blocks
Sound Classifier | VGGish Embeddings | VGGish Preprocess | VGGish | YAMNet | YAMNet Preprocess

Functions
`classifySound` | `vggish` | `vggishEmbeddings` | `vggishPreprocess` | `yamnet` | `yamnetPreprocess`

classifySound

Classify sounds in audio signal

Syntax

```
sounds = classifySound(audioIn,fs)
sounds = classifySound(audioIn,fs,Name,Value)

[sounds,timestamps] = classifySound( ___ )
[sounds,timestamps,resultsTable] = classifySound( ___ )

classifySound( ___ )
```

Description

`sounds = classifySound(audioIn,fs)` returns the sound classes detected over time in the audio input, `audioIn`, with sample rate `fs`.

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

Example: `sounds = classifySound(audioIn,fs,'SpecificityLevel','low')` classifies sounds using low specificity.

`[sounds,timestamps] = classifySound( ___ )` also returns time stamps associated with each detected sound.

`[sounds,timestamps,resultsTable] = classifySound( ___ )` also returns a table containing result details.

`classifySound( ___ )` with no output arguments creates a word cloud of the identified sounds in the audio signal.

This function requires both Audio Toolbox and Deep Learning Toolbox.

Examples

Download `classifySound`

Download and unzip the Audio Toolbox™ support for YAMNet.

If the Audio Toolbox support for YAMNet is not installed, then the first call to the function provides a link to the download location. To download the model, click the link. Unzip the file to a location on the MATLAB path.

Alternatively, execute the following commands to download and unzip the YAMNet model to your temporary directory.

```
downloadFolder = fullfile(tempdir,'YAMNetDownload');
loc = websave(downloadFolder,'https://ssd.mathworks.com/supportfiles/audio/yamnet.zip');
```

```
YAMNetLocation = tempdir;
unzip(loc,YAMNetLocation)
addpath(fullfile(YAMNetLocation,'yamnet'))
```

Identify Colored Noise

Generate 1 second of pink noise assuming a 16 kHz sample rate.

```
fs = 16e3;
x = pinknoise(fs);
```

Call `classifySound` with the pink noise signal and the sample rate.

```
identifiedSound = classifySound(x,fs)
```

```
identifiedSound =
"Pink noise"
```

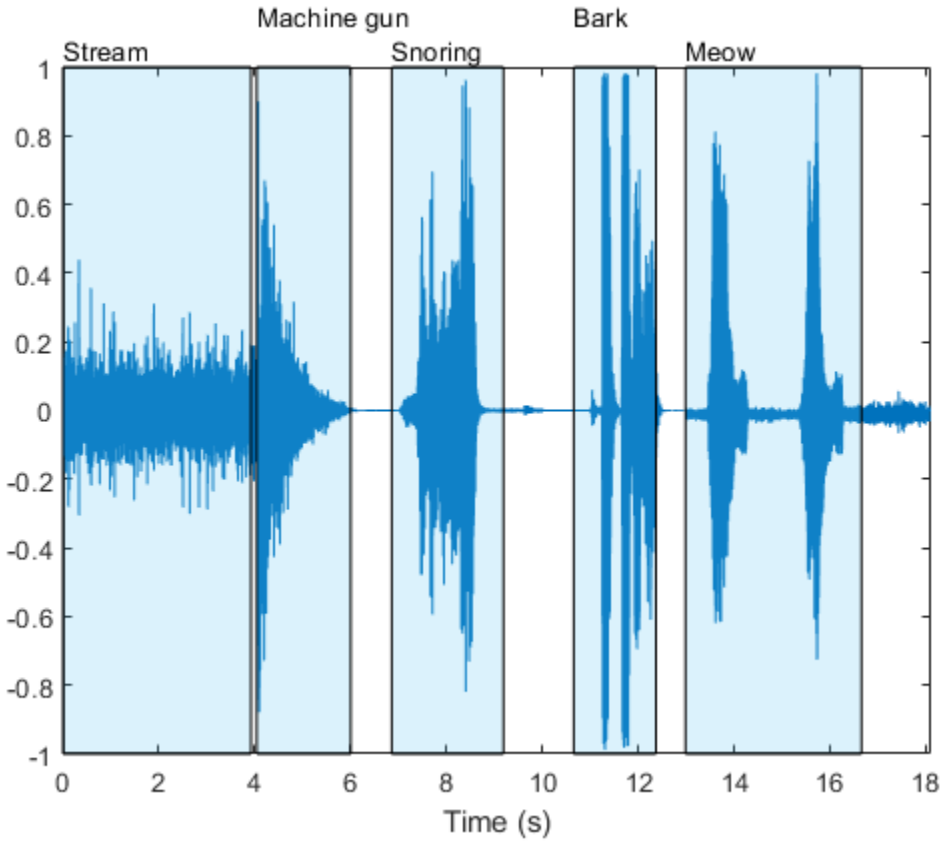
Identify and Locate Sounds in Time

Read in an audio signal. Call `classifySound` to return the detected sounds and corresponding time stamps.

```
[audioIn,fs] = audioread('multipleSounds-16-16-mono-18secs.wav');
[sounds,timeStamps] = classifySound(audioIn,fs);
```

Plot the audio signal and label the detected sound regions.

```
t = (0:numel(audioIn)-1)/fs;
plot(t,audioIn)
xlabel('Time (s)')
axis([t(1),t(end),-1,1])

textHeight = 1.1;
for idx = 1:numel(sounds)
    patch([timeStamps(idx,1),timeStamps(idx,1),timeStamps(idx,2),timeStamps(idx,2)], ...
        [-1,1,1,-1], ...
        [0.3010 0.7450 0.9330], ...
        'FaceAlpha',0.2);
    text(timeStamps(idx,1),textHeight+0.05*(-1)^idx,sounds(idx))
end
```



Select a region and listen only to the selected region.

```
sampleStamps = floor(timeStamps*fs)+1;
soundEvent = three;

isolatedSoundEvent = audioIn(sampleStamps(soundEvent,1):sampleStamps(soundEvent,2));
sound(isolatedSoundEvent,fs);
display('Detected Sound = ' + sounds(soundEvent))
```

```
    "Detected Sound = Snoring"
```

Identify Only Specific Sounds

Read in an audio signal containing multiple different sound events.

```
[audioIn,fs] = audioread('multipleSounds-16-16-mono-18secs.wav');
```

Call `classifySound` with the audio signal and sample rate.

```
[sounds,~,soundTable] = classifySound(audioIn,fs);
```

The `sounds` string array contains the most likely sound event in each region.

```
sounds
```

```
sounds = 1×5 string
    "Stream"    "Machine gun"    "Snoring"    "Bark"    "Meow"
```

The `soundTable` contains detailed information regarding the sounds detected in each region, including score means and maximums over the analyzed signal.

```
soundTable
```

```
soundTable=5×2 table
       TimeStamps          Results
    ________________    ___________

         0      3.92    {4×3 table}
    4.0425    6.0025    {3×3 table}
      6.86    9.1875    {2×3 table}
    10.658    12.373    {4×3 table}
    12.985     16.66    {4×3 table}
```

View the last detected region.

```
soundTable.Results{end}
```

```
ans=4×3 table
             Sounds              AverageScores    MaxScores
    ____________________________    _____________    _________

    "Animal"                           0.79514        0.99941
    "Domestic animals, pets"           0.80243        0.99831
    "Cat"                               0.8048        0.99046
    "Meow"                              0.6342        0.90177
```

Call `classifySound` again. This time, set `IncludedSounds` to `Animal` so that the function retains only regions in which the `Animal` sound class is detected.

```
[sounds,timeStamps,soundTable] = classifySound(audioIn,fs, ...
    'IncludedSounds','Animal');
```

The sounds array only returns sounds specified as included sounds. The `sounds` array now contains two instances of `Animal` that correspond to the regions declared as `Bark` and `Meow` previously.

```
sounds
```

```
sounds = 1×2 string
    "Animal"    "Animal"
```

The sound table only includes regions where the specified sound classes were detected.

```
soundTable
```

```
soundTable=2×2 table
       TimeStamps          Results
    ________________    ___________

    10.658    12.373    {4×3 table}
```

```
    12.985     16.66    {4×3 table}
```

View the last detected region in `soundTable`. The results table still includes statistics for all detected sounds in the region.

```
soundTable.Results{end}
```

```
ans=4×3 table
             Sounds              AverageScores    MaxScores
    ____________________________    _____________    _________

    "Animal"                          0.79514        0.99941
    "Domestic animals, pets"          0.80243        0.99831
    "Cat"                              0.8048        0.99046
    "Meow"                             0.6342        0.90177
```

To explore which sound classes are supported by `classifySound`, use `yamnetGraph`.

Exclude Specific Sounds

Read in an audio signal and call `classifySound` to inspect the most likely sounds arranged in chronological order of detection.

```
[audioIn,fs] = audioread("multipleSounds-16-16-mono-18secs.wav");
sounds = classifySound(audioIn,fs)
```

```
sounds = 1×5 string
    "Stream"    "Machine gun"    "Snoring"    "Bark"    "Meow"
```

Call `classifySound` again and set `ExcludedSounds` to `Meow` to exclude the sound `Meow` from the results. The segment previously classified as `Meow` is now classified as `Cat`, which is its immediate predecessor in the AudioSet ontology.

```
sounds = classifySound(audioIn,fs,"ExcludedSounds","Meow")
```

```
sounds = 1×5 string
    "Stream"    "Machine gun"    "Snoring"    "Bark"    "Cat"
```

Call `classifySound` again, and set `ExcludedSounds` to `Cat`. When you exclude a sound, all successors are also excluded. This means that excluding the sound `Cat` also excludes the sound `Meow`. The segment originally classified as `Meow` is now classified as `Domestic animals, pets`, which is the immediate predecessor to `Cat` in the AudioSet ontology.

```
sounds = classifySound(audioIn,fs,"ExcludedSounds","Cat")
```

```
sounds = 1×5 string
    "Stream"    "Machine gun"    "Snoring"    "Bark"    "Domestic animals, pets"
```

Call `classifySound` again and set `ExcludedSounds` to `Domestic animals, pets`. The sound class, `Domestic animals, pets` is a predecessor to both `Bark` and `Meow`, so by excluding it, the

sounds previously identified as `Bark` and `Meow` are now both identified as the predecessor of `Domestic animals, pets`, which is `Animal`.

```
sounds = classifySound(audioIn,fs,"ExcludedSounds","Domestic animals, pets")
```

```
sounds = 1×5 string
    "Stream"    "Machine gun"    "Snoring"    "Animal"    "Animal"
```

Call `classifySound` again and set `ExcludedSounds` to `Animal`. The sound class `Animal` has no predecessors.

```
sounds = classifySound(audioIn,fs,"ExcludedSounds","Animal")
```

```
sounds = 1×3 string
    "Stream"    "Machine gun"    "Snoring"
```

If you want to avoid detecting `Meow` and its predecessors, but continue detecting successors under the same predecessors, use the `IncludedSounds` option. Call `yamnetGraph` to get a list of all supported classes. Remove `Meow` and its predecessors from the array of all classes, and then call `classifySound` again.

```
[~,classes] = yamnetGraph;
classesToInclude = setxor(classes,["Meow","Cat","Domestic animals, pets","Animal"]);
sounds = classifySound(audioIn,fs,"IncludedSounds",classesToInclude)
```

```
sounds = 1×4 string
    "Stream"    "Machine gun"    "Snoring"    "Bark"
```

Generate Word Cloud

Read in an audio signal and listen to it.

```
[audioIn,fs] = audioread('multipleSounds-16-16-mono-18secs.wav');
sound(audioIn,fs)
```

Call `classifySound` with no output arguments to generate a word cloud of the detected sounds.

```
classifySound(audioIn,fs);
```

Modify default parameters of `classifySound` to explore the effect on the word cloud.

```
threshold = 0.1 ;
minimumSoundSeparation = 0.92 ;
minimumSoundDuration = 1.02 ;

classifySound(audioIn,fs, ...
    'Threshold',threshold, ...
    'MinimumSoundSeparation',minimumSoundSeparation, ...
    'MinimumSoundDuration',minimumSoundDuration);
```

Input Arguments

`audioIn` — Audio input

column vector

Audio input, specified as a one-channel signal (column vector).

Data Types: `single` | `double`

`fs` — Sample rate (Hz)

positive scalar

Sample rate in Hz, specified as a positive scalar.

Data Types: `single` | `double`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `'Threshold',0.1`

`Threshold` — Confidence threshold for reporting sounds
`0.35` (default) | scalar in the range (0,1)

Confidence threshold for reporting sounds, specified as the comma-separated pair consisting of `'Threshold'` and a scalar in the range (0,1).

Data Types: `single` | `double`

`MinimumSoundSeparation` — Minimum separation between detected sound regions (s)
`0.25` (default) | positive scalar

Minimum separation between consecutive regions of the same detected sound in seconds, specified as the comma-separated pair consisting of `'MinimumSoundSeparation'` and a positive scalar. Regions closer than the minimum sound separation are merged.

Data Types: `single` | `double`

`MinimumSoundDuration` — Minimum duration of detected sound region (s)
`0.5` (default) | positive scalar

Minimum duration of detected sound regions in seconds, specified as the comma-separated pair consisting of `'MinimumSoundDuration'` and a positive scalar. Regions shorter than the minimum sound duration are discarded.

Data Types: `single` | `double`

`IncludedSounds` — Sounds to include in results
character vector | cell array of character vectors | string scalar | string array

Sounds to include in results, specified as the comma-separated pair consisting of `'IncludedSounds'` and a character vector, cell array of character vectors, string scalar, or string array. Use `yamnetGraph` to inspect and analyze the sounds supported by `classifySound`. By default, all supported sounds are included.

This option cannot be used with the `'ExcludedSounds'` option.

Data Types: `char` | `string` | `cell`

`ExcludedSounds` — Sounds to exclude from results
character vector | cell array of character vectors | string scalar | string array

Sounds to exclude from results, specified as the comma-separated pair consisting of `'ExcludedSounds'` and a character vector, cell array of character vectors, string scalar, or string array. When you specify an excluded sound, any successors of the excluded sound are also excluded. Use `yamnetGraph` to inspect valid sound classes and their predecessors and successors according to the AudioSet ontology. By default, no sounds are excluded.

This option cannot be used with the `'IncludedSounds'` option.

Data Types: `char` | `string` | `cell`

`SpecificityLevel` — Specificity of reported sounds
`'high'` (default) | `'low'` | `'none'`

Specificity of reported sounds, specified as the comma-separated pair consisting of `'SpecificityLevel'` and `'high'`, `'low'`, or `'none'`. Set `SpecificityLevel` to `'high'` to make the function emphasize specific sound classes instead of general categories. Set `SpecificityLevel`

to `'low'` to make the function return the most general sound categories instead of specific sound classes. Set `SpecificityLevel` to `'none'` to make the function return the most likely sound, regardless of its specificity.

Data Types: `char` | `string`

Output Arguments

`sounds` — Sounds detected over time in audio input

string array

Sounds detected over time in audio input, returned as a string array containing the detected sounds in chronological order.

`timestamps` — Time stamps associated with detected sounds (s)

N-by-2 matrix

Time stamps associated with detected sounds in seconds, returned as an *N*-by-2 matrix. *N* is the number of detected sounds. Each row of `timestamps` contains the start and end times of the detected sound region.

`resultsTable` — Detailed results of sound classification

table

Detailed results of sound classification, returned as a table. The number of rows in the table is equal to the number of detected sound regions. The columns are as follows.

- `TimeStamps` –– Time stamps corresponding to each analyzed region.
- `Results` –– Table with three variables:
  - `Sounds` –– Sounds detected in each region.
  - `AverageScores` –– Mean network scores corresponding to each detected sound class in the region.
  - `MaxScores` –– Maximum network scores corresponding to each detected sound class in the region.

Algorithms

The `classifySound` function uses YAMNet to classify audio segments into sound classes described by the AudioSet ontology. The `classifySound` function preprocesses the audio so that it is in the format required by YAMNet and postprocesses YAMNet's predictions with common tasks that make the results more interpretable.

Preprocess

1. Resample `audioIn` to 16 kHz and cast to single precision.
2. Buffer into *L* overlapping segments. Each segment is 0.98 seconds and the segments are overlapped by 0.8575 seconds.
3. Pass each segment through a one-sided short time Fourier transform using a 25 ms periodic Hann window with a 10 ms hop and a 512-point DFT. The audio is now represented by a 257-by-96-by-*L* array, where 257 is the number of bins in the one-sided spectra and 96 is the number of spectra in the spectrograms.

4 Convert the complex spectral values to magnitude and discard phase information.

5 Pass the one-sided magnitude spectrum through a 64-band mel-spaced filter bank and then sum the magnitudes in each band. The audio is now represented by a 96-by-64-by-1-by-*L* array, where 96 is the number of spectra in the mel spectrogram, 64 is the number of mel bands, and the spectrograms are now spaced along the fourth dimension for compatibility with the YAMNet model.

6 Convert the mel spectrograms to a log scale.

Prediction

Pass the 96-by-64-by-1-by-*L* array of mel spectrograms through YAMNet to return an *L*-by-521 matrix. The output from YAMNet corresponds to confidence scores for each of the 521 sound classes over time.

Postprocess

Sound Event Region Detection

1 Pass each of the 521 confidence signals through a moving mean filter with a window length of 7.

2 Pass each of the signals through a moving median filter with a window length of 3.

3 Convert the confidence signals to binary masks using the specified `Threshold`.

4 Discard any sound shorter than `MinimumSoundDuration`.

5 Merge regions that are closer than `MinimumSoundSeparation`.

Consolidate Overlapping Sound Regions

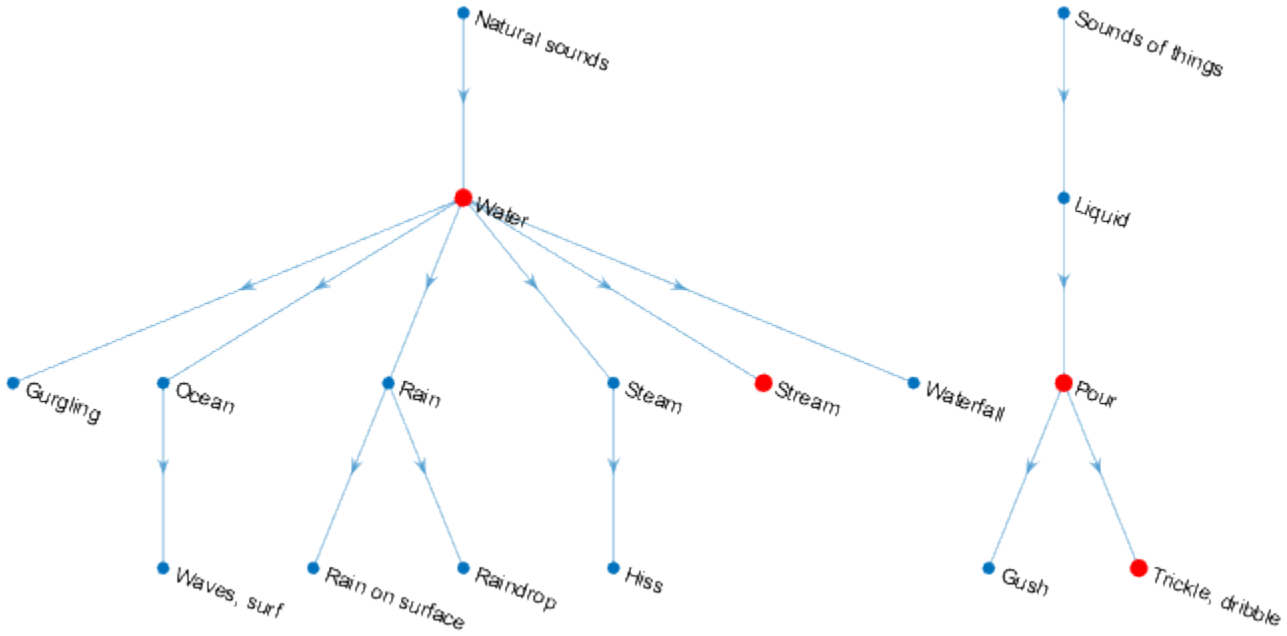
Consolidate identified sound regions that overlap by 50% or more into single regions. The region start time is the smallest start time of all sounds in the group. The region end time is the largest end time of all sounds in the group. The function returns time stamps, sounds classes, and the mean and maximum confidence of the sound classes within the region in the `resultsTable`.

Select Specificity of Sound Group

You can set the specificity level of your sound classification using the `SpecificityLevel` option. For example, assume there are four sound classes in a sound group with the following corresponding mean scores over the sound region:

- `Water` -- `0.82817`
- `Stream` -- `0.81266`
- `Trickle, dribble` -- `0.23102`
- `Pour` -- `0.20732`

The sound classes, `Water`, `Stream`, `Trickle, dribble`, and `Pour` are situated in AudioSet ontology as indicated by the graph:



The functions returns the sound class for the sound group in the `sounds` output argument depending on the `SpecificityLevel`:

- `"high"` (default) –– In this mode, `Stream` is preferred to `Water`, and `Trickle, dribble` is preferred to `Pour`. `Stream` has a higher mean score over the region, so the function returns `Stream` in the `sounds` output for the region.
- `"low"` –– In this mode, the most general ontological category for the sound class with the highest mean confidence over the region is returned. For `Trickle, dribble` and `Pour`, the most general category is `Sounds of things`. For `Stream` and `Water`, the most general category is `Natural sounds`. Because `Water` has the highest mean confidence over the sound region, the function returns `Natural sounds`.
- `"none"` –– In this mode, the function returns the sound class with the highest mean confidence score, which in this example is `Water`.

Version History

Introduced in R2020b

References

[1] Gemmeke, Jort F., et al. “Audio Set: An Ontology and Human-Labeled Dataset for Audio Events.” *2017 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, IEEE, 2017, pp. 776–80. *DOI.org (Crossref)*, doi:10.1109/ICASSP.2017.7952261.

[2] Hershey, Shawn, et al. “CNN Architectures for Large-Scale Audio Classification.” *2017 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, IEEE, 2017, pp. 131–35. *DOI.org (Crossref)*, doi:10.1109/ICASSP.2017.7952132.

Extended Capabilities

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

See Also

Apps

Signal Labeler

Blocks

Sound Classifier | VGGish Embeddings | VGGish Preprocess | VGGish | YAMNet | YAMNet Preprocess

Functions

`vggish` | `vggishEmbeddings` | `vggishPreprocess` | `yamnet` | `yamnetGraph` | `yamnetPreprocess`

acousticFluctuation

Perceived fluctuation strength of acoustic signal

Syntax

```
fluctuation = acousticFluctuation(audioIn,fs)
fluctuation = acousticFluctuation(audioIn,fs,calibrationFactor)
fluctuation = acousticFluctuation(specificLoudnessIn)
fluctuation = acousticFluctuation( ___ ,Name,Value)

[fluctuation,specificFluctuation] = acousticFluctuation( ___ )
[fluctuation,specificFluctuation,fMod] = acousticFluctuation( ___ )

acousticFluctuation( ___ )
```

Description

`fluctuation = acousticFluctuation(audioIn,fs)` returns fluctuation strength in vacil based on Zwicker et al. [1] and ISO 532-1 time-varying loudness [2].

`fluctuation = acousticFluctuation(audioIn,fs,calibrationFactor)` specifies a nondefault microphone calibration factor used to compute loudness.

`fluctuation = acousticFluctuation(specificLoudnessIn)` computes fluctuation using time-varying specific loudness.

`fluctuation = acousticFluctuation( ___ ,Name,Value)` specifies options using one or more `Name,Value` pair arguments.

Example: `fluctuation = acousticFluctuation(audioIn,fs,'SoundField','diffuse')` returns fluctuation assuming a diffuse sound field.

`[fluctuation,specificFluctuation] = acousticFluctuation( ___ )` also returns specific fluctuation strength.

`[fluctuation,specificFluctuation,fMod] = acousticFluctuation( ___ )` also returns the dominant modulation frequency.

`acousticFluctuation( ___ )` with no output arguments plots fluctuation strength and specific fluctuation strength and displays the modulation frequency textually. If the input is stereo, the 3-D plot shows the sum of both channels.

Examples

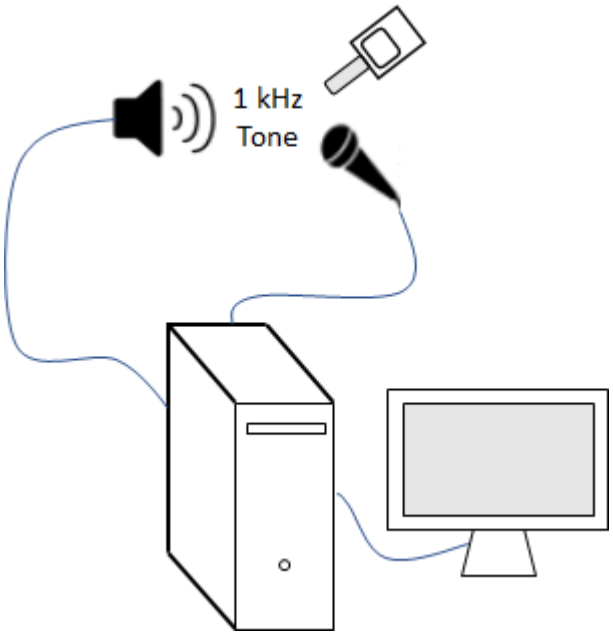
Measure Acoustic Fluctuation

Measure acoustic fluctuation based on Zwicker et al [2] and ISO 532-1 [1]. Assume the recording level is calibrated such that a 1 kHz tone registers as 100 dB on an SPL meter.

```
[audioIn,fs] = audioread('WashingMachine-16-44p1-stereo-10secs.wav');

fluctuation = acousticFluctuation(audioIn,fs);
```

Fluctuation Measurements Using Calibrated Microphone

Set up an experiment as indicated by the diagram.



Create an `audioDeviceReader` object to read from the microphone and an `audioDeviceWriter` object to write to your speaker.

```
fs = 48e3;
deviceReader = audioDeviceReader(fs,"SamplesPerFrame",2048);
deviceWriter = audioDeviceWriter(fs);
```

Create an `audioOscillator` object to generate a 1 kHz sinusoid.

```
osc = audioOscillator("sine",1e3,"SampleRate",fs,"SamplesPerFrame",2048);
```

Create a `dsp.AsyncBuffer` object to buffer data acquired from the microphone.

```
dur = 5;
buff = dsp.AsyncBuffer(dur*fs);
```

For five seconds, play the sinusoid through your speaker and record using your microphone. While the audio streams, note the loudness as reported by your SPL meter. Once complete, read the contents of the buffer object.

```
numFrames = dur*(fs/osc.SamplesPerFrame);
for ii = 1:numFrames
    audioOut = osc();
    deviceWriter(audioOut);
```

```
    audioIn = deviceReader();
    write(buff,audioIn);
end

SPLreading = 60.4;

micRecording = read(buff);
```

To compute the calibration factor for the microphone, use the `calibrateMicrophone` function.

```
calibrationFactor = calibrateMicrophone(micRecording(fs+1:end,:),deviceReader.SampleRate,SPLread
```

You can now use the calibration factor you determined to measure the fluctuation of any sound that is acquired through the same microphone recording chain.

Perform the experiment again, this time, add 100% amplitude modulation at 4 Hz. To create the modulation signal, use `audioOscillator` and specify the amplitude as `0.5` and the DC offset as `0.5` to oscillate between 0 and 1.

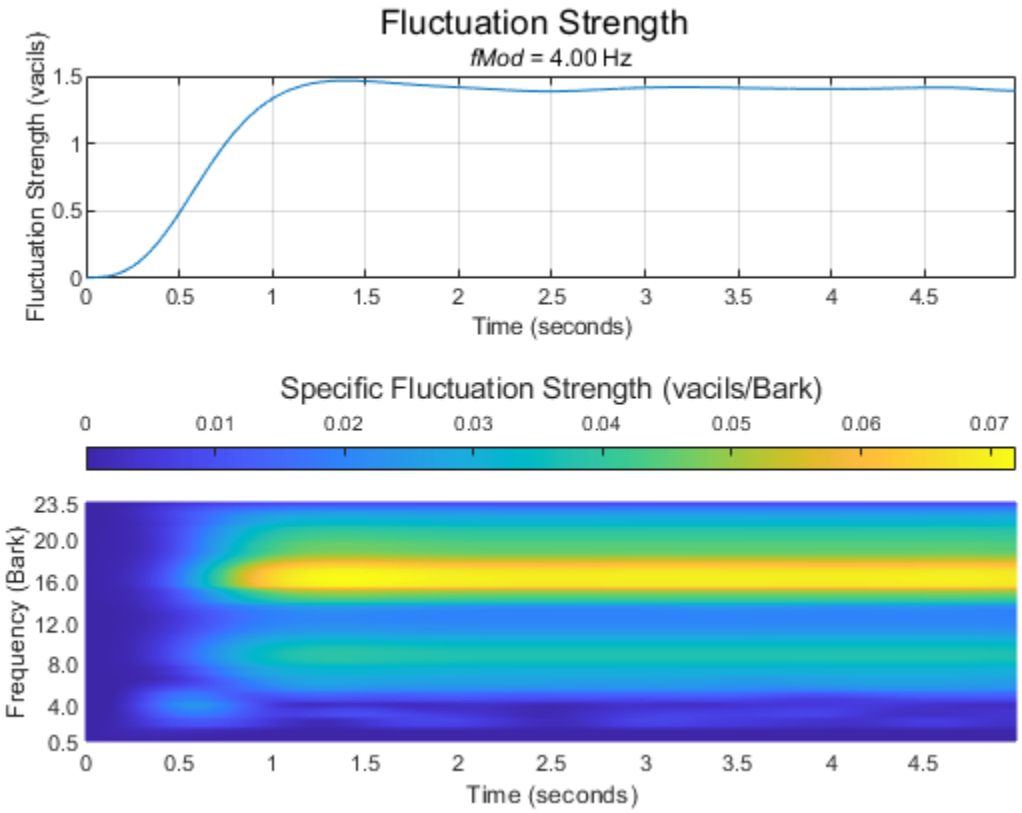
```
mod = audioOscillator("sine",4,"SampleRate",fs, ...
    "Amplitude",0.5,"DCOffset",0.5,"SamplesPerFrame",2048);

dur = 5;
buff = dsp.AsyncBuffer(dur*fs);
numFrames = dur*(fs/osc.SamplesPerFrame);
for ii = 1:numFrames
    audioOut = osc().*mod();
    deviceWriter(audioOut);

    audioIn = deviceReader();
    write(buff,audioIn);
end

micRecording = read(buff);
```

Call `acousticFluctuation` with the microphone recording, sample rate, and calibration factor. The fluctuation reported from `acousticFluctuation` uses the true acoustic loudness measurement as specified by 532-1. Display the average fluctuation strength over the 5 seconds.

```
fluctuation = acousticFluctuation(micRecording,deviceReader.SampleRate,calibrationFactor);
fprintf('Average fluctuation = %d (vacil)',mean(fluctuation(501:end,:)))
```

```
Average fluctuation = 1.413824e+00 (vacil)
```

```
acousticFluctuation(micRecording,deviceReader.SampleRate,calibrationFactor)
```



Measure Fluctuation from Specific Loudness

Read in an audio file.

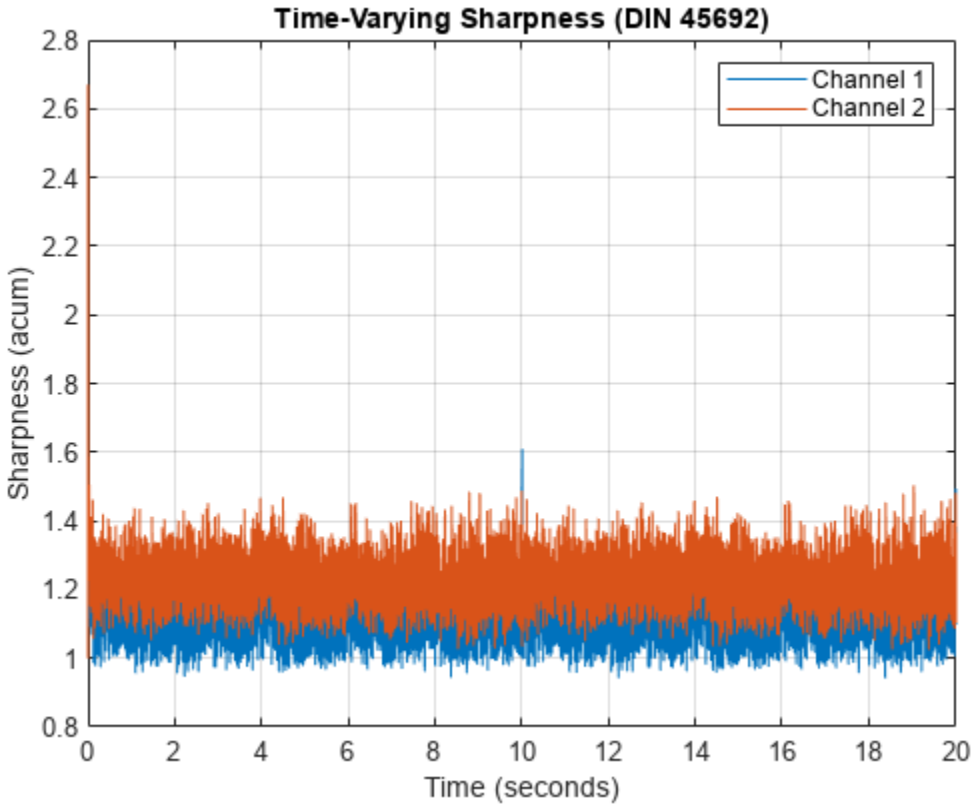
```
[audioIn,fs] = audioread("Engine-16-44p1-stereo-20sec.wav");
```

Call `acousticLoudness` to calculate the specific loudness.

```
[~,specificLoudness] = acousticLoudness(audioIn,fs,'TimeVarying',true);
```

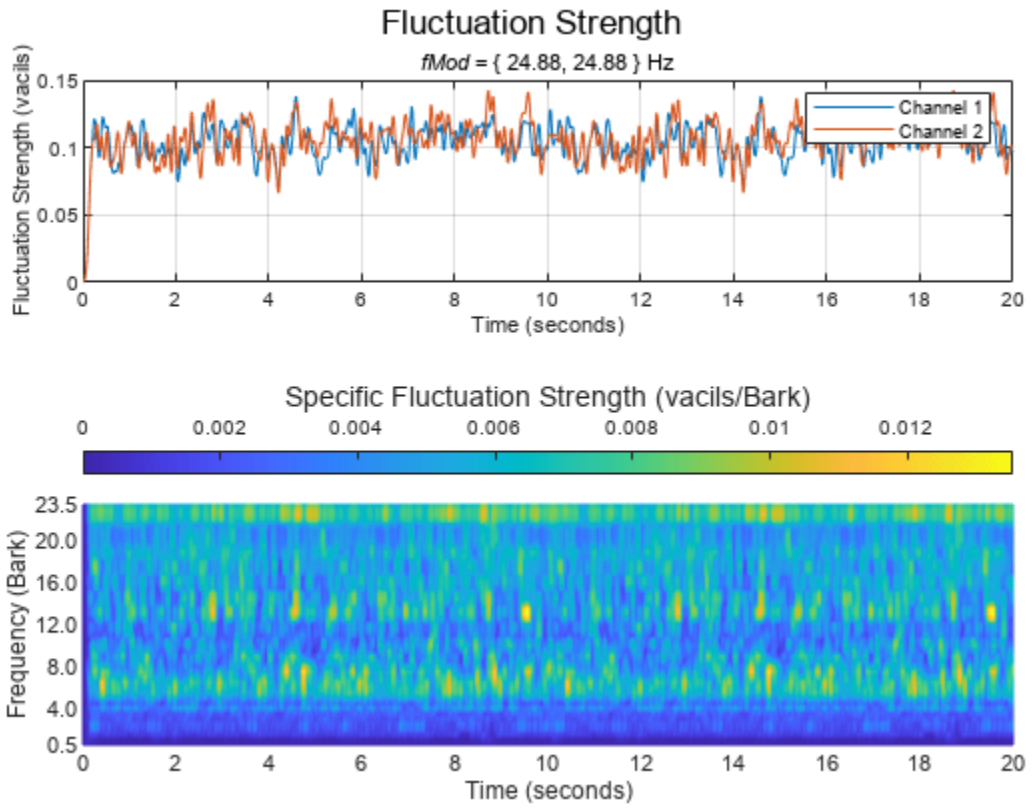
Call `acousticSharpness` without any outputs to plot the acoustic sharpness.

```
acousticSharpness(specificLoudness,'TimeVarying',true)
```



Call `acousticFluctuation` without any outputs to plot the acoustic fluctuation.

```
acousticFluctuation(specificLoudness)
```



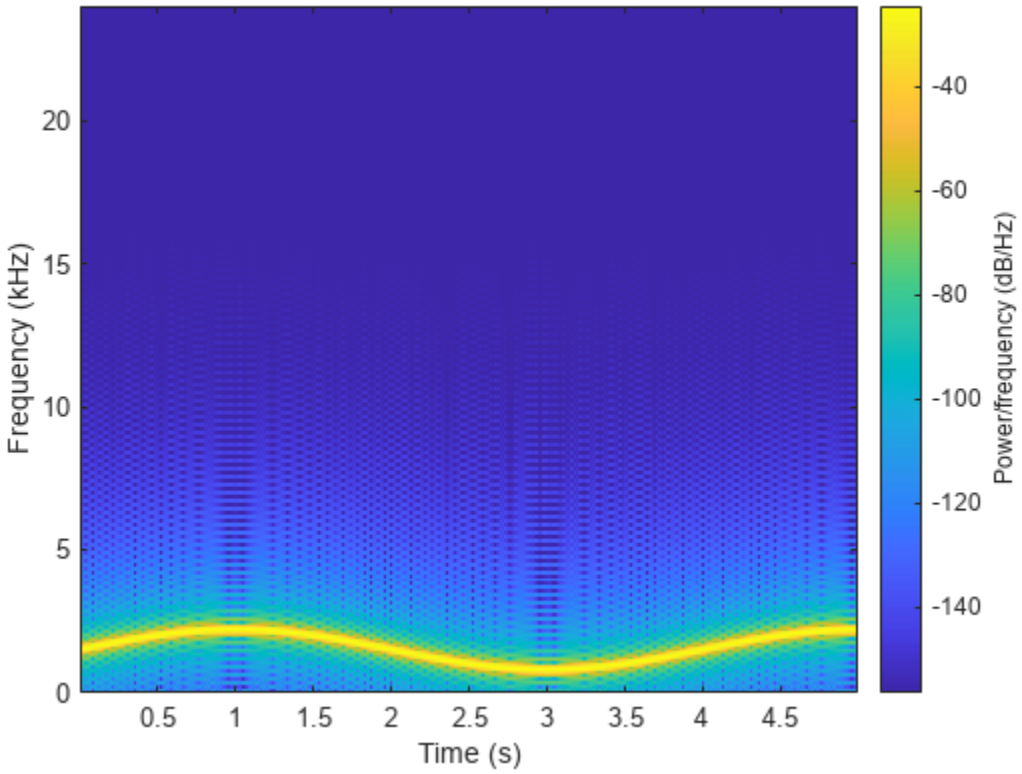
Effect of Frequency Modulation on Acoustic Fluctuation

Generate a pure tone with a 1500 Hz center frequency and approximately 700 Hz frequency deviation at a modulation frequency of 0.25 Hz.

```
fs = 48e3;

fMod = 0.25 ;
dur = 20 ;

numSamples = dur*fs;
t = (0:numSamples-1)/fs;

tone = sin(2*pi*t*fMod)';

fc = 1500 ;
excursionRatio = 0.47 ;

excursion = 2*pi*(fc*excursionRatio/fs);
audioIn = modulate(tone,fc,fs,'fm',excursion);
```

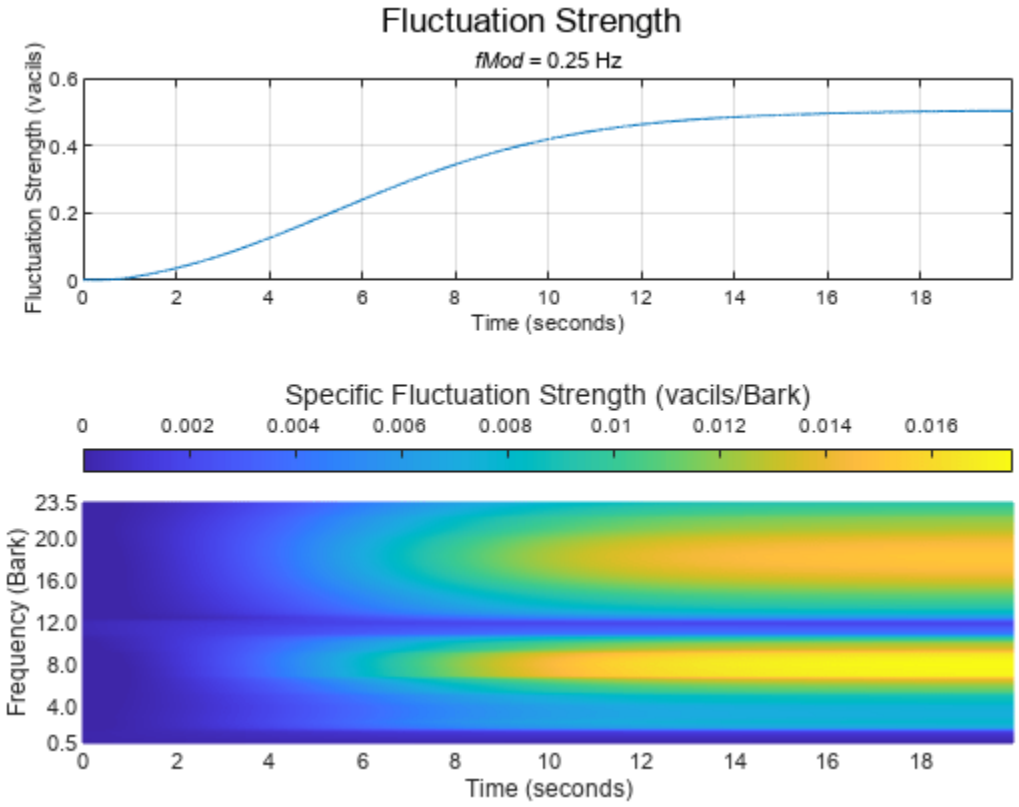
Listen to the first 5 seconds of the audio and plot the spectrogram.

```
sound(audioIn(1:5*fs),fs)
spectrogram(audioIn(1:5*fs),hann(512,'periodic'),256,1024,fs,'yaxis')
```



Call `acousticFluctuation` with no output arguments to plot the acoustic fluctuation strength.

```
acousticFluctuation(audioIn,fs);
```



Specify Known Modulation Frequency

The `acousticFluctuation` function enables you to specify a known fluctuation frequency. If you do not specify a known fluctuation frequency, the function auto-detects the fluctuation.

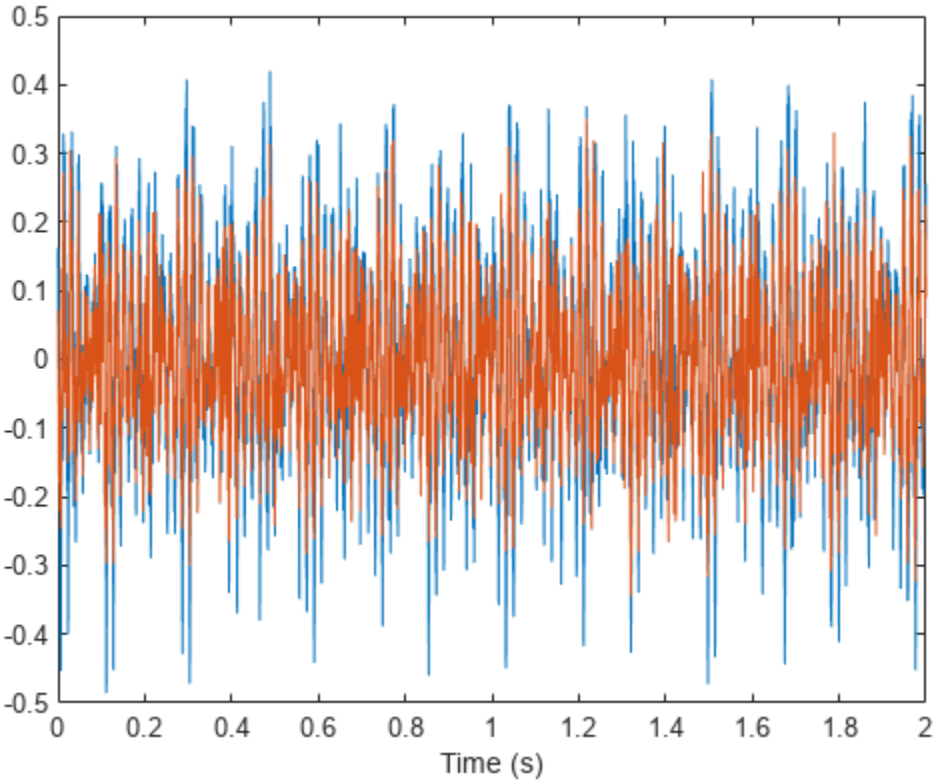
Create a `dsp.AudioFileReader` object to read in an audio signal frame-by-frame. Create an `audioOscillator` object to create a modulation wave. Apply the modulation wave to the audio file.

```
fileReader = dsp.AudioFileReader('Engine-16-44p1-stereo-20sec.wav');

fmod = 10.8 ;

amplitude = 0.15 ;

osc = audioOscillator('sine',fmod, ...
    "DCOffset",0.5, ...
    "Amplitude",amplitude, ...
    "SampleRate",fileReader.SampleRate, ...
    "SamplesPerFrame",fileReader.SamplesPerFrame);

testSignal = [];
while ~isDone(fileReader)
    x = fileReader();
```

```
    testSignal = [testSignal;osc().*fileReader()];
end
```

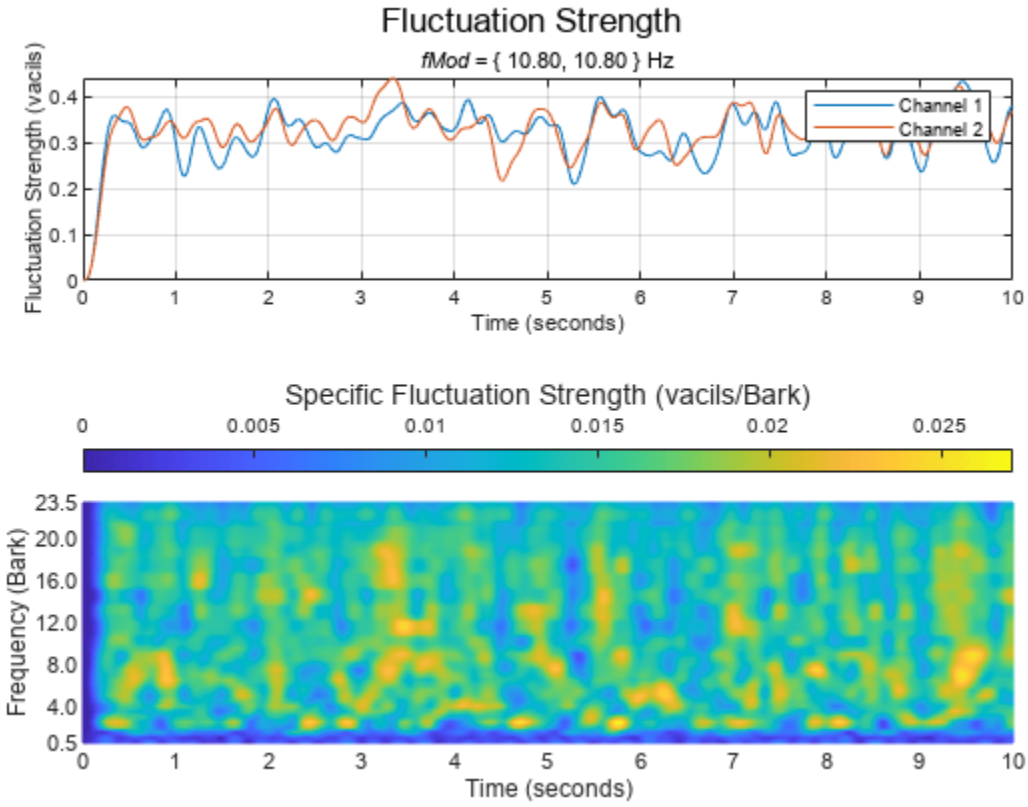
Listen to two seconds of the test signal and plot its waveform.

```
samplesToView = 1:2*fileReader.SampleRate;
sound(testSignal(samplesToView,:),fileReader.SampleRate);

plot(samplesToView/fileReader.SampleRate,testSignal(samplesToView,:))
xlabel('Time (s)')
```



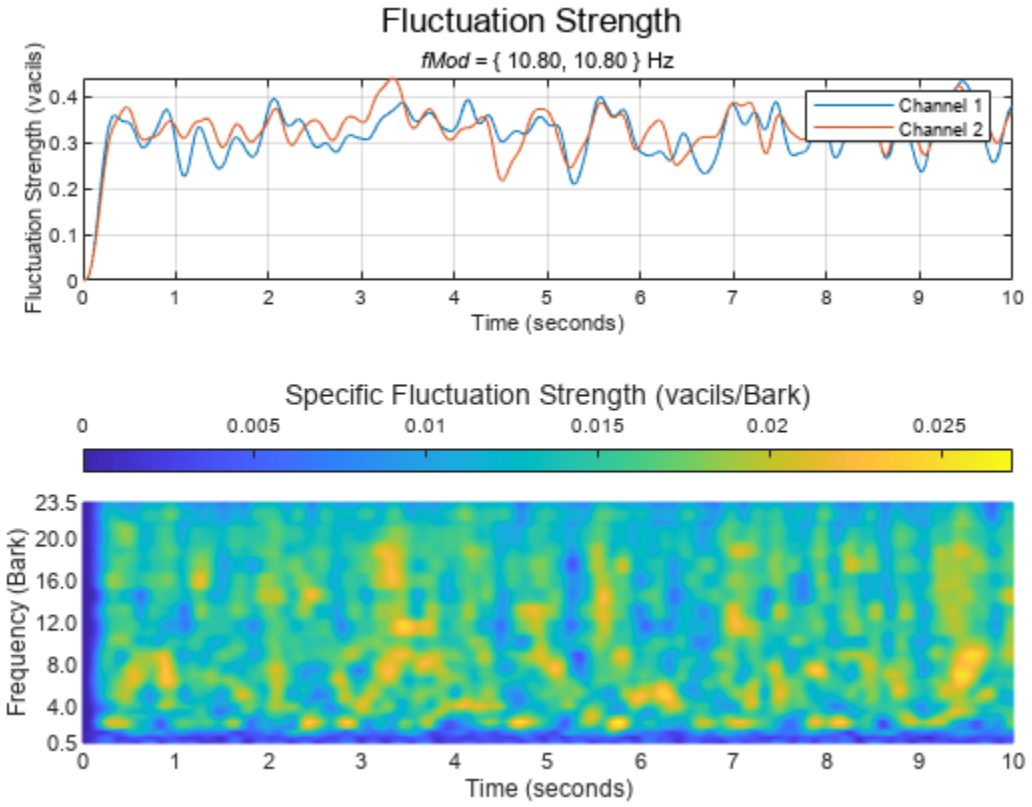
Plot the acoustic fluctuation. The detected frequency of the modulation is displayed textually.

```
acousticFluctuation(testSignal,fileReader.SampleRate);
```



Specify the known modulation frequency and then plot the acoustic fluctuation again.

```
acousticFluctuation(testSignal,fileReader.SampleRate,'ModulationFrequency',fmod)
```



Input Arguments

`audioIn` — Audio input

column vector | two-column matrix

Audio input, specified as a column vector (mono) or matrix with two columns (stereo).

Tip To measure fluctuation strength given any modulation frequency, the recommended minimum signal duration is 10 seconds.

Data Types: `single` | `double`

`fs` — Sample rate (Hz)

positive scalar

Sample rate in Hz, specified as a positive scalar. The recommended sample rate for new recordings is 48 kHz.

Note The minimum acceptable sample rate is 8 kHz.

Data Types: `single` | `double`

`calibrationFactor` — Microphone calibration factor

`sqrt(8)` (default) | positive scalar

Microphone calibration factor, specified as a positive scalar. The default calibration factor corresponds to a full-scale 1 kHz sine wave with a sound pressure level of 100 dB (SPL). To compute the calibration factor specific to your system, use the `calibrateMicrophone` function.

Data Types: `single` | `double`

`specificLoudnessIn` — Specific loudness (sones/Bark)

T-by-240-by-*C*

Specific loudness in sones/Bark, specified as a *T*-by-240-by-*C* array, where:

- *T* is one per 2 ms of the time-varying signal.
- 240 is the number of Bark bins in the domain for specific loudness. The Bark bins are `0.1:0.1:24`.
- *C* is the number of channels.

You can use the `acousticLoudness` function to calculate time-varying specific loudness using this syntax:

```
[~,specificLoudness] = acousticLoudness(audioIn,fs,'TimeVarying',true);
```

Data Types: `single` | `double`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `acousticFluctuation(audioIn,fs,'ModulationFrequency',50)`

`ModulationFrequency` — Known modulation frequency (Hz)

`'auto-detect'` (default) | scalar or two-element vector with values in the range `[0.1,100]`

Known modulation frequency in Hz, specified either `'auto-detect'` or as a scalar or two-element vector with values in the range `[0.1,100]`. If `ModulationFrequency` is set to `'auto-detect'`, then the function limits the search range to between 0.2 and 64 Hz. If the input is mono, then the modulation frequency must be specified as a scalar. If the input is stereo, then the modulation frequency can be specified as either a scalar or two-element vector.

Data Types: `single` | `double` | `char` | `string`

`SoundField` — Sound field

`'free'` (default) | `'diffuse'`

Sound field of audio recording, specified as `'free'` or `'diffuse'`.

Data Types: `char` | `string`

`PressureReference` — Reference pressure (Pa)

`20e-6` (default) | positive scalar

Reference pressure for dB calculation in pascals, specified as a positive scalar. The default value, 20 micropascals, is the common value of air.

Data Types: `single` | `double`

Output Arguments

`fluctuation` — Fluctuation strength (vacil)

K-by-1 | *K*-by-2

Fluctuation strength in vacil, returned as a *K*-by-1 column vector or *K*-by-2 matrix of independent channels. *K* corresponds to the time dimension.

Data Types: `single` | `double`

`specificFluctuation` — Specific fluctuation strength (vacil/Bark)

K-by-47 matrix | *K*-by-47-by-2 array

Specific fluctuation strength in vacil/Bark, returned as a *K*-by-47 matrix or a *K*-by-47-by-2 array. The first dimension of `specificFluctation`, *K*, corresponds to the time dimension and matches the first dimension of `fluctuation`. The second dimension of `specificFluctation`, 47, corresponds to bands on the Bark scale, with centers from 0.5 to 23.5, inclusive, in 0.5 increments. The third dimension of `specificFluctation` corresponds to the number of channels and matches the second dimension of `fluctuation`.

Data Types: `single` | `double`

`fMod` — Dominant modulation frequency (Hz)

scalar (mono input) | 1-by-2 vector (stereo input)

Dominant modulation frequency in Hz, returned as a scalar for mono input or a 1-by-2 vector for stereo input.

Data Types: `single` | `double`

Algorithms

Acoustic fluctuation strength is a perceptual measurement of slow modulations in amplitude or frequency. The acoustic loudness algorithm is described in [1] and implemented in the `acousticLoudness` function. The acoustic fluctuation calculation is described in [2]. The algorithm for acoustic fluctuation is outlined as follows.

$$fluctuation = \frac{0.008 \int_{z=0}^{24} \Delta L \, dz}{\left(\frac{f_{\mathrm{mod}}}{4}\right) + \left(\frac{4}{f_{\mathrm{mod}}}\right)}$$

Where f_{mod} is the detected or known modulation frequency and ΔL is the perceived modulation depth. If the modulation frequency is not specified when calling `acousticFluctuation`, it is auto-detected by peak-picking a frequency-domain representation of the acoustic loudness. The perceived modulation depth, ΔL, is calculated by passing rectified specific loudness bands through ½ octave filters centered around f_{mod}, followed by a lowpass filter to determine the envelope.

Version History

Introduced in R2020b

References

[1] ISO 532-1:2017(E). "Acoustics – Methods for calculating loudness – Part 1: Zwicker method." *International Organization for Standardization*.

[2] Zwicker, Eberhard, and H. Fastl. *Psychoacoustics: Facts and Models*. 2nd updated ed, Springer, 1999.

Extended Capabilities

C/C++ Code Generation

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

See Also

`acousticLoudness` | `acousticSharpness` | `calibrateMicrophone` | `acousticRoughness`

Topics

“Effect of Soundproofing on Perceived Noise Levels”

cepstralCoefficients

Extract cepstral coefficients

Syntax

```
coeffs = cepstralCoefficients(S)
coeffs = cepstralCoefficients(S,Name=Value)
```

Description

`coeffs = cepstralCoefficients(S)` returns the cepstral coefficients over time. The input, `S`, must be a real-valued spectrogram or auditory spectrogram.

`coeffs = cepstralCoefficients(S,Name=Value)` specifies options using one or more name-value arguments.

For example, `coeffs = cepstralCoefficients(S,Rectification="cubic-root")` uses cubic-root rectification to calculate the coefficients.

Examples

Mel Frequency Cepstral Coefficients

Read an audio file into the workspace.

```
[audioIn,fs] = audioread('SpeechDFT-16-8-mono-5secs.wav');
```

Convert the audio signal to a frequency-domain representation using 30 ms windows with 15 ms overlap. Because the input is real and therefore the spectrum is symmetric, you can use just one side of the frequency domain representation without any loss of information. Convert the complex spectrum to the magnitude spectrum: phase information is discarded when calculating mel frequency cepstral coefficients (MFCC).

```
windowLength = round(0.03*fs);
overlapLength = round(0.015*fs);
S = stft(audioIn,"Window",hann(windowLength,"periodic"),"OverlapLength",overlapLength,"FrequencyR
S = abs(S);
```

Design a one-sided frequency-domain mel filter bank. Apply the filter bank to the frequency-domain representation to create a mel spectrogram.

```
filterBank = designAuditoryFilterBank(fs,'FFTLength',windowLength);
melSpec = filterBank*S;
```

Call `cepstralCofficients` with the mel spectrogram to create MFCC.

```
melcc = cepstralCoefficients(melSpec);
```

Gammatone Frequency Cepstral Coefficients

Read an audio signal and convert it to a one-sided magnitude short-time Fourier transform. Use a 50 ms periodic Hamming window with a 10 ms hop.

```
[audioIn,fs] = audioread('NoisySpeech-16-22p5-mono-5secs.wav');

windowLength = round(0.05*fs);
hopLength = round(0.01*fs);
overlapLength = windowLength - hopLength;

S = stft(audioIn,"Window",hamming(windowLength,'periodic'),"OverlapLength",overlapLength,"Freque
S = abs(S);
```

Design a one-sided frequency-domain gammatone filter bank. Apply the filter bank to the frequency-domain representation to create a gammatone spectrogram.

```
filterBank = designAuditoryFilterBank(fs,'FFTLength',windowLength,"FrequencyScale","erb");
gammaSpec = filterBank*S;
```

Call `cepstralCoefficients` with the gammatone spectrogram to create gammatone frequency cepstral coefficients. Use a cubic-root rectification.

```
gammacc = cepstralCoefficients(gammaSpec,"Rectification","cubic-root");
```

Custom Cepstral Coefficients

Cepstral coefficients are commonly used as compact representations of audio signals. Generally, they are calculated after an audio signal is passed through a filter bank and the energy in the individual filters is summed. Researchers have proposed various filter banks based on psychoacoustic experiments (such as mel, Bark, and ERB). Using the `cepstralCoefficients` function, you can define your own custom filter bank and then analyze the resulting cepstral coefficients.

Read in an audio file for analysis.

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

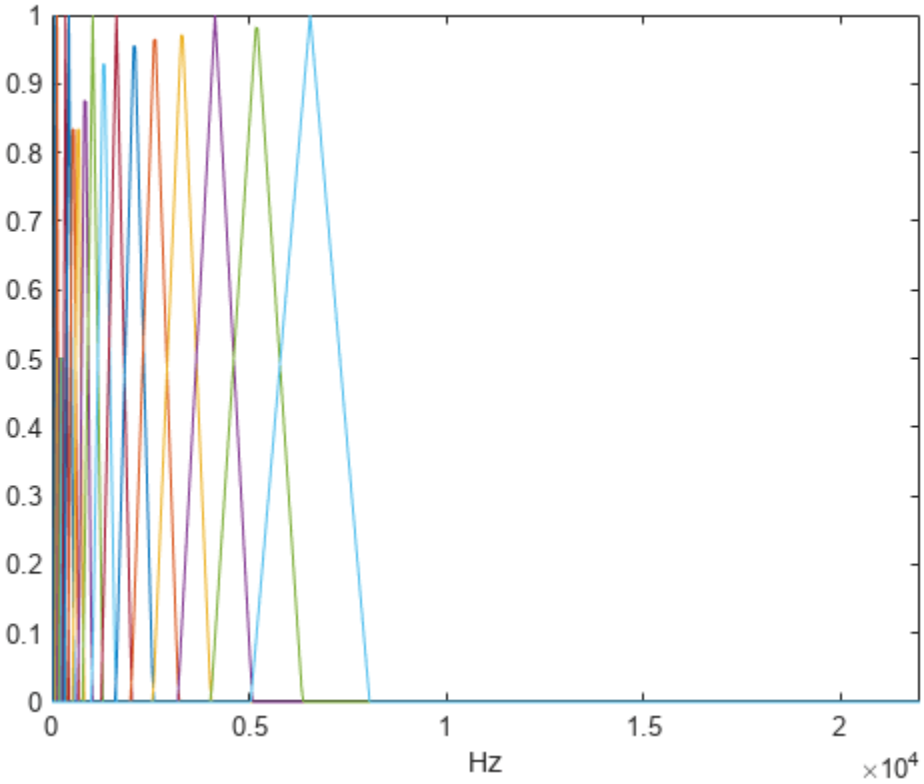
Design a filter bank that consists of 20 triangular filters with band edges over the range 62.5 Hz to 8000 Hz. Spread the filters evenly in the log domain. For simplicity, design the filters in bins. Most popular auditory filter banks are designed in a continuous domain, such as Hz, mel, or Bark, and then warped back to bins.

```
numFilters = 20 ;

filterbankStart = 62.5 ;
filterbankEnd = 8000 ;

numBandEdges = numFilters + 2;
NFFT = 1024;
filterBank = zeros(numFilters,NFFT);

bandEdges = logspace(log10(filterbankStart),log10(filterbankEnd),numBandEdges);
```

```
bandEdgesBins = round((bandEdges/fs)*NFFT) + 1;

for ii = 1:numFilters
     filt = triang(bandEdgesBins(ii+2)-bandEdgesBins(ii));
     leftPad = bandEdgesBins(ii);
     rightPad = NFFT - numel(filt) - leftPad;
     filterBank(ii,:) = [zeros(1,leftPad),filt',zeros(1,rightPad)];
end
```

Plot the filter bank.

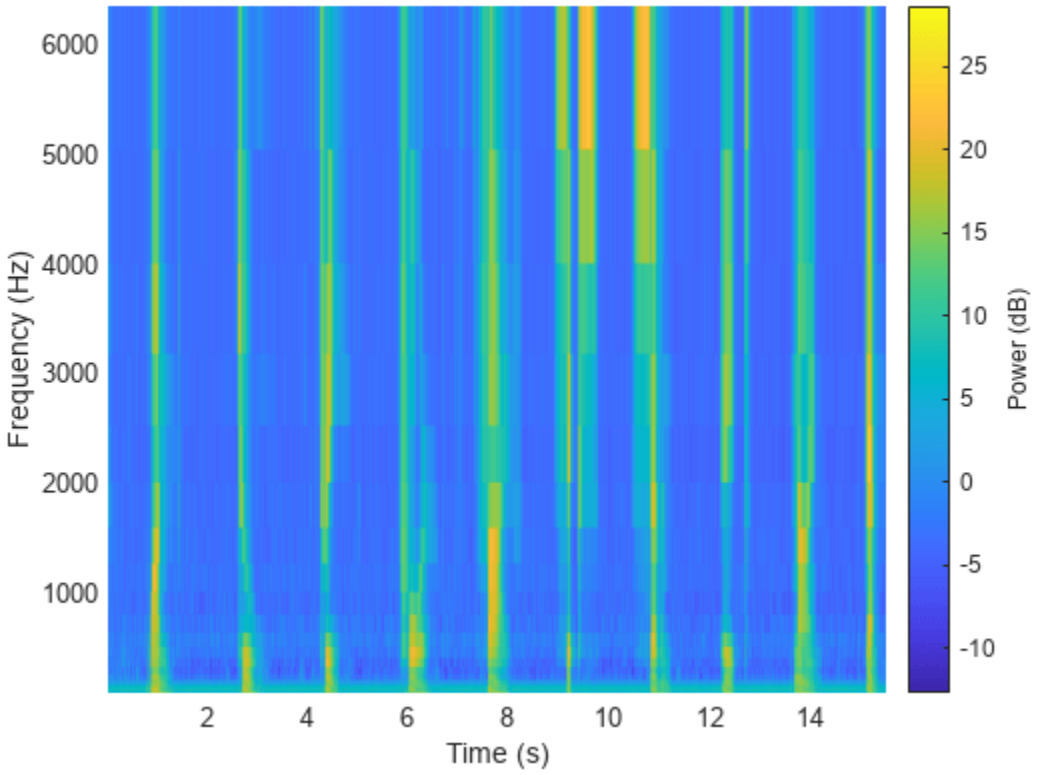
```
frequencyVector = (fs/NFFT)*(0:NFFT-1);
plot(frequencyVector,filterBank');
xlabel('Hz')
axis([0 frequencyVector(NFFT/2) 0 1])
```



Transform the audio signal using the `stft` function, and then apply the custom filter bank. Apply the filter bank to the frequency-domain representation to create a custom auditory spectrogram. Plot the spectrogram.

```
[S,~,t] = stft(audioIn,fs,"Window",hann(NFFT,'periodic'),"FrequencyRange","twosided");
S = abs(S);
spec = filterBank*S;

surf(t,bandEdges(2:end-1),10*log10(spec),'EdgeColor','none')
view([0,90])
axis([t(1) t(end) bandEdges(2) bandEdges(end-1)])
xlabel('Time (s)')
```

```
ylabel('Frequency (Hz)')
c = colorbar;
c.Label.String = 'Power (dB)';
```



Call `cepstralCoefficients` with the custom auditory spectrogram to create custom cepstral coefficients.

```
ccc = cepstralCoefficients(S);
```

Extract Cepstral Coefficients from Streaming Audio

Create a `dsp.AudioFileReader` object to read in audio frame-by-frame. Create a `dsp.AsyncBuffer` object to buffer the input into overlapped frames.

```
fileReader = dsp.AudioFileReader("Ambiance-16-44p1-mono-12secs.wav");
buff = dsp.AsyncBuffer;
```

Design a two-sided mel filter bank that is compatible with 30 ms windows.

```
windowLength = round(0.03*fileReader.SampleRate);
filterBank = designAuditoryFilterBank(fileReader.SampleRate,"FFTLength",windowLength,"OneSided",
```

In an audio stream loop:

1. Read a frame of data from the audio file.

2 Write the frame of data to the buffer.

3 If enough data is available for a hop, read a 30 ms frame of data from the buffer with a 20 ms overlap between frames.

4 Transform the data to a magnitude spectrum.

5 Apply the mel filter bank to create a mel spectrum.

6 Call `cepstralCoefficients` to return the mel frequency cepstral coefficients (MFCC).

```
win = hann(windowLength,'periodic');
overlapLength = round(0.02*fileReader.SampleRate);
hopLength = windowLength - overlapLength;

while ~isDone(fileReader)
    audioIn = fileReader();
    write(buff,audioIn);
    while buff.NumUnreadSamples > hopLength
        x = read(buff,windowLength,overlapLength);
        X = abs(fft(x.*win));
        melSpectrum = filterBank*X;
        melcc = cepstralCoefficients(melSpectrum);
    end
end
```

Input Arguments

S — Spectrogram or auditory spectrogram

matrix | 3-D array

Spectrogram or auditory spectrogram, specified as an *L*-by-*M* matrix or *L*-by-*M*-by-*N* array, where:

- *L* –- Number of frequency bands
- *M* –- Number of frames
- *N* –- Number of channels

Data Types: `single` | `double`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `cepstralCoefficients(S,NumCoeffs=16)`

NumCoeffs — Number of cepstral coefficients returned

13 (default) | positive integer greater than 1

Number of coefficients returned for each window of data, specified as a positive integer greater than 1.

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

`Rectification` — Type of nonlinear rectification
`"log"` (default) | `"cubic-root"` | `"none"`

Type of nonlinear rectification applied prior to the discrete cosine transform, specified as `"log"`, `"cubic-root"`, or `"none"`.

Data Types: `char` | `string`

Output Arguments

`coeffs` — Cepstral coefficients
matrix | 3-D array

Cepstral coefficients, returned as an *M*-by-*B* matrix or *M*-by-*B*-by-*N* array, where:

- *M* –- Number of frames (columns) of the input.
- *B* –- Number of coefficients returned per frame. This is determined by `NumCoeffs`.
- *N* –- Number of channels (pages) of the input.

Data Types: `single` | `double`

Algorithms

Given an auditory spectrogram, the algorithm to extract *N* cepstral coefficients from each individual spectrum comprises the following steps.

1. Rectify the spectrum by applying a logarithm, cubic root, or optionally perform no rectification.
2. Apply the discrete cosine transform (DCT-II) to the rectified spectrum.
3. Return the first *N* coefficients from the cepstral representation.

For more information, see [1].

Version History

Introduced in R2020b

References

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

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

GPU Arrays
Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

See Also

Functions
`mfcc` | `gtcc` | `audioDelta` | `designAuditoryFilterBank` | `melSpectrogram` | `stft`

Blocks
Cepstral Coefficients | MFCC | Audio Delta

Objects
`audioFeatureExtractor`

audioDelta

Compute delta features

Syntax

```
delta = audioDelta(x)
delta = audioDelta(x,deltaWindowLength)
delta = audioDelta(x,deltaWindowLength,initialCondition)
[delta,finalCondition] = audioDelta(x,___)
```

Description

`delta = audioDelta(x)` returns the delta of audio features `x`.

`delta = audioDelta(x,deltaWindowLength)` specifies the delta window length.

`delta = audioDelta(x,deltaWindowLength,initialCondition)` specifies the initial condition of the filter.

`[delta,finalCondition] = audioDelta(x,___)` also returns the final condition of the filter.

Examples

Delta of Audio Features

Read in an audio file.

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

Create an `audioFeatureExtractor` object to extract some spectral features over time from the audio. Call `extract` to extract the audio features.

```
afe = audioFeatureExtractor('SampleRate',fs, ...
    'spectralCentroid',true, ...
    'spectralSlope',true);

audioFeatures = extract(afe,audioIn);
```

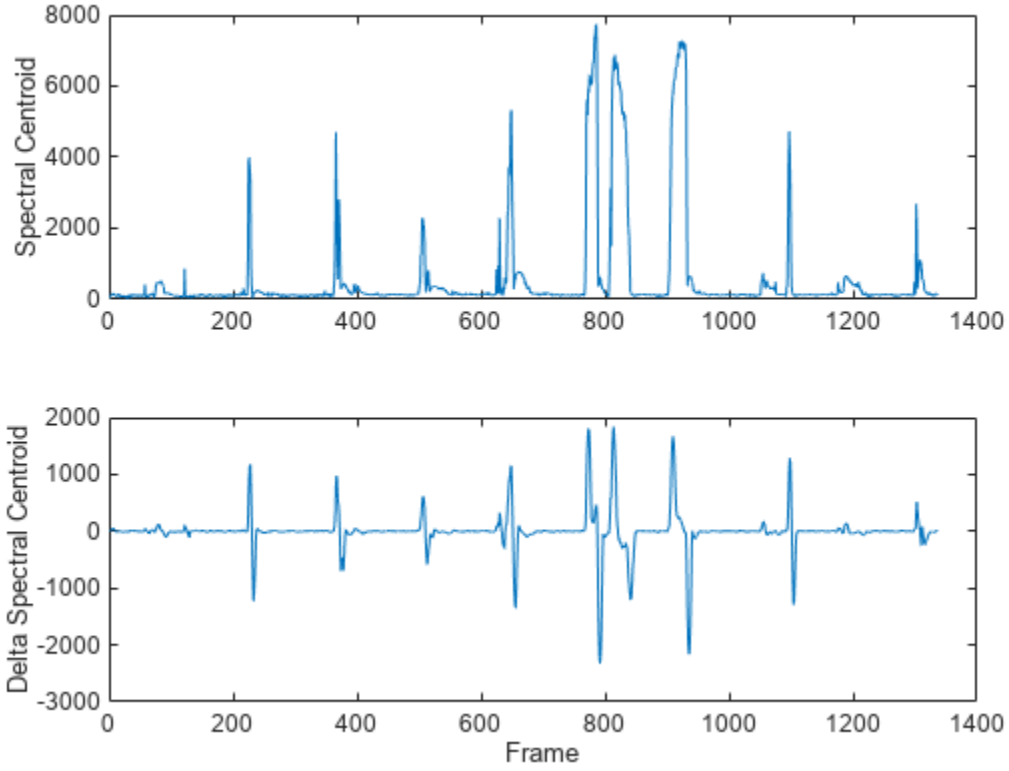
Call `audioDelta` to approximate the first derivative of the spectral features over time.

```
deltaAudioFeatures = audioDelta(audioFeatures);
```

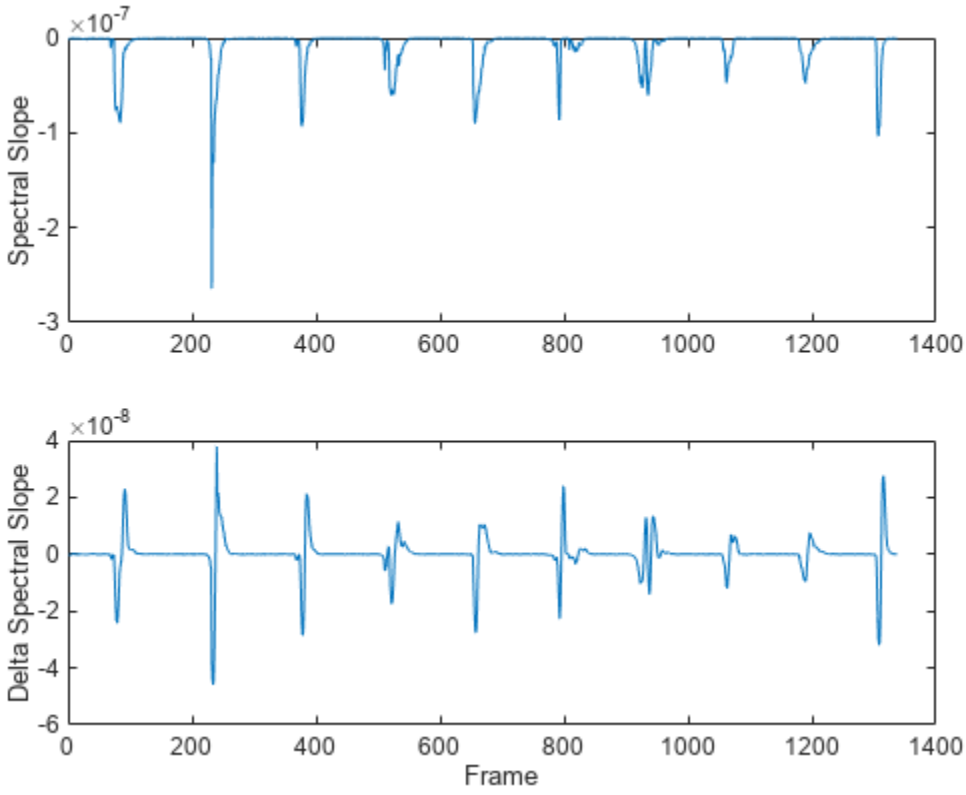
Plot the spectral features and the delta of the spectral features.

```
map = info(afe);
tiledlayout(2,1)
nexttile
plot(audioFeatures(:,map.spectralCentroid))
ylabel('Spectral Centroid')
nexttile
plot(deltaAudioFeatures(:,map.spectralCentroid))
```

```
ylabel('Delta Spectral Centroid')
xlabel('Frame')
```




```
tiledlayout(2,1)
nexttile
plot(audioFeatures(:,map.spectralSlope))
ylabel('Spectral Slope')
nexttile
plot(deltaAudioFeatures(:,map.spectralSlope))
ylabel('Delta Spectral Slope')
xlabel('Frame')
```



Delta and Delta-Delta of MFCC

The delta and delta-delta of mel frequency cepstral coefficients (MFCC) are often used with the MFCC for machine learning and deep learning applications.

Read in an audio file.

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

Use the `designAuditoryFilterBank` function to design a one-sided frequency-domain mel filter bank.

```
analysisWindowLength = round(fs*0.03);
fb = designAuditoryFilterBank(fs,"FFTLength",analysisWindowLength);
```

Use the `stft` function to convert the audio signal to a complex, one-sided frequency-domain representation. Convert the STFT to magnitude and apply the frequency-domain filtering.

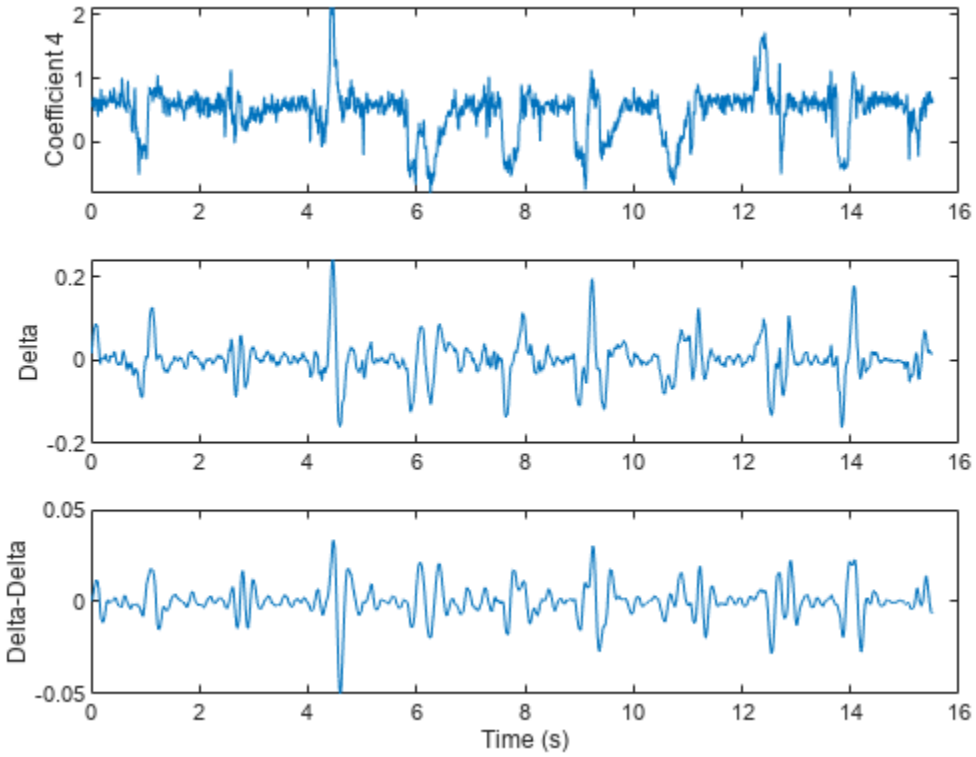
```
[S,~,t] = stft(audioIn,fs,"Window",hann(analysisWindowLength,"periodic"),"FrequencyRange","onesi
auditorySTFT = fb*abs(S);
```

Call the `cepstralCoefficients` function to extract the MFCC.

```
melcc = cepstralCoefficients(auditorySTFT);
```

Call the `audioDelta` function to compute the delta MFCC. Call `audioDelta` again to compute the delta-delta MFCC. Plot the results.

```
deltaWindowLength = 21;

melccDelta = audioDelta(melcc,deltaWindowLength);
melccDeltaDelta = audioDelta(melccDelta,deltaWindowLength);

coefficientToDisplay = 4;

tiledlayout(3,1)
nexttile
plot(t,melcc(:,coefficientToDisplay+1))
ylabel('Coefficient ' + string(coefficientToDisplay))
nexttile
plot(t,melccDelta(:,coefficientToDisplay+1))
ylabel('Delta')
nexttile
plot(t,melccDeltaDelta(:,coefficientToDisplay+1))
xlabel('Time (s)')
ylabel('Delta-Delta')
```



Delta of Streaming Signals

You can calculate the delta of streaming signals by passing state in and out of the `audioDelta` function.

Create a `dsp.AudioFileReader` object to read an audio file frame-by-frame. Create an `audioDeviceWriter` object to write audio to your speaker. Create a `timescope` object to visualize the change in harmonic ratio over time.

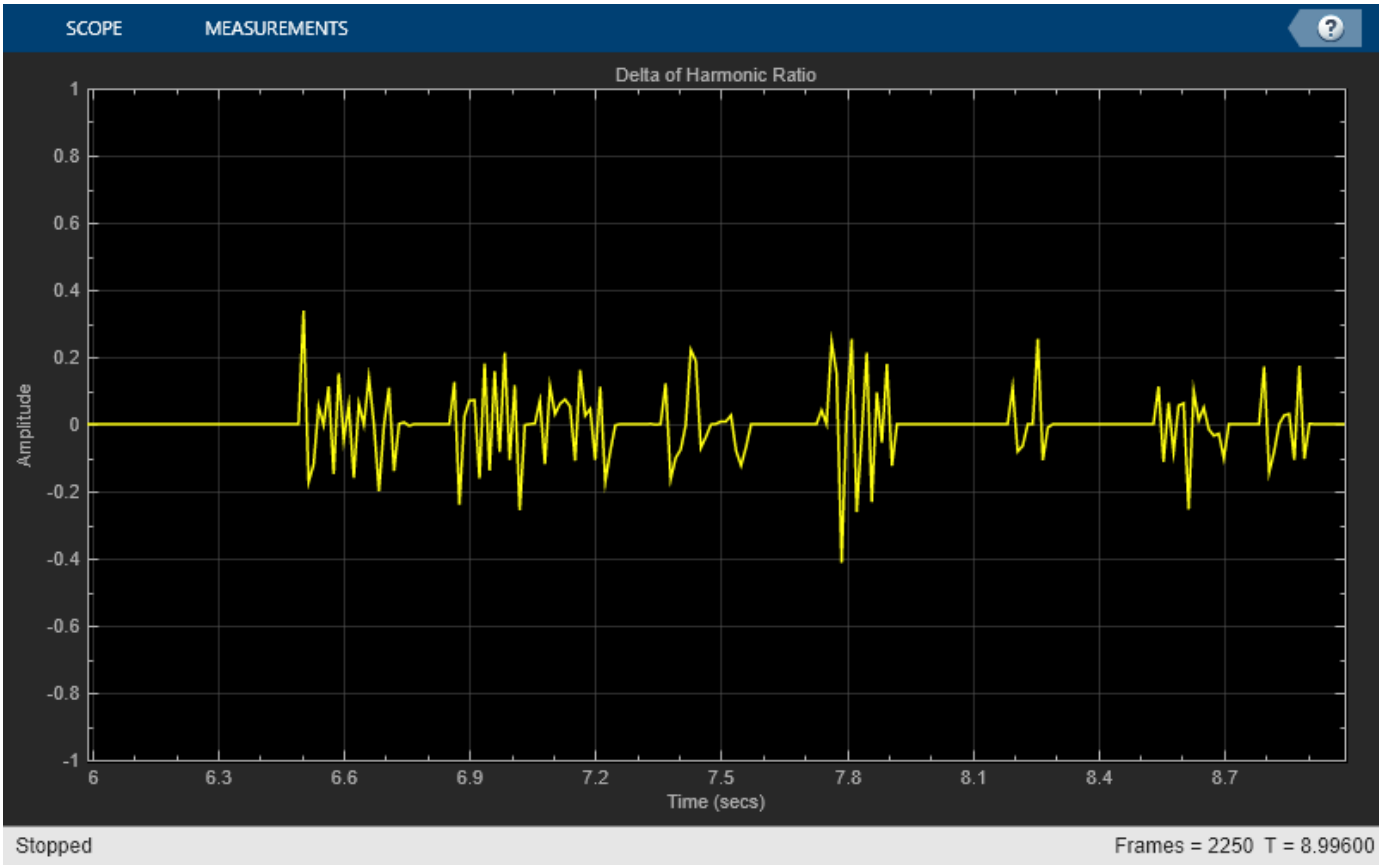
```
fileReader = dsp.AudioFileReader("FemaleSpeech-16-8-mono-3secs.wav","SamplesPerFrame",32,"PlayCou
deviceWriter = audioDeviceWriter("SampleRate",fileReader.SampleRate);
scope = timescope("SampleRate",fileReader.SampleRate/fileReader.SamplesPerFrame, ...
    "TimeSpanSource","Property", ...
    "TimeSpan",3, ...
    "YLimits",[-1,1], ...
    "Title","Delta of Harmonic Ratio");
```

While the audio file has unread frames of data:

1. Read a frame from the audio file
2. Calculate the harmonic ratio of that frame
3. Calculate the delta of the harmonic ratio
4. Write the audio frame to your speaker
5. Write the change in the harmonic ratio to your scope

On each call to `audioDelta`, overwrite the previous state. Initialize the state using an empty array.

```
z = [];
while ~isDone(fileReader)
    audioIn = fileReader();

    hr = harmonicRatio(audioIn,fileReader.SampleRate,"Window",hann(fileReader.SamplesPerFrame,'pe

    [deltaHR, z] = audioDelta(hr,5,z);

    deviceWriter(audioIn);

    scope(deltaHR)
end
release(scope)
```



Input Arguments

x — Audio features

vector | matrix | array

Audio features, specified as a vector, matrix, or multi-dimensional array. The delta computation operates along the first dimension. All other dimensions are treated as independent channels.

Data Types: `single` | `double`

`deltaWindowLength` — Window length over which to calculate delta

9 (default) | odd integer greater than 2

Window length over which to calculate delta, specified as an odd integer greater than 2.

Data Types: `single` | `double`

`initialCondition` — Initial condition of filter

`[]` (default) | vector | matrix | array

Initial condition of the filter used to calculate the delta, specified as a vector, matrix, or multi-dimensional array. The first dimension of `initialCondition` must equal `deltaWindowLength-1`. The remaining dimensions of `initialCondition` must match the remaining dimensions of the input `x`. The default initial condition, `[]`, is equivalent to initializing the state with all zeros.

Data Types: `single` | `double`

Output Arguments

`delta` — Delta of audio features

vector | matrix | array

Delta of audio features, returned as a vector, matrix, or multi-dimensional array with the same dimensions as the input `x`.

Data Types: `single` | `double`

`finalCondition` — Final condition of filter

vector | matrix | array

Final condition of filter, returned as a vector, matrix, or multi-dimensional array. The final condition is returned as the same size as the `initialCondition`.

Data Types: `single` | `double`

Algorithms

The delta of an audio feature x is a least-squares approximation of the local slope of a region centered on sample $x(k)$, which includes M samples before the current sample and M samples after the current sample.

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

The delta window length defines the length of the region from $-M$ to M.

For more information, see [1].

Version History

Introduced in R2020b

References

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

Extended Capabilities

C/C++ Code Generation

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

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

See Also

Functions
`mfcc` | `gtcc` | `cepstralCoefficients` | `designAuditoryFilterBank` | `stft`

Blocks
Audio Delta | Cepstral Coefficients | MFCC

Objects
`audioFeatureExtractor`

Live Editor Tasks
Extract Audio Features

showaudioblockdatatypetable

Simulink block data type support table

Syntax

```
showaudioblockdatatypetable
```

Description

`showaudioblockdatatypetable` shows a table of characteristics for Audio Toolbox blocks. The table lists capabilities and limitations about blocks, such as support for code generation and variable-sized input.

Examples

Show Block Characteristics for Audio Toolbox™

Show a table of Audio Toolbox™ block characteristics. The table opens in a separate window.

```
showaudioblockdatatypetable
```

```
Loading Audio Toolbox Library.
```

Version History

Introduced in R2016a

See Also

Topics

“Real-Time Audio in Simulink”

audioPluginGridLayout

Specify layout for audio plugin UI

Syntax

```
gridLayout = audioPluginGridLayout
gridLayout = audioPluginGridLayout(Name,Value)
```

Description

`gridLayout = audioPluginGridLayout` creates an object that specifies the layout grid for an audio plugin graphical user interface. Use the plugin grid layout object, `gridLayout`, as an argument to `audioPluginInterface` in your plugin class definition. `audioPluginGridLayout` specifies only the grid. The placement of individual graphical elements is specified using `audioPluginParameter`.

To learn how to design a graphic user interface, see “Design User Interface for Audio Plugin”.

For example plugins, see “Audio Plugin Example Gallery”.

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

Examples

Use Default Audio Plugin Grid Layout

The default audio plugin grid layout specifies a 2-by-2 grid. Call `audioPluginGridLayout` with no arguments to view the default settings.

```
audioPluginGridLayout
```

```
ans =

  audioPluginGridLayout with properties:

        RowHeight: [100 100]
      ColumnWidth: [100 100]
        RowSpacing: 10
     ColumnSpacing: 10
           Padding: [10 10 10 10]
```

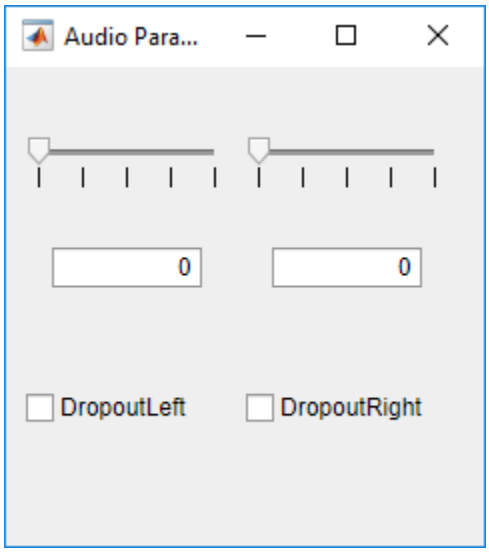
`noisifyClassic` uses a default grid layout by passing `audioPluginGridLayout`, without any arguments, to `audioPluginInterface`. When you use `audioPluginGridLayout`, you must specify the position of each `audioPluginParameter` on the grid using `Layout`. Display names corresponding to parameters occupy cells on the grid also. The default grid contains only four cells and `noisifyClassic` has four parameters, so you must set `DisplayNameLocation` to `none` to fit

all elements on the grid. `audioPluginGridLayout` is passed to the `audioPluginInterface`. Save `noisifyClassic` to your current folder.

```
classdef noisifyClassic < audioPlugin
    properties
        DropoutLeft = false
        DropoutRight = false
        NoiseLeftGain = 0
        NoiseRightGain = 0
    end
    properties (Constant)
        PluginInterface = audioPluginInterface( ...
            audioPluginParameter('DropoutLeft', ...
                'Layout',[2,1], ...
                'DisplayNameLocation','none'), ...
            audioPluginParameter('DropoutRight', ...
                'Layout',[2,2], ...
                'DisplayNameLocation','none'), ...
            audioPluginParameter('NoiseLeftGain', ...
                'Layout',[1,1], ...
                'DisplayNameLocation','none'), ...
            audioPluginParameter('NoiseRightGain', ...
                'Layout',[1,2], ...
                'DisplayNameLocation','none'), ...
            ...
            audioPluginGridLayout)
    end
    methods
        function out = process(plugin,in)
            r = size(in,1);
            dropRate = 0.1;

            if plugin.DropoutLeft
                idx = randperm(r,round(r*dropRate));
                in(idx,1) = 0;
            end
            if plugin.DropoutRight
                idx = randperm(r,round(r*dropRate));
                in(idx,2) = 0;
            end

            in(:,1) = in(:,1) + plugin.NoiseLeftGain*(2*rand(r,1,'like',in)-1);
            in(:,2) = in(:,2) + plugin.NoiseRightGain*(2*rand(r,1,'like',in)-1);

            out = in;
        end
    end
end
```

You can quickly iterate on your UI design by using `parameterTuner` to visualize the plugin UI. Call `parameterTuner` on `noisifyClassic`.

```
parameterTuner(noisifyClassic)
```



Design Audio Plugin Grid Layout

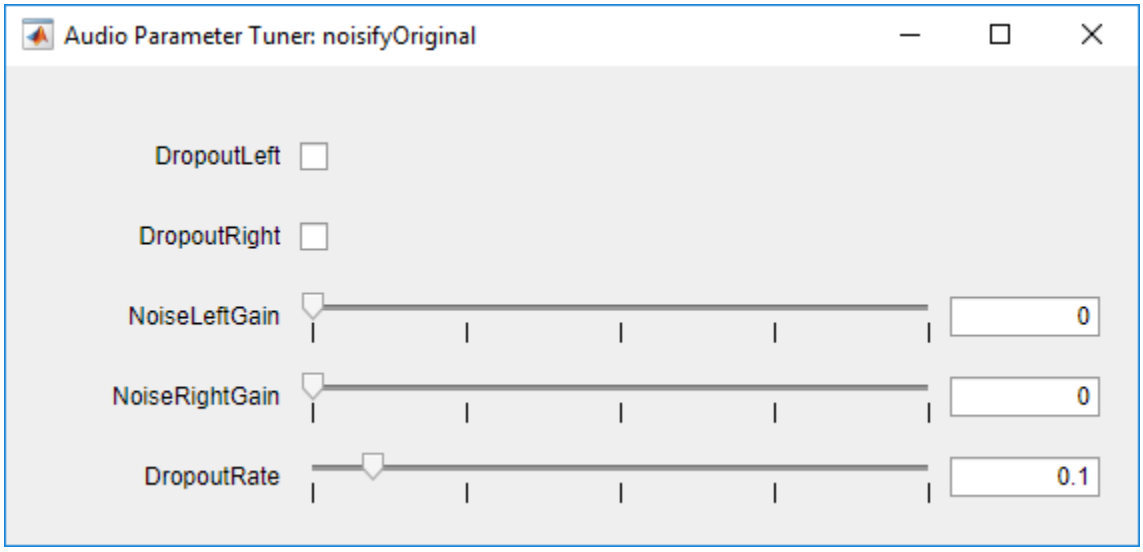
The example plugin, `noisify`, adds noise to your audio signal channel-wise at a specified gain (per channel) and dropout rate.

```
classdef noisifyOriginal < audioPlugin
    properties
        DropoutLeft = false;
        DropoutRight = false;
        NoiseLeftGain = 0;
        NoiseRightGain = 0;
        DropoutRate = 0.1;
    end
    properties (Constant)
        PluginInterface = audioPluginInterface(...
            audioPluginParameter('DropoutLeft'), ...
            audioPluginParameter('DropoutRight'), ...
            audioPluginParameter('NoiseLeftGain'), ...
            audioPluginParameter('NoiseRightGain'), ...
            audioPluginParameter('DropoutRate'))
    end
    methods
        function out = process(plugin,in)
            r = size(in,1);

            if plugin.DropoutLeft
                idx = randperm(r,round(r*plugin.DropoutRate));
                in(idx,1) = 0;
            end
            if plugin.DropoutRight
                idx = randperm(r,round(r*plugin.DropoutRate));
                in(idx,2) = 0;
            end

            in(:,1) = in(:,1) + plugin.NoiseLeftGain*randn(r,1,'like',in);
```

```
            in(:,2) = in(:,2) + plugin.NoiseRightGain*randn(r,1,'like',in);

            out = in;
        end
    end
end
```

To see the corresponding UI for the plugin, call `parameterTuner` with the plugin. When you generate an audio plugin and deploy it to a DAW, the DAW uses a default UI that is similar to the default UI of `parameterTuner`.

```
parameterTuner(noisifyOriginal)
```



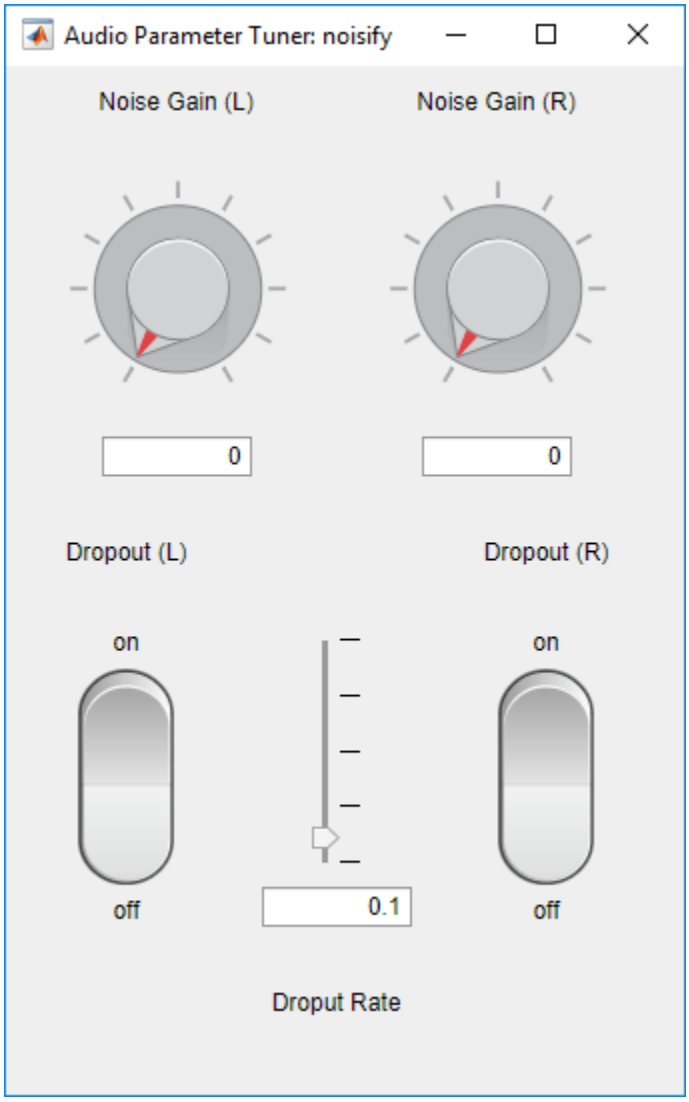
You can create a more intuitive and visually pleasing UI using `audioPluginInterface`, `audioPluginGridLayout`, and `audioPluginParameter`. For example, to create a more intuitive UI for `noisyOriginal`, you could update the `audioPluginInterface` as follows:

```
classdef noisify < audioPlugin
    properties
        DropoutLeft = false;
        DropoutRight = false;
        NoiseLeftGain = 0;
        NoiseRightGain = 0;
        DropoutRate = 0.1;
    end
    properties (Constant)
        PluginInterface = audioPluginInterface(...
            audioPluginParameter('DropoutLeft', ...
                'Layout',[4,1], ...
                'DisplayName','Dropout (L)', ...
                'DisplayNameLocation','above', ...
                'Style','vrocker'), ...
            audioPluginParameter('DropoutRight', ...
                'Layout',[4,4], ...
                'DisplayName','Dropout (R)', ...
                'DisplayNameLocation','above', ...
```

```
                'Style','vrocker'), ...
            audioPluginParameter('NoiseLeftGain', ...
                'DisplayName','Noise Gain (L)', ...
                'Layout',[2,1;2,2], ...
                'DisplayNameLocation','above', ...
                'Style','rotaryknob'), ...
            audioPluginParameter('NoiseRightGain', ...
                'Layout',[2,3;2,4], ...
                'DisplayName','Noise Gain (R)', ...
                'DisplayNameLocation','above', ...
                'Style','rotaryknob'), ...
            audioPluginParameter('DropoutRate', ...
                'Layout',[4,2;4,3], ...
                'DisplayName','Droput Rate', ...
                'DisplayNameLocation','below', ...
                'Style','vslider'), ...
            ...
            audioPluginGridLayout( ...
                'RowHeight',[15,150,15,150,15], ...
                'ColumnWidth',[100,40,40,100], ...
                'RowSpacing',30))
    end
    methods
        function out = process(plugin,in)
            r = size(in,1);

            if plugin.DropoutLeft
                idx = randperm(r,round(r*plugin.DropoutRate));
                in(idx,1) = 0;
            end
            if plugin.DropoutRight
                idx = randperm(r,round(r*plugin.DropoutRate));
                in(idx,2) = 0;
            end

            in(:,1) = in(:,1) + plugin.NoiseLeftGain*randn(r,1,'like',in);
            in(:,2) = in(:,2) + plugin.NoiseRightGain*randn(r,1,'like',in);

            out = in;
        end
    end
end
```

You can quickly iterate on your UI design by using `parameterTuner` to visualize incremental changes. Call `parameterTuner` on `noisify`. When you generate an audio plugin and deploy it to a DAW, the DAW uses the enhanced UI.

```
parameterTuner(noisify)
```



Input Arguments

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `'RowHeight', [50,200,150]` species a grid with three rows. The first row is 50 pixels high, the second row is 200 pixels high, and the third row is 150 pixels high.

`RowHeight` — Height of each row (pixels)
`[100, 100]` (default) | row vector of positive integers

Height in pixels of each row in the grid, specified as a comma-separated pair consisting of `'RowHeight'` and a row vector of positive integers.

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

`ColumnWidth` — Width of each column (pixels)
`[100, 100]` (default) | row vector of positive integers

Width in pixels of each column in the grid, specified as a comma-separated pair consisting of `'ColumnWidth'` and a row vector of positive integers.

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

`RowSpacing` — Distance between rows (pixels)
`10` (default) | nonnegative integer

Distance between rows in pixels, specified as a comma-separated pair consisting of `'RowSpacing'` and a nonnegative integer.

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

`ColumnSpacing` — Distance between columns (pixels)
`10` (default) | nonnegative integer

Distance between columns in pixels, specified as a comma-separated pair consisting of `'ColumnSpacing'` and a nonnegative integer.

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

`Padding` — Padding around the outer perimeter of grid (pixels)
`[10, 10, 10, 10]` (default) | [left, bottom, right, top]

Padding around the outer perimeter of the grid in pixels, specified as a comma-separated pair consisting of `'Padding'` and a four-element row vector of nonnegative integers. The elements of the vector are interpreted as [left, bottom, right, top], where:

- left –- Distance in pixels from the left edge of the grid to the left edge of the parent container.
- bottom –- Distance in pixels from the bottom edge of the grid to the bottom edge of the parent container.
- right –- Distance in pixels from the right edge of the grid to the right edge of the parent container.
- top –- Distance in pixels from the top edge of the grid to the top edge of the parent container.

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

Version History

Introduced in R2019b

See Also

`audioPlugin` | `audioPluginSource` | `audioPluginInterface` | `validateAudioPlugin` | `generateAudioPlugin` | `audioPluginParameter` | **Audio Test Bench** | `parameterTuner`

Topics
“Design User Interface for Audio Plugin”
“Audio Plugin Example Gallery”
“Audio Plugins in MATLAB”

“Export a MATLAB Plugin to a DAW”

pinknoise

Generate pink noise

Syntax

```
X = pinknoise(n)
X = pinknoise(sz1,sz2)
X = pinknoise(sz)
X = pinknoise(___,typename)
X = pinknoise(___,'like',p)
```

Description

`X = pinknoise(n)` returns a pink noise column vector of length `n`.

`X = pinknoise(sz1,sz2)` returns a `sz1`-by-`sz2` matrix. Each channel (column) of the output `X` is an independent pink noise signal.

`X = pinknoise(sz)` returns a vector or matrix with dimensions defined by the elements of vector `sz`. `sz` must be a one- or two-element row vector of positive integers. Each channel (column) of the output `X` is an independent pink noise signal.

`X = pinknoise(___,typename)` returns an array of pink noise of data type `typename`. The `typename` input can be either `'single'` or `'double'`. You can combine `typename` with any of the input arguments in the previous syntaxes.

`X = pinknoise(___,'like',p)` returns an array of pink noise like `p`. You can specify either `typename` or `'like'`, but not both.

Examples

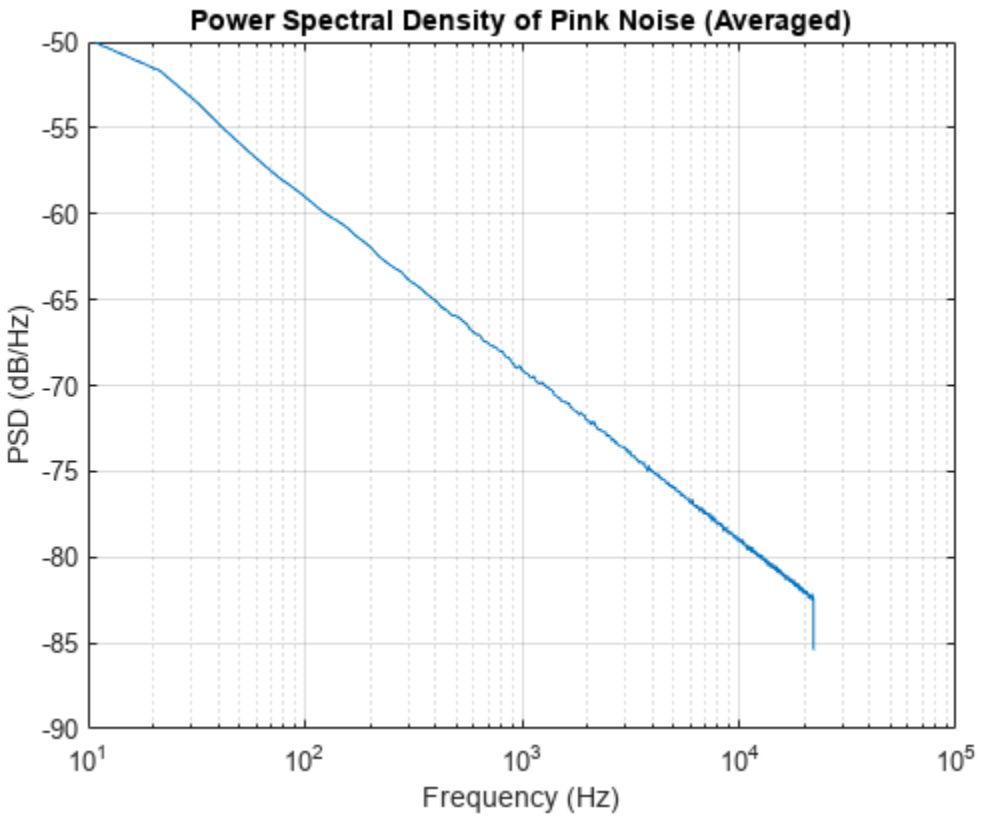
Generate Pink Noise

Generate 100 seconds of pink noise with a sample rate of 44.1 kHz.

```
fs = 44.1e3;
duration = 100;

y = pinknoise(duration*fs);
```

Plot the average power spectral density (PSD) of the generated pink noise.

```
[~,freqVec,~,psd] = spectrogram(y,round(0.05*fs),[],[],fs);
meanPSD = mean(psd,2);

semilogx(freqVec,db(meanPSD,"power"))
xlabel('Frequency (Hz)')
ylabel('PSD (dB/Hz)')
title('Power Spectral Density of Pink Noise (Averaged)')
grid on
```



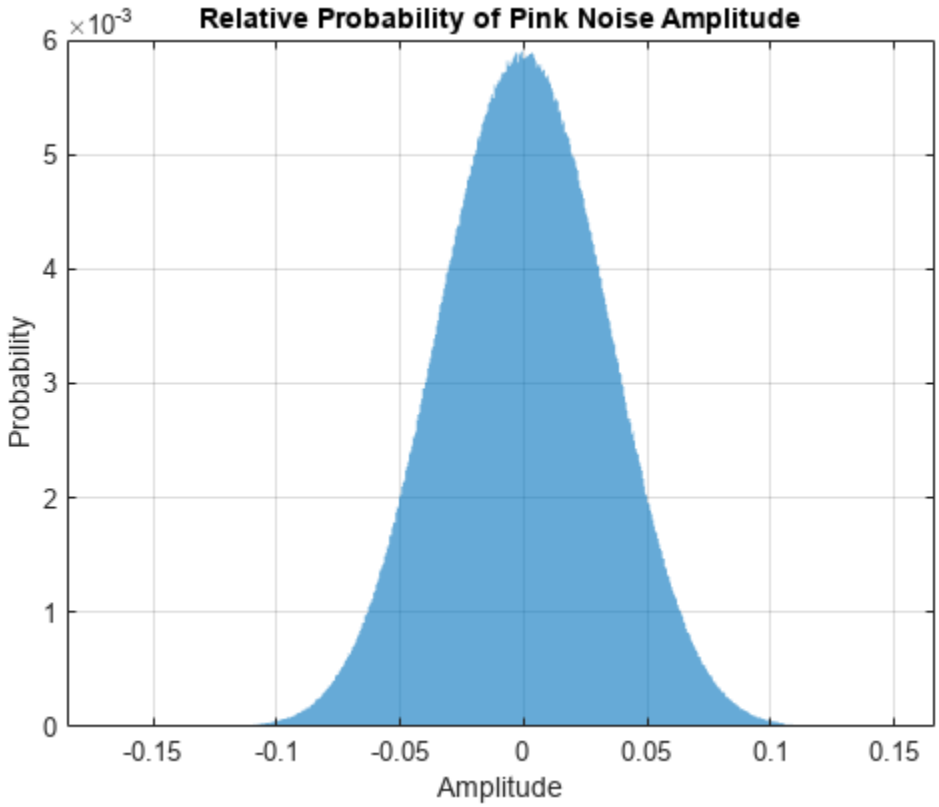
Amplitude Distribution of Pink Noise

Generate 500 seconds of pink noise with a sample rate of 16 kHz.

```
fs = 16e3;
duration = 500;

y = pinknoise(duration*fs);
```

Plot the relative probability of the pink noise amplitude. The amplitude is always bounded between −1 and 1.

```
histogram(y,"Normalization","probability","EdgeColor","none")
xlabel("Amplitude")
ylabel("Probability")
title("Relative Probability of Pink Noise Amplitude")
grid on
```



Generate Multiple Independent Channels of Pink Noise

Create a 5 second stereo pink noise signal with a 48 kHz sample rate.

```
fs = 48e3;
duration = 5;
numChan = 2;

pn = pinknoise(duration*fs,numChan);
```

Listen to the stereo pink noise signal.

```
sound(pn,fs)
```

Channels of the pink noise function are generated independently. Note that the off-diagonal correlation coefficients are close to zero (uncorrelated).

```
R = corrcoef(pn(:,1),pn(:,2))
```

```
R = 2×2

    1.0000   -0.0190
   -0.0190    1.0000
```

Correlated and uncorrelated pink noise have different psychoacoustic effects. When the noise is correlated, the sound is less ambient and more centralized. To listen to correlated pink noise, send a single channel of the pink noise signal to your stereo device. The effect is most pronounced when using headphones.

```
sound([pn(:,1),pn(:,1)],fs)
```

Add Pink Noise to Audio Signal

Read in an audio file.

```
[audioIn,fs] = audioread("MainStreetOne-16-16-mono-12secs.wav");
```

Create a pink noise signal of the same size and data type as `audioIn`.

```
noise = pinknoise(size(audioIn),'like',audioIn);
```

Add the pink noise to the audio signal and then listen to the first 5 seconds.

```
noisyMainStreet = noise + audioIn;
sound(noisyMainStreet(1:fs*5,:),fs)
```

The `pinknoise` function generates an approximate −29.5 dB signal level, which is close to the power of the audio signal.

```
noisePower = sum(noise.^2,1)/size(noise,1);
signalPower = sum(audioIn.^2,1)/size(audioIn,1);
snr = 10*log10(signalPower./noisePower)
```

```
snr = 1.9791
```

```
noisePowerdB = 10*log10(noisePower)
```

```
noisePowerdB = -29.6665
```

```
signalPowerdB = 10*log10(signalPower)
```

```
signalPowerdB = -27.6874
```

Mix the input audio with the generated pink noise at an 8 dB SNR.

```
desiredSNR = 8;
scaleFactor = sqrt(signalPower./(noisePower*(10^(desiredSNR/10))));

noise = noise.*scaleFactor;
```

Verify the resulting SNR is 8 dB and then listen to the first 5 seconds.

```
noisePower = sum(noise.^2,1)/size(noise,1);
snr = 10*log10(signalPower./noisePower)
```

```
snr = 8.0000
```

```
noisyMainStreet = noise + audioIn;
sound(noisyMainStreet(1:fs*5,:),fs)
```

Input Arguments

`n` — Number of rows of pink noise

nonnegative integer

Number of rows of pink noise, specified as a nonnegative integer.

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

`sz1,sz2` — Size of each dimension (as separate arguments)

nonnegative integers

Size of each dimension, specified as a nonnegative integer or two separate arguments of nonnegative integers.

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

`sz` — Size of each dimension (as a row vector)

one- or two-element row vector of nonnegative integers

Size of each dimension, specified as a one- or two-element row vector of nonnegative integers. Each element of this vector indicates the size of the corresponding dimension.

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

`typename` — Data type to create

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

Data type to create, specified as `'double'` or `'single'`.

Data Types: `char` | `string`

`p` — Prototype of array to create

numeric array

Prototype of array to create, specified as a numeric array. The generated pink noise is the same data type as `p`.

Data Types: `single` | `double`

Output Arguments

`X` — Pink noise

column vector | matrix

Pink noise, returned as a column vector or matrix of independent channels.

Data Types: `single` | `double`

Tips

- The concatenation of multiple pink noise vectors does not result in pink noise. For streaming applications, use `dsp.ColoredNoise`.

Algorithms

Pink noise is generated by passing uniformly distributed random numbers through a series of randomly initiated SOS filters. The resulting pink noise amplitude distribution is quasi-Gaussian and bounded between −1 and 1. The resulting pink noise power spectral density (PSD) is inversely proportional to frequency:

$$S(f) \propto \frac{1}{f}$$

Version History

Introduced in R2019b

Extended Capabilities

C/C++ Code Generation

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

See Also

`dsp.ColoredNoise` | `rng` | `rand`

stretchAudio

Time-stretch audio

Syntax

```
audioOut = stretchAudio(audioIn,alpha)
audioOut = stretchAudio(audioIn,alpha,Name,Value)
```

Description

`audioOut = stretchAudio(audioIn,alpha)` applies time scale modification (TSM) on the input audio by the TSM factor `alpha`.

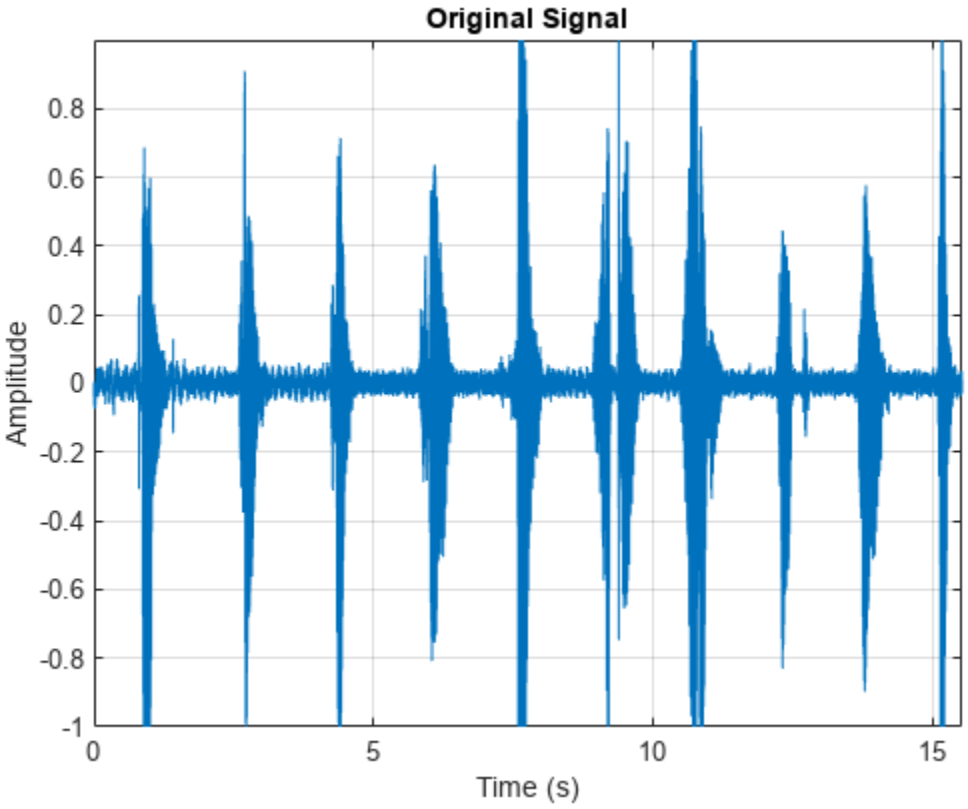
`audioOut = stretchAudio(audioIn,alpha,Name,Value)` specifies options using one or more `Name,Value` pair arguments.

Examples

Apply TSM

Read in an audio signal. Listen to the audio signal and plot it over time.

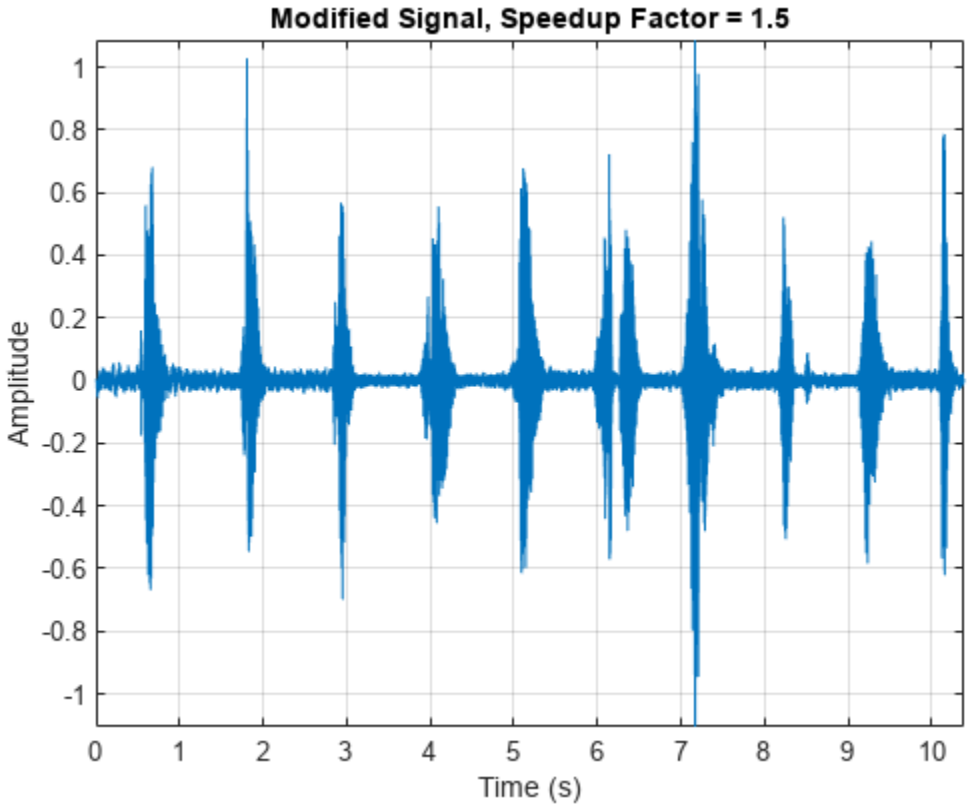
```
[audioIn,fs] = audioread("Counting-16-44p1-mono-15secs.wav");

t = (0:size(audioIn,1)-1)/fs;
plot(t,audioIn)
xlabel('Time (s)')
ylabel('Amplitude')
title('Original Signal')
axis tight
grid on
```




```
sound(audioIn,fs)
```

Use `stretchAudio` to apply a 1.5 speedup factor. Listen to the modified audio signal and plot it over time. The sample rate remains the same, but the duration of the signal has decreased.

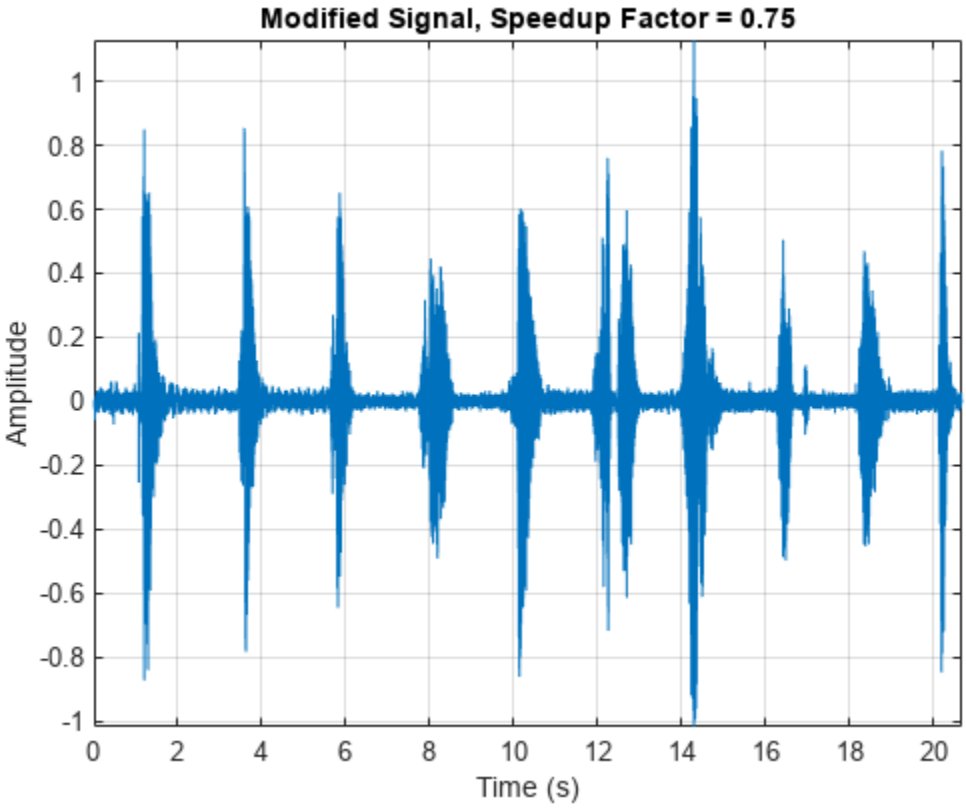
```
audioOut = stretchAudio(audioIn,1.5);

t = (0:size(audioOut,1)-1)/fs;
plot(t,audioOut)
xlabel('Time (s)')
ylabel('Amplitude')
title('Modified Signal, Speedup Factor = 1.5')
axis tight
grid on
```




```
sound(audioOut,fs)
```

Slow down the original audio signal by a 0.75 factor. Listen to the modified audio signal and plot it over time. The sample rate remains the same as the original audio, but the duration of the signal has increased.

```
audioOut = stretchAudio(audioIn,0.75);

t = (0:size(audioOut,1)-1)/fs;
plot(t,audioOut)
xlabel('Time (s)')
ylabel('Amplitude')
title('Modified Signal, Speedup Factor = 0.75')
axis tight
grid on
```



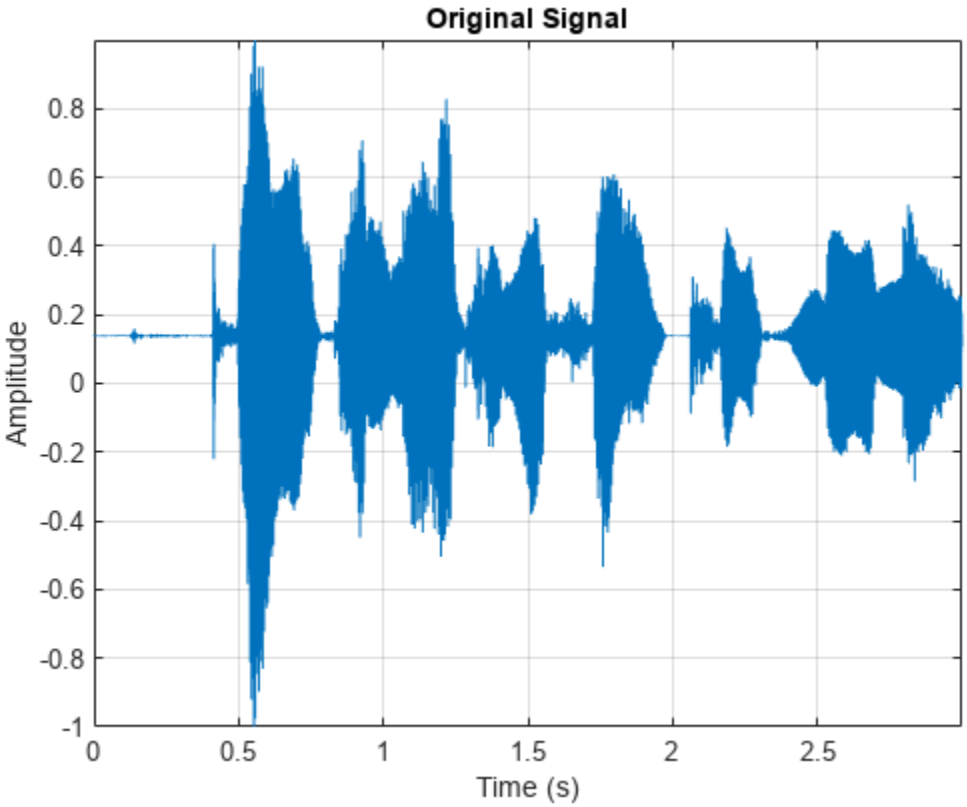

```
sound(audioOut,fs)
```

Apply TSM to Frequency-Domain Audio

`stretchAudio` supports TSM on frequency-domain audio when using the default vocoder method. Applying TSM to frequency-domain audio enables you to reuse your STFT computation for multiple TSM factors.

Read in an audio signal. Listen to the audio signal and plot it over time.

```
[audioIn,fs] = audioread('FemaleSpeech-16-8-mono-3secs.wav');

sound(audioIn,fs)

t = (0:size(audioIn,1)-1)/fs;
plot(t,audioIn)
xlabel('Time (s)')
ylabel('Amplitude')
title('Original Signal')
axis tight
grid on
```

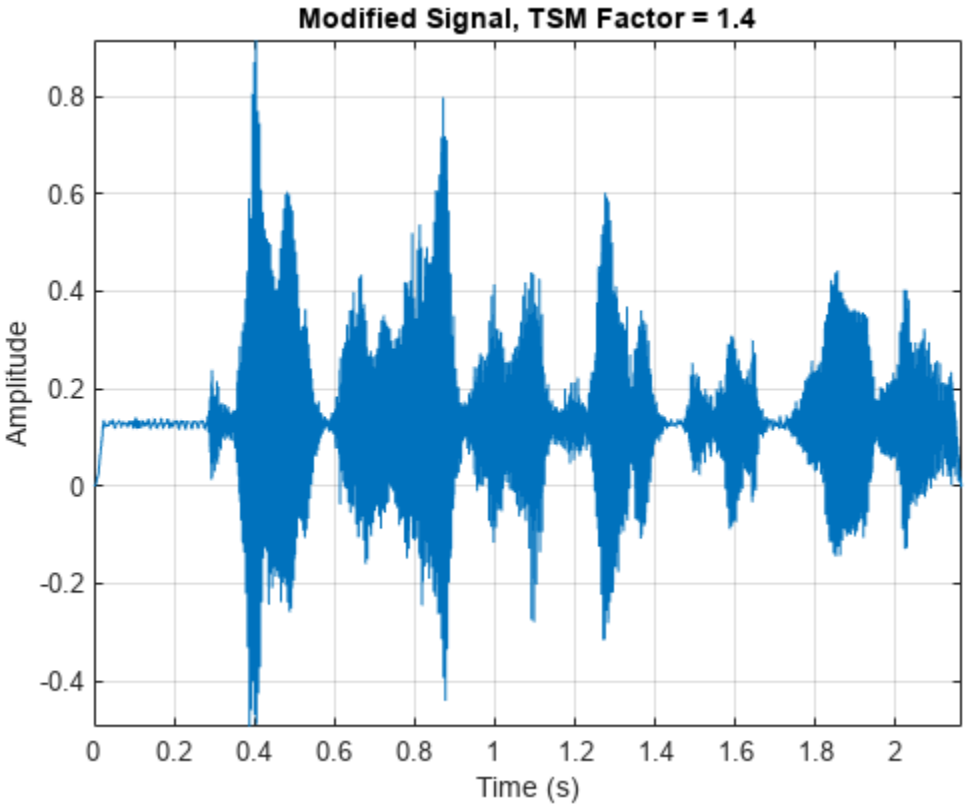


Convert the audio signal to the frequency domain.

```
win = sqrt(hann(256,'periodic'));
ovrlp = 192;
S = stft(audioIn,'Window',win,'OverlapLength',ovrlp,'Centered',false);
```

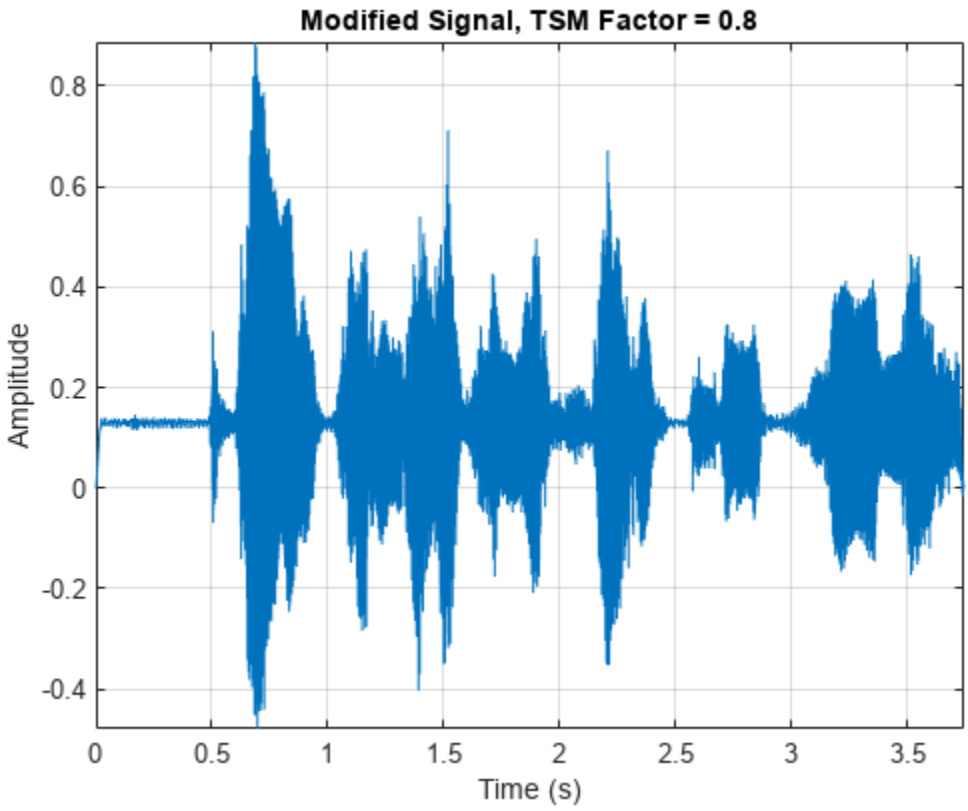
Speed up the audio signal by a factor of 1.4. Specify the window and overlap length used to create the frequency-domain representation.

```
alpha = 1.4;
audioOut = stretchAudio(S,alpha,'Window',win,'OverlapLength',ovrlp);

sound(audioOut,fs)

t = (0:size(audioOut,1)-1)/fs;
plot(t,audioOut)
xlabel('Time (s)')
ylabel('Amplitude')
title('Modified Signal, TSM Factor = 1.4')
axis tight
grid on
```



Slow down the audio signal by a factor of 0.8. Specify the window and overlap length used to create the frequency-domain representation.

```
alpha = 0.8;
audioOut = stretchAudio(S,alpha,'Window',win,'OverlapLength',ovrlp);

sound(audioOut,fs)

t = (0:size(audioOut,1)-1)/fs;
plot(t,audioOut)
xlabel('Time (s)')
ylabel('Amplitude')
title('Modified Signal, TSM Factor = 0.8')
axis tight
grid on
```

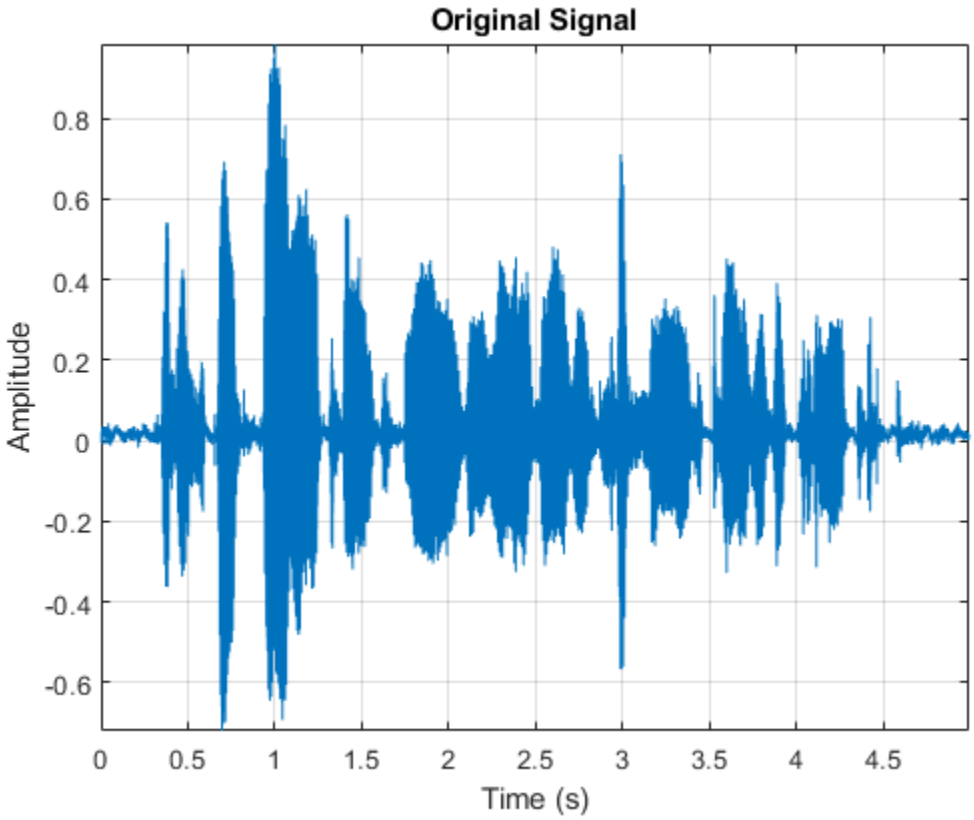


Increase Fidelity Using Phase-Locking

The default TSM method (vocoder) enables you to additionally apply phase-locking to increase the fidelity to the original audio.

Read in an audio signal. Listen to the audio signal and plot it over time.

```
[audioIn,fs] = audioread("SpeechDFT-16-8-mono-5secs.wav");

sound(audioIn,fs)

t = (0:size(audioIn,1)-1)/fs;
plot(t,audioIn)
xlabel('Time (s)')
ylabel('Amplitude')
title('Original Signal')
axis tight
grid on
```

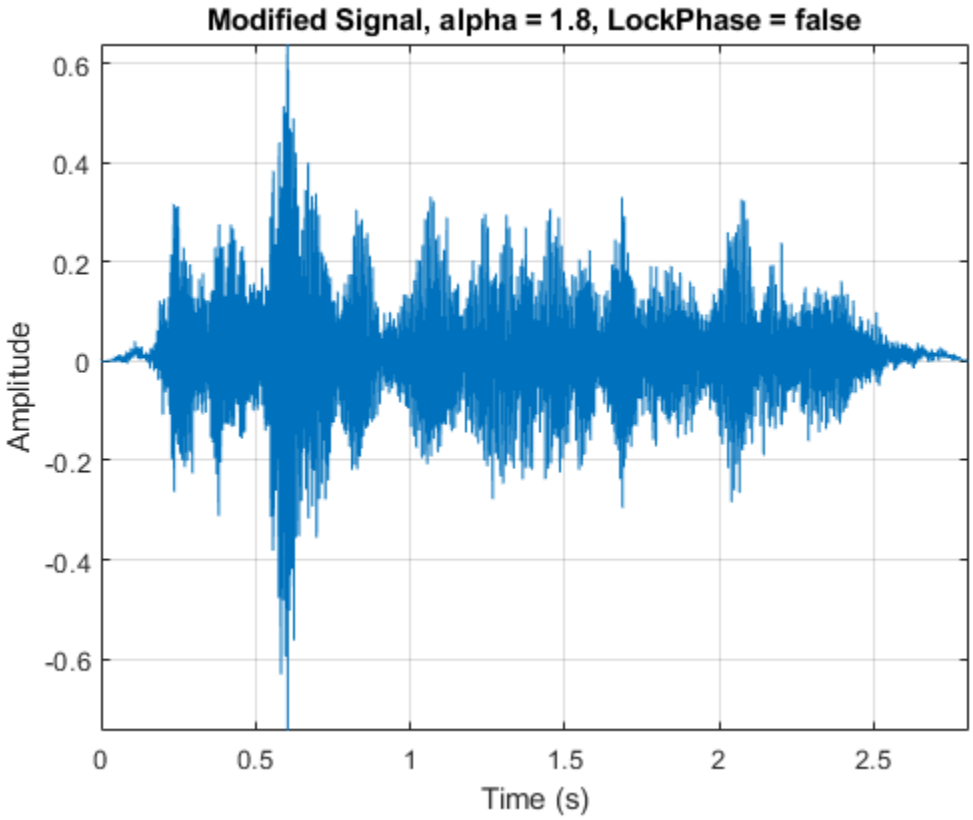


Phase-locking adds a nontrivial computational load to TSM and is not always required. By default, phase-locking is disabled. Apply a speedup factor of 1.8 to the input audio signal. Listen to the audio signal and plot it over time.

```
alpha = 1.8;

tic
audioOut = stretchAudio(audioIn,alpha);
processingTimeWithoutPhaseLocking = toc
```

```
processingTimeWithoutPhaseLocking = 0.0798
```

```
sound(audioOut,fs)

t = (0:size(audioOut,1)-1)/fs;
plot(t,audioOut)
xlabel('Time (s)')
ylabel('Amplitude')
title('Modified Signal, alpha = 1.8, LockPhase = false')
axis tight
grid on
```



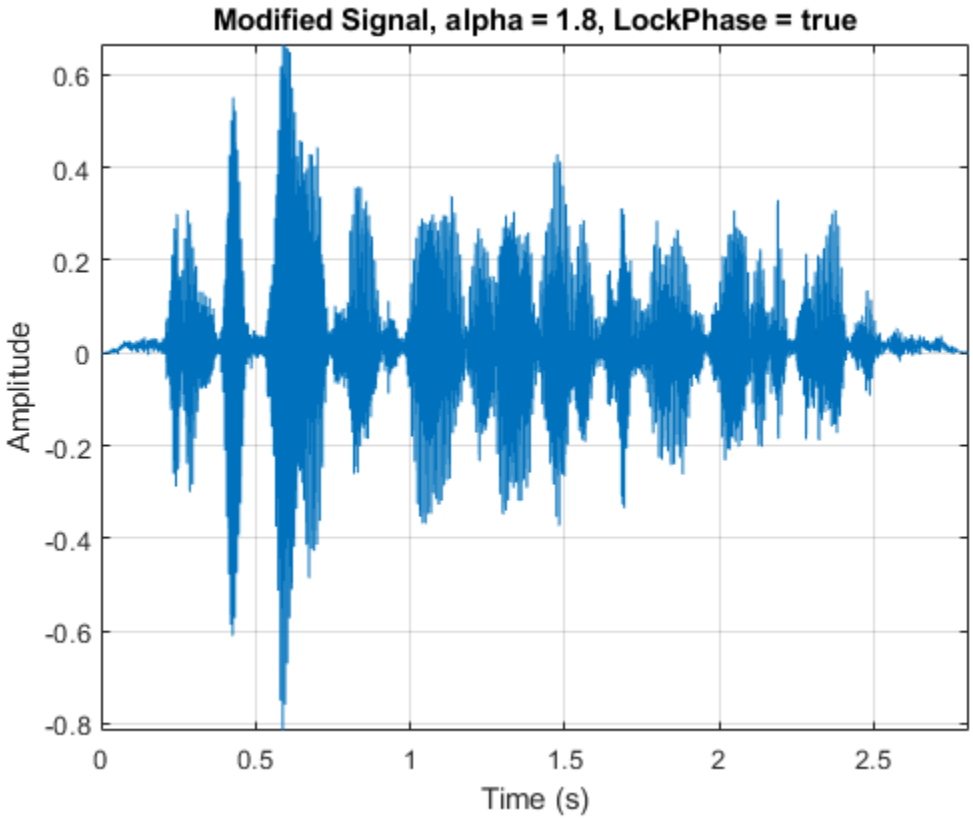
Apply the same 1.8 speedup factor to the input audio signal, this time enabling phase-locking. Listen to the audio signal and plot it over time.

```
tic
audioOut = stretchAudio(audioIn,alpha,"LockPhase",true);
processingTimeWithPhaseLocking = toc
```

```
processingTimeWithPhaseLocking = 0.1154
```

```
sound(audioOut,fs)
```

```
t = (0:size(audioOut,1)-1)/fs;
plot(t,audioOut)
xlabel('Time (s)')
ylabel('Amplitude')
title('Modified Signal, alpha = 1.8, LockPhase = true')
axis tight
grid on
```



Increase Fidelity Using WSOLA Delta

The waveform similarity overlap-add (WSOLA) TSM method enables you to specify the maximum number of samples to search for the best signal alignment. By default, WSOLA delta is the number of samples in the analysis window minus the number of samples overlapped between adjacent analysis windows. Increasing the WSOLA delta increases the computational load but might also increase fidelity.

Read in an audio signal. Listen to the first 10 seconds of the audio signal.

```
[audioIn,fs] = audioread('RockGuitar-16-96-stereo-72secs.flac');

sound(audioIn(1:10*fs,:),fs)
```

Apply a TSM factor of 0.75 to the input audio signal using the WSOLA method. Listen to the first 10 seconds of the resulting audio signal.

```
alpha = 0.75;
tic
audioOut = stretchAudio(audioIn,alpha,"Method","wsola");
processingTimeWithDefaultWSOLADelta = toc

processingTimeWithDefaultWSOLADelta = 19.4403

sound(audioOut(1:10*fs,:),fs)
```

Apply a TSM factor of 0.75 to the input audio signal, this time increasing the WSOLA delta to 1024. Listen to the first 10 seconds of the resulting audio signal.

```
tic
audioOut = stretchAudio(audioIn,alpha,"Method","wsola","WSOLADelta",1024);
processingTimeWithIncreasedWSOLADelta = toc
```

```
processingTimeWithIncreasedWSOLADelta = 25.5306
```

```
sound(audioOut(1:10*fs,:),fs)
```

Input Arguments

`audioIn` — Input signal

column vector | matrix | 3-D array

Input signal, specified as a column vector, matrix, or 3-D array. How the function interprets `audioIn` depends on the complexity of `audioIn` and the value of `Method`:

- If `audioIn` is real, `audioIn` is interpreted as a time-domain signal. In this case, `audioIn` must be a column vector or matrix. Columns are interpreted as individual channels.

  This syntax applies when `Method` is set to `'vocoder'` or `'wsola'`.
- If `audioIn` is complex, `audioIn` is interpreted as a frequency-domain signal. In this case, `audioIn` must be an *L*-by-*M*-by-*N* array, where *L* is the FFT length, *M* is the number of individual spectra, and *N* is the number of channels.

  This syntax only applies when `Method` is set to `'vocoder'`.

Data Types: `single` | `double`
Complex Number Support: Yes

`alpha` — TSM factor

positive scalar

TSM factor, specified as a positive scalar.

Data Types: `single` | `double`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `'Window',kbdwin(512)`

`Method` — Method used to time-scale audio

`'vocoder'` (default) | `'wsola'`

Method used to time-scale audio, specified as the comma-separated pair consisting of `'Method'` and `'vocoder'` or `'wsola'`. Set `'Method'` to `'vocoder'` to use the phase vocoder method. Set `'Method'` to `'wsola'` to use the WSOLA method.

If `'Method'` is set to `'vocoder'`, `audioIn` can be real or complex. If `'Method'` is set to `'wsola'`, `audioIn` must be real.

Data Types: `single` | `double`

`Window` — Window applied in time domain

`sqrt(hann(1024,'periodic'))` (default) | real vector

Window applied in the time domain, specified as the comma-separated pair consisting of `'Window'` and a real vector. The number of elements in the vector must be in the range [1, `size(audioIn,1)`]. The number of elements in the vector must also be greater than `OverlapLength`.

Note If using `stretchAudio` with frequency-domain input, you must specify `Window` as the same window used to transform `audioIn` to the frequency domain.

Data Types: `single` | `double`

`OverlapLength` — Number of samples overlapped between adjacent windows

`round(0.75*numel(Window))` (default) | scalar in the range [0 `numel(Window)`)

Number of samples overlapped between adjacent windows, specified as the comma-separated pair consisting of `'OverlapLength'` and an integer in the range [0, `numel(Window)`).

Note If using `stretchAudio` with frequency-domain input, you must specify `OverlapLength` as the same overlap length used to transform `audioIn` to a time-frequency representation.

Data Types: `single` | `double`

`LockPhase` — Apply identity phase-locking

`false` (default) | `true`

Apply identity phase-locking, specified as the comma-separated pair consisting of `'LockPhase'` and `false` or `true`.

Dependencies

To enable this name-value pair argument, set `Method` to `'vocoder'`.

Data Types: `logical`

`WSOLADelta` — Maximum samples used to search for best signal alignment

`numel(Window)-OverlapLength` (default) | nonnegative scalar

Maximum number of samples used to search for the best signal alignment, specified as the comma-separated pair consisting of `'WSOLADelta'` and a nonnegative scalar.

Dependencies

To enable this name-value pair argument, set `Method` to `'wsola'`.

Data Types: `single` | `double`

Output Arguments

`audioOut` — Time-scale modified audio

column vector | matrix

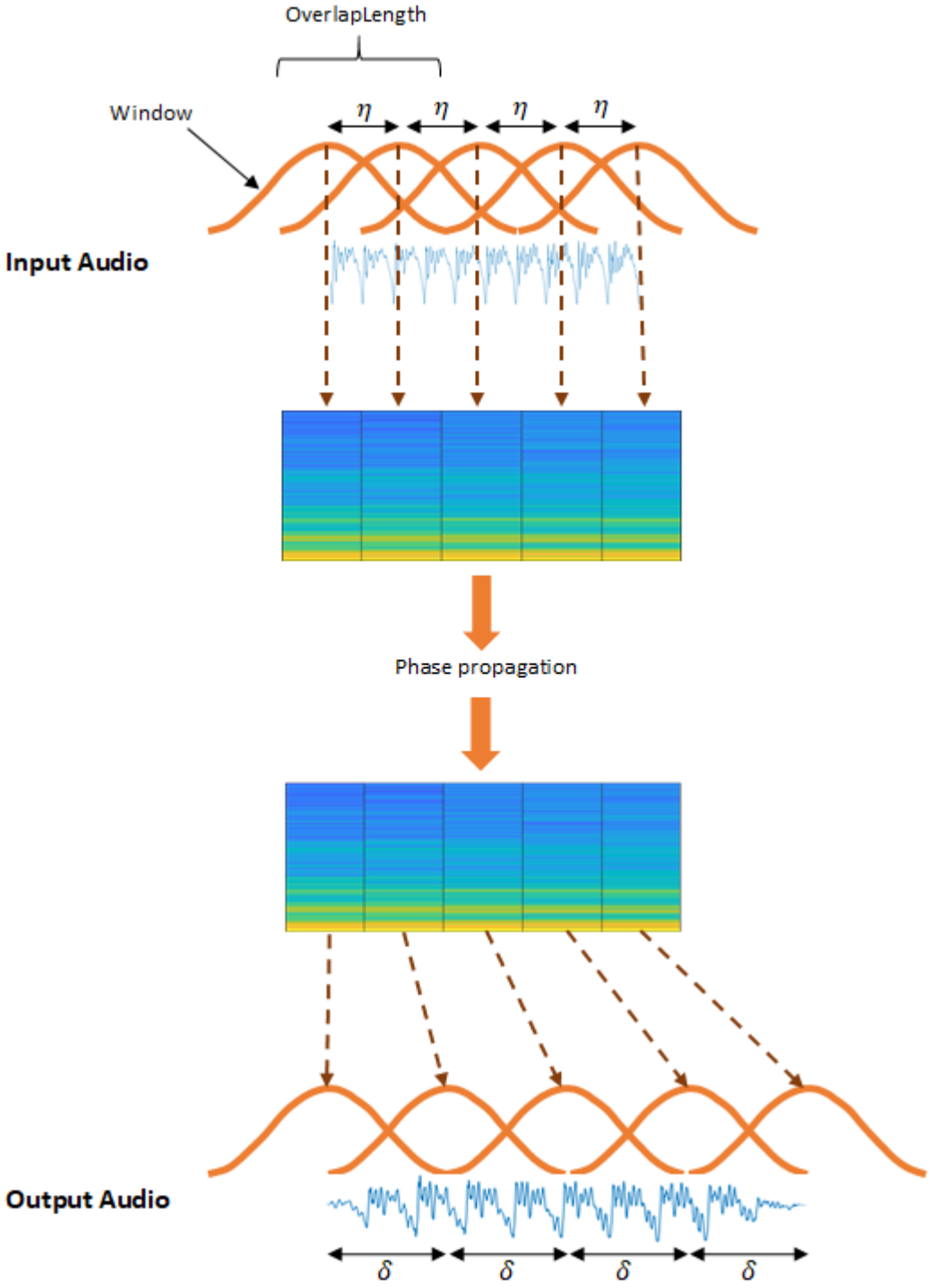
Time-scale modified audio, returned as a column vector or matrix of independent channels.

Algorithms

Phase Vocoder

The phase vocoder algorithm is a frequency-domain approach to TSM [1][2]. The basic steps of the phase vocoder algorithm are:

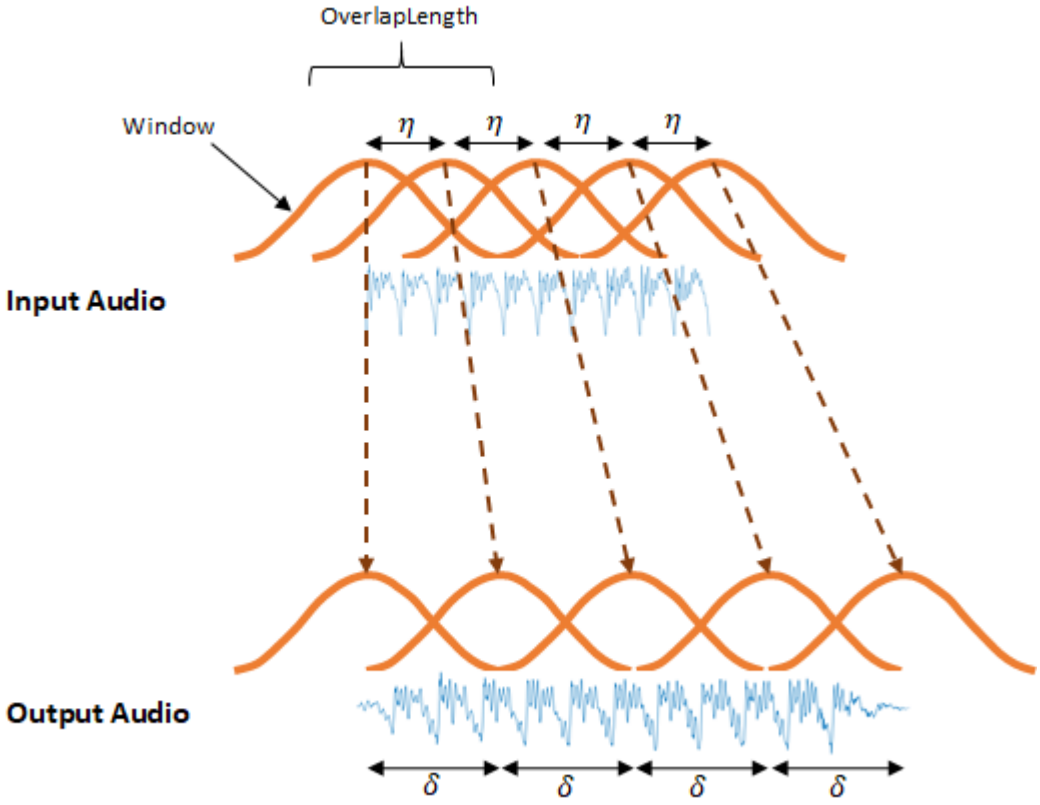
1. The algorithm windows a time-domain signal at interval η, where `η = numel(Window) - OverlapLength`. The windows are then converted to the frequency domain.
2. To preserve horizontal (across time) phase coherence, the algorithm treats each bin as an independent sinusoid whose phase is computed by accumulating the estimates of its instantaneous frequency.
3. To preserve vertical (across an individual spectrum) phase coherence, the algorithm locks the phase advance of groups of bins to the phase advance of local peaks. This step only applies if `LockPhase` is set to `true`.
4. The algorithm returns the modified spectrogram to the time domain, with windows spaced at intervals of δ, where δ ≈ η/α. α is the speedup factor specified by the `alpha` input argument.



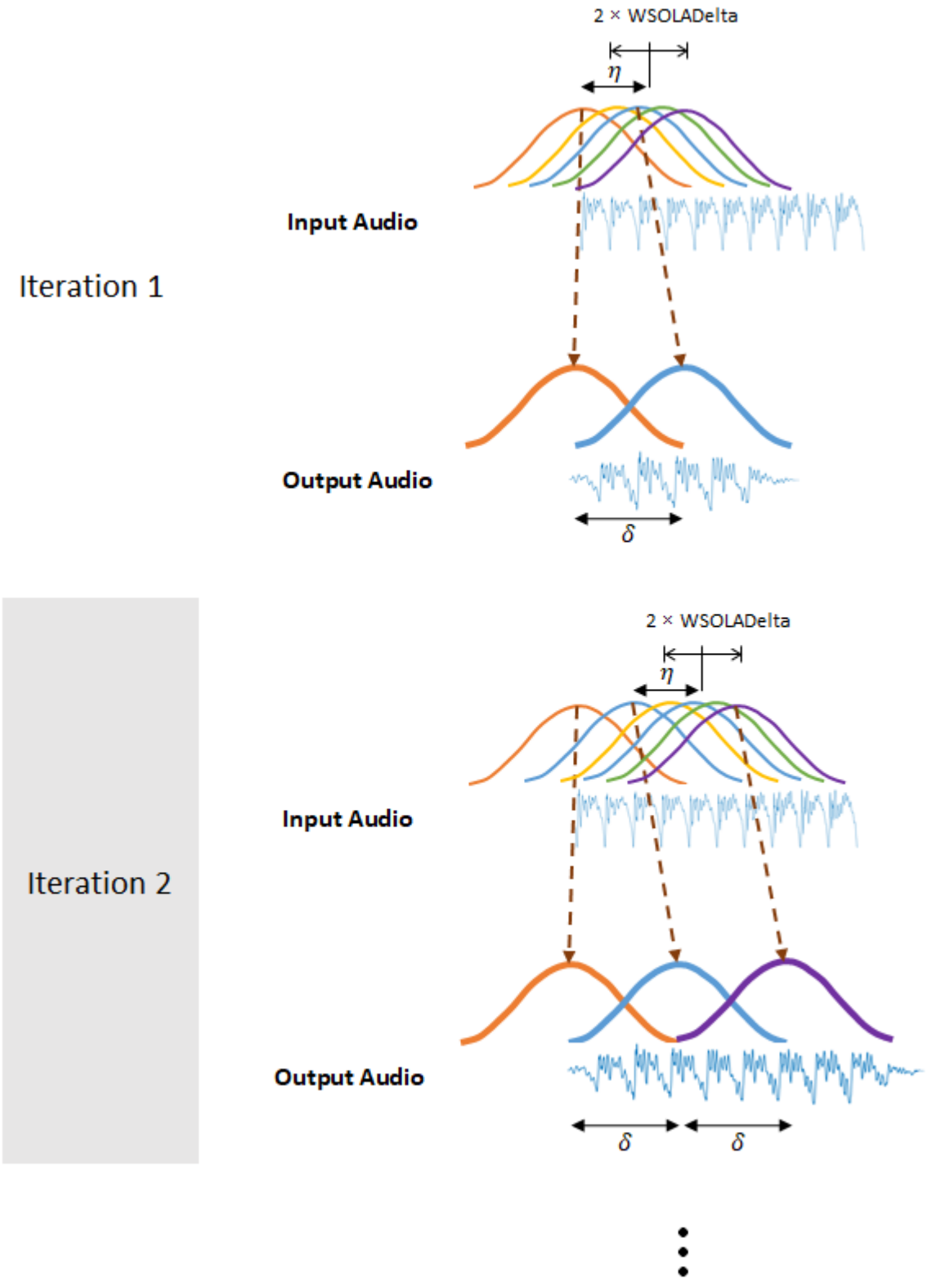
WSOLA

The WSOLA algorithm is a time-domain approach to TSM [1][2]. WSOLA is an extension of the overlap and add (OLA) algorithm. In the OLA algorithm, a time-domain signal is windowed at interval

η, where `η = numel(Window) - OverlapLength`. To construct the time-scale modified output audio, the windows are spaced at interval δ, where δ ≈ η/α. α is the TSM factor specified by the `alpha` input argument.



The OLA algorithm does a good job of recreating the magnitude spectra but can introduce phase jumps between windows. The WSOLA algorithm attempts to smooth the phase jumps by searching `WSOLADelta` samples around the η interval for a window that minimizes phase jumps. The algorithm searches for the best window iteratively, so that each successive window is chosen relative to the previously selected window.



If `WSOLADelta` is set to `0`, then the algorithm reduces to OLA.

Version History

Introduced in R2019b

References

[1] Driedger, Johnathan, and Meinard Müller. "A Review of Time-Scale Modification of Music Signals." *Applied Sciences*. Vol. 6, Issue 2, 2016.

[2] Driedger, Johnathan. "Time-Scale Modification Algorithms for Music Audio Signals", Master's thesis, Saarland University, Saarbrücken, Germany, 2011.

Extended Capabilities

C/C++ Code Generation

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

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- `Method` must be set to `'vocoder'`.
- `LockPhase` must be set to `false`.
- Using `gpuArray` (Parallel Computing Toolbox) input with `stretchAudio` is only recommended for a GPU with compute capability 7.0 ("Volta") or above. Other hardware might not offer any performance advantage. To check your GPU compute capability, see `ComputeCompability` in the output from the `gpuDevice` (Parallel Computing Toolbox) function. For more information, see “GPU Computing Requirements” (Parallel Computing Toolbox).

For an overview of GPU usage in MATLAB, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

See Also

`shiftPitch` | `reverberator` | `audioTimeScaler` | `audioDataAugmenter`

Topics

“Time-Frequency Masking for Harmonic-Percussive Source Separation”

shiftPitch

Shift audio pitch

Syntax

```
audioOut = shiftPitch(audioIn,nsemitones)
audioOut = shiftPitch(audioIn,nsemitones,Name,Value)
```

Description

`audioOut = shiftPitch(audioIn,nsemitones)` shifts the pitch of the audio input by the specified number of semitones, `nsemitones`.

`audioOut = shiftPitch(audioIn,nsemitones,Name,Value)` specifies options using one or more `Name,Value` pair arguments.

Examples

Apply Pitch-Shifting to Time-Domain Audio

Read in an audio file and listen to it.

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

Increase the pitch by 3 semitones and listen to the result.

```
nsemitones = 3;
audioOut = shiftPitch(audioIn,nsemitones);
sound(audioOut,fs)
```

Decrease the pitch of the original audio by 3 semitones and listen to the result.

```
nsemitones = -3;
audioOut = shiftPitch(audioIn,nsemitones);
sound(audioOut,fs)
```

Apply Pitch-Shifting to Frequency-Domain Audio

Read in an audio file and listen to it.

```
[audioIn,fs] = audioread("SpeechDFT-16-8-mono-5secs.wav");
sound(audioIn,fs)
```

Convert the audio signal to a time-frequency representation using `stft`. Use a 512-point `kbdwin` with 75% overlap.

```
win = kbdwin(512);
overlapLength = 0.75*numel(win);
```

```
S = stft(audioIn, ...
    "Window",win, ...
    "OverlapLength",overlapLength, ...
    "Centered",false);
```

Increase the pitch by 8 semitones and listen to the result. Specify the window and overlap length you used to compute the STFT.

```
nsemitones =  8                          ;
lockPhase = false                        ;
audioOut = shiftPitch(S,nsemitones, ...
                      "Window",win, ...
                      "OverlapLength",overlapLength, ...
                      "LockPhase",lockPhase);

sound(audioOut,fs)
```

Decrease the pitch of the original audio by 8 semitones and listen to the result. Specify the window and overlap length you used to compute the STFT.

```
nsemitones =  -8                         ;
lockPhase = false                        ;
audioOut = shiftPitch(S,nsemitones, ...
                      "Window",win, ...
                      "OverlapLength",overlapLength, ...
                      "LockPhase",lockPhase);

sound(audioOut,fs)
```

Increase Fidelity Using Phase Locking

Read in an audio file and listen to it.

```
[audioIn,fs] = audioread('FemaleSpeech-16-8-mono-3secs.wav');
sound(audioIn,fs)
```

Increase the pitch by 6 semitones and listen to the result.

```
nsemitones = 6;
lockPhase = false;
audioOut = shiftPitch(audioIn,nsemitones, ...
                      'LockPhase',lockPhase);
sound(audioOut,fs)
```

To increase fidelity, set `LockPhase` to `true`. Apply pitch shifting, and listen to the results.

```
lockPhase = true;
audioOut = shiftPitch(audioIn,nsemitones, ...
                      'LockPhase',lockPhase);
sound(audioOut,fs)
```

Increase Fidelity Using Formant Preservation

Read in the first 11.5 seconds of an audio file and listen to it.

```
[audioIn,fs] = audioread('Rainbow-16-8-mono-114secs.wav',[1,8e3*11.5]);
sound(audioIn,fs)
```

Increase the pitch by 4 semitones and apply phase locking. Listen to the results. The resulting audio has a "chipmunk effect" that sounds unnatural.

```
nsemitones = 4 ;
lockPhase = true;
audioOut = shiftPitch(audioIn,nsemitones, ...
    "LockPhase",lockPhase);

sound(audioOut,fs)
```

To increase fidelity, set `PreserveFormants` to `true`. Use the default cepstral order of `30`. Listen to the result.

```
cepstralOrder = 30 ;
audioOut = shiftPitch(audioIn,nsemitones, ...
    "LockPhase",lockPhase, ...
    "PreserveFormants",true, ...
    "CepstralOrder",cepstralOrder);

sound(audioOut,fs)
```

Input Arguments

`audioIn` — Input signal

column vector | matrix | 3-D array

Input signal, specified as a column vector, matrix, or 3-D array. How the function interprets `audioIn` depends on the complexity of `audioIn`:

- If `audioIn` is real, `audioIn` is interpreted as a time-domain signal. In this case, `audioIn` must be a column vector or matrix. Columns are interpreted as individual channels.
- If `audioIn` is complex, `audioIn` is interpreted as a frequency-domain signal. In this case, `audioIn` must be an *L*-by-*M*-by-*N* array, where *L* is the FFT length, *M* is the number of individual spectra, and *N* is the number of channels.

Data Types: `single` | `double`
Complex Number Support: Yes

`nsemitones` — Number of semitones to shift audio by

real scalar

Number of semitones to shift the audio by, specified as a real scalar.

The range of `nsemitones` depends on the window length (`numel(Window)`) and the overlap length (`OverlapLength`):

`-12*log2(numel(Window)-OverlapLength)` ≤ `nsemitones` ≤ `-12*log2((numel(Window)-OverlapLength)/numel(Window))`

Data Types: `single` | `double`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `'Window',kbdwin(512)`

`Window` — Window applied in time domain

`sqrt(hann(1024,'periodic'))` (default) | real vector

Window applied in the time domain, specified as the comma-separated pair consisting of `'Window'` and a real vector. The number of elements in the vector must be in the range [1, `size(audioIn,1)`]. The number of elements in the vector must also be greater than `OverlapLength`.

Note If using `shiftPitch` with frequency-domain input, you must specify `Window` as the same window used to transform `audioIn` to the frequency domain.

Data Types: `single` | `double`

`OverlapLength` — Number of samples overlapped between adjacent windows

`round(0.75*numel(Window))` (default) | scalar in the range [0, `numel(Window)`)

Number of samples overlapped between adjacent windows, specified as the comma-separated pair consisting of `'OverlapLength'` and an integer in the range [0, `numel(Window)`).

Note If using `shiftPitch` with frequency-domain input, you must specify `OverlapLength` as the same overlap length used to transform `audioIn` to a time-frequency representation.

Data Types: `single` | `double`

`LockPhase` — Apply identity phase locking

`false` (default) | `true`

Apply identity phase locking, specified as the comma-separated pair consisting of `'LockPhase'` and `false` or `true`.

Data Types: `logical`

`PreserveFormants` — Preserve formants

`false` (default) | `true`

Preserves formants, specified as the comma-separated pair consisting of `'PreserveFormants'` and `true` or `false`. Formant preservation is attempted using spectral envelope estimation with cepstral analysis.

Data Types: `logical`

`CepstralOrder` — Cepstral order used for formant preservation
30 (default) | nonnegative integer

Cepstral order used for formant preservation, specified as the comma-separated pair consisting of `'CepstralOrder'` and a nonnegative integer.

Dependencies

To enable this name-value pair argument, set `PreserveFormants` to `true`.

Data Types: `single` | `double`

Output Arguments

`audioOut` — Pitch-shifted audio
column vector | matrix

Pitch-shifted audio, returned as a column vector or matrix of independent channels.

Algorithms

To apply pitch shifting, `shiftPitch` modifies the time-scale of audio using a phase vocoder and then resamples the modified audio. The time scale modification algorithm is based on [1] and [2] and is implemented as in `stretchAudio`.

After time-scale modification, `shiftPitch` performs sample rate conversion using an interpolation factor equal to the analysis hop length and a decimation factor equal to the synthesis hop length. The interpolation and decimation factors of the resampling stage are selected as follows: The analysis hop length is determined as `analysisHopLength = numel(Window)-OverlapLength`. The `shiftPitch` function assumes that there are 12 semitones in an octave, so the speedup factor used to stretch the audio is `speedupFactor = 2^(-nsemitones/12)`. The speedup factor and analysis hop length determine the synthesis hop length for time-scale modification as `synthesisHopLength = round((1/SpeedupFactor)*analysisHopLength)`.

The achievable pitch shift is determined by the window length (`numel(Window)`) and `OverlapLength`. To see the relationship, note that the equation for speedup factor can be rewritten as: `nsemitones = -12*log2(speedupFactor)`, and the equation for synthesis hop length can be rewritten as `speedupFactor = analysisHopLengh/synthesisHopLength`. Using simple substitution, `nsemitones = -12*log2(analysisHopLength/synthesisHopLength)`. The practical range of a synthesis hop length is [1, `numel(Window)`]. The range of achievable pitch shifts is:

- Max number of semitones lowered: `-12*log2(numel(Window)-OverlapLength)`
- Max number of semitones raised: `-12*log2((numel(Window)-OverlapLength)/numel(Window))`

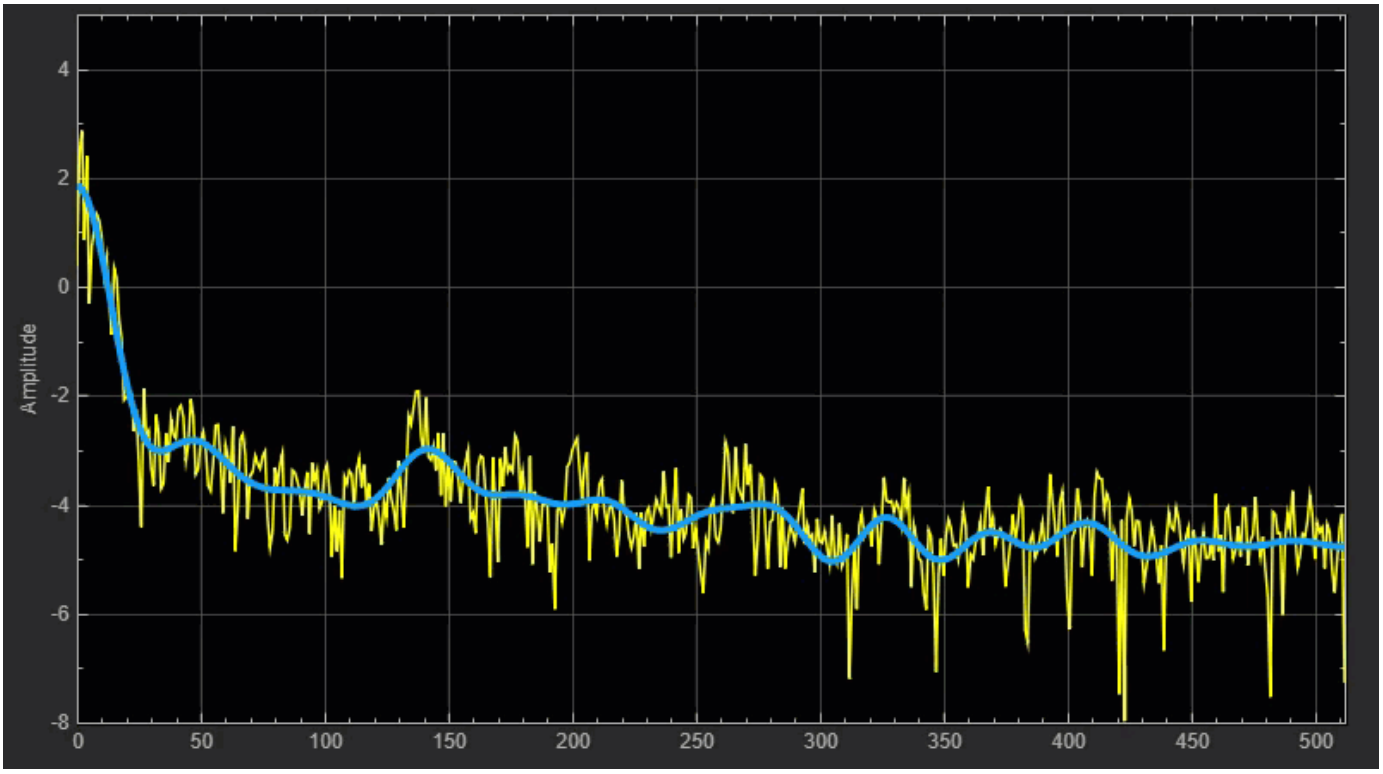
Formant Preservation

Pitch shifting can alter the spectral envelope of the pitch-shifted signal. To diminish this effect, you can set `PreserveFormants` to `true`. If `PreserveFormants` is set to `true`, the algorithm attempts to estimate the spectral envelope using an iterative procedure in the cepstral domain, as described in [3] and [4]. For both the original spectrum, X, and the pitch-shifted spectrum, Y, the algorithm estimates the spectral envelope as follows.

For the first iteration, $EnvX_a$ is set to X. Then, the algorithm repeats these two steps in a loop:

1. Lowpass filters the cepstral representation of $EnvX_a$ to get a new estimate, $EnvX_b$. The `CepstralOrder` parameter controls the quefrency bandwidth.
2. To update the current best fit, the algorithm takes the element-by-element maximum of the current spectral envelope estimate and the previous spectral envelope estimate:

   $$EnvX_a = \max(EnvX_a, EnvX_b).$$



The loop ends if either a maximum number of iterations (`100`) is reached, or if all bins of the estimated log envelope are within a given tolerance of the original log spectrum. The tolerance is set to `log(10^(1/20))`.

Finally, the algorithm scales the spectrum of the pitch-shifted audio by the ratio of estimated envelopes, element-wise:

$$Y = Y \times \left(\frac{EnvX_b}{EnvY_b}\right).$$

Version History

Introduced in R2019b

References

[1] Driedger, Johnathan, and Meinard Müller. "A Review of Time-Scale Modification of Music Signals." *Applied Sciences*. Vol. 6, Issue 2, 2016.

[2] Driedger, Johnathan. "Time-Scale Modification Algorithms for Music Audio Signals." Master's Thesis. Saarland University, Saarbrücken, Germany, 2011.

[3] Axel Roebel, and Xavier Rodet. "Efficient Spectral Envelope Estimation and its application to pitch shifting and envelope preservation." International Conference on Digital Audio Effects, pp. 30–35. Madrid, Spain, September 2005. hal-01161334

[4] S. Imai, and Y. Abe. "Spectral envelope extraction by improved cepstral method." *Electron. and Commun. in Japan*. Vol. 62-A, Issue 4, 1997, pp. 10–17.

Extended Capabilities

C/C++ Code Generation

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

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- `LockPhase` must be set to `false`.
- Using `gpuArray` (Parallel Computing Toolbox) input with `shiftPitch` is only recommended for a GPU with compute capability 7.0 ("Volta") or above. Other hardware might not offer any performance advantage. To check your GPU compute capability, see `ComputeCompability` in the output from the `gpuDevice` (Parallel Computing Toolbox) function. For more information, see “GPU Computing Requirements” (Parallel Computing Toolbox).

For an overview of GPU usage in MATLAB, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

See Also

`stretchAudio` | `reverberator` | `audioTimeScaler` | `audioDataAugmenter`

designAuditoryFilterBank

Design auditory filter bank

Syntax

```
filterBank = designAuditoryFilterBank(fs)
filterBank = designAuditoryFilterBank(fs,Name,Value)
[filterBank,Fc,BW] = designAuditoryFilterBank( ___ )
```

Description

`filterBank = designAuditoryFilterBank(fs)` returns a frequency-domain auditory filter bank, `filterBank`.

`filterBank = designAuditoryFilterBank(fs,Name,Value)` specifies options using one or more `Name,Value` pair arguments.

`[filterBank,Fc,BW] = designAuditoryFilterBank( ___ )` returns the center frequency and bandwidth of each filter in the filter bank. You can use this output syntax with any of the previous input syntaxes.

Examples

Create Default Auditory Filter Bank

Call `designAuditoryFilterBank` with a specified sample rate to design the default auditory filter bank.

```
fs = 44.1e3;
fb = designAuditoryFilterBank(fs);
```

The default filter bank consists of 32 triangular bandpass filters spaced evenly on the mel scale between 0 and `fs`/2 Hz.

```
numBands = size(fb,1)
```

```
numBands = 32
```

`designAuditoryFilterBank` is intended for frequency-domain filtering. By default, `designAuditoryFilterBank` assumes a 1024-point DFT and returns a half-sided frequency-domain filter bank with 513 points.

```
numPoints = size(fb,2)
```

```
numPoints = 513
```

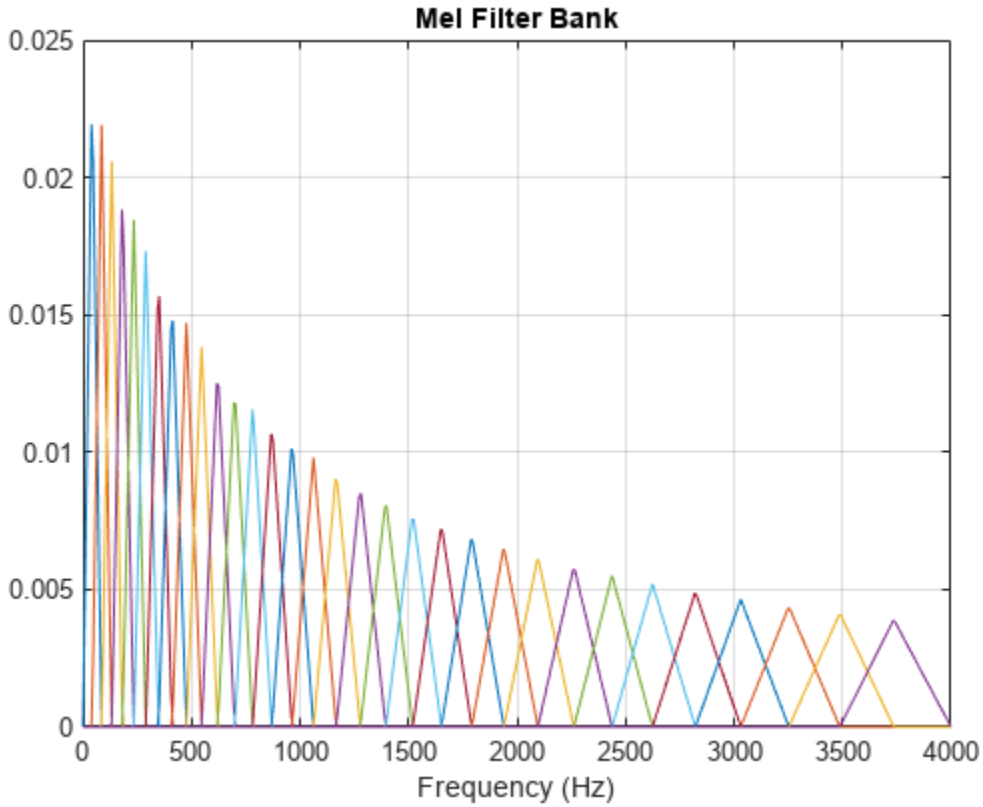
Design Mel-Based Auditory Filter Bank

Read in audio and convert it to a one-sided power spectrum.

```
[audioIn,fs] = audioread("Laughter-16-8-mono-4secs.wav");

win = hamming(1024,"periodic");
noverlap = 512;
fftLength = 1024;
[S,F,t] = stft(audioIn,fs, ...
               "Window",win, ...
               "OverlapLength",noverlap, ...
               "FFTLength",fftLength, ...
               "FrequencyRange","onesided");
PowerSpectrum = S.*conj(S);
```

Design a mel-based auditory filter bank. Plot the filter bank.

```
numBands = 32 ;
range = [ 0 , 4000 ];
normalization = bandwidth;

[fb,cf] = designAuditoryFilterBank(fs, ...
                                   "FFTLength",fftLength, ...
                                   "NumBands",numBands, ...
                                   "FrequencyRange",range, ...
                                   "Normalization",normalization);

plot(F,fb.')
grid on
title("Mel Filter Bank")
xlabel("Frequency (Hz)")
```

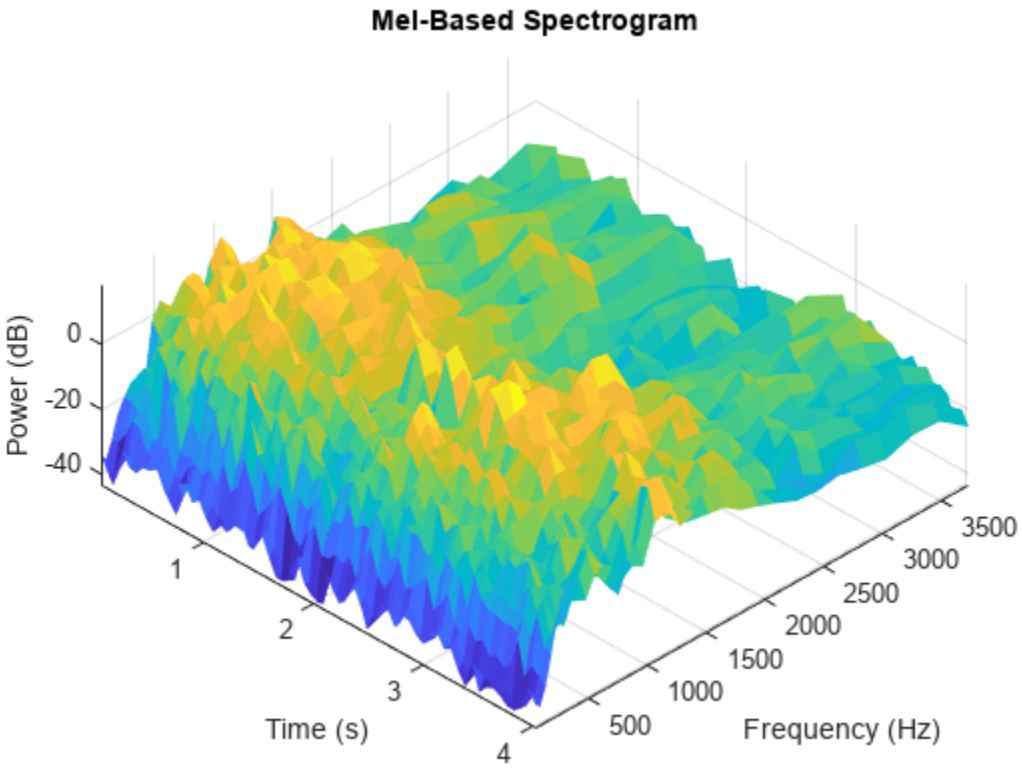


To apply frequency domain filtering, perform a matrix multiplication of the filter bank and the power spectrogram.

```
X = fb*PowerSpectrum;
```

Visualize the power-per-band in dB.

```
XdB = 10*log10(X);

surf(t,cf,XdB,"EdgeColor","none");
xlabel("Time (s)")
ylabel("Frequency (Hz)")
zlabel("Power (dB)")
view([45,60])
title('Mel-Based Spectrogram')
axis tight
```



Design Bark-Based Auditory Filter Bank

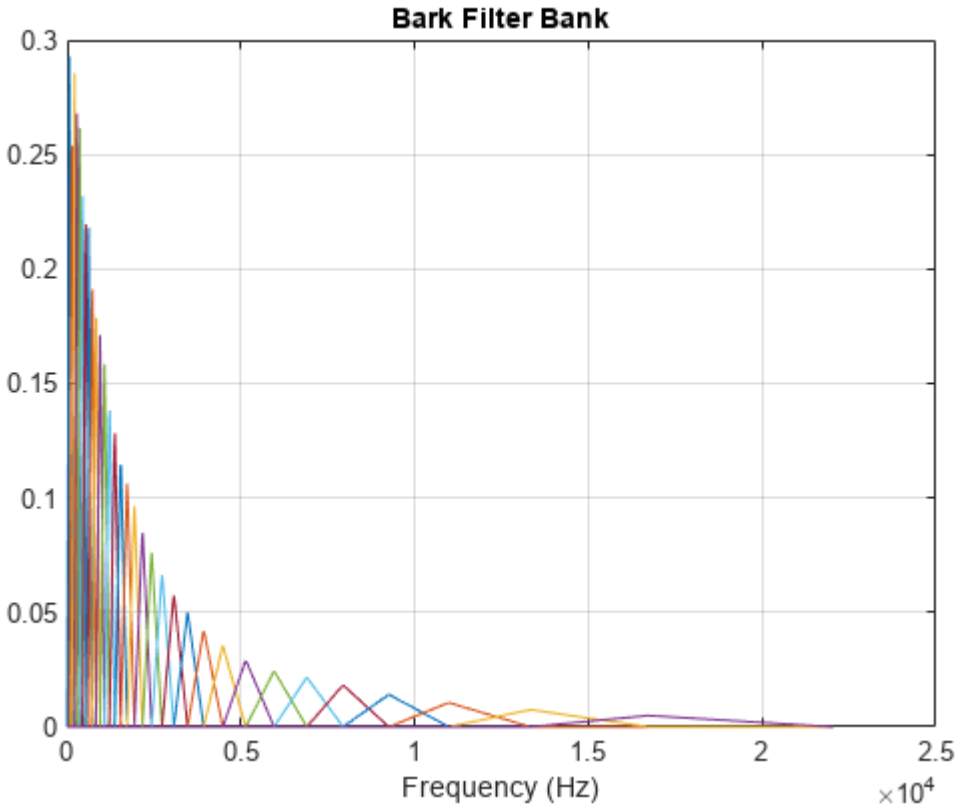
Read in audio and convert it to a one-sided power spectrum.

```
[audioIn,fs] = audioread("RockDrums-44p1-stereo-11secs.mp3");

win = hann(round(0.03*fs),"periodic");
noverlap = round(0.02*fs);
fftLength = 2048;

[S,F,t] = stft(audioIn,fs, ...
               "Window",win, ...
               "OverlapLength",noverlap, ...
               "FFTLength",fftLength, ...
               "FrequencyRange","onesided");
PowerSpectrum = S.*conj(S);
```

Design a Bark-based auditory filter bank. Plot the filter bank.

```
numBands = 32 ;
range = [ 0 , 22050 ];
normalization = area;
```

```
designDomain = linear;

[fb,cf] = designAuditoryFilterBank(fs, ...
    "FrequencyScale","bark", ...
    "FFTLength",fftLength, ...
    "NumBands",numBands, ...
    "FrequencyRange",range, ...
    "Normalization",normalization, ...
    "FilterBankDesignDomain",designDomain);

plot(F,fb.');
grid on
title("Bark Filter Bank")
xlabel("Frequency (Hz)")
```



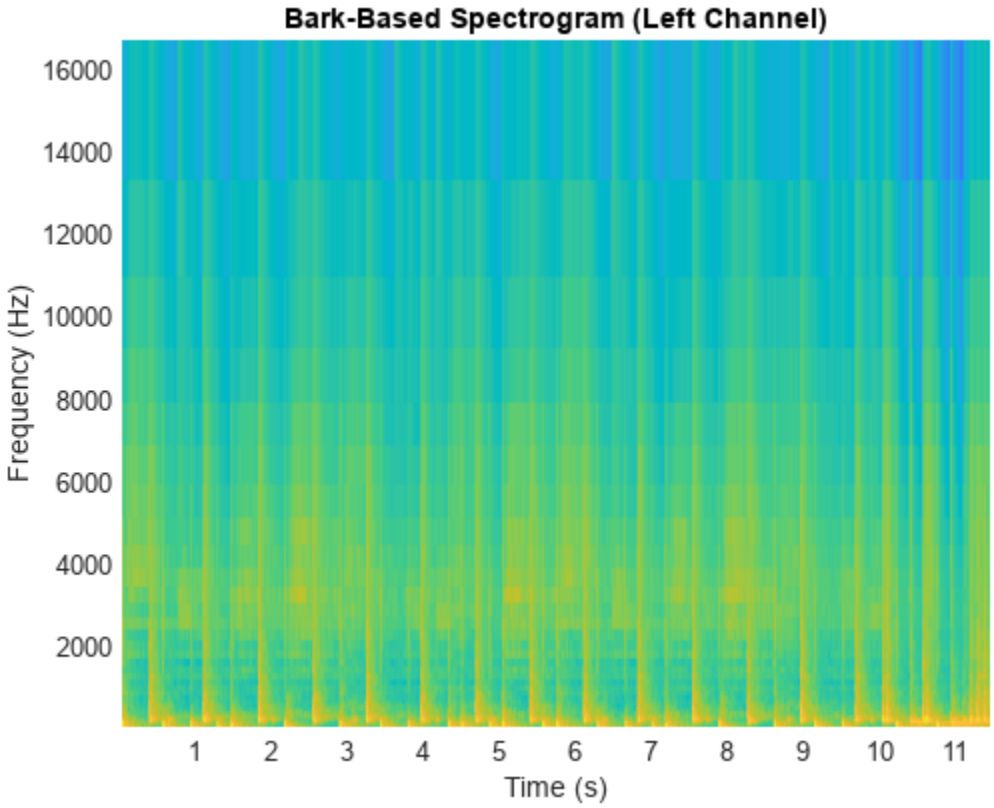
To apply frequency domain filtering, perform a matrix multiplication of the filter bank and the left and right power spectrograms.

```
X = pagemtimes(fb,PowerSpectrum);
```

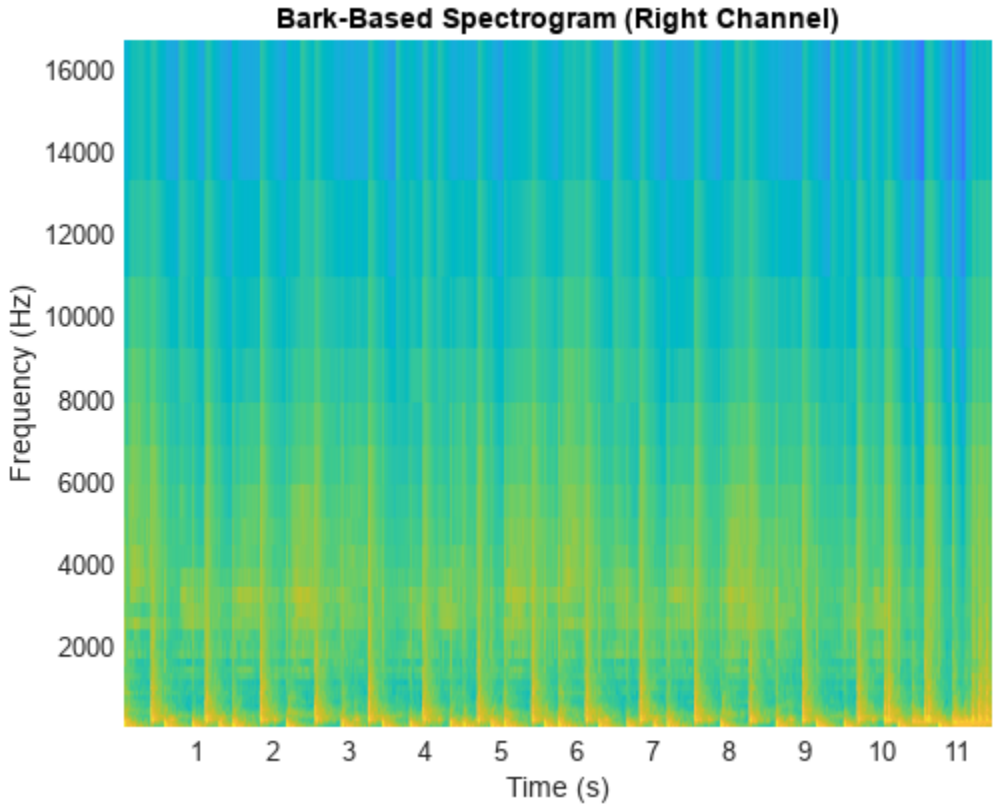
Visualize the power-per-band in dB.

```
XLdB = 10*log10(X(:,:,1));
XRdB = 10*log10(X(:,:,2));

surf(t,cf,XLdB,"EdgeColor","none");
xlabel("Time (s)")
```

```
ylabel("Frequency (Hz)")
view([0,90])
title("Bark-Based Spectrogram (Left Channel)")
axis tight
```




```
surf(t,cf,XRdB,"EdgeColor","none");
xlabel("Time (s)")
ylabel("Frequency (Hz)")
view([0,90])
title("Bark-Based Spectrogram (Right Channel)")
axis tight
```



Design ERB-Based Auditory Filter Bank

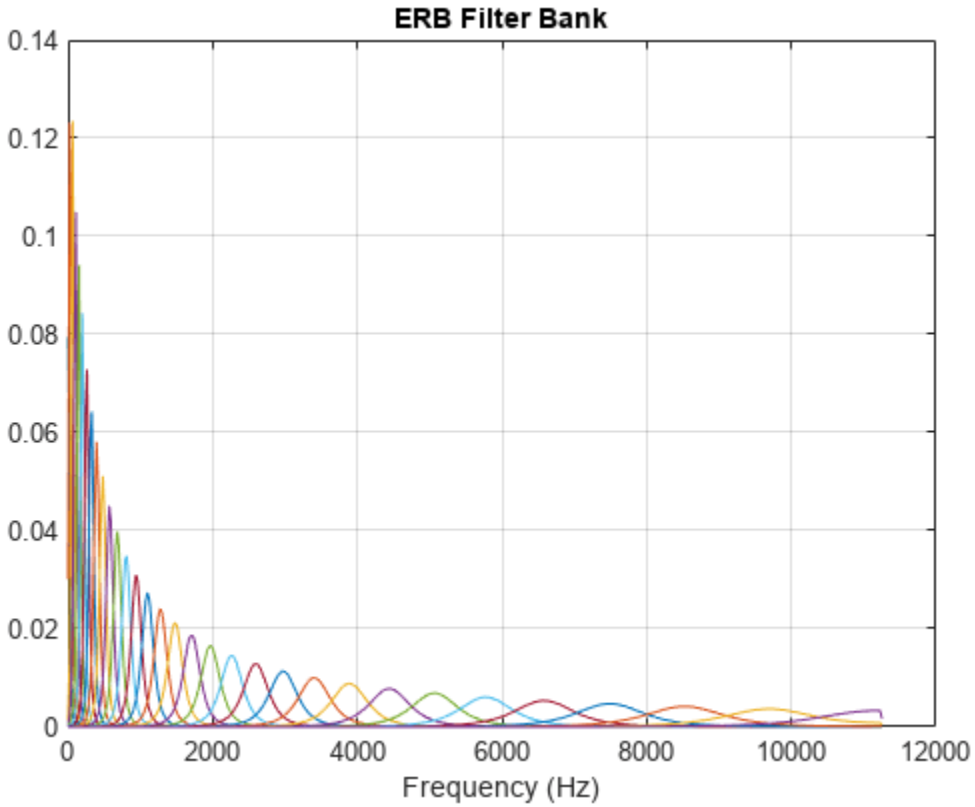
Read in audio and convert it to a one-sided power spectrum.

```
[audioIn,fs] = audioread("NoisySpeech-16-22p5-mono-5secs.wav");

win = hann(round(0.04*fs),"periodic");
noverlap = round(0.02*fs);
fftLength = 1024;

[S,F,t] = stft(audioIn,fs, ...
               "Window",win, ...
               "OverlapLength",noverlap, ...
               "FFTLength",fftLength, ...
               "FrequencyRange","onesided");
PowerSpectrum = S.*conj(S);
```

Design an ERB-based auditory filter bank. Plot the filter bank.

```
numBands =  32  ;
range = [  0  ,  11025  ];
normalization =  bandwidth ;
```

```
[fb,cf] = designAuditoryFilterBank(fs, ...
    "FrequencyScale","erb", ...
    "FFTLength",fftLength, ...
    "NumBands",numBands, ...
    "FrequencyRange",range, ...
    "Normalization",normalization);

plot(F,fb.');
grid on
title("ERB Filter Bank")
xlabel("Frequency (Hz)")
```

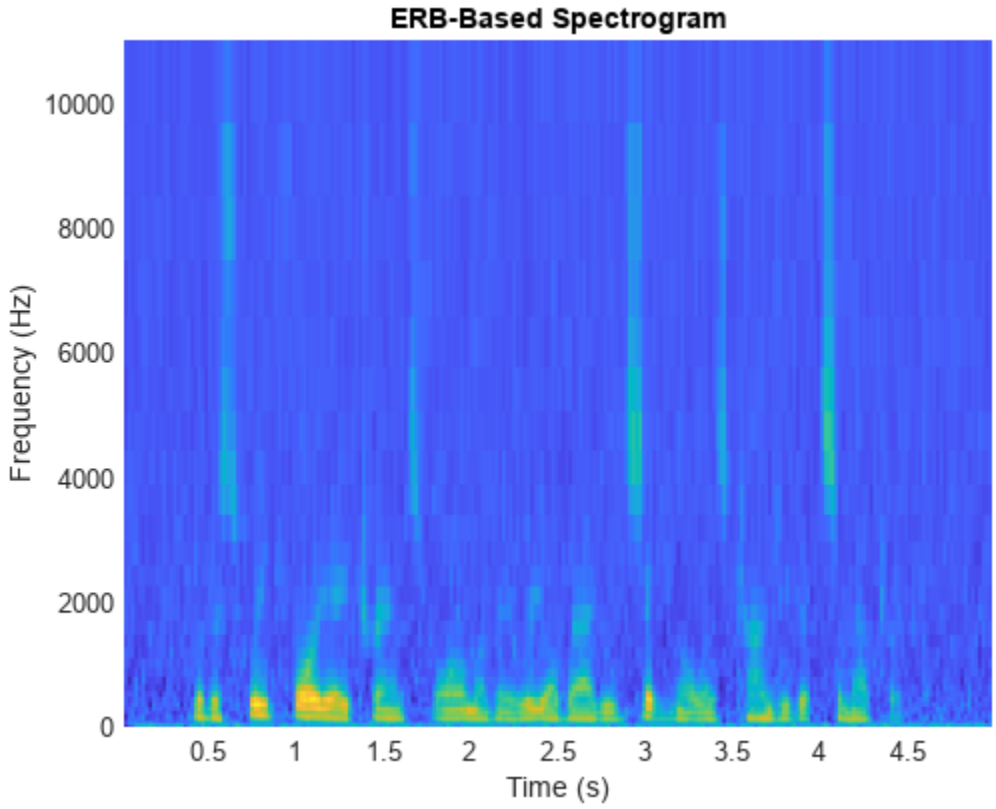


To apply frequency-domain filtering, perform a matrix multiplication of the filter bank and the power spectrogram.

```
X = fb*PowerSpectrum;
```

Visualize the power-per-band in dB.

```
XdB = 10*log10(X);
surf(t,cf,XdB,"EdgeColor","none");
xlabel("Time (s)")
ylabel("Frequency (Hz)")
view([0,90])
title("ERB-Based Spectrogram")
axis tight
```



Input Arguments

`fs` — Sample rate of filter design (Hz)

positive scalar

Sample rate of filter design in Hz, specified as a positive scalar.

Data Types: `single` | `double`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `"FrequencyScale","mel"`

`FrequencyScale` — Frequency scale

`"mel"` (default) | `"bark"` | `"erb"`

Frequency scale used to design the auditory filter bank, specified as the comma-separated pair consisting of `'FrequencyScale'` and `"mel"`, `"bark"`, or `"erb"`.

Data Types: `char` | `string`

`FFTLength` — Number of DFT points
`1024` (default) | positive integer

Number of points used to calculate the DFT, specified as the comma-separated pair consisting of `'FFTLength'` and a positive integer.

Data Types: `single` | `double`

`NumBands` — Number of bandpass filters
positive integer

Number of bandpass filters, specified as the comma-separated pair consisting of `'NumBands'` and a positive integer. The default number of bandpass filters depends on the `FrequencyScale`:

- If `FrequencyScale` is set to `"bark"` or `"mel"`, then `NumBands` defaults to `32`.
- If `FrequencyScale` is set to `"erb"`, then `NumBands` defaults to `ceil(hz2erb(FrequencyRange(2))-hz2erb(FrequencyRange(1)))`.

Data Types: `single` | `double`

`FrequencyRange` — Frequency range over which to design auditory filter bank (Hz)
`[0 fs/2]` (default) | two-element row vector

Frequency range over which to design auditory filter bank in Hz, specified as the comma-separated pair consisting of `'FrequencyRange'` and a two-element row vector of monotonically increasing values in the range [0, `fs`/2].

Data Types: `single` | `double`

`Normalization` — Normalize filter bank
`"bandwidth"` (default) | `"area"` | `"none"`

Normalization technique used on the weights of the filter bank:

- `"bandwidth"` –– The weights of each bandpass filter are normalized by the corresponding bandwidth of the filter.
- `"area"` –– The weights of each bandpass filter are normalized by the corresponding area of the bandpass filter.
- `"none"` –– The weights of the filters are not normalized.

Data Types: `char` | `string`

`OneSided` — Design one-sided or two-sided filter bank
`true` (default) | `false`

Design a one-sided or two-sided filter bank, specified as the comma-separated pair consisting of `'OneSided'` and either `true` or `false`.

Data Types: `logical`

`FilterBankDesignDomain` — Domain in which filter bank is designed
`"linear"` (default) | `"warped"`

Domain in which filter bank is designed, specified as the comma-separated pair consisting of `'FilterBankDesignDomain'` and either `"linear"` or `"warped"`. Set the filter bank design domain

to `"linear"` to design the bandpass filters in the linear (Hz) domain. Set the filter bank design domain to `"warped"` to design the bandpass filters in the warped (mel or Bark) domain.

Dependencies

This parameter only applies if `FrequencyScale` is set to `"mel"` (default) or `"bark"`.

Data Types: `char` | `string`

Output Arguments

`filterBank` — Auditory filter bank

column vector | matrix

Auditory filter bank, returned as an *M*-by-*N* matrix, where *M* is the number of bands (`NumBands`), and *N* is the number of frequency points of a one-sided spectrum (`ceil(FFTLength/2)`).

Data Types: `double`

`Fc` — Center frequencies of bandpass filters (Hz)

row vector

Center frequencies of bandpass filters in Hz, returned as a row vector with `NumBands` elements.

Data Types: `double`

`BW` — Bandwidth of bandpass filters (Hz)

row vector

Bandwidth of bandpass filters in Hz, returned as a row vector with `NumBands` elements.

Data Types: `double`

Algorithms

The mel filter bank is designed as half-overlapped triangles equally spaced on the mel scale. [1]

The Bark filter bank is designed as half-overlapped triangles equally spaced on the Bark scale. [2]

The ERB filter bank is designed as gammatone filters [4] whose center frequencies are equally spaced on the ERB scale. [3]

Version History

Introduced in R2019b

R2020b: `designAuditoryFilterBank` scaling changed for ERB filter banks

Behavior changed in R2020b

The half-sided ERB filter bank returned from `designAuditoryFilterBank` is now scaled by 2. This change provides consistent results when applying one-sided or two-sided filtering, without requiring multiplications in the processing loop.

References

[1] O'Shaughnessy, Douglas. *Speech Communication: Human and Machine*. Reading, MA: Addison-Wesley Publishing Company, 1987.

[2] Traunmüller, Hartmut. "Analytical Expressions for the Tonotopic Sensory Scale." *Journal of the Acoustical Society of America*. Vol. 88, Issue 1, 1990, pp. 97–100.

[3] Glasberg, Brian R., and Brian C. J. Moore. "Derivation of Auditory Filter Shapes from Notched-Noise Data." *Hearing Research*. Vol. 47, Issues 1–2, 1990, pp. 103–138.

[4] Slaney, Malcolm. "An Efficient Implementation of the Patterson-Holdsworth Auditory Filter Bank." Apple Computer Technical Report 35, 1993.

Extended Capabilities

C/C++ Code Generation

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

See Also

`gammatoneFilterBank` | `melSpectrogram` | `hz2mel` | `hz2bark` | `hz2erb` | `erb2hz` | `bark2hz` | `mel2hz`

melSpectrogram

Mel spectrogram

Syntax

```
S = melSpectrogram(audioIn,fs)
S = melSpectrogram(audioIn,fs,Name,Value)
[S,F,T] = melSpectrogram( ___ )
melSpectrogram( ___ )
```

Description

`S = melSpectrogram(audioIn,fs)` returns the mel spectrogram of the audio input at sample rate `fs`. The function treats columns of the input as individual channels.

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

`[S,F,T] = melSpectrogram( ___ )` returns the center frequencies of the bands in Hz and the location of each window of data in seconds. The location corresponds to the center of each window. You can use this output syntax with any of the previous input syntaxes.

`melSpectrogram( ___ )` plots the mel spectrogram on a surface in the current figure.

Examples

Calculate Mel Spectrogram

Use the default settings to calculate the mel spectrogram for an entire audio file. Print the number of bandpass filters in the filter bank and the number of frames in the mel spectrogram.

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

S = melSpectrogram(audioIn,fs);

[numBands,numFrames] = size(S);
fprintf("Number of bandpass filters in filterbank: %d\n",numBands)
```

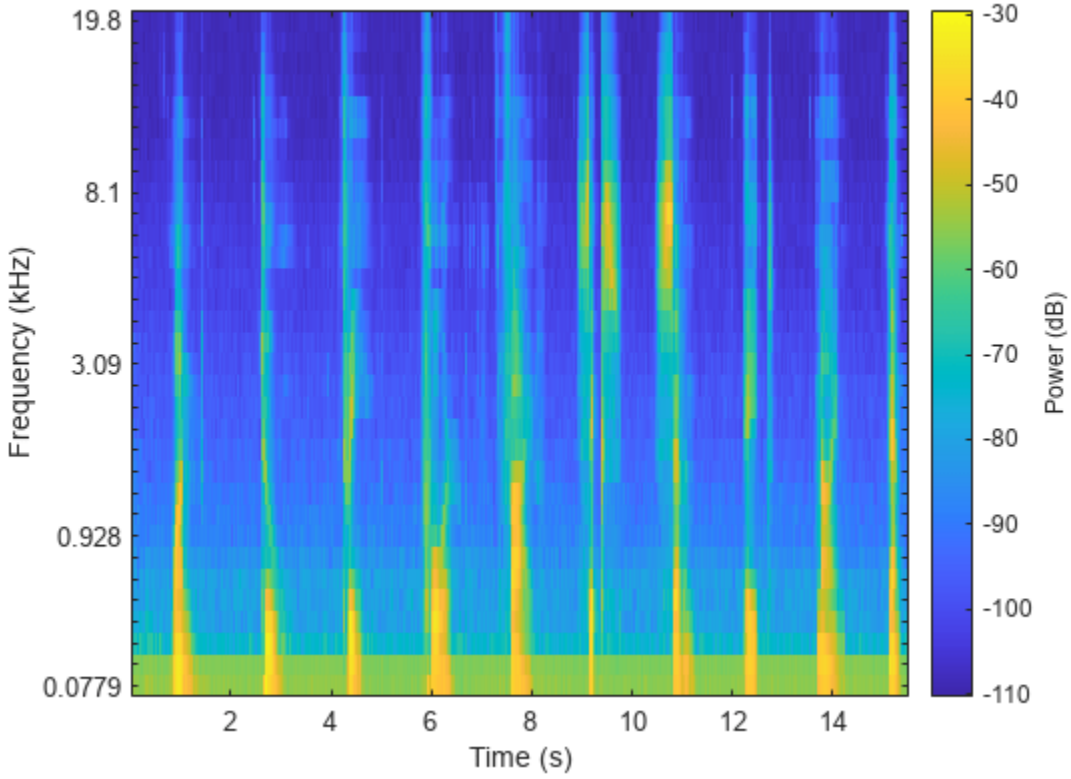
```
Number of bandpass filters in filterbank: 32
```

```
fprintf("Number of frames in spectrogram: %d\n",numFrames)
```

```
Number of frames in spectrogram: 1551
```

Plot the mel spectrogram.

```
melSpectrogram(audioIn,fs)
```



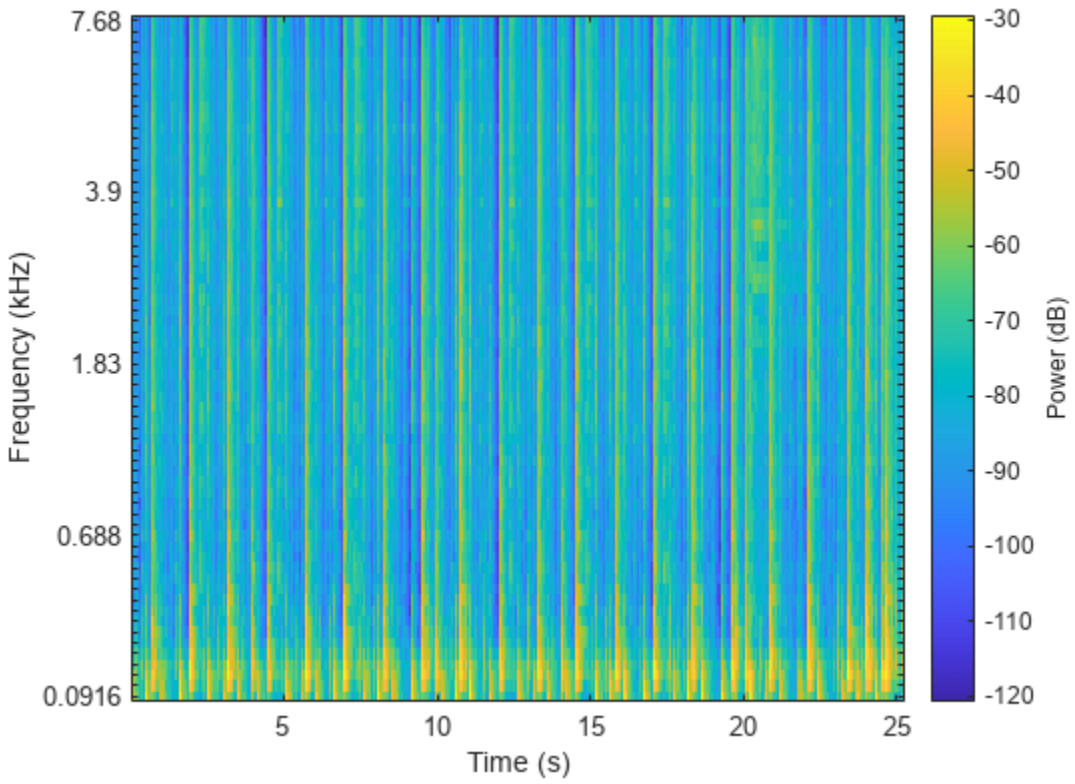
Calculate Mel Spectrums of 2048-Point Windows

Calculate the mel spectrums of 2048-point periodic Hann windows with 1024-point overlap. Convert to the frequency domain using a 4096-point FFT. Pass the frequency-domain representation through 64 half-overlapped triangular bandpass filters that span the range 62.5 Hz to 8 kHz.

```
[audioIn,fs] = audioread('FunkyDrums-44p1-stereo-25secs.mp3');

S = melSpectrogram(audioIn,fs, ...
                   'Window',hann(2048,'periodic'), ...
                   'OverlapLength',1024, ...
                   'FFTLength',4096, ...
                   'NumBands',64, ...
                   'FrequencyRange',[62.5,8e3]);
```

Call `melSpectrogram` again, this time with no output arguments so that you can visualize the mel spectrogram. The input audio is a multichannel signal. If you call `melSpectrogram` with a multichannel input and with no output arguments, only the first channel is plotted.

```
melSpectrogram(audioIn,fs, ...
               'Window',hann(2048,'periodic'), ...
               'OverlapLength',1024, ...
               'FFTLength',4096, ...
               'NumBands',64, ...
               'FrequencyRange',[62.5,8e3])
```

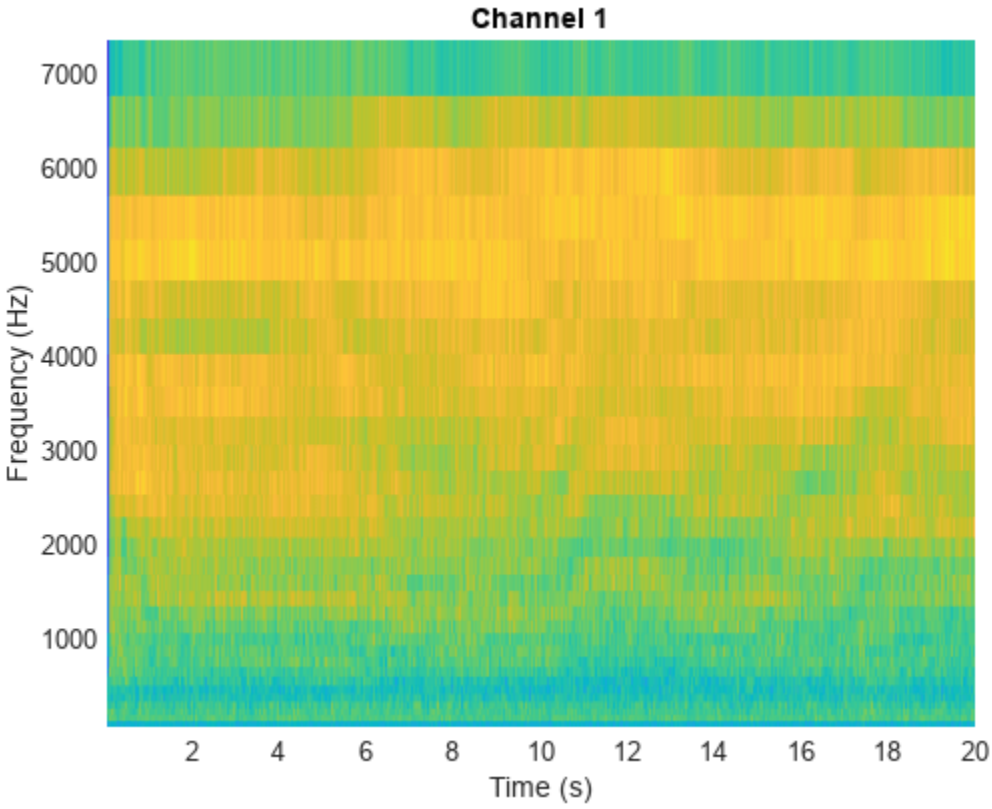


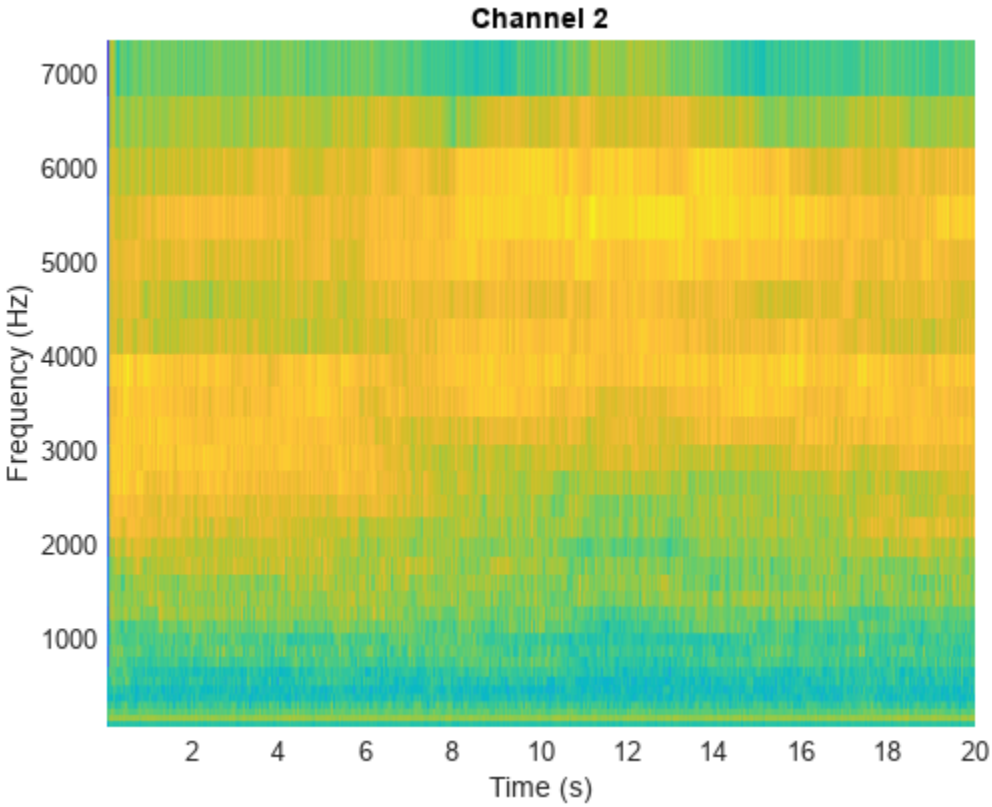
Get Filter Bank Center Frequencies and Analysis Window Time Instants

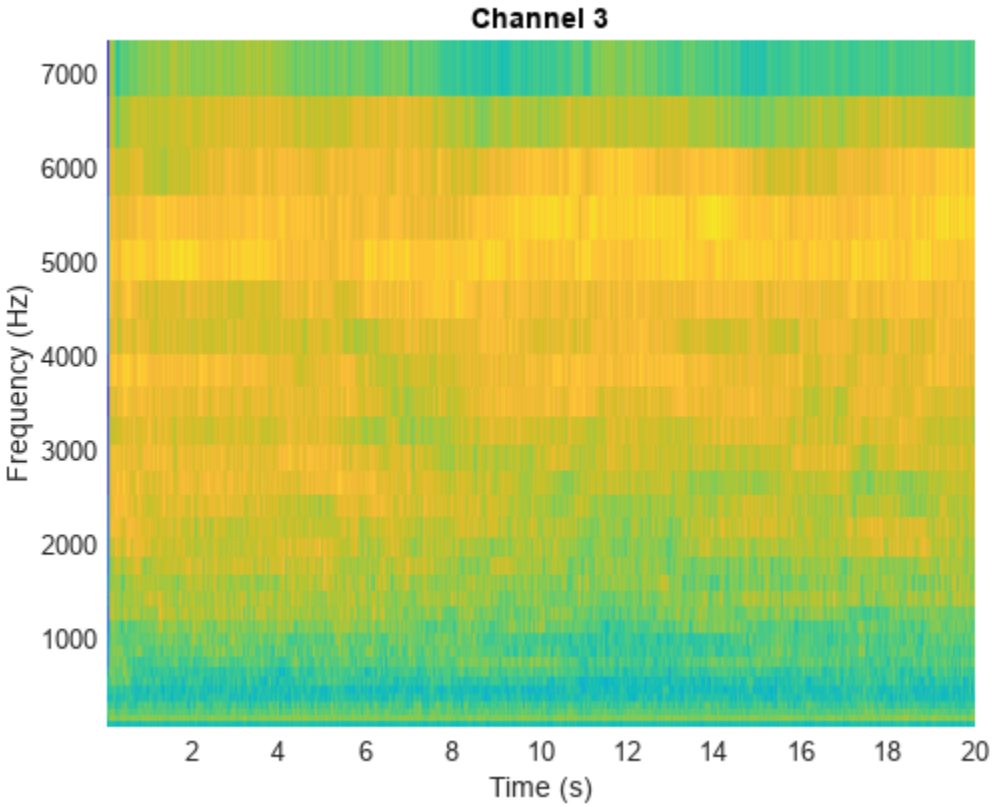
`melSpectrogram` applies a frequency-domain filter bank to audio signals that are windowed in time. You can get the center frequencies of the filters and the time instants corresponding to the analysis windows as the second and third output arguments from `melSpectrogram`.

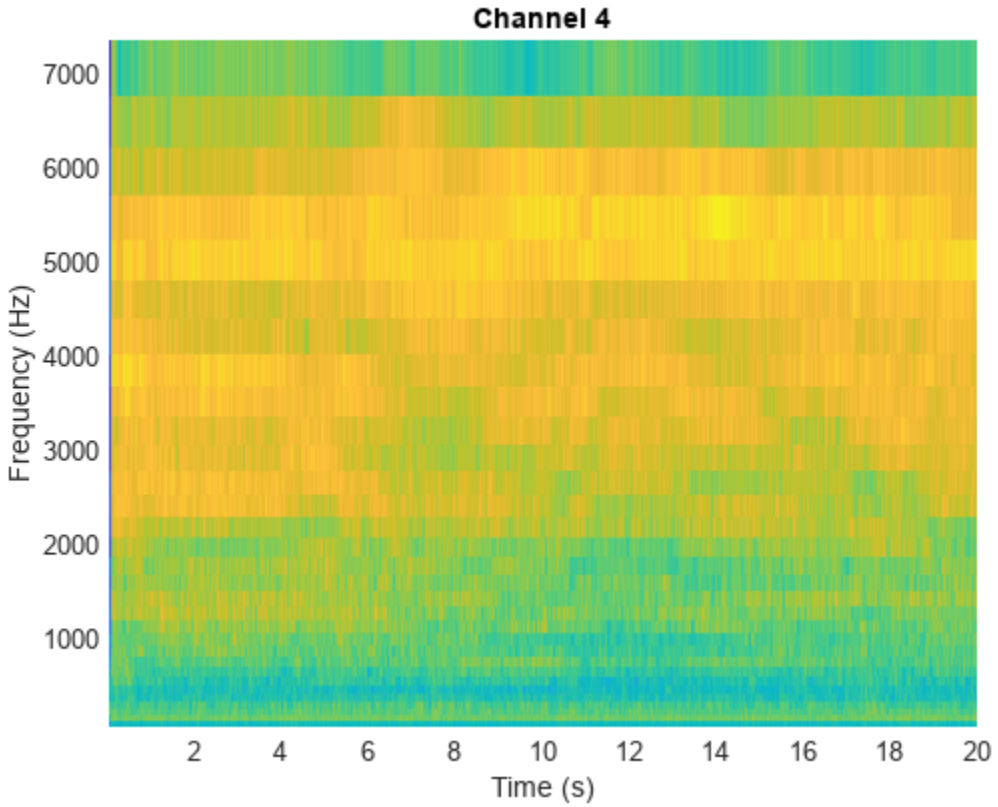
Get the mel spectrogram, filter bank center frequencies, and analysis window time instants of a multichannel audio signal. Use the center frequencies and time instants to plot the mel spectrogram for each channel.

```
[audioIn,fs] = audioread('AudioArray-16-16-4channels-20secs.wav');

[S,cF,t] = melSpectrogram(audioIn,fs);

S = 10*log10(S+eps); % Convert to dB for plotting

for i = 1:size(S,3)
    figure(i)
    surf(t,cF,S(:,:,i),'EdgeColor','none');
    xlabel('Time (s)')
    ylabel('Frequency (Hz)')
    view([0,90])
    title(sprintf('Channel %d',i))
    axis([t(1) t(end) cF(1) cF(end)])
end
```


Channel 1
7000
6000
5000
4000
3000
2000
1000
Frequency (Hz)
2
4
6
8
10
12
14
16
18
20
Time (s)


Channel 2
7000
6000
5000
4000
3000
2000
1000
Frequency (Hz)
2
4
6
8
10
12
14
16
18
20
Time (s)


Channel 3
7000
6000
5000
4000
3000
2000
1000
Frequency (Hz)
2
4
6
8
10
12
14
16
18
20
Time (s)



Input Arguments

`audioIn` — Audio input

column vector | matrix

Audio input, specified as a column vector or matrix. If specified as a matrix, the function treats columns as independent audio channels.

Data Types: `single` | `double`

`fs` — Input sample rate (Hz)

positive scalar

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

Data Types: `single` | `double`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `'WindowLength',1024`

`Window` — Window applied in time domain
`hamming(round(fs*0.3),'periodic')` (default) | vector

Window applied in time domain, specified as the comma-separated pair consisting of `'Window'` and a real vector. The number of elements in the vector must be in the range [1,`size(audioIn,1)`]. The number of elements in the vector must also be greater than `OverlapLength`.

Data Types: `single` | `double`

`OverlapLength` — Analysis window overlap length (samples)
`round(0.02*fs)` (default) | integer in the range `[0, (WindowLength - 1)]`

Analysis window overlap length in samples, specified as the comma-separated pair consisting of `'OverlapLength'` and an integer in the range `[0, (WindowLength - 1)]`.

Data Types: `single` | `double`

`FFTLength` — Number of DFT points
`WindowLength` (default) | positive integer

Number of points used to calculate the DFT, specified as the comma-separated pair consisting of `'FFTLength'` and a positive integer greater than or equal to `WindowLength`. If unspecified, `FFTLength` defaults to `WindowLength`.

Data Types: `single` | `double`

`NumBands` — Number of mel bandpass filters
`32` (default) | positive integer

Number of mel bandpass filters, specified as the comma-separated pair consisting of `'NumBands'` and a positive integer.

Data Types: `single` | `double`

`FrequencyRange` — Frequency range over which to compute mel spectrogram (Hz)
`[0 fs/2]` (default) | two-element row vector

Frequency range over which to compute the mel spectrogram in Hz, specified as the comma-separated pair consisting of `'FrequencyRange'` and a two-element row vector of monotonically increasing values in the range `[0, fs/2]`.

Data Types: `single` | `double`

`SpectrumType` — Type of mel spectrogram
`'power'` (default) | `'magnitude'`

Type of mel spectrogram, specified as the comma-separated pair consisting of `'SpectrumType'` and `'power'` or `'magnitude'`.

Data Types: `char` | `string`

`WindowNormalization` — Apply window normalization
`true` (default) | `false`

Apply window normalization, specified as the comma-separated pair consisting of `'WindowNormalization'` and `true` or `false`. When `WindowNormalization` is set to `true`, the power (or magnitude) in the mel spectrogram is normalized to remove the power (or magnitude) of the time domain `Window`.

Data Types: `char` | `string`

`FilterBankNormalization` — Type of filter bank normalization
`'bandwidth'` (default) | `'area'` | `'none'`

Type of filter bank normalization, specified as the comma-separated pair consisting of `'FilterBankNormalization'` and `'bandwidth'`, `'area'`, or `'none'`.

Data Types: `char` | `string`

Output Arguments

`S` — Mel spectrogram
column vector | matrix | 3-D array

Mel spectrogram, returned as a column vector, matrix, or 3-D array. The dimensions of `S` are *L*-by-*M*-by-*N*, where:

- *L* is the number of frequency bins in each mel spectrum. `NumBands` and `fs` determine *L*.
- *M* is the number of frames the audio signal is partitioned into. `size(audioIn,1)`, `WindowLength`, and `OverlapLength` determine *M*.
- *N* is the number of channels such that *N* = `size(audioIn,2)`.

Trailing singleton dimensions are removed from the output `S`.

Data Types: `single` | `double`

`F` — Center frequencies of mel bandpass filters (Hz)
row vector

Center frequencies of mel bandpass filters in Hz, returned as a row vector with length `size(S,1)`.

Data Types: `single` | `double`

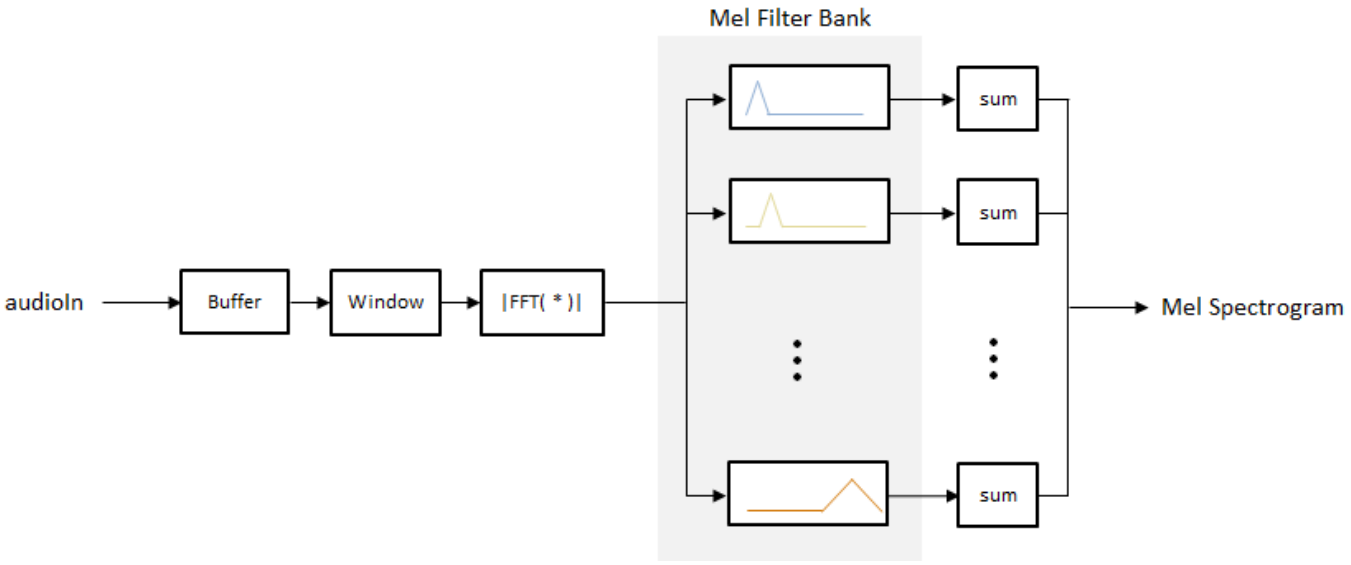
`T` — Location of each window of audio (s)
row vector

Location of each analysis window of audio in seconds, returned as a row vector length `size(S,2)`. The location corresponds to the center of each window.

Data Types: `single` | `double`

Algorithms

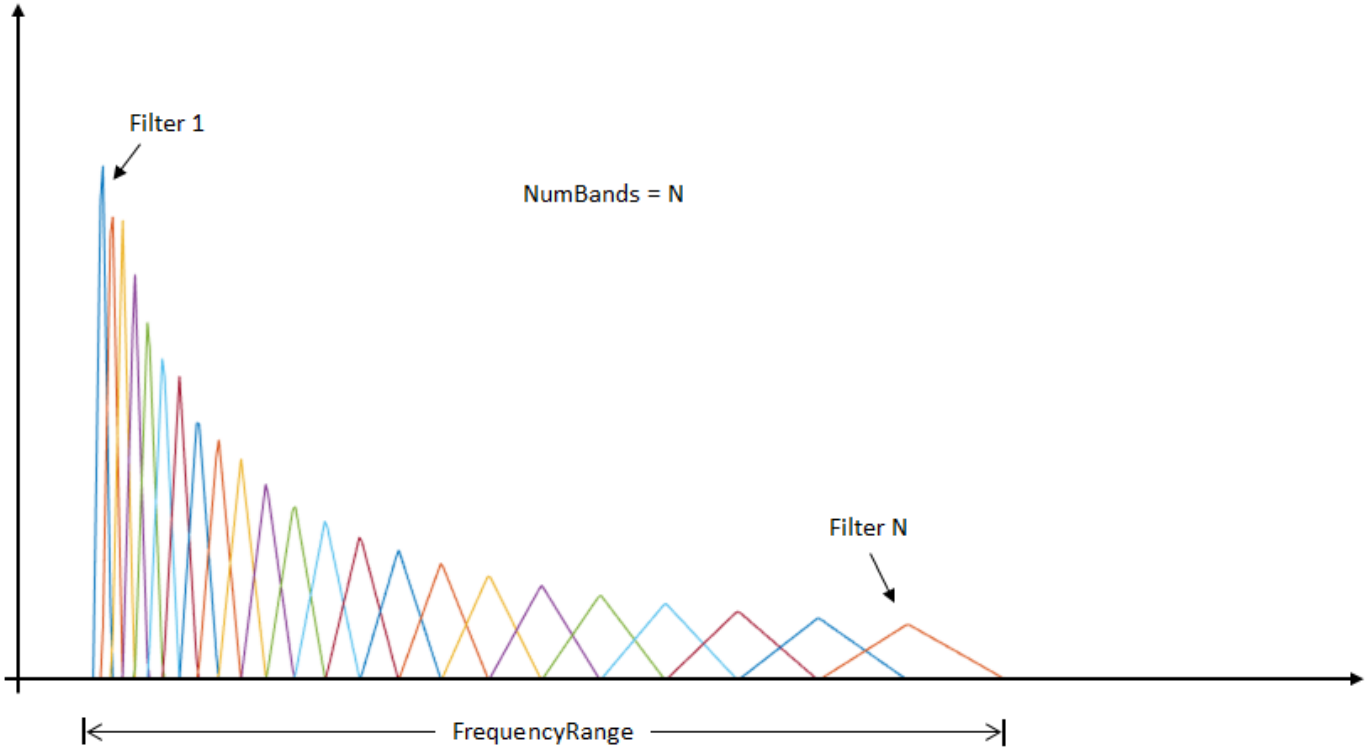
The `melSpectrogram` function follows the general algorithm to compute a mel spectrogram as described in [1].



In this algorithm, the audio input is first buffered into frames of `numel(Window)` number of samples. The frames are overlapped by `OverlapLength` number of samples. The specified `Window` is applied to each frame, and then the frame is converted to frequency-domain representation with `FFTLength` number of points. The frequency-domain representation can be either magnitude or power, specified by `SpectrumType`. If `WindowNormalization` is set to `true`, the spectrum is normalized by the window. Each frame of the frequency-domain representation passes through a mel filter bank. The spectral values output from the mel filter bank are summed, and then the channels are concatenated so that each frame is transformed to a `NumBands`-element column vector.

Filter Bank Design

The mel filter bank is designed as half-overlapped triangular filters equally spaced on the mel scale. `NumBands` controls the number of mel bandpass filters. `FrequencyRange` controls the band edges of the first and last filters in the mel filter bank. `FilterBankNormalization` specifies the type of normalization applied to the individual bands.



Version History

Introduced in R2019a

R2023a: Generate optimized C/C++ code for computing mel spectrogram

`melSpectrogram` supports optimized C/C++ code generation using single instruction, multiple data (SIMD) instructions.

R2020b: `WindowLength` will be removed in a future release
Behavior change in future release

The `WindowLength` parameter will be removed from the `melSpectrogram` function in a future release. Use the `Window` parameter instead.

In releases prior to R2020b, you could only specify the length of a time-domain window. The window was always designed as a periodic Hamming window. You can replace instances of the code

```
S = melSpectrogram(audioin,fs,'WindowLength',1024);
```

With this code:

```
S = melSpectrogram(audioIn,fs,'Window',hamming(1024,'periodic'));
```

References

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

Extended Capabilities

C/C++ Code Generation

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

The `melSpectrogram` function supports optimized code generation using single instruction, multiple data (SIMD) instructions. For more information about SIMD code generation, see “Generate SIMD Code for MATLAB Functions” (MATLAB Coder).

GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

See Also

`spectrogram` | `mfcc` | `gtcc` | `mdct` | `audioFeatureExtractor`

Topics

“Train Speech Command Recognition Model Using Deep Learning”

kbdwin

Kaiser-Bessel-derived window

Syntax

```
wdw = kbdwin(N)
wdw = kbdwin(N,Beta)
```

Description

`wdw = kbdwin(N)` returns an `N`-point Kaiser-Bessel-derived (KBD) window.

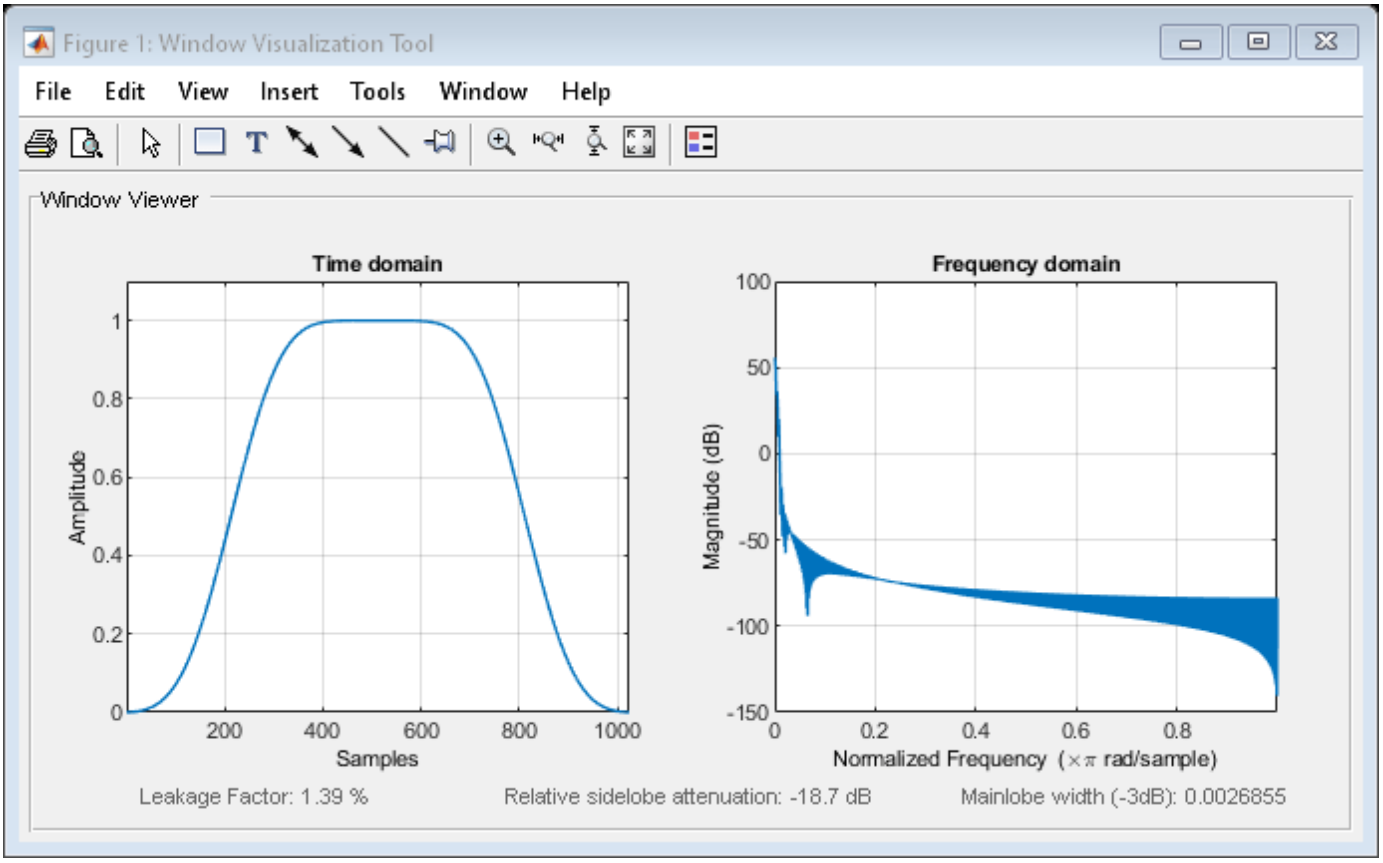
`wdw = kbdwin(N,Beta)` specifies the tuning parameter, `Beta`.

Examples

Create Kaiser-Bessel-Derived Window

Create a 1024-point Kaiser-Bessel-derived (KBD) window. Visualize the KBD window in the time and frequency domains using `wvtool`.

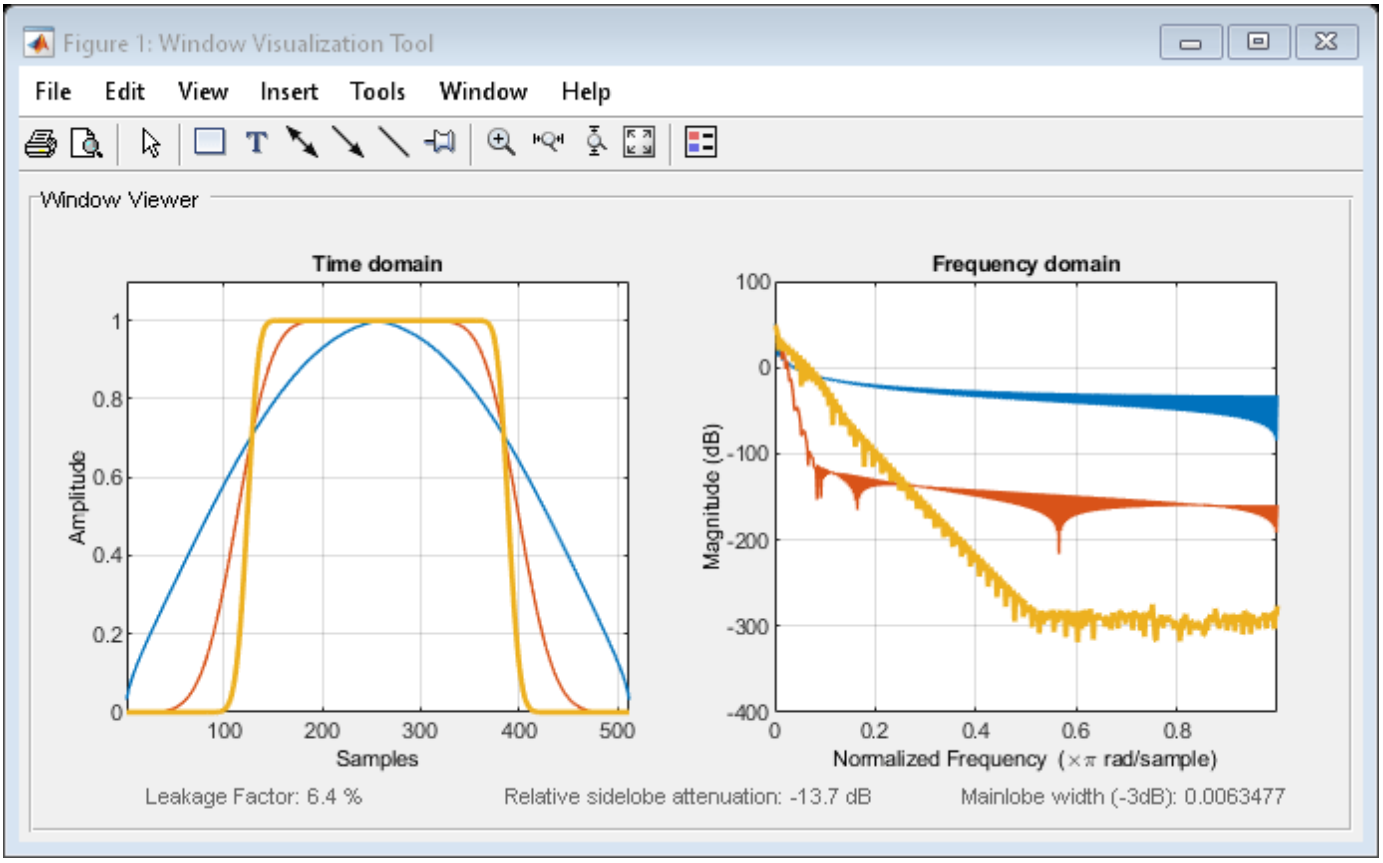
```
wdw = kbdwin(1024);
wvtool(wdw)
```



Effect of Tuning Parameter Beta

Create three 512-point KBD windows, with `Beta` set to 1, 10, and 100. Display the windows for comparison using `wvtool`.

```
N = 512;
beta1 = kbdwin(N,1);
beta10 = kbdwin(N,10);
beta100 = kbdwin(N,100);

wvtool(beta1,beta10,beta100)
```



Input Arguments

`N` — Number of points in KBD window

even positive integer scalar

Number of points in the KBD window, specified as an even positive integer scalar.

Data Types: `single` | `double`

`Beta` — Tuning parameter

5 (default) | nonnegative real scalar

Tuning parameter, specified as a nonnegative real scalar. If unspecified, `Beta` defaults to `5`.

Data Types: `single` | `double`

Output Arguments

`wdw` — Kaiser-Bessel-derived window

N-point column vector

Kaiser-Bessel-derived window, returned as an `N`-point column vector.

Algorithms

The coefficients of a Kaiser-Bessel-derived window are computed using the equation:

$$wdw[\mathrm{n}] = \begin{cases} \sqrt{\dfrac{\sum_{i=1}^{n} w[i]}{\sum_{i=1}^{N/2+1} w[i]}} & \text{if } 1 \le n < \left(N/2\right) \\ \sqrt{\dfrac{\sum_{i=1}^{N-n} w[i]}{\sum_{i=1}^{N/2+1} w[i]}} & \text{if } \left(N/2 + 1\right) \le n < N \end{cases}$$

where w is a Kaiser window designed using the `kaiser` function:

```
w = kaiser(N/2+1,Beta*pi)
```

where `N` is the number of points in the KBD window and `Beta` is the tuning parameter.

Version History

Introduced in R2019a

References

[1] Bosi, Marina, and Richard E. Goldberg. *Introduction to Digital Audio Coding and Standards*. Dordrecht: Kluwer, 2003.

Extended Capabilities

C/C++ Code Generation

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

See Also

`kaiser` | `window` | `mdct`

mdct

Modified discrete cosine transform

Syntax

```
Y = mdct(X,win)
Y = mdct(X,win,Name,Value)
[Y,S,Z] = mdct( ___ )
```

Description

`Y = mdct(X,win)` returns the modified discrete cosine transform (MDCT) of `X`. Before the MDCT is calculated, `X` is buffered into 50% overlapping frames that are each multiplied by the time window `win`. The function treats each column of `X` as an independent channel.

`Y = mdct(X,win,Name,Value)` sets each property `Name` to the specified `Value`. Unspecified properties have default values.

`[Y,S,Z] = mdct( ___ )` returns the modified discrete sine transform (MDST), `S`, and the odd discrete Fourier transform (ODFT), `Z`.

Examples

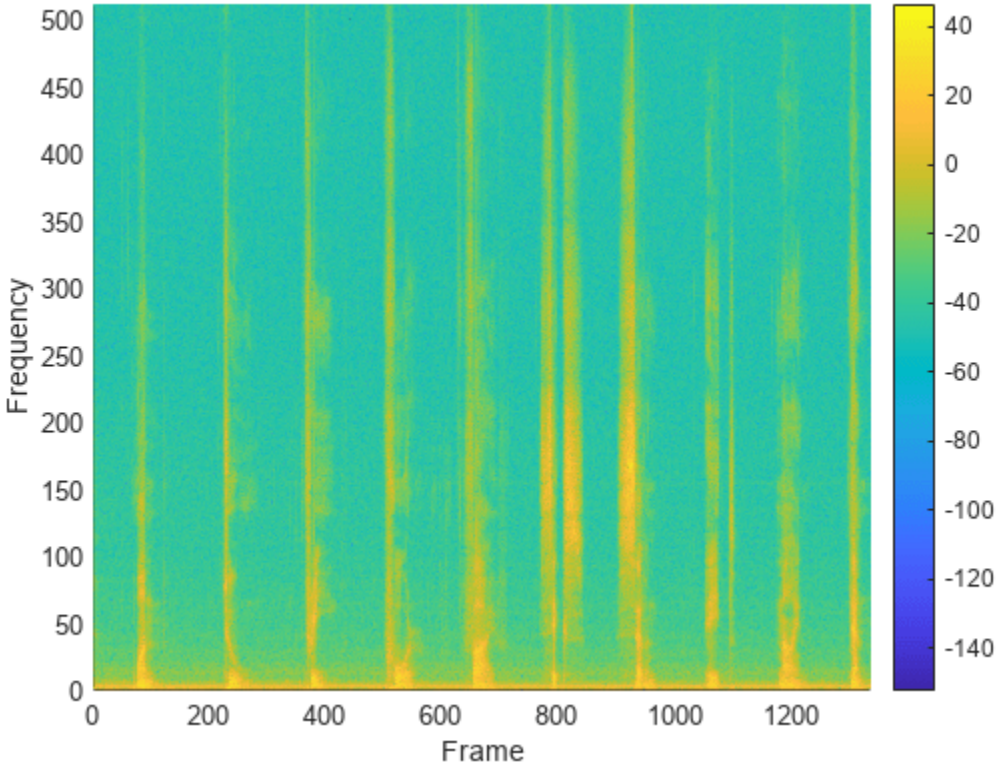
Calculate MDCT

Read in an audio file and then calculate the MDCT using a 1024-point Kaiser-Bessel-derived window.

```
audioIn = audioread('Counting-16-44p1-mono-15secs.wav');

coef = mdct(audioIn,kbdwin(1024));
```

Plot the power of the MDCT coefficients over time.

```
surf(pow2db(coef.^2),'EdgeColor','none');
view([0 90])
xlabel('Frame')
ylabel('Frequency')
axis([0 size(coef,2) 0 size(coef,1)])
colorbar
```



Effect of Input Padding on Perfect Reconstruction

To enable perfect reconstruction, the `mdct` function zero-pads the front and back of the audio input signal. The signal returned from `imdct` removes the zero padding added for perfect reconstruction.

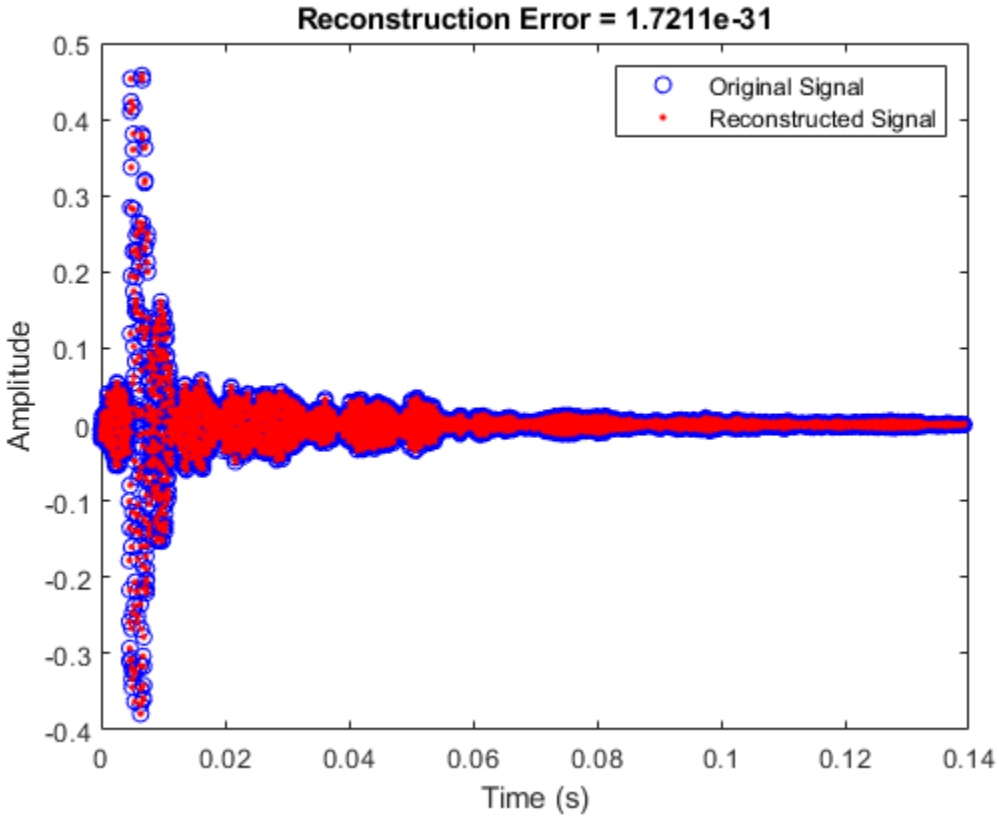
Read in an audio file, create a 2048-point Kaiser-Bessel-derived window, and then clip the audio signal so that its length is a multiple of 2048.

```
[x,fs] = audioread('Click-16-44p1-mono-0.2secs.wav');
win = kbdwin(2048);

xClipped = x(1:end - rem(size(x,1),numel(win)));
```

Convert the signal to the frequency domain, and then reconstruct it back in the time domain. Plot the original and reconstructed signals and display the reconstruction error.

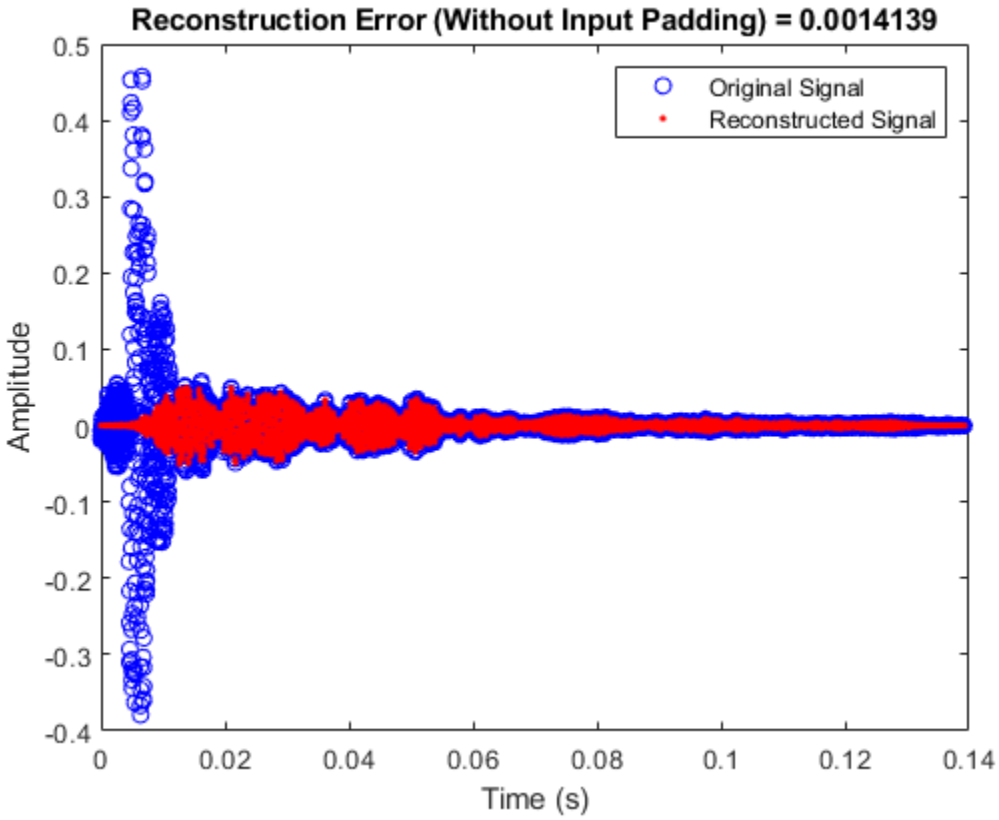
```
C = mdct(xClipped,win);
y = imdct(C,win);

figure(1)
t = (0:size(xClipped,1)-1)'/fs;
plot(t,xClipped,'bo',t,y,'r.')
legend('Original Signal','Reconstructed Signal')
title(strcat("Reconstruction Error = ",num2str(mean((xClipped-y).^2))))
```

```
xlabel('Time (s)')
ylabel('Amplitude')
```



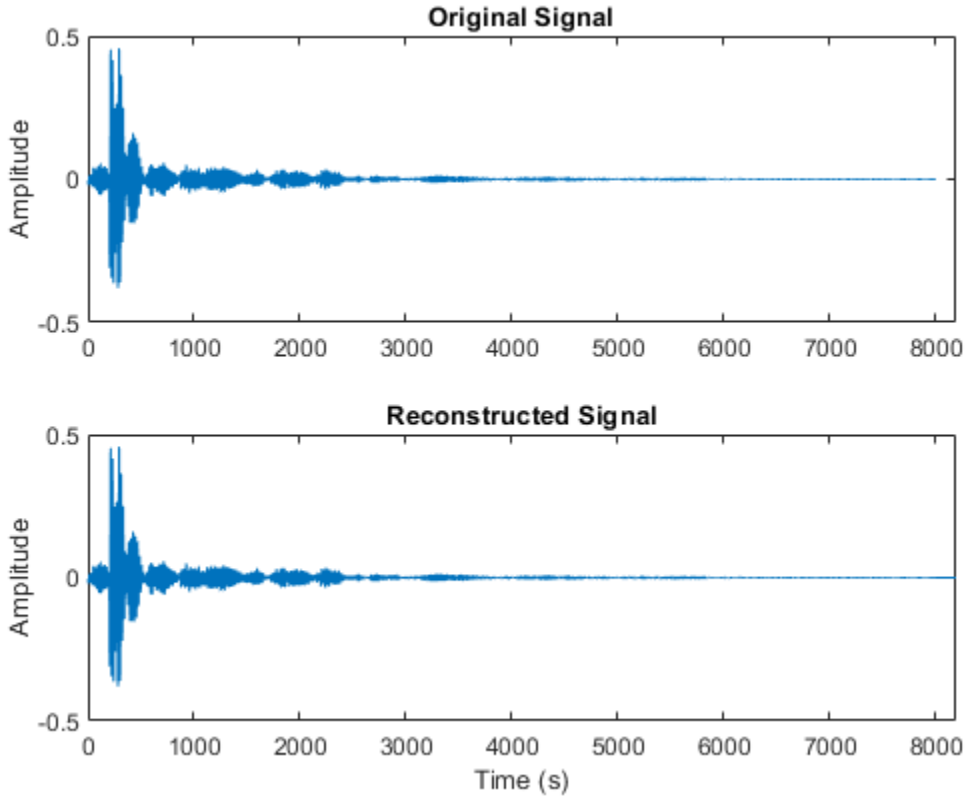
You can perform the MDCT and IMDCT without input padding using the `PadInput` name-value pair. However, there will be a reconstruction error in the first half-frame and last half-frame of the signal.

```
C = mdct(xClipped,win,'PadInput',false);
y = imdct(C,win,'PadInput',false);

figure(2)
t = (0:size(xClipped,1)-1)'/fs;
plot(t,xClipped,'bo',t,y,'r.')
legend('Original Signal','Reconstructed Signal')
title(strcat("Reconstruction Error (Without Input Padding) = ",num2str(mean((xClipped-y).^2))))
xlabel('Time (s)')
ylabel('Amplitude')
```



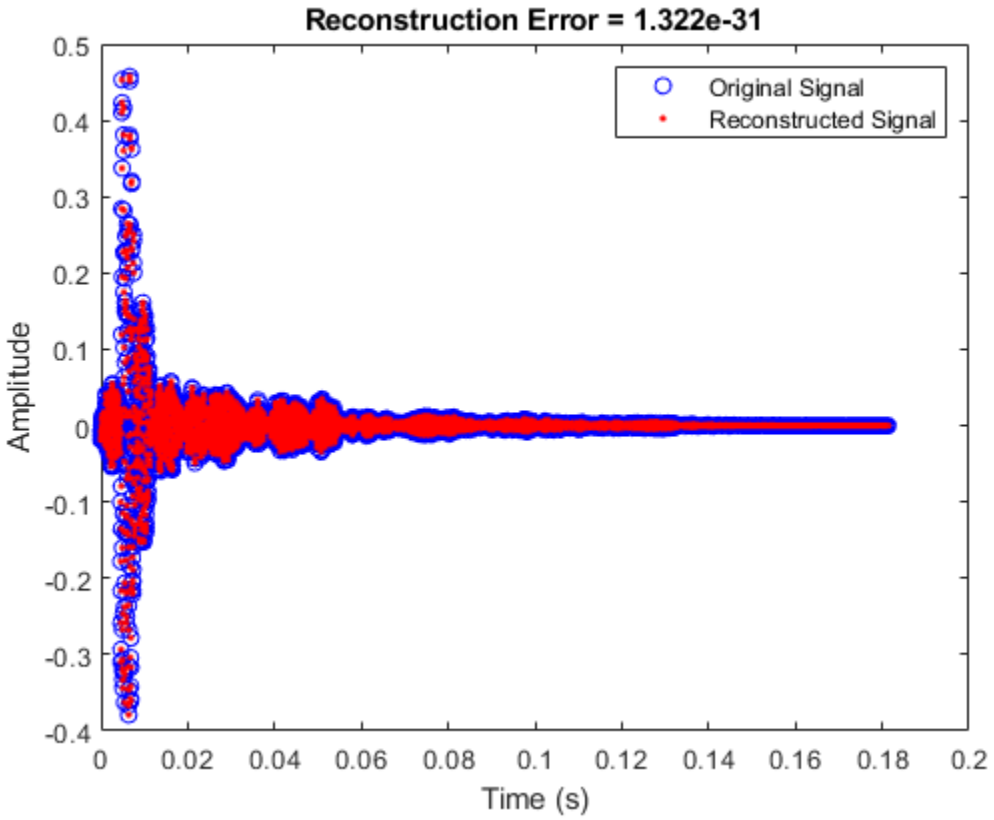
If you specify an input signal to the `mdct` that is not a multiple of the window length, then the input signal is padded with zeros. Pass the original unclipped signal through the transform pair and compare the original signal and the reconstructed signal.

```
C = mdct(x,win);
y = imdct(C,win);

figure(3)

subplot(2,1,1)
plot(x)
title('Original Signal')
ylabel('Amplitude')
axis([0,max(size(y,1),size(x,1)),-0.5,0.5])

subplot(2,1,2)
plot(y)
title('Reconstructed Signal')
xlabel('Time (s)')
ylabel('Amplitude')
axis([0,max(size(y,1),size(x,1)),-0.5,0.5])
```



The reconstructed signal is padded with zeros at the back end. Remove the zero-padding from the reconstructed signal, plot the original and reconstructed signal, and then display the reconstruction error.

```
figure(4)
y = y(1:size(x,1));
t = (0:size(x,1)-1)'/fs;
plot(t,x,'bo',t,y,'r.')
legend('Original Signal','Reconstructed Signal')
title(strcat("Reconstruction Error = ",num2str(mean((x-y).^2))))
xlabel('Time (s)')
ylabel('Amplitude')
```



MDCT and IMDCT for Streaming Audio

Create a `dsp.AudioFileReader` object to read in audio data frame-by-frame. Create a `dsp.SignalSink` to log the reconstructed signal for comparison. Create a `dsp.AsyncBuffer` to buffer the input stream.

```
fileReader = dsp.AudioFileReader('FunkyDrums-44p1-stereo-25secs.mp3');
logger = dsp.SignalSink;
buff = dsp.AsyncBuffer;
```

Create a 512-point Kaiser-Bessel-derived window.

```
N = 512;
win = kbdwin(N);
```

In an audio stream loop:

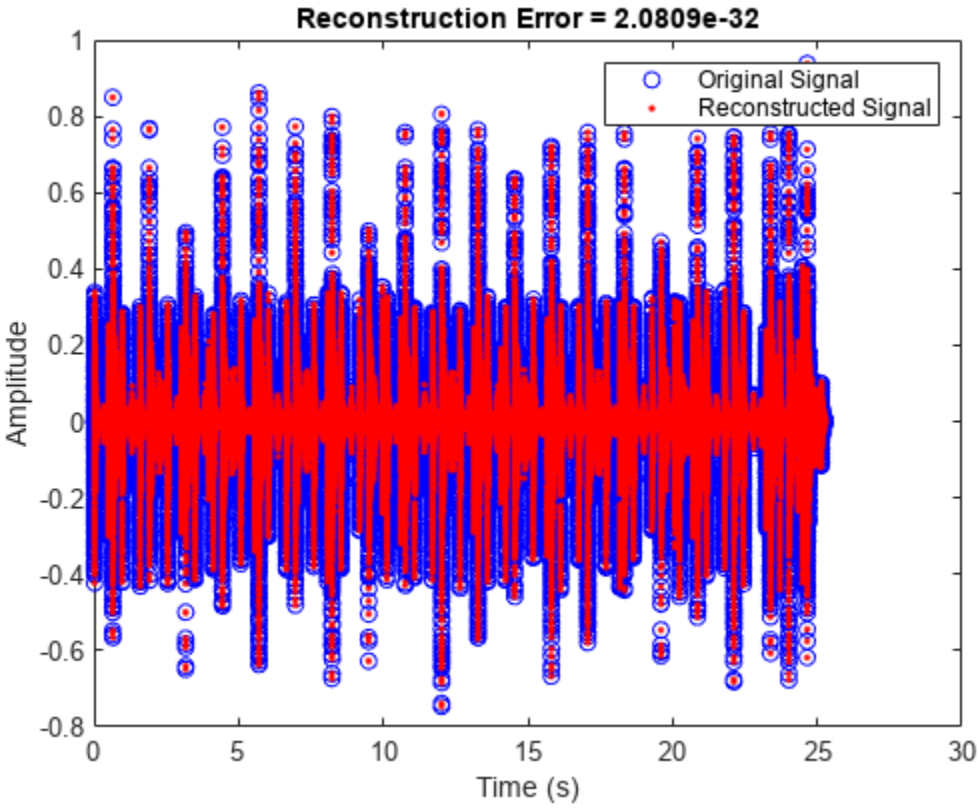
1. Read a frame of data from the file.
2. Write the frame of data to the async buffer.
3. If half a frame of data is present, read from the buffer and then perform the transform pair. Overlap-add the current output from `imdct` with the previous output, and log the results. Update the memory.

```
mem = zeros(N/2,2); % initialize an empty memory
```

```
while ~isDone(fileReader)
    audioIn = fileReader();
    write(buff,audioIn);

    while buff.NumUnreadSamples >= N/2
        x = read(buff,N,N/2);
        C = mdct(x,win,'PadInput',false);
        y = imdct(C,win,'PadInput',false);

        logger(y(1:N/2,:)+mem)
        mem = y(N/2+1:end,:);
    end

end

% Perform the transform pair one last time with a zero-padded final signal.
x = read(buff,N,N/2);
C = mdct(x,win,'PadInput',false);
y = imdct(C,win,'PadInput',false);
logger(y(1:N/2,:)+mem)

reconstructedSignal = logger.Buffer;
```

Read in the entire original audio signal. Trim the front and back zero padding from the reconstructed signal for comparison. Plot one channel of the original and reconstructed signals and display the reconstruction error.

```
[originalSignal,fs] = audioread(fileReader.Filename);
signalLength = size(originalSignal,1);
reconstructedSignal = reconstructedSignal((N/2+1):(N/2+1)+signalLength-1,:);

t = (0:size(originalSignal,1)-1)'/fs;
plot(t,originalSignal(:,1),'bo',t,reconstructedSignal(:,1),'r.')
legend('Original Signal','Reconstructed Signal')
title(strcat("Reconstruction Error = ", ...
      num2str(mean((originalSignal-reconstructedSignal).^2,'all'))))
xlabel('Time (s)')
ylabel('Amplitude')
```



Input Arguments

`X` — Input array

column vector | matrix

Input array, specified as a column vector or matrix. If specified as a matrix, the columns are treated as independent audio channels.

Data Types: `single` | `double`

`win` — Window applied in time domain

even-length vector

Window applied in the time domain, specified as an even-length vector. The transform performed by `mdct` has the same number of points as `win`. To enable perfect reconstruction, use a window that satisfies the Princen-Bradley condition ($w_n^2 + w_{n+N}^2 = 1$), such as a sine window or `kbdwin`.

Data Types: `single` | `double`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `'PadInput',false`

`PadInput` — Flag to pad input array
`true` (default) | `false`

Flag to pad input array, specified as the comma-separated pair consisting of '`PadInput`' and `true` or `false`. If set to `true`, zero-padding is added to the input `X` at both ends to enable perfect reconstruction. The number of zeros at each end is `numel(win)/2`.

Data Types: `logical`

Output Arguments

`Y` — Modified discrete cosine transform
vector | matrix | 3-D array

Modified discrete cosine transform (MDCT), returned as a vector, matrix, or 3-D array. The dimensions of `Y` are *L*-by-*M*-by-*N*, where:

- *L* –– Number of points in the frequency-domain representation of each frame, equal to `numel(win)/2`.
- *M* –– Number of frames the input array is partitioned into.
  - If `PadInput` is set to `true`, `M = ceil(2*size(X,1)/numel(win))+1`.
  - If `PadInput` is set to `false`, `M = ceil(2*size(X,1)/numel(win))-1`.
- *N* –– Number of channels, equal to `size(X,2)`.

Trailing singleton dimensions are removed from the output `Y`.

Data Types: `single` | `double`

`S` — Modified discrete sine transform
vector | matrix | 3-D array

Modified discrete sine transform (MDST), returned as a vector, matrix, or 3-D array. The dimensions of `S` are the same as the MDCT output, `Y`.

Data Types: `single` | `double`

`Z` — Half-sided odd discrete Fourier transform
vector | matrix | 3-D array

Half-sided odd discrete Fourier transform (ODFT), returned as a vector, matrix, or 3-D array of complex numbers. The dimensions of `Z` are the same as the MDCT output, `Y`.

To construct the complete (two-sided) ODFT, mirror the half-sided ODFT: `cat(1,Z,conj(flip(Z,1)))`.

Data Types: `single` | `double`
Complex Number Support: Yes

Algorithms

The modified discrete cosine transform is a time-frequency transform. Given an input signal `X` and window `win`, the `mdct` function performs the following steps for each independent channel:

1. The frame size is the number of elements in the specified window, N = `numel(win)`. By default, `PadInput` is set to `true`, so the input signal `X` is padded with $N/2$ zeros on the front and back. If the input signal is not divisible by N, additional padding is added on the back. After padding, the input signal is buffered into 50% overlapped frames.
2. Each frame of the buffered and padded input signal is multiplied by the window, `win`.
3. The input is converted into a frequency representation using the modified discrete cosine transform:

$$Y(k) = \sum_{n=0}^{N-1} X(n)\cos\left[\frac{\pi}{(N/2)}\left(n + \frac{(N/2)+1}{2}\right)\left(k + \frac{1}{2}\right)\right], \quad k = 0, 1, \ldots, (N/2) - 1$$

To take advantage of the FFT algorithm, the MDCT is calculated by first calculating the odd DFT:

$$Y_O(k) = \sum_{n=0}^{N-1} X(n)e^{-j\frac{\pi n}{N}(2k+1)}, \quad k = 0, 1, \ldots, N-1$$

and then calculating the MDCT:

$$Y(k) = \Re e\left\{Y_o(k)\right\}\cos\left(\frac{\pi}{N}\left(k + \frac{1}{2}\right)\left(1 + \frac{N}{2}\right)\right), \quad k = 0, 1, \ldots, (N/2) - 1$$

If a second argument is requested from the `mdct` function, the modified discrete sine transform (MDST) is also computed and returned:

$$X(k) = \Im m\left\{X_o(k)\right\}\sin\left(\frac{\pi}{N}\left(k + \frac{1}{2}\right)\left(1 + \frac{N}{2}\right)\right), \quad k = 0, 1, \ldots, (N/2) - 1$$

Version History

Introduced in R2019a

References

[1] Princen, J., A. Johnson, and A. Bradley. "Subband/Transform Coding Using Filter Bank Designs Based on Time Domain Aliasing Cancellation." *IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*. 1987, pp. 2161–2164.

[2] Princen, J., and A. Bradley. "Analysis/Synthesis Filter Bank Design Based on Time Domain Aliasing Cancellation." *IEEE Transactions on Acoustics, Speech, and Signal Processing.* Vol. 34, Issue 5, 1986, pp. 1153–1161.

Extended Capabilities

C/C++ Code Generation

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

See Also

`imdct` | `kbdwin` | `spectrogram`

Topics

“Vorbis Decoder”

imdct

Inverse modified discrete cosine transform

Syntax

```
X = imdct(Y,win)
X = imdct(Y,win,Name,Value)
```

Description

`X = imdct(Y,win)` returns the inverse modified discrete cosine transform (IMDCT) of `Y`, followed by multiplication with time window `win` and overlap-addition of the frames with 50% overlap.

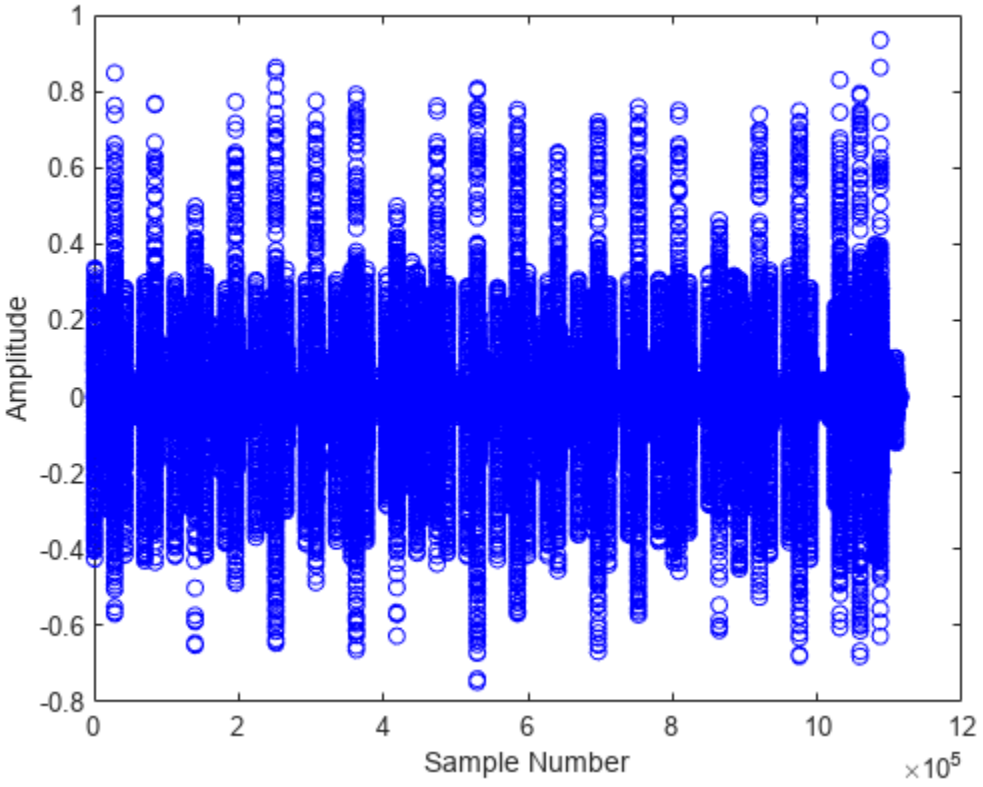
`X = imdct(Y,win,Name,Value)` sets each property `Name` to the specified `Value`. Unspecified properties have default values.

Examples

Calculate IMDCT

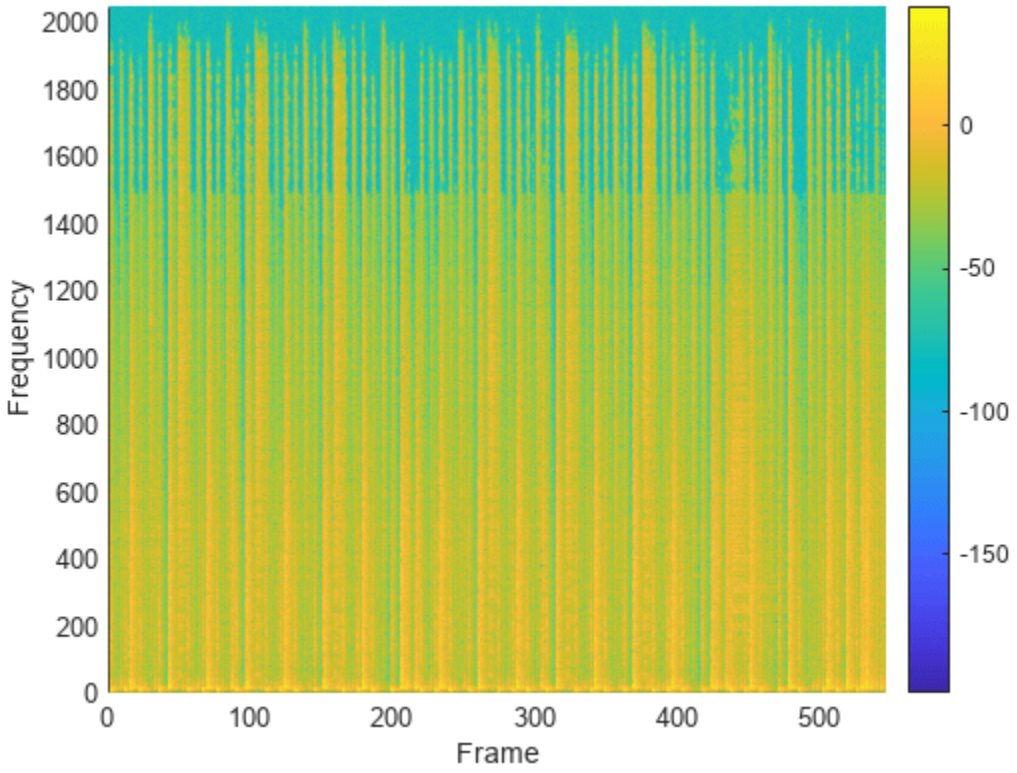
Read in an audio file, convert it to mono, and then plot it.

```
audioIn = audioread('FunkyDrums-44p1-stereo-25secs.mp3');
audioIn = mean(audioIn,2);

figure(1)
plot(audioIn,'bo')
ylabel('Amplitude')
xlabel('Sample Number')
```



Calculate the MDCT using a 4096-point sine window. Plot the power of the MDCT coefficients over time.

```
N = 4096;
wdw = sin(pi*((1:N)-0.5)/N);

C = mdct(audioIn,wdw);

figure(2)
surf(pow2db(C.*conj(C)),'EdgeColor','none');
view([0 90])
xlabel('Frame')
ylabel('Frequency')
axis([0 size(C,2) 0 size(C,1)])
colorbar
```

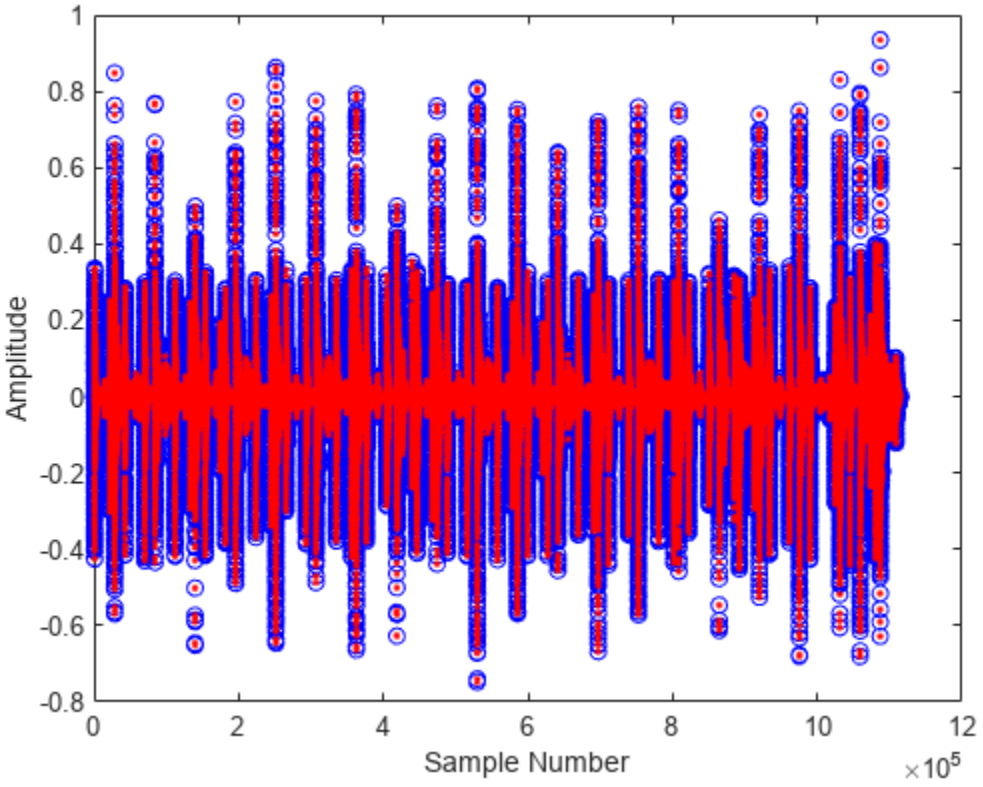


Transform the representation back to the time domain. Verify the perfect reconstruction property by computing the mean squared error. Plot the reconstructed signal over the original signal.

```
audioReconstructed = imdct(C,wdw);
err = mean((audioIn-audioReconstructed(1:size(audioIn,1),:)).^2)
```

```
err = 9.5889e-31
```

```
figure(1)
hold on
plot(audioReconstructed,'r.')
ylabel('Amplitude')
xlabel('Sample Number')
```



Effect of Input Padding on Perfect Reconstruction

To enable perfect reconstruction, the `mdct` function zero-pads the front and back of the audio input signal. The signal returned from `imdct` removes the zero padding added for perfect reconstruction.

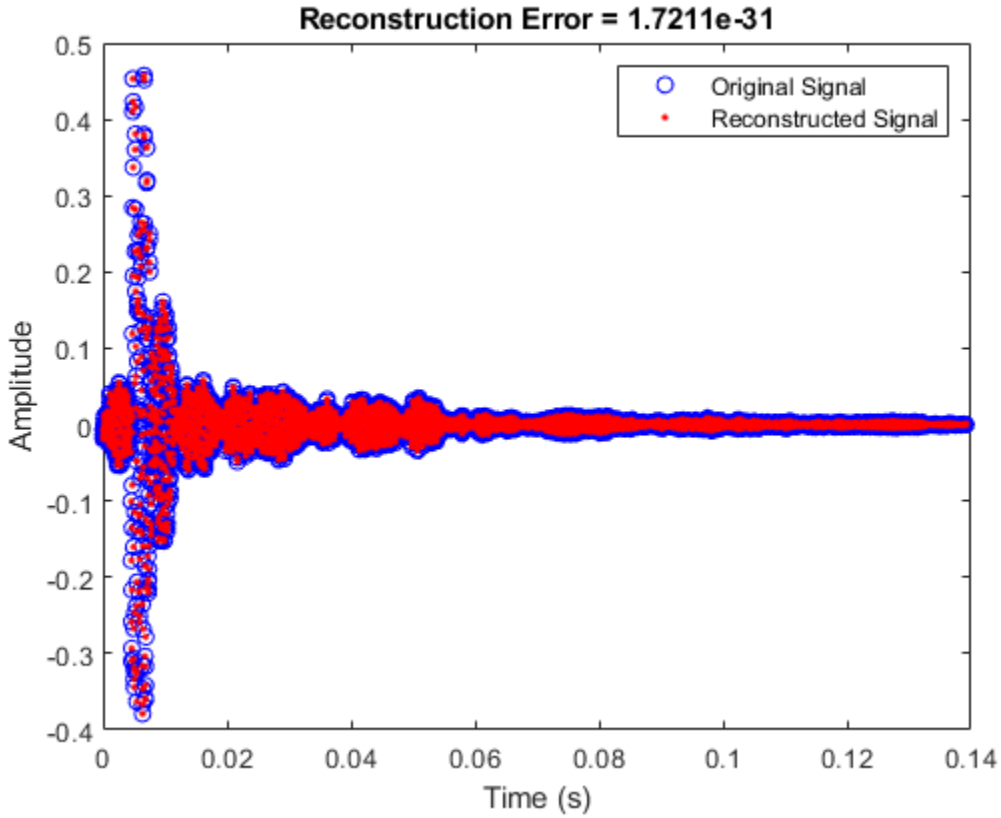
Read in an audio file, create a 2048-point Kaiser-Bessel-derived window, and then clip the audio signal so that its length is a multiple of 2048.

```
[x,fs] = audioread('Click-16-44p1-mono-0.2secs.wav');
win = kbdwin(2048);

xClipped = x(1:end - rem(size(x,1),numel(win)));
```

Convert the signal to the frequency domain, and then reconstruct it back in the time domain. Plot the original and reconstructed signals and display the reconstruction error.

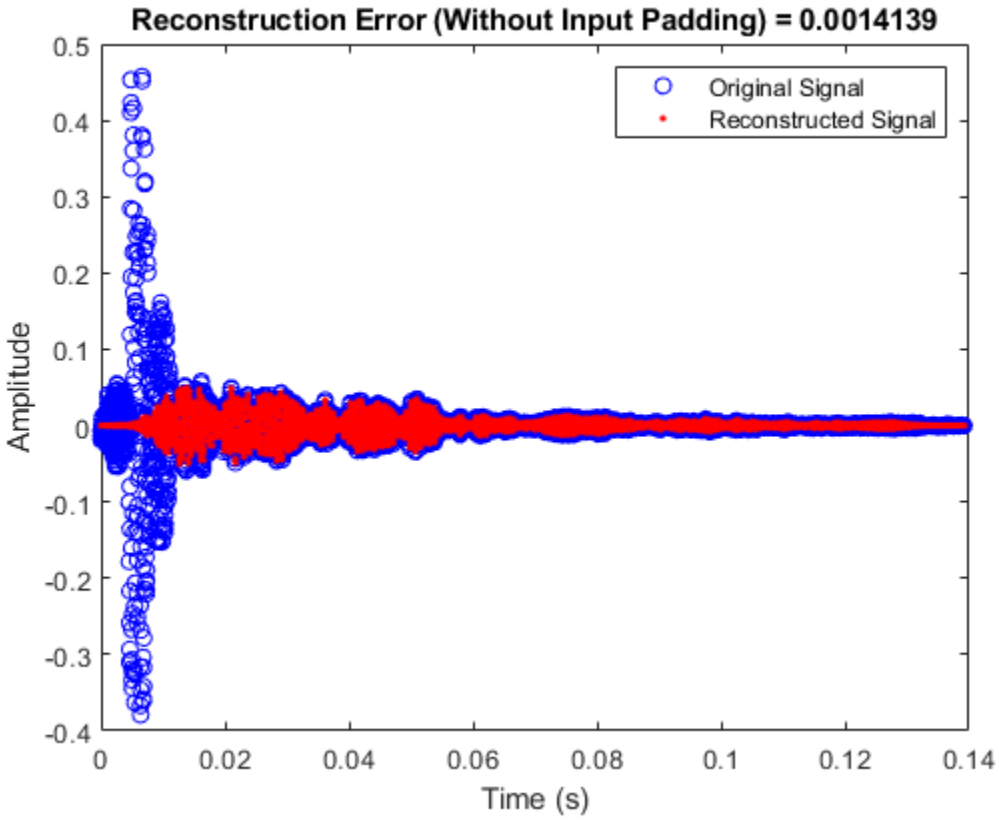
```
C = mdct(xClipped,win);
y = imdct(C,win);

figure(1)
t = (0:size(xClipped,1)-1)'/fs;
plot(t,xClipped,'bo',t,y,'r.')
legend('Original Signal','Reconstructed Signal')
title(strcat("Reconstruction Error = ",num2str(mean((xClipped-y).^2))))
```

```
xlabel('Time (s)')
ylabel('Amplitude')
```



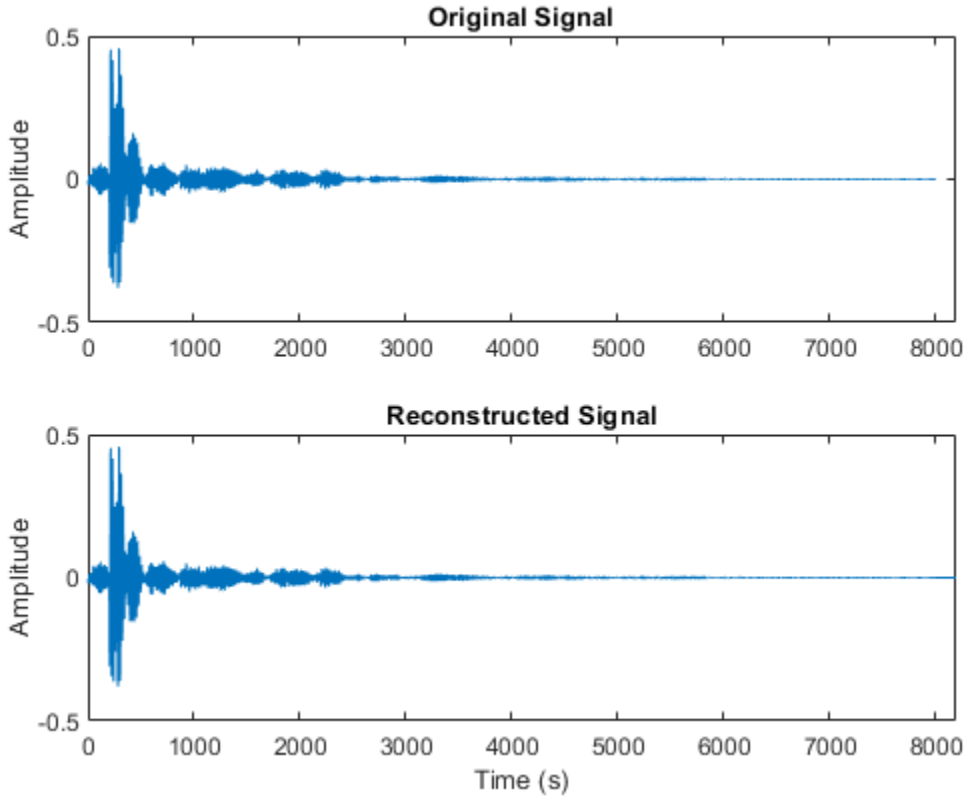
You can perform the MDCT and IMDCT without input padding using the `PadInput` name-value pair. However, there will be a reconstruction error in the first half-frame and last half-frame of the signal.

```
C = mdct(xClipped,win,'PadInput',false);
y = imdct(C,win,'PadInput',false);

figure(2)
t = (0:size(xClipped,1)-1)'/fs;
plot(t,xClipped,'bo',t,y,'r.')
legend('Original Signal','Reconstructed Signal')
title(strcat("Reconstruction Error (Without Input Padding) = ",num2str(mean((xClipped-y).^2))))
xlabel('Time (s)')
ylabel('Amplitude')
```



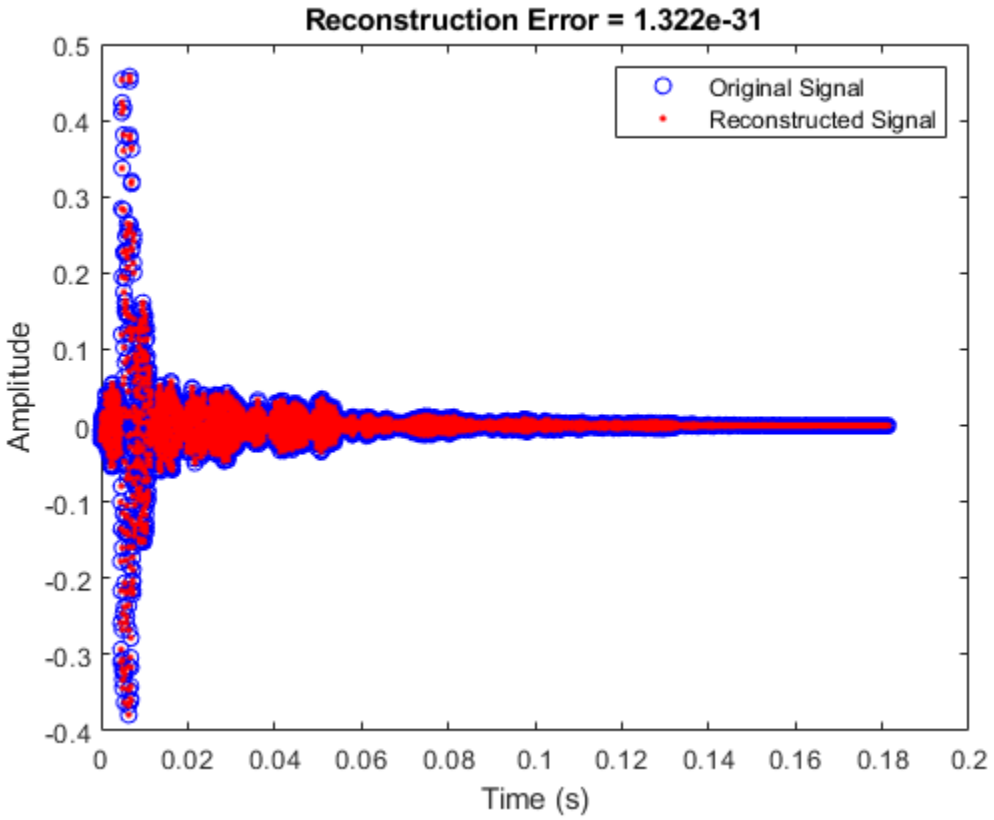
If you specify an input signal to the `mdct` that is not a multiple of the window length, then the input signal is padded with zeros. Pass the original unclipped signal through the transform pair and compare the original signal and the reconstructed signal.

```
C = mdct(x,win);
y = imdct(C,win);

figure(3)

subplot(2,1,1)
plot(x)
title('Original Signal')
ylabel('Amplitude')
axis([0,max(size(y,1),size(x,1)),-0.5,0.5])

subplot(2,1,2)
plot(y)
title('Reconstructed Signal')
xlabel('Time (s)')
ylabel('Amplitude')
axis([0,max(size(y,1),size(x,1)),-0.5,0.5])
```



The reconstructed signal is padded with zeros at the back end. Remove the zero-padding from the reconstructed signal, plot the original and reconstructed signal, and then display the reconstruction error.

```
figure(4)
y = y(1:size(x,1));
t = (0:size(x,1)-1)'/fs;
plot(t,x,'bo',t,y,'r.')
legend('Original Signal','Reconstructed Signal')
title(strcat("Reconstruction Error = ",num2str(mean((x-y).^2))))
xlabel('Time (s)')
ylabel('Amplitude')
```



MDCT and IMDCT for Streaming Audio

Create a `dsp.AudioFileReader` object to read in audio data frame-by-frame. Create a `dsp.SignalSink` to log the reconstructed signal for comparison. Create a `dsp.AsyncBuffer` to buffer the input stream.

```
fileReader = dsp.AudioFileReader('FunkyDrums-44p1-stereo-25secs.mp3');
logger = dsp.SignalSink;
buff = dsp.AsyncBuffer;
```

Create a 512-point Kaiser-Bessel-derived window.

```
N = 512;
win = kbdwin(N);
```

In an audio stream loop:

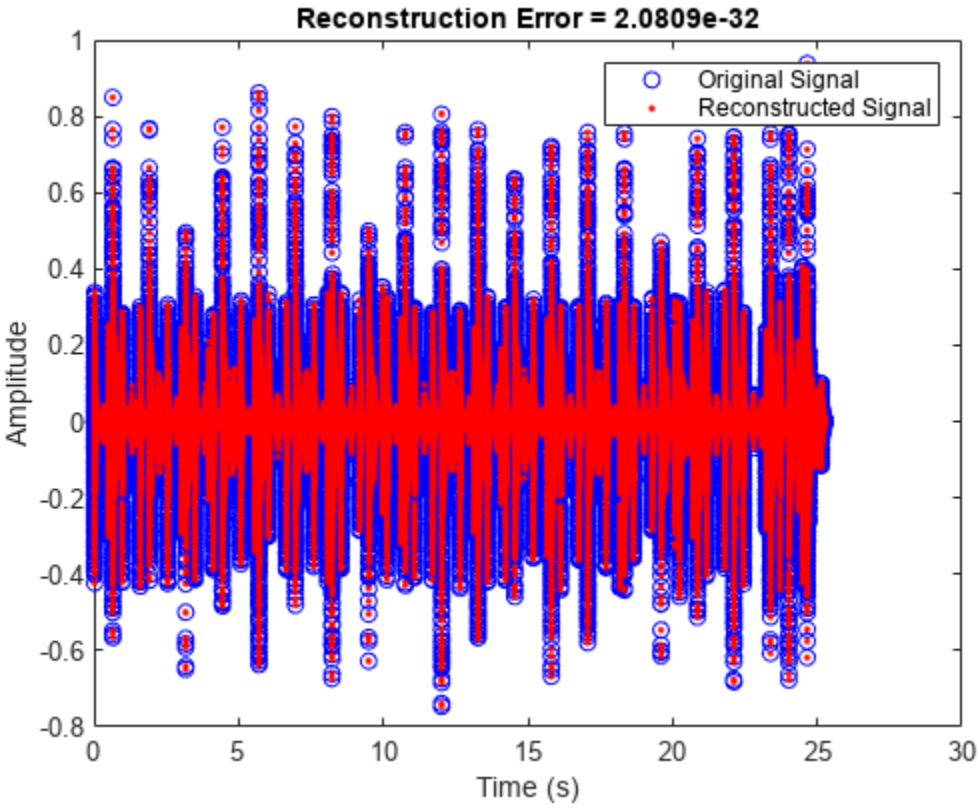
1. Read a frame of data from the file.
2. Write the frame of data to the async buffer.
3. If half a frame of data is present, read from the buffer and then perform the transform pair. Overlap-add the current output from `imdct` with the previous output, and log the results. Update the memory.

```
mem = zeros(N/2,2); % initialize an empty memory
```

```
while ~isDone(fileReader)
    audioIn = fileReader();
    write(buff,audioIn);

    while buff.NumUnreadSamples >= N/2
        x = read(buff,N,N/2);
        C = mdct(x,win,'PadInput',false);
        y = imdct(C,win,'PadInput',false);

        logger(y(1:N/2,:)+mem)
        mem = y(N/2+1:end,:);
    end

end

% Perform the transform pair one last time with a zero-padded final signal.
x = read(buff,N,N/2);
C = mdct(x,win,'PadInput',false);
y = imdct(C,win,'PadInput',false);
logger(y(1:N/2,:)+mem)

reconstructedSignal = logger.Buffer;
```

Read in the entire original audio signal. Trim the front and back zero padding from the reconstructed signal for comparison. Plot one channel of the original and reconstructed signals and display the reconstruction error.

```
[originalSignal,fs] = audioread(fileReader.Filename);
signalLength = size(originalSignal,1);
reconstructedSignal = reconstructedSignal((N/2+1):(N/2+1)+signalLength-1,:);

t = (0:size(originalSignal,1)-1)'/fs;
plot(t,originalSignal(:,1),'bo',t,reconstructedSignal(:,1),'r.')
legend('Original Signal','Reconstructed Signal')
title(strcat("Reconstruction Error = ", ...
      num2str(mean((originalSignal-reconstructedSignal).^2,'all'))))
xlabel('Time (s)')
ylabel('Amplitude')
```



Input Arguments

Y — Modified discrete cosine transform

vector | matrix | 3-D array

Modified discrete cosine transform (MDCT), specified as a vector, matrix, or 3-D array. The dimensions of `Y` are interpreted as output from the `mdct` function. If `Y` is an *L*-by-*M*-by-*N* array, the dimensions are interpreted as:

- *L* –- Number of points in the frequency-domain representation of each frame. *L* must be half the number of points in the window, `win`.
- *M* –- Number of frames.
- *N* –- Number of channels.

Data Types: `single` | `double`

win — Window applied in time domain

vector

Window applied in the time domain, specified as vector. The length of win must be twice the number of rows of `Y`: `numel(win)==2*size(Y,1)`. To enable perfect reconstruction, use the same window used in the forward transformation `mdct`.

Data Types: `single` | `double`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `'PadInput',false`

`PadInput` — Flag if input was padded

`true` (default) | `false`

Flag if input to the forward `mdct` was padded. If set to `true`, the output is truncated at both ends to remove the zero-padding that the forward `mdct` added.

Data Types: `logical`

Output Arguments

`X` — Inverse modified discrete cosine transform

column vector | matrix

Inverse modified discrete cosine transform (IMDCT) of input array `Y`, returned as a column vector or matrix of independent channels.

Data Types: `single` | `double`

Algorithms

The inverse modified discrete cosine transform is a time-frequency transform. Given a frequency domain input signal `Y` and window `win`, the `imdct` function performs the follows steps for each independent channel:

1. Each frame of the input is converted into a time-domain representation:

$$X(n) = \sum_{k=0}^{\frac{N}{2}-1} Y(k)\cos\left[\frac{\pi}{(N/2)}\left(n + \frac{(N/2)+1}{2}\right)\left(k + \frac{1}{2}\right)\right], \quad n = 0, 1, \ldots, N-1$$

   where N is the number of elements in `win`.

2. Each frame of the time-domain signal is multiplied by the window, `win`.

3. The frames are overlap-added with 50% overlap to construct a contiguous time-domain signal. If `PadInput` is set to true, the `imdct` function assumes the original input signal in the forward transform (`mdct`) was padded with $N/2$ zeros on the front and back and removes the padding. By default, `PadInput` is set to `true`.

Version History

Introduced in R2019a

References

[1] Princen, J., A. Johnson, and A. Bradley. "Subband/Transform Coding Using Filter Bank Designs Based on Time Domain Aliasing Cancellation." *IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*. 1987, pp. 2161–2164.

[2] Princen, J., and A. Bradley. "Analysis/Synthesis Filter Bank Design Based on Time Domain Aliasing Cancellation." *IEEE Transactions on Acoustics, Speech, and Signal Processing*. Vol. 34, Issue 5, 1986, pp. 1153–1161.

Extended Capabilities

C/C++ Code Generation

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

See Also

`mdct` | `kbdwin` | `spectrogram`

Topics

“Vorbis Decoder”

harmonicRatio

Harmonic ratio

Syntax

```
hr = harmonicRatio(audioIn,fs)
hr = harmonicRatio(audioIn,fs,Name=Value)
harmonicRatio(___)
```

Description

`hr = harmonicRatio(audioIn,fs)` returns the harmonic ratio of the signal, `audioIn`, over time. Columns of the input are treated as individual channels.

`hr = harmonicRatio(audioIn,fs,Name=Value)` specifies options using one or more name-value arguments.

Example: `hr = harmonicRatio(audioIn,fs,Window=rectwin(round(fs*0.1)),OverlapLength=round(fs*0.05))` returns the harmonic ratio for the audio input signal sampled at `fs` Hz. The harmonic ratio is calculated for 100 ms rectangular windows with 50 ms overlap.

`harmonicRatio(___)` with no output arguments plots the harmonic ratio against time. You can specify an input combination from any of the previous syntaxes.

Examples

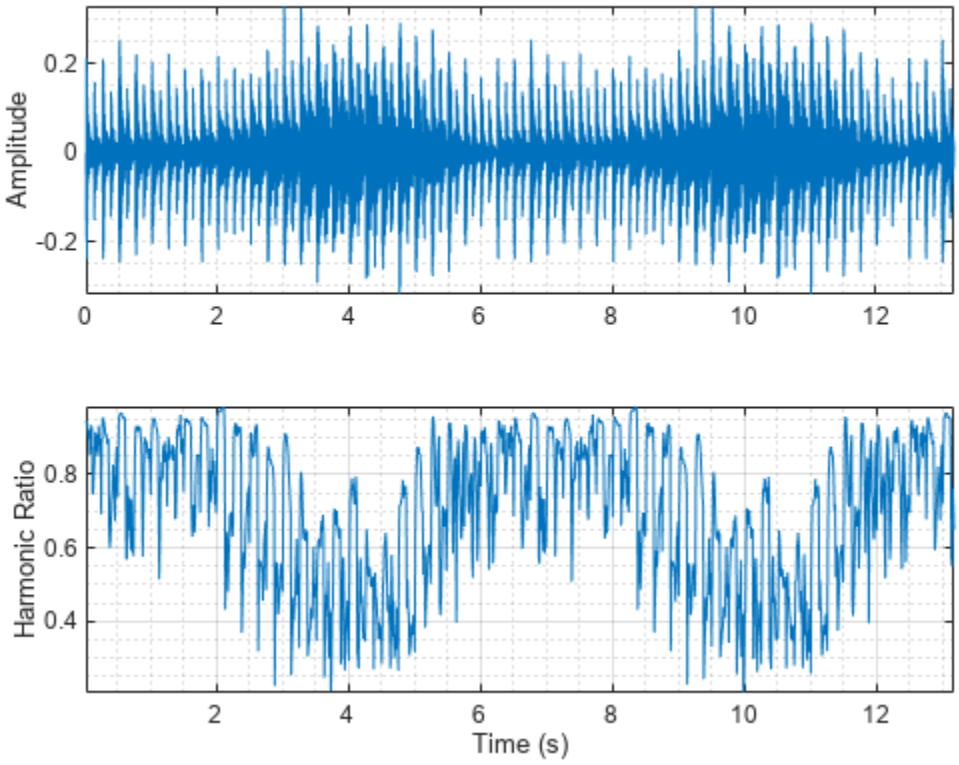
Calculate Harmonic Ratio

Read in an audio file and calculate the harmonic ratio using default parameters.

```
[audioIn,fs] = audioread("RandomOscThree-24-96-stereo-13secs.aif");
audioInMono = mean(audioIn,2);

hr = harmonicRatio(audioInMono,fs);
```

Plot the amplitude and the harmonic ratio of the signal against time.

```
t = (0:length(audioInMono)-1)/fs;
subplot(2,1,1)
plot(t,audioInMono)
ylabel("Amplitude")
grid minor
axis tight

subplot(2,1,2)
harmonicRatio(audioInMono,fs)
```



Specify Nondefault Parameters

Read in an audio file.

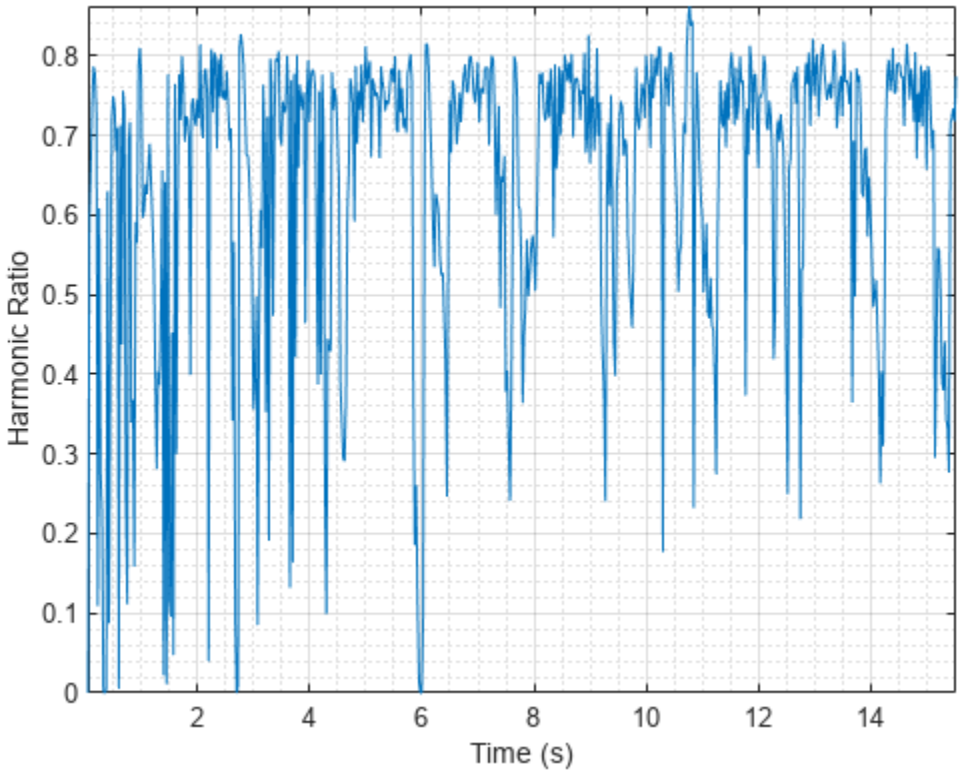
```
[audioIn,fs] = audioread("Counting-16-44p1-mono-15secs.wav");
```

Calculate the harmonic ratio of the audio file using 50 ms Hann windows with 25 ms overlap.

```
hr = harmonicRatio(audioIn,fs, ...
                   Window=hann(round(fs.*0.05),"periodic"), ...
                   OverlapLength=round(fs.*0.025));
```

Plot the harmonic ratio.

```
harmonicRatio(audioIn,fs, ...
              Window=hann(round(fs.*0.05),"periodic"), ...
              OverlapLength=round(fs.*0.025))
```

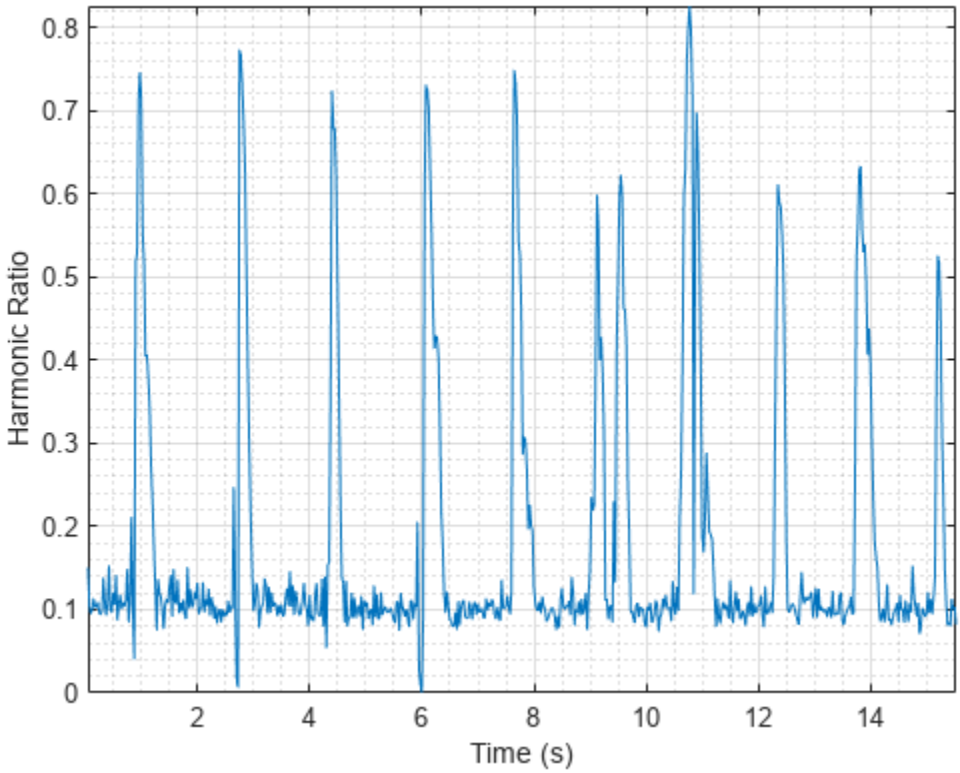


The harmonic ratio indicates the ratio of energy in the harmonic portion of audio to the total energy of the audio. Because the audio signal in this example has regions of near silence, where the total energy is very low, the harmonic ratio does a poor job discriminating between regions of speech and regions of silence. Add white noise to the audio signal and then calculate the harmonic ratio.

```
audioIn = audioIn + 0.1*randn(size(audioIn));
hr = harmonicRatio(audioIn,fs, ...
                   Window=hann(round(fs.*0.05),"periodic"), ...
                   OverlapLength=round(fs.*0.025));
```

Plot the new harmonic ratio of the audio signal combined with white noise.

```
harmonicRatio(audioIn,fs, ...
              Window=hann(round(fs.*0.05),"periodic"), ...
              OverlapLength=round(fs.*0.025))
```



Calculate Harmonic Ratio of Streaming Audio

Create a `dsp.AudioFileReader` object to read in stereo audio data frame-by-frame. Create a `dsp.SignalSink` object to log the harmonic ratio calculation.

```
fileReader = dsp.AudioFileReader('RandomOscThree-24-96-stereo-13secs.aif');
logger = dsp.SignalSink;
```

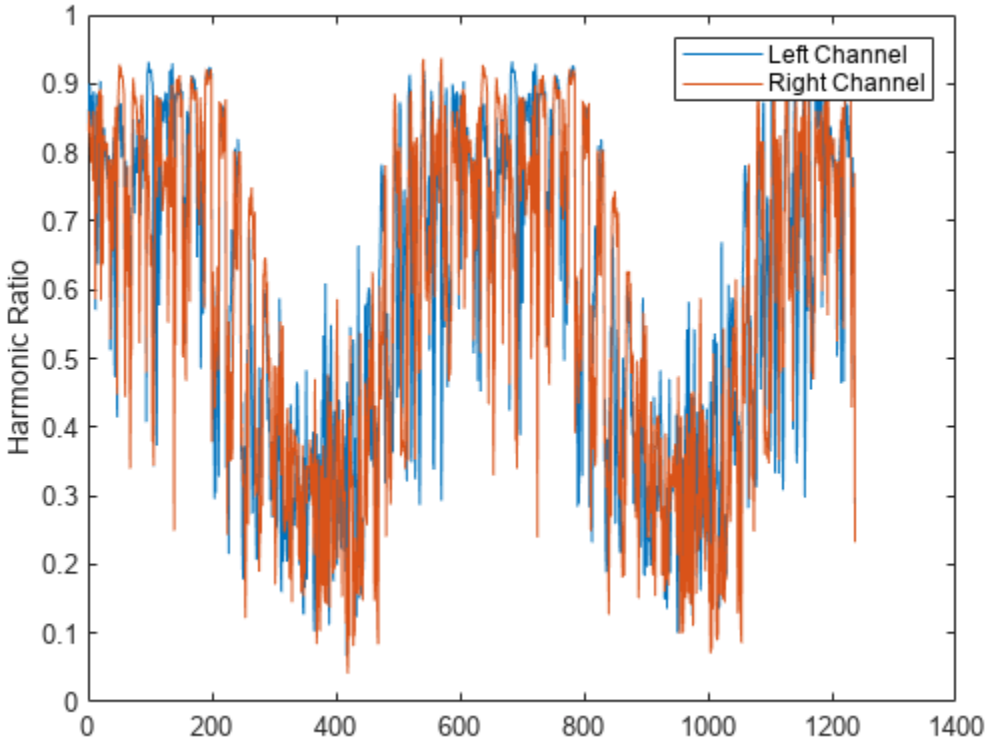
In an audio stream loop:

1. Read in a frame of audio data.
2. Calculate the harmonic ratio for each channel of the frame of audio.
3. Log the harmonic ratio for later plotting.

To calculate the harmonic ratio for only a given input frame, specify a window with the same number of samples as the input, and set the overlap length to zero. Plot the logged data.

```
win = hamming(fileReader.SamplesPerFrame,'periodic');
while ~isDone(fileReader)
    audioIn = fileReader();

    hr = harmonicRatio(audioIn,fileReader.SampleRate, ...
                       'Window',win, ...
                       'OverlapLength',0);
```

```
    logger(hr)
end

plot(logger.Buffer)
ylabel('Harmonic Ratio')
legend('Left Channel','Right Channel')
```



If the input to your audio stream loop has a variable samples-per-frame, an inconsistent samples-per-frame with the analysis window size of `harmonicRatio`, or if you want to calculate the harmonic ratio of overlapped data, use `dsp.AsyncBuffer`.

Create a `dsp.AsyncBuffer` object, reset the logger, and release the file reader.

```
buff = dsp.AsyncBuffer;
reset(logger)
release(fileReader)
```

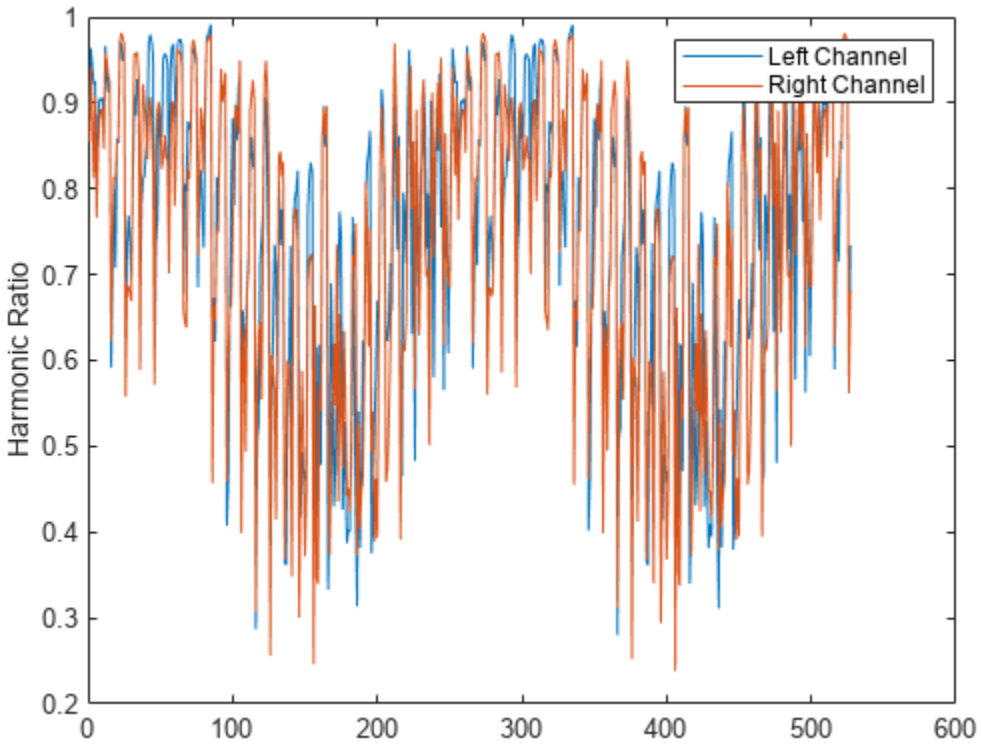
Calculate the harmonic ratio using 50 ms frames with a 25 ms overlap.

```
fs = fileReader.SampleRate;

samplesPerFrame = round(fs*0.05);
samplesOverlap = round(fs*0.025);

samplesPerHop = samplesPerFrame - samplesOverlap;

win = hamming(samplesPerFrame);
```

```
while ~isDone(fileReader)
    audioIn = fileReader();
    write(buff,audioIn);

    while buff.NumUnreadSamples >= samplesPerHop
        audioBuffered = read(buff,samplesPerFrame,samplesOverlap);

        hr = harmonicRatio(audioBuffered,fs, ...
                           'Window',win, ...
                           'OverlapLength',0);
        logger(hr)
    end

end
release(fileReader)
```

Plot the logged data.

```
plot(logger.Buffer)
ylabel('Harmonic Ratio')
legend('Left Channel','Right Channel')
```



Harmonic Ratio of Tones and White Noise

The harmonic ratio measures the amount of energy in the tonal part of the signal compared to the amount of energy in the total signal.

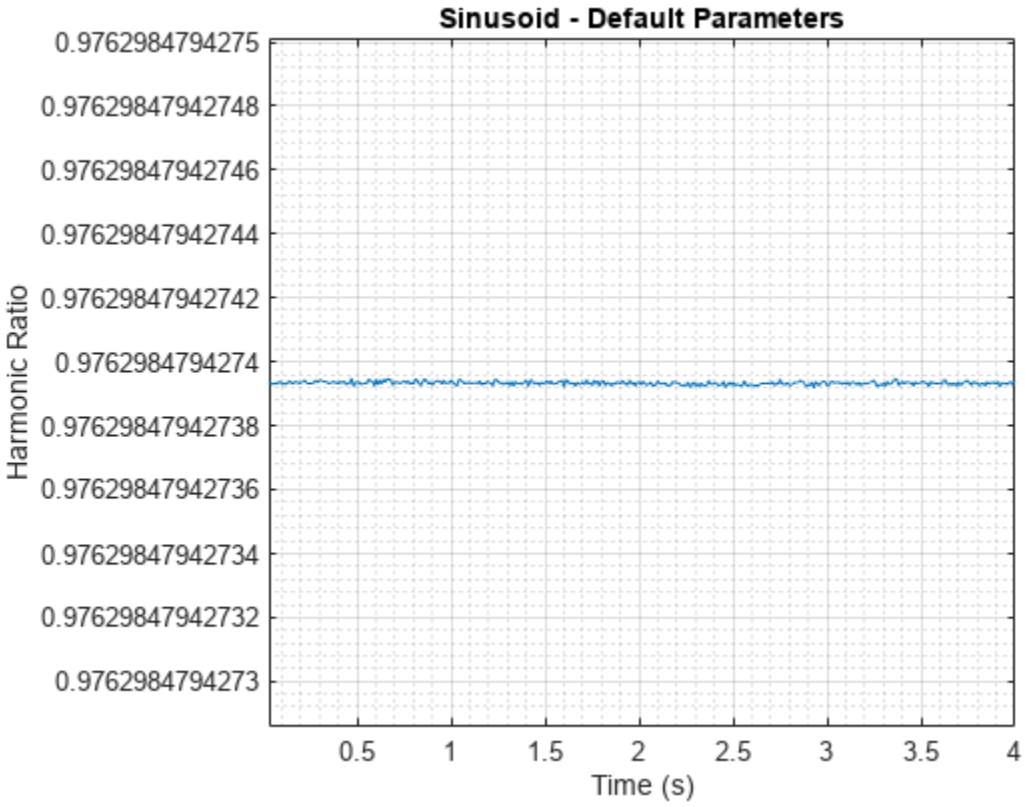
Harmonic Ratio of Pure Tone

Create a pure tone and then calculate the harmonic ratio using default parameters. By default, the harmonic ratio is calculated for 30 ms Hamming windows with 10 ms hops. Plot the results. The harmonic ratio is near 1, which is the theoretical maximum.

```
fs = 48e3;
osc = audioOscillator(Frequency=500,SamplesPerFrame=192e3,SampleRate=fs);
sinewave = osc();

hr = harmonicRatio(sinewave,fs);
```

Plot the harmonic ratio against time. The harmonic ratio is near 1, which is the theoretical maximum.

```
harmonicRatio(sinewave,fs)
title("Sinusoid - Default Parameters")
```



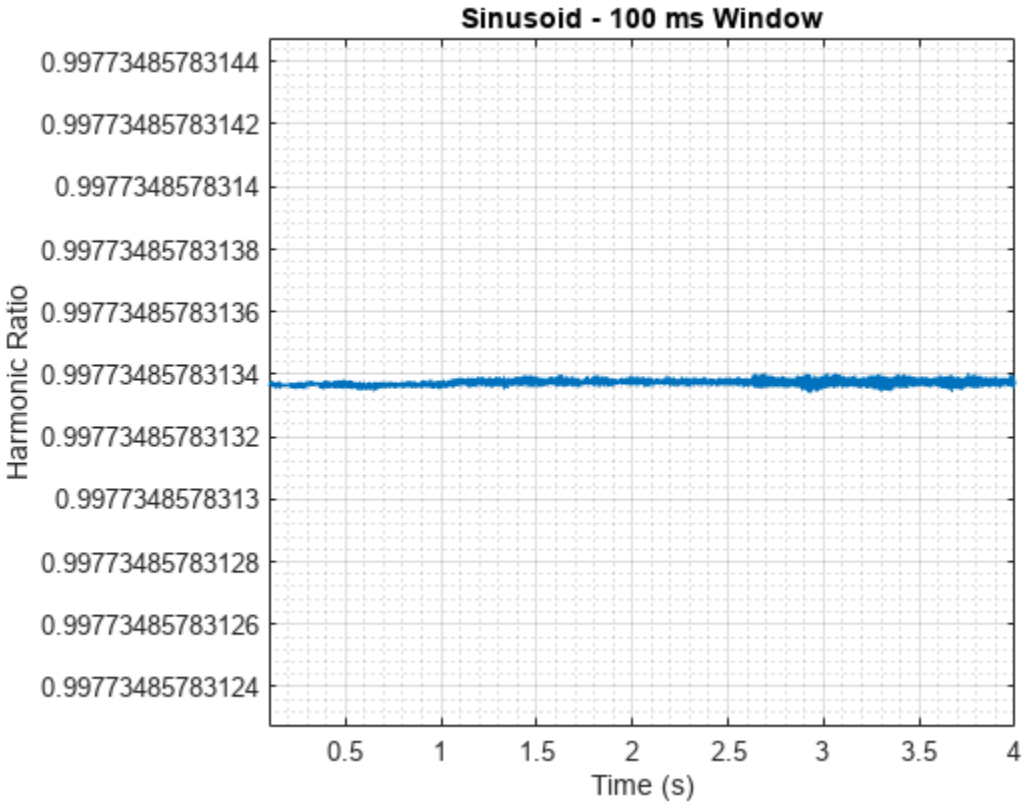
The short-time analysis required for windowing lowers the harmonic ratio from the theoretical value of 1. To diminish the effect of windowing, you can increase the window size. Use a 100 ms Hamming window and a 10 ms hop.

```
win = hamming(round(fs*0.1),"periodic");
overlap = round(fs*0.099);
```

```
hr = harmonicRatio(sinewave,fs,Window=win,OverlapLength=overlap);
```

Plot the harmonic ratio and observe that it is closer to one than when using the default window length.

```
harmonicRatio(sinewave,fs,Window=win,OverlapLength=overlap)
title("Sinusoid - 100 ms Window")
```



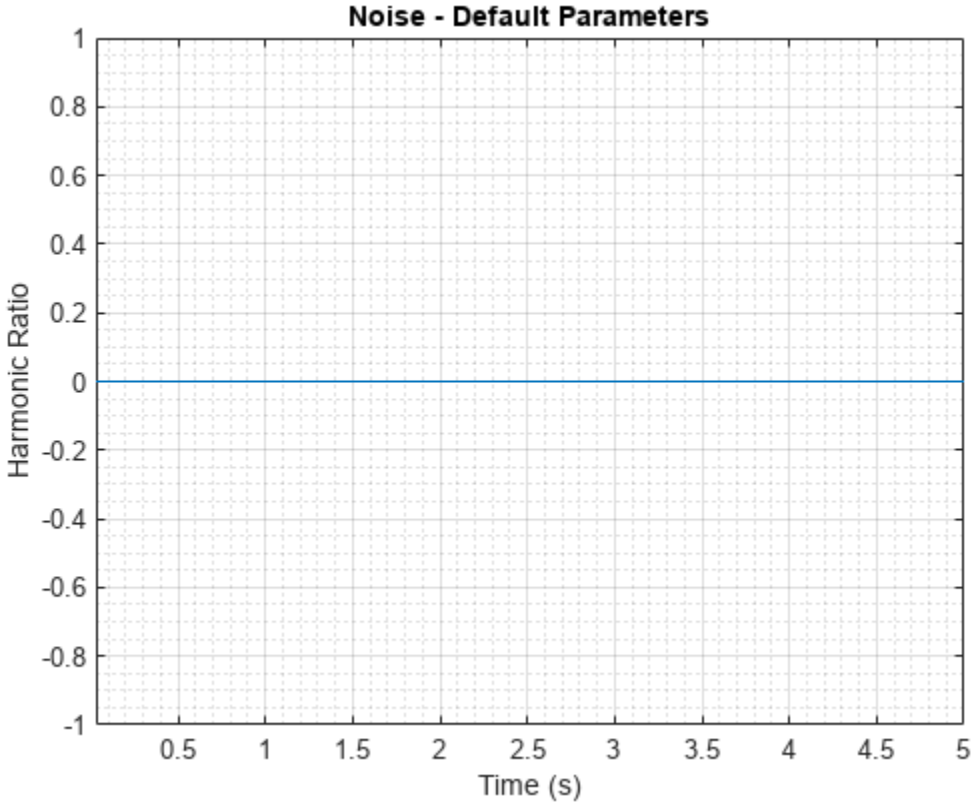
Harmonic Ratio of White Noise

Create 5 seconds of white noise and then calculate the harmonic ratio using default parameters. By default, the harmonic ratio is calculated for 30 ms Hamming windows with 10 ms hops.

```
fs = 48e3;
noise = rand(fs*5,1);

hr = harmonicRatio(noise,fs);
```

Plot the results. The harmonic ratio is 0.

```
harmonicRatio(noise,fs)
title("Noise - Default Parameters")
```



Input Arguments

`audioIn` — Input signal

column vector | matrix

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

Data Types: `single` | `double`

`fs` — Sample rate (Hz)

positive scalar

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

Data Types: `single` | `double`

Name-Value Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `Window=hamming(256)`

`Window` — Window applied in time domain
`hamming(round(fs*0.03),"periodic")` (default) | vector

Window applied in the time domain, specified as a real vector. The number of elements in the vector must be in the range [1, `size(audioIn,1)`]. The number of elements in the vector must also be greater than `OverlapLength`.

Data Types: `single` | `double`

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

Number of samples overlapped between adjacent windows, specified as an integer in the range [0, `size(Window,1)`).

Data Types: `single` | `double`

Output Arguments

`hr` — Harmonic ratio
scalar | vector | matrix

Harmonic ratio, returned as a scalar, vector, or matrix. Each row of `hr` corresponds to the harmonic ratio of a window of `audioIn`. The harmonic ratio is returned with values in the range [0,1]. A value of 0 represents low harmonicity, and a value of 1 represents high harmonicity.

Data Types: `single` | `double`

Algorithms

The harmonic ratio is calculated as described in [1]. The following algorithm is applied independently to each window of audio data. The normalized autocorrelation of the signal is determined as:

$$\Gamma(m) = \frac{\sum_{n=1}^{N} s(n)s(n-m)}{\sqrt{\sum_{n=1}^{N} s(n)^2 \sum_{n=0}^{N} s(n-m)^2}} \text{ for } (1 \le m \le M)$$

where

- s is a single frame of audio data with N elements.
- M is the maximum lag in the calculation. The maximum lag is 40 ms, which corresponds to a minimum fundamental frequency of 25 Hz.

A first estimate of the harmonic ratio is determined as the maximum of the normalized autocorrelation, within a given range:

$$\beta HR = \max_{M_0 \le m \le M} \{\Gamma(m)\}$$

where M_0 is the lower edge of the search range, determined as the first zero crossing of the normalized autocorrelation.

Finally, the harmonic ratio estimate is improved using parabolic interpolation, as described in [2].

Version History

Introduced in R2019a

References

[1] Kim, Hyoung-Gook, Nicholas Moreau, and Thomas Sikora. *MPEG-7 Audio and Beyond: Audio Content Indexing and Retrieval*. John Wiley & Sons, 2005.

[2] Quadratic Interpolation of Spectral Peaks. Accessed October 11, 2018. https://ccrma.stanford.edu/~jos/sasp/Quadratic_Interpolation_Spectral_Peaks.html

Extended Capabilities

C/C++ Code Generation

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

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

See Also

`pitch` | `spectralCentroid` | `voiceActivityDetector`

gtcc

Extract gammatone cepstral coefficients, log-energy, delta, and delta-delta

Syntax

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

Description

`coeffs = gtcc(audioIn,fs)` returns the gammatone cepstral coefficients (GTCCs) for the audio input, sampled at a frequency of `fs` Hz.

`coeffs = gtcc( ___ ,Name=Value)` specifies options using one or more name-value arguments.

`[coeffs,delta,deltaDelta,loc] = gtcc( ___ )` also returns the delta, delta-delta, and location in samples corresponding to each window of data. You can specify an input combination from any of the previous syntaxes.

`gtcc( ___ )` with no output arguments plots the gammatone cepstral coefficients. Before plotting, the coefficients are normalized to have mean 0 and standard deviation 1.

- If the input is in the time domain, the coefficients are plotted against time.
- If the input is in the frequency domain, the coefficients are plotted against frame number.
- If the log-energy is extracted, then it is also plotted.

Examples

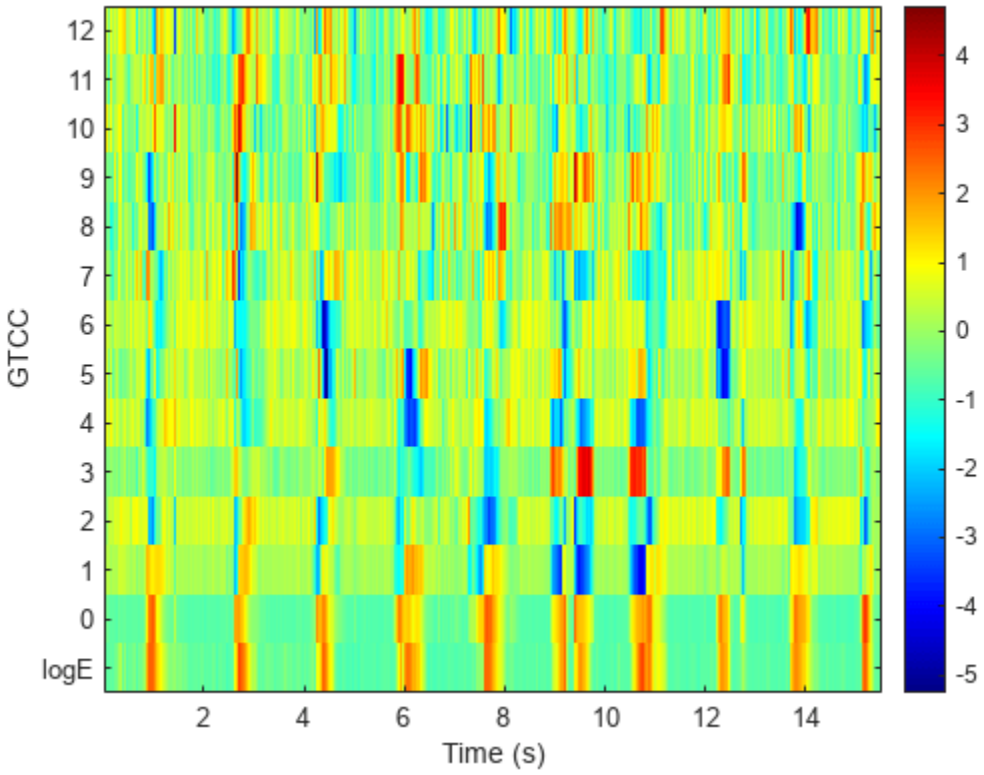
Extract GTCC from Audio Signal

Get the gammatone cepstral coefficients for an audio file using default settings.

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

[coeffs,~,~,loc] = gtcc(audioIn,fs);
```

Plot the normalized coefficients.

```
gtcc(audioIn,fs)
```



Specify Nondefault Parameters

Read in an audio file.

```
[audioIn,fs] = audioread("Turbine-16-44p1-mono-22secs.wav");
```

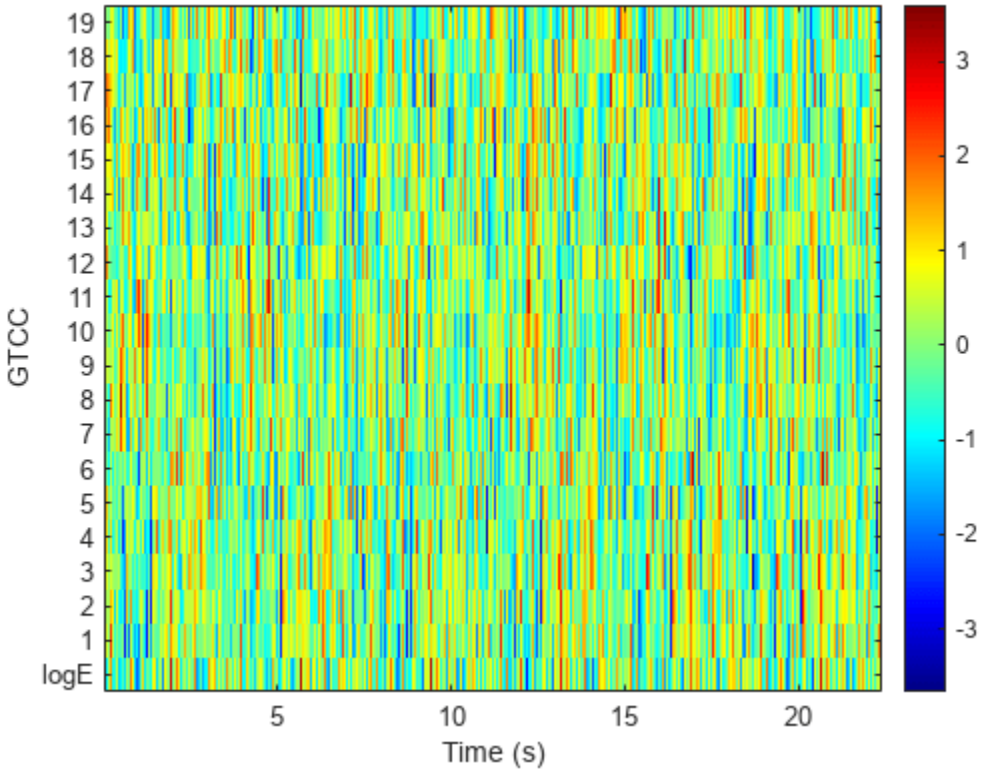
Calculate 20 GTCCs using filters equally spaced on the ERB scale between `hz2erb(62.5)` and `hz2erb(12000)`. Calculate the coefficients using 50 ms periodic Hann windows with 25 ms overlap. Replace the 0th coefficient with the log-energy. Use time-domain filtering.

```
[coeffs,~,~,loc] = gtcc(audioIn,fs, ...
                        NumCoeffs=20, ...
                        FrequencyRange=[62.5,12000], ...
                        Window=hann(round(0.05*fs),"periodic"), ...
                        OverlapLength=round(0.025*fs), ...
                        LogEnergy="replace", ...
                        FilterDomain="time");
```

Plot the normalized coefficients.

```
gtcc(audioIn,fs, ...
     NumCoeffs=20, ...
     FrequencyRange=[62.5,12000], ...
     Window=hann(round(0.05*fs),"periodic"), ...
     OverlapLength=round(0.025*fs), ...
```

```
LogEnergy="replace", ...
FilterDomain="time")
```



Extract GTCC from Frequency-Domain Audio

Read in an audio file and convert it to a frequency representation.

```
[audioIn,fs] = audioread("Rainbow-16-8-mono-114secs.wav");

win = hann(1024,"periodic");
S = stft(audioIn,"Window",win,"OverlapLength",512,"Centered",false);
```

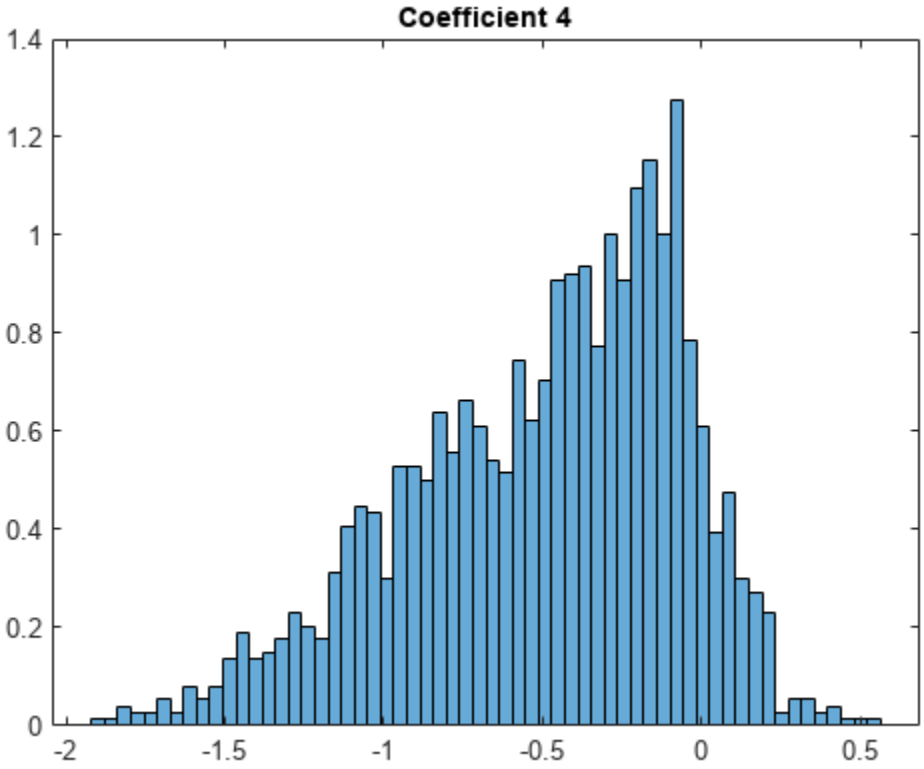
To extract the gammatone cepstral coefficients, call `gtcc` with the frequency-domain audio. Ignore the log-energy.

```
coeffs = gtcc(S,fs,"LogEnergy","Ignore");
```

In many applications, GTCC observations are converted to summary statistics for use in classification tasks. Plot a probability density function for one of the gammatone cepstral coefficients to observe its distributions.

```
nbins = 60;

coefficientToAnalyze = 4;
```

```
histogram(coeffs(:,coefficientToAnalyze+1),nbins,'Normalization','pdf')
title(sprintf("Coefficient %d",coefficientToAnalyze))
```



Input Arguments

`audioIn` — Input signal

vector | matrix | 3-D array

Input signal, specified as a vector, matrix, or 3-D array.

If `FilterDomain` is set to `"frequency"` (default), then `audioIn` can be real or complex.

- If `audioIn` is real, it is interpreted as a time-domain signal and must be a column vector or a matrix. Columns of the matrix are treated as independent audio channels.
- If `audioIn` is complex, it is interpreted as a frequency-domain signal. In this case, `audioIn` must be an *L*-by-*M*-by-*N* array, where *L* is the number of DFT points, *M* is the number of individual spectra, and *N* is the number of individual channels.

If `FilterDomain` is set to `"time"`, then `audioIn` must be a real column vector or matrix. Columns of the matrix are treated as independent audio channels.

Data Types: `single` | `double`
Complex Number Support: Yes

`fs` — Sample rate (Hz)
positive scalar

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

Data Types: `single` | `double`

Name-Value Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `coeffs = gtcc(audioIn,fs,LogEnergy="replace")` returns gammatone cepstral coefficients for the audio input signal sampled at `fs` Hz. For each analysis window, the first coefficient in the `coeffs` vector is replaced with the log energy of the input signal.

`Window` — Window applied in time domain
`hamming(round(0.03*fs),"periodic")` (default) | vector

Window applied in time domain, specified as a real vector. The number of elements in the vector must be in the range `[1,size(audioIn,1)]`. The number of elements in the vector must also be greater than `OverlapLength`.

Data Types: `single` | `double`

`OverlapLength` — Number of samples overlapped between adjacent windows
`round(0.02*fs)` (default) | non-negative scalar

Number of samples overlapped between adjacent windows, specified as an integer in the range [0, `numel(Window)`). If unspecified, `OverlapLength` defaults to `round(0.02*fs)`.

Data Types: `single` | `double`

`NumCoeffs` — Number of coefficients returned
`13` (default) | positive scalar integer

Number of coefficients returned for each window of data, specified as an integer in the range [2, v]. v is the number of valid passbands. If unspecified, `NumCoeffs` defaults to `13`.

The number of valid passbands is defined as the number of ERB steps (ERB_N) in the frequency range of the filter bank. The frequency range of the filter bank is specified by `FrequencyRange`.

Data Types: `single` | `double`

`FilterDomain` — Domain in which to apply filtering
`"frequency"` (default) | `"time"`

Domain in which to apply filtering, specified as `"frequency"` or `"time"`. If unspecified, `FilterDomain` defaults to `"frequency"`.

Data Types: `string` | `char`

`FrequencyRange` — Frequency range of gammatone filter bank (Hz)
`[50 fs/2]` (default) | two-element row vector

Frequency range of gammatone filter bank in Hz, specified as a two-element row vector of increasing values in the range [0, `fs`/2]. If unspecified, `FrequencyRange` defaults to `[50, fs/2]`

Data Types: `single` | `double`

`FFTLength` — Number of bins in DFT
`numel(Window)` (default) | positive scalar integer

Number of bins used to calculate the discrete Fourier transform (DFT) of windowed input samples. The FFT length must be greater than or equal to the number of elements in the `Window`.

Data Types: `single` | `double`

`Rectification` — Type of nonlinear rectification
`'log'` (default) | `'cubic-root'`

Type of nonlinear rectification applied prior to the discrete cosine transform, specified as `'log'` or `'cubic-root'`.

Data Types: `char` | `string`

`DeltaWindowLength` — Number of coefficients used to calculate delta and delta-delta
9 (default) | odd integer greater than two

Number of coefficients used to calculate the delta and the delta-delta values, specified as an odd integer greater than two. If unspecified, `DeltaWindowLength` defaults to 9.

Deltas are computed using the `audioDelta` function.

Data Types: `single` | `double`

`LogEnergy` — Log energy usage
`"append"` (default) | `"replace"` | `"ignore"`

Log energy usage, specified as `"append"`, `"replace"`, or `"ignore"`. If unspecified, `LogEnergy` defaults to `"append"`.

- `"append"` –– The function prepends the log energy to the coefficients vector. The length of the coefficients vector is 1 + `NumCoeffs`.
- `"replace"` –– The function replaces the first coefficient with the log energy of the signal. The length of the coefficients vector is `NumCoeffs`.
- `"ignore"` –– The function does not calculate or return the log energy.

Data Types: `char` | `string`

Output Arguments

`coeffs` — Gammatone cepstral coefficients
matrix | array

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

- *L* –– Number of analysis windows the audio signal is partitioned into. The input size, `Window`, and `OverlapLength` control this dimension: `L = floor((size(audioIn,1) - numel(Window)))/(numel(Window) - OverlapLength) + 1`.

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

  When `LogEnergy` is set to:

  - `"append"` –– The function prepends the log energy value to the coefficients vector. The length of the coefficients vector is 1 + `NumCoeffs`.
  - `"replace"` –– The function replaces the first coefficient with the log energy of the signal. The length of the coefficients vector is `NumCoeffs`.
  - `"ignore"` –– The function does not calculate or return the log energy. The length of the coefficients vector is `NumCoeffs`.
- *N* –– Number of input channels (columns). This value is `size(audioIn,2)`.

Data Types: `single` | `double`

`delta` — Change in coefficients
matrix | array

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

Data Types: `single` | `double`

`deltaDelta` — Change in delta values
matrix | array

Change in `delta` values, returned as an *L*-by-*M* matrix or an *L*-by-*M*-by-*N* array. The `deltaDelta` array is the same size and data type as the `coeffs` and `delta` arrays. See `coeffs` for the definitions of *L*, *M*, and *N*.

Data Types: `single` | `double`

`loc` — Location of the last sample in each analysis window
column vector

Location of last sample in each analysis window, returned as a column vector with the same number of rows as `coeffs`.

Data Types: `single` | `double`

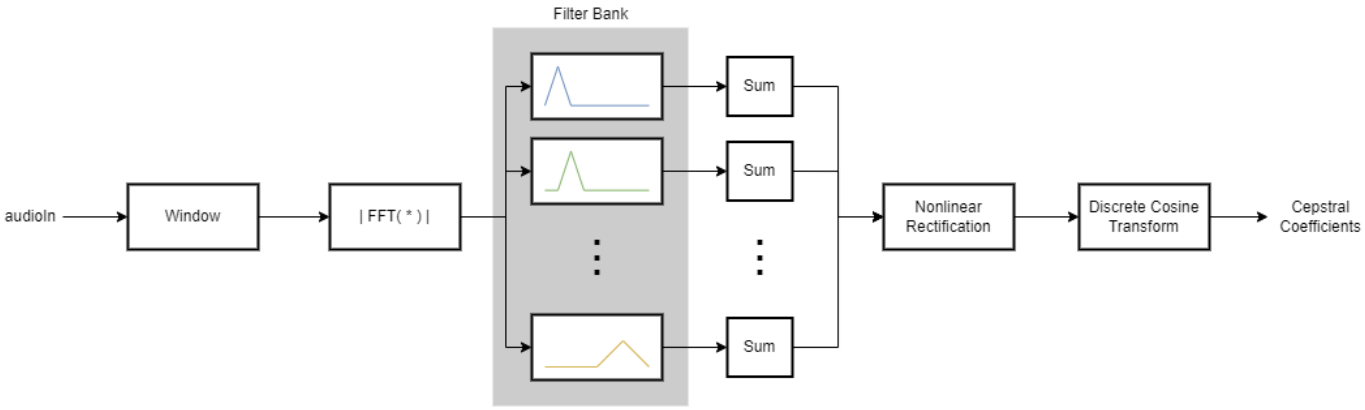
Algorithms

The `gtcc` function splits the entire data into overlapping segments. The length of each analysis window is determined by `Window`. The length of overlap between analysis windows is determined by `OverlapLength`. The algorithm to determine the gammatone cepstral coefficients depends on the filter domain, specified by `FilterDomain`. The default filter domain is frequency.

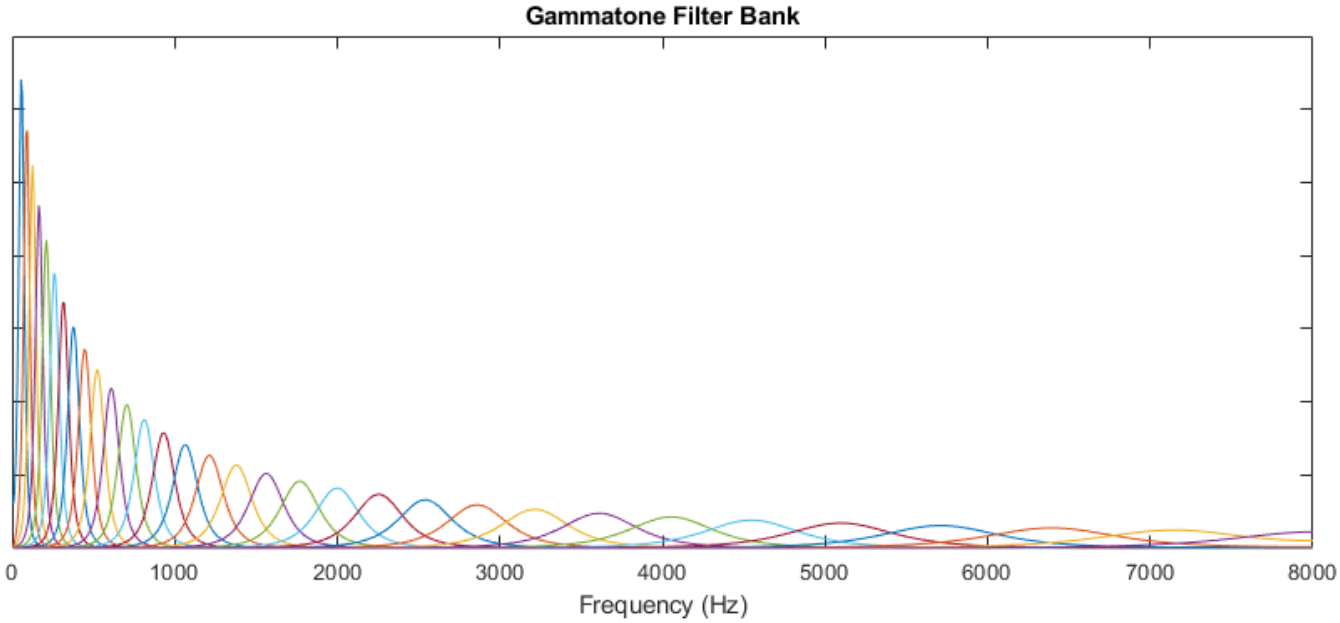
Frequency-Domain Filtering

Gammatone cepstrum coefficients are popular features extracted from speech signals for use in recognition tasks. In the source-filter model of speech, cepstral coefficients are understood to represent the filter (vocal tract). The vocal tract frequency response is relatively smooth, whereas the source of voiced speech can be modeled as an impulse train. As a result, the vocal tract can be estimated by the spectral envelope of a speech segment.

The motivating idea of gammatone cepstral coefficients is to compress information about the vocal tract (smoothed spectrum) into a small number of coefficients based on an understanding of the cochlea. Although there is no hard standard for calculating the coefficients, the basic steps are outlined by the diagram.



The default gammatone filter bank is composed of gammatone filters spaced linearly on the ERB scale between 50 and 8000 Hz. The filter bank is designed by `designAuditoryFilterBank`.



The information contained in the zeroth gammatone cepstral coefficient is often augmented with or replaced by the log energy. The log energy calculation depends on the input domain.

If the input is a time-domain signal, the log energy is computed using the following equation:

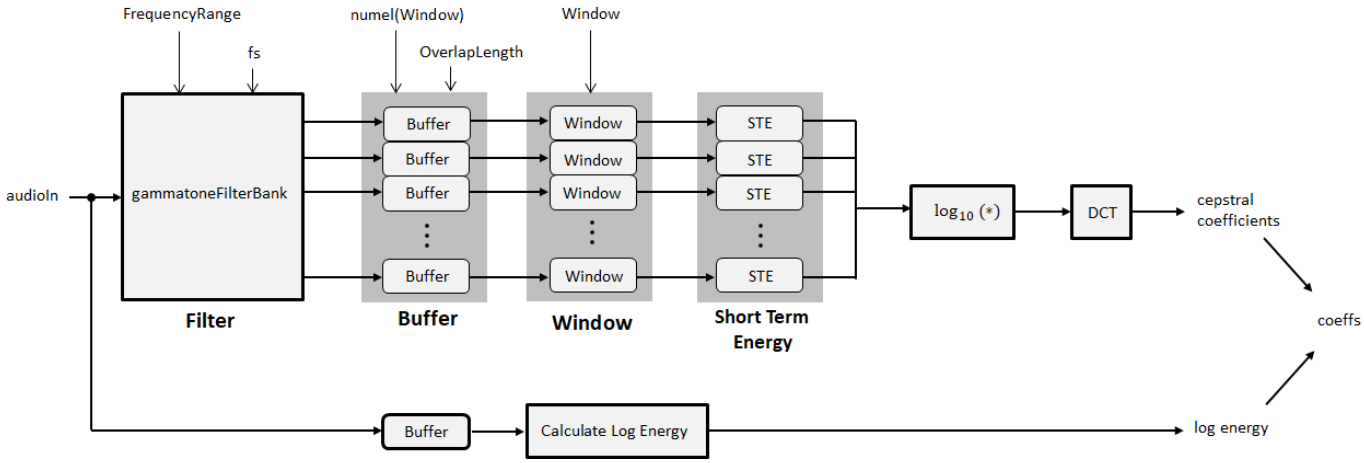
$$\log E = \log(\text{sum}(x^2))$$

If the input is a frequency-domain signal, the log energy is computed using the following equation:

$$\log E = \log\left(\text{sum}\left(|x|^2\right)/FFTLength\right)$$

Time-Domain Filtering

If `FilterDomain` is specified as `"time"`, the `gtcc` function uses the `gammatoneFilterBank` to apply time-domain filtering. The basic steps of the `gtcc` algorithm are outlined by the diagram.



The `FrequencyRange` and sample rate (`fs`) parameters are set on the filter bank using the name-value pairs input to the `gtcc` function. The number of filters in the gammatone filter bank is defined as `hz2erb(FrequencyRange(2)) - hz2erb(FrequencyRange(1))`.This roughly corresponds to placing a gammatone filter every 0.9 mm in the cochlea.

The output from the gammatone filter bank is a multichannel signal. Each channel output from the gammatone filter bank is buffered into overlapped analysis windows, as specified by the `Window` and `OverlapLength` parameters. The energy for each analysis window of data is calculated. The STE of the channels are concatenated. The concatenated signal is then passed through a logarithm function and transformed to the cepstral domain using a discrete cosine transform (DCT).

The log-energy is calculated on the original audio signal using the same buffering scheme applied to the gammatone filter bank output.

Version History

Introduced in R2019a

R2020b: Delta and delta-delta computation

Behavior changed in R2020b

The delta and delta-delta calculations are now computed using the `audioDelta` function, which has a different startup behavior than the previous algorithm. The default value of the `DeltaWindowLength` parameter has changed from 2 to 9. A delta window length of 2 is no longer supported.

R2020b: `WindowLength` will be removed in a future release

Behavior change in future release

The `WindowLength` parameter will be removed from the `gtcc` function in a future release. Use the `Window` parameter instead.

In releases prior to R2020b, you could only specify the length of a time-domain window. The window was always designed as a periodic Hamming window. You can replace instances of the code

```
coeffs = gtcc(audioin,fs,WindowLength=1024);
```

With this code:

```
coeffs = gtcc(audioIn,fs,Window=hamming(1024,"periodic"));
```

References

[1] Shao, Yang, Zhaozhang Jin, Deliang Wang, and Soundararajan Srinivasan. "An Auditory-Based Feature for Robust Speech Recognition." *IEEE International Conference on Acoustics, Speech and Signal Processing*. 2009.

[2] Valero, X., and F. Alias. "Gammatone Cepstral Coefficients: Biologically Inspired Features for Non-Speech Audio Classification." *IEEE Transactions on Multimedia*. Vol. 14, Issue 6, 2012, pp. 1684–1689.

Extended Capabilities

C/C++ Code Generation

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

See Also

`mfcc` | `audioDelta` | `cepstralCoefficients` | `audioFeatureExtractor` | `detectSpeech` | MFCC | Cepstral Coefficients

Topics

“Speech Emotion Recognition”

spectralSpread

Spectral spread for audio signals and auditory spectrograms

Syntax

```
spread = spectralSpread(x,f)
spread = spectralSpread(x,f,Name=Value)
[spread,centroid] = spectralSpread( ___ )
spectralSpread( ___ )
```

Description

`spread = spectralSpread(x,f)` returns the spectral spread of the signal, `x`, over time. How the function interprets `x` depends on the shape of `f`.

`spread = spectralSpread(x,f,Name=Value)` specifies options using one or more name-value arguments.

`[spread,centroid] = spectralSpread( ___ )` returns the spectral centroid. You can specify an input combination from any of the previous syntaxes.

`spectralSpread( ___ )` with no output arguments plots the spectral spread.

- If the input is in the time domain, the spectral spread is plotted against time.
- If the input is in the frequency domain, the spectral spread is plotted against frame number.

Examples

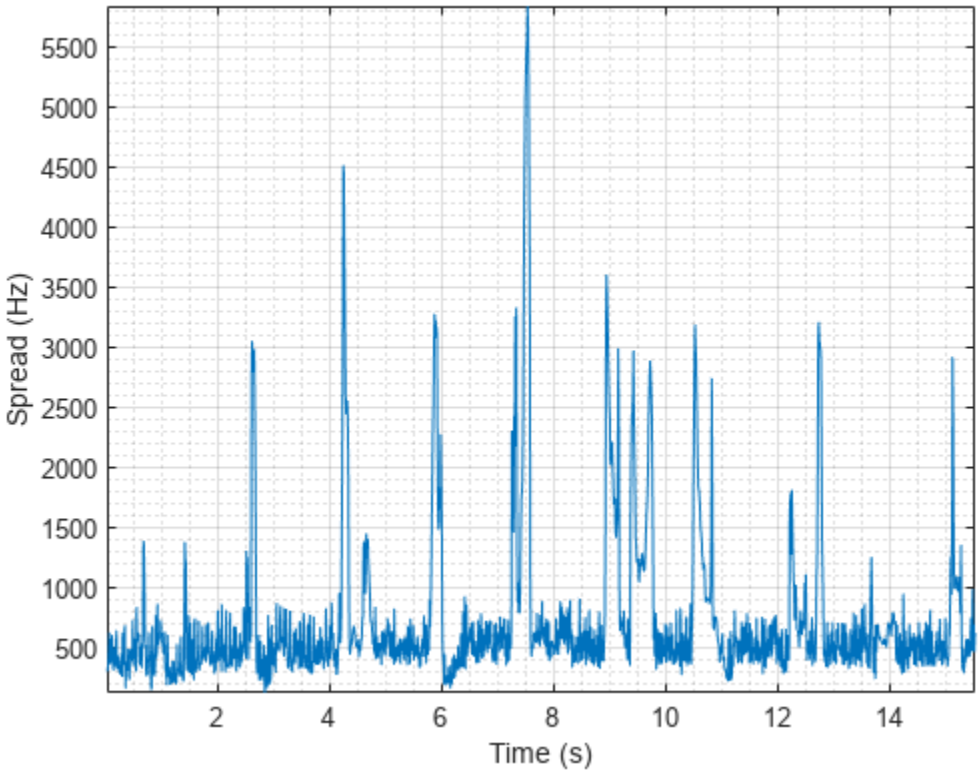
Spectral Spread of Time-Domain Audio

Read in an audio file, calculate the spread using default parameters.

```
[audioIn,fs] = audioread("Counting-16-44p1-mono-15secs.wav");
spread = spectralSpread(audioIn,fs);
```

Plot the spectral spread against time.

```
spectralSpread(audioIn,fs)
```



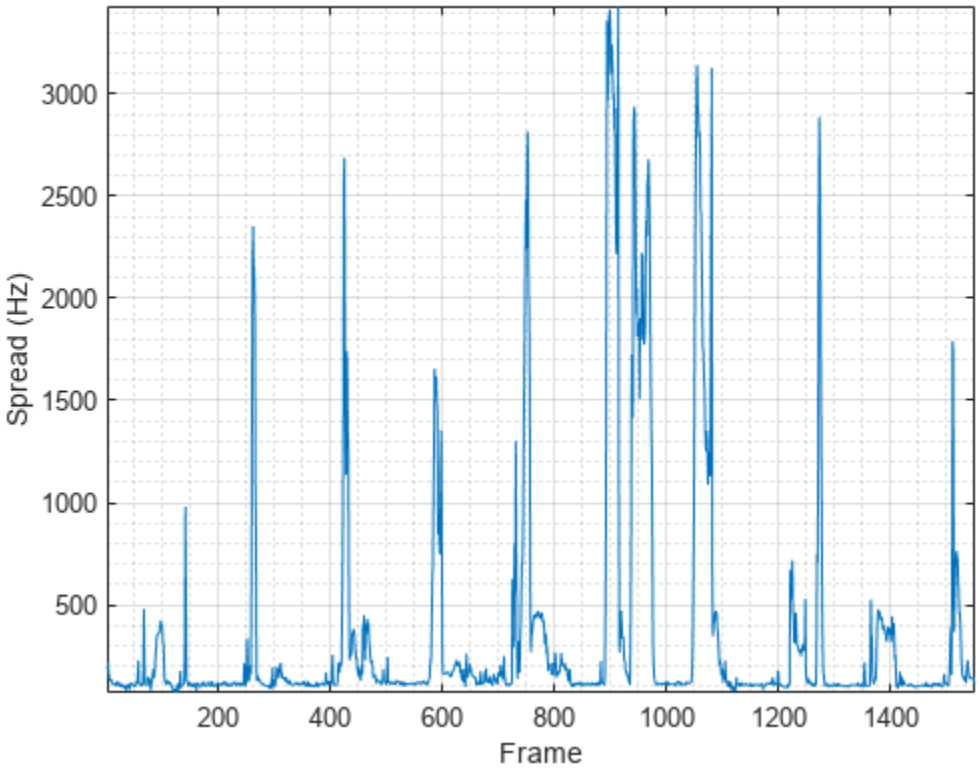
Spectral Spread of Frequency-Domain Audio Data

Read in an audio file and then calculate the mel spectrogram using the `melSpectrogram` function. Calculate the spread of the mel spectrums over time.

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

[s,cf,t] = melSpectrogram(audioIn,fs);

spread = spectralSpread(s,cf);
```

Plot spectral spread against the frame number.

```
spectralSpread(s,cf)
```



Specify Nondefault Parameters

Read in an audio file.

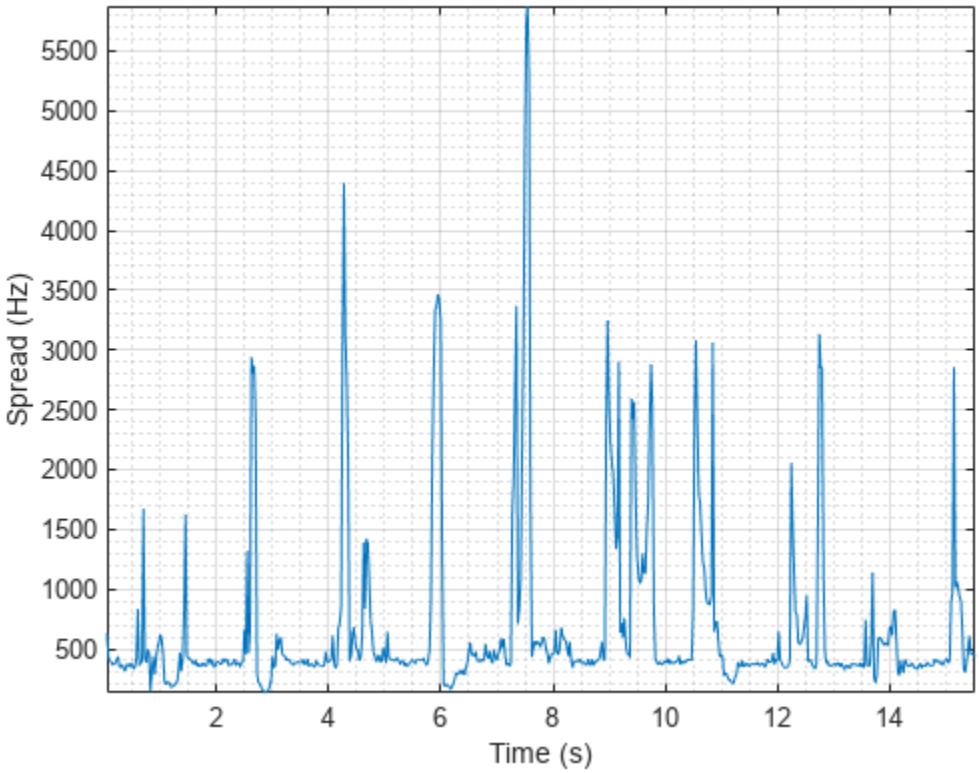
```
[audioIn,fs] = audioread("Counting-16-44p1-mono-15secs.wav");
```

Calculate the spread of the power spectrum over time. Calculate the spread for 50 ms Hamming windows of data with 25 ms overlap. Use the range from 62.5 Hz to `fs`/2 for the spread calculation.

```
spread = spectralSpread(audioIn,fs, ...
                        Window=hamming(round(0.05*fs)), ...
                        OverlapLength=round(0.025*fs), ...
                        Range=[62.5,fs/2]);
```

Plot the spectral spread.

```
spectralSpread(audioIn,fs, ...
               Window=hamming(round(0.05*fs)), ...
               OverlapLength=round(0.025*fs), ...
               Range=[62.5,fs/2]);
```



Calculate Spectral Spread of Streaming Audio

Create a `dsp.AudioFileReader` object to read in audio data frame-by-frame. Create a `dsp.SignalSink` to log the spectral spread calculation.

```
fileReader = dsp.AudioFileReader('Counting-16-44p1-mono-15secs.wav');
logger = dsp.SignalSink;
```

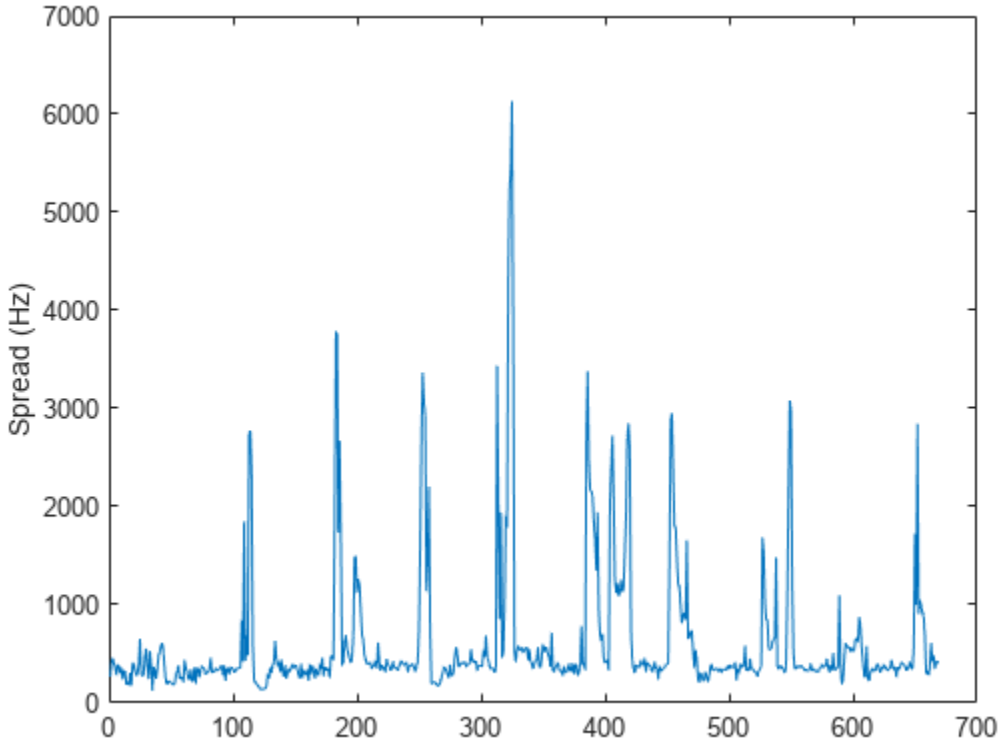
In an audio stream loop:

1. Read in a frame of audio data.
2. Calculate the spectral spread for the frame of audio.
3. Log the spectral spread for later plotting.

To calculate the spectral spread for only a given input frame, specify a window with the same number of samples as the input, and set the overlap length to zero. Plot the logged data.

```
win = hamming(fileReader.SamplesPerFrame);
while ~isDone(fileReader)
    audioIn = fileReader();
    spread = spectralSpread(audioIn,fileReader.SampleRate, ...
                            'Window',win, ...
                            'OverlapLength',0);
    logger(spread)
```

```
end

plot(logger.Buffer)
ylabel('Spread (Hz)')
```



Use `dsp.AsyncBuffer` if

- The input to your audio stream loop has a variable samples-per-frame.
- The input to your audio stream loop has an inconsistent samples-per-frame with the analysis window of `spectralSpread`.
- You want to calculate the spectral spread for overlapped data.

Create a `dsp.AsyncBuffer` object, reset the logger, and release the file reader.

```
buff = dsp.AsyncBuffer;
reset(logger)
release(fileReader)
```

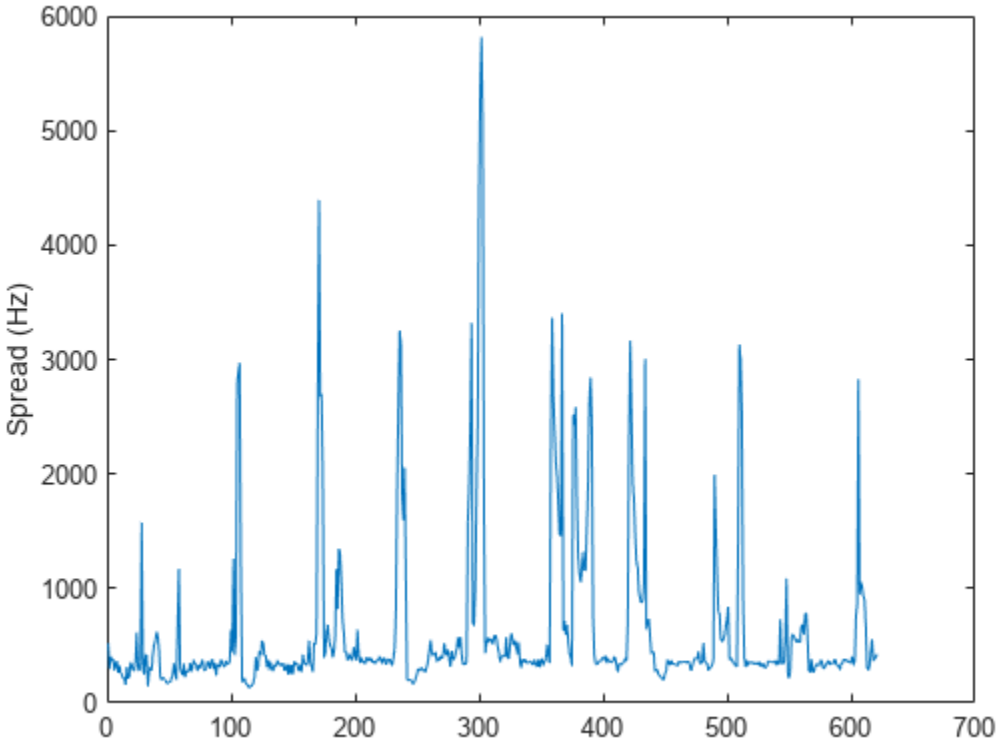
Specify that the spectral spread is calculated for 50 ms frames with a 25 ms overlap.

```
fs = fileReader.SampleRate;

samplesPerFrame = round(fs*0.05);
samplesOverlap = round(fs*0.025);

samplesPerHop = samplesPerFrame - samplesOverlap;
```

```
win = hamming(samplesPerFrame);

while ~isDone(fileReader)
    audioIn = fileReader();
    write(buff,audioIn);

    while buff.NumUnreadSamples >= samplesPerHop
        audioBuffered = read(buff,samplesPerFrame,samplesOverlap);

        spread = spectralSpread(audioBuffered,fs, ...
                                'Window',win, ...
                                'OverlapLength',0);
        logger(spread)
    end

end
release(fileReader)
```

Plot the logged data.

```
plot(logger.Buffer)
ylabel('Spread (Hz)')
```



Input Arguments

x — Input signal

column vector | matrix | 3-D array

Input signal, specified as a vector, matrix, or 3-D array. How the function interprets `x` depends on the shape of `f`.

Data Types: `single` | `double`

f — Sample rate or frequency vector (Hz)

scalar | vector

Sample rate or frequency vector in Hz, specified as a scalar or vector, respectively. How the function interprets `x` depends on the shape of `f`:

- If `f` is a scalar, `x` is interpreted as a time-domain signal, and `f` is interpreted as the sample rate. In this case, `x` must be a real vector or matrix. If `x` is specified as a matrix, the columns are interpreted as individual channels.
- If `f` is a vector, `x` is interpreted as a frequency-domain signal, and `f` is interpreted as the frequencies, in Hz, corresponding to the rows of `x`. In this case, `x` must be a real *L*-by-*M*-by-*N* array, where *L* is the number of spectral values at given frequencies of `f`, *M* is the number of individual spectra, and *N* is the number of channels.
- The number of rows of `x`, *L*, must be equal to the number of elements of `f`.

Data Types: `single` | `double`

Name-Value Arguments

Note The following name-value arguments apply if `x` is a time-domain signal. If `x` is a frequency-domain signal, name-value arguments are ignored.

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `Window=hamming(256)`

Window — Window applied in time domain

`rectwin(round(f*0.03))` (default) | vector

Window applied in the time domain, specified as a real vector. The number of elements in the vector must be in the range [1, `size(x,1)`]. The number of elements in the vector must also be greater than `OverlapLength`.

Data Types: `single` | `double`

OverlapLength — Number of samples overlapped between adjacent windows

`round(f*0.02)` (default) | non-negative scalar

Number of samples overlapped between adjacent windows, specified as an integer in the range [0, `size(Window,1)`).

Data Types: `single` | `double`

`FFTLength` — Number of bins in DFT
`numel(Window)` (default) | positive scalar integer

Number of bins used to calculate the DFT of windowed input samples, specified as a positive scalar integer. If unspecified, `FFTLength` defaults to the number of elements in the `Window`.

Data Types: `single` | `double`

`Range` — Frequency range (Hz)
`[0,f/2]` (default) | two-element row vector

Frequency range in Hz, specified as a two-element row vector of increasing real values in the range [0, `f`/2].

Data Types: `single` | `double`

`SpectrumType` — Spectrum type
`"power"` (default) | `"magnitude"`

Spectrum type, specified as `"power"` or `"magnitude"`:

- `"power"` –- The spectral spread is calculated for the one-sided power spectrum.
- `"magnitude"` –- The spectral spread is calculated for the one-sided magnitude spectrum.

Data Types: `char` | `string`

Output Arguments

`spread` — Spectral spread (Hz)
scalar | vector | matrix

Spectral spread in Hz, returned as a scalar, vector, or matrix. Each row of `centroid` corresponds to the spectral spread of a window of `x`. Each column of `spread` corresponds to an independent channel.

`centroid` — Spectral centroid (Hz)
scalar | vector | matrix

Spectral centroid in Hz, returned as a scalar, vector, or matrix. Each row of `centroid` corresponds to the spectral centroid of a window of `x`. Each column of `centroid` corresponds to an independent channel.

Algorithms

The spectral spread is calculated as described in [1]:

$$\text{spread} = \sqrt{\frac{\sum_{k=b_1}^{b_2} (f_k - \mu_1)^2 s_k}{\sum_{k=b_1}^{b_2} s_k}}$$

where

- f_k is the frequency in Hz corresponding to bin k.
- s_k is the spectral value at bin k.
- b_1 and b_2 are the band edges, in bins, over which to calculate the spectral spread.
- μ_1 is the spectral centroid, calculated as described by the `spectralCentroid` function.

Version History

Introduced in R2019a

References

[1] Peeters, G. "A Large Set of Audio Features for Sound Description (Similarity and Classification) in the CUIDADO Project." Technical Report; IRCAM: Paris, France, 2004.

Extended Capabilities

C/C++ Code Generation

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

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

See Also

`spectralCentroid` | `spectralSkewness` | `spectralKurtosis`

Topics

“Spectral Descriptors”

spectralSlope

Spectral slope for audio signals and auditory spectrograms

Syntax

```
slope = spectralSlope(x,f)
slope = spectralSlope(x,f,Name=Value)
spectralSlope(___)
```

Description

`slope = spectralSlope(x,f)` returns the spectral slope of the signal, `x`, over time. How the function interprets `x` depends on the shape of `f`.

`slope = spectralSlope(x,f,Name=Value)` specifies options using one or more name-value arguments.

`spectralSlope(___)` with no output arguments plots the spectral slope. You can specify an input combination from any of the previous syntaxes.

- If the input is in the time domain, the spectral slope is plotted against time.
- If the input is in the frequency domain, the spectral slope is plotted against frame number.

Examples

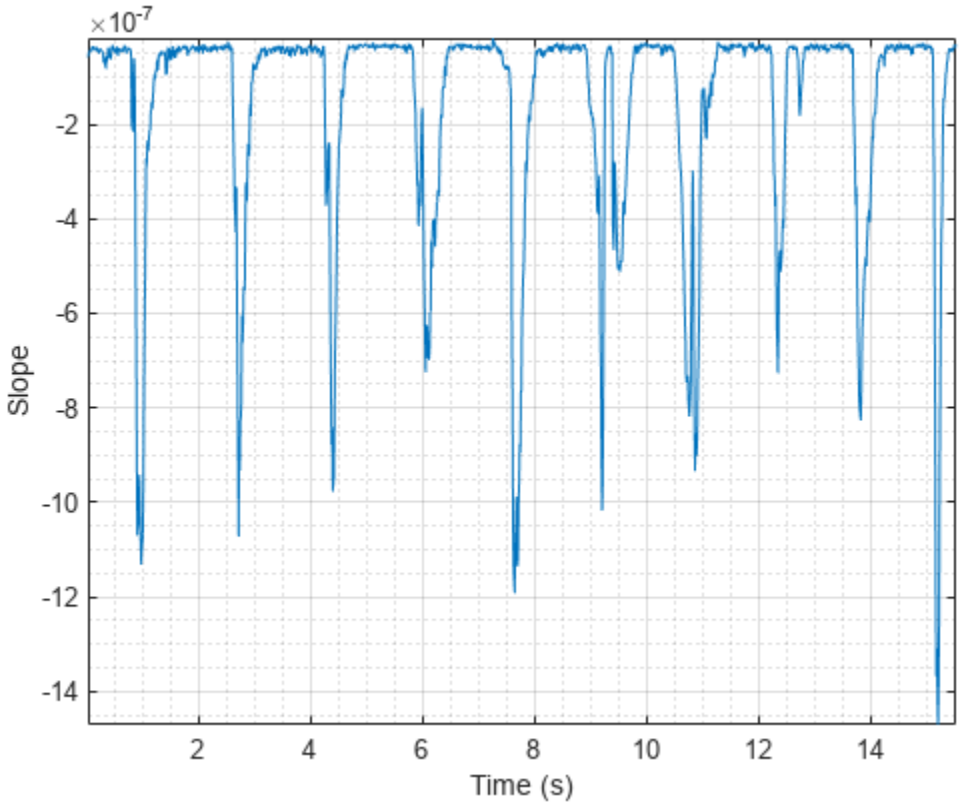
Spectral Slope of Time-Domain Audio

Read in an audio file, calculate the slope using default parameters.

```
[audioIn,fs] = audioread("Counting-16-44p1-mono-15secs.wav");
slope = spectralSlope(audioIn,fs);
```

Plot the spectral slope against time.

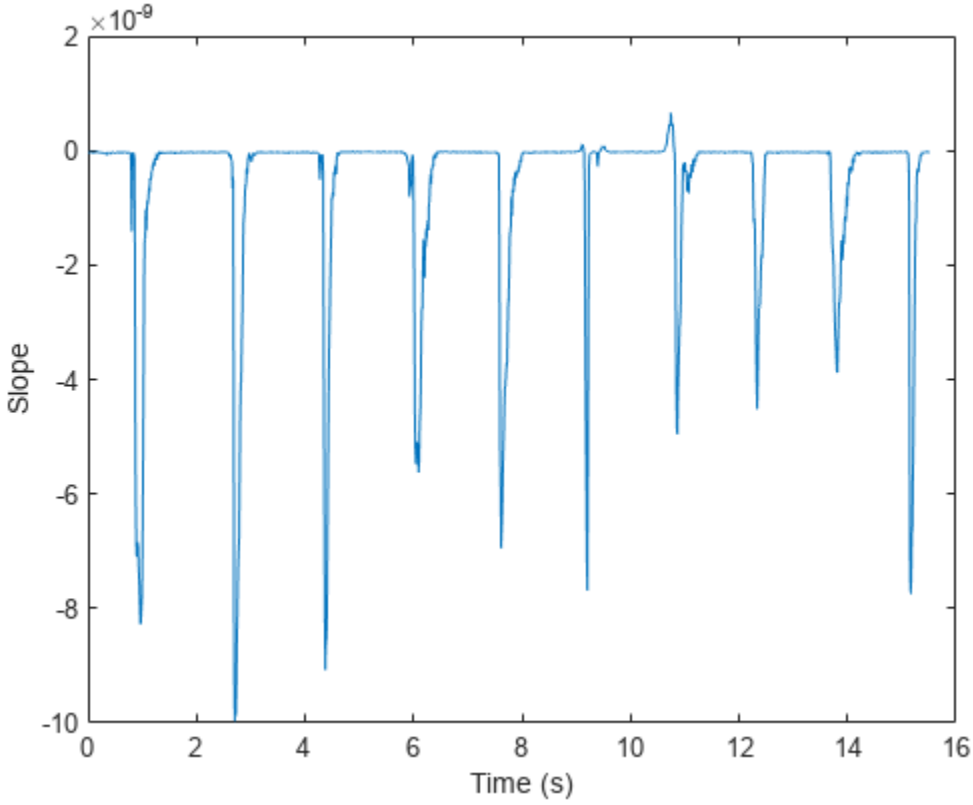
```
spectralSlope(audioIn,fs)
```



Spectral Slope of Frequency-Domain Audio Data

Read in an audio file and then calculate the mel spectrogram using the `melSpectrogram` function. Calculate the slope of the mel spectrogram over time. Plot the results.

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

[s,cf,t] = melSpectrogram(audioIn,fs);

slope = spectralSlope(s,cf);

plot(t,slope)
xlabel('Time (s)')
ylabel('Slope')
```



Specify Nondefault Parameters

Read in an audio file.

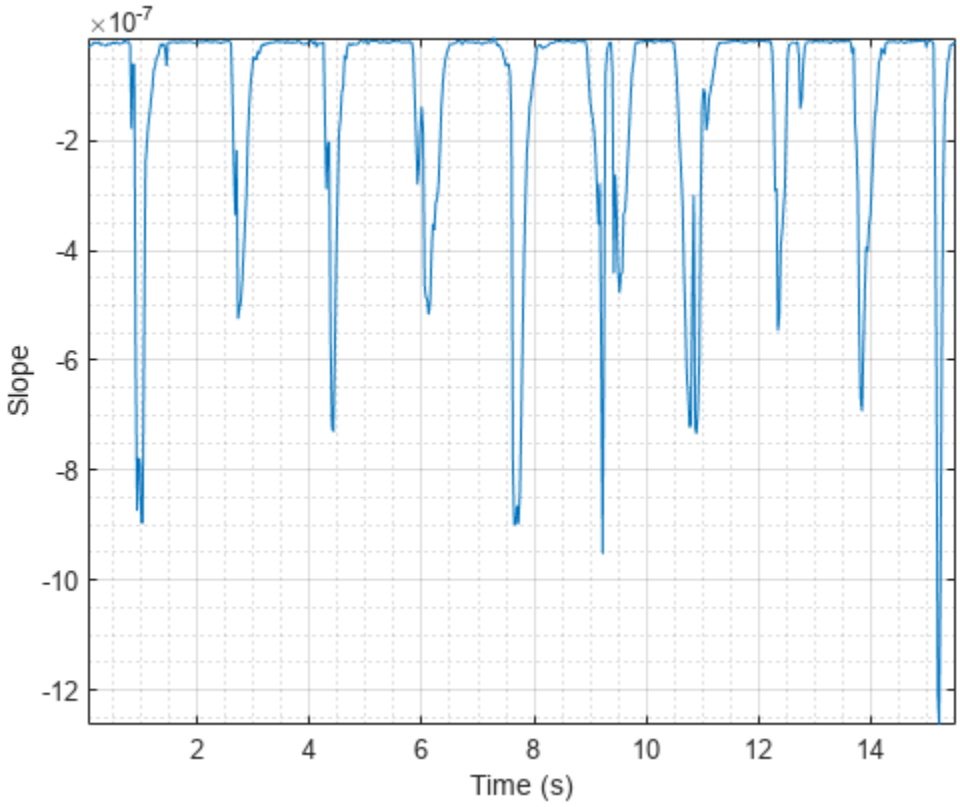
```
[audioIn,fs] = audioread("Counting-16-44p1-mono-15secs.wav");
```

Calculate the slope of the magnitude spectrum over time. Calculate the slope for 50 ms Hamming windows of data with 25 ms overlap. Use the range from 62.5 Hz to `fs`/2 for the slope calculation.

```
slope = spectralSlope(audioIn,fs, ...
                      Window=hamming(round(0.05*fs)), ...
                      OverlapLength=round(0.025*fs), ...
                      Range=[62.5,fs/2]);
```

Plot the spectral slope against time.

```
spectralSlope(audioIn,fs, ...
              Window=hamming(round(0.05*fs)), ...
              OverlapLength=round(0.025*fs), ...
              Range=[62.5,fs/2]);
```



Calculate Spectral Slope of Streaming Audio

Create a `dsp.AudioFileReader` object to read in audio data frame-by-frame. Create a `dsp.SignalSink` to log the spectral slope calculation.

```
fileReader = dsp.AudioFileReader('Counting-16-44p1-mono-15secs.wav');
logger = dsp.SignalSink;
```

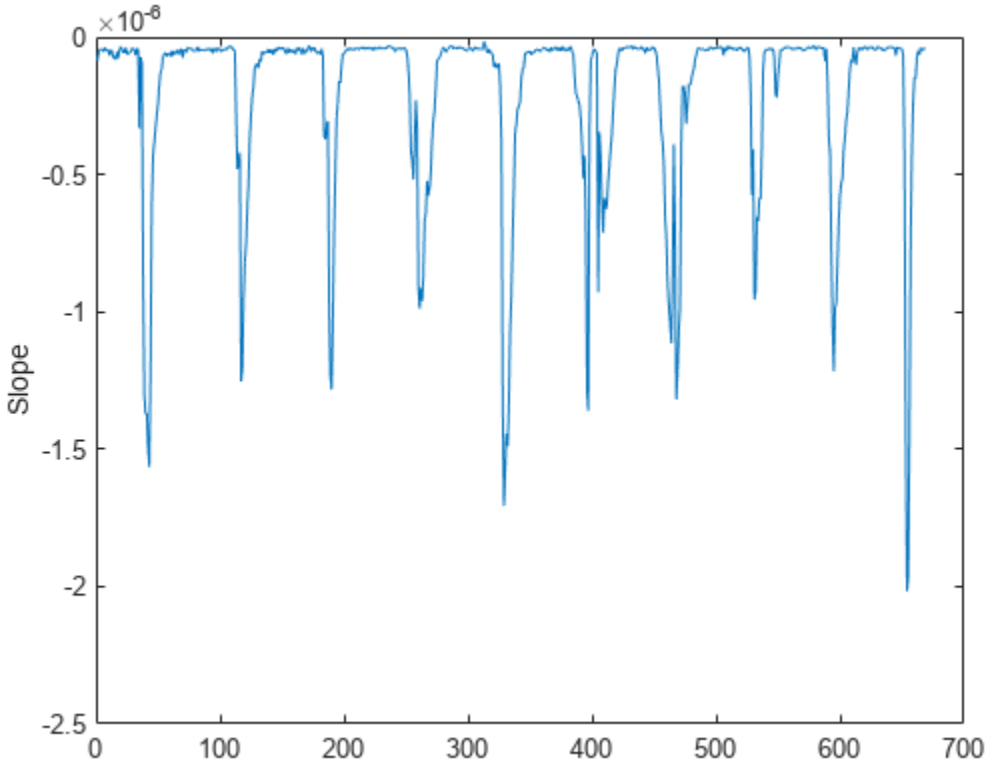
In an audio stream loop:

1. Read in a frame of audio data.
2. Calculate the spectral slope for the frame of audio.
3. Log the spectral slope for later plotting.

To calculate the spectral slope for only a given input frame, specify a window with the same number of samples as the input, and set the overlap length to zero. Plot the logged data.

```
win = hamming(fileReader.SamplesPerFrame);
while ~isDone(fileReader)
    audioIn = fileReader();
    slope = spectralSlope(audioIn,fileReader.SampleRate, ...
                          'Window',win, ...
                          'OverlapLength',0);
    logger(slope)
```

```
end

plot(logger.Buffer)
ylabel('Slope')
```



Use `dsp.AsyncBuffer` if

- The input to your audio stream loop has a variable samples-per-frame.
- The input to your audio stream loop has an inconsistent samples-per-frame with the analysis window of `spectralSlope`.
- You want to calculate the spectral slope for overlapped data.

Create a `dsp.AsyncBuffer` object, reset the logger, and release the file reader.

```
buff = dsp.AsyncBuffer;
reset(logger)
release(fileReader)
```

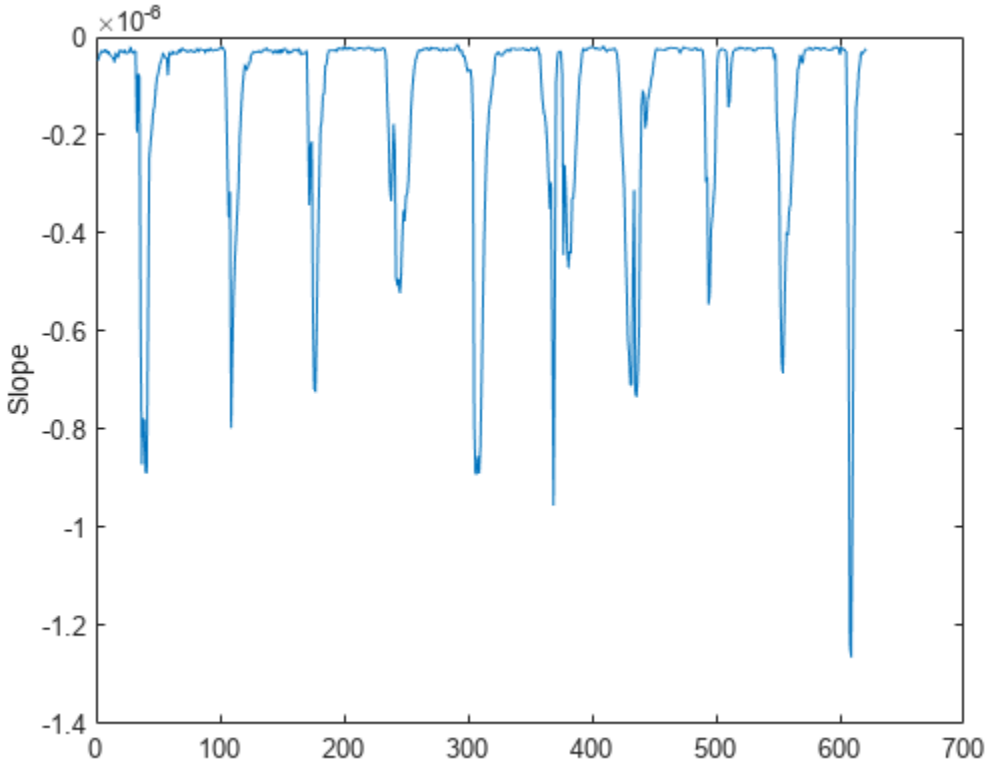
Specify that the spectral slope is calculated for 50 ms frames with a 25 ms overlap.

```
fs = fileReader.SampleRate;

samplesPerFrame = round(fs*0.05);
samplesOverlap = round(fs*0.025);

samplesPerHop = samplesPerFrame - samplesOverlap;
```

```
win = hamming(samplesPerFrame);

while ~isDone(fileReader)
    audioIn = fileReader();
    write(buff,audioIn);

    while buff.NumUnreadSamples >= samplesPerHop
        audioBuffered = read(buff,samplesPerFrame,samplesOverlap);

        slope = spectralSlope(audioBuffered,fs, ...
                              'Window',win, ...
                              'OverlapLength',0);
        logger(slope)
    end

end
release(fileReader)
```

Plot the logged data.

```
plot(logger.Buffer)
ylabel('Slope')
```



Input Arguments

`x` — Input signal

column vector | matrix | 3-D array

Input signal, specified as a vector, matrix, or 3-D array. How the function interprets `x` depends on the shape of `f`.

Data Types: `single` | `double`

`f` — Sample rate or frequency vector (Hz)

scalar | vector

Sample rate or frequency vector in Hz, specified as a scalar or vector, respectively. How the function interprets `x` depends on the shape of `f`:

- If `f` is a scalar, `x` is interpreted as a time-domain signal, and `f` is interpreted as the sample rate. In this case, `x` must be a real vector or matrix. If `x` is specified as a matrix, the columns are interpreted as individual channels.
- If `f` is a vector, `x` is interpreted as a frequency-domain signal, and `f` is interpreted as the frequencies, in Hz, corresponding to the rows of `x`. In this case, `x` must be a real *L*-by-*M*-by-*N* array, where *L* is the number of spectral values at given frequencies of `f`, *M* is the number of individual spectra, and *N* is the number of channels.
- The number of rows of `x`, *L*, must be equal to the number of elements of `f`.

Data Types: `single` | `double`

Name-Value Arguments

Note The following name-value arguments apply if `x` is a time-domain signal. If `x` is a frequency-domain signal, name-value arguments are ignored.

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `Window=hamming(256)`

`Window` — Window applied in time domain

`rectwin(round(f*0.03))` (default) | vector

Window applied in the time domain, specified as a real vector. The number of elements in the vector must be in the range [1, `size(x,1)`]. The number of elements in the vector must also be greater than `OverlapLength`.

Data Types: `single` | `double`

`OverlapLength` — Number of samples overlapped between adjacent windows

`round(f*0.02)` (default) | non-negative scalar

Number of samples overlapped between adjacent windows, specified as an integer in the range [0, `size(Window,1)`).

Data Types: `single` | `double`

`FFTLength` — Number of bins in DFT
`numel(Window)` (default) | positive scalar integer

Number of bins used to calculate the DFT of windowed input samples, specified as a positive scalar integer. If unspecified, `FFTLength` defaults to the number of elements in the `Window`.

Data Types: `single` | `double`

`Range` — Frequency range (Hz)
`[0,f/2]` (default) | two-element row vector

Frequency range in Hz, specified as a two-element row vector of increasing real values in the range [0, `f`/2].

Data Types: `single` | `double`

`SpectrumType` — Spectrum type
`"magnitude"` (default) | `"power"`

Spectrum type, specified as `"power"` or `"magnitude"`:

- `"power"` –– The spectral slope is calculated for the one-sided power spectrum.
- `"magnitude"` –– The spectral slope is calculated for the one-sided magnitude spectrum.

Data Types: `char` | `string`

Output Arguments

`slope` — Spectral slope
scalar | vector | matrix

Spectral slope in Hz, returned as a scalar, vector, or matrix. Each row of `slope` corresponds to the spectral slope of a window of `x`. Each column of `slope` corresponds to an independent channel.

Algorithms

The spectral slope is calculated as described in [1]:

$$\text{slope} = \frac{\sum_{k=b_1}^{b_2} (f_k - \mu_f)(s_k - \mu_S)}{\sum_{k=b_1}^{b_2} (f_k - \mu_f)^2}$$

where

- f_k is the frequency in Hz corresponding to bin k.
- μ_f is the mean frequency.
- s_k is the spectral value at bin k.
- μ_s is the mean spectral value.

- b_1 and b_2 are the band edges, in bins, over which to calculate the spectral slope.

Version History

Introduced in R2019a

References

[1] Lerch, Alexander. *An Introduction to Audio Content Analysis Applications in Signal Processing and Music Informatics*. Piscataway, NJ: IEEE Press, 2012.

Extended Capabilities

C/C++ Code Generation

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

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

See Also

`spectralCrest` | `spectralDecrease`

Topics

“Spectral Descriptors”

spectralSkewness

Spectral skewness for audio signals and auditory spectrograms

Syntax

```
skewness = spectralSkewness(x,f)
skewness = spectralSkewness(x,f,Name=Value)
[skewness,spread,centroid] = spectralSkewness( ___ )
spectralSkewness( ___ )
```

Description

`skewness = spectralSkewness(x,f)` returns the spectral skewness of the signal, `x`, over time. How the function interprets `x` depends on the shape of `f`.

`skewness = spectralSkewness(x,f,Name=Value)` specifies options using one or more name-value arguments.

`[skewness,spread,centroid] = spectralSkewness( ___ )` returns the spectral spread and spectral centroid. You can specify an input combination from any of the previous syntaxes.

`spectralSkewness( ___ )` with no output arguments plots the spectral skewness.

- If the input is in the time domain, the spectral skewness is plotted against time.
- If the input is in the frequency domain, the spectral skewness is plotted against frame number.

Examples

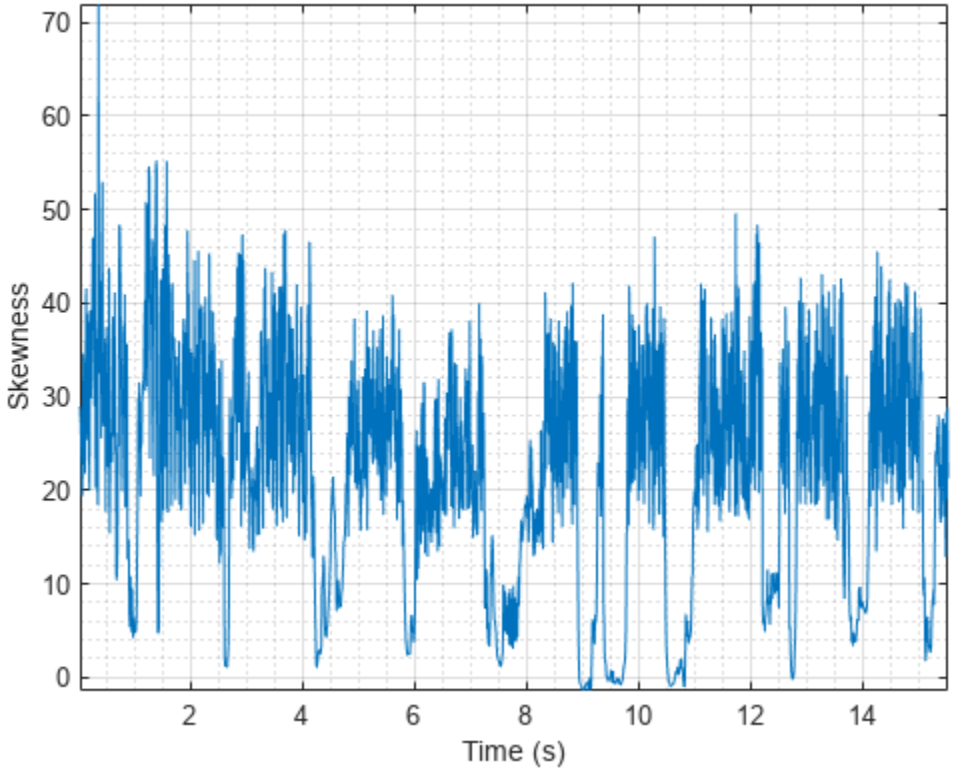
Spectral Skewness of Time-Domain Audio

Read in an audio file, calculate the skewness using default parameters.

```
[audioIn,fs] = audioread("Counting-16-44p1-mono-15secs.wav");
skewness = spectralSkewness(audioIn,fs);
```

Plot the spectral skewness against time.

```
spectralSkewness(audioIn,fs)
```



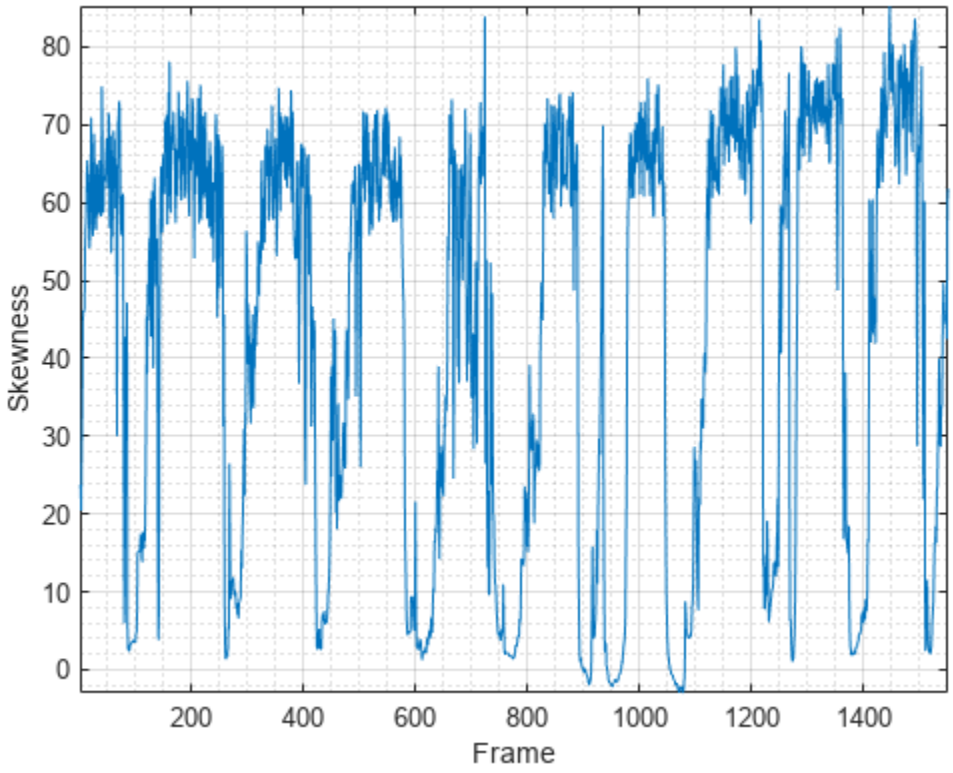
Spectral Skewness of Frequency-Domain Audio Data

Read in an audio file and then calculate the mel spectrogram using the `melSpectrogram` function. Calculate the skewness of the mel spectrogram over time.

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

[s,cf,t] = melSpectrogram(audioIn,fs);

skewness = spectralSkewness(s,cf);
```

Plot the spectral skewness against the frame number.

```
spectralSkewness(s,cf)
```



Specify Nondefault Parameters

Read in an audio file.

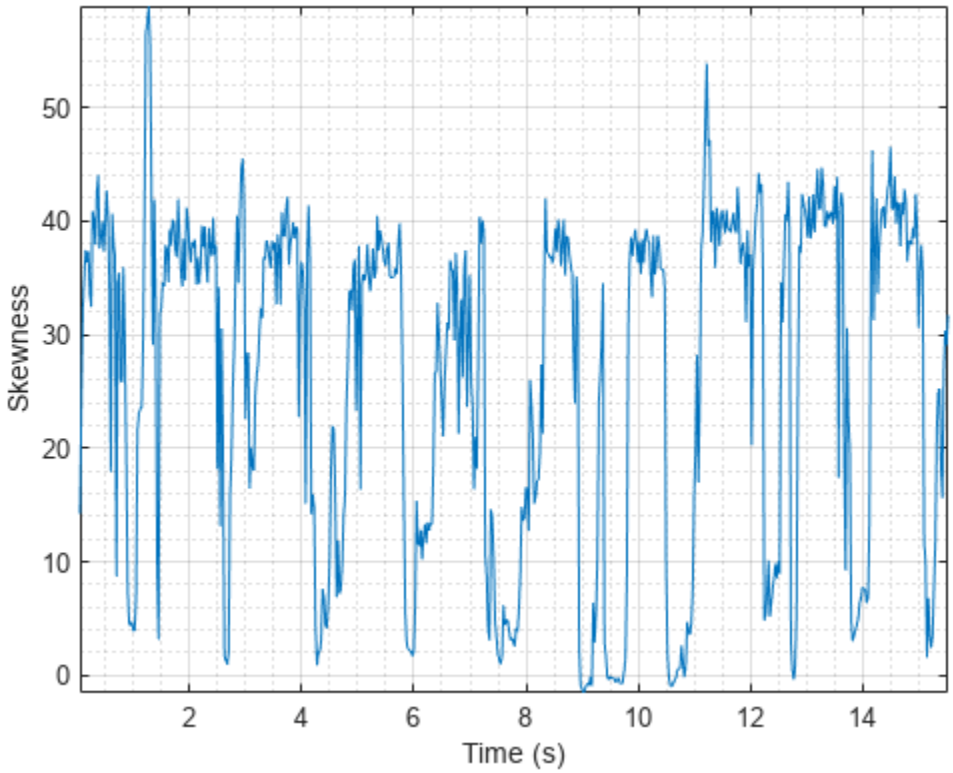
```
[audioIn,fs] = audioread("Counting-16-44p1-mono-15secs.wav");
```

Calculate the skewness of the power spectrum over time. Calculate the skewness for 50 ms Hamming windows of data with 25 ms overlap. Use the range from 62.5 Hz to `fs`/2 for the skewness calculation.

```
skewness = spectralSkewness(audioIn,fs, ...
                    Window=hamming(round(0.05*fs)), ...
                    OverlapLength=round(0.025*fs), ...
                    Range=[62.5,fs/2]);
```

Plot the spectral skewness.

```
spectralSkewness(audioIn,fs, ...
                 Window=hamming(round(0.05*fs)), ...
                 OverlapLength=round(0.025*fs), ...
                 Range=[62.5,fs/2])
```



Calculate Spectral Skewness of Streaming Audio

Create a `dsp.AudioFileReader` object to read in audio data frame-by-frame. Create a `dsp.SignalSink` to log the spectral skewness calculation.

```
fileReader = dsp.AudioFileReader('Counting-16-44p1-mono-15secs.wav');
logger = dsp.SignalSink;
```

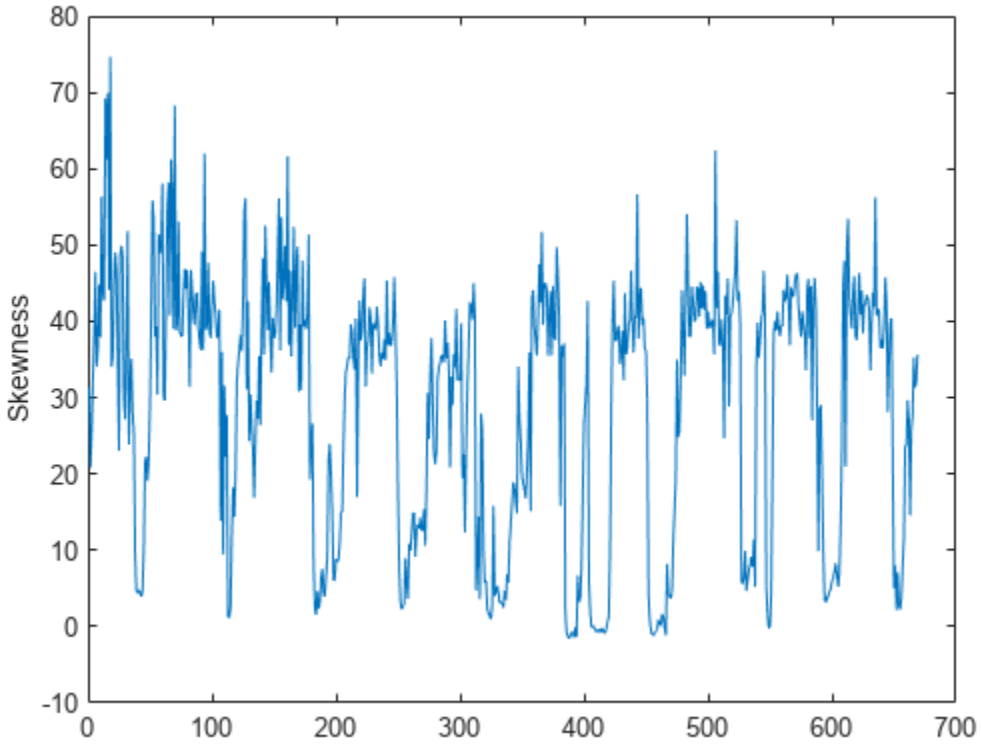
In an audio stream loop:

1. Read in a frame of audio data.
2. Calculate the spectral skewness for the frame of audio.
3. Log the spectral skewness for later plotting.

To calculate the spectral skewness for only a given input frame, specify a window with the same number of samples as the input, and set the overlap length to zero. Plot the logged data.

```
win = hamming(fileReader.SamplesPerFrame);
while ~isDone(fileReader)
    audioIn = fileReader();
    skewness = spectralSkewness(audioIn,fileReader.SampleRate, ...
                                'Window',win, ...
                                'OverlapLength',0);
    logger(skewness)
```

```
end

plot(logger.Buffer)
ylabel('Skewness')
```



Use `dsp.AsyncBuffer` if

- The input to your audio stream loop has a variable samples-per-frame.
- The input to your audio stream loop has an inconsistent samples-per-frame with the analysis window of `spectralSkewness`.
- You want to calculate the spectral skewness for overlapped data.

Create a `dsp.AsyncBuffer` object, reset the logger, and release the file reader.

```
buff = dsp.AsyncBuffer;
reset(logger)
release(fileReader)
```

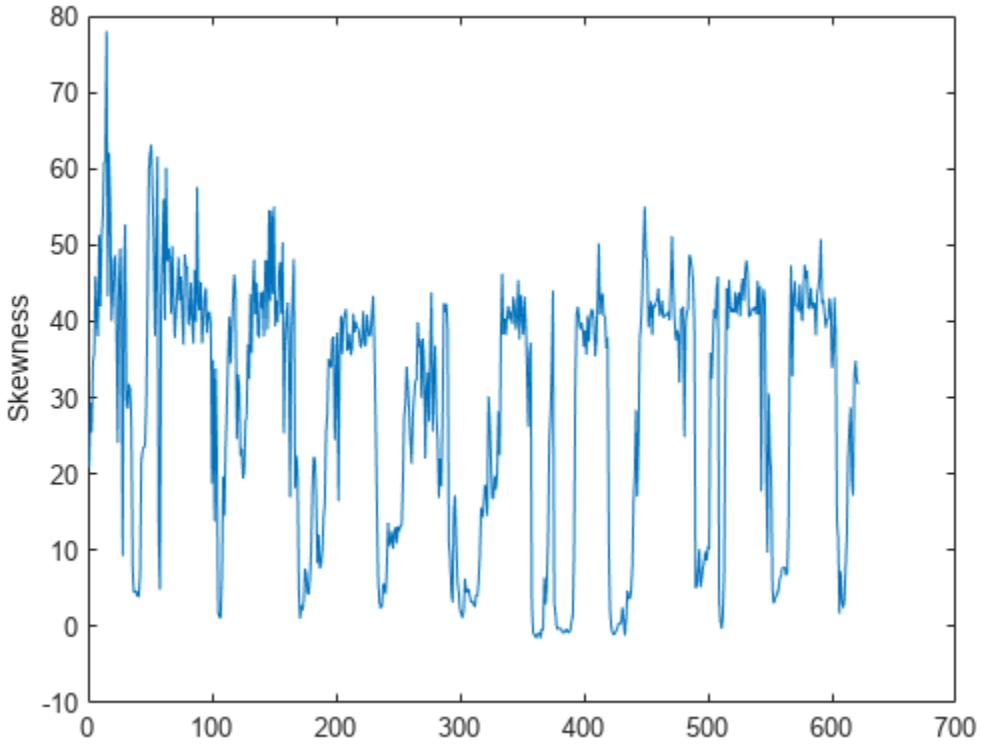
Specify that the spectral skewness is calculated for 50 ms frames with a 25 ms overlap.

```
fs = fileReader.SampleRate;

samplesPerFrame = round(fs*0.05);
samplesOverlap = round(fs*0.025);

samplesPerHop = samplesPerFrame - samplesOverlap;
```

```
win = hamming(samplesPerFrame);

while ~isDone(fileReader)
    audioIn = fileReader();
    write(buff,audioIn);

    while buff.NumUnreadSamples >= samplesPerHop
        audioBuffered = read(buff,samplesPerFrame,samplesOverlap);

        skewness = spectralSkewness(audioBuffered,fs, ...
                                    'Window',win, ...
                                    'OverlapLength',0);
        logger(skewness)
    end

end
release(fileReader)
```

Plot the logged data.

```
plot(logger.Buffer)
ylabel('Skewness')
```



Input Arguments

x — Input signal

column vector | matrix | 3-D array

Input signal, specified as a vector, matrix, or 3-D array. How the function interprets `x` depends on the shape of `f`.

Data Types: `single` | `double`

f — Sample rate or frequency vector (Hz)

scalar | vector

Sample rate or frequency vector in Hz, specified as a scalar or vector, respectively. How the function interprets `x` depends on the shape of `f`:

- If `f` is a scalar, `x` is interpreted as a time-domain signal, and `f` is interpreted as the sample rate. In this case, `x` must be a real vector or matrix. If `x` is specified as a matrix, the columns are interpreted as individual channels.
- If `f` is a vector, `x` is interpreted as a frequency-domain signal, and `f` is interpreted as the frequencies, in Hz, corresponding to the rows of `x`. In this case, `x` must be a real *L*-by-*M*-by-*N* array, where *L* is the number of spectral values at given frequencies of `f`, *M* is the number of individual spectra, and *N* is the number of channels.
- The number of rows of `x`, *L*, must be equal to the number of elements of `f`.

Data Types: `single` | `double`

Name-Value Arguments

Note The following name-value arguments apply if `x` is a time-domain signal. If `x` is a frequency-domain signal, name-value arguments are ignored.

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `Window=hamming(256)`

Window — Window applied in time domain

`rectwin(round(f*0.03))` (default) | vector

Window applied in the time domain, specified as a real vector. The number of elements in the vector must be in the range [1, `size(x,1)`]. The number of elements in the vector must also be greater than `OverlapLength`.

Data Types: `single` | `double`

OverlapLength — Number of samples overlapped between adjacent windows

`round(f*0.02)` (default) | non-negative scalar

Number of samples overlapped between adjacent windows, specified as an integer in the range [0, `size(Window,1)`).

Data Types: `single` | `double`

`FFTLength` — Number of bins in DFT
`numel(Window)` (default) | positive scalar integer

Number of bins used to calculate the DFT of windowed input samples, specified as a positive scalar integer. If unspecified, `FFTLength` defaults to the number of elements in the `Window`.

Data Types: `single` | `double`

`Range` — Frequency range (Hz)
`[0,f/2]` (default) | two-element row vector

Frequency range in Hz, specified as a two-element row vector of increasing real values in the range [0, `f`/2].

Data Types: `single` | `double`

`SpectrumType` — Spectrum type
`"power"` (default) | `"magnitude"`

Spectrum type, specified as `"power"` or `"magnitude"`:

- `"power"` –- The spectral skewness is calculated for the one-sided power spectrum.
- `"magnitude"` –- The spectral skewness is calculated for the one-sided magnitude spectrum.

Data Types: `char` | `string`

Output Arguments

`skewness` — Spectral skewness
scalar | vector | matrix

Spectral skewness, returned as a scalar, vector, or matrix. Each row of `skewness` corresponds to the spectral skewness of a window of `x`. Each column of `skewness` corresponds to an independent channel.

`spread` — Spectral spread
scalar | vector | matrix

Spectral spread, returned as a scalar, vector, or matrix. Each row of `spread` corresponds to the spectral spread of a window of `x`. Each column of `spread` corresponds to an independent channel.

`centroid` — Spectral centroid (Hz)
scalar | vector | matrix

Spectral centroid in Hz, returned as a scalar, vector, or matrix. Each row of `centroid` corresponds to the spectral centroid of a window of `x`. Each column of `centroid` corresponds to an independent channel.

Algorithms

The spectral skewness is calculated as described in [1]:

$$\text{skewness} = \frac{\sum_{k=b_1}^{b_2} (f_k - \mu_1)^3 s_k}{(\mu_2)^3 \sum_{k=b_1}^{b_2} s_k}$$

where

- f_k is the frequency in Hz corresponding to bin k.
- s_k is the spectral value at bin k.
- b_1 and b_2 are the band edges, in bins, over which to calculate the spectral skewness.
- μ_1 is the spectral centroid, calculated as described by the `spectralCentroid` function.
- μ_2 is the spectral spread, calculated as described by the `spectralSpread` function.

Version History

Introduced in R2019a

References

[1] Peeters, G. "A Large Set of Audio Features for Sound Description (Similarity and Classification) in the CUIDADO Project." Technical Report; IRCAM: Paris, France, 2004.

Extended Capabilities

C/C++ Code Generation

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

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

See Also

`spectralCentroid` | `spectralSpread` | `spectralKurtosis`

Topics

“Spectral Descriptors”

spectralRolloffPoint

Spectral rolloff point for audio signals and auditory spectrograms

Syntax

```
rolloffPoint = spectralRolloffPoint(x,f)
rolloffPoint = spectralRolloffPoint(x,f,Name=Value)
spectralRolloffPoint( ___ )
```

Description

`rolloffPoint = spectralRolloffPoint(x,f)` returns the spectral rolloff point of the signal, `x`, over time. How the function interprets `x` depends on the shape of `f`.

`rolloffPoint = spectralRolloffPoint(x,f,Name=Value)` specifies options using one or more name-value arguments.

`spectralRolloffPoint( ___ )` with no output arguments plots the spectral rolloff point. You can specify an input combination from any of the previous syntaxes.

- If the input is in the time domain, the spectral rolloff point is plotted against time.
- If the input is in the frequency domain, the spectral rolloff point is plotted against frame number.

Examples

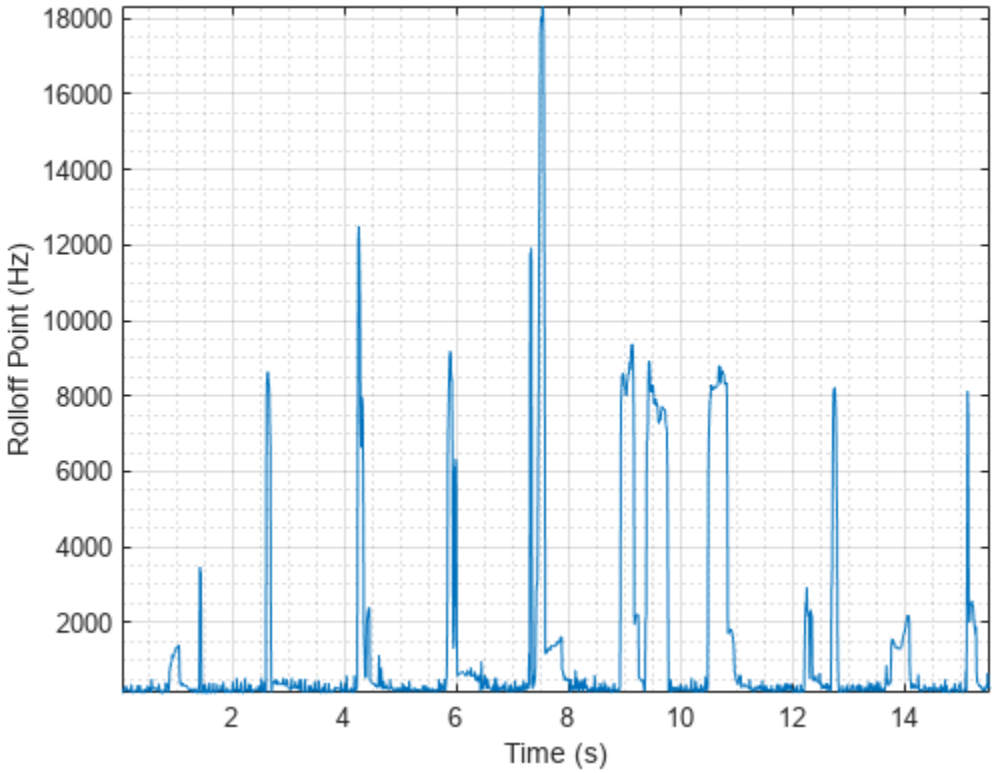
Spectral Rolloff Point of Time-Domain Audio

Read in an audio file. Calculate the rolloff point using default parameters.

```
[audioIn,fs] = audioread("Counting-16-44p1-mono-15secs.wav");
rolloffPoint = spectralRolloffPoint(audioIn,fs);
```

Plot the spectral rolloff point against time.

```
spectralRolloffPoint(audioIn,fs)
```



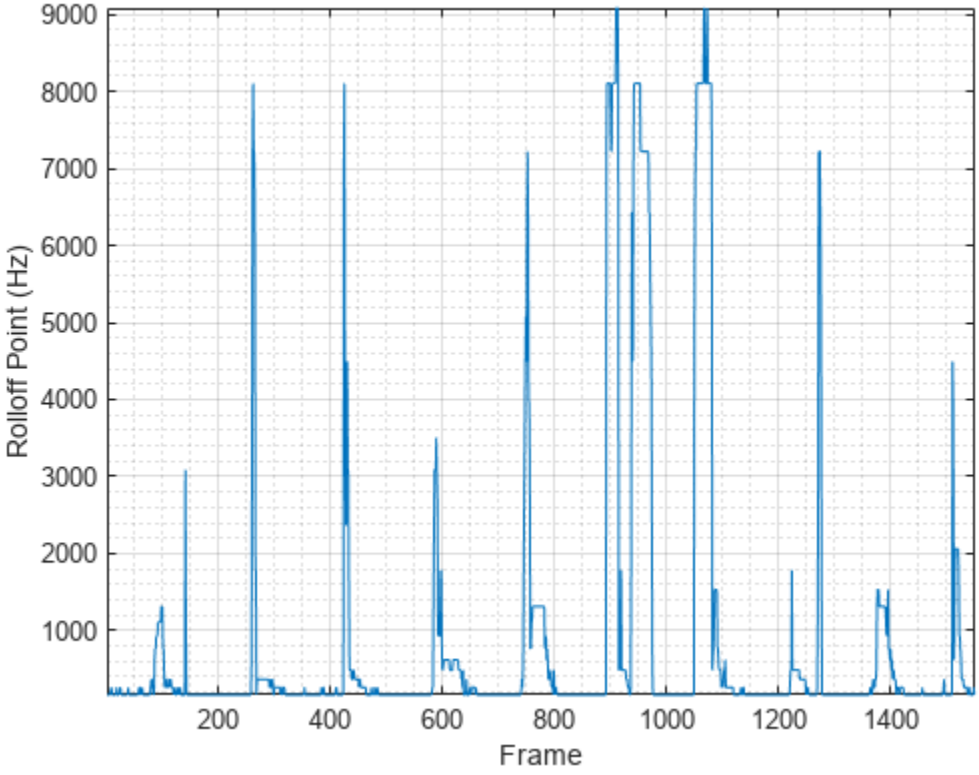
Spectral Rolloff Point of Frequency-Domain Audio Data

Read in an audio file and then calculate the mel spectrogram using the `melSpectrogram` function. Calculate the rolloff point of the mel spectrogram over time.

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

[s,cf,t] = melSpectrogram(audioIn,fs);

rolloffPoint = spectralRolloffPoint(s,cf);
```

Plot the spectral rolloff point against the frame number.

```
spectralRolloffPoint(s,cf)
```



Specify Nondefault Parameters

Read in an audio file.

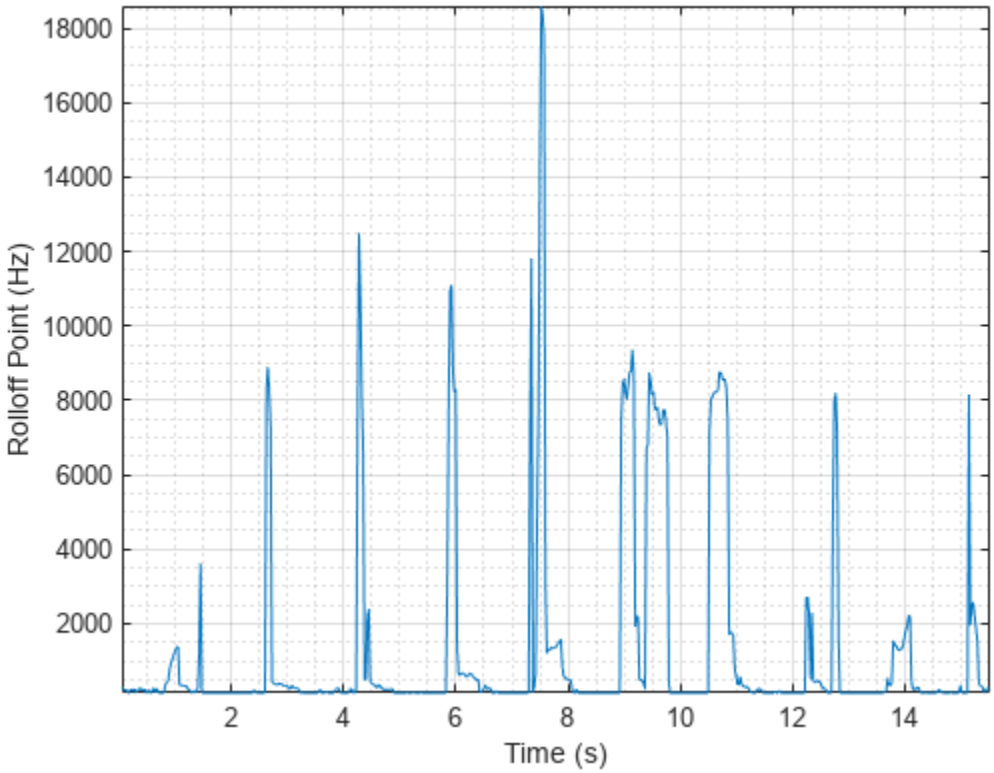
```
[audioIn,fs] = audioread("Counting-16-44p1-mono-15secs.wav");
```

Calculate the rolloff point of the power spectrum over time. Calculate the rolloff point for 50 ms Hamming windows of data with 25 ms overlap. Use the range from 62.5 Hz to `fs`/2 for the rolloff point calculation.

```
rolloffPoint = spectralRolloffPoint(audioIn,fs, ...
                    Window=hamming(round(0.05*fs)), ...
                    OverlapLength=round(0.025*fs), ...
                    Range=[62.5,fs/2]);
```

Plot the spectral rolloff point against time.

```
spectralRolloffPoint(audioIn,fs, ...
                    Window=hamming(round(0.05*fs)), ...
                    OverlapLength=round(0.025*fs), ...
                    Range=[62.5,fs/2])
```



Calculate Spectral Rolloff Point of Streaming Audio

Create a `dsp.AudioFileReader` object to read in audio data frame-by-frame. Create a `dsp.SignalSink` to log the spectral rolloff point calculation.

```
fileReader = dsp.AudioFileReader('Counting-16-44p1-mono-15secs.wav');
logger = dsp.SignalSink;
```

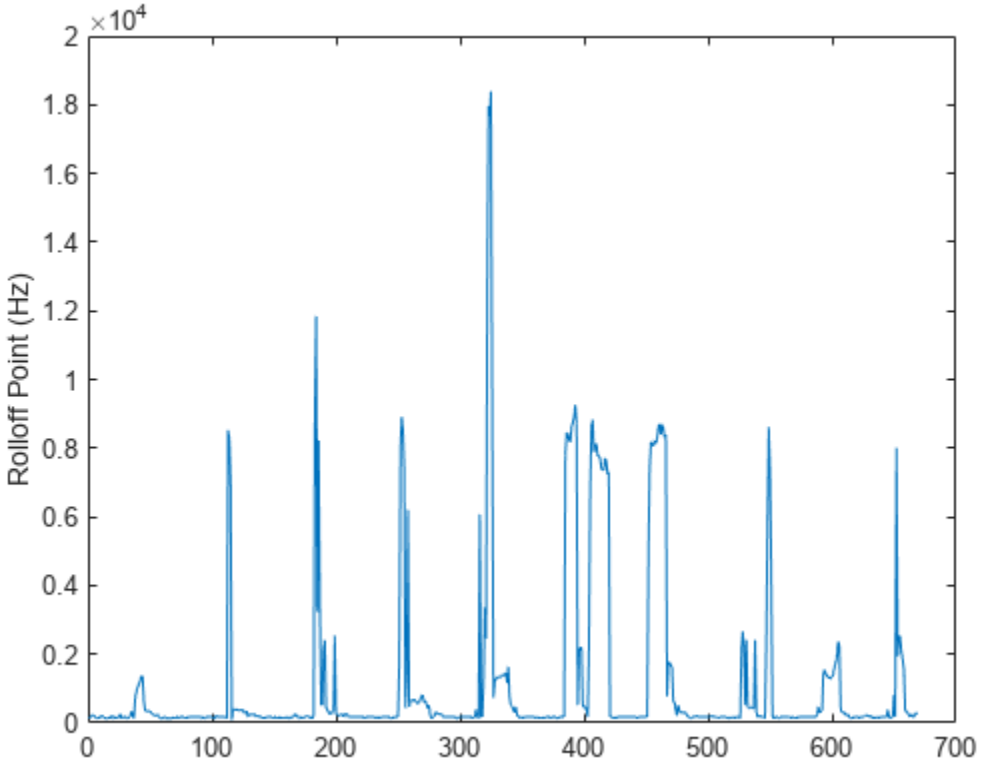
In an audio stream loop:

1. Read in a frame of audio data.
2. Calculate the spectral rolloff point for the frame of audio.
3. Log the spectral rolloff point for later plotting.

To calculate the spectral rolloff point for only a given input frame, specify a window with the same number of samples as the input, and set the overlap length to zero. Plot the logged data.

```
win = hamming(fileReader.SamplesPerFrame);
while ~isDone(fileReader)
    audioIn = fileReader();
    rolloffPoint = spectralRolloffPoint(audioIn,fileReader.SampleRate, ...
                                        'Window',win, ...
                                        'OverlapLength',0);
    logger(rolloffPoint)
```

```
end

plot(logger.Buffer)
ylabel('Rolloff Point (Hz)')
```



Use `dsp.AsyncBuffer` if

- The input to your audio stream loop has a variable samples-per-frame.
- The input to your audio stream loop has an inconsistent samples-per-frame with the analysis window of `spectralRolloffPoint`.
- You want to calculate the spectral rolloff point for overlapped data.

Create a `dsp.AsyncBuffer` object, reset the logger, and release the file reader.

```
buff = dsp.AsyncBuffer;
reset(logger)
release(fileReader)
```

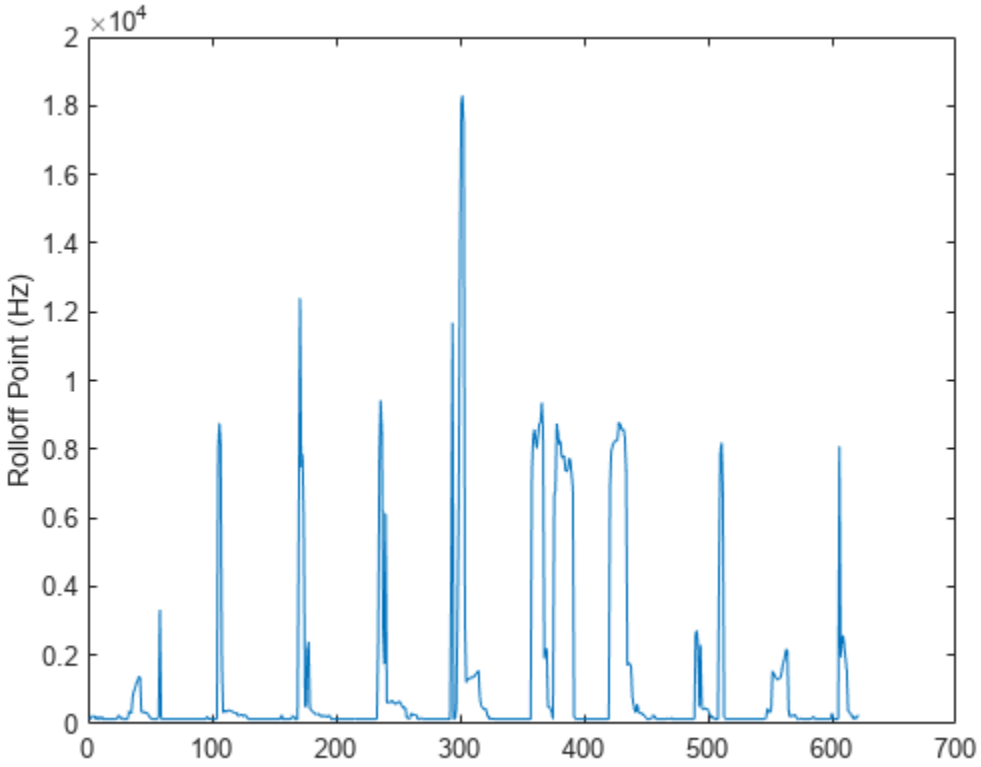
Specify that the spectral rolloff point is calculated for 50 ms frames with a 25 ms overlap.

```
fs = fileReader.SampleRate;

samplesPerFrame = round(fs*0.05);
samplesOverlap = round(fs*0.025);

samplesPerHop = samplesPerFrame - samplesOverlap;
```

```
win = hamming(samplesPerFrame);

while ~isDone(fileReader)
    audioIn = fileReader();
    write(buff,audioIn);

    while buff.NumUnreadSamples >= samplesPerHop
        audioBuffered = read(buff,samplesPerFrame,samplesOverlap);

        rolloffPoint = spectralRolloffPoint(audioBuffered,fs, ...
                                    'Window',win, ...
                                    'OverlapLength',0);
        logger(rolloffPoint)
    end

end
release(fileReader)
```

Plot the logged data.

```
plot(logger.Buffer)
ylabel('Rolloff Point (Hz)')
```



Input Arguments

`x` — Input signal

column vector | matrix | 3-D array

Input signal, specified as a vector, matrix, or 3-D array. How the function interprets `x` depends on the shape of `f`.

Data Types: `single` | `double`

`f` — Sample rate or frequency vector (Hz)

scalar | vector

Sample rate or frequency vector in Hz, specified as a scalar or vector, respectively. How the function interprets `x` depends on the shape of `f`:

- If `f` is a scalar, `x` is interpreted as a time-domain signal, and `f` is interpreted as the sample rate. In this case, `x` must be a real vector or matrix. If `x` is specified as a matrix, the columns are interpreted as individual channels.
- If `f` is a vector, `x` is interpreted as a frequency-domain signal, and `f` is interpreted as the frequencies, in Hz, corresponding to the rows of `x`. In this case, `x` must be a real *L*-by-*M*-by-*N* array, where *L* is the number of spectral values at given frequencies of `f`, *M* is the number of individual spectra, and *N* is the number of channels.
- The number of rows of `x`, *L*, must be equal to the number of elements of `f`.

Data Types: `single` | `double`

Name-Value Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `Window=hamming(256)`

`Threshold` — Threshold of rolloff point

`0.95` (default) | scalar in the range (0,1)

Threshold of rolloff point, specified as a scalar between zero and one, exclusive.

Data Types: `single` | `double`

Note The following name-value arguments apply if `x` is a time-domain signal. If `x` is a frequency-domain signal, name-value arguments are ignored.

`Window` — Window applied in time domain

`rectwin(round(f*0.03))` (default) | vector

Window applied in the time domain, specified as a real vector. The number of elements in the vector must be in the range [1, `size(x,1)`]. The number of elements in the vector must also be greater than `OverlapLength`.

Data Types: `single` | `double`

`OverlapLength` — Number of samples overlapped between adjacent windows
`round(f*0.02)` (default) | non-negative scalar

Number of samples overlapped between adjacent windows, specified as an integer in the range [0, `size(Window,1)`).

Data Types: `single` | `double`

`FFTLength` — Number of bins in DFT
`numel(Window)` (default) | positive scalar integer

Number of bins used to calculate the DFT of windowed input samples, specified as a positive scalar integer. If unspecified, `FFTLength` defaults to the number of elements in the `Window`.

Data Types: `single` | `double`

Range — Frequency range (Hz)
`[0,f/2]` (default) | two-element row vector

Frequency range in Hz, specified as a two-element row vector of increasing real values in the range [0, `f`/2].

Data Types: `single` | `double`

`SpectrumType` — Spectrum type
`"power"` (default) | `"magnitude"`

Spectrum type, specified as `"power"` or `"magnitude"`:

- `"power"` –– The spectral rolloff point is calculated for the one-sided power spectrum.
- `"magnitude"` –– The spectral rolloff point is calculated for the one-sided magnitude spectrum.

Data Types: `char` | `string`

Output Arguments

`rolloffPoint` — Spectral rolloff point (Hz)
scalar | vector | matrix

Spectral rolloff point in Hz, returned as a scalar, vector, or matrix. Each row of `rolloffPoint` corresponds to the spectral rolloff point of a window of `x`. Each column of `rolloffPoint` corresponds to an independent channel.

Algorithms

The spectral rolloff point is calculated as described in [1]:

rolloffPoint = i

such that

$$\sum_{k=b_1}^{i} s_k = \kappa \sum_{k=b_1}^{b_2} s_k$$

where

- s_k is the spectral value at bin k.
- b_1 and b_2 are the band edges, in bins, over which to calculate the spectral spread.
- κ is the percentage of total energy contained between b_1 and i. You can set κ using `Threshold`.

Version History

Introduced in R2019a

References

[1] Scheirer, E., and M. Slaney, "Construction and Evaluation of a Robust Multifeature Speech/Music Discriminator," *IEEE International Conference on Acoustics, Speech, and Signal Processing*. Volume 2, 1997, pp. 1221–1224.

Extended Capabilities

C/C++ Code Generation

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

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

See Also

`spectralSpread` | `spectralSkewness` | `spectralKurtosis`

Topics

“Spectral Descriptors”

spectralKurtosis

Spectral kurtosis for audio signals and auditory spectrograms

Syntax

```
kurtosis = spectralKurtosis(x,f)
kurtosis = spectralKurtosis(x,f,Name=Value)
[kurtosis,spread,centroid] = spectralKurtosis( ___ )
spectralKurtosis( ___ )
```

Description

`kurtosis = spectralKurtosis(x,f)` returns the spectral kurtosis of the signal, `x`, over time. How the function interprets `x` depends on the shape of `f`.

`kurtosis = spectralKurtosis(x,f,Name=Value)` specifies options using one or more name-value arguments.

`[kurtosis,spread,centroid] = spectralKurtosis( ___ )` returns the spectral spread and spectral centroid. You can specify an input combination from any of the previous syntaxes.

`spectralKurtosis( ___ )` with no output arguments plots the spectral kurtosis.

- If the input is in the time domain, the spectral kurtosis is plotted against time.
- If the input is in the frequency domain, the spectral kurtosis is plotted against frame number.

Examples

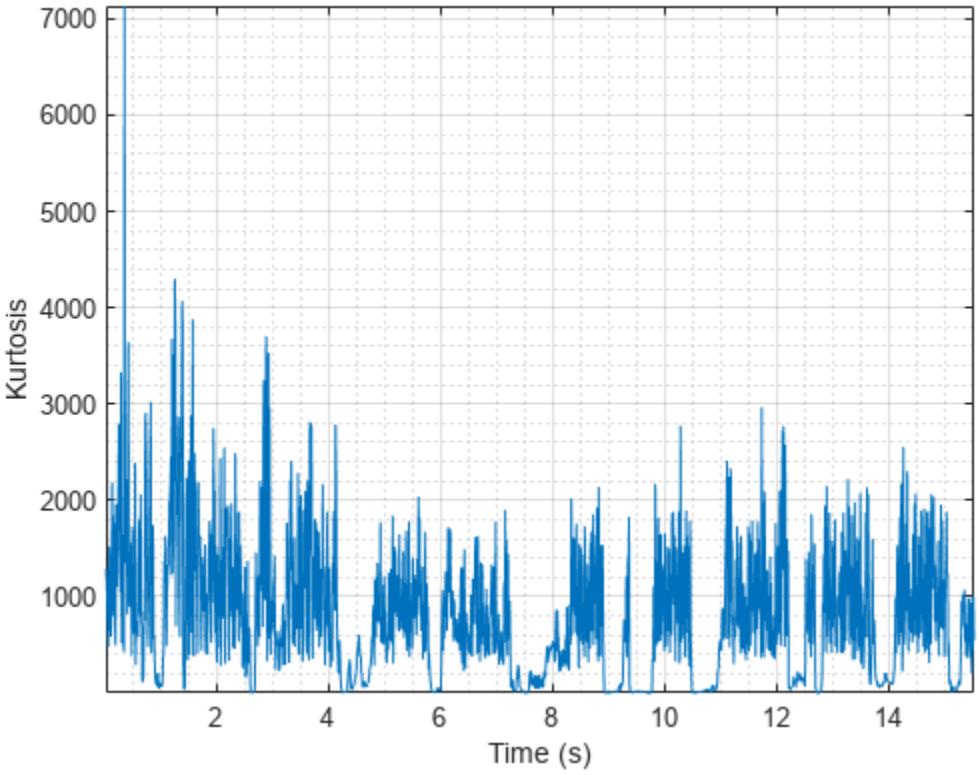
Spectral Kurtosis of Time-Domain Audio

Read in an audio file and calculate the kurtosis using default parameters.

```
[audioIn,fs] = audioread("Counting-16-44p1-mono-15secs.wav");
kurtosis = spectralKurtosis(audioIn,fs);
```

Plot the spectral kurtosis against time.

```
spectralKurtosis(audioIn,fs)
```



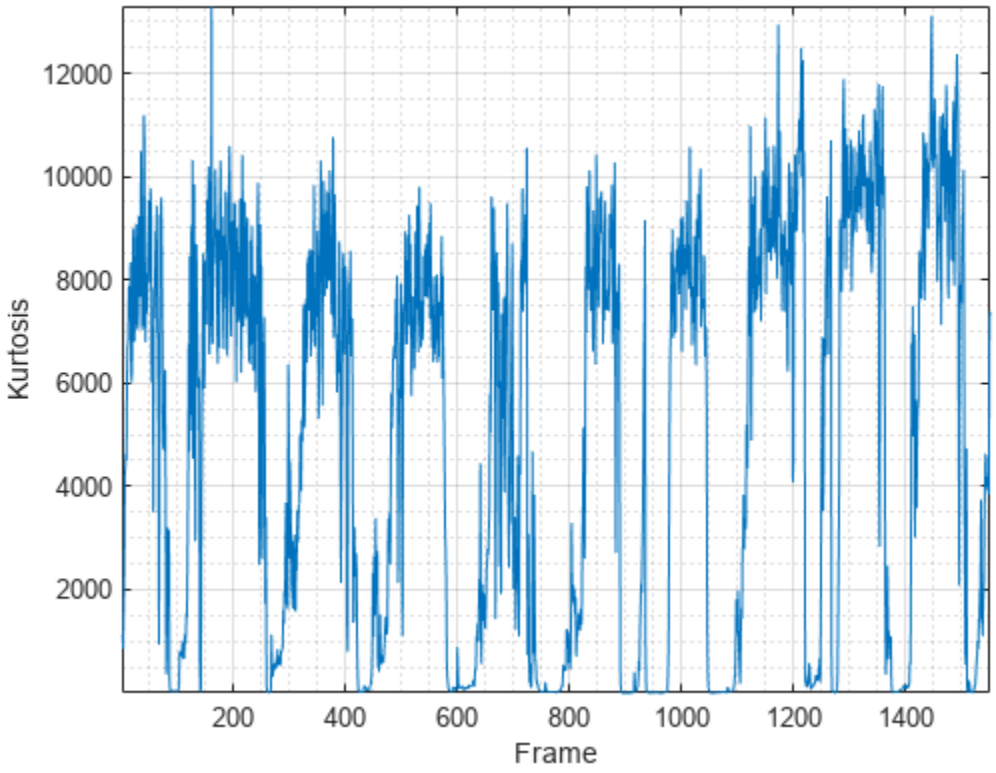
Spectral Kurtosis of Frequency-Domain Audio Data

Read in an audio file and then calculate the mel spectrogram using the `melSpectrogram` function. Calculate the kurtosis of the mel spectrogram over time.

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

[s,cf,t] = melSpectrogram(audioIn,fs);

kurtosis = spectralKurtosis(s,cf);
```

Plot the spectral kurtosis against the frame number.

```
spectralKurtosis(s,cf)
```



Specify Nondefault Parameters

Read in an audio file.

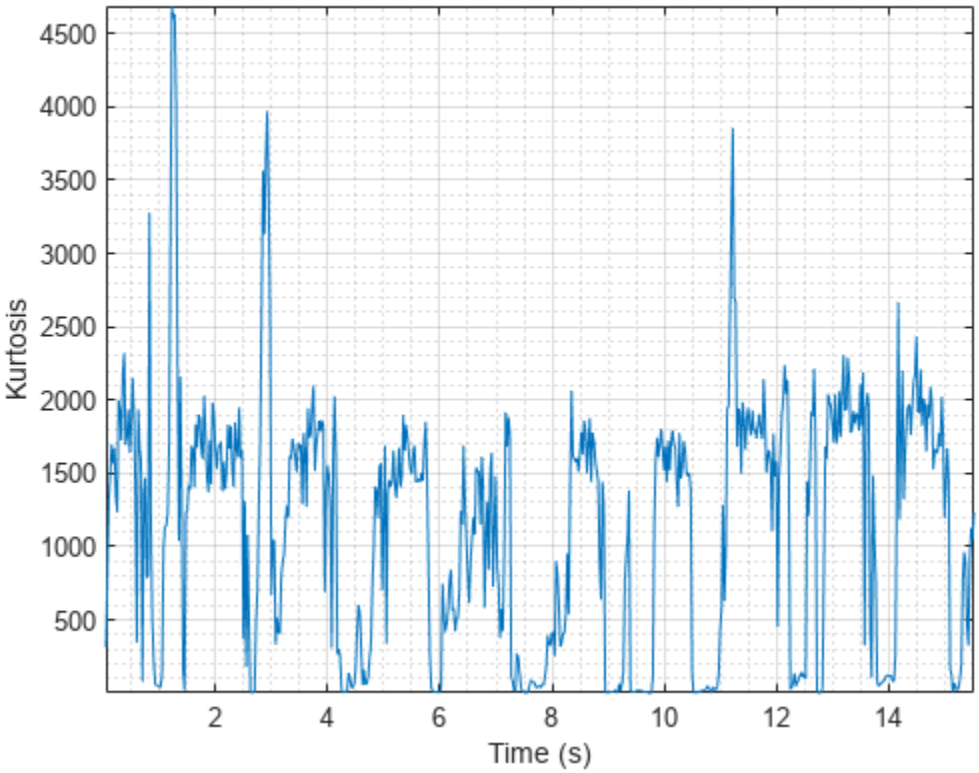
```
[audioIn,fs] = audioread("Counting-16-44p1-mono-15secs.wav");
```

Calculate the kurtosis of the power spectrum over time. Calculate the kurtosis for 50 ms Hamming windows of data with 25 ms overlap. Use the range from 62.5 Hz to `fs`/2 for the kurtosis calculation.

```
kurtosis = spectralKurtosis(audioIn,fs, ...
                    Window=hamming(round(0.05*fs)), ...
                    OverlapLength=round(0.025*fs), ...
                    Range=[62.5,fs/2]);
```

Plot the spectral kurtosis.

```
spectralKurtosis(audioIn,fs, ...
                 Window=hamming(round(0.05*fs)), ...
                 OverlapLength=round(0.025*fs), ...
                 Range=[62.5,fs/2])
```



Calculate Spectral Kurtosis of Streaming Audio

Create a `dsp.AudioFileReader` object to read in audio data frame-by-frame. Create a `dsp.SignalSink` to log the spectral kurtosis calculation.

```
fileReader = dsp.AudioFileReader('Counting-16-44p1-mono-15secs.wav');
logger = dsp.SignalSink;
```

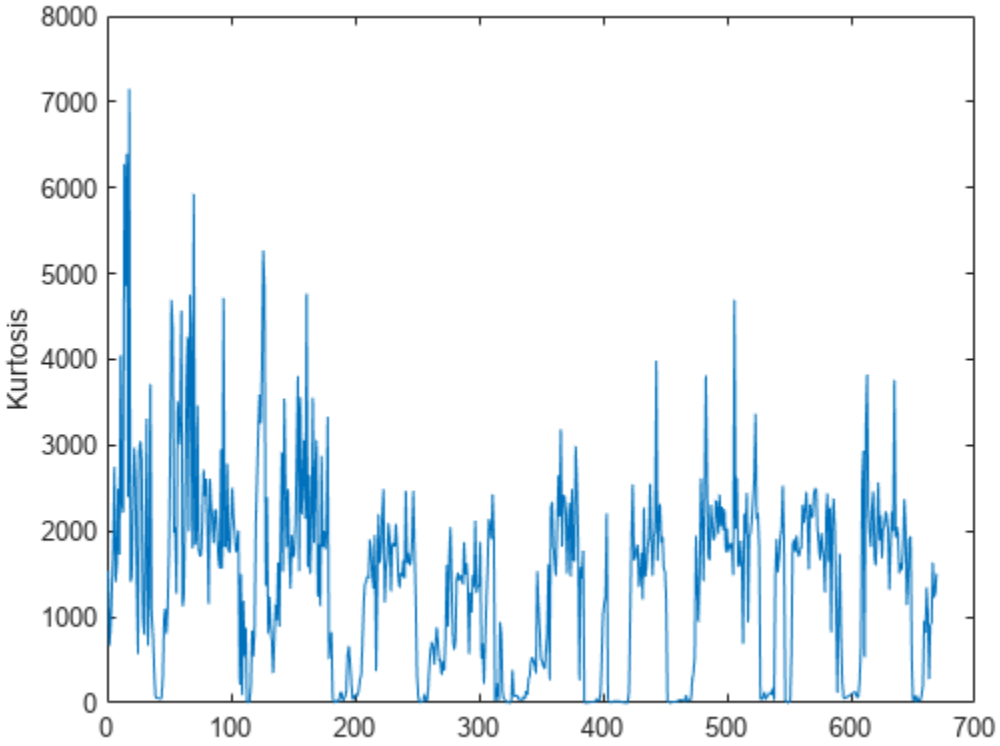
In an audio stream loop:

1. Read in a frame of audio data.
2. Calculate the spectral kurtosis for the frame of audio.
3. Log the spectral kurtosis for later plotting.

To calculate the spectral kurtosis for only a given input frame, specify a window with the same number of samples as the input, and set the overlap length to zero. Plot the logged data.

```
win = hamming(fileReader.SamplesPerFrame);
while ~isDone(fileReader)
    audioIn = fileReader();
    kurtosis = spectralKurtosis(audioIn,fileReader.SampleRate, ...
                                'Window',win, ...
                                'OverlapLength',0);
    logger(kurtosis)
```

```
end

plot(logger.Buffer)
ylabel('Kurtosis')
```



Use `dsp.AsyncBuffer` if

- The input to your audio stream loop has a variable samples-per-frame.
- The input to your audio stream loop has an inconsistent samples-per-frame with the analysis window of `spectralKurtosis`.
- You want to calculate the spectral kurtosis for overlapped data.

Create a `dsp.AsyncBuffer` object, reset the logger, and release the file reader.

```
buff = dsp.AsyncBuffer;
reset(logger)
release(fileReader)
```

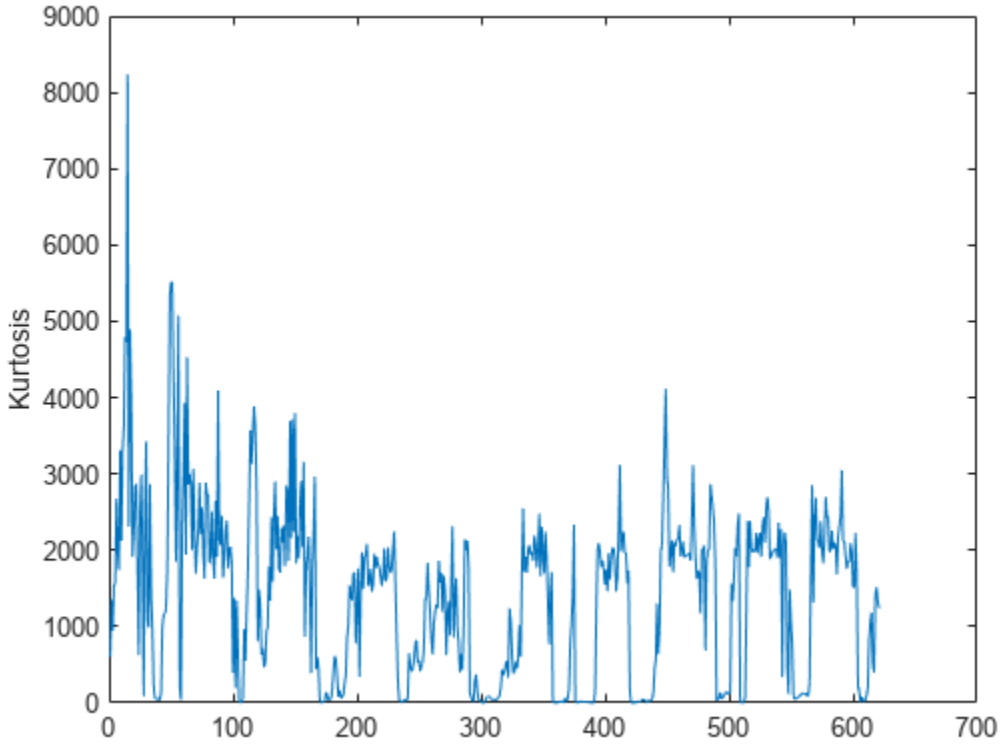
Specify that the spectral kurtosis is calculated for 50 ms frames with a 25 ms overlap.

```
fs = fileReader.SampleRate;

samplesPerFrame = round(fs*0.05);
samplesOverlap = round(fs*0.025);

samplesPerHop = samplesPerFrame - samplesOverlap;
```

```
win = hamming(samplesPerFrame);

while ~isDone(fileReader)
    audioIn = fileReader();
    write(buff,audioIn);

    while buff.NumUnreadSamples >= samplesPerHop
        audioBuffered = read(buff,samplesPerFrame,samplesOverlap);

        kurtosis = spectralKurtosis(audioBuffered,fs, ...
                                    'Window',win, ...
                                    'OverlapLength',0);
        logger(kurtosis)
    end

end
release(fileReader)
```

Plot the logged data.

```
plot(logger.Buffer)
ylabel('Kurtosis')
```



Input Arguments

x — Input signal

column vector | matrix | 3-D array

Input signal, specified as a vector, matrix, or 3-D array. How the function interprets `x` depends on the shape of `f`.

Data Types: `single` | `double`

f — Sample rate or frequency vector (Hz)

scalar | vector

Sample rate or frequency vector in Hz, specified as a scalar or vector, respectively. How the function interprets `x` depends on the shape of `f`:

- If `f` is a scalar, `x` is interpreted as a time-domain signal, and `f` is interpreted as the sample rate. In this case, `x` must be a real vector or matrix. If `x` is specified as a matrix, the columns are interpreted as individual channels.
- If `f` is a vector, `x` is interpreted as a frequency-domain signal, and `f` is interpreted as the frequencies, in Hz, corresponding to the rows of `x`. In this case, `x` must be a real *L*-by-*M*-by-*N* array, where *L* is the number of spectral values at given frequencies of `f`, *M* is the number of individual spectra, and *N* is the number of channels.
- The number of rows of `x`, *L*, must be equal to the number of elements of `f`.

Data Types: `single` | `double`

Name-Value Arguments

Note The following name-value arguments apply if `x` is a time-domain signal. If `x` is a frequency-domain signal, name-value arguments are ignored.

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `Window=hamming(256)`

Window — Window applied in time domain

`rectwin(round(f*0.03))` (default) | vector

Window applied in the time domain, specified as a real vector. The number of elements in the vector must be in the range [1, `size(x,1)`]. The number of elements in the vector must also be greater than `OverlapLength`.

Data Types: `single` | `double`

OverlapLength — Number of samples overlapped between adjacent windows

`round(f*0.02)` (default) | non-negative scalar

Number of samples overlapped between adjacent windows, specified as an integer in the range [0, `size(Window,1)`).

Data Types: `single` | `double`

`FFTLength` — Number of bins in DFT
`numel(Window)` (default) | positive scalar integer

Number of bins used to calculate the DFT of windowed input samples, specified as a positive scalar integer. If unspecified, `FFTLength` defaults to the number of elements in the `Window`.

Data Types: `single` | `double`

`Range` — Frequency range (Hz)
`[0,f/2]` (default) | two-element row vector

Frequency range in Hz, specified as a two-element row vector of increasing real values in the range [0, `f`/2].

Data Types: `single` | `double`

`SpectrumType` — Spectrum type
`"power"` (default) | `"magnitude"`

Spectrum type, specified as `"power"` or `"magnitude"`:

- `"power"` –- The spectral kurtosis is calculated for the one-sided power spectrum.
- `"magnitude"` –- The spectral kurtosis is calculated for the one-sided magnitude spectrum.

Data Types: `char` | `string`

Output Arguments

`kurtosis` — Spectral kurtosis
scalar | vector | matrix

Spectral kurtosis, returned as a scalar, vector, or matrix. Each row of `kurtosis` corresponds to the spectral kurtosis of a window of `x`. Each column of `kurtosis` corresponds to an independent channel.

`spread` — Spectral spread
scalar | vector | matrix

Spectral spread, returned as a scalar, vector, or matrix. Each row of `spread` corresponds to the spectral spread of a window of `x`. Each column of `spread` corresponds to an independent channel.

`centroid` — Spectral centroid (Hz)
scalar | vector | matrix

Spectral centroid in Hz, returned as a scalar, vector, or matrix. Each row of `centroid` corresponds to the spectral centroid of a window of `x`. Each column of `centroid` corresponds to an independent channel.

Algorithms

The spectral kurtosis is calculated as described in [1]:

$$\text{kurtosis} = \frac{\sum_{k=b_1}^{b_2} (f_k - \mu_1)^4 s_k}{(\mu_2)^4 \sum_{k=b_1}^{b_2} s_k}$$

where

- f_k is the frequency in Hz corresponding to bin k.
- s_k is the spectral value at bin k.
- b_1 and b_2 are the band edges, in bins, over which to calculate the spectral skewness.
- μ_1 is the spectral centroid, calculated as described by the `spectralCentroid` function.
- μ_2 is the spectral spread, calculated as described by the `spectralSpread` function.

Version History

Introduced in R2019a

References

[1] Peeters, G. "A Large Set of Audio Features for Sound Description (Similarity and Classification) in the CUIDADO Project." Technical Report; IRCAM: Paris, France, 2004.

Extended Capabilities

C/C++ Code Generation

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

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

See Also

`spectralCentroid` | `spectralSpread` | `spectralSkewness`

Topics

“Spectral Descriptors”

spectralFlux

Spectral flux for audio signals and auditory spectrograms

Syntax

```
flux = spectralFlux(x,f)
flux = spectralFlux(x,f,initialCondition)
flux = spectralFlux( ___ ,Name=Value)
[flux,finalCondition] = spectralFlux( ___ )
spectralFlux( ___ )
```

Description

`flux = spectralFlux(x,f)` returns the spectral flux of the signal, `x`, over time. Spectral flux is a measure of the variability of the spectrum over time. How the function interprets `x` depends on the shape of `f`.

`flux = spectralFlux(x,f,initialCondition)` specifies the previous spectral state. This syntax is supported only for frequency-domain inputs.

`flux = spectralFlux( ___ ,Name=Value)` specifies options using one or more name-value arguments.

For example, `flux = spectralFlux(x,f,NormType=1)` calculates spectral flux using norm type 1.

`[flux,finalCondition] = spectralFlux( ___ )` also returns the final spectral state. You can specify an input combination from any of the previous syntaxes.

`spectralFlux( ___ )` with no output arguments plots the spectral flux.

- If the input is in the time domain, the spectral flux is plotted against time.
- If the input is in the frequency domain, the spectral flux is plotted against frame number.

Examples

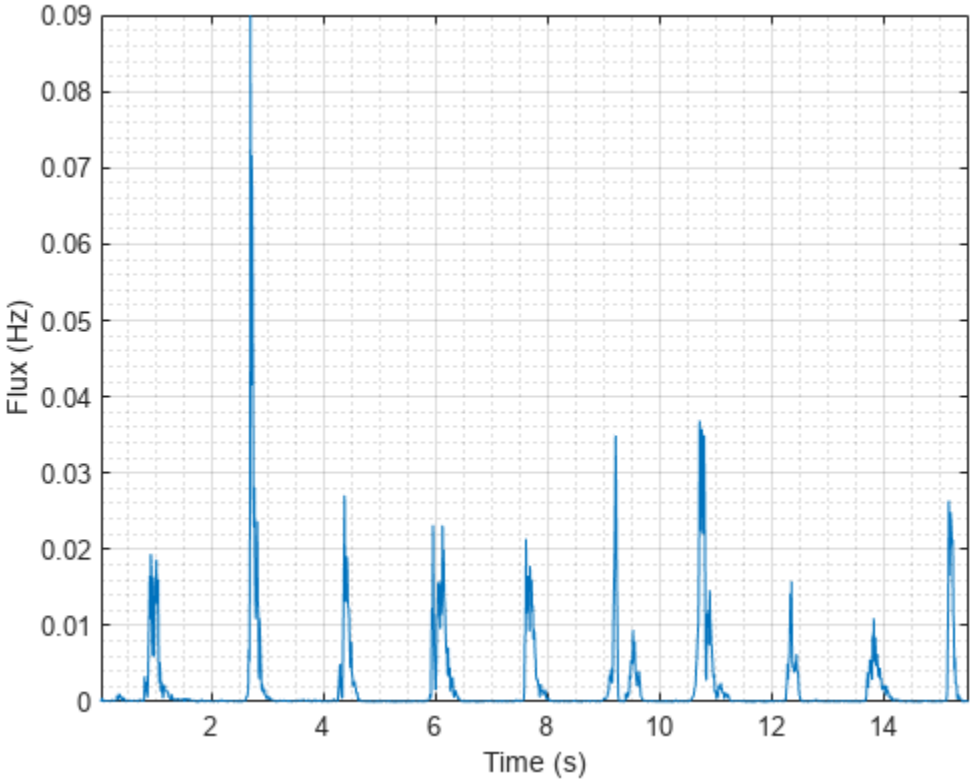
Spectral Flux of Time-Domain Audio

Read in an audio file and calculate the flux using default parameters.

```
[audioIn,fs] = audioread("Counting-16-44p1-mono-15secs.wav");
flux = spectralFlux(audioIn,fs);
```

Plot the spectral flux against time.

```
spectralFlux(audioIn,fs)
```



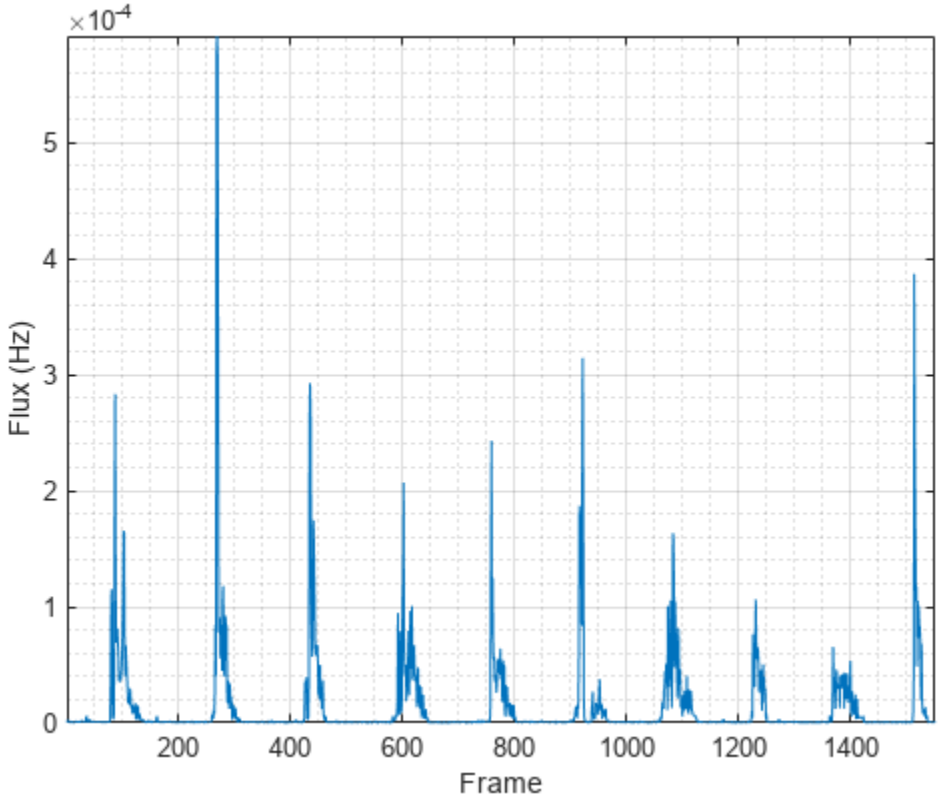
Spectral Flux of Frequency-Domain Audio Data

Read in an audio file and then calculate the mel spectrogram using the `melSpectrogram` function. Calculate the flux of the mel spectrogram over time.

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

[s,cf,t] = melSpectrogram(audioIn,fs);

flux = spectralFlux(s,cf);
```

Plot the spectral flux against the frame number.

```
spectralFlux(s,cf)
```



Specify Nondefault Parameters

Read in an audio file.

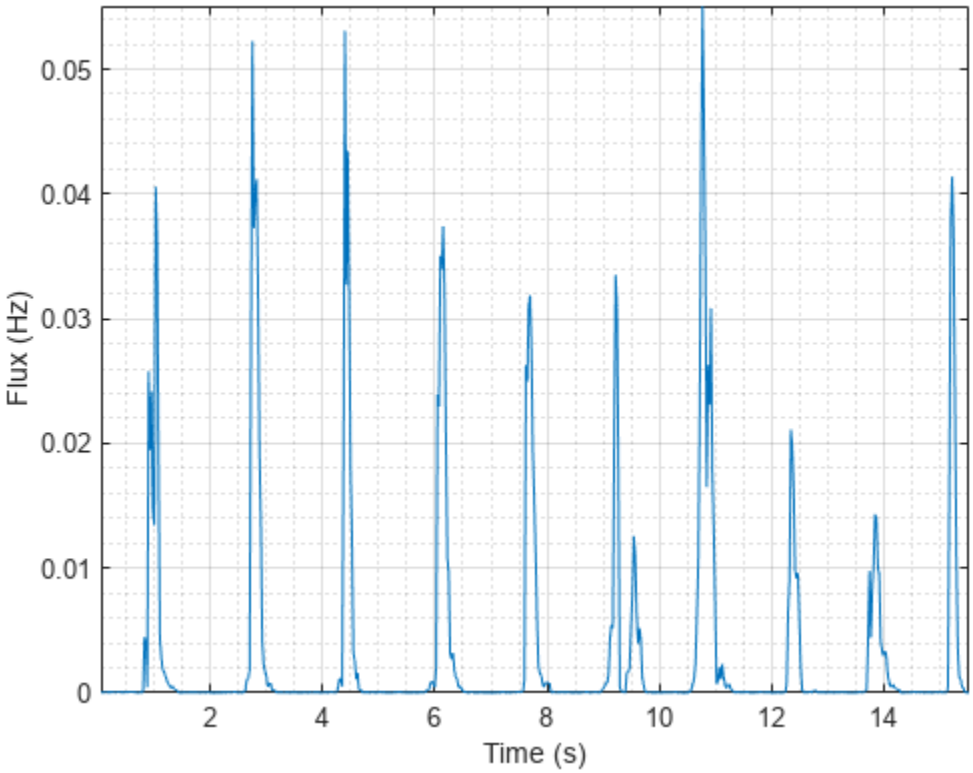
```
[audioIn,fs] = audioread("Counting-16-44p1-mono-15secs.wav");
```

Calculate the flux of the power spectrum over time. Calculate the flux for 50 ms Hamming windows of data with 25 ms overlap. Use the range from 62.5 Hz to `fs`/2 for the flux calculation.

```
flux = spectralFlux(audioIn,fs, ...
                    Window=hamming(round(0.05*fs)), ...
                    OverlapLength=round(0.025*fs), ...
                    Range=[62.5,fs/2]);
```

Plot the spectral flux.

```
spectralFlux(audioIn,fs, ...
             Window=hamming(round(0.05*fs)), ...
             OverlapLength=round(0.025*fs), ...
             Range=[62.5,fs/2])
```



Calculate Spectral Flux of Streaming Audio

Spectral flux measures the change in consecutive spectra. To calculate spectral flux of streaming audio, you can pass the state in and out of the function.

Create a `dsp.AudioFileReader` object to read in audio data frame-by-frame. Create a `dsp.AsyncBuffer` object to buffer the audio input into overlapped frames. Create a second `dsp.AsyncBuffer` object to log the spectral flux calculation.

```
fileReader = dsp.AudioFileReader('Counting-16-44p1-mono-15secs.wav');
inputBuffer = dsp.AsyncBuffer;
logger = dsp.AsyncBuffer;
```

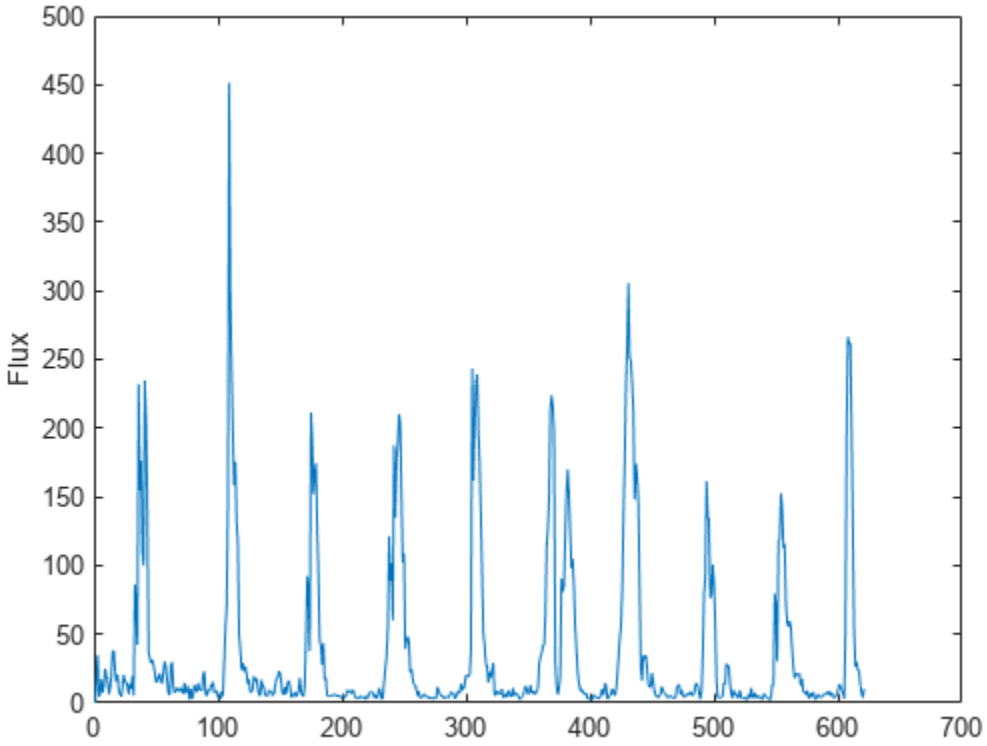
In an audio stream loop:

1. Read in a frame of audio data from your source.
2. Write the audio data to the input buffer.
3. If a hop of data is available from the buffer, read a frame of data with overlap.
4. Calculate the one-sided magnitude short time Fourier transform.
5. Calculate the spectral flux.
6. Log the spectral flux for later plotting.

```
fs = fileReader.SampleRate;

samplesPerFrame = round(fs*0.05);
samplesOverlap = round(fs*0.025);

samplesPerHop = samplesPerFrame - samplesOverlap;

win = hamming(samplesPerFrame,'periodic');

Sprev = [];
while ~isDone(fileReader)
    audioIn = fileReader();
    write(inputBuffer,audioIn);

    while inputBuffer.NumUnreadSamples >= samplesPerHop
        audioBuffered = read(inputBuffer,samplesPerFrame,samplesOverlap);
        [S,f] = stft(audioBuffered,fs,"Window",win,"OverlapLength",samplesOverlap,"FrequencyRange
        [flux,Sprev] = spectralFlux(abs(S),f,Sprev);
        write(logger,flux);
    end

end
release(fileReader)
```

Plot the logged data.

```
plot(read(logger))
ylabel('Flux')
```



Input Arguments

x — Input signal

column vector | matrix | 3-D array

Input signal, specified as a vector, matrix, or 3-D array. How the function interprets `x` depends on the shape of `f`.

Data Types: `single` | `double`

f — Sample rate or frequency vector (Hz)

scalar | vector

Sample rate or frequency vector in Hz, specified as a scalar or vector, respectively. How the function interprets `x` depends on the shape of `f`:

- If `f` is a scalar, `x` is interpreted as a time-domain signal, and `f` is interpreted as the sample rate. In this case, `x` must be a real vector or matrix. If `x` is specified as a matrix, the columns are interpreted as individual channels.
- If `f` is a vector, `x` is interpreted as a frequency-domain signal, and `f` is interpreted as the frequencies, in Hz, corresponding to the rows of `x`. In this case, `x` must be a real *L*-by-*M*-by-*N* array, where *L* is the number of spectral values at given frequencies of `f`, *M* is the number of individual spectra, and *N* is the number of channels.

Data Types: `single` | `double`

initialCondition — Previous spectral state

`[]` (default) | matrix

Previous spectral state, specified as an *L*-by-*N* matrix, where:

- *L* is the number of bins in the one-sided spectral representation, equal to `numel(f)`.
- *N* is the number of channels of audio data, equal to `size(x,3)`.

If `initialCondition` is unspecified, or specified as an empty, `spectralFlux` considers the first spectrum as repeating. That is, the first `flux` output is zero.

Dependencies

This input argument is only valid if the input, `x`, is a frequency-domain representation of audio. The `spectralFlux` function interprets the domain of the input `x` based on the size of `f`.

Data Types: `single` | `double`

Name-Value Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `Window=hamming(256)`

`NormType` — Norm type
2 (default) | 1

Norm type used to calculate flux, specified as 2 or 1.

Data Types: `single` | `double`

Note The following name-value arguments apply if `x` is a time-domain signal. If `x` is a frequency-domain signal, the following name-value arguments are ignored.

`Window` — Window applied in time domain
`rectwin(round(f*0.03))` (default) | vector

Window applied in the time domain, specified as a real vector. The number of elements in the vector must be in the range [1, `size(x,1)`]. The number of elements in the vector must also be greater than `OverlapLength`.

Data Types: `single` | `double`

`OverlapLength` — Number of samples overlapped between adjacent windows
`round(f*0.02)` (default) | non-negative scalar

Number of samples overlapped between adjacent windows, specified as an integer in the range [0, `size(Window,1)`).

Data Types: `single` | `double`

`FFTLength` — Number of bins in DFT
`numel(Window)` (default) | positive scalar integer

Number of bins used to calculate the DFT of windowed input samples, specified as a positive scalar integer. If unspecified, `FFTLength` defaults to the number of elements in the `Window`.

Data Types: `single` | `double`

`Range` — Frequency range (Hz)
`[0,f/2]` (default) | two-element row vector

Frequency range in Hz, specified as a two-element row vector of increasing real values in the range [0, `f`/2].

Data Types: `single` | `double`

`SpectrumType` — Spectrum type
`"power"` (default) | `"magnitude"`

Spectrum type, specified as `"power"` or `"magnitude"`:

- `"power"` –– The spectral flux is calculated for the one-sided power spectrum.
- `"magnitude"` –– The spectral flux is calculated for the one-sided magnitude spectrum.

Data Types: `char` | `string`

Output Arguments

`flux` — Spectral flux (Hz)

scalar | vector | matrix

Spectral flux in Hz, returned as a scalar, vector, or matrix. Each row of `flux` corresponds to the spectral flux of a window of `x`. Each column of `flux` corresponds to an independent channel.

`finalCondition` — Final spectral state

matrix

Final spectral state, returned as an *L*-by-*N* matrix, where:

- *L* is the number of bins in the one-sided spectral representation, equal to `numel(f)`.
- *N* is the number of channels of audio data, equal to `size(x,3)`.

Dependencies

This output argument is only valid if the input, `x`, is a frequency-domain representation of audio. The `spectralFlux` function interprets the domain of the input `x` based on the size of `f`.

Algorithms

The spectral flux is calculated as described in [1]:

$$\text{flux}(t) = \left(\sum_{k=b_1}^{b_2} |s_k(t) - s_k(t-1)|^P \right)^{1/P}$$

where

- s_k is the spectral value at bin k.
- b_1 and b_2 are the band edges, in bins, over which to calculate the spectral flux.
- P is the norm type. You can specify the norm type using `NormType`.

Version History

Introduced in R2019a

References

[1] Scheirer, E., and M. Slaney. "Construction and Evaluation of a Robust Multifeature Speech/Music Discriminator." *IEEE International Conference on Acoustics, Speech, and Signal Processing*. Volume 2, 1997, pp. 1221–1224.

Extended Capabilities

C/C++ Code Generation

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

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

See Also

`spectralCentroid` | `integratedLoudness` | `splMeter` | `acousticFluctuation`

Topics

“Spectral Descriptors”

spectralFlatness

Spectral flatness for audio signals and auditory spectrograms

Syntax

```
flatness = spectralFlatness(x,f)
flatness = spectralFlatness(x,f,Name=Value)
[flatness,arithmeticMean,geometricMean] = spectralFlatness( ___ )
spectralFlatness( ___ )
```

Description

`flatness = spectralFlatness(x,f)` returns the spectral flatness of the signal, `x`, over time. How the function interprets `x` depends on the shape of `f`.

`flatness = spectralFlatness(x,f,Name=Value)` specifies options using one or more name-value arguments.

`[flatness,arithmeticMean,geometricMean] = spectralFlatness( ___ )` returns the spectral arithmetic mean and spectral geometric mean. You can specify an input combination from any of the previous syntaxes.

`spectralFlatness( ___ )` with no output arguments plots the spectral flatness.

- If the input is in the time domain, the spectral flatness is plotted against time.
- If the input is in the frequency domain, the spectral flatness is plotted against frame number.

Examples

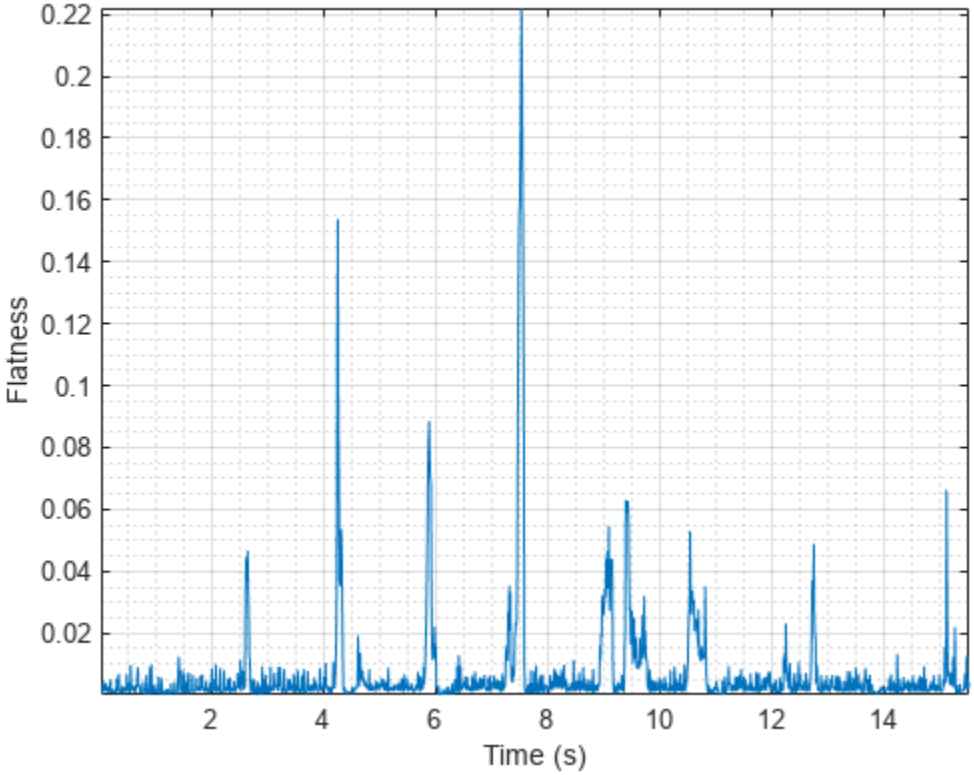
Spectral Flatness of Time-Domain Audio

Read in an audio file and calculate the flatness using default parameters.

```
[audioIn,fs] = audioread("Counting-16-44p1-mono-15secs.wav");
flatness = spectralFlatness(audioIn,fs);
```

Plot the spectral flatness against time.

```
spectralFlatness(audioIn,fs)
```



Spectral Flatness of Frequency-Domain Audio Data

Read in an audio file and then calculate the mel spectrogram using the `melSpectrogram` function.

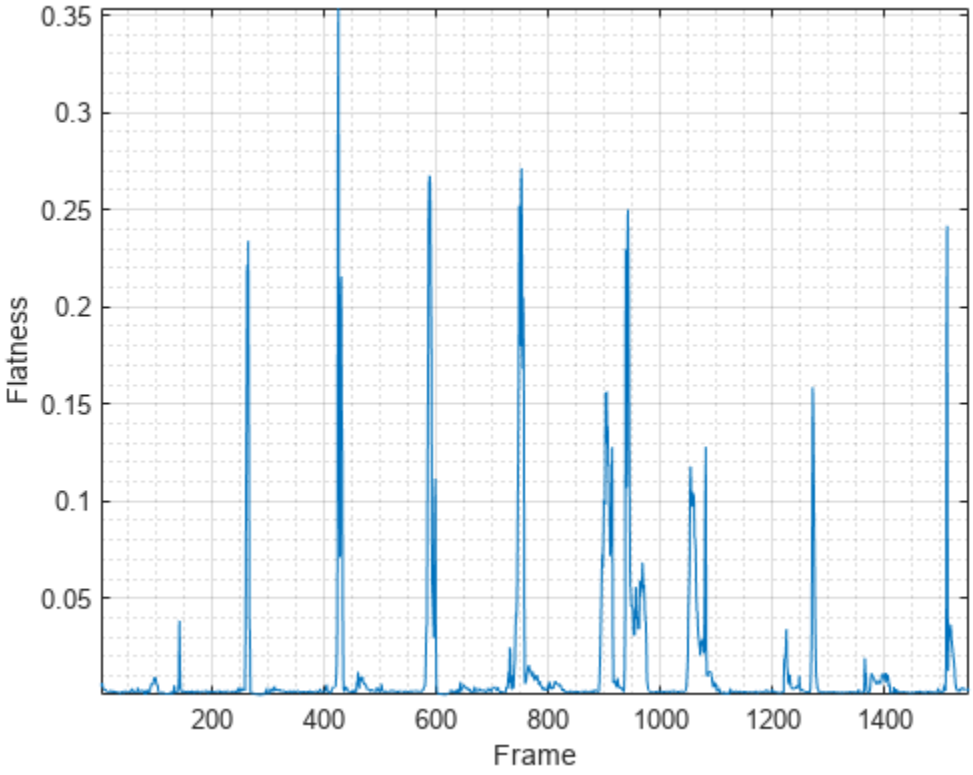
```
[audioIn,fs] = audioread("Counting-16-44p1-mono-15secs.wav");
[s,cf,t] = melSpectrogram(audioIn,fs);
```

Calculate the flatness of the mel spectrogram over time.

```
flatness = spectralFlatness(s,cf);
```

Plot the spectral flatness against the frame number.

```
spectralFlatness(s,cf)
```



Specify Nondefault Parameters

Read in an audio file.

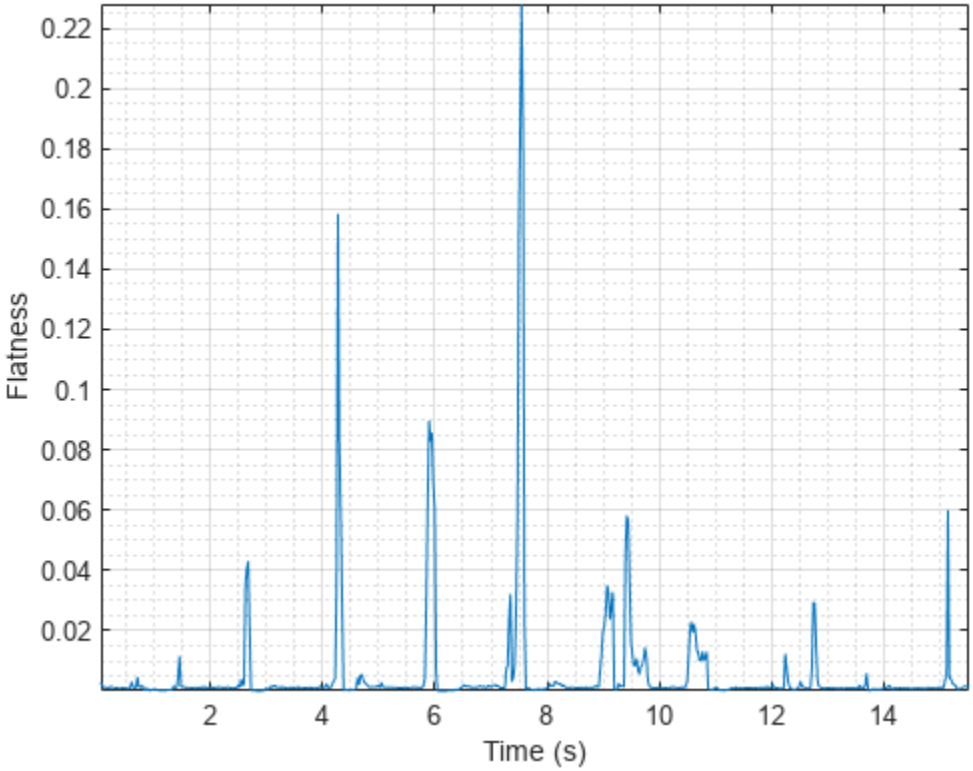
```
[audioIn,fs] = audioread("Counting-16-44p1-mono-15secs.wav");
```

Calculate the flatness of the power spectrum over time. Calculate the flatness for 50 ms Hamming windows of data with 25 ms overlap. Use the range from 62.5 Hz to `fs`/2 for the flatness calculation.

```
flatness = spectralFlatness(audioIn,fs, ...
                            Window=hamming(round(0.05*fs)), ...
                            OverlapLength=round(0.025*fs), ...
                            Range=[62.5,fs/2]);
```

Plot the spectral flatness.

```
spectralFlatness(audioIn,fs, ...
                 Window=hamming(round(0.05*fs)), ...
                 OverlapLength=round(0.025*fs), ...
                 Range=[62.5,fs/2]);
```



Calculate Spectral Flatness of Streaming Audio

Create a `dsp.AudioFileReader` object to read in audio data frame-by-frame. Create a `dsp.SignalSink` to log the spectral flatness calculation.

```
fileReader = dsp.AudioFileReader('Counting-16-44p1-mono-15secs.wav');
logger = dsp.SignalSink;
```

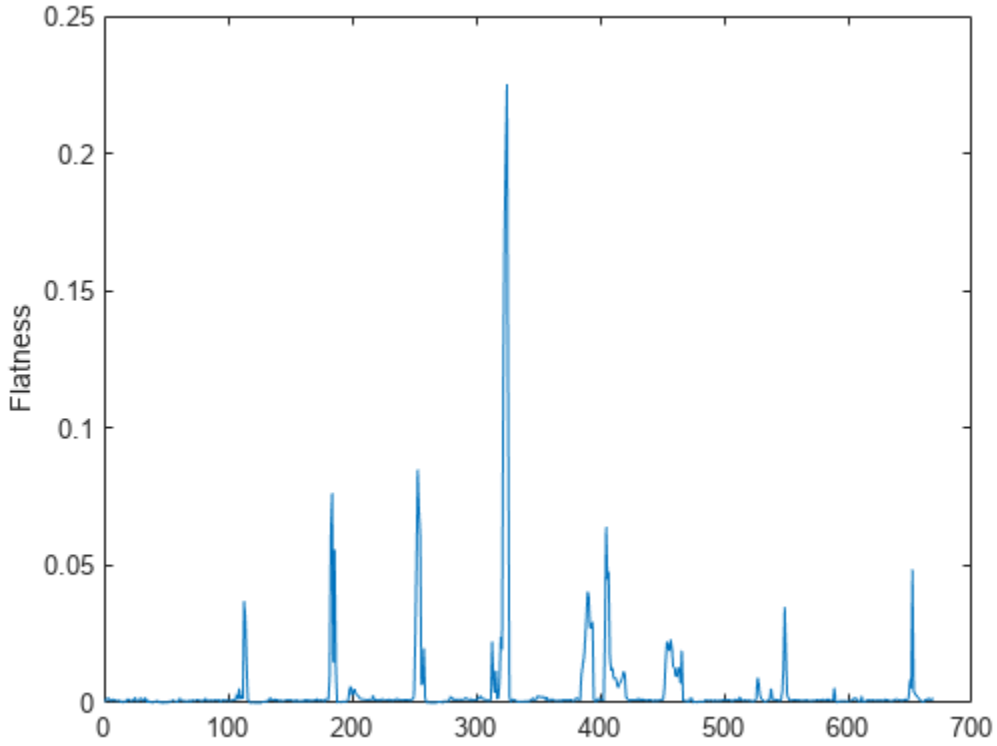
In an audio stream loop:

1. Read in a frame of audio data.
2. Calculate the spectral flatness for the frame of audio.
3. Log the spectral flatness for later plotting.

To calculate the spectral flatness for only a given input frame, specify a window with the same number of samples as the input, and set the overlap length to zero. Plot the logged data.

```
win = hamming(fileReader.SamplesPerFrame);
while ~isDone(fileReader)
    audioIn = fileReader();
    flatness = spectralFlatness(audioIn,fileReader.SampleRate, ...
                                'Window',win, ...
                                'OverlapLength',0);
    logger(flatness)
```

```
end

plot(logger.Buffer)
ylabel('Flatness')
```



Use `dsp.AsyncBuffer` if

- The input to your audio stream loop has a variable samples-per-frame.
- The input to your audio stream loop has an inconsistent samples-per-frame with the analysis window of `spectralFlatness`.
- You want to calculate the spectral flatness for overlapped data.

Create a `dsp.AsyncBuffer` object, reset the logger, and release the file reader.

```
buff = dsp.AsyncBuffer;
reset(logger)
release(fileReader)
```

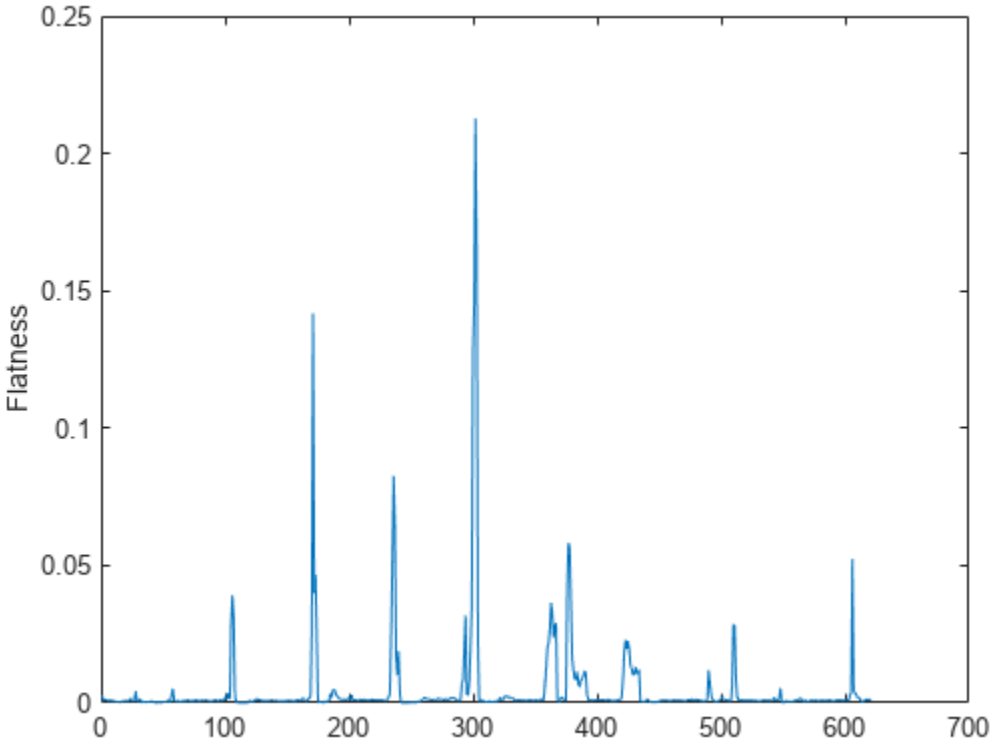
Specify that the spectral flatness is calculated for 50 ms frames with a 25 ms overlap.

```
fs = fileReader.SampleRate;

samplesPerFrame = round(fs*0.05);
samplesOverlap = round(fs*0.025);

samplesPerHop = samplesPerFrame - samplesOverlap;
```

```
win = hamming(samplesPerFrame);

while ~isDone(fileReader)
    audioIn = fileReader();
    write(buff,audioIn);

    while buff.NumUnreadSamples >= samplesPerHop
        audioBuffered = read(buff,samplesPerFrame,samplesOverlap);

        flatness = spectralFlatness(audioBuffered,fs, ...
                                    'Window',win, ...
                                    'OverlapLength',0);
        logger(flatness)
    end

end
release(fileReader)
```

Plot the logged data.

```
plot(logger.Buffer)
ylabel('Flatness')
```



Input Arguments

x — Input signal

column vector | matrix | 3-D array

Input signal, specified as a vector, matrix, or 3-D array. How the function interprets `x` depends on the shape of `f`.

Data Types: `single` | `double`

f — Sample rate or frequency vector (Hz)

scalar | vector

Sample rate or frequency vector in Hz, specified as a scalar or vector, respectively. How the function interprets `x` depends on the shape of `f`:

- If `f` is a scalar, `x` is interpreted as a time-domain signal, and `f` is interpreted as the sample rate. In this case, `x` must be a real vector or matrix. If `x` is specified as a matrix, the columns are interpreted as individual channels.
- If `f` is a vector, `x` is interpreted as a frequency-domain signal, and `f` is interpreted as the frequencies, in Hz, corresponding to the rows of `x`. In this case, `x` must be a real *L*-by-*M*-by-*N* array, where *L* is the number of spectral values at given frequencies of `f`, *M* is the number of individual spectra, and *N* is the number of channels.
- The number of rows of `x`, *L*, must be equal to the number of elements of `f`.

Data Types: `single` | `double`

Name-Value Arguments

Note The following name-value arguments apply if `x` is a time-domain signal. If `x` is a frequency-domain signal, name-value arguments are ignored.

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `Window=hamming(256)`

Window — Window applied in time domain

`rectwin(round(f*0.03))` (default) | vector

Window applied in the time domain, specified as a real vector. The number of elements in the vector must be in the range [1, `size(x,1)`]. The number of elements in the vector must also be greater than `OverlapLength`.

Data Types: `single` | `double`

OverlapLength — Number of samples overlapped between adjacent windows

`round(f*0.02)` (default) | non-negative scalar

Number of samples overlapped between adjacent windows, specified as an integer in the range [0, `size(Window,1)`).

Data Types: `single` | `double`

`FFTLength` — Number of bins in DFT
`numel(Window)` (default) | positive scalar integer

Number of bins used to calculate the DFT of windowed input samples, specified as a positive scalar integer. If unspecified, `FFTLength` defaults to the number of elements in the `Window`.

Data Types: `single` | `double`

`Range` — Frequency range (Hz)
`[0,f/2]` (default) | two-element row vector

Frequency range in Hz, specified as a two-element row vector of increasing real values in the range [0, `f`/2].

Data Types: `single` | `double`

`SpectrumType` — Spectrum type
`"power"` (default) | `"magnitude"`

Spectrum type, specified as `"power"` or `"magnitude"`:

- `"power"` –- The spectral flatness is calculated for the one-sided power spectrum.
- `"magnitude"` –- The spectral flatness is calculated for the one-sided magnitude spectrum.

Data Types: `char` | `string`

Output Arguments

`flatness` — Spectral flatness
scalar | vector | matrix

Spectral flatness, returned as a scalar, vector, or matrix. Each row of `flatness` corresponds to the spectral flatness of a window of `x`. Each column of `flatness` corresponds to an independent channel.

`arithmeticMean` — Spectral arithmetic mean
scalar | vector | matrix

Spectral arithmetic mean, returned as a scalar, vector, or matrix. Each row of `arithmeticMean` corresponds to the arithmetic mean of the spectrum of a window of `x`. Each column of `arithmeticMean` corresponds to an independent channel.

`geometricMean` — Spectral geometric mean
scalar | vector | matrix

Spectral geometric mean, returned as a scalar, vector, or matrix. Each row of `geometricMean` corresponds to the geometric mean of the spectrum of a window of `x`. Each column of `geometricMean` corresponds to an independent channel.

Algorithms

The spectral flatness is calculated as described in [1]:

$$\text{flatness} = \frac{\left(\prod_{k=b_1}^{b_2} s_k\right)^{\frac{1}{b_2 - b_1}}}{\frac{1}{b_2 - b_1}\sum_{k=b_1}^{b_2} s_k}$$

where

- s_k is the spectral value at bin k.
- b_1 and b_2 are the band edges, in bins, over which to calculate the spectral spread.

Version History

Introduced in R2019a

References

[1] Johnston, J. D. "Transform Coding of Audio Signals Using Perceptual Noise Criteria." *IEEE Journal on Selected Areas in Communications*. Vol. 6, Number 2, 1988, pp. 314–323.

Extended Capabilities

C/C++ Code Generation

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

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

See Also

`spectralCrest`

Topics

“Spectral Descriptors”

spectralEntropy

Spectral entropy for audio signals and auditory spectrograms

Syntax

```
entropy = spectralEntropy(x,f)
entropy = spectralEntropy(x,f,Name=Value)
spectralEntropy( ___ )
```

Description

`entropy = spectralEntropy(x,f)` returns the spectral entropy of the signal, `x`, over time. How the function interprets `x` depends on the shape of `f`.

`entropy = spectralEntropy(x,f,Name=Value)` specifies options using one or more name-value arguments.

`spectralEntropy( ___ )` with no output arguments plots the spectral entropy. You can specify an input combination from any of the previous syntaxes.

- If the input is in the time domain, the spectral entropy is plotted against time.
- If the input is in the frequency domain, the spectral entropy is plotted against frame number.

Examples

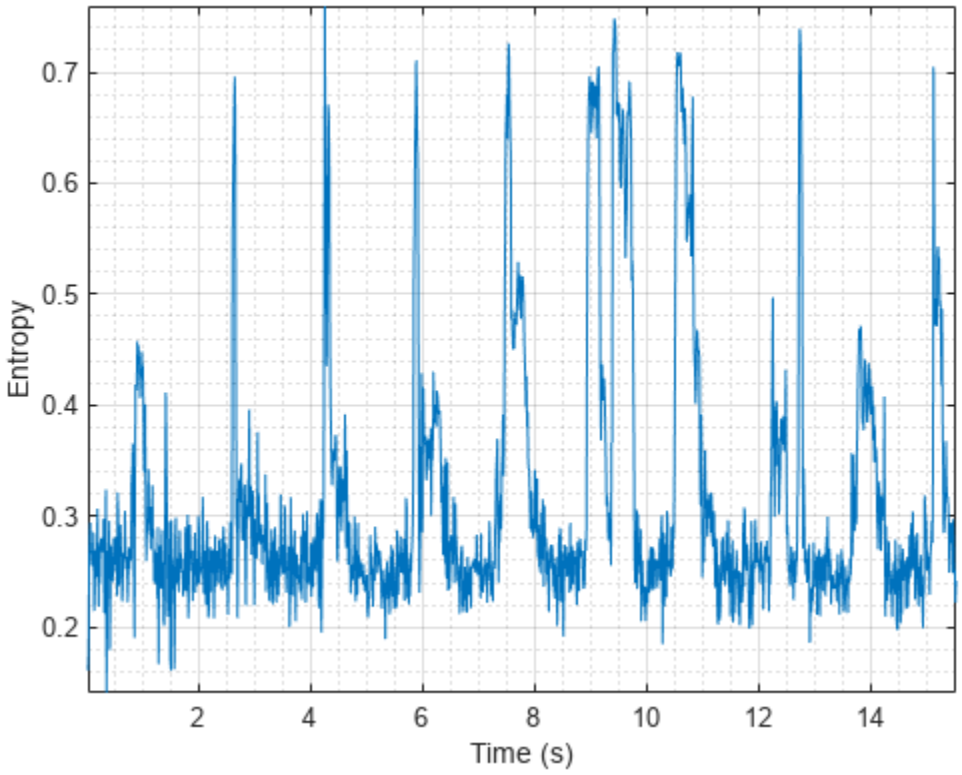
Spectral Entropy of Time-Domain Audio

Read in an audio file and calculate the entropy using default parameters.

```
[audioIn,fs] = audioread("Counting-16-44p1-mono-15secs.wav");
entropy = spectralEntropy(audioIn,fs);
```

Plot the spectral entropy against time

```
spectralEntropy(audioIn,fs)
```



Spectral Entropy of Frequency-Domain Audio Data

Read in an audio file and then calculate the mel spectrogram using the `melSpectrogram` function.

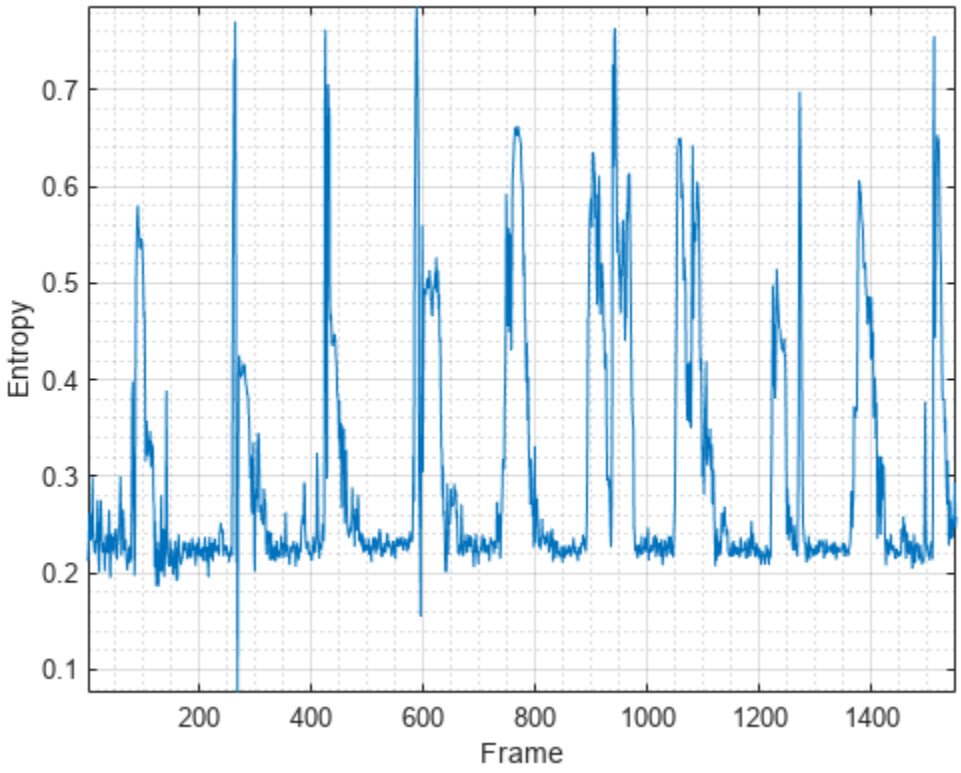
```
[audioIn,fs] = audioread("Counting-16-44p1-mono-15secs.wav");
[s,cf,t] = melSpectrogram(audioIn,fs);
```

Calculate the entropy of the mel spectrogram over time.

```
entropy = spectralEntropy(s,cf);
```

Plot the spectral entropy against the frame number.

```
spectralEntropy(s,cf)
```



Specify Nondefault Parameters

Read in an audio file.

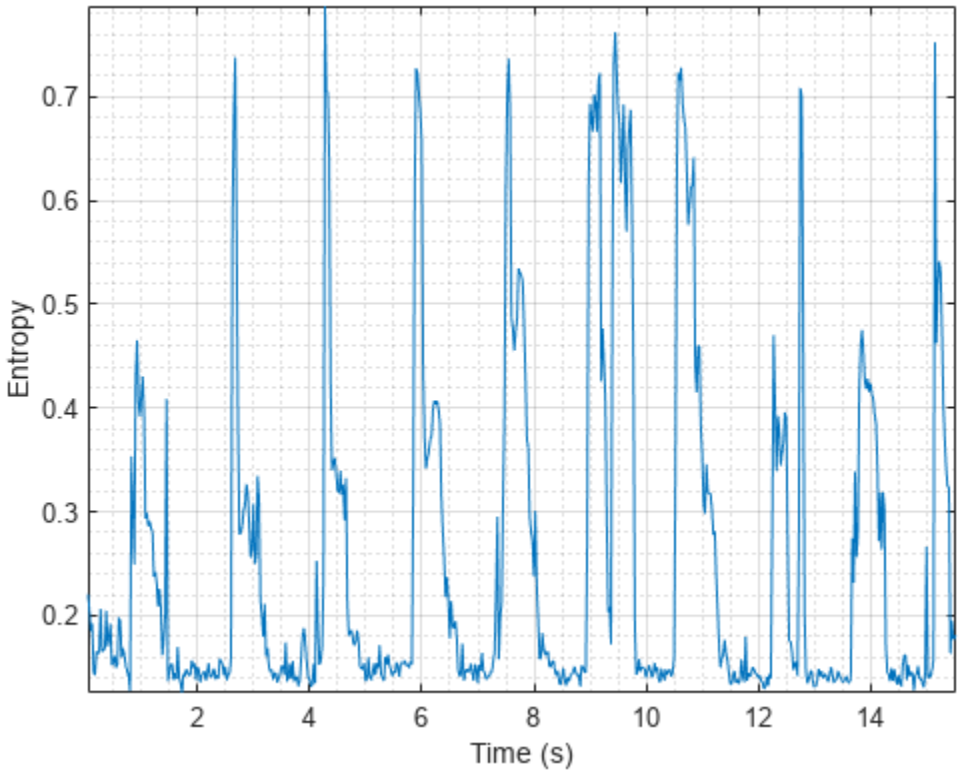
```
[audioIn,fs] = audioread("Counting-16-44p1-mono-15secs.wav");
```

Calculate the entropy of the power spectrum over time. Calculate the entropy for 50 ms Hamming windows of data with 25 ms overlap. Use the range from 62.5 Hz to `fs`/2 for the entropy calculation.

```
entropy = spectralEntropy(audioIn,fs, ...
                          Window=hamming(round(0.05*fs)), ...
                          OverlapLength=round(0.025*fs), ...
                          Range=[62.5,fs/2]);
```

Plot the spectral entropy against time.

```
spectralEntropy(audioIn,fs, ...
                Window=hamming(round(0.05*fs)), ...
                OverlapLength=round(0.025*fs), ...
                Range=[62.5,fs/2])
```



Calculate Spectral Entropy of Streaming Audio

Create a `dsp.AudioFileReader` object to read in audio data frame-by-frame. Create a `dsp.SignalSink` to log the spectral entropy calculation.

```
fileReader = dsp.AudioFileReader('Counting-16-44p1-mono-15secs.wav');
logger = dsp.SignalSink;
```

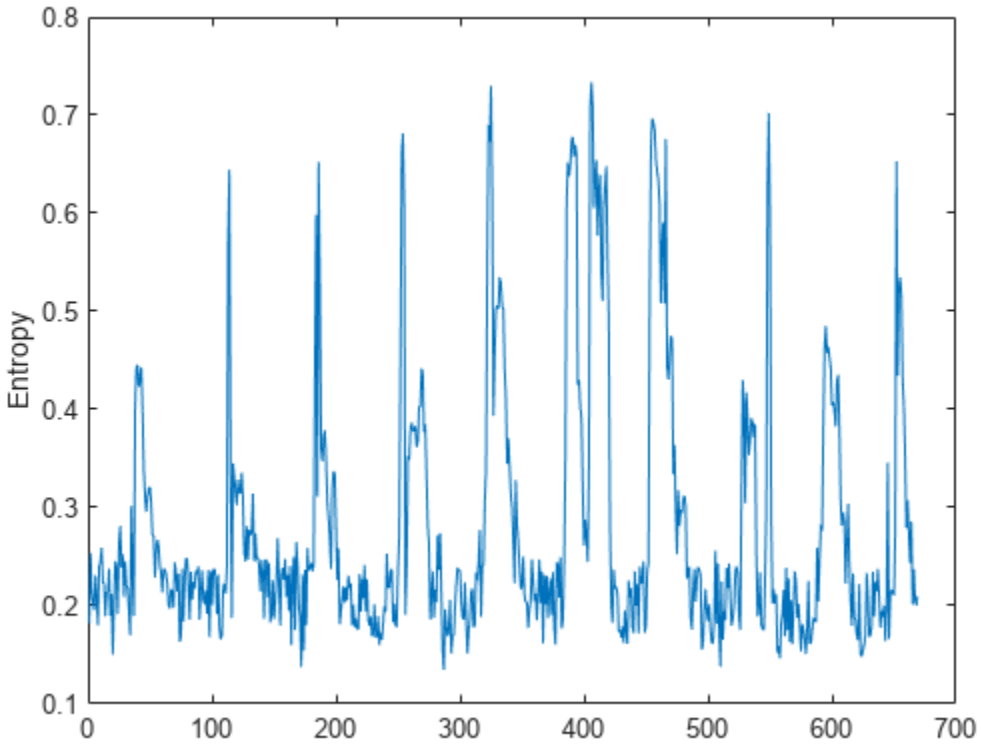
In an audio stream loop:

1. Read in a frame of audio data.
2. Calculate the spectral entropy for the frame of audio.
3. Log the spectral entropy for later plotting.

To calculate the spectral entropy for only a given input frame, specify a window with the same number of samples as the input, and set the overlap length to zero. Plot the logged data.

```
while ~isDone(fileReader)
    audioIn = fileReader();
    entropy = spectralEntropy(audioIn,fileReader.SampleRate, ...
                              'Window',hamming(size(audioIn,1)), ...
                              'OverlapLength',0);
    logger(entropy)
end
```

```
plot(logger.Buffer)
ylabel('Entropy')
```



Use `dsp.AsyncBuffer` if

- The input to your audio stream loop has a variable samples-per-frame.
- The input to your audio stream loop has an inconsistent samples-per-frame with the analysis window of `spectralEntropy`.
- You want to calculate the spectral entropy for overlapped data.

Create a `dsp.AsyncBuffer` object, reset the logger, and release the file reader.

```
buff = dsp.AsyncBuffer;
reset(logger)
release(fileReader)
```

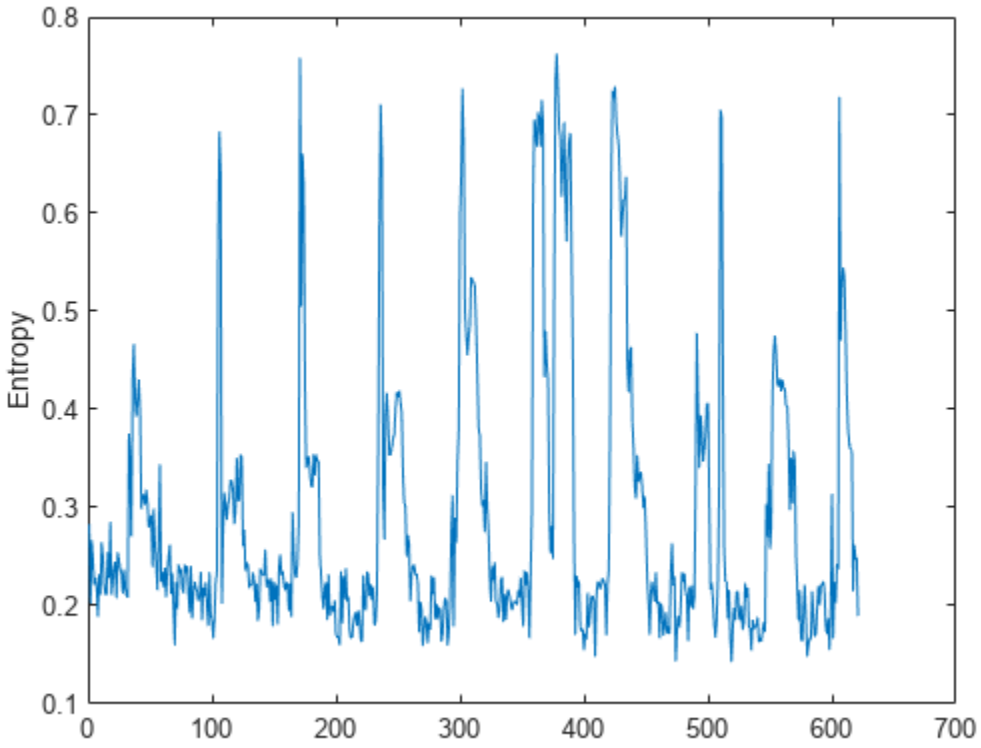
Specify that the spectral entropy is calculated for 50 ms frames with a 25 ms overlap.

```
fs = fileReader.SampleRate;

samplesPerFrame = round(fs*0.05);
samplesOverlap = round(fs*0.025);

samplesPerHop = samplesPerFrame - samplesOverlap;

win = hamming(samplesPerFrame);
```

```
while ~isDone(fileReader)
    audioIn = fileReader();
    write(buff,audioIn);

    while buff.NumUnreadSamples >= samplesPerHop
        audioBuffered = read(buff,samplesPerFrame,samplesOverlap);

        entropy = spectralEntropy(audioBuffered,fs, ...
                                  'Window',win, ...
                                  'OverlapLength',0);
        logger(entropy)
    end

end
release(fileReader)
```

Plot the logged data.

```
plot(logger.Buffer)
ylabel('Entropy')
```



Input Arguments

x — Input signal

column vector | matrix | 3-D array

Input signal, specified as a vector, matrix, or 3-D array. How the function interprets `x` depends on the shape of `f`.

Data Types: `single` | `double`

`f` — Sample rate or frequency vector (Hz)

scalar | vector

Sample rate or frequency vector in Hz, specified as a scalar or vector, respectively. How the function interprets `x` depends on the shape of `f`:

- If `f` is a scalar, `x` is interpreted as a time-domain signal, and `f` is interpreted as the sample rate. In this case, `x` must be a real vector or matrix. If `x` is specified as a matrix, the columns are interpreted as individual channels.
- If `f` is a vector, `x` is interpreted as a frequency-domain signal, and `f` is interpreted as the frequencies, in Hz, corresponding to the rows of `x`. In this case, `x` must be a real *L*-by-*M*-by-*N* array, where *L* is the number of spectral values at given frequencies of `f`, *M* is the number of individual spectra, and *N* is the number of channels.
- The number of rows of `x`, *L*, must be equal to the number of elements of `f`.

Data Types: `single` | `double`

Name-Value Arguments

Note The following name-value arguments apply if `x` is a time-domain signal. If `x` is a frequency-domain signal, name-value arguments are ignored.

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `Window=hamming(256)`

`Window` — Window applied in time domain

`rectwin(round(f*0.03))` (default) | vector

Window applied in the time domain, specified as a real vector. The number of elements in the vector must be in the range [1, `size(x,1)`]. The number of elements in the vector must also be greater than `OverlapLength`.

Data Types: `single` | `double`

`OverlapLength` — Number of samples overlapped between adjacent windows

`round(f*0.02)` (default) | non-negative scalar

Number of samples overlapped between adjacent windows, specified as an integer in the range [0, `size(Window,1)`).

Data Types: `single` | `double`

`FFTLength` — Number of bins in DFT

`numel(Window)` (default) | positive scalar integer

Number of bins used to calculate the DFT of windowed input samples, specified as a positive scalar integer. If unspecified, `FFTLength` defaults to the number of elements in the `Window`.

Data Types: `single` | `double`

Range — Frequency range (Hz)
`[0,f/2]` (default) | two-element row vector

Frequency range in Hz, specified as a two-element row vector of increasing real values in the range [0, `f`/2].

Data Types: `single` | `double`

`SpectrumType` — Spectrum type
`"power"` (default) | `"magnitude"`

Spectrum type, specified as `"power"` or `"magnitude"`:

- `"power"` –- The spectral entropy is calculated for the one-sided power spectrum.
- `"magnitude"` –- The spectral entropy is calculated for the one-sided magnitude spectrum.

Data Types: `char` | `string`

Output Arguments

`entropy` — Spectral entropy
scalar | vector | matrix

Spectral entropy, returned as a scalar, vector, or matrix. Each row of `entropy` corresponds to the spectral entropy of a window of `x`. Each column of `entropy` corresponds to an independent channel.

Algorithms

The spectral entropy is calculated as described in [1]:

$$\text{entropy} = \frac{-\sum_{k=b_1}^{b_2} s_k \log(s_k)}{\log(b_2 - b_1)}$$

where

- s_k is the spectral value at bin k.
- b_1 and b_2 are the band edges, in bins, over which to calculate the spectral entropy.

Version History

Introduced in R2019a

References

[1] Misra, H., S. Ikbal, H. Bourlard, and H. Hermansky. "Spectral Entropy Based Feature for Robust ASR." *2004 IEEE International Conference on Acoustics, Speech, and Signal Processing*.

Extended Capabilities

C/C++ Code Generation

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

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

See Also

`spectralSpread` | `spectralSkewness` | `spectralKurtosis`

Topics

“Spectral Descriptors”

spectralDecrease

Spectral decrease for audio signals and auditory spectrograms

Syntax

```
decrease = spectralDecrease(x,f)
decrease = spectralDecrease(x,f,Name=Value)
spectralDecrease( ___ )
```

Description

`decrease = spectralDecrease(x,f)` returns the spectral decrease of the signal, `x`, over time. How the function interprets `x` depends on the shape of `f`.

`decrease = spectralDecrease(x,f,Name=Value)` specifies options using one or more name-value arguments.

`spectralDecrease( ___ )` with no output arguments plots the spectral decrease. You can specify an input combination from any of the previous syntaxes.

- If the input is in the time domain, the spectral decrease is plotted against time.
- If the input is in the frequency domain, the spectral decrease is plotted against frame number.

Examples

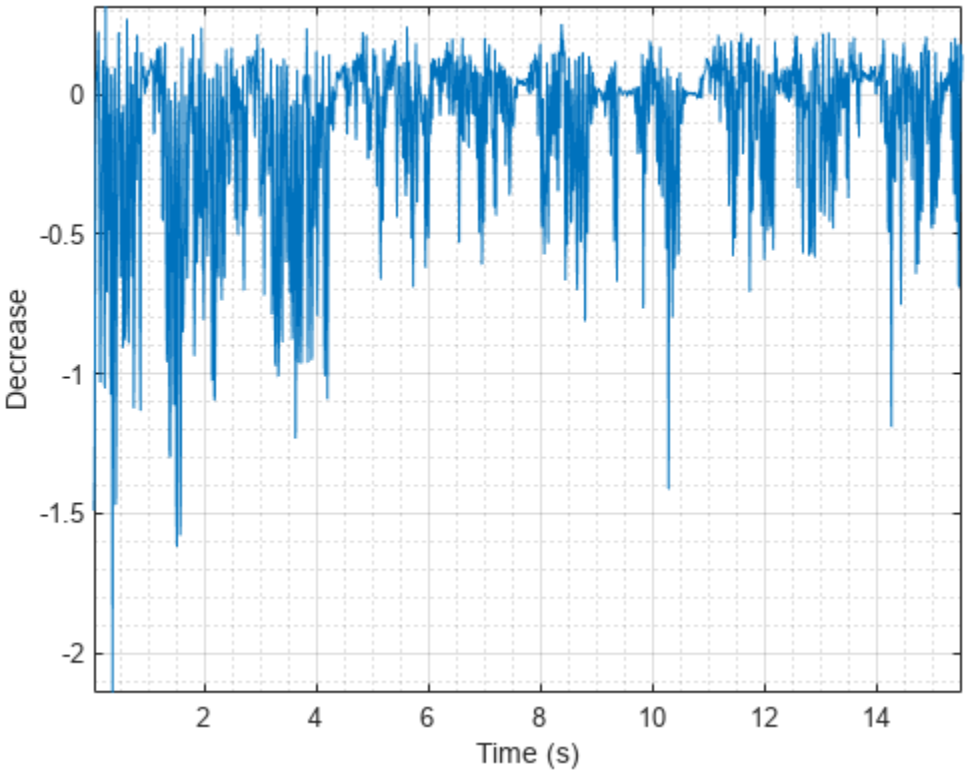
Spectral Decrease of Time-Domain Audio

Read in an audio file and calculate the decrease using default parameters.

```
[audioIn,fs] = audioread("Counting-16-44p1-mono-15secs.wav");
decrease = spectralDecrease(audioIn,fs);
```

Plot the spectral decrease against time.

```
spectralDecrease(audioIn,fs)
```



Spectral Decrease of Frequency-Domain Audio Data

Read in an audio file and then calculate the mel spectrogram using the `melSpectrogram` function.

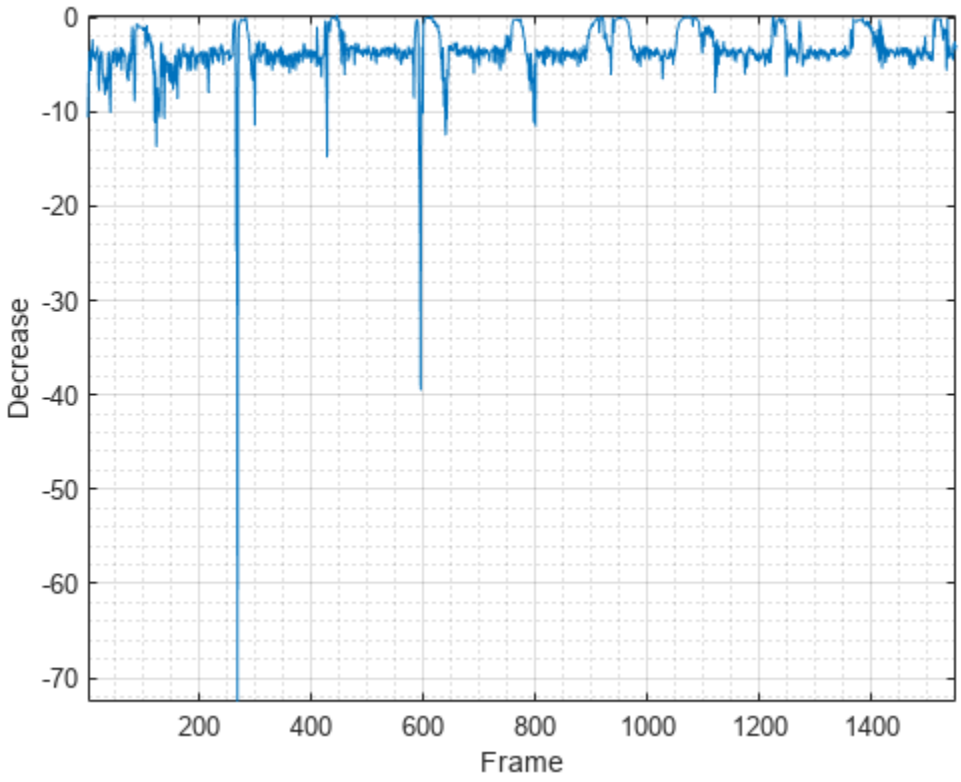
```
[audioIn,fs] = audioread("Counting-16-44p1-mono-15secs.wav");
[s,cf] = melSpectrogram(audioIn,fs);
```

Calculate the decrease of the mel spectrogram over time.

```
decrease = spectralDecrease(s,cf);
```

Plot the spectral decrease against the frame number.

```
spectralDecrease(s,cf)
```



Specify Nondefault Parameters

Read in an audio file.

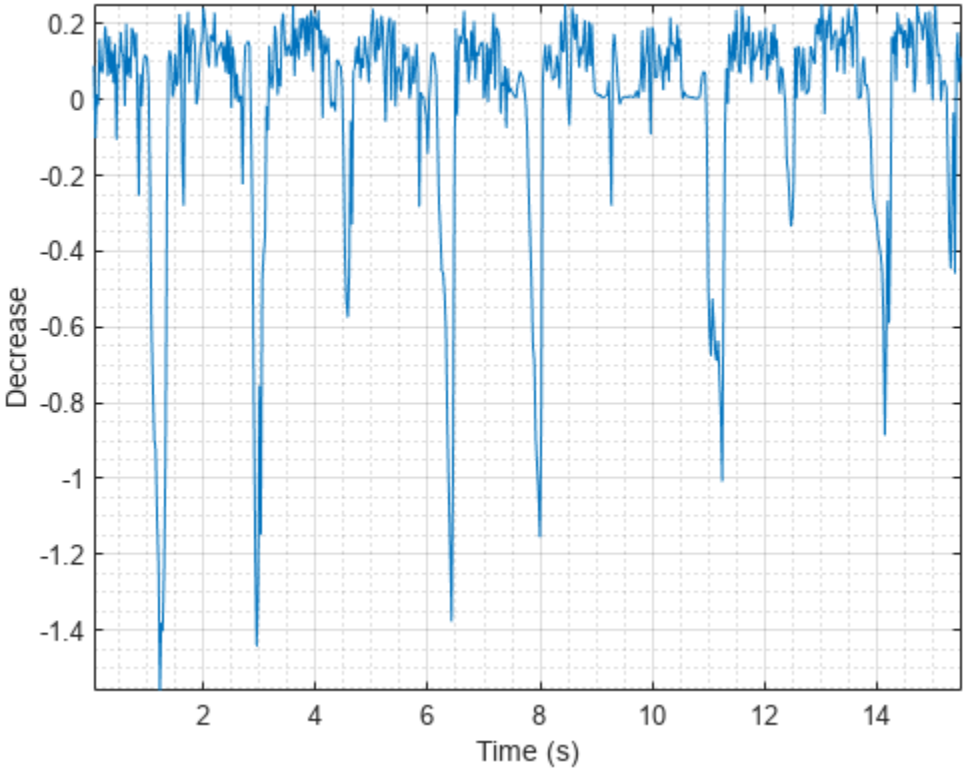
```
[audioIn,fs] = audioread("Counting-16-44p1-mono-15secs.wav");
```

Calculate the decrease of the magnitude spectrum over time. Calculate the decrease for 50 ms Hamming windows of data with 25 ms overlap. Use the range from 62.5 Hz to `fs`/2 for the decrease calculation.

```
decrease = spectralDecrease(audioIn,fs, ...
                            Window=hamming(round(0.05*fs)), ...
                            OverlapLength=round(0.025*fs), ...
                            Range=[62.5,fs/2]);
```

Plot the spectral decrease.

```
spectralDecrease(audioIn,fs, ...
                 Window=hamming(round(0.05*fs)), ...
                 OverlapLength=round(0.025*fs), ...
                 Range=[62.5,fs/2])
```



Calculate Spectral Decrease of Streaming Audio

Create a `dsp.AudioFileReader` object to read in audio data frame-by-frame. Create a `dsp.SignalSink` to log the spectral decrease calculation.

```
fileReader = dsp.AudioFileReader('Counting-16-44p1-mono-15secs.wav');
logger = dsp.SignalSink;
```

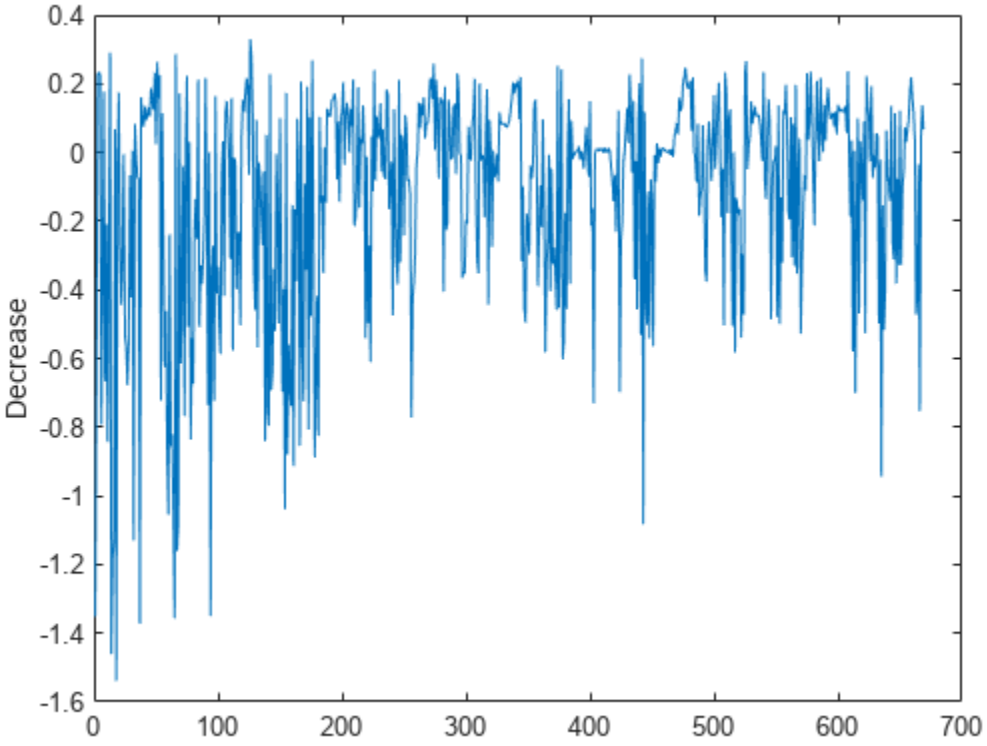
In an audio stream loop:

1. Read in a frame of audio data.
2. Calculate the spectral decrease for the frame of audio.
3. Log the spectral decrease for later plotting.

To calculate the spectral decrease for only a given input frame, specify a window with the same number of samples as the input, and set the overlap length to zero. Plot the logged data.

```
while ~isDone(fileReader)
    audioIn = fileReader();
    decrease = spectralDecrease(audioIn,fileReader.SampleRate, ...
                                'Window',hamming(size(audioIn,1)), ...
                                'OverlapLength',0);
    logger(decrease)
end
```

```
plot(logger.Buffer)
ylabel('Decrease')
```



Use `dsp.AsyncBuffer` if

- The input to your audio stream loop has a variable samples-per-frame.
- The input to your audio stream loop has an inconsistent samples-per-frame with the analysis window of `spectralDecrease`.
- You want to calculate the spectral decrease for overlapped data.

Create a `dsp.AsyncBuffer` object, reset the logger, and release the file reader.

```
buff = dsp.AsyncBuffer;
reset(logger)
release(fileReader)
```

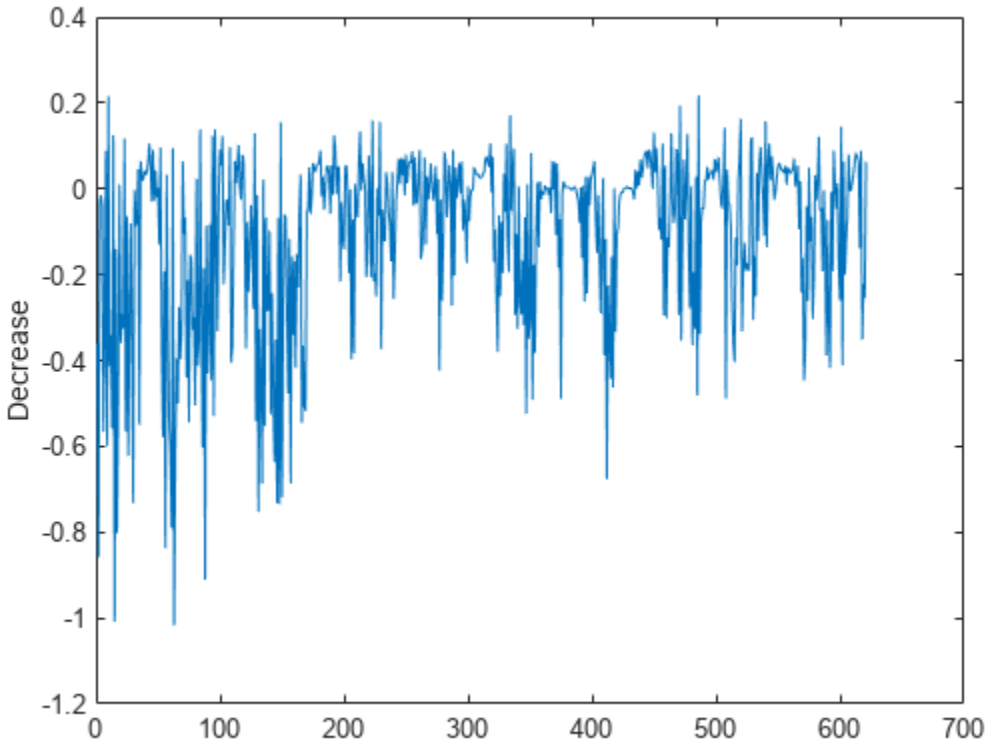
Specify that the spectral decrease is calculated for 50 ms frames with a 25 ms overlap.

```
fs = fileReader.SampleRate;

samplesPerFrame = round(fs*0.05);
samplesOverlap = round(fs*0.025);

samplesPerHop = samplesPerFrame - samplesOverlap;

win = hamming(samplesPerFrame);
```

```
while ~isDone(fileReader)
    audioIn = fileReader();
    write(buff,audioIn);

    while buff.NumUnreadSamples >= samplesPerHop
        audioBuffered = read(buff,samplesPerFrame,samplesOverlap);

        decrease = spectralDecrease(audioBuffered,fs, ...
                                    'Window',win, ...
                                    'OverlapLength',0);
        logger(decrease)
    end

end
release(fileReader)
```

Plot the logged data.

```
plot(logger.Buffer)
ylabel('Decrease')
```



Input Arguments

x — Input signal

column vector | matrix | 3-D array

Input signal, specified as a vector, matrix, or 3-D array. How the function interprets `x` depends on the shape of `f`.

Data Types: `single` | `double`

`f` — Sample rate or frequency vector (Hz)

scalar | vector

Sample rate or frequency vector in Hz, specified as a scalar or vector, respectively. How the function interprets `x` depends on the shape of `f`:

- If `f` is a scalar, `x` is interpreted as a time-domain signal, and `f` is interpreted as the sample rate. In this case, `x` must be a real vector or matrix. If `x` is specified as a matrix, the columns are interpreted as individual channels.
- If `f` is a vector, `x` is interpreted as a frequency-domain signal, and `f` is interpreted as the frequencies, in Hz, corresponding to the rows of `x`. In this case, `x` must be a real *L*-by-*M*-by-*N* array, where *L* is the number of spectral values at given frequencies of `f`, *M* is the number of individual spectra, and *N* is the number of channels.
- The number of rows of `x`, *L*, must be equal to the number of elements of `f`.

Data Types: `single` | `double`

Name-Value Arguments

Note The following name-value arguments apply if `x` is a time-domain signal. If `x` is a frequency-domain signal, name-value arguments are ignored.

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `Window=hamming(256)`

`Window` — Window applied in time domain

`rectwin(round(f*0.03))` (default) | vector

Window applied in the time domain, specified as a real vector. The number of elements in the vector must be in the range [1, `size(x,1)`]. The number of elements in the vector must also be greater than `OverlapLength`.

Data Types: `single` | `double`

`OverlapLength` — Number of samples overlapped between adjacent windows

`round(f*0.02)` (default) | non-negative scalar

Number of samples overlapped between adjacent windows, specified as an integer in the range [0, `size(Window,1)`).

Data Types: `single` | `double`

`FFTLength` — Number of bins in DFT

`numel(Window)` (default) | positive scalar integer

Number of bins used to calculate the DFT of windowed input samples, specified as a positive scalar integer. If unspecified, `FFTLength` defaults to the number of elements in the `Window`.

Data Types: `single` | `double`

Range — Frequency range (Hz)
`[0,f/2]` (default) | two-element row vector

Frequency range in Hz, specified as a two-element row vector of increasing real values in the range [0, `f`/2].

Data Types: `single` | `double`

SpectrumType — Spectrum type
`"magnitude"` (default) | `"power"`

Spectrum type, specified as `"power"` or `"magnitude"`:

- `"power"` –- The spectral decrease is calculated for the one-sided power spectrum.
- `"magnitude"` –- The spectral decrease is calculated for the one-sided magnitude spectrum.

Data Types: `char` | `string`

Output Arguments

decrease — Spectral decrease
scalar | vector | matrix

Spectral decrease in Hz, returned as a scalar, vector, or matrix. Each row of `decrease` corresponds to the spectral centroid of a window of `x`. Each column of `decrease` corresponds to an independent channel.

Algorithms

The spectral decrease is calculated as described in [1]:

$$\text{decrease} = \frac{\sum_{k=b_1+1}^{b_2} \frac{s_k - s_{b_1}}{k-1}}{\sum_{k=b_1+1}^{b_2} s_k}$$

where

- s_k is the spectral value at bin k.
- b_1 and b_2 are the band edges, in bins, over which to calculate the spectral decrease.

Version History

Introduced in R2019a

References

[1] Peeters, G. "A Large Set of Audio Features for Sound Description (Similarity and Classification) in the CUIDADO Project." Technical Report; IRCAM: Paris, France, 2004.

Extended Capabilities

C/C++ Code Generation

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

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

See Also

`spectralCrest` | `spectralSlope`

Topics

“Spectral Descriptors”

spectralCrest

Spectral crest for audio signals and auditory spectrograms

Syntax

```
crest = spectralCrest(x,f)
crest = spectralCrest(x,f,Name=Value)
[crest,spectralPeak,spectralMean] = spectralCrest( ___ )
spectralCrest( ___ )
```

Description

`crest = spectralCrest(x,f)` returns the spectral crest of the signal, `x`, over time. How the function interprets `x` depends on the shape of `f`.

`crest = spectralCrest(x,f,Name=Value)` specifies options using one or more name-value arguments.

`[crest,spectralPeak,spectralMean] = spectralCrest( ___ )` returns the spectral peak and spectral mean. You can specify an input combination from any of the previous syntaxes.

`spectralCrest( ___ )` with no output arguments plots the spectral crest.

- If the input is in the time domain, the spectral crest is plotted against time.
- If the input is in the frequency domain, the spectral crest is plotted against frame number.

Examples

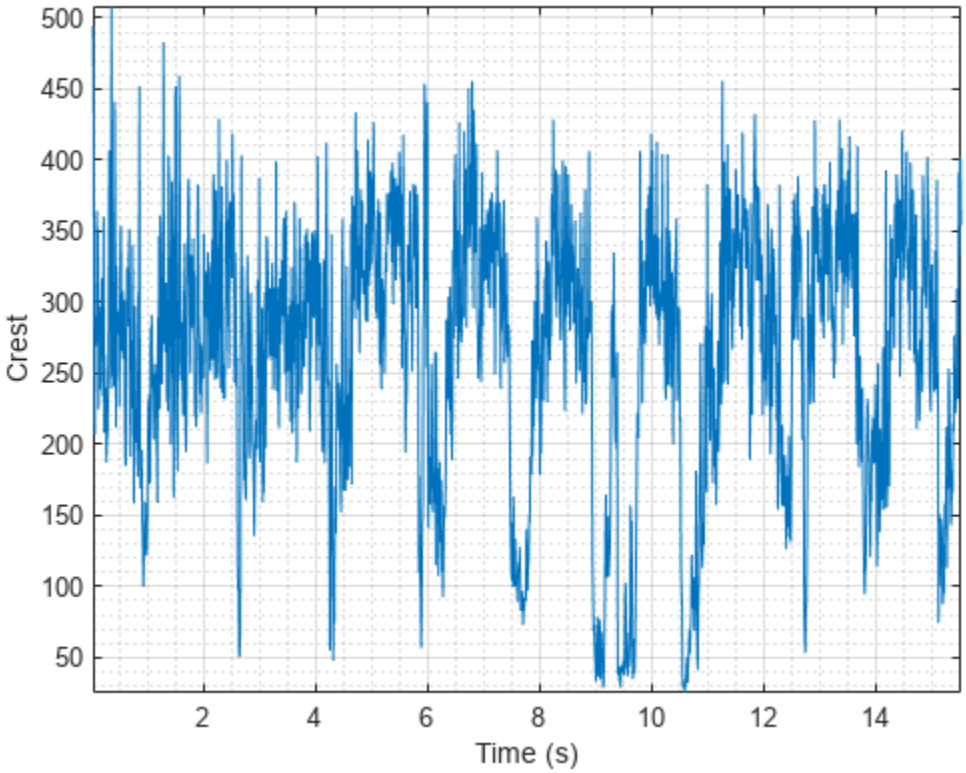
Spectral Crest of Time-Domain Audio

Read in an audio file and calculate the crest using default parameters.

```
[audioIn,fs] = audioread("Counting-16-44p1-mono-15secs.wav");
crest = spectralCrest(audioIn,fs);
```

Plot the spectral crest against time.

```
spectralCrest(audioIn,fs)
```



Spectral Crest of Frequency-Domain Audio Data

Read in an audio file and then calculate the mel spectrogram using the `melSpectrogram` function.

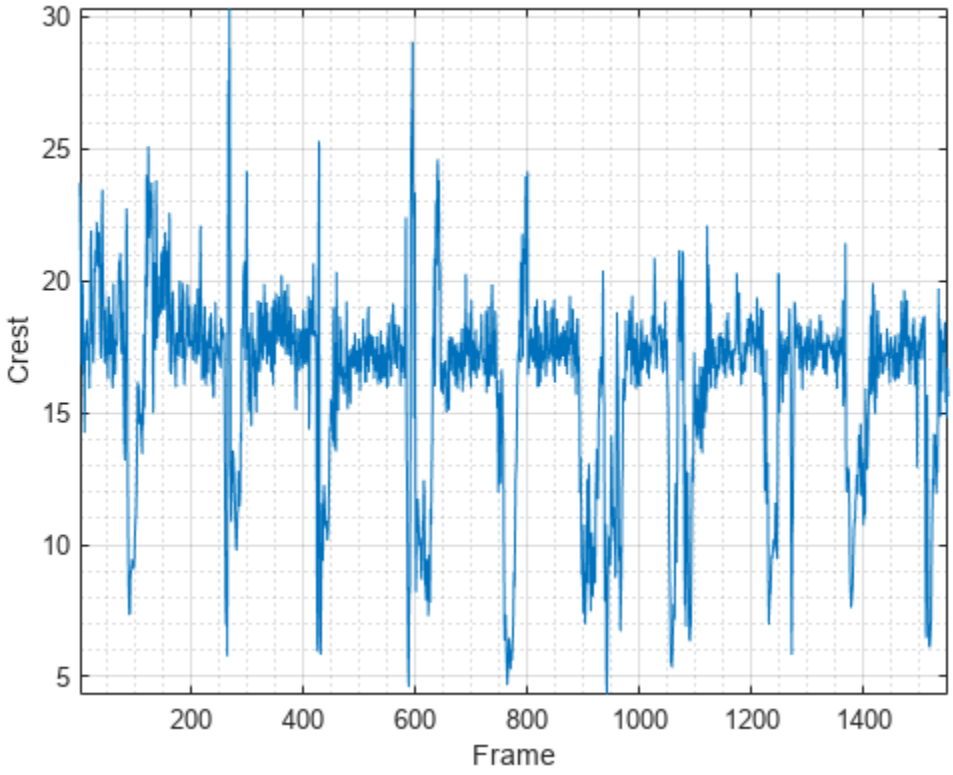
```
[audioIn,fs] = audioread("Counting-16-44p1-mono-15secs.wav");
[s,cf] = melSpectrogram(audioIn,fs);
```

Calculate the crest of the mel spectrogram over time.

```
crest = spectralCrest(s,cf);
```

Plot the spectral crest against the frame number.

```
spectralCrest(s,cf)
```



Specify Nondefault Parameters

Read in an audio file.

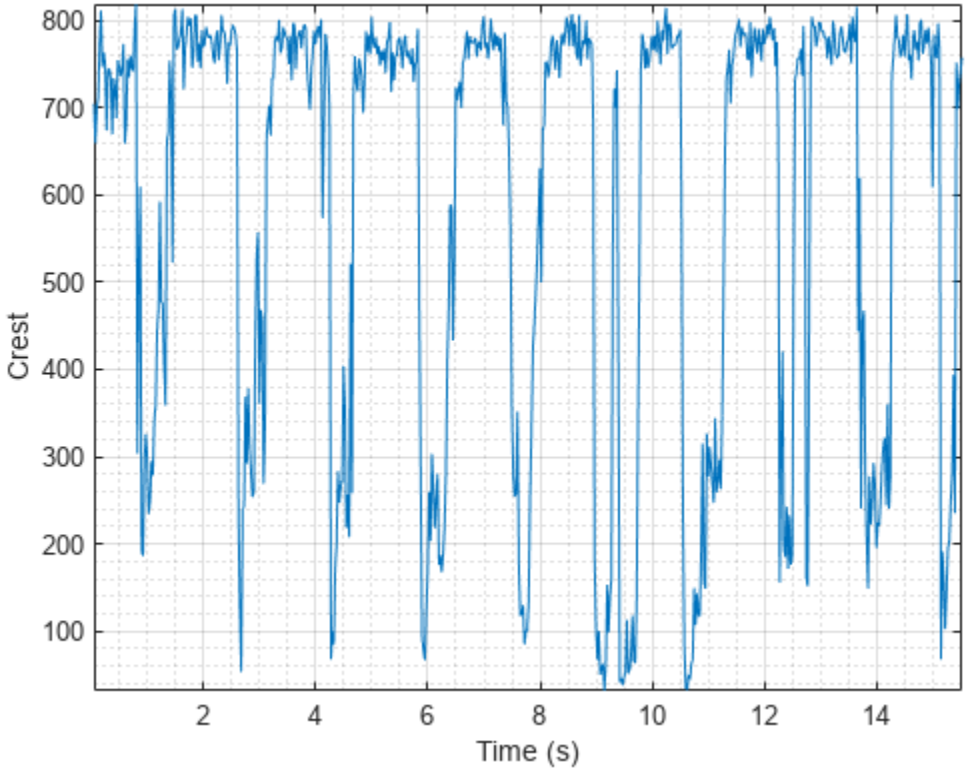
```
[audioIn,fs] = audioread("Counting-16-44p1-mono-15secs.wav");
```

Calculate the crest of the power spectrum over time. Calculate the crest for 50 ms Hamming windows of data with 25 ms overlap. Use the range from 62.5 Hz to `fs`/2 for the crest calculation.

```
crest = spectralCrest(audioIn,fs, ...
                      Window=hamming(round(0.05*fs)), ...
                      OverlapLength=round(0.025*fs), ...
                      Range=[62.5,fs/2]);
```

Plot the crest against time.

```
spectralCrest(audioIn,fs, ...
              Window=hamming(round(0.05*fs)), ...
              OverlapLength=round(0.025*fs), ...
              Range=[62.5,fs/2])
```



Calculate Spectral Crest of Streaming Audio

Create a `dsp.AudioFileReader` object to read in audio data frame-by-frame. Create a `dsp.SignalSink` to log the spectral crest calculation.

```
fileReader = dsp.AudioFileReader('Counting-16-44p1-mono-15secs.wav');
logger = dsp.SignalSink;
```

In an audio stream loop:

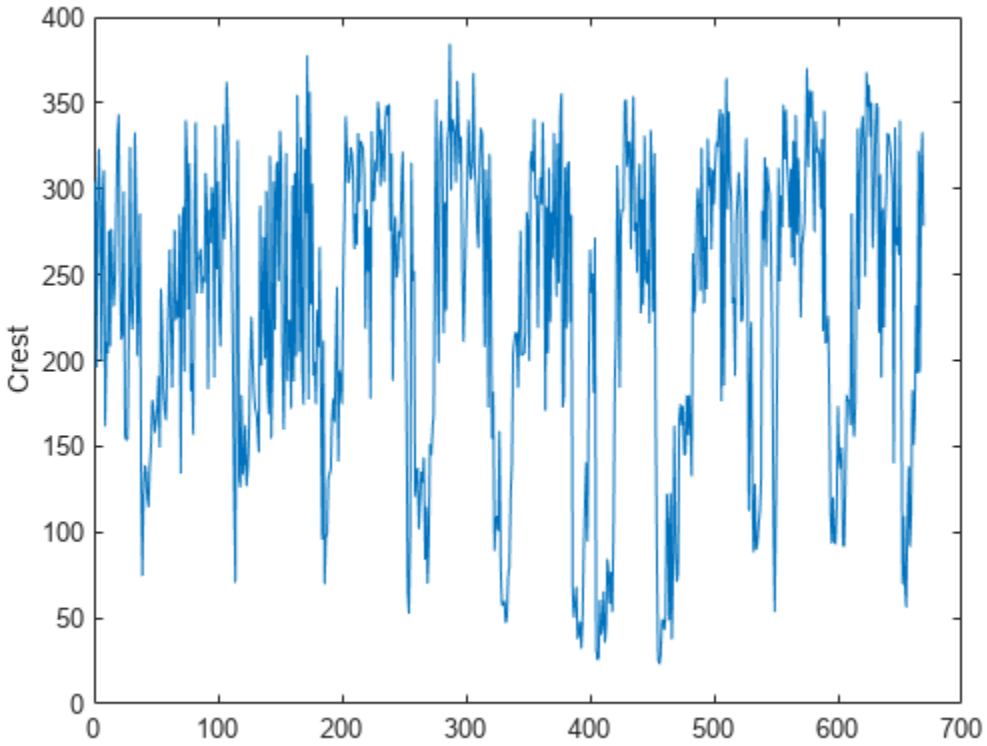
1. Read in a frame of audio data.
2. Calculate the spectral crest for the frame of audio.
3. Log the spectral crest for later plotting.

To calculate the spectral crest for only a given input frame, specify a window with the same number of samples as the input, and set the overlap length to zero.

Plot the logged data.

```
while ~isDone(fileReader)
    audioIn = fileReader();
    crest = spectralCrest(audioIn,fileReader.SampleRate, ...
                          'Window',hamming(size(audioIn,1)), ...
                          'OverlapLength',0);
```

```
    logger(crest)
end

plot(logger.Buffer)
ylabel('Crest')
```



Use `dsp.AsyncBuffer` if

- The input to your audio stream loop has a variable samples-per-frame.
- The input to your audio stream loop has an inconsistent samples-per-frame with the analysis window of `spectralCrest`.
- You want to calculate the spectral crest for overlapped data.

Create a `dsp.AsyncBuffer` object, reset the logger, and release the file reader.

```
buff = dsp.AsyncBuffer;
reset(logger)
release(fileReader)
```

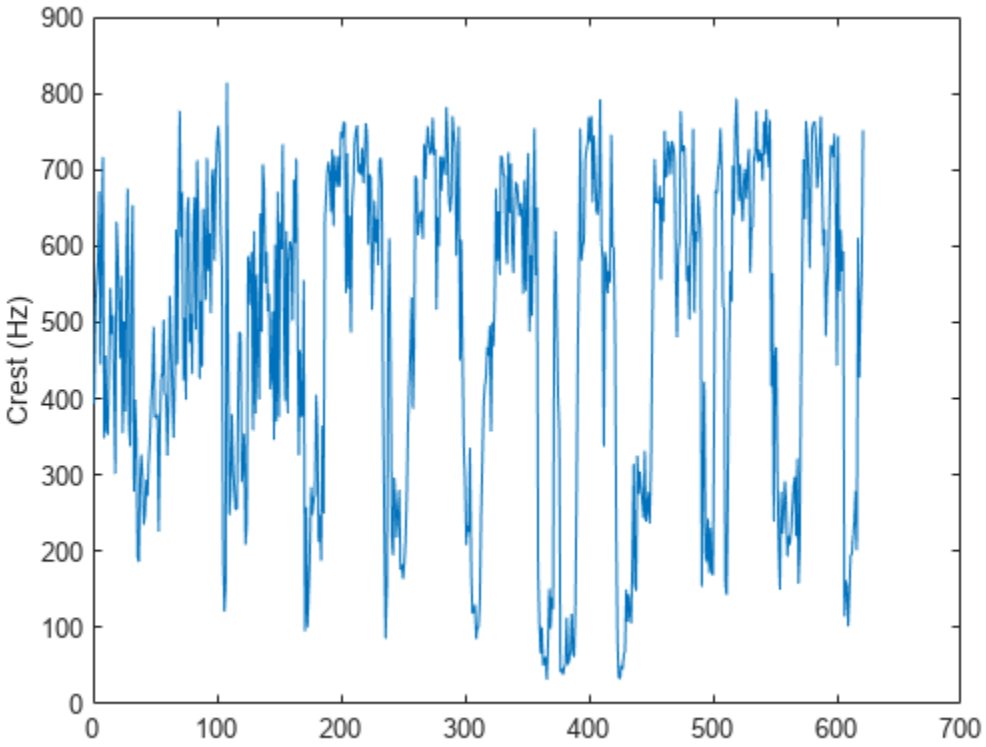
Specify that the spectral crest is calculated for 50 ms frames with a 25 ms overlap.

```
fs = fileReader.SampleRate;

samplesPerFrame = round(fs*0.05);
samplesOverlap = round(fs*0.025);

samplesPerHop = samplesPerFrame - samplesOverlap;
```

```
win = hamming(samplesPerFrame);

while ~isDone(fileReader)
    audioIn = fileReader();
    write(buff,audioIn);

    while buff.NumUnreadSamples >= samplesPerHop
        audioBuffered = read(buff,samplesPerFrame,samplesOverlap);

        crest = spectralCrest(audioBuffered,fs, ...
                              'Window',win, ...
                              'OverlapLength',0);
        logger(crest)
    end

end
release(fileReader)
```

Plot the logged data.

```
plot(logger.Buffer)
ylabel('Crest (Hz)')
```



Input Arguments

`x` — Input signal

column vector | matrix | 3-D array

Input signal, specified as a vector, matrix, or 3-D array. How the function interprets `x` depends on the shape of `f`.

Data Types: `single` | `double`

`f` — Sample rate or frequency vector (Hz)

scalar | vector

Sample rate or frequency vector in Hz, specified as a scalar or vector, respectively. How the function interprets `x` depends on the shape of `f`:

- If `f` is a scalar, `x` is interpreted as a time-domain signal, and `f` is interpreted as the sample rate. In this case, `x` must be a real vector or matrix. If `x` is specified as a matrix, the columns are interpreted as individual channels.
- If `f` is a vector, `x` is interpreted as a frequency-domain signal, and `f` is interpreted as the frequencies, in Hz, corresponding to the rows of `x`. In this case, `x` must be a real *L*-by-*M*-by-*N* array, where *L* is the number of spectral values at given frequencies of `f`, *M* is the number of individual spectra, and *N* is the number of channels.
- The number of rows of `x`, *L*, must be equal to the number of elements of `f`.

Data Types: `single` | `double`

Name-Value Arguments

Note The following name-value arguments apply if `x` is a time-domain signal. If `x` is a frequency-domain signal, name-value arguments are ignored.

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `Window=hamming(256)`

`Window` — Window applied in time domain

`rectwin(round(f*0.03))` (default) | vector

Window applied in the time domain, specified as a real vector. The number of elements in the vector must be in the range [1, `size(x,1)`]. The number of elements in the vector must also be greater than `OverlapLength`.

Data Types: `single` | `double`

`OverlapLength` — Number of samples overlapped between adjacent windows

`round(f*0.02)` (default) | non-negative scalar

Number of samples overlapped between adjacent windows, specified as an integer in the range [0, `size(Window,1)`).

Data Types: `single` | `double`

FFTLength — Number of bins in DFT
`numel(Window)` (default) | positive scalar integer

Number of bins used to calculate the DFT of windowed input samples, specified as a positive scalar integer. If unspecified, `FFTLength` defaults to the number of elements in the `Window`.

Data Types: `single` | `double`

Range — Frequency range (Hz)
`[0,f/2]` (default) | two-element row vector

Frequency range in Hz, specified as a two-element row vector of increasing real values in the range [0, `f`/2].

Data Types: `single` | `double`

SpectrumType — Spectrum type
`"power"` (default) | `"magnitude"`

Spectrum type, specified as `"power"` or `"magnitude"`:

- `"power"` -– The spectral crest is calculated for the one-sided power spectrum.
- `"magnitude"` -– The spectral crest is calculated for the one-sided magnitude spectrum.

Data Types: `char` | `string`

Output Arguments

`crest` — Spectral crest
scalar | vector | matrix

Spectral crest, returned as a scalar, vector, or matrix. Each row of `crest` corresponds to the spectral crest of a window of `x`. Each column of `crest` corresponds to an independent channel.

`spectralPeak` — Spectral peak
scalar | vector | matrix

Spectral peak, returned as a scalar, vector, or matrix. Each row of `spectralPeak` corresponds to the spectral crest of a window of `x`. Each column of `spectralPeak` corresponds to an independent channel.

`spectralMean` — Spectral mean
scalar | vector | matrix

Spectral mean, returned as a scalar, vector, or matrix. Each row of `spectralMean` corresponds to the spectral crest of a window of `x`. Each column of `spectralMean` corresponds to an independent channel.

Algorithms

The spectral crest is calculated as described in [1]:

$$\text{crest} = \frac{\max\left(s_k \in [b_1, b_2]\right)}{\frac{1}{b_2 - b_1}\sum_{k=b_1}^{b_2} s_k}$$

where

- s_k is the spectral value at bin k.
- b_1 and b_2 are the band edges, in bins, over which to calculate the spectral crest.

Version History

Introduced in R2019a

References

[1] Peeters, G. "A Large Set of Audio Features for Sound Description (Similarity and Classification) in the CUIDADO Project." Technical Report; IRCAM: Paris, France, 2004.

Extended Capabilities

C/C++ Code Generation

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

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

See Also

`spectralSpread` | `spectralFlatness` | `spectralSkewness`

Topics

“Spectral Descriptors”

spectralCentroid

Spectral centroid for audio signals and auditory spectrograms

Syntax

```
centroid = spectralCentroid(x,f)
centroid = spectralCentroid(x,f,Name=Value)
spectralCentroid( ___ )
```

Description

`centroid = spectralCentroid(x,f)` returns the spectral centroid of the signal, `x`, over time. How the function interprets `x` depends on the shape of `f`.

`centroid = spectralCentroid(x,f,Name=Value)` specifies options using one or more name-value arguments.

`spectralCentroid( ___ )` with no output arguments plots the spectral centroid. You can specify an input combination from any of the previous syntaxes.

- If the input is in the time domain, the spectral centroid is plotted against time.
- If the input is in the frequency domain, the spectral centroid is plotted against frame number.

Examples

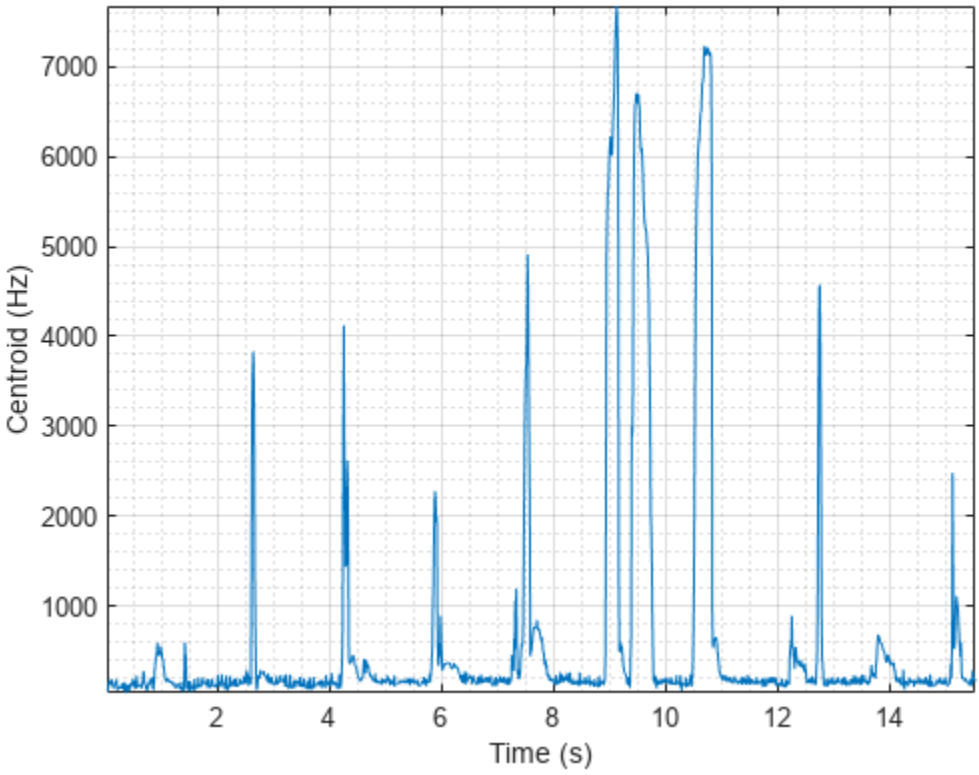
Spectral Centroid of Time-Domain Audio

Read in an audio file and calculate the centroid using default parameters.

```
[audioIn,fs] = audioread("Counting-16-44p1-mono-15secs.wav");
centroid = spectralCentroid(audioIn,fs);
```

Plot the centroid against time.

```
spectralCentroid(audioIn,fs);
```



Spectral Centroid of Frequency-Domain Audio Data

Read in an audio file and then buffer the signal into 30 ms frames with 20 ms overlap. Calculate the octave power spectrum using the `poctave` function.

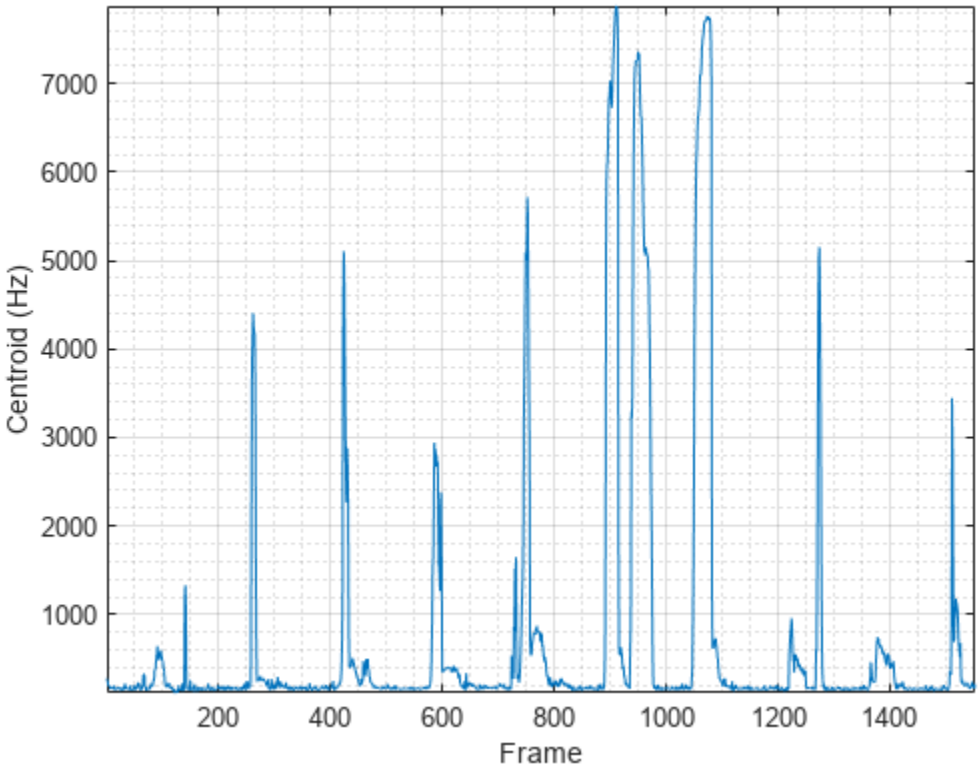
```
[audioIn,fs] = audioread("Counting-16-44p1-mono-15secs.wav");
audioBuffered = buffer(audioIn,round(fs*0.03),round(fs*0.02),"nodelay");
[p,cf] = poctave(audioBuffered,fs);
```

Calculate the centroid of the octave power spectrum over time.

```
centroid = spectralCentroid(p,cf);
```

Plot the centroid against the frame number.

```
spectralCentroid(p,cf)
```



Specify Nondefault Parameters

Read in an audio file.

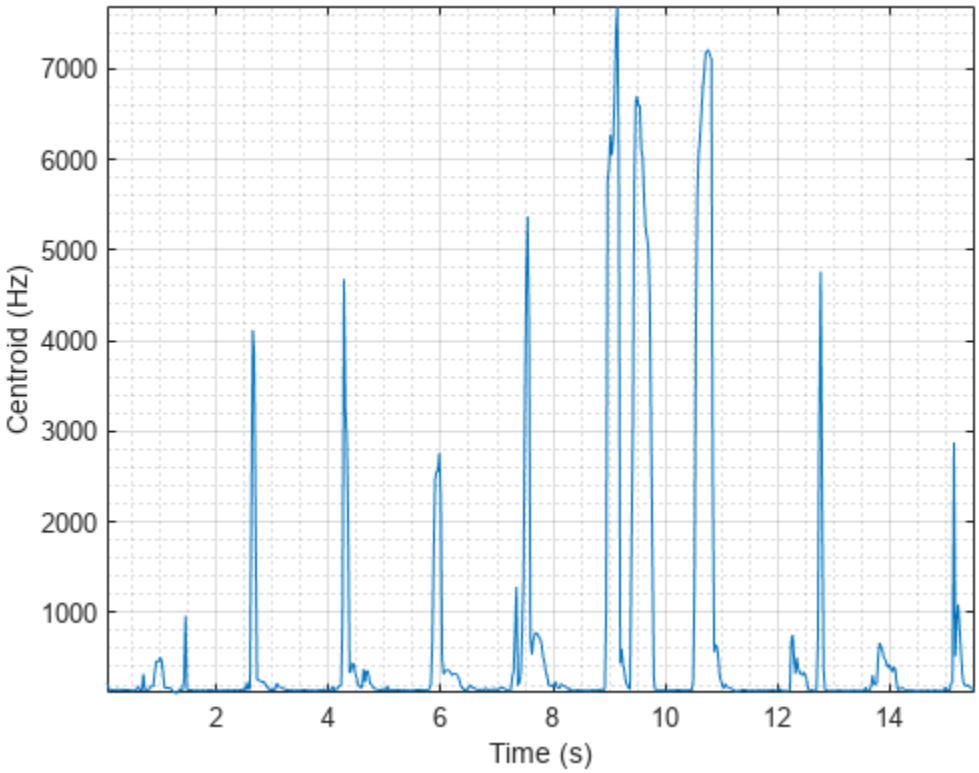
```
[audioIn,fs] = audioread("Counting-16-44p1-mono-15secs.wav");
```

Calculate the centroid of the power spectrum over time. Calculate the centroid for 50 ms Hamming windows of data with 25 ms overlap. Use the range from 62.5 Hz to `fs`/2 for the centroid calculation.

```
centroid = spectralCentroid(audioIn,fs, ...
                            Window=hamming(round(0.05*fs)), ...
                            OverlapLength=round(0.025*fs), ...
                            Range=[62.5,fs/2]);
```

Plot the centroid against time.

```
spectralCentroid(audioIn,fs, ...
                 Window=hamming(round(0.05*fs)), ...
                 OverlapLength=round(0.025*fs), ...
                 Range=[62.5,fs/2])
```



Calculate Spectral Centroid of Streaming Audio

Create a `dsp.AudioFileReader` object to read in audio data frame-by-frame. Create a `dsp.SignalSink` to log the spectral centroid calculation.

```
fileReader = dsp.AudioFileReader('Counting-16-44p1-mono-15secs.wav');
logger = dsp.SignalSink;
```

In an audio stream loop:

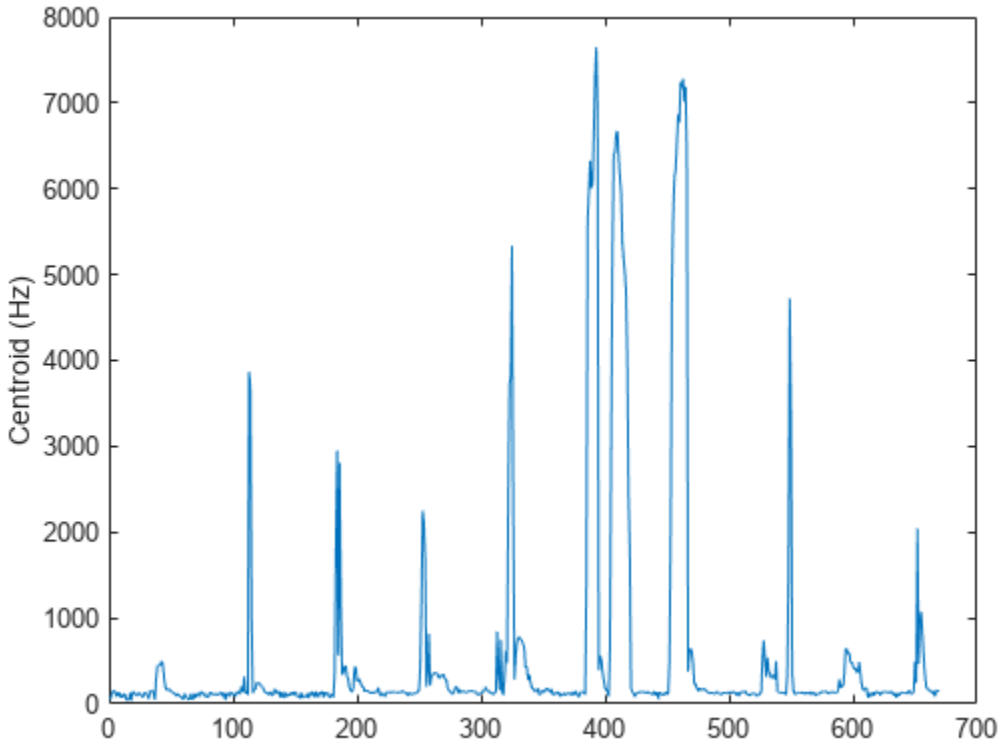
1. Read in a frame of audio data.
2. Calculate the spectral centroid for the frame of audio.
3. Log the spectral centroid for later plotting.

To calculate the spectral centroid for only a given input frame, specify a window with the same number of samples as the input, and set the overlap length to zero.

Plot the logged data.

```
while ~isDone(fileReader)
    audioIn = fileReader();
    centroid = spectralCentroid(audioIn,fileReader.SampleRate, ...
                                'Window',hamming(size(audioIn,1)), ...
                                'OverlapLength',0);
```

```
    logger(centroid)
end

plot(logger.Buffer)
ylabel('Centroid (Hz)')
```



If the input to your audio stream loop has a variable samples-per-frame, an inconsistent samples-per-frame with the analysis window size of `spectralCentroid`, or if you want to calculate the spectral centroid for overlapped data, use `dsp.AsyncBuffer`.

Create a `dsp.AsyncBuffer` object, reset the logger, and release the file reader.

```
buff = dsp.AsyncBuffer;
reset(logger)
release(fileReader)
```

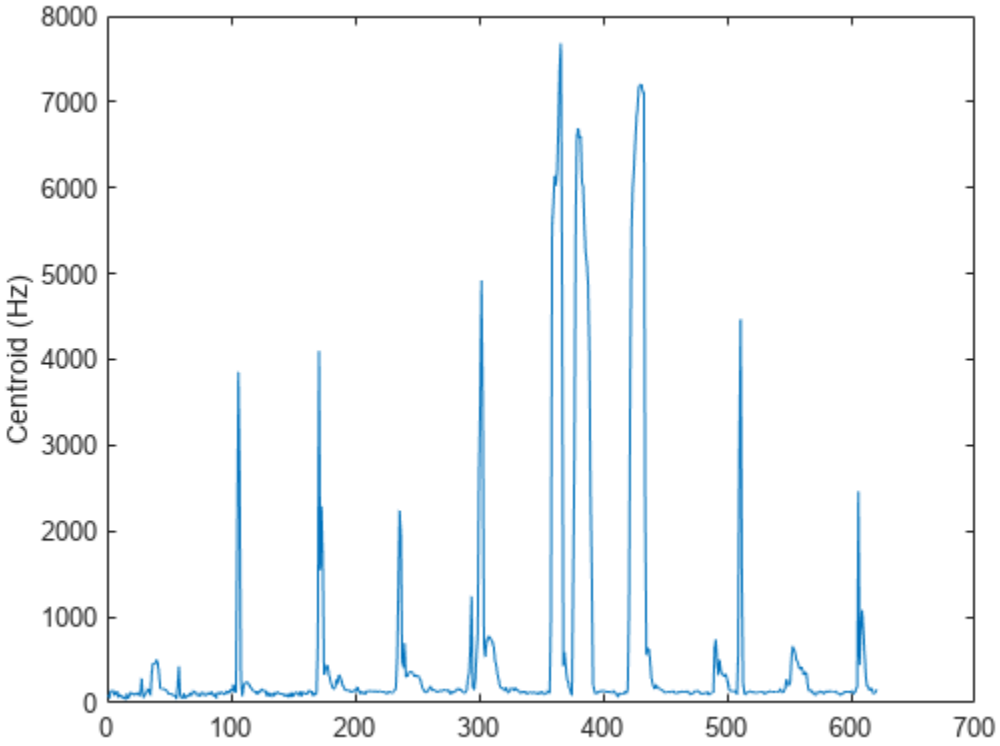
Specify that the spectral centroid is calculated for 50 ms frames with a 25 ms overlap.

```
fs = fileReader.SampleRate;

samplesPerFrame = round(fs*0.05);
samplesOverlap = round(fs*0.025);

samplesPerHop = samplesPerFrame - samplesOverlap;

win = hamming(samplesPerFrame);

while ~isDone(fileReader)
```

```
    audioIn = fileReader();
    write(buff,audioIn);

    while buff.NumUnreadSamples >= samplesPerHop
        audioBuffered = read(buff,samplesPerFrame,samplesOverlap);

        centroid = spectralCentroid(audioBuffered,fs, ...
                                    'Window',win, ...
                                    'OverlapLength',0);
        logger(centroid)
    end

end
release(fileReader)
```

Plot the logged data.

```
plot(logger.Buffer)
ylabel('Centroid (Hz)')
```



Input Arguments

x — Input signal

column vector | matrix | 3-D array

Input signal, specified as a vector, matrix, or 3-D array. How the function interprets `x` depends on the shape of `f`.

Data Types: `single` | `double`

`f` — Sample rate or frequency vector (Hz)

scalar | vector

Sample rate or frequency vector in Hz, specified as a scalar or vector, respectively. How the function interprets `x` depends on the shape of `f`:

- If `f` is a scalar, `x` is interpreted as a time-domain signal, and `f` is interpreted as the sample rate. In this case, `x` must be a real vector or matrix. If `x` is specified as a matrix, the columns are interpreted as individual channels.
- If `f` is a vector, `x` is interpreted as a frequency-domain signal, and `f` is interpreted as the frequencies, in Hz, corresponding to the rows of `x`. In this case, `x` must be a real *L*-by-*M*-by-*N* array, where *L* is the number of spectral values at given frequencies of `f`, *M* is the number of individual spectra, and *N* is the number of channels.
- The number of rows of `x`, *L*, must be equal to the number of elements of `f`.

Data Types: `single` | `double`

Name-Value Arguments

Note The following name-value arguments apply if `x` is a time-domain signal. If `x` is a frequency-domain signal, name-value arguments are ignored.

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `Window=hamming(256)`

`Window` — Window applied in time domain

`rectwin(round(f*0.03))` (default) | vector

Window applied in the time domain, specified as a real vector. The number of elements in the vector must be in the range [1, `size(x,1)`]. The number of elements in the vector must also be greater than `OverlapLength`.

Data Types: `single` | `double`

`OverlapLength` — Number of samples overlapped between adjacent windows

`round(f*0.02)` (default) | non-negative scalar

Number of samples overlapped between adjacent windows, specified as an integer in the range [0, `size(Window,1)`).

Data Types: `single` | `double`

`FFTLength` — Number of bins in DFT

`numel(Window)` (default) | positive scalar integer

Number of bins used to calculate the DFT of windowed input samples, specified as a positive scalar integer. If unspecified, `FFTLength` defaults to the number of elements in the `Window`.

Data Types: `single` | `double`

Range — Frequency range (Hz)
`[0,f/2]` (default) | two-element row vector

Frequency range in Hz, specified as a two-element row vector of increasing real values in the range [0, `f`/2].

Data Types: `single` | `double`

`SpectrumType` — Spectrum type
`"power"` (default) | `"magnitude"`

Spectrum type, specified as `"power"` or `"magnitude"`:

- `"power"` –- The spectral centroid is calculated for the one-sided power spectrum.
- `"magnitude"` –- The spectral centroid is calculated for the one-sided magnitude spectrum.

Data Types: `char` | `string`

Output Arguments

`centroid` — Spectral centroid (Hz)
scalar | vector | matrix

Spectral centroid in Hz, returned as a scalar, vector, or matrix. Each row of `centroid` corresponds to the spectral centroid of a window of `x`. Each column of `centroid` corresponds to an independent channel.

Algorithms

The spectral centroid is calculated as described in [1]:

$$\text{centroid} = \frac{\sum_{k=b_1}^{b_2} f_k s_k}{\sum_{k=b_1}^{b_2} s_k}$$

where

- f_k is the frequency in Hz corresponding to bin k.
- s_k is the spectral value at bin k.
- b_1 and b_2 are the band edges, in bins, over which to calculate the spectral centroid.

Version History

Introduced in R2019a

References

[1] Peeters, G. "A Large Set of Audio Features for Sound Description (Similarity and Classification) in the CUIDADO Project." Technical Report; IRCAM: Paris, France, 2004.

Extended Capabilities

C/C++ Code Generation

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

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

See Also

`spectralSkewness` | `spectralKurtosis` | `spectralSpread`

Topics

“Spectral Descriptors”

hz2mel

Convert from hertz to mel scale

Syntax

```
mel = hz2mel(hz)
```

Description

`mel = hz2mel(hz)` converts values in hertz to values on the mel frequency scale.

Examples

Convert Between Mel Scale and Hz

Set two bounding frequencies in Hz and then convert them to the mel scale.

```
b = hz2mel([20,8000]);
```

Generate a row vector of 32 values uniformly spaced on the mel scale.

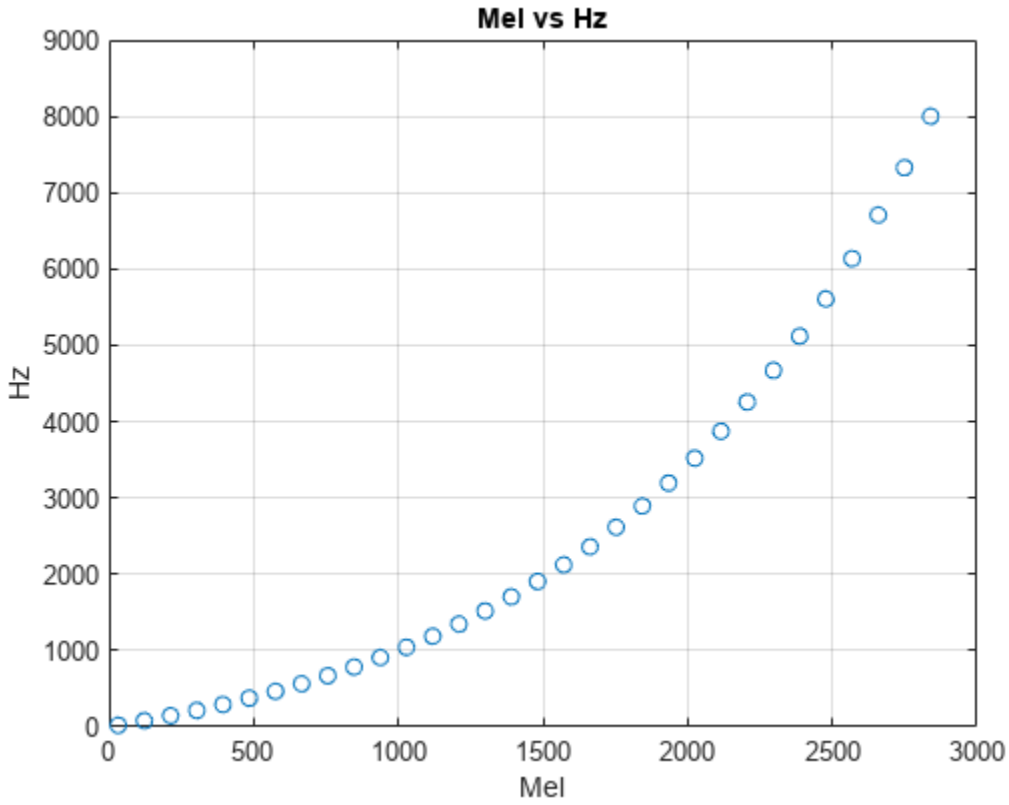
```
melVect = linspace(b(1),b(2),32);
```

Convert the row vector of values into equivalent frequencies in Hz.

```
hzVect = mel2hz(melVect);
```

Plot the two vectors for comparison. As mel values increase linearly, Hz values increase exponentially.

```
plot(melVect,hzVect,'o')
title('Mel vs Hz')
xlabel('Mel')
ylabel('Hz')
grid on
```



Input Arguments

`hz` — Input frequency in Hz

scalar | vector | matrix | multidimensional array

Input frequency in Hz, specified as a scalar, vector, matrix, or multidimensional array.

Data Types: `single` | `double`

Output Arguments

`mel` — Output frequency on mel scale

scalar | vector | matrix | multidimensional array

Output frequency on the mel scale, returned as a scalar, vector, matrix, or multidimensional array the same size as `hz`.

Data Types: `single` | `double`

Algorithms

The frequency conversion from Hz to the mel scale uses the following formula:

$$mel = 2595\log_{10}\left(1 + \frac{hz}{700}\right)$$

Version History

Introduced in R2019a

References

[1] O'Shaghnessy, Douglas. *Speech Communication: Human and Machine.* Reading, MA: Addison-Wesley Publishing Company, 1987.

Extended Capabilities

C/C++ Code Generation

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

See Also

`mel2hz` | `hz2erb` | `erb2hz` | `hz2bark` | `bark2hz`

hz2bark

Convert from hertz to Bark scale

Syntax

```
bark = hz2bark(hz)
```

Description

`bark = hz2bark(hz)` converts values in hertz to values on the Bark frequency scale.

Examples

Convert Between Bark Scale and Hz

Set two bounding frequencies in Hz and then convert them to the Bark scale.

```
b = hz2bark([20,8000]);
```

Generate a row vector of 32 values uniformly spaced on the Bark scale.

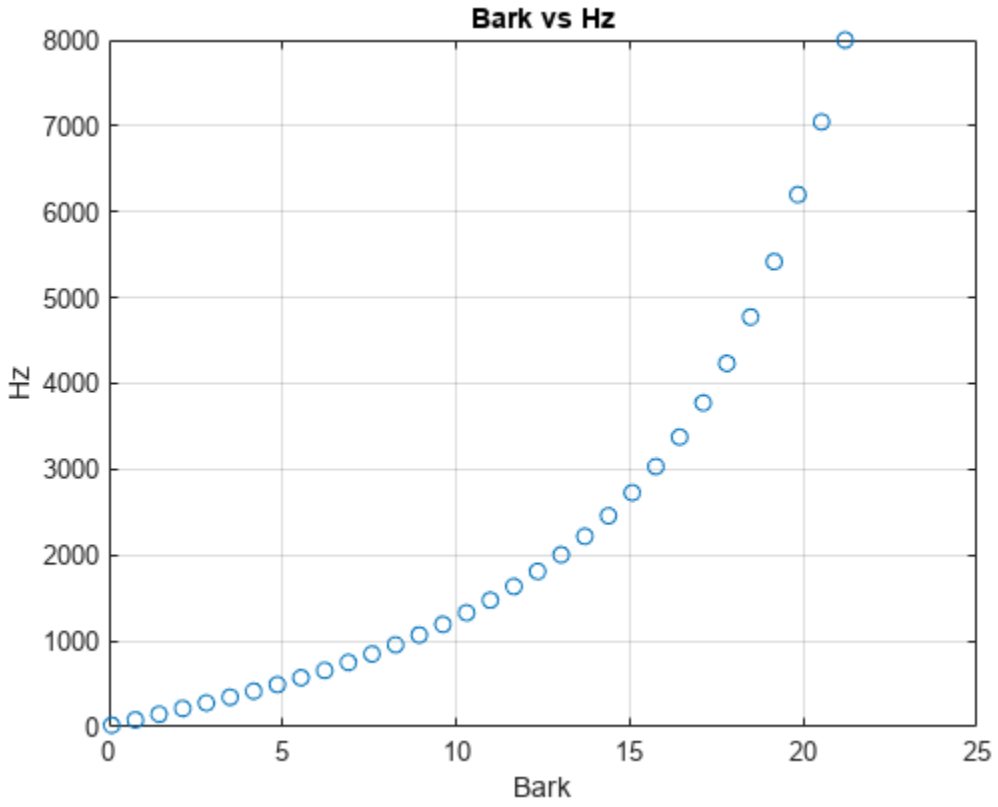
```
barkVect = linspace(b(1),b(2),32);
```

Convert the row vector of values into equivalent frequencies in Hz.

```
hzVect = bark2hz(barkVect);
```

Plot the two vectors for comparison. As Bark values increase linearly, Hz values increase exponentially.

```
plot(barkVect,hzVect,'o')
title('Bark vs Hz')
xlabel('Bark')
ylabel('Hz')
grid on
```



Input Arguments

hz — Input frequency in Hz

scalar | vector | matrix | multidimensional array

Input frequency in Hz, specified as a scalar, vector, matrix, or multidimensional array.

Data Types: `single` | `double`

Output Arguments

bark — Output frequency on Bark scale

scalar | vector | matrix | multidimensional array

Output frequency on the Bark scale, returned as a scalar, vector, matrix, or multidimensional array the same size as `hz`.

Data Types: `single` | `double`

Algorithms

The frequency conversion from Hz to the Bark scale uses the following formula:

$$bark = \frac{(26.81)(hz)}{1960 + hz} - 0.53$$

$$if: bark < 2 \rightarrow bark = bark + (0.15)(2 - bark)$$

$$if: bark > 20.1 \rightarrow bark = bark + (0.22)(bark - 20.1)$$

The Bark value correction occurs after the conversion from Hz to the Bark scale.

Version History

Introduced in R2019a

References

[1] Traunmüller, Hartmut. "Analytical Expressions for the Tonotopic Sensory Scale." *Journal of the Acoustical Society of America.* Vol. 88, Issue 1, 1990, pp. 97–100.

Extended Capabilities

C/C++ Code Generation

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

See Also

`bark2hz` | `hz2mel` | `mel2hz` | `hz2erb` | `erb2hz`

hz2erb

Convert from hertz to equivalent rectangular bandwidth (ERB) scale

Syntax

```
erb = hz2erb(hz)
```

Description

`erb = hz2erb(hz)` converts values in hertz to values on the ERB frequency scale.

Examples

Convert Between ERB Scale and Hz

Set two bounding frequencies in Hz and then convert them to the ERB scale.

```
b = hz2erb([20,8000]);
```

Generate a row vector of 32 values uniformly spaced on the ERB scale.

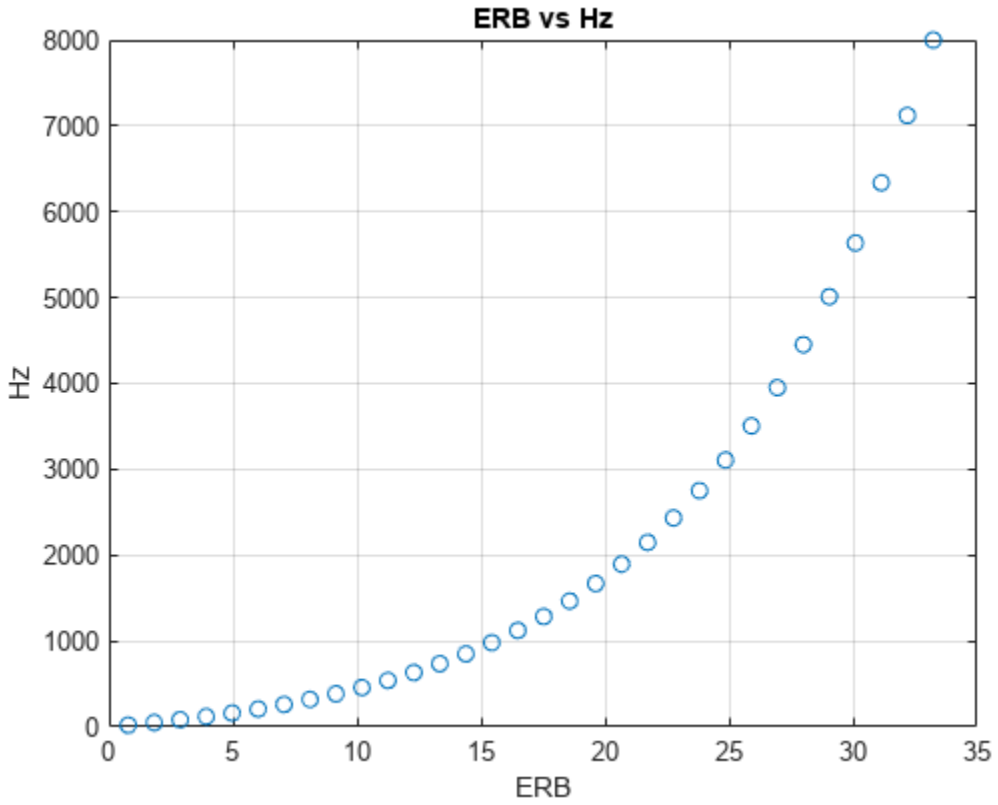
```
erbVect = linspace(b(1),b(2),32);
```

Convert the row vector of values into equivalent frequencies in Hz.

```
hzVect = erb2hz(erbVect);
```

Plot the two vectors for comparison. As ERB values increase linearly, Hz values increase exponentially.

```
plot(erbVect,hzVect,'o')
title('ERB vs Hz')
xlabel('ERB')
ylabel('Hz')
grid on
```



Input Arguments

hz — Input frequency in Hz

scalar | vector | matrix | multidimensional array

Input frequency in Hz, specified as a scalar, vector, matrix, or multidimensional array.

Data Types: `single` | `double`

Output Arguments

erb — Output frequency on ERB scale

scalar | vector | matrix | multidimensional array

Output frequency on the ERB scale, returned as a scalar, vector, matrix, or multidimensional array the same size as `hz`.

Data Types: `single` | `double`

Algorithms

The frequency conversion from Hz to the ERB scale uses the following formula:

$erb = A\log_{10}(1 + hz(0.00437))$

where

$A = \frac{1000\log_e(10)}{(24.7)(4.37)}$

Version History

Introduced in R2019a

References

[1] Glasberg, Brian R., and Brian C. J. Moore. "Derivation of Auditory Filter Shapes from Notched-Noise Data." *Hearing Research*. Vol. 47, Issues 1–2, 1990, pp. 103–138.

Extended Capabilities

C/C++ Code Generation

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

See Also

`erb2hz` | `hz2mel` | `mel2hz` | `hz2bark` | `bark2hz`

mel2hz

Convert from mel scale to hertz

Syntax

```
hz = mel2hz(mel)
```

Description

`hz = mel2hz(mel)` converts values on the mel frequency scale to values in hertz.

Examples

Convert Between Mel Scale and Hz

Set two bounding frequencies in Hz and then convert them to the mel scale.

```
b = hz2mel([20,8000]);
```

Generate a row vector of 32 values uniformly spaced on the mel scale.

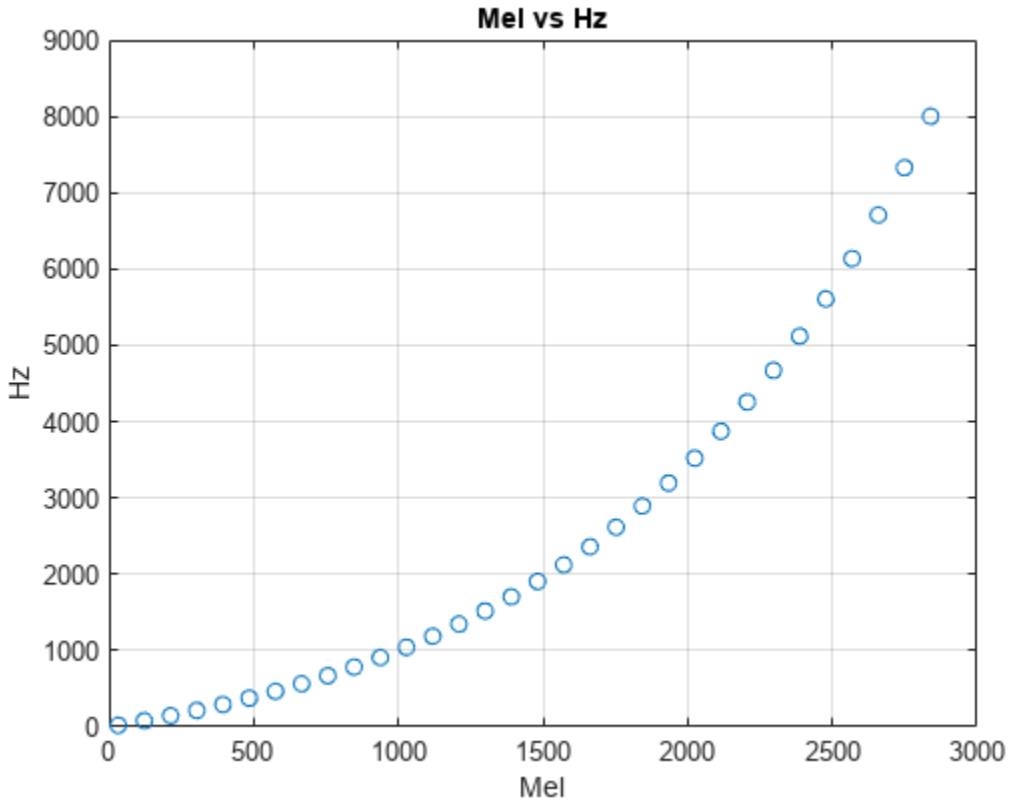
```
melVect = linspace(b(1),b(2),32);
```

Convert the row vector of values into equivalent frequencies in Hz.

```
hzVect = mel2hz(melVect);
```

Plot the two vectors for comparison. As mel values increase linearly, Hz values increase exponentially.

```
plot(melVect,hzVect,'o')
title('Mel vs Hz')
xlabel('Mel')
ylabel('Hz')
grid on
```



Input Arguments

`mel` — Input frequency on mel scale

scalar | vector | matrix | multidimensional array

Input frequency on the mel scale, specified as a scalar, vector, matrix, or multidimensional array.

Data Types: `single` | `double`

Output Arguments

`hz` — Output frequency in Hz

scalar | vector | matrix | multidimensional array

Output frequency in Hz, returned as a scalar, vector, matrix, or multidimensional array the same size as `mel`.

Data Types: `single` | `double`

Algorithms

The frequency conversion from the mel scale to Hz uses the following formula:

$$hz = 700\left(10^{\frac{mel}{2595}} - 1\right)$$

Version History

Introduced in R2019a

References

[1] O'Shaughnessy, Douglas. *Speech Communication: Human and Machine*. Reading, MA: Addison-Wesley Publishing Company, 1987.

Extended Capabilities

C/C++ Code Generation

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

See Also

`hz2mel` | `hz2erb` | `erb2hz` | `hz2bark` | `bark2hz`

bark2hz

Convert from Bark scale to hertz

Syntax

```
hz = bark2hz(bark)
```

Description

`hz = bark2hz(bark)` converts values on the Bark frequency scale to values in hertz.

Examples

Convert Between Bark Scale and Hz

Set two bounding frequencies in Hz and then convert them to the Bark scale.

```
b = hz2bark([20,8000]);
```

Generate a row vector of 32 values uniformly spaced on the Bark scale.

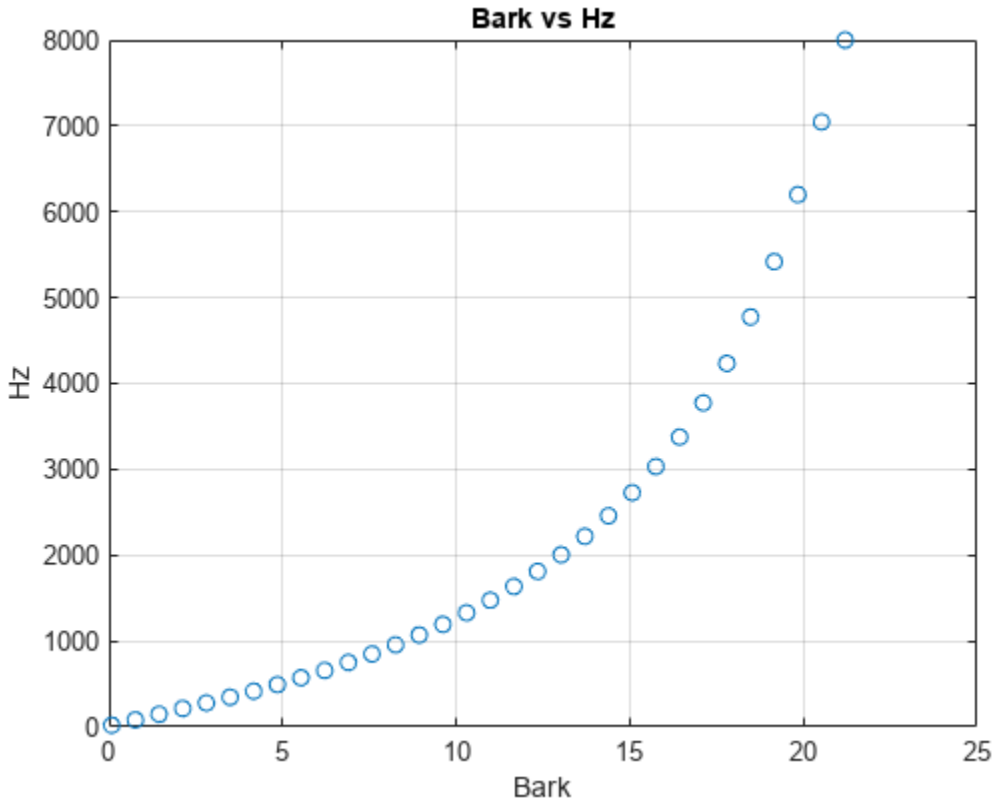
```
barkVect = linspace(b(1),b(2),32);
```

Convert the row vector of values into equivalent frequencies in Hz.

```
hzVect = bark2hz(barkVect);
```

Plot the two vectors for comparison. As Bark values increase linearly, Hz values increase exponentially.

```
plot(barkVect,hzVect,'o')
title('Bark vs Hz')
xlabel('Bark')
ylabel('Hz')
grid on
```



Input Arguments

bark — Input frequency on Bark scale

scalar | vector | matrix | multidimensional array

Input frequency on the Bark scale, specified as a scalar, vector, matrix, or multidimensional array.

Data Types: `single` | `double`

Output Arguments

hz — Output frequency in Hz

scalar | vector | matrix | multidimensional array

Output frequency in Hz, returned as a scalar, vector, matrix, or multidimensional array the same size as `bark`.

Data Types: `single` | `double`

Algorithms

The frequency conversion from the Bark scale to Hz uses the following formula:

$$if: bark < 2 \rightarrow bark = \frac{(bark - 0.3)}{0.85}$$

$$if: bark > 20.1 \rightarrow bark = \frac{(bark + 4.422)}{1.22}$$

$$hz = 1960\left(\frac{bark + 0.53}{26.28 - bark}\right)$$

The Bark value correction occurs before the conversion from the Bark scale to Hz.

Version History

Introduced in R2019a

References

[1] Traunmüller, Hartmut. "Analytical Expressions for the Tonotopic Sensory Scale." *Journal of the Acoustical Society of America*. Vol. 88, Issue 1, 1990, pp. 97–100.

Extended Capabilities

C/C++ Code Generation

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

See Also

`hz2bark` | `hz2mel` | `mel2hz` | `hz2erb` | `erb2hz`

erb2hz

Convert from equivalent rectangular bandwidth (ERB) scale to hertz

Syntax

```
hz = erb2hz(erb)
```

Description

`hz = erb2hz(erb)` converts values on the ERB frequency scale to values in hertz.

Examples

Convert Between ERB Scale and Hz

Set two bounding frequencies in Hz and then convert them to the ERB scale.

```
b = hz2erb([20,8000]);
```

Generate a row vector of 32 values uniformly spaced on the ERB scale.

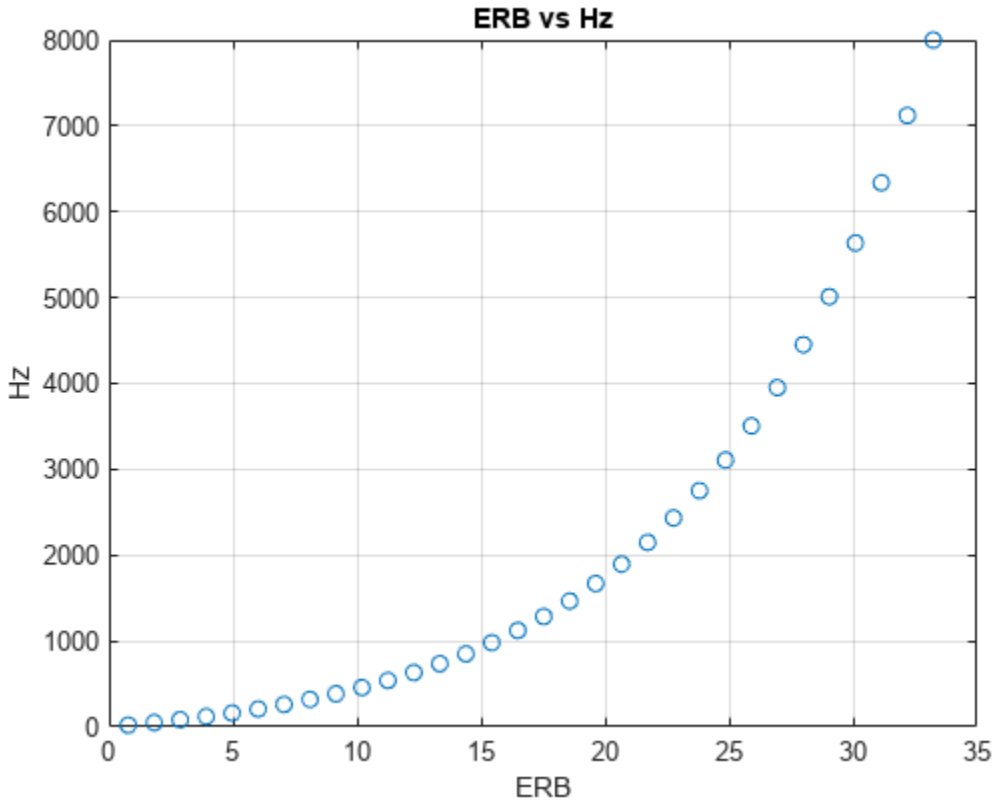
```
erbVect = linspace(b(1),b(2),32);
```

Convert the row vector of values into equivalent frequencies in Hz.

```
hzVect = erb2hz(erbVect);
```

Plot the two vectors for comparison. As ERB values increase linearly, Hz values increase exponentially.

```
plot(erbVect,hzVect,'o')
title('ERB vs Hz')
xlabel('ERB')
ylabel('Hz')
grid on
```



Input Arguments

`erb` — Input frequency on ERB scale

scalar | vector | matrix | multidimensional array

Input frequency on the equivalent rectangular band (ERB) scale, specified as a scalar, vector, matrix, or multidimensional array.

Data Types: `single` | `double`

Output Arguments

`hz` — Output frequency in Hz

scalar | vector | matrix | multidimensional array

Output frequency in Hz, returned as a scalar, vector, matrix, or multidimensional array the same size as `erb`.

Data Types: `single` | `double`

Algorithms

The frequency conversion from the ERB scale to Hz uses the following formula:

$$hz = \frac{10^{\frac{erb}{A}} - 1}{0.00437}$$

where

$$A = \frac{1000\log_e(10)}{(24.7)(4.37)}$$

Version History

Introduced in R2019a

References

[1] Glasberg, Brian R., and Brian C. J. Moore. "Derivation of Auditory Filter Shapes from Notched-Noise Data." *Hearing Research*. Vol. 47, Issues 1–2, 1990, pp. 103–138.

Extended Capabilities

C/C++ Code Generation

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

See Also

`hz2erb` | `hz2mel` | `mel2hz` | `hz2bark` | `bark2hz`

mls

Maximum length sequence

Syntax

```
excitation = mls
excitation = mls(L)
excitation = mls(L,Name,Value)
```

Description

`excitation = mls` returns an excitation signal generated using the maximum length sequence (MLS) technique. This type of sequence is a pseudo-random binary sequence.

`excitation = mls(L)` specifies the output length `L` of the excitation signal.

`excitation = mls(L,Name,Value)` specifies options using one or more `Name,Value` pair arguments, in addition to the input arguments in the previous syntaxes.

Examples

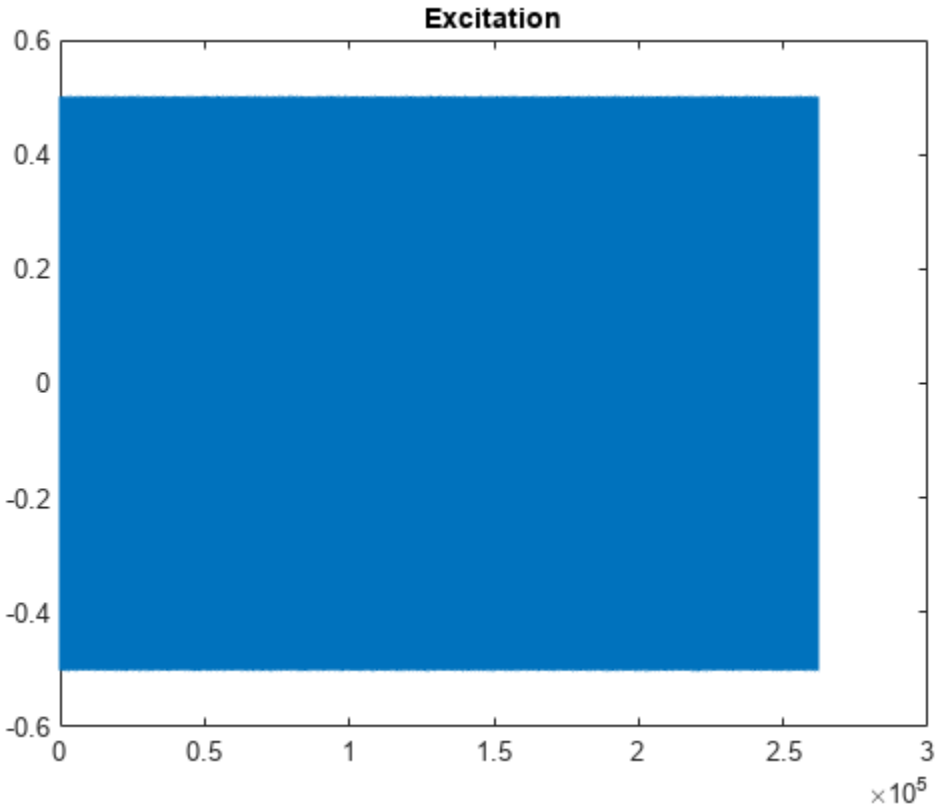
Estimate Impulse Response Using MLS Excitation

Use `audioread` to read in an impulse response recording. Create a `dsp.FrequencyDomainFIRFilter` object to perform frequency domain filtering using the known impulse response.

```
[irKnown,fs] = audioread('ChurchImpulseResponse-16-44p1-mono-5secs.wav');
systemModel = dsp.FrequencyDomainFIRFilter(irKnown');
```

Create an MLS excitation signal by using the `mls` function. The MLS excitation signal must be longer than the impulse response. Note that the length of the MLS excitation is extended to the next power of two minus one.

```
excitation = mls(numel(irKnown)+1);

plot(excitation)
title('Excitation')
```

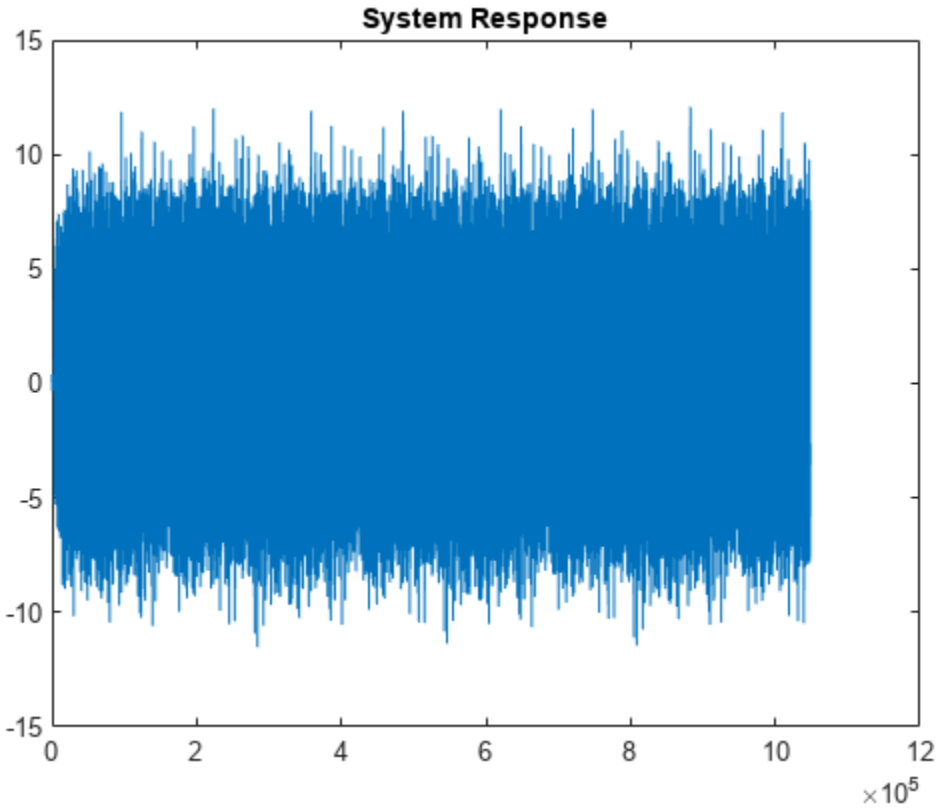


Replicate the excitation signal four times to measure the average of three measurements. The recording of the first MLS sequence does include all the impulse response information, so `impzest` discards it as a warmup run. Pad the excitation signal with zeros to account for the filter latency.

```
numRuns = 4;
excrep = repmat(excitation,numRuns,1);
excrep = [excrep;zeros(numel(irKnown)+1,1)];
```

Pass the excitation signal through the known filter and then add noise to model a real-word recording (system response). Cut the delay introduced at the beginning by the filter.

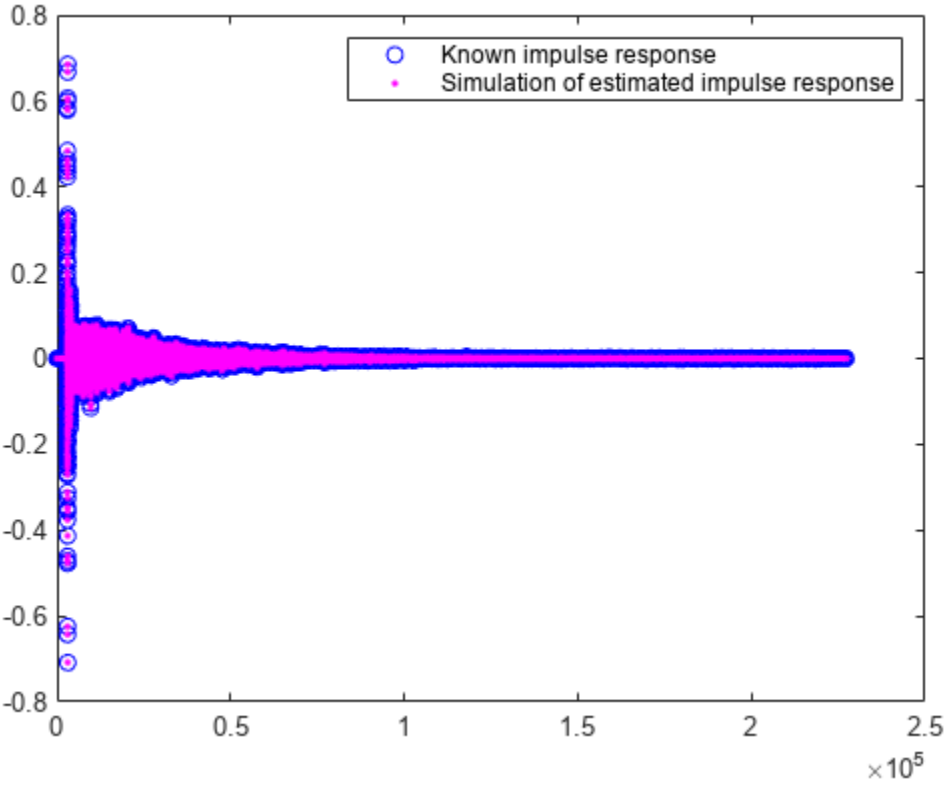
```
rec = systemModel(excrep);
rec = rec + 0.1*randn(size(rec));

rec = rec(numel(irKnown)+2:end,:);

plot(rec)
title('System Response')
```



In a real-world scenario, the MLS sequence is played back in the system under test while recording. The recording would be cut so that it begins at the moment the MLS sequence is picked-up and truncated to last the duration of the repeated sequence.

Pass the excitation signal and the system response to the `impzest` function to estimate the impulse response. Plot the known impulse response and the simulation of the estimated impulse response for comparison.

```
irEstimate = impzest(excitation,rec);

samples = 1:numel(irKnown);
plot(samples,irEstimate(samples),'bo', ...
     samples,irKnown(samples),'m.')

legend('Known impulse response','Simulation of estimated impulse response')
```



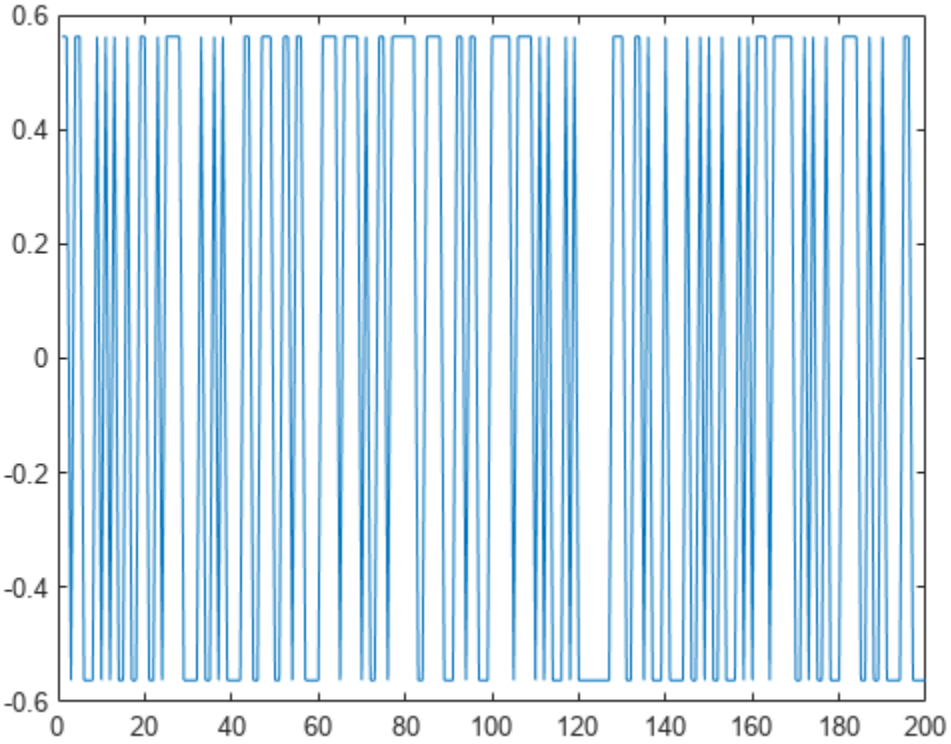
Generate MLS Signal

Generate an MLS signal that is 2^14-1 samples long and has a level of -5 dB.

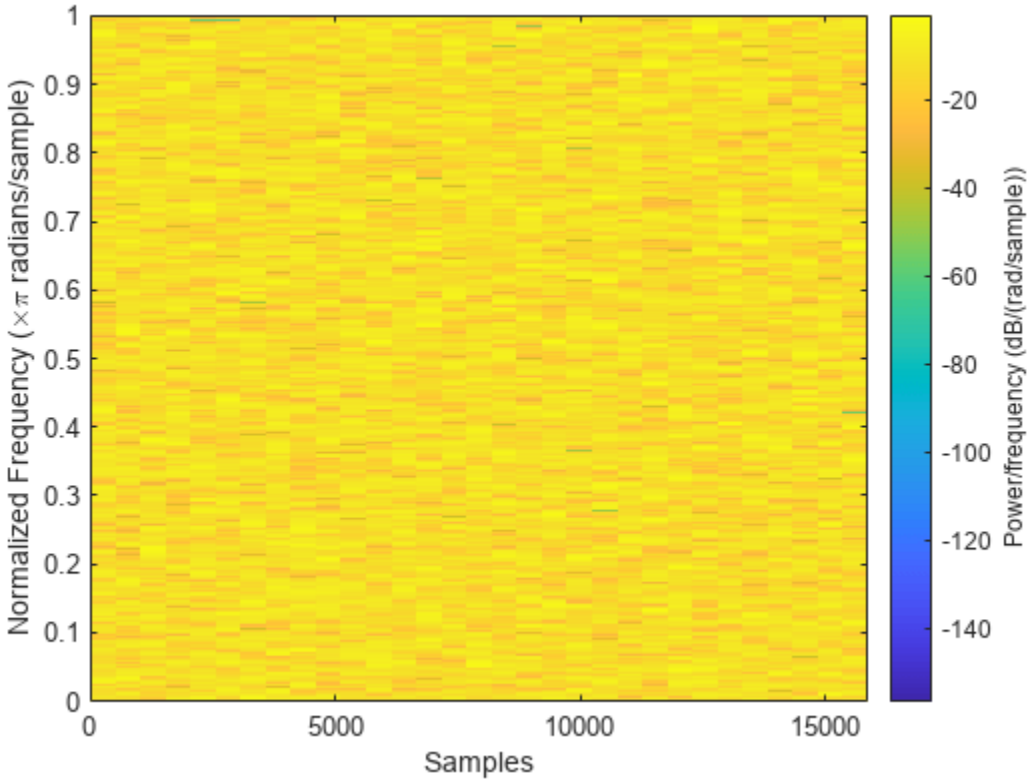
```
L = 2^14-1;
level = -5;
excitation = mls(L,'ExcitationLevel',level);
```

Visualize the excitation in time and time-frequency. For the time-domain plot, plot only the first 200 samples for visibility. The pattern is constant.

```
plot(excitation(1:200))
```




```
spectrogram(excitation,512,0,1024,'yaxis')
```



Input Arguments

L — Length of excitation signal

32767 (default) | scalar in the range $[3,2^{29})$

Length of excitation signal to generate, specified as a scalar in the range $[3,2^{29})$.

The requested output length `L` must be a power of two minus one. Otherwise, the output length increases to the next valid length.

Note If you use the excitation signal generated by the `mls` function to record and estimate the impulse response of a system, then the length of the excitation signal must be at least as long as the impulse response that you want to estimate.

Data Types: `single` | `double`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `'ExcitationLevel',-5`

`ExcitationLevel` — Level of the excitation signal to generate (dB)
scalar in the range `[-42,0]`

Level of the excitation signal to generate in dB, specified as a scalar in the range `[-42,0]`.

Data Types: `single` | `double`

Output Arguments

`excitation` — Excitation signal
column vector

Excitation signal generated using the maximum length sequence (MLS) technique, returned as a column vector.

Data Types: `single` | `double`

Version History

Introduced in R2018b

References

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

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also

`impzest` | `sweeptone` | **Impulse Response Measurer**

sweeptone

Exponential swept sine

Syntax

```
excitation = sweeptone()
excitation = sweeptone(swDur)
excitation = sweeptone(swDur,silDur)
excitation = sweeptone(swDur,silDur,fs)
excitation = sweeptone( ___ ,Name=Value)
```

Description

`excitation = sweeptone()` returns an excitation signal generated using the exponential swept sine (ESS) technique. By default, the signal has a 6-second duration, followed by 4 seconds of silence, for a sample rate of 44100 Hz.

`excitation = sweeptone(swDur)` specifies the duration of the exponential swept sine signal.

`excitation = sweeptone(swDur,silDur)` specifies the duration of the silence following the exponential swept sine signal.

`excitation = sweeptone(swDur,silDur,fs)` specifies the sample rate of the sweep tone as `fs` Hz.

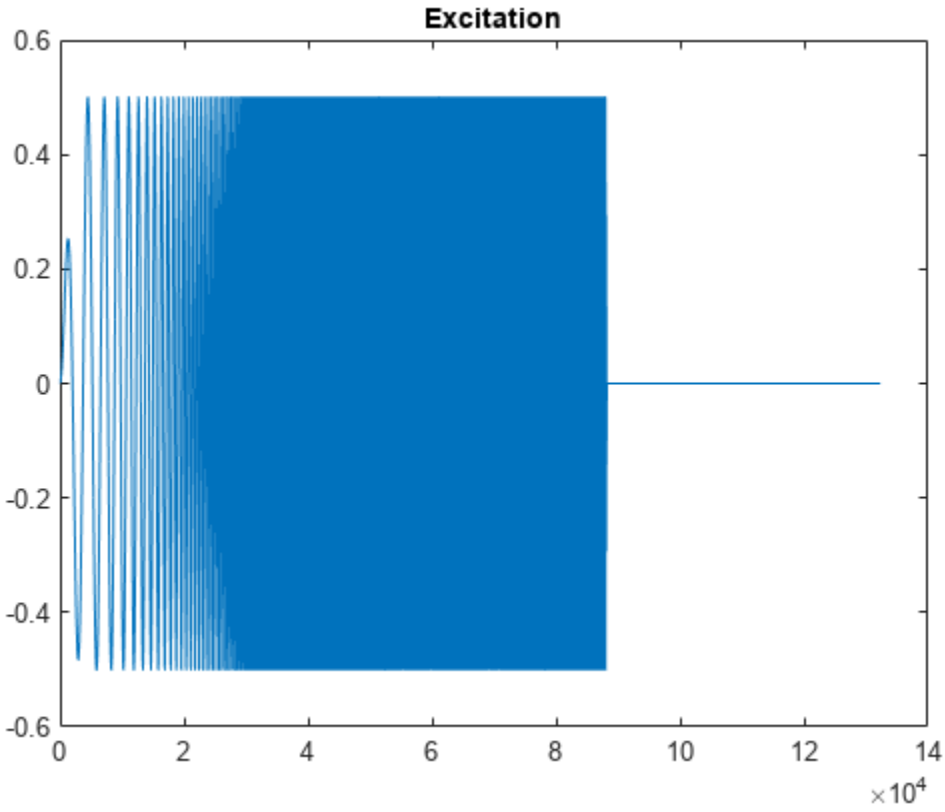
`excitation = sweeptone( ___ ,Name=Value)` specifies options using one or more name-value arguments, in addition to the input arguments in the previous syntaxes.

Examples

Estimate Impulse Response Using Sweep Tone Excitation

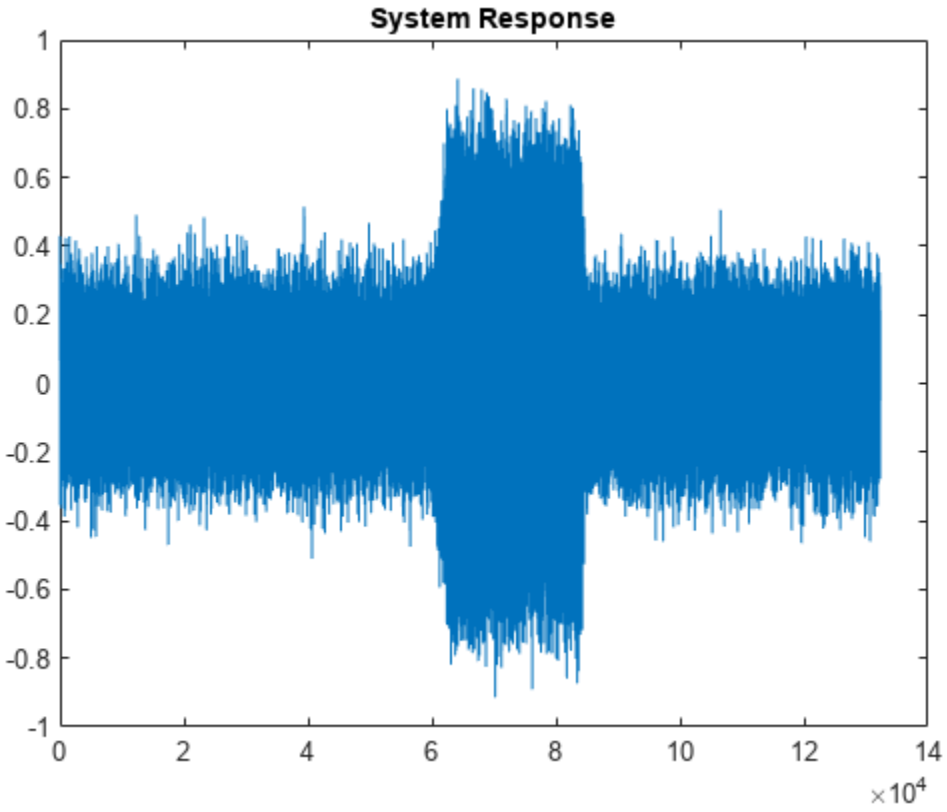
Create a sweep tone excitation signal by using the `sweeptone` function.

```
excitation = sweeptone(2,1,44100);

plot(excitation)
title('Excitation')
```



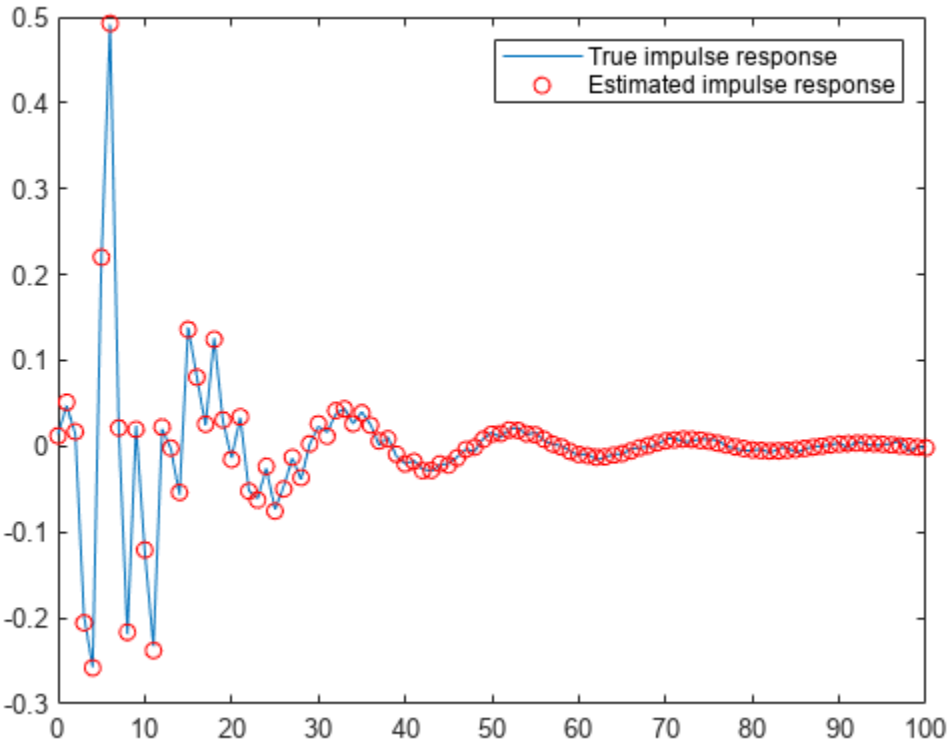
Pass the excitation signal through an infinite impulse response (IIR) filter and add noise to model a real-world recording (system response).

```
[B,A] = butter(10,[.1 .7]);
rec = filter(B,A,excitation);
nrec = rec + 0.12*randn(size(rec));

plot(nrec)
title('System Response')
```



Pass the excitation signal and the system response to the `impzest` function to estimate the impulse response. Truncate the estimate to 100 points. Use `impz` to determine the true impulse response of the system. Plot the true impulse response and the estimated impulse response for comparison.

```
irEstimate = impzest(excitation,nrec);
irEstimate = irEstimate(1:101);

irTrue = impz(B,A,101);
plot(0:100,irEstimate, ...
     0:100,irTrue,'ro')

legend('True impulse response','Estimated impulse response')
```



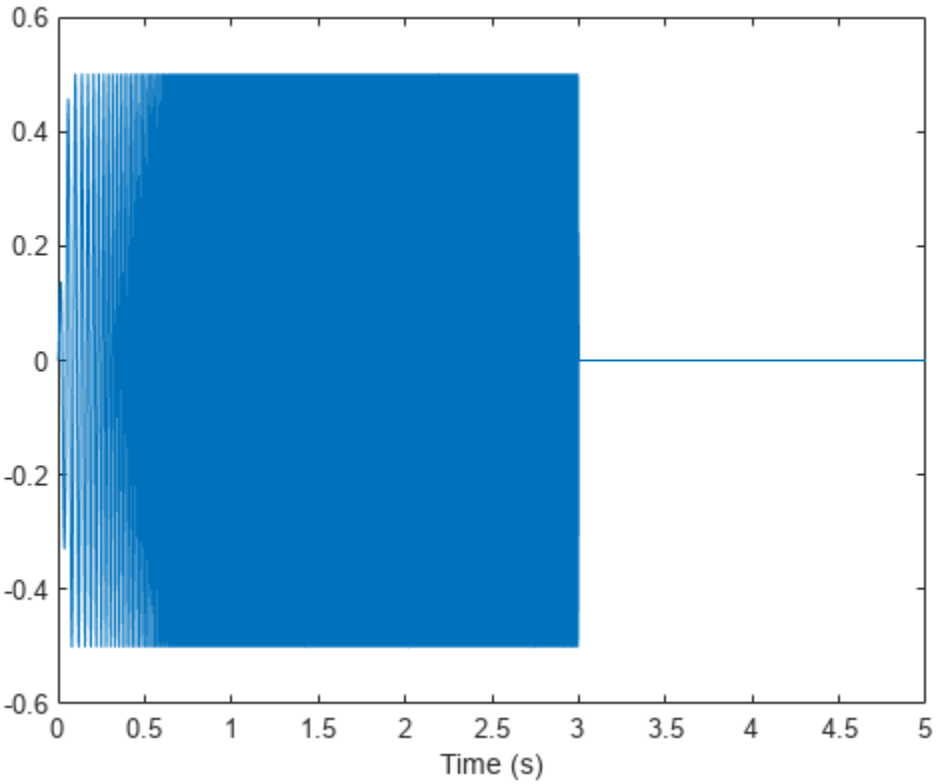
Generate ESS Signal

Generate an exponential swept sine (ESS) signal with a 3-second sweep that goes from 20 Hz to 20 kHz, and ends with a 2-second silence. Specify the sample rate as 48 kHz.

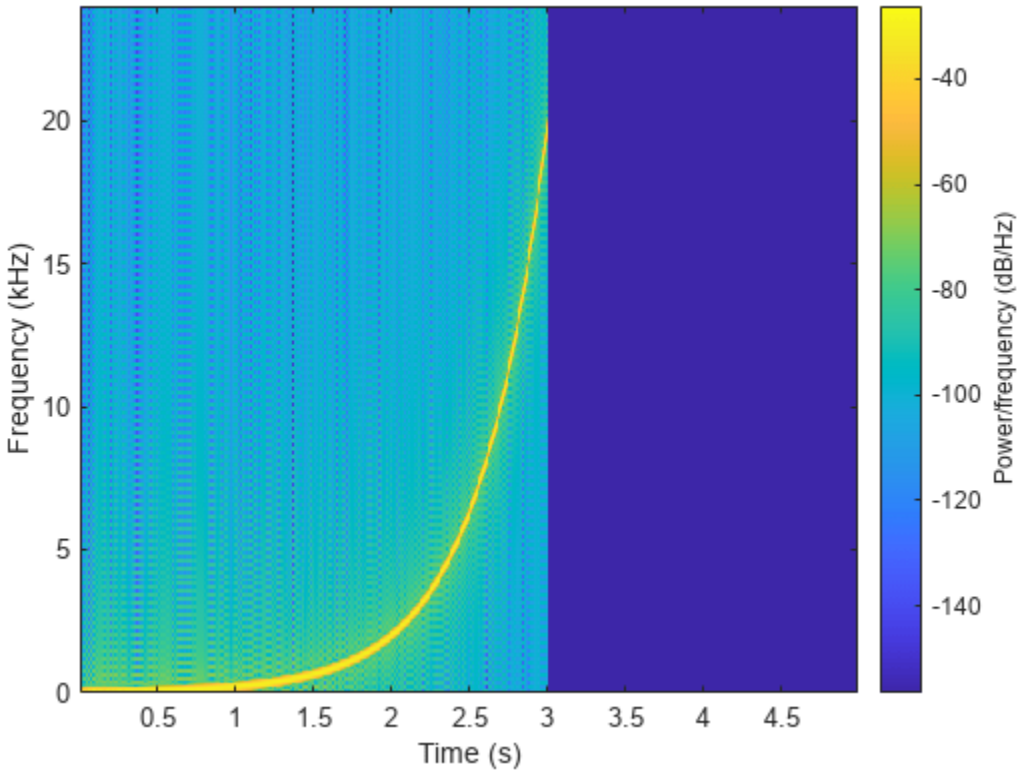
```
fs = 48e3;
excitation = sweeptone(3,2,fs,'SweepFrequencyRange',[20 20e3]);
```

Visualize the excitation in time and time-frequency.

```
t = (0:numel(excitation)-1)/fs;
plot(t,excitation)
xlabel('Time (s)')
```




```
spectrogram(excitation,512,0,1024,fs,'yaxis')
```



Input Arguments

`swDur` — Duration of exponential swept sine signal (s)

6 (default) | scalar in the range [0.5,60]

Duration of exponential swept sine signal in seconds, specified as a scalar in the range [0.5,60].

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

`silDur` — Duration of silence after exponential swept sine signal (s)

4 (default) | positive scalar

Duration of silence after exponential swept sine, specified as a positive scalar.

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

`fs` — Sample rate (Hz)

`44100` (default) | positive scalar

Sample rate in Hz, specified as a positive scalar.

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

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `'ExcitationLevel',-5`

`ExcitationLevel` — Level of excitation signal to generate (dB)

`-6` (default) | scalar in the range `[-42,0]`

Level of the excitation signal to generate in dB, specified as a scalar in the range `[-42,0]`.

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

`SweepFrequencyRange` — Range of sweep frequency (Hz)

`[10 22000]` | two-element positive row vector

Range of sweep frequency in Hz, specified as a two-element row vector. The sweep frequency range can be specified low to high or high to low. That is, `[10 22000]` and `[22000 10]` are both valid inputs. The largest value of the sweep frequency range must be less than or equal to `fs`/2.

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

Output Arguments

`excitation` — Excitation signal

column vector

Excitation signal generated using the ESS technique, returned as a column vector. The length of the column vector is approximately (`swDur+silDur`)*`fs` samples.

Data Types: `double`

Version History

Introduced in R2018b

R2022b: Exponential swept sine supports longer duration

Behavior changed in R2022b

The duration of the exponential swept sine signal, specified by the `swDur` argument, can be a maximum of 60 seconds. Previously, the combination of the exponential swept sine signal duration and the duration of the following silence could not exceed 15 seconds.

References

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

Extended Capabilities

C/C++ Code Generation

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

See Also

`impzest` | `mls` | **Impulse Response Measurer**

interpolateHRTF

3-D head-related transfer function (HRTF) interpolation

Syntax

```
interpolatedHRTF = interpolateHRTF(HRTF,sourcePositions,
desiredSourcePositions)
interpolatedHRTF = interpolateHRTF( ___ ,Name,Value)
```

Description

`interpolatedHRTF = interpolateHRTF(HRTF,sourcePositions, desiredSourcePositions)` returns the interpolated head-related transfer function (HRTF) at the desired position.

`interpolatedHRTF = interpolateHRTF( ___ ,Name,Value)` specifies options using one or more `Name,Value` pair arguments.

Examples

Render 3-D Audio on Headphones

Modify the 3-D audio image of a sound file by filtering it through a head-related transfer function (HRTF). Set the location of the sound source by specifying the desired azimuth and elevation.

```
load 'ReferenceHRTF.mat' hrtfData sourcePosition

hrtfData = permute(double(hrtfData),[2,3,1]);

sourcePosition = sourcePosition(:,[1,2]);
```

Calculate the head-related impulse response (HRIR) using the VBAP algorithm at a desired source position. Separate the output, `interpolatedIR`, into the impulse responses for the left and right ears.

```
desiredAz = 110;
desiredEl = -45;
desiredPosition = [desiredAz desiredEl];

interpolatedIR  = interpolateHRTF(hrtfData,sourcePosition,desiredPosition, ...
                                  "Algorithm","VBAP");

leftIR = squeeze(interpolatedIR(:,1,:))';
rightIR = squeeze(interpolatedIR(:,2,:))';
```

Create a `dsp.AudioFileReader` object to read in a file frame by frame. Create an `audioDeviceWriter` object to play audio to your sound card frame by frame. Create two `dsp.FIRFilter` objects and specify the filter coefficients using the head-related transfer function interpolated impulse responses.

```
fileReader = dsp.AudioFileReader('RockDrums-48-stereo-11secs.mp3');
deviceWriter = audioDeviceWriter('SampleRate',fileReader.SampleRate);

leftFilter = dsp.FIRFilter('Numerator',leftIR);
rightFilter = dsp.FIRFilter('Numerator',rightIR);
```

In an audio stream loop:

1. Read in a frame of audio data.
2. Feed the stereo audio data through the left and right HRIR filters, respectively.
3. Concatenate the left and right channels and write the audio to your output device.

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

    leftChannel = leftFilter(audioIn(:,1));
    rightChannel = rightFilter(audioIn(:,2));

    deviceWriter([leftChannel,rightChannel]);
end
```

As a best practice, release your System objects when complete.

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

Model Moving Source Using HRIR Filtering

Create arrays of head-related impulse responses corresponding to desired source positions. Filter mono input to model a moving source.

Load the ARI HRTF dataset. Cast the `hrtfData` to type double, and reshape it to the required dimensions: (number of source positions)-by-2-by-(number of HRTF samples). Use the first two columns of the `sourcePosition` matrix only, which correspond to the azimuth and elevation of the source in degrees.

```
load 'ReferenceHRTF.mat' hrtfData sourcePosition

hrtfData = permute(double(hrtfData),[2,3,1]);

sourcePosition = sourcePosition(:,[1,2]);
```

Specify the desired source positions and then calculate the HRTF at these locations using the `interpolateHRTF` function. Separate the output, `interpolatedIR`, into the impulse responses for the left and right ears.

```
desiredAz = [-120;-60;0;60;120;0;-120;120];
desiredEl = [-90;90;45;0;-45;0;45;45];
desiredPosition = [desiredAz desiredEl];

interpolatedIR  = interpolateHRTF(hrtfData,sourcePosition,desiredPosition);

leftIR = squeeze(interpolatedIR(:,1,:));
rightIR = squeeze(interpolatedIR(:,2,:));
```

Create an audio file sampled at 48 kHz for compatibility with the HRTF dataset.

```
desiredFs = 48e3;
[audio,fs] = audioread('Counting-16-44p1-mono-15secs.wav');
audio = 0.8*resample(audio,desiredFs,fs);
audiowrite('Counting-16-48-mono-15secs.wav',audio,desiredFs);
```

Create a `dsp.AudioFileReader` object to read in a file frame by frame. Create an `audioDeviceWriter` object to play audio to your sound card frame by frame. Create two `dsp.FIRFilter` objects with `NumeratorSource` set to `Input port`. Setting `NumeratorSource` to `Input port` enables you to modify the filter coefficients while streaming.

```
fileReader = dsp.AudioFileReader('Counting-16-48-mono-15secs.wav');
deviceWriter = audioDeviceWriter('SampleRate',fileReader.SampleRate);

leftFilter = dsp.FIRFilter('NumeratorSource','Input port');
rightFilter = dsp.FIRFilter('NumeratorSource','Input port');
```

In an audio stream loop:

1. Read in a frame of audio data.
2. Feed the audio data through the left and right HRIR filters.
3. Concatenate the left and right channels and write the audio to your output device. If you have a stereo output hardware, such as headphones, you can hear the source shifting position over time.
4. Modify the desired source position in 2-second intervals by updating the filter coefficients.

```
durationPerPosition = 2;
samplesPerPosition = durationPerPosition*fileReader.SampleRate;
samplesPerPosition = samplesPerPosition - rem(samplesPerPosition,fileReader.SamplesPerFrame);

sourcePositionIndex = 1;
samplesRead = 0;
while ~isDone(fileReader)
    audioIn = fileReader();
    samplesRead = samplesRead + fileReader.SamplesPerFrame;

    leftChannel = leftFilter(audioIn,leftIR(sourcePositionIndex,:));
    rightChannel = rightFilter(audioIn,rightIR(sourcePositionIndex,:));

    deviceWriter([leftChannel,rightChannel]);

    if mod(samplesRead,samplesPerPosition) == 0
        sourcePositionIndex = sourcePositionIndex + 1;
    end
end
```

As a best practice, release your System objects when complete.

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

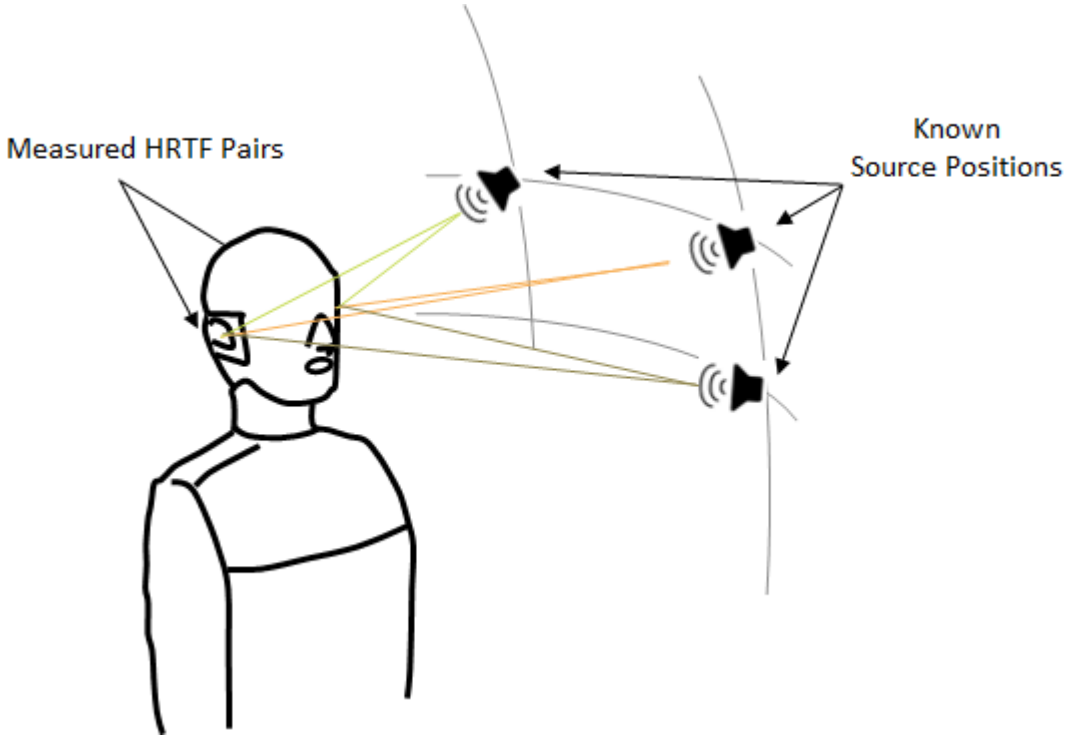
Input Arguments

HRTF — HRTF values measured at source positions

N-by-2-by-*M* array

HRTF values measured at the source positions, specified as a *N*-by-2-by-*M* array.

- *N* -- Number of known HRTF pairs
- *M* -- Number of samples in each known HRTF



If you specify `HRTF` with real numbers, the function assumes that the input represents an impulse response, and *M* corresponds to the length of the impulse response. If you specify `HRTF` with complex numbers, the function assumes that the input represents a transfer function, and *M* corresponds to the number of bins in the frequency response. The output of the `interpolateHRTF` function has the same complexity and interpretation as the input.

Data Types: `single` | `double`
Complex Number Support: Yes

`sourcePositions` — Source positions corresponding to measured HRTF values
N-by-2 matrix

Source positions corresponding to measured HRTF values, specified as a *N*-by-2 matrix. *N* is the number of known HRTF pairs. The two columns correspond to the azimuth and elevation of the source in degrees, respectively.

Azimuth must be in the range [−180,360]. You can use the −180 to 180 convention or the 0 to 360 convention.

Elevation must be in the range [−90,180]. You can use the −90 to 90 convention or the 0 to 180 convention.

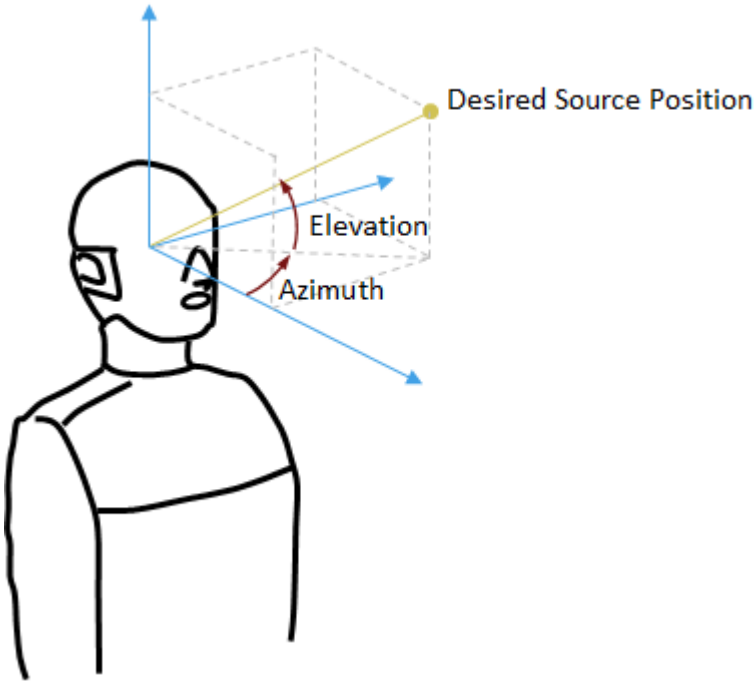
Data Types: `single` | `double`

`desiredSourcePositions` — Desired source positions for HRTF interpolation
P-by-2 matrix

Desired source position for HRTF interpolation, specified as a *P*-by-2 matrix. *P* is the number of desired source positions. The columns correspond to the desired azimuth and elevation of the source in degrees, respectively.

Azimuth must be in the range [−180,360]. You can use the −180 to 180 convention or the 0 to 360 convention.

Elevation must be in the range [−90,180]. You can use the −90 to 90 convention or the 0 to 180 convention.



Data Types: `single` | `double`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `'Algorithm','VBAP'`

`Algorithm` — Interpolation algorithm

`'Bilinear'` (default) | `'VBAP'`

Interpolation algorithm, specified as `"Bilinear"` or `"VBAP"`.

- `Bilinear` –– 3-D bilinear interpolation, as specified by [1].
- `VBAP` –– Vector base amplitude panning interpolation, as specified by [2].

Data Types: `char` | `string`

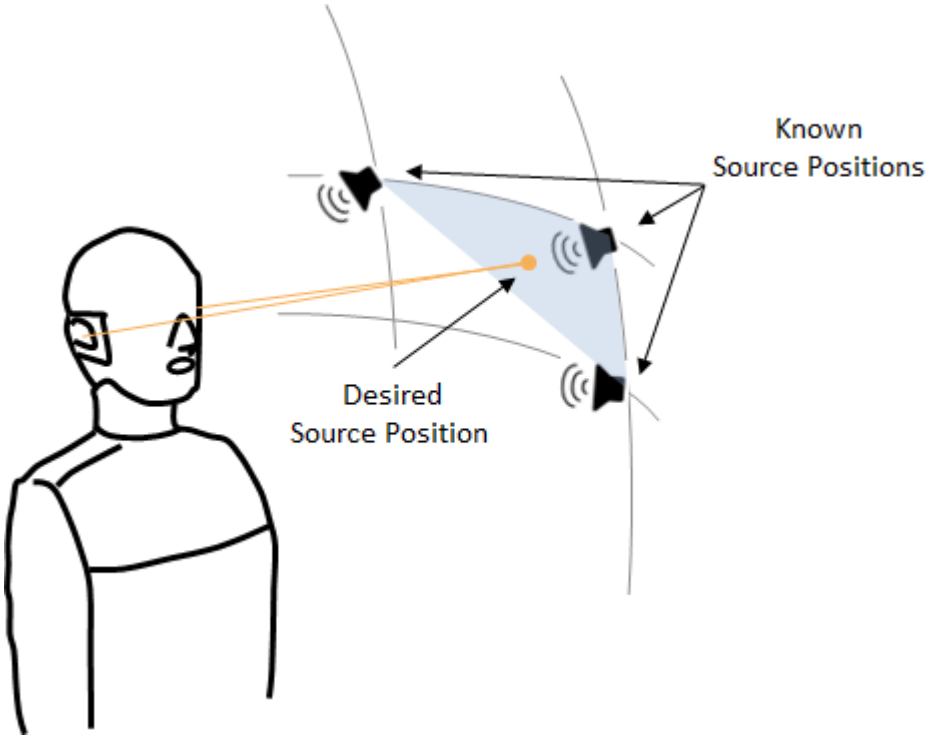
Output Arguments

`interpolatedHRTF` — Interpolated HRTF

P-by-2-by-*M*

Interpolated HRTF, returned as a *P*-by-2-by-*M* array.

- *P* –– Number of desired source positions, specified by the number of rows in the `desiredSourcePositions` input argument.
- *M* –– Number of samples in each known HRTF, specified by the number of pages in the `HRTF` input argument.



`interpolatedHRTF` has the same complexity and interpretation as the input. If you specify the input, `HRTF`, with real numbers, the function assumes that the input represents an impulse response. If you specify the input with complex numbers, the function assumes that the input represents a transfer function.

Data Types: `single` | `double`
Complex Number Support: Yes

Version History

Introduced in R2018b

References

[1] F.P. Freeland, L.W.P. Biscainho and P.S.R. Diniz, "Interpolation of Head-Related Transfer Functions (HRTFS): A multi-source approach." *2004 12th European Signal Processing Conference*. Vienna, 2004, pp. 1761–1764.

[2] Pulkki, Ville. "Virtual Sound Source Positioning Using Vector Based Amplitude Panning." *Journal of Audio Engineering Society*. Vol. 45. Issue 6, pp. 456–466.

Extended Capabilities

C/C++ Code Generation

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

See Also

`dsp.FIRFilter` | `dsp.FrequencyDomainFIRFilter`

impzest

Estimate impulse response of audio system

Syntax

```
ir = impzest(excitation,response)
ir = impzest(excitation,response,Name,Value)
```

Description

`ir = impzest(excitation,response)` returns an estimate of the impulse response (IR) based on the `excitation` and `response`.

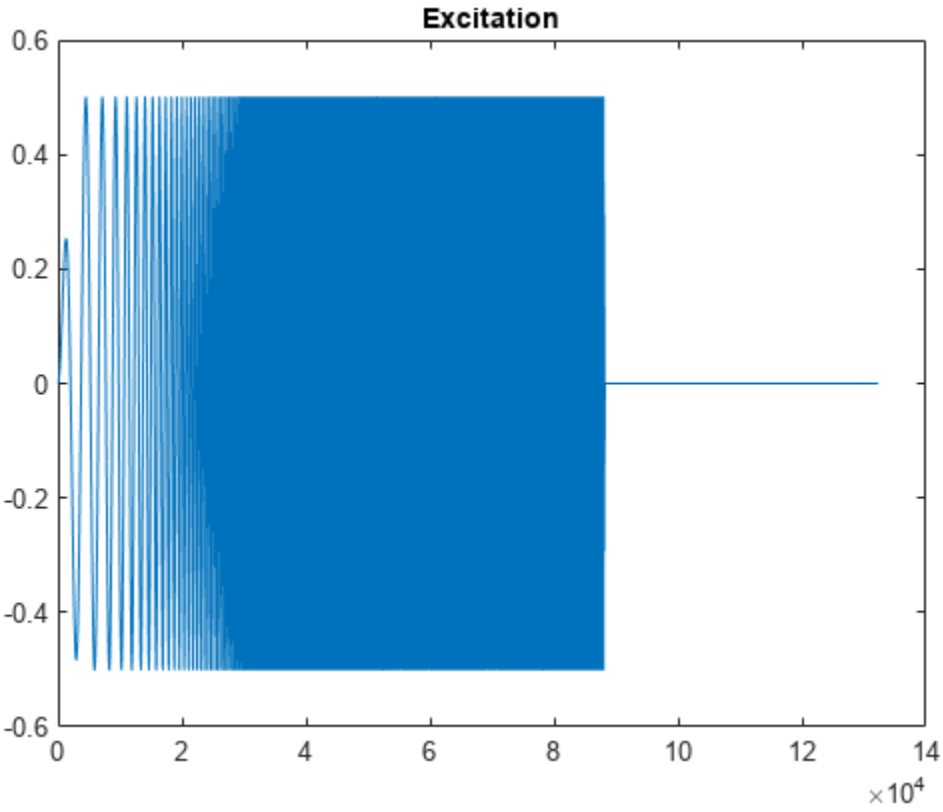
`ir = impzest(excitation,response,Name,Value)` specifies options using one or more `Name,Value` pair arguments.

Examples

Estimate Impulse Response Using Sweep Tone Excitation

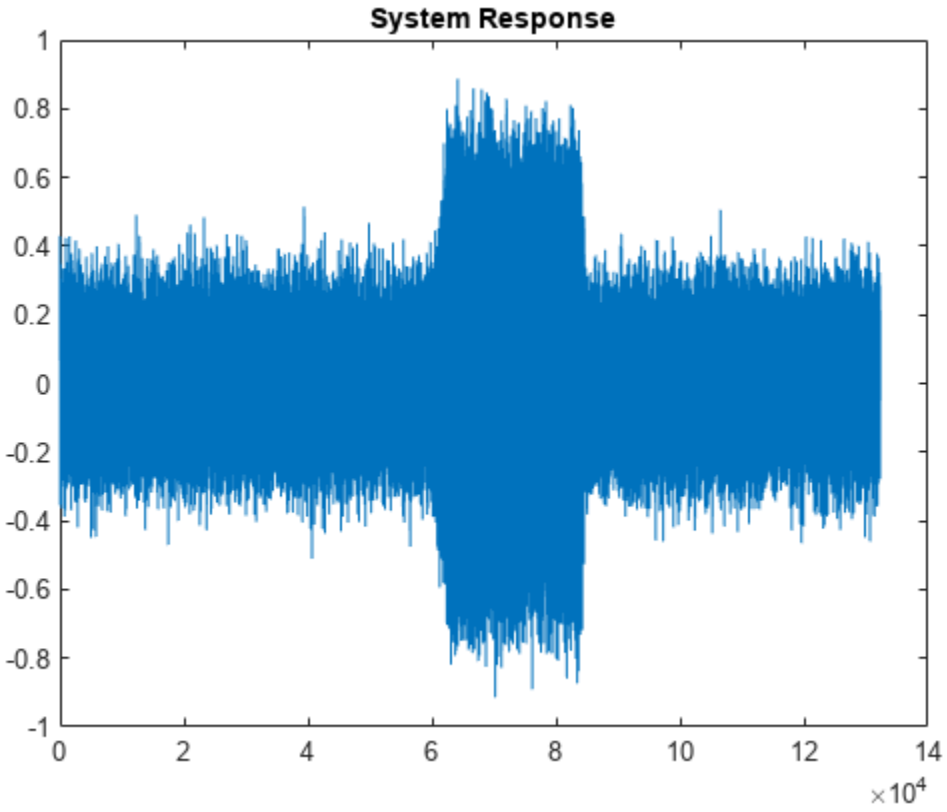
Create a sweep tone excitation signal by using the `sweeptone` function.

```
excitation = sweeptone(2,1,44100);

plot(excitation)
title('Excitation')
```



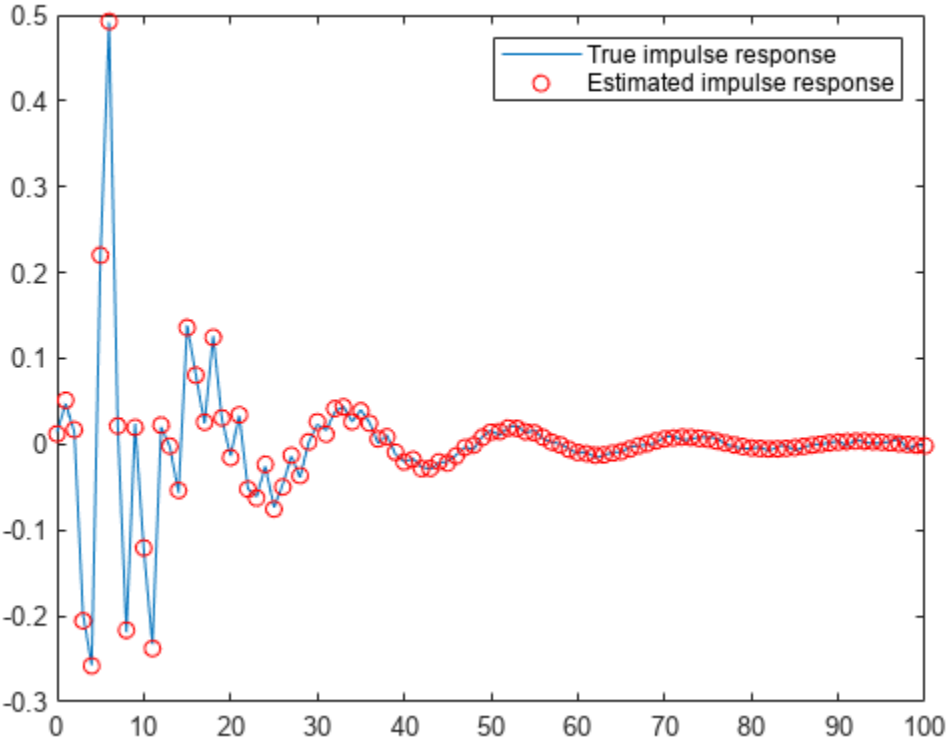
Pass the excitation signal through an infinite impulse response (IIR) filter and add noise to model a real-world recording (system response).

```
[B,A] = butter(10,[.1 .7]);
rec = filter(B,A,excitation);
nrec = rec + 0.12*randn(size(rec));

plot(nrec)
title('System Response')
```



Pass the excitation signal and the system response to the `impzest` function to estimate the impulse response. Truncate the estimate to 100 points. Use `impz` to determine the true impulse response of the system. Plot the true impulse response and the estimated impulse response for comparison.

```
irEstimate = impzest(excitation,nrec);
irEstimate = irEstimate(1:101);

irTrue = impz(B,A,101);
plot(0:100,irEstimate, ...
     0:100,irTrue,'ro')

legend('True impulse response','Estimated impulse response')
```



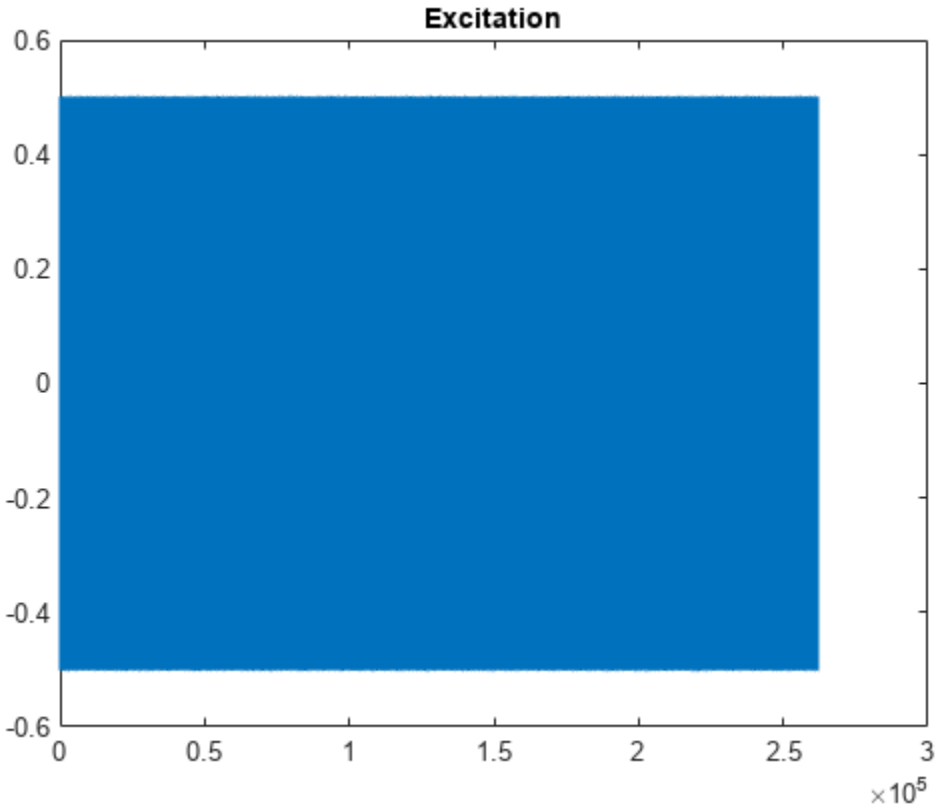
Estimate Impulse Response Using MLS Excitation

Use `audioread` to read in an impulse response recording. Create a `dsp.FrequencyDomainFIRFilter` object to perform frequency domain filtering using the known impulse response.

```
[irKnown,fs] = audioread('ChurchImpulseResponse-16-44p1-mono-5secs.wav');
systemModel = dsp.FrequencyDomainFIRFilter(irKnown');
```

Create an MLS excitation signal by using the `mls` function. The MLS excitation signal must be longer than the impulse response. Note that the length of the MLS excitation is extended to the next power of two minus one.

```
excitation = mls(numel(irKnown)+1);

plot(excitation)
title('Excitation')
```

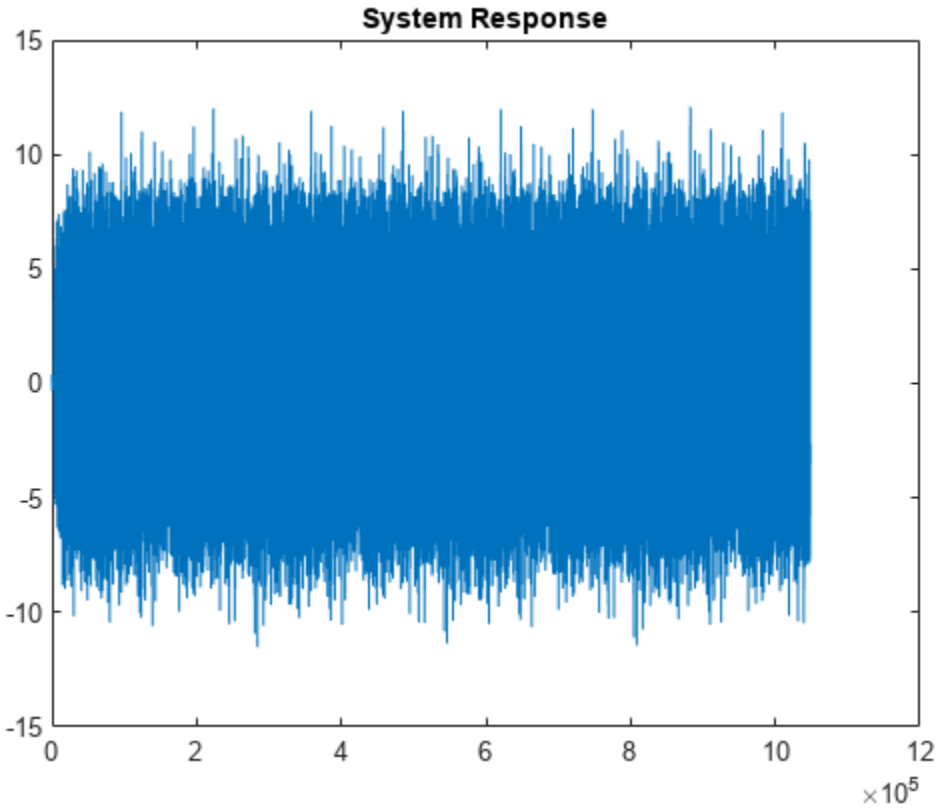


Replicate the excitation signal four times to measure the average of three measurements. The recording of the first MLS sequence does include all the impulse response information, so `impzest` discards it as a warmup run. Pad the excitation signal with zeros to account for the filter latency.

```
numRuns = 4;
excrep = repmat(excitation,numRuns,1);
excrep = [excrep;zeros(numel(irKnown)+1,1)];
```

Pass the excitation signal through the known filter and then add noise to model a real-word recording (system response). Cut the delay introduced at the beginning by the filter.

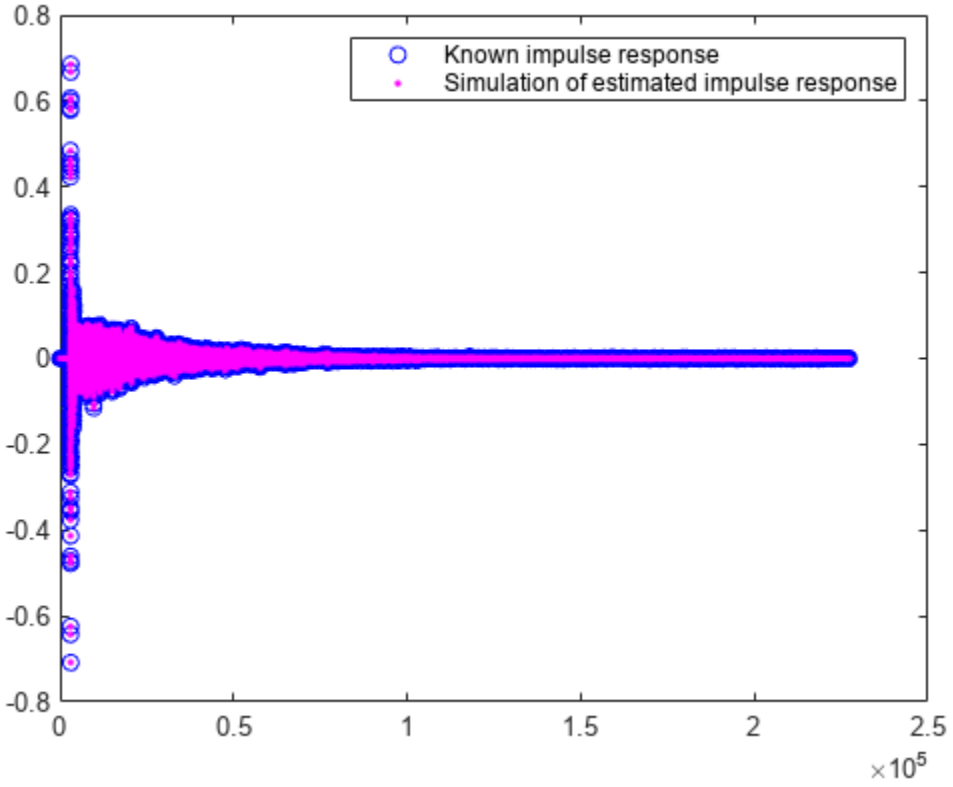
```
rec = systemModel(excrep);
rec = rec + 0.1*randn(size(rec));

rec = rec(numel(irKnown)+2:end,:);

plot(rec)
title('System Response')
```



In a real-world scenario, the MLS sequence is played back in the system under test while recording. The recording would be cut so that it begins at the moment the MLS sequence is picked-up and truncated to last the duration of the repeated sequence.

Pass the excitation signal and the system response to the `impzest` function to estimate the impulse response. Plot the known impulse response and the simulation of the estimated impulse response for comparison.

```
irEstimate = impzest(excitation,rec);

samples = 1:numel(irKnown);
plot(samples,irEstimate(samples),'bo', ...
     samples,irKnown(samples),'m.')

legend('Known impulse response','Simulation of estimated impulse response')
```



Input Arguments

`excitation` — Single period of excitation signal input to audio system

column vector

Single period of excitation signal input to audio system, specified as a column vector.

You can generate excitation signals by using `mls` (maximum length sequence) or `sweeptone` (exponential sine sweep).

Data Types: `single` | `double`

`response` — Recorded signal output from audio system

column vector | matrix

Recorded signal output from audio system, specified as a column vector or matrix. If specified as a matrix, each column of the matrix is treated as an independent channel.

Data Types: `single` | `double`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `'WarmupRuns',2`

`WarmupRuns` — Number of warmup runs in response
nonnegative integer

Number of warmup runs in the response, specified as a nonnegative integer. The `impzest` function estimates the impulse response after discarding the specified number of warmup runs from the `response`.

The default number of warmup runs depends on whether the excitation signal was generated using the `mls` or `sweeptone` function:

- `mls` –– 1
- `sweeptone` –– 0

Data Types: `single` | `double`

Output Arguments

`ir` — Estimate of the impulse response of an audio system
column vector | matrix

Estimate of the impulse response of an audio system, returned as a column vector or matrix. The size of `ir` is *L*-by-*C*, where:

- *L* –– MLS length or duration of sweep tone silence
- *C* –– Number of columns (channels) in the response signal

Data Types: `single` | `double`

Version History

Introduced in R2018b

References

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

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

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

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also

`sweeptone` | `mls` | **Impulse Response Measurer**

mididevinfo

MIDI device information

Syntax

```
mididevinfo
deviceInformation = mididevinfo
```

Description

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

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

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

Examples

Display MIDI Device Connections

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

```
mididevinfo
```

```
  MIDI devices available:
  ID  Direction  Interface   Name
   0   output    MMSystem   'Microsoft MIDI Mapper'
   1    input    MMSystem   'BCF2000'
   2    input    MMSystem   'MIDIIN2 (BCF2000)'
   3   output    MMSystem   'Microsoft GS Wavetable Synth'
   4   output    MMSystem   'BCF2000'
   5   output    MMSystem   'MIDIOUT2 (BCF2000)'
   6   output    MMSystem   'MIDIOUT3 (BCF2000)'
```

Return Structure of MIDI Device Connections

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

```
deviceInformation = mididevinfo
```

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

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

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

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

Output Arguments

`deviceInformation` — Description of available devices

struct

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

Data Types: `struct`

Version History

Introduced in R2018a

See Also

`parameterTuner` | **Audio Test Bench** | `mididevice` | `midimsg` | `midireceive` | `midisend`

Topics

“MIDI Device Interface”

External Websites

MIDI Manufacturers Association

pitch

Estimate fundamental frequency of audio signal

Syntax

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

Description

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

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

`[f0,loc] = pitch( ___ )` returns the locations, `loc`, associated with fundamental frequency estimates. You can specify an input combination from any of the previous syntaxes.

`pitch( ___ )` with no output arguments plots the estimated pitch against time.

Examples

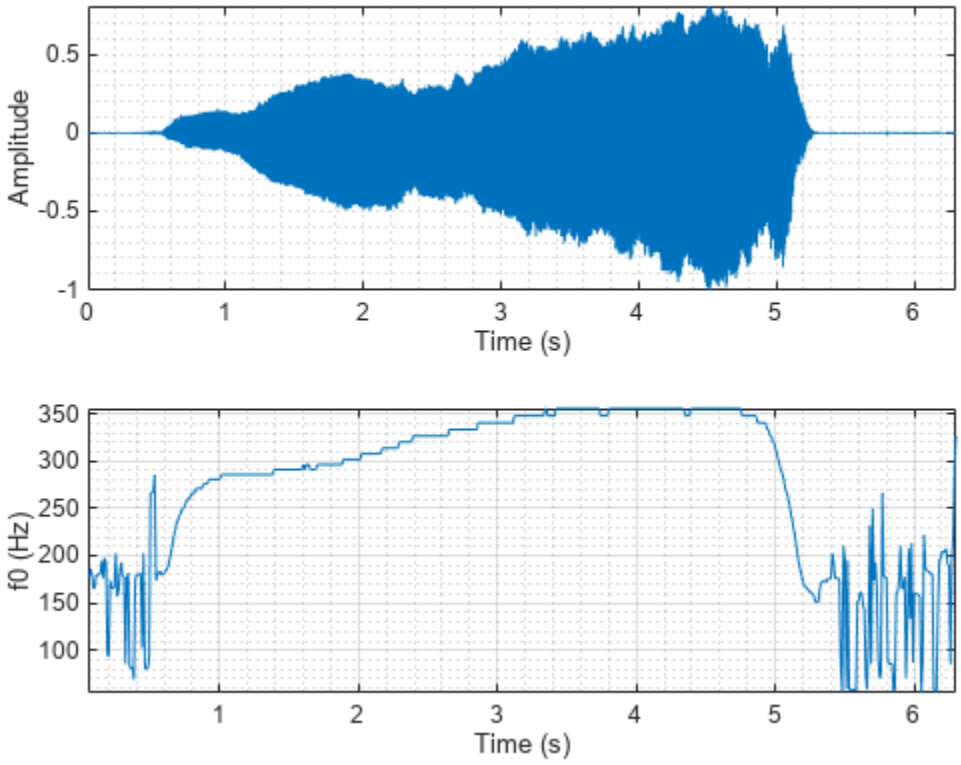
Estimate Pitch

Read in an audio signal. Call `pitch` to estimate the fundamental frequency over time.

```
[audioIn,fs] = audioread("Hey-16-mono-6secs.ogg");

f0 = pitch(audioIn,fs);
```

Listen to the audio signal and plot the signal and pitch. The `pitch` function estimates the fundamental frequency over time, but the estimate is only valid for regions that are harmonic.

```
sound(audioIn,fs)

tiledlayout(2,1)

nexttile
t = (0:length(audioIn)-1)/fs;
plot(t,audioIn)
xlabel("Time (s)")
ylabel("Amplitude")
grid minor
axis tight

nexttile
pitch(audioIn,fs)
```



Estimate Pitch For Singing Voice

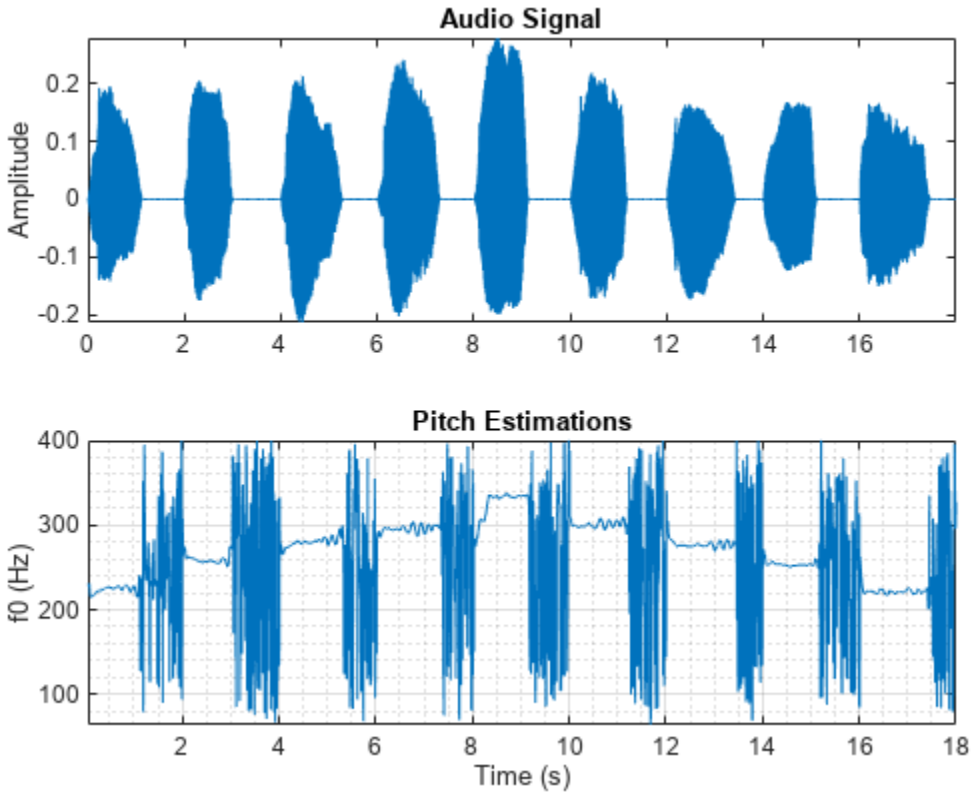
Read in an audio signal and extract the pitch.

```
[x,fs] = audioread("SingingAMajor-16-mono-18secs.ogg");
t = (0:size(x,1)-1)/fs;

winLength = round(0.05*fs);
overlapLength = round(0.045*fs);
[f0,idx] = pitch(x,fs,Method="SRH",WindowLength=winLength,OverlapLength=overlapLength);
tf0 = idx/fs;
```

Listen to the audio and plot the audio and pitch estimations.

```
sound(x,fs)

figure
tiledlayout(2,1)

nexttile
plot(t,x)
ylabel("Amplitude")
title("Audio Signal")
axis tight
```

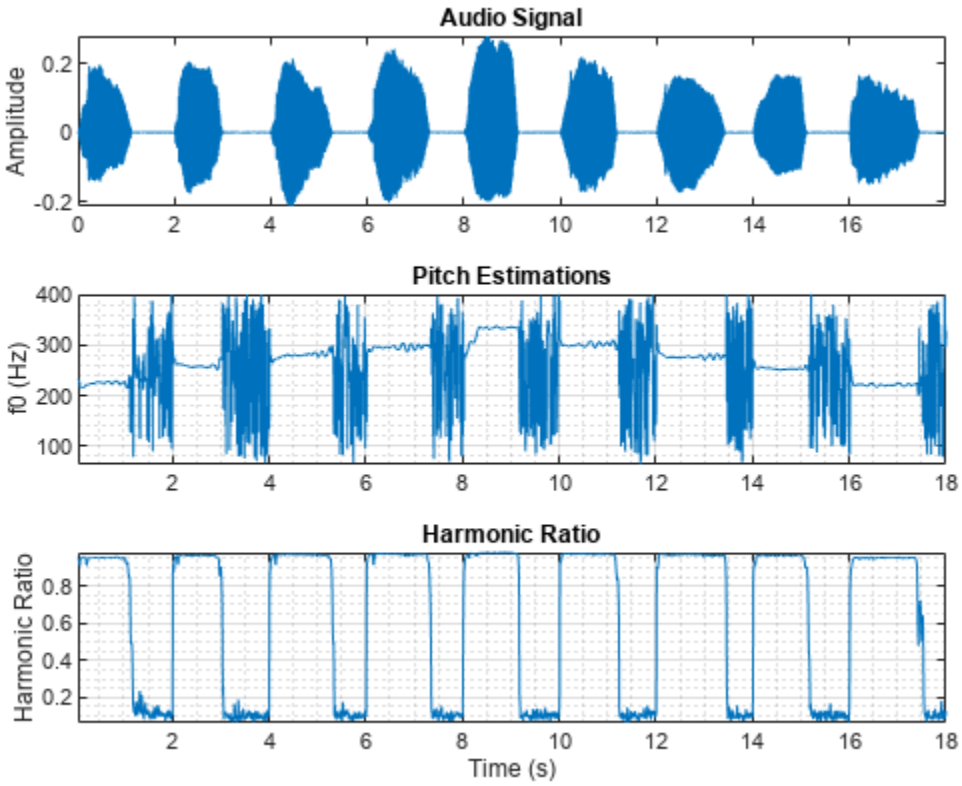
```
nexttile
pitch(x,fs,Method="SRH",WindowLength=winLength,OverlapLength=overlapLength)
title("Pitch Estimations")
```



The `pitch` function estimates the pitch for overlapped analysis windows. The pitch estimates are only valid if the analysis window has a harmonic component. Call the `harmonicRatio` function using the same window and overlap length used for pitch detection. Plot the audio, pitch, and harmonic ratio.

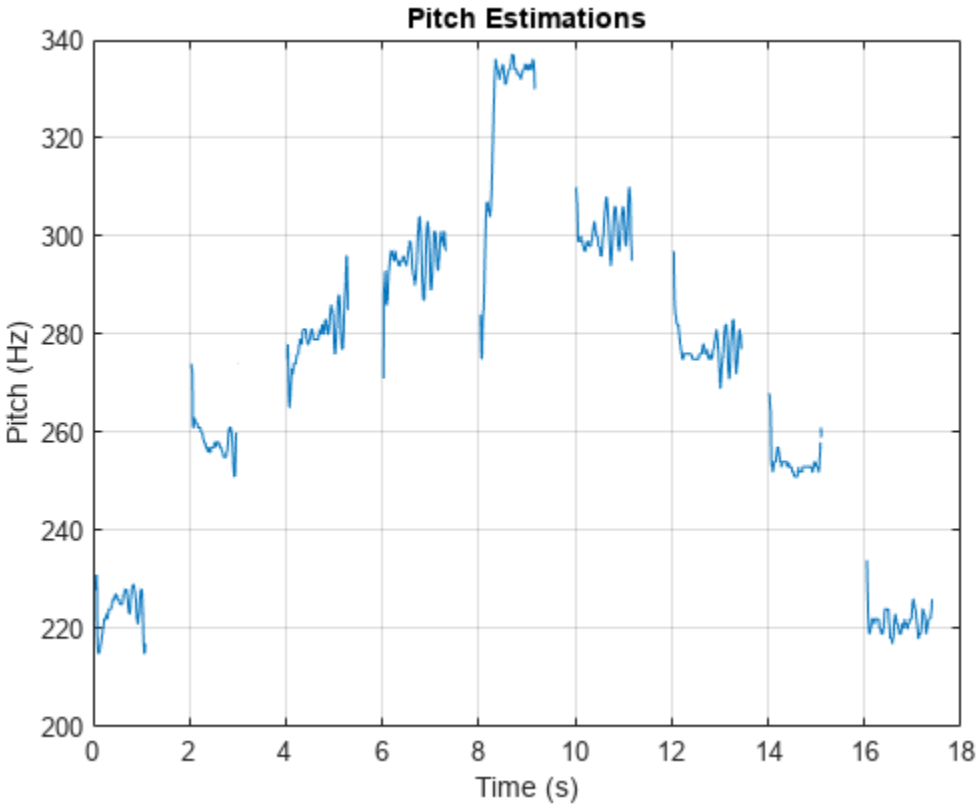
```
hr = harmonicRatio(x,fs,Window=hamming(winLength,"periodic"),OverlapLength=overlapLength);

figure
tiledlayout(3,1)

nexttile
plot(t,x)
ylabel("Amplitude")
title("Audio Signal")
axis tight

nexttile
pitch(x,fs,Method="SRH",WindowLength=winLength,OverlapLength=overlapLength)
title("Pitch Estimations")
xlabel("")

nexttile
```

```
harmonicRatio(x,fs,Window=hamming(winLength,"periodic"),OverlapLength=overlapLength)
title("Harmonic Ratio")
```



Use the harmonic ratio as the threshold for valid pitch decisions. If the harmonic ratio is less than the threshold, set the pitch decision to `NaN`. Plot the results.

```
threshold = 0.9;
f0(hr < threshold) = nan;

figure
plot(tf0,f0)
xlabel("Time (s)")
ylabel("Pitch (Hz)")
title("Pitch Estimations")
grid on
```



Compare Pitch of Two Voices

Read in an audio signal of a female voice saying "volume up" five times. Listen to the audio.

```
[femaleVoice,fs] = audioread("FemaleVolumeUp-16-mono-11secs.ogg");
sound(femaleVoice,fs)
```

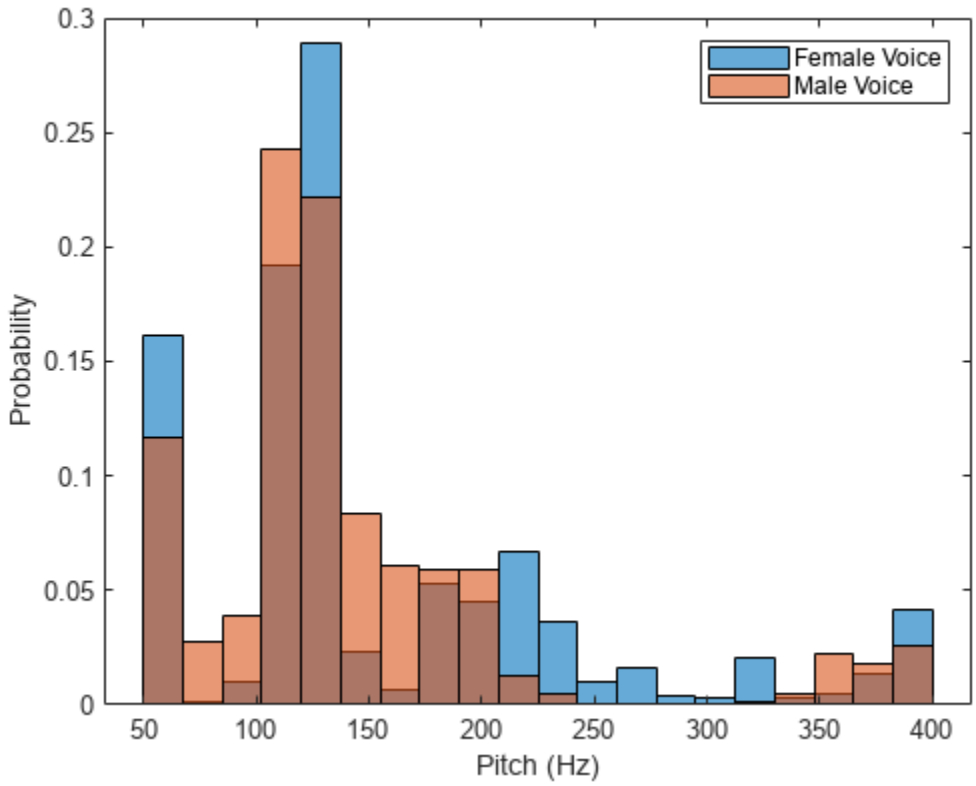
Read in an audio signal of a male voice saying "volume up" five times. Listen to the audio.

```
maleVoice = audioread("MaleVolumeUp-16-mono-6secs.ogg");
sound(maleVoice,fs)
```

Extract the pitch from both the female and male recordings. Plot histograms of the pitch estimations for the male and female audio recordings. The histograms have a similar shape. This is because the pitch decisions contain results for unvoiced speech and regions of silence.

```
f0Female = pitch(femaleVoice,fs);
f0Male = pitch(maleVoice,fs);

figure
numBins = 20;
histogram(f0Female,numBins,Normalization="probability");
hold on
histogram(f0Male,numBins,Normalization="probability");
legend("Female Voice","Male Voice")
```

```
xlabel("Pitch (Hz)")
ylabel("Probability")
hold off
```



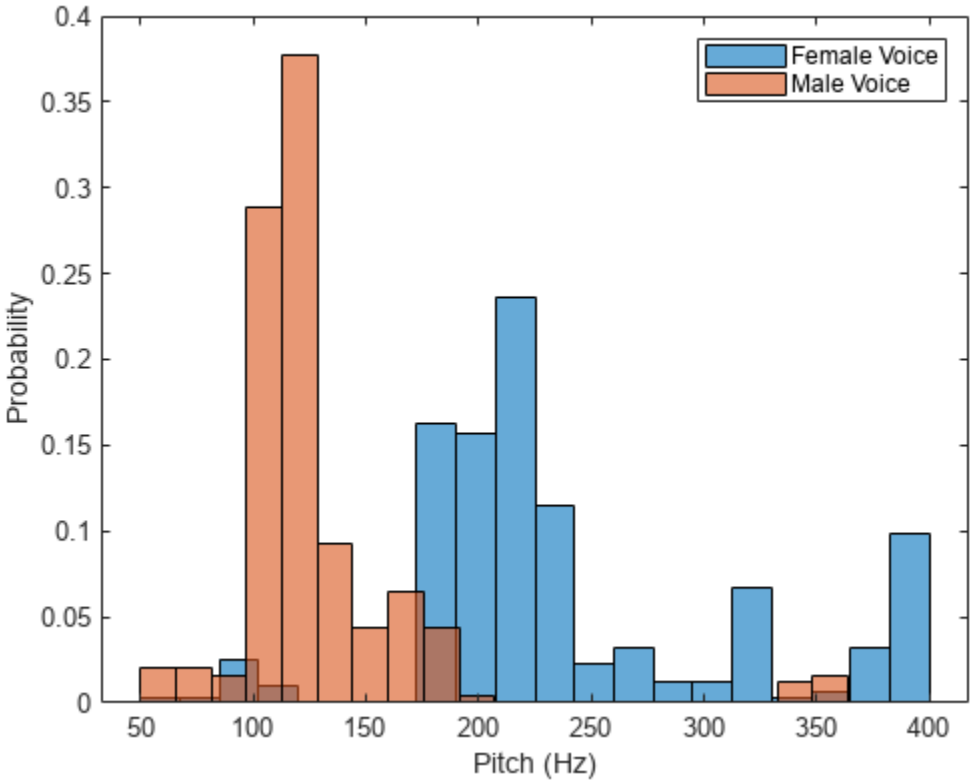
Use the `detectSpeech` function to isolate regions of speech in the audio signal and then extract pitch from only those speech regions.

```
speechIndices = detectSpeech(femaleVoice,fs);
f0Female = [];
for ii = 1:size(speechIndices,1)
    speechSegment = femaleVoice(speechIndices(ii,1):speechIndices(ii,2));
    f0Female = [f0Female;pitch(speechSegment,fs)];
end

speechIndices = detectSpeech(maleVoice,fs);
f0Male = [];
for ii = 1:size(speechIndices,1)
    speechSegment = maleVoice(speechIndices(ii,1):speechIndices(ii,2));
    f0Male = [f0Male;pitch(speechSegment,fs)];
end
```

Plot histograms of the pitch estimations for the male and female audio recordings. The pitch distributions now appear as expected.

```
figure
histogram(f0Female,numBins,Normalization="probability");
hold on
histogram(f0Male,numBins,Normalization="probability");
```

```
legend("Female Voice","Male Voice")
xlabel("Pitch (Hz)")
ylabel("Probability")
```



Estimate Pitch of Musical Signal Using Nondefault Parameters

Load an audio file of the Für Elise introduction and the sample rate of the audio.

```
load FurElise.mat song fs
sound(song,fs)
```

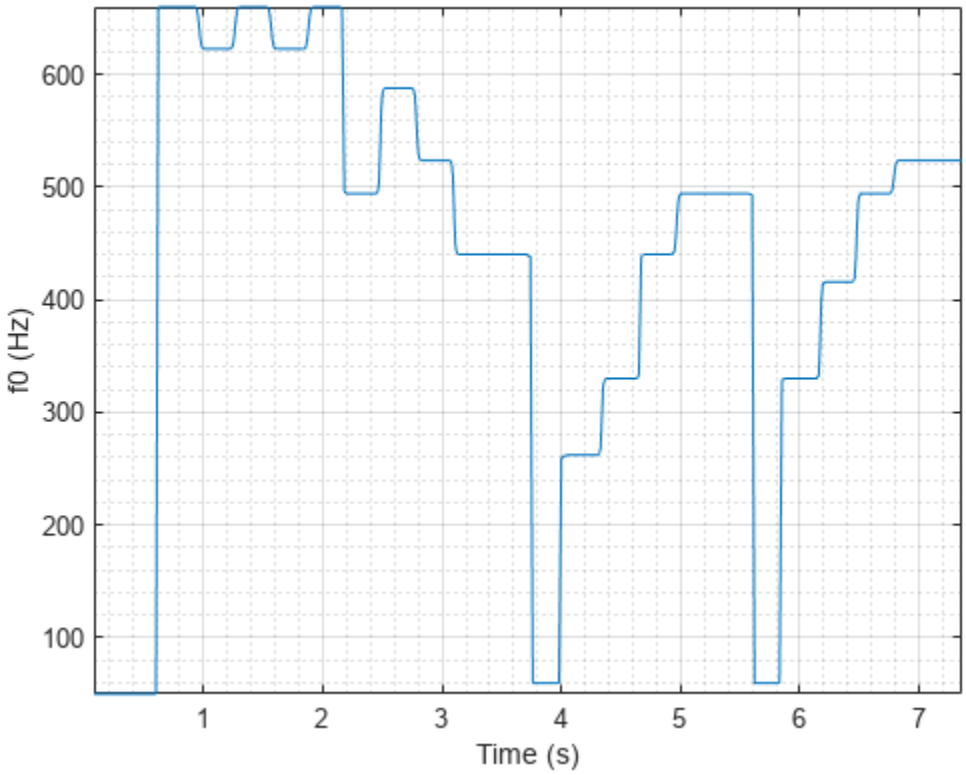
Call the `pitch` function using the pitch estimate filter (PEF), a search range of 50 to 800 Hz, a window duration of 80 ms, an overlap duration of 70 ms, and a median filter length of 10.

```
method = PEF;
range = [ 50 , 800 ]; % hertz
winDur = 0.08 ; % seconds
overlapDur = 0.07 ; % seconds
medFiltLength = 10 ; % frames
```

```
winLength = round(winDur*fs);
overlapLength = round(overlapDur*fs);
[f0,loc] = pitch(song,fs, ...
    Method=method, ...
    Range=range, ...
    WindowLength=winLength, ...
    OverlapLength=overlapLength, ...
    MedianFilterLength=medFiltLength);
```

Plot the estimated pitch against time.

```
pitch(song,fs, ...
    Method=method, ...
    Range=range, ...
    WindowLength=winLength, ...
    OverlapLength=overlapLength, ...
    MedianFilterLength=medFiltLength);
```



Determine Pitch Contour of Streaming Audio

Create a `dsp.AudioFileReader` object to read in audio frame-by-frame.

```
fileReader = dsp.AudioFileReader("SingingAMajor-16-mono-18secs.ogg");
```

Create a `voiceActivityDetector` object to detect the presence of voice in streaming audio.

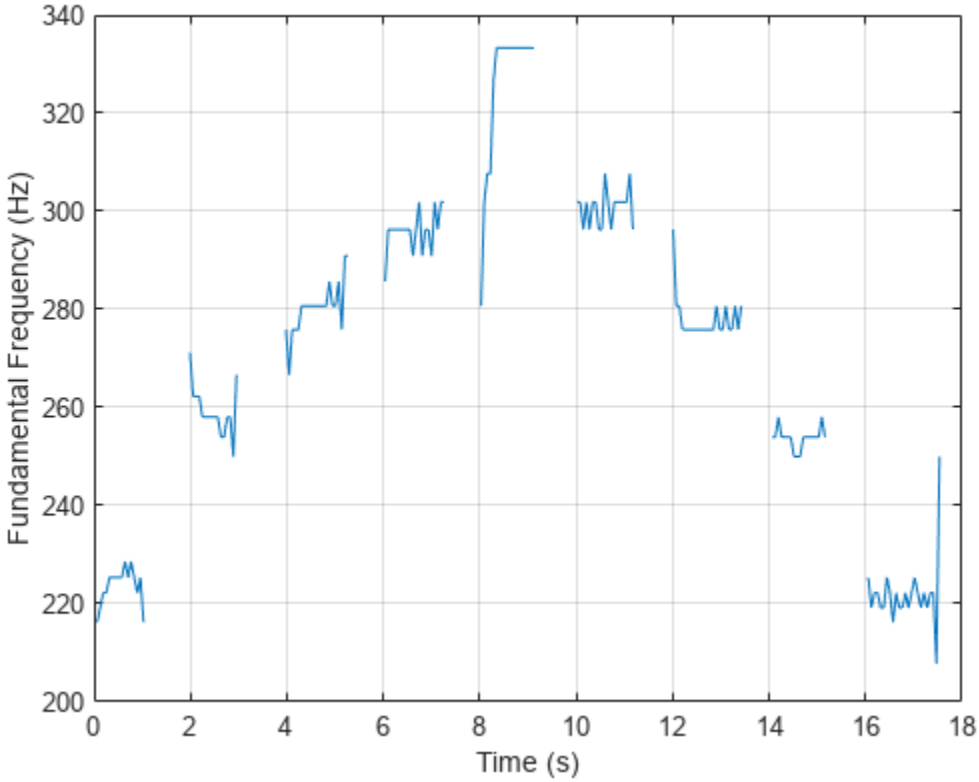
```
VAD = voiceActivityDetector;
```

While there are unread samples, read from the file and determine the probability that the frame contains voice activity. If the frame contains voice activity, call `pitch` to estimate the fundamental frequency of the audio frame. If the frame does not contain voice activity, declare the fundamental frequency as `NaN`.

```
f0 = [];
while ~isDone(fileReader)
    x = fileReader();

    if VAD(x) > 0.99
        decision = pitch(x,fileReader.SampleRate, ...
            WindowLength=size(x,1), ...
            OverlapLength=0, ...
            Range=[200,340]);
    else
        decision = NaN;
    end
    f0 = [f0;decision];
end
```

Plot the detected pitch contour over time.

```
t = linspace(0,(length(f0)*fileReader.SamplesPerFrame)/fileReader.SampleRate,length(f0));
plot(t,f0)
ylabel("Fundamental Frequency (Hz)")
xlabel("Time (s)")
grid on
```



Compare Pitch Detection Algorithms

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

Prepare Test Signals

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

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

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

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

turbine = audioread('Turbine-16-44p1-mono-22secs.wav');
```

```
testSignals(:,1) = mixSNR(audioIn,turbine,20);
testSignals(:,2) = mixSNR(audioIn,turbine,0);

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

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

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

Run Pitch Detection Algorithms

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

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

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

Compare Gross Pitch Error

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

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

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

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

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

```
- NCF : 0.9 %
- PEF : 0.4 %
- CEP : 8.2 %
- LHS : 8.2 %
- SRH : 6.0 %

GPE (p = 0.20), Noise = Turbine (0 dB).

- NCF : 5.6 %
- PEF : 24.5 %
- CEP : 11.6 %
- LHS : 9.4 %
- SRH : 46.8 %

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

- NCF : 0.9 %
- PEF : 0.0 %
- CEP : 12.9 %
- LHS : 6.9 %
- SRH : 2.6 %

GPE (p = 0.20), Noise = WhiteNoise (0 dB).

- NCF : 0.4 %
- PEF : 0.0 %
- CEP : 23.6 %
- LHS : 7.3 %
- SRH : 1.7 %
```

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

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

```
- NCF : 0.021 (s)
- PEF : 0.066 (s)
- CEP : 0.018 (s)
- LHS : 0.024 (s)
- SRH : 0.062 (s)
```

Input Arguments

`audioIn` — Audio input signal

vector | matrix

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

Data Types: `single` | `double`

`fs` — Sample rate (Hz)

positive scalar

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

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

Data Types: `single` | `double`

Name-Value Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `pitch(audioIn,fs,Range=[50,150],Method="PEF")`

Range — Search range for pitch estimates

`[50,400]` (default) | two-element row vector with increasing positive integer values

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

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

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

Data Types: `single` | `double`

WindowLength — Number of samples in analysis window

`round(fs*0.052)` (default) | integer

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

Data Types: `single` | `double`

OverlapLength — Number of samples of overlap between adjacent analysis windows

`round(fs*0.042)` (default) | integer

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

Data Types: `single` | `double`

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

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

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

Data Types: `char` | `string`

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

Median filter length used to smooth pitch estimates over time, specified as a positive integer. The default, `1`, corresponds to no median filtering. Median filtering is a postprocessing technique used to remove outliers while estimating pitch. The function uses `movmedian` after estimating the pitch using the specified `Method`.

Data Types: `single` | `double`

Output Arguments

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

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

Data Types: `single` | `double`

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

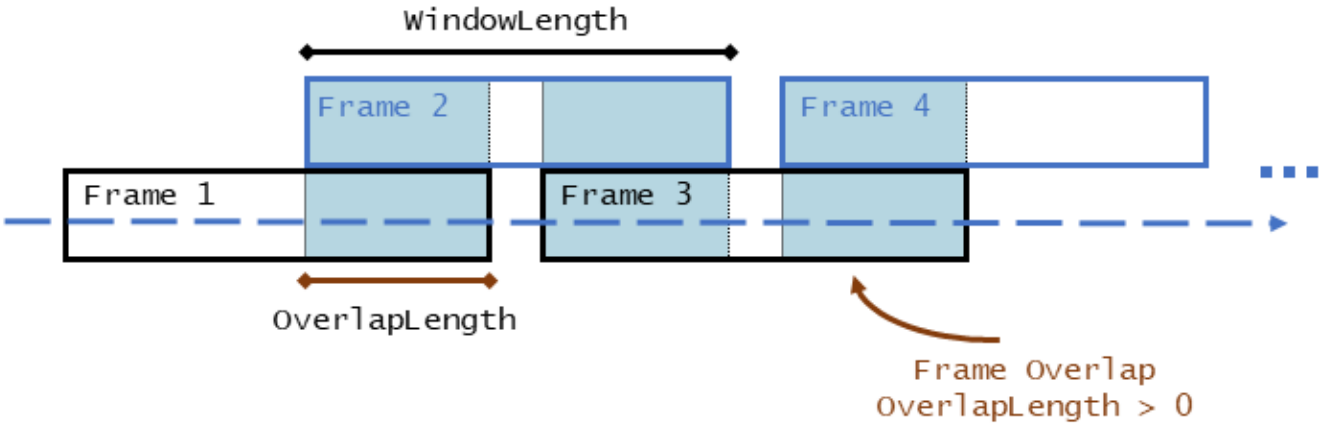
Locations associated with fundamental frequency estimations, returned as a scalar, vector, or matrix the same size as `f0`.

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

Data Types: `single` | `double`

Algorithms

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



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

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

Version History

Introduced in R2018a

References

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

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

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

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

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

Extended Capabilities

C/C++ Code Generation

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

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

See Also

`audioFeatureExtractor` | `mfcc` | `detectSpeech` | `harmonicRatio` | `shiftPitch`

Topics

“Pitch Tracking Using Multiple Pitch Estimations and HMM”
“Speaker Identification Using Pitch and MFCC”
“Delay-Based Pitch Shifter”
“Pitch Shifting and Time Dilation Using a Phase Vocoder in MATLAB”

mfcc

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

Syntax

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

Description

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

`coeffs = mfcc( ___ ,Name=Value)` specifies options using one or more name-value arguments.

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

`[coeffs,delta,deltaDelta,loc] = mfcc( ___ )` also returns the delta, delta-delta, and location of samples corresponding to each window of data. You can specify an input combination from any of the previous syntaxes.

`mfcc( ___ )` with no output arguments plots the mel-frequency cepstral coefficients. Before plotting, the coefficients are normalized to have mean 0 and standard deviation 1.

- If the input is in the time domain, the coefficients are plotted against time.
- If the input is in the frequency domain, the coefficients are plotted against frame number.
- If the log energy is extracted, then it is also plotted.

Examples

Compute Mel Frequency Cepstral Coefficients

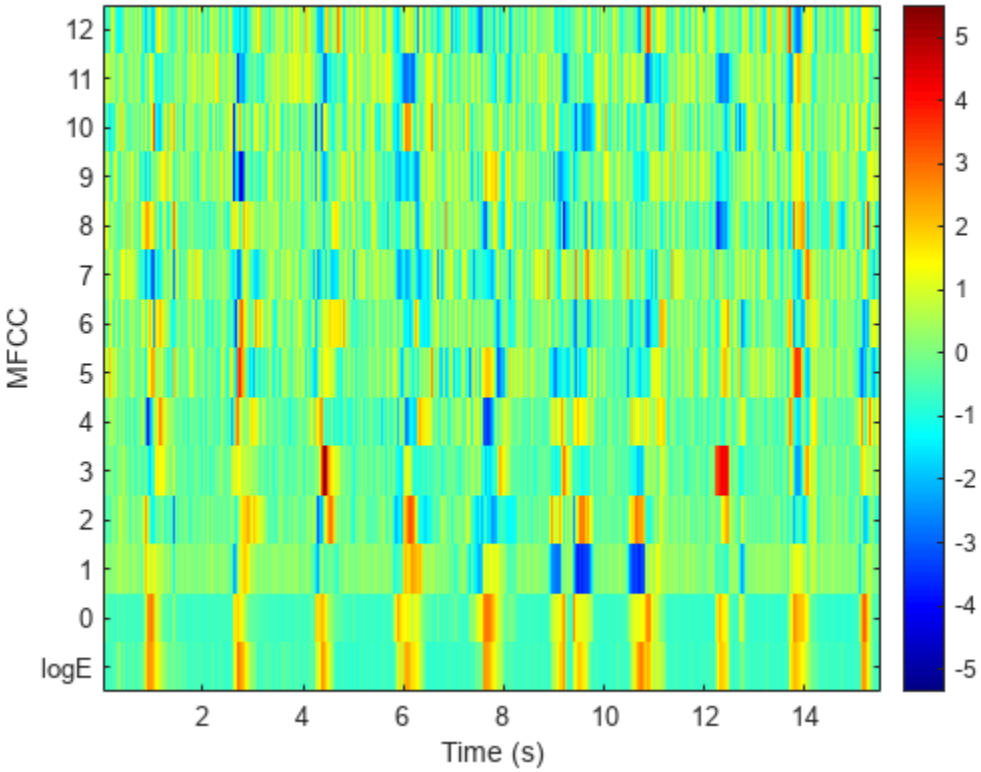
Compute the mel frequency cepstral coefficients of a speech signal using the `mfcc` function. The function returns `delta`, the change in coefficients, and `deltaDelta`, the change in delta values. The log energy value that the function computes can prepend the coefficients vector or replace the first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

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

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

Plot the normalized coefficients.

```
mfcc(audioIn,fs)
```



Extract MFCC from Frequency-Domain Audio

Read in an audio file and convert it to a frequency representation.

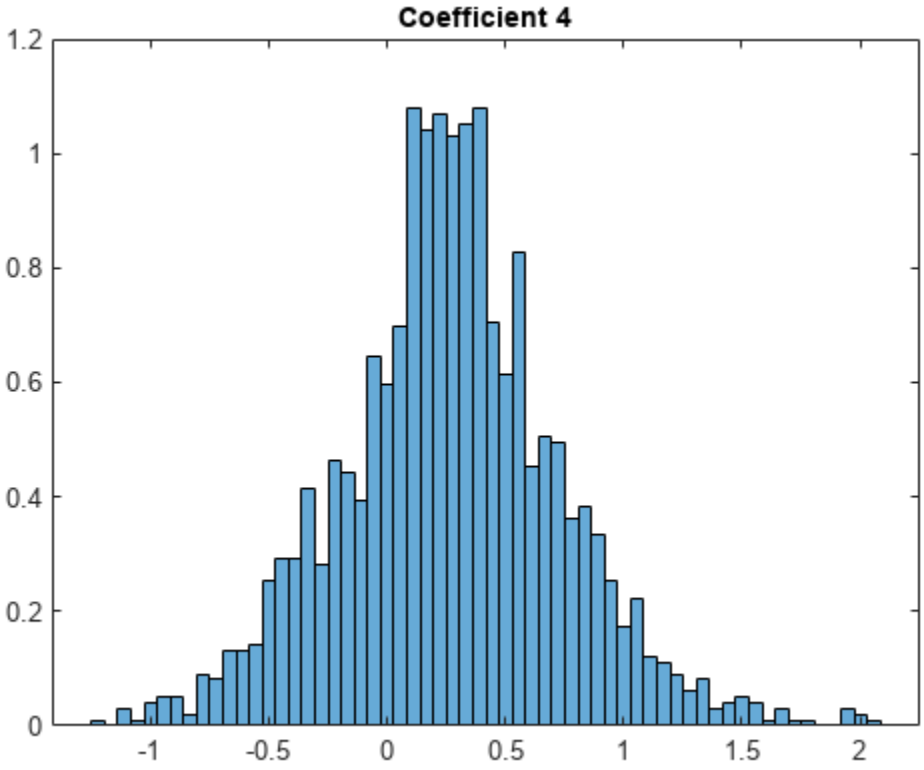
```
[audioIn,fs] = audioread("Rainbow-16-8-mono-114secs.wav");

win = hann(1024,"periodic");
S = stft(audioIn,"Window",win,"OverlapLength",512,"Centered",false);
```

To extract the mel-frequency cepstral coefficients, call `mfcc` with the frequency-domain audio. Ignore the log-energy.

```
coeffs = mfcc(S,fs,"LogEnergy","Ignore");
```

In many applications, MFCC observations are converted to summary statistics for use in classification tasks. Plot a probability density function for one of the mel-frequency cepstral coefficients to observe its distributions.

```
nbins = 60;

coefficientToAnalyze = 4;

histogram(coeffs(:,coefficientToAnalyze+1),nbins,"Normalization","pdf")
title(sprintf("Coefficient %d",coefficientToAnalyze))
```



Input Arguments

`audioIn` — Input signal

vector | matrix | 3-D array

Input signal, specified as a vector, matrix, or 3-D array.

- If `audioIn` is real, it is interpreted as a time-domain signal and must be a column vector or a matrix. Columns of the matrix are treated as independent audio channels.
- If `audioIn` is complex, it is interpreted as a frequency-domain signal. In this case, `audioIn` must be an *L*-by-*M*-by-*N* array, where *L* is the number of DFT points, *M* is the number of individual spectra, and *N* is the number of individual channels.

Data Types: `single` | `double`
Complex Number Support: Yes

`fs` — Sample rate (Hz)

positive scalar

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

Data Types: `single` | `double`

Name-Value Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `[coeffs,delta,deltaDelta,loc] = mfcc(audioIn,fs,LogEnergy="replace",DeltaWindowLength=5)` returns mel frequency cepstral coefficients for the audio input signal sampled at `fs` Hz. The first coefficient in the `coeffs` vector is replaced with the log energy value. A set of 5 cepstral coefficients is used to compute the delta and the delta-delta values.

`Window` — Window applied in time domain

`hamming(round(0.03*fs),"periodic")` (default) | vector

Window applied in time domain, specified as a real vector. The number of elements in the vector must be in the range [1,`size(audioIn,1)`]. The number of elements in the vector must also be greater than `OverlapLength`.

Data Types: `single` | `double`

`OverlapLength` — Number of overlapping samples between adjacent windows

`round(fs*0.02)` (default) | integer

Number of samples overlapped between adjacent windows, specified as an integer in the range [0, `numel(Window)`). If unspecified, `OverlapLength` defaults to `round(0.02*fs)`.

Data Types: `single` | `double`

`NumCoeffs` — Number of coefficients returned

`13` (default) | positive scalar integer

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

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

Data Types: `single` | `double`

`BandEdges` — Band edges of filter bank (Hz)

row vector

Band edges of the filter bank in Hz, specified as a nonnegative monotonically increasing row vector in the range [0, `fs`/2]. The number of band edges must be in the range [4, 160]. The `mfcc` function designs half-overlapped triangular filters based on `BandEdges`. This means that all band edges, except for the first and last, are also center frequencies of the designed bandpass filters.

By default, `BandEdges` is a 42-element vector, which results in a 40-band filter bank that spans approximately 133 Hz to 6864 Hz. The default bands are spaced as described in [2].

Data Types: `single` | `double`

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

Number of bins used to calculate the discrete Fourier transform (DFT) of windowed input samples. The FFT length must be greater than or equal to the number of elements in the `Window`.

Data Types: `single` | `double`

`Rectification` — Type of non-linear rectification
`"log"` (default) | `"cubic-root"`

Type of nonlinear rectification applied prior to the discrete cosine transform, specified as `"log"` or `"cubic-root"`.

Data Types: `char` | `string`

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

Number of coefficients used to calculate the delta and the delta-delta values, specified as an odd integer greater than two. If unspecified, `DeltaWindowLength` defaults to 9.

Deltas are computed using the `audioDelta` function.

Data Types: `single` | `double`

`LogEnergy` — Specify how the log energy is shown
`"append"` (default) | `"replace"` | `"ignore"`

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

- `"append"` –– The function prepends the log energy to the coefficients vector. The length of the coefficients vector is 1 + `NumCoeffs`.
- `"replace"` –– The function replaces the first coefficient with the log energy of the signal. The length of the coefficients vector is `NumCoeffs`.
- `"ignore"` –– The object does not calculate or return the log energy.

Data Types: `char` | `string`

Output Arguments

`coeffs` — Mel-frequency cepstral coefficients (MFCCs)
matrix | 3-D array

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

- *L* –– Number of analysis windows the audio signal is partitioned into. The input size, `Window`, and `OverlapLength` control this dimension: `L = floor((size(audioIn,1) - numel(Window)))/(numel(Window) - OverlapLength) + 1`.
- *M* –– Number of coefficients returned per frame. This value is determined by `NumCoeffs` and `LogEnergy`.

  When `LogEnergy` is set to:

- `"append"` –– The function prepends the log energy value to the coefficients vector. The length of the coefficients vector is 1 + `NumCoeffs`.
- `"replace"` –– The function replaces the first coefficient with the log energy of the signal. The length of the coefficients vector is `NumCoeffs`.
- `"ignore"` –– The function does not calculate or return the log energy. The length of the coefficients vector is `NumCoeffs`.

- *N* –– Number of input channels (columns). This value is `size(audioIn,2)`.

Data Types: `single` | `double`

`delta` — Change in coefficients
matrix | array

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

Data Types: `single` | `double`

`deltaDelta` — Change in delta values
matrix | array

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

Data Types: `single` | `double`

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

Location of last sample in each analysis window, returned as a column vector with the same number of rows as `coeffs`.

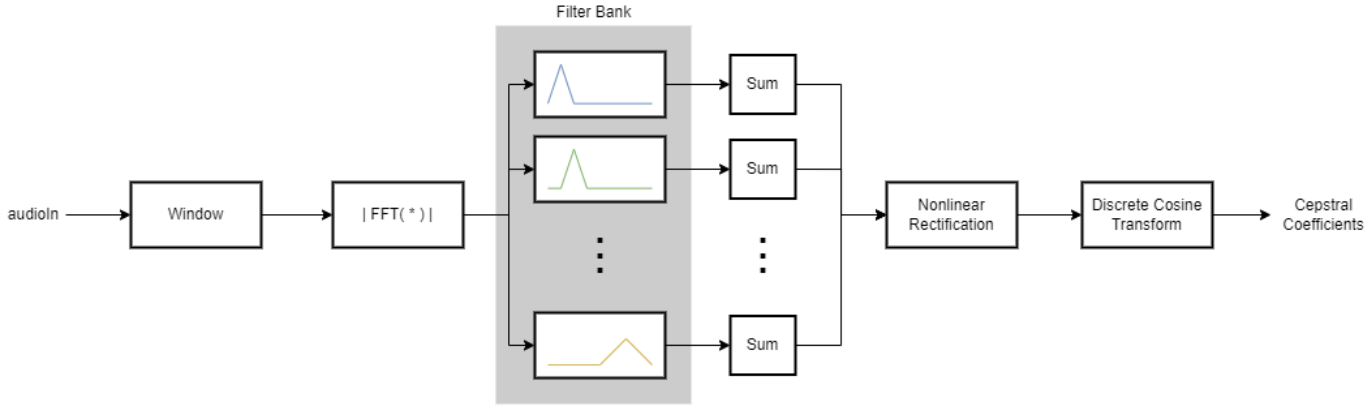
Data Types: `single` | `double`

Algorithms

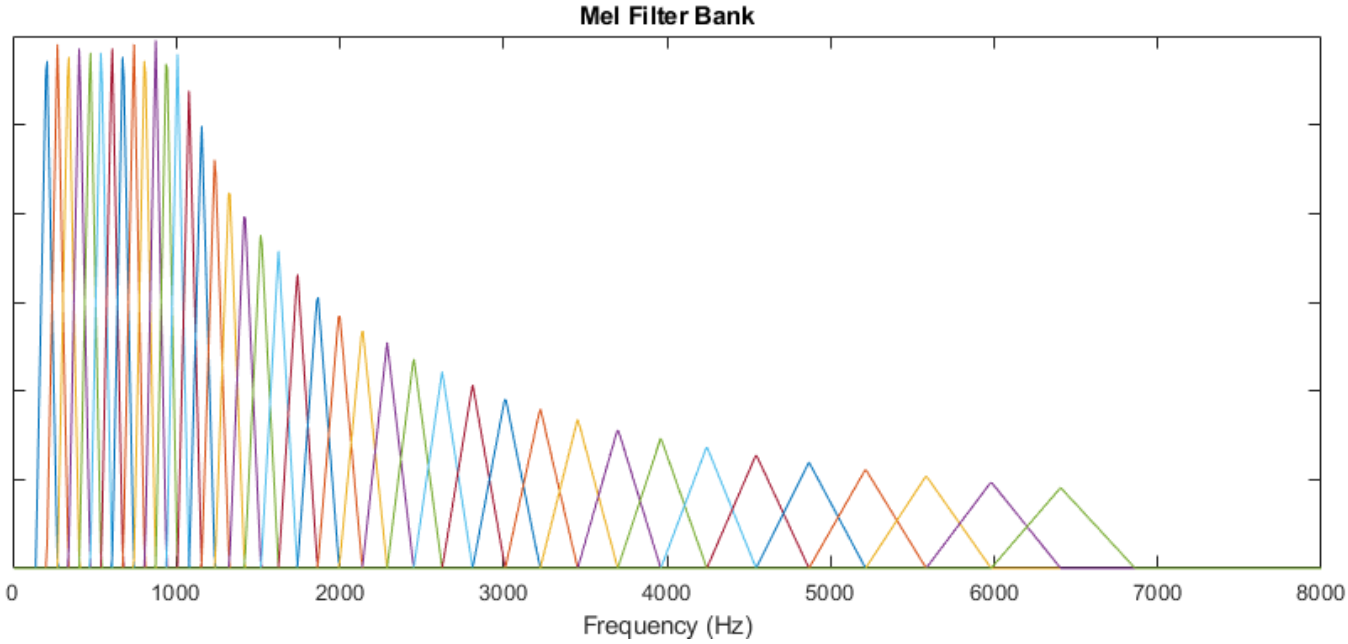
MFCC

Mel-frequency cepstrum coefficients are popular features extracted from speech signals for use in recognition tasks. In the source-filter model of speech, cepstral coefficients are understood to represent the filter (vocal tract). The vocal tract frequency response is relatively smooth, whereas the source of voiced speech can be modeled as an impulse train. As a result, the vocal tract can be estimated by the spectral envelope of a speech segment.

The motivating idea of mel-frequency cepstral coefficients is to compress information about the vocal tract (smoothed spectrum) into a small number of coefficients based on an understanding of the cochlea. Although there is no hard standard for calculating the coefficients, the basic steps are outlined by the diagram.



The default mel filter bank linearly spaces the first 10 triangular filters and logarithmically spaces the remaining filters.



Log Energy

The information contained in the zeroth mel-frequency cepstral coefficient is often augmented with or replaced by the log energy. The log energy calculation depends on the input domain.

If the input (*audioIn*) is a time-domain signal, the log energy is computed using the following equation:

$$\log E = \log(\text{sum}(x^2))$$

If the input (*audioIn*) is a frequency-domain signal, the log energy is computed using the following equation:

$$\log E = \log\left(\text{sum}\left(|x|^2\right)/FFTLength\right)$$

Version History

Introduced in R2018a

R2020b: Delta and delta-delta computation

Behavior changed in R2020b

The delta and delta-delta calculations are now computed using the `audioDelta` function, which has a different startup behavior than the previous algorithm. The default value of the `DeltaWindowLength` parameter has changed from `2` to `9`. A delta window length of `2` is no longer supported.

R2020b: `WindowLength` will be removed in a future release

Behavior change in future release

The `WindowLength` parameter will be removed from the `mfcc` function in a future release. Use the `Window` parameter instead.

In releases prior to R2020b, you could only specify the length of a time-domain window. The window was always designed as a periodic Hamming window. You can replace instances of the code

```
coeffs = mfcc(audioin,fs,WindowLength=1024);
```

With this code:

```
coeffs = mfcc(audioIn,fs,Window=hamming(1024,"periodic"));
```

References

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

[2] Auditory Toolbox. https://engineering.purdue.edu/~malcolm/interval/1998-010/AuditoryToolboxTechReport.pdf

Extended Capabilities

C/C++ Code Generation

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

GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

See Also

Functions

`audioDelta` | `cepstralCoefficients` | `detectSpeech`

Blocks

MFCC | Cepstral Coefficients | Audio Delta

Objects
`audioFeatureExtractor`

Topics
“Keyword Spotting in Noise Using MFCC and LSTM Networks”
“Speaker Identification Using Pitch and MFCC”

asiosettings

Open settings panel for ASIO driver

Syntax

```
asiosettings
asiosettings(deviceName)
```

Description

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

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

Examples

Open ASIO Settings Panel for Specified Device

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

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

Open ASIO Settings Panel for Default Device

Call the `asiosettings` function with no arguments.

```
asiosettings()
```

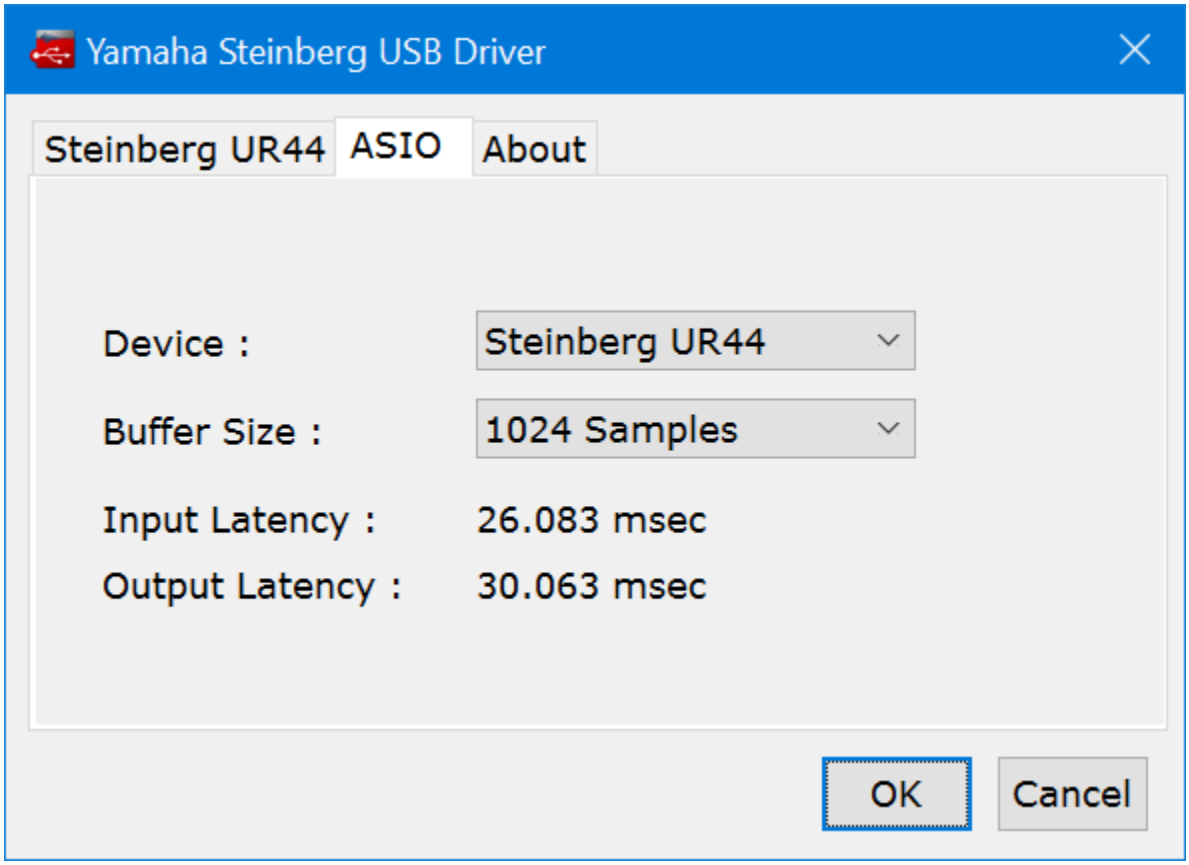
Optimize Latency

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

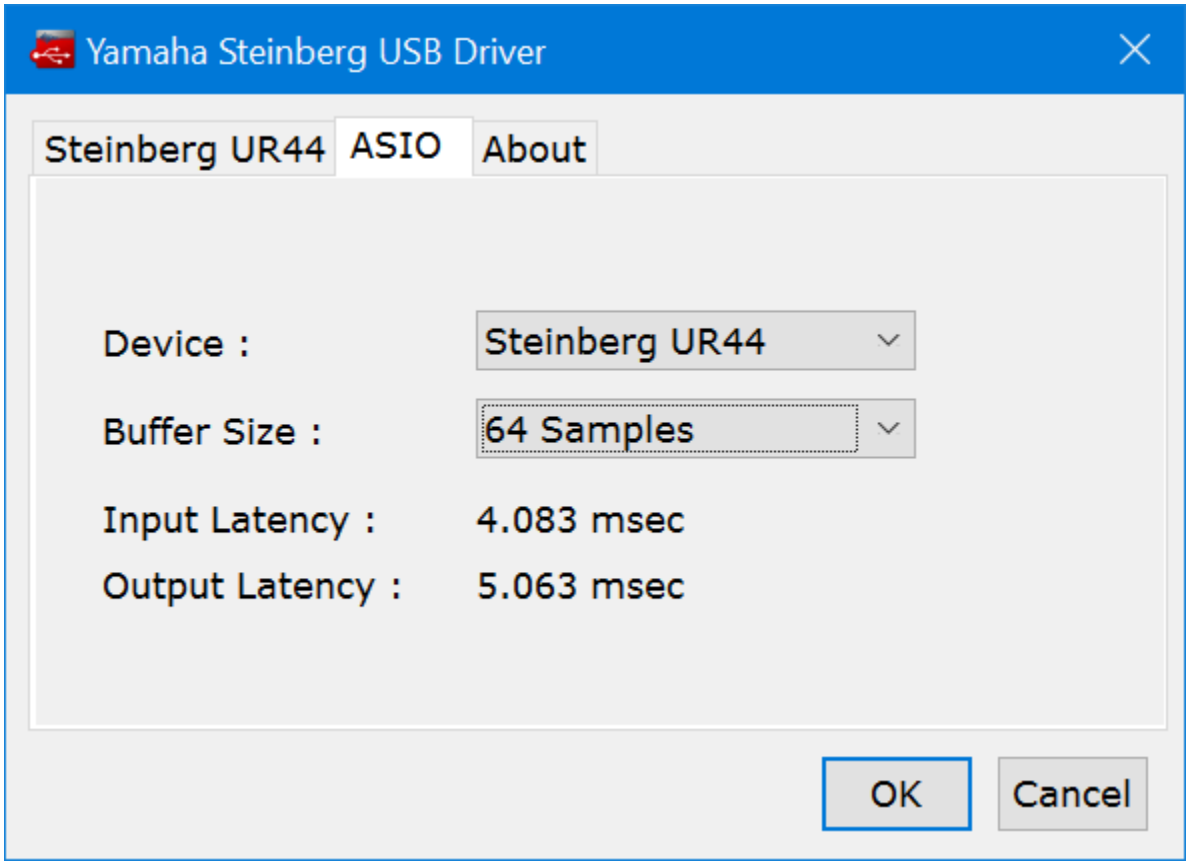
Create an `audioDeviceWriter` System object™. Open the ASIO settings panel for an ASIO-compatible device associated with your device writer.

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

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



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



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

Input Arguments

`deviceName` — Name of ASIO-compatible device
default ASIO-compatible device (default) | character vector | string

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

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

Data Types: `char` | `string`

Tips

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

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

```
Error using audio_asiosettings
PortAudio Error: Device unavailable


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

Version History

Introduced in R2017b

See Also

`audioDeviceReader` | `audioDeviceWriter` | `audioPlayerRecorder`

Topics

“Audio I/O: Buffering, Latency, and Throughput”

getAudioDevices

List available audio devices

Syntax

```
devices = getAudioDevices(obj)
```

Description

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

Examples

List Audio Devices Available to audioDeviceReader

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

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

```
devices = 1×4 cell
    {'Default'}    {'Primary Sound Capture Driver'}    {'Headset Microphone (Plantronics C325-M)
```

List Audio Devices Available to audioDeviceWriter

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

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

```
devices = 1×6 cell
    {'Default'}    {'Primary Sound Driver'}    {'Headset Earphone (Plantronics C325-M)'}    {'LE
```

List Audio Devices Available to audioPlayerRecorder

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

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

```
devices = 1×2 cell
    {'Default'}    {'ASIO4ALL v2'}
```

Input Arguments

`obj` — Audio I/O object

`audioDeviceReader` object | `audioDeviceWriter` object | `audioPlayerRecorder` object

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

Data Types: `object`

Output Arguments

`devices` — List of available and compatible devices

array

List of available and compatible devices.

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

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

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

Data Types: `cell`

Version History

Introduced in R2016a

See Also

`audioDeviceWriter` | `audioDeviceReader` | `audioPlayerRecorder`

Topics

“Audio I/O: Buffering, Latency, and Throughput”

audioPluginInterface

Specify audio plugin interface

Syntax

```
PluginInterface = audioPluginInterface
PluginInterface = audioPluginInterface(pluginParameters)
PluginInterface = audioPluginInterface(pluginParameters,gridLayout)
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.

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

`PluginInterface = audioPluginInterface(pluginParameters,gridLayout)` specifies a grid layout for audio plugin parameter UI controls.

`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, number of output channels, and a yellow background.

```
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, ...
            'BackgroundColor','y');
    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, audio plugin parameters synchronize plugin class properties with user-facing parameters.

`gridLayout` — Layout for plugin UI

none (default) | `audioPluginGridLayout` object

Audio plugin grid layout, specified as an `audioPluginGridLayout` object.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

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

`PluginName` — Name of generated plugin
name of plugin class (default) | character vector | string

Name of your generated plugin, as seen by a host audio application, specified as a comma-separated pair consisting of `'PluginName'` and a character vector or 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 plugin creator
`''` (default) | character vector

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

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

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

Example: `'1'`

Example: `'1.4'`

Example: `'1.3.5'`

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

Unique identifier for your plugin, specified as a comma-separated pair consisting of `'UniqueID'` and a four-element character vector or 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.

`BackgroundColor` — Color used for GUI background

RGB triplet | short name | long name

Color used for GUI background, specified as short or long color name string, or an RGB triplet

Example: `'BackgroundColor',[1 1 0]` specifies the GUI background to be yellow.

Example: `'BackgroundColor','y'` specifies the GUI background to be yellow.

Example: `'BackgroundColor','yellow'` specifies the GUI background to be yellow.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
`"red"`	`"r"`	`[1 0 0]`	`"#FF0000"`	
`"green"`	`"g"`	`[0 1 0]`	`"#00FF00"`	
`"blue"`	`"b"`	`[0 0 1]`	`"#0000FF"`	
`"cyan"`	`"c"`	`[0 1 1]`	`"#00FFFF"`	
`"magenta"`	`"m"`	`[1 0 1]`	`"#FF00FF"`	
`"yellow"`	`"y"`	`[1 1 0]`	`"#FFFF00"`	
`"black"`	`"k"`	`[0 0 0]`	`"#000000"`	
`"white"`	`"w"`	`[1 1 1]`	`"#FFFFFF"`	

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
`[0 0.4470 0.7410]`	`"#0072BD"`	
`[0.8500 0.3250 0.0980]`	`"#D95319"`	
`[0.9290 0.6940 0.1250]`	`"#EDB120"`	
`[0.4940 0.1840 0.5560]`	`"#7E2F8E"`	
`[0.4660 0.6740 0.1880]`	`"#77AC30"`	
`[0.3010 0.7450 0.9330]`	`"#4DBEEE"`	
`[0.6350 0.0780 0.1840]`	`"#A2142F"`	

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

`BackgroundImage` — Image used for GUI background

char | string

Image used for GUI background, specified by its file name using either a character vector or string. If the file is not on path, you must specify the full file path. Supported file types are PNG, GIF, and JPG.

The background image may include transparencies, in which case the `BackgroundColor` is used.

Example: `'BackgroundImage','Sunrise.png'` specifies the GUI background image to be the `'Sunrise'` image.

Example: `'BackgroundImage',fullfile(matlabroot,"mySkins","Sunset.jpg")` specifies the GUI background to be the `'Sunset'` image.

Data Types: `char` | `string`

Version History

Introduced in R2016a

See Also

`audioPlugin` | `audioPluginSource` | `audioPluginParameter` | `generateAudioPlugin` | `validateAudioPlugin` | `audioPluginGridLayout`

Topics

“Audio Plugins in MATLAB”

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 `audioPluginInterface` in your plugin class definition.

In a digital audio workstation (DAW) environment, or when using **Audio Test Bench** or `parameterTuner` in the MATLAB environment, plugin parameters are tunable, user-facing values 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 2-520.

`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
```

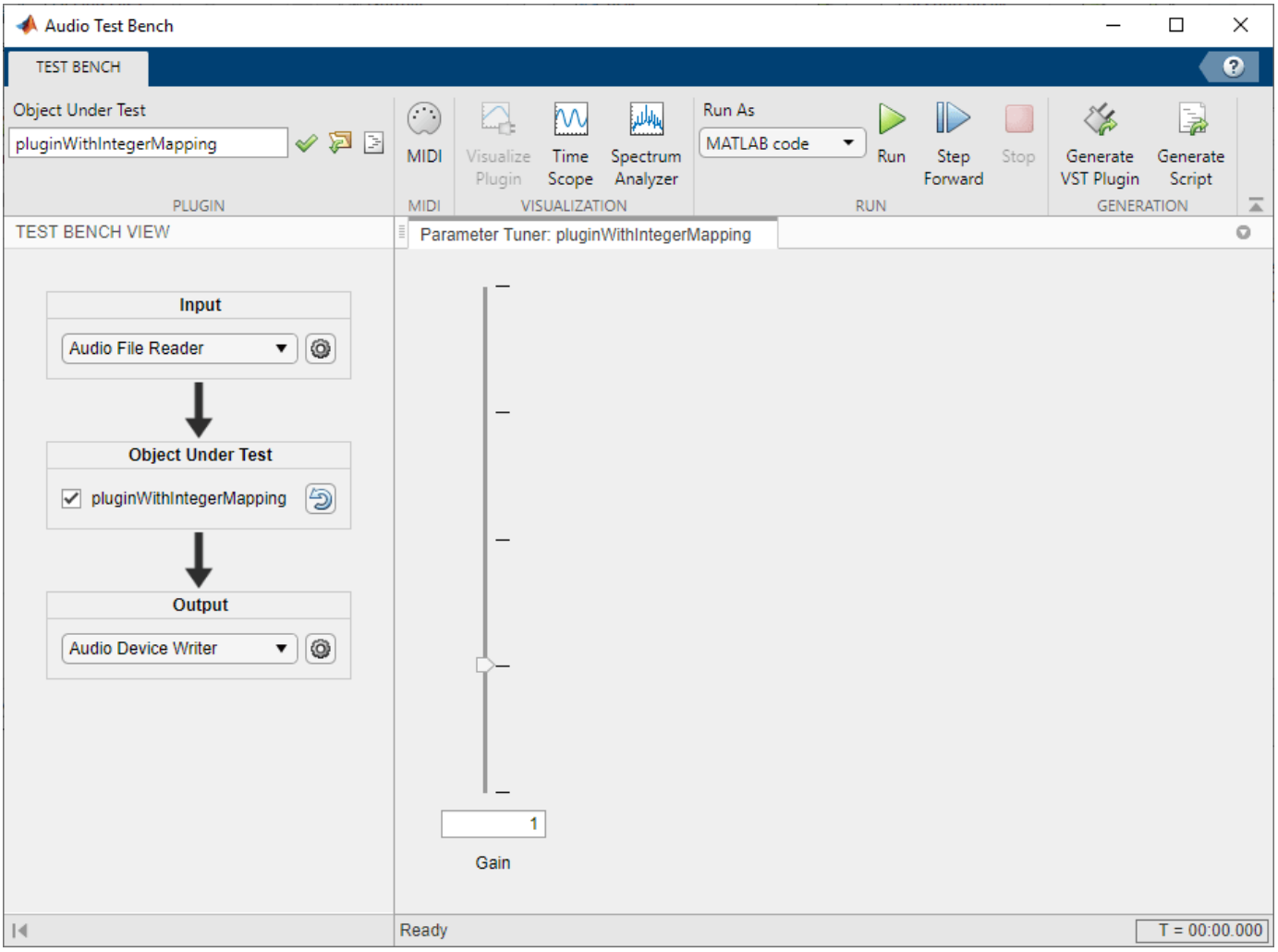
Integer Parameter Mapping

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',0,4}, ...
            'Layout',[1,1], ...
            'Style','vslider'), ...
            audioPluginGridLayout('RowHeight',[400,20]));
    end
    methods
        function out = process(plugin,in)
            out = in*plugin.Gain;
        end
    end
end
```

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

```
audioTestBench(pluginWithIntegerMapping)
```



Power Parameter Mapping

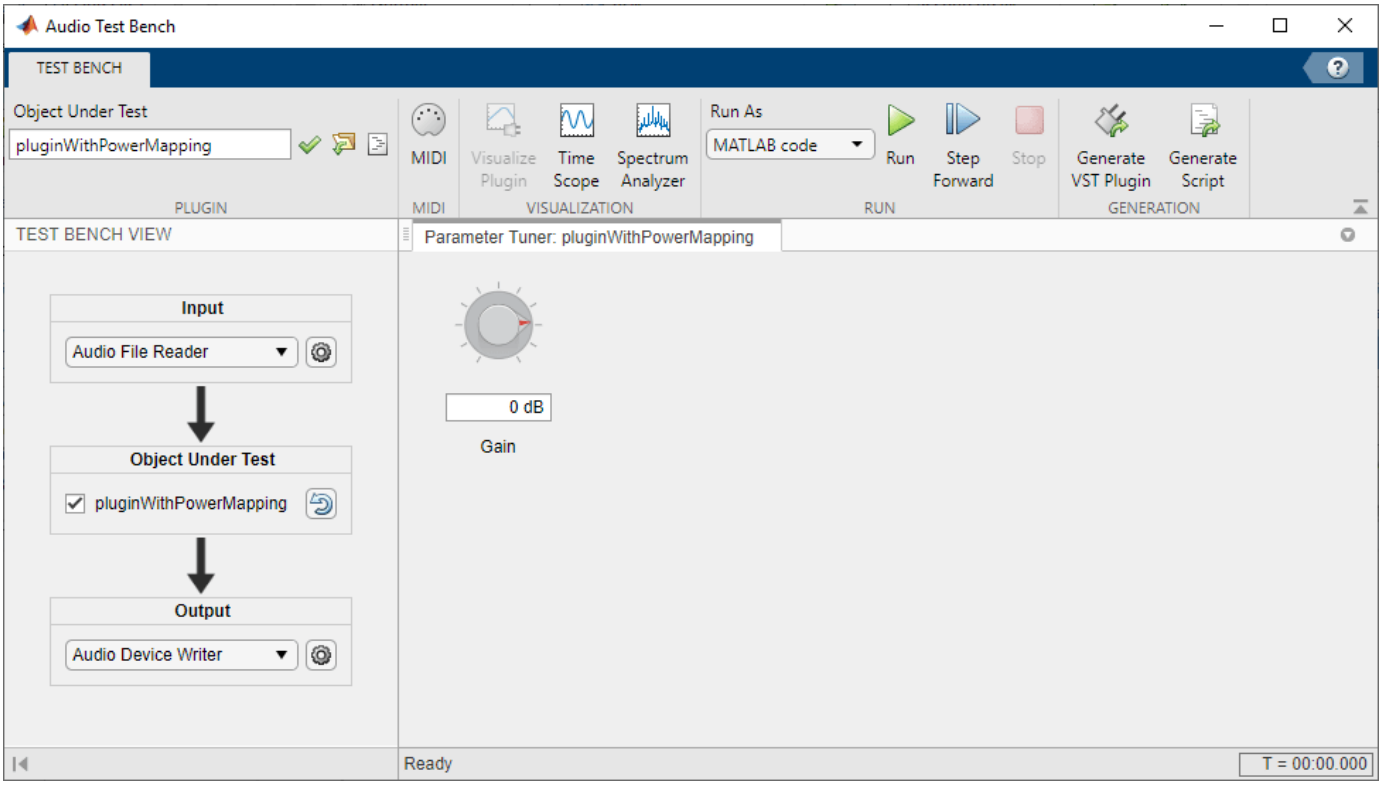
The following class definition uses power parameter mapping to define the relationship between a property and a parameter. You can use the plugin created from this class to tune the gain of an audio signal in dB.

```
classdef pluginWithPowerMapping < audioPlugin
    properties
        Gain = 0;
    end
    properties (Constant)
        PluginInterface = audioPluginInterface( ...
            audioPluginParameter('Gain', ...
            'Label','dB', ...
            'Mapping',{'pow', 1/3, -140, 12}, ...
            'Style','rotary', ...
            'Layout',[1,1]), ...
            audioPluginGridLayout);
```

```
    end
    methods
        function out = process(plugin,in)
            dBGain = 10^(plugin.Gain/20);
            out = in*dBGain;
        end
    end
end
```

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

```
audioTestBench(pluginWithPowerMapping)
```



Logarithmic Parameter Mapping

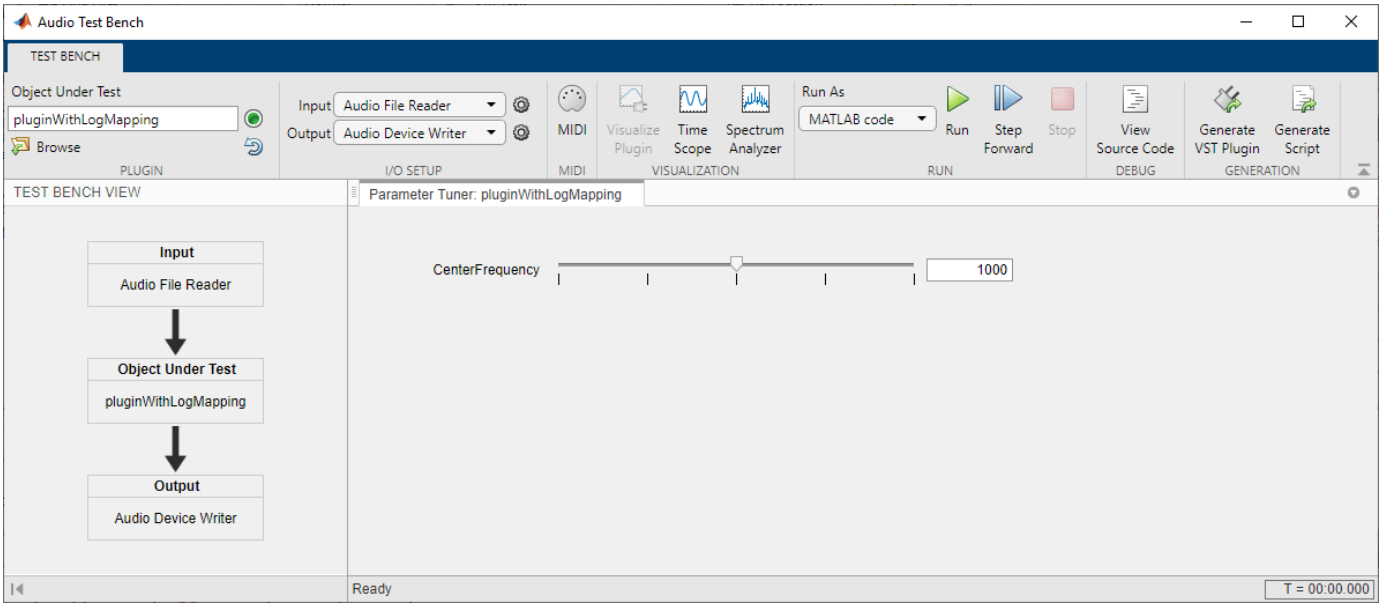
The following class definition uses logarithmic parameter mapping to define the relationship between a property and a parameter. You can use the plugin created from this class to tune the center frequency of a single-band EQ filter from 100 to 10000.

```
classdef pluginWithLogMapping < audioPlugin
    properties
        EQ
        CenterFrequency = 1000;
    end
    properties (Constant)
        PluginInterface = audioPluginInterface( ...
            audioPluginParameter('CenterFrequency', ...
```

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

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

```
audioTestBench(pluginWithLogMapping)
```



Enumeration for Logical Properties Parameter Mapping

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

```
classdef pluginWithLogicalEnumMapping < audioPlugin
    properties
```

```
        PassThrough = true;
    end
    properties (Constant)
        PluginInterface = audioPluginInterface( ...
            audioPluginParameter('PassThrough', ...
            'Mapping', {'enum','Block signal','Pass through'}, ...
            'Layout',[1,1], ...
            'Style','vtoggle', ...
            'DisplayNameLocation','none'), ...
            audioPluginGridLayout);
    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 create an audio I/O stream loop.

First, create objects to read from a file and write to your device.

```
fileReader = dsp.AudioFileReader('Engine-16-44p1-stereo-20sec.wav');
deviceWriter = audioDeviceWriter('SampleRate',fileReader.SampleRate);
```

Create a plugin object and set the sample rate to the sample rate of the file.

```
passThrough = pluginWithLogicalEnumMapping;
setSampleRate(passThrough,fileReader.SampleRate)
```

Open a `parameterTuner` so that you can toggle the logical parameter of the plugin while stream processing.

```
parameterTuner(passThrough)
```

While the file contains unread data:

1. Read a frame from the file.
2. Feed the frame through the plugin
3. Write the processed audio to your device.

While the audio stream runs, toggle the `PassThrough` parameter and listen to the effect.

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

    audioOut = process(passThrough,audioIn);

    deviceWriter(audioOut);

    drawnow limitrate
end
```

'enum' for Enumeration Class Parameter Mapping

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

Plugin Class Definition

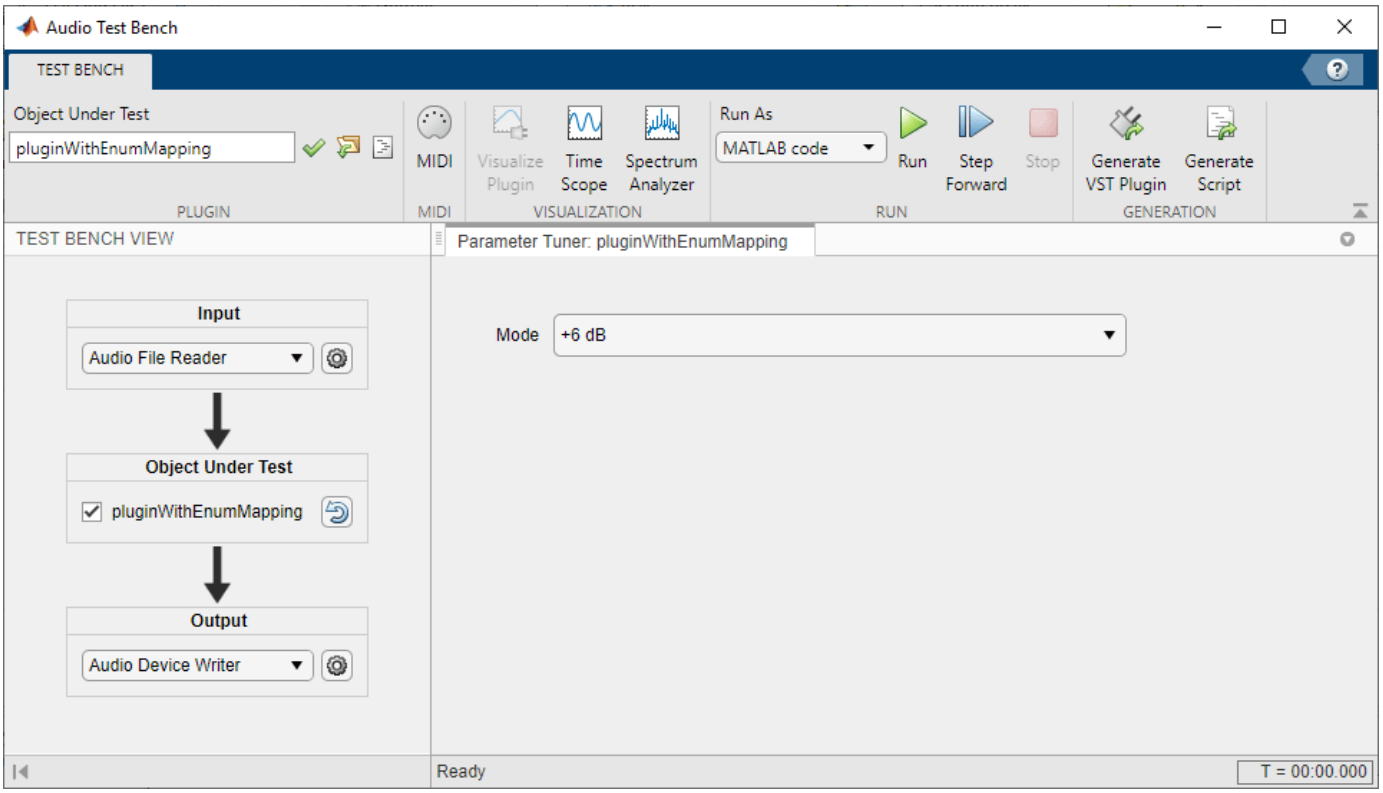
```
classdef pluginWithEnumMapping < audioPlugin
    properties
        Mode = OperatingMode.boost;
    end
    properties (Constant)
        PluginInterface = audioPluginInterface(...
            audioPluginParameter('Mode',...
                'Mapping',{'enum','+6 dB','-6 dB','silence','white noise'}));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

Enumeration Class Definition

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

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

```
audioTestBench(pluginWithEnumMapping)
```



Input Arguments

`propertyName` — Name of audio plugin property

character vector | string

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

Data Types: `char` | `string`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

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

Mappings

`Mapping` — Mapping between property and parameter range
cell array

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

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

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

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

Mapping	Description	Example
`'lin'`	Specifies a linear relationship with given minimum and maximum values. (*property value*) = min + (max − min) × (*parameter value*)	`{'lin', 0, 24}` specifies a linear relationship with a minimum of 0 and maximum of 24. **Example:** “Specify Parameter Information” on page 2-507
`'log'`	Specifies a logarithmic relationship with given minimum and maximum values, where the control position maps to the logarithm of the property value. The minimum value must be greater than 0. (*property value*) = min × $(\max/\min)^{(\textit{parameter value})}$	`{'log', 1, 22050}` specifies a logarithmic relationship with a minimum of 1 and a maximum of 22,050. **Example:** “Logarithmic Parameter Mapping” on page 2-510

Mapping	Description	Example
`'pow'`	Specifies a power law relationship with given exponent, minimum, and maximum values. The property value is related to the control position raised to the exponent: (*property value*) = min + (max − min) × (*parameter value*)^{exp}	`{'pow', 1/3, -140, 12}` specifies a power law relationship with an exponent of 1/3, a minimum of –140, and a maximum of 12. **Example:** “Power Parameter Mapping” on page 2-509
`'int'`	Quantizes the control position and maps it to the range of consecutive integers with given minimum and maximum values. (*property value*) = floor (0.5 + min + (max − min) × (*parameter value*))	`{'int', 0, 3}` specifies a linear, quantized relationship with a minimum of 0 and maximum of 3. The property value is mapped as an integer in the range [0, 3]. **Example:** “Integer Parameter Mapping” on page 2-507
`'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`. **Example:** “Enumeration for Logical Properties Parameter Mapping” on page 2-511
`'enum'` (enumeration class)	Optionally provides character vectors for the members of the enumeration class.	`{'enum', '+6 dB', '-6 dB', 'silence', 'white noise'}` specifies the character vectors `'+6 dB'`, `'-6 dB'`, `'silence'`, and `'white noise'`. **Example:** “'enum' for Enumeration Class Parameter Mapping” on page 2-513

Graphical User Interface

`Layout` — Grid cells occupied by parameter control

[row, column] (single-cell specification) | [upper, left; lower, right] (multi-cell specification)

Grid cells occupied by parameter control, specified as a comma-separated pair consisting of `'Layout'` and a two-element row vector or 2-by-2 matrix. To use a single cell, specify [row, column] of the cell. To span multiple cells, specify the upper left and lower right cells as [upper, left; lower, right].

Example: `'Layout',[2,3]`

Example: `'Layout',[2,3;3,6]`

Dependencies

To enable this name-value pair, pass an `audioPluginGridLayout` object to `audioPluginInterface`.

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

`DisplayName` — Display name of parameter

associated property name (default) | character vector | string

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

Data Types: `char` | `string`

`DisplayNameLocation` — Location of display name

`"left"` | `"right"` | `"above"` | `"below"` | `"none"`

Location of `DisplayName` relative to `Layout`, specified as `"left"`, `"right"`, `"above"`, `"below"`, or `"none"`.

- `"left"` –– The display name is located in the column to the left of `Layout` and spans the same rows as `Layout`.
- `"right"` –– The display name is located in the column to the right of `Layout` and spans the same rows as `Layout`.
- `"above"` –– The display name is located in the row above `Layout` and spans the same columns as `Layout`
- `"below"` –– The display name is located in the row below `Layout` and spans the same columns as `Layout`.
- `"none"` –– `DisplayName` is suppressed.

The `DisplayName` of the parameter does not occupy the same grid cells as the control for the parameter.

Example: `DisplayNameLocation="left"`

Dependencies

To enable this name-value argument, pass an `audioPluginGridLayout` object to `audioPluginInterface`.

Data Types: `char` | `string`

`EditBoxLocation` — Location of edit box

`"left"` | `"right"` | `"above"` | `"below"` | `"none"`

Location of edit box for the parameter relative to the control, specified as `"left"`, `"right"`, `"above"`, `"below"`, or `"none"`.

- `"left"` –– The edit box is located to the left of the control.
- `"right"` –– The edit box is located to the right of the control.
- `"above"` –– The edit box is located above the control.
- `"below"` –– The edit box is located below the control.
- `"none"` –– The edit box is suppressed.

The edit box exists so that users can directly enter a numeric value if the control `Style` is `"hslider"`, `"vslider"`, or `"rotaryknob"`.

The edit box occupies the same grid cells as the control for the parameter, which are the cells specified by `Layout`.

Example: `EditBoxLocation="right"`

Dependencies

To enable this name-value argument, pass an `audioPluginGridLayout` object to `audioPluginInterface`.

This argument only applies if `Style` is `"hslider"`, `"vslider"`, or `"rotaryknob"`.

Data Types: `char` | `string`

`Label` — Display label for parameter value units

`''` (default) | character vector | string

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

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

Data Types: `char` | `string`

`Style` — Visual control for plugin parameter

`'hslider'` | `'vslider'` | `'rotaryknob'` | `'checkbox'` | `'vrocker'` | `'vtoggle'` | `'dropdown'`

Visual control for plugin parameter, specified as a comma-separated pair consisting of `'Style'` and a string or character vector:

Style	Description
`'hslider'`	Horizontal slider
`'vslider'`	Vertical slider
`'rotaryknob'`	Rotary knob
`'checkbox'`	Check box
`'vrocker'`	Vertical rocker switch
`'vtoggle'`	Vertical toggle switch
`'dropdown'`	Dropdown

Default and valid styles depends on the plugin parameter `Mapping` and corresponding property class:

Mapping	Property Class	Default Style	Additional Valid Styl

`lin`
`log`
`pow`
`int`
	single
	double
		`hslider`	`vslider`
			`rotaryknob`

`enum`	logical	`checkbox`	`dropdown`
			`vrocker`
			`vtoggle`

Mapping	Property Class	Default Style	Additional Valid Style
`enum`	enumeration with 2 values	`vrocker`	`dropdown` `vtoggle`
`enum`	enumeration	`dropdown`	

Dependencies

To enable this name-value pair, pass an `audioPluginGridLayout` object to `audioPluginInterface`.

Data Types: `char` | `string`

`Filmstrip` — Name of PNG, GIF, or JPG graphics file

character vector | string

Name of PNG, GIF, or JPG graphics file, specified as the comma-separated pair consisting of `'Filmstrip'` and a character vector or string. The graphics file contains a sequence of images of controls.

Filmstrips enable you to replace default control graphics with your own custom images. Filmstrips support all control `Style` values except for dropdowns. A filmstrip is a single image created by concatenating smaller images called frames. Each frame is an image of a control in a particular position. For example, a filmstrip for a switch contains two frames: one depicting the "off" state and another depicting the "on" state. Frames can be concatenated vertically or horizontally. Suppose that the switch frames are 50 pixels wide by 100 pixels high. Then vertical concatenation produces a 50-by-200 pixel filmstrip image, with the top frame used for the switch "off" state. Horizontal concatenation produces a 100-by-100 pixel image, with the left frame used for the switch "off" state. Filmstrips for sliders and knobs typically contain many more frames. The top/left frame corresponds to the minimum control position, and the bottom/right frame corresponds to the maximum control position. The relative control position determines which frame is displayed for a given parameter value.

Dependencies

To enable this name-value pair, pass an `audioPluginGridLayout` object to `audioPluginInterface` and specify `'FilmstripFrameSize'`.

Data Types: `char` | `string`

`FilmstripFrameSize` — Size of individual frames (pixels)

[width, height]

Size of individual frames in the film strip in pixels, specified as the comma-separated pair consisting of `'FilmstripFrameSize'` and a two-element row vector of integers that specify [width, height].

Dependencies

To enable this name-value pair, pass an `audioPluginGridLayout` object to `audioPluginInterface` and specify a `'Filmstrip'`.

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

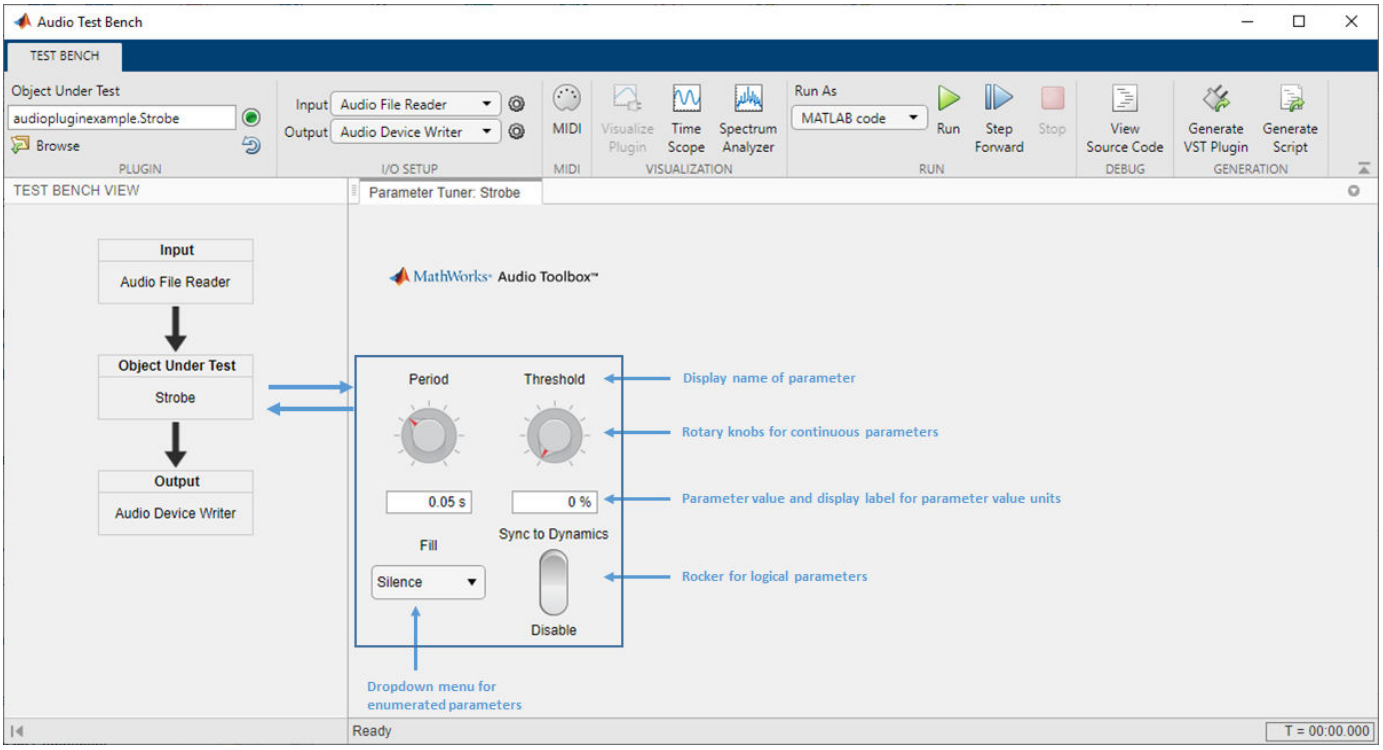
To learn how to design a graphic user interface, see “Design User Interface for Audio Plugin”.

More About

Implementation of Audio Plugin Parameters

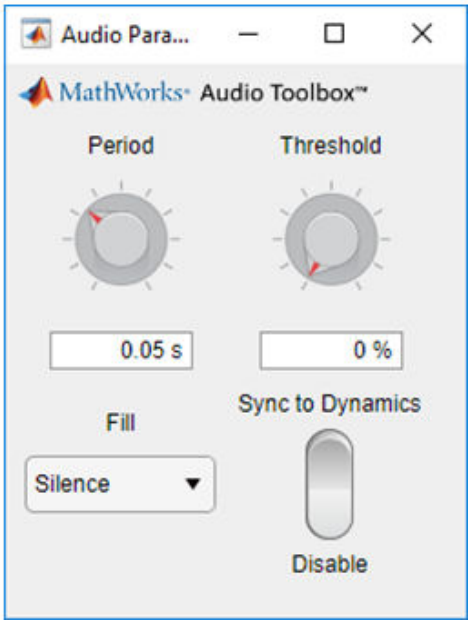
Audio plugin parameters are visible and tunable in both the MATLAB and digital audio workstation (DAW) environments. The different environments and corresponding renderings of the audio plugin parameters are outlined here. For an example describing how your class definition maps to the UI, see “Design User Interface for Audio Plugin”.

MATLAB Environment Using Audio Test Bench. Use **Audio Test Bench** to interact with plugin parameters in the MATLAB environment in a complete GUI-based workflow. Using the **Audio Test Bench**, you can specify audio input and output, analyze your plugin using time- and frequency-domain scopes, connect to MIDI controls, and validate and generate your plugin. The **Audio Test Bench** honors the graphical user interface you specified in `audioPluginParameter`, `audioPluginGridLayout`, and `audioPluginInterface` (except for filmstrips).

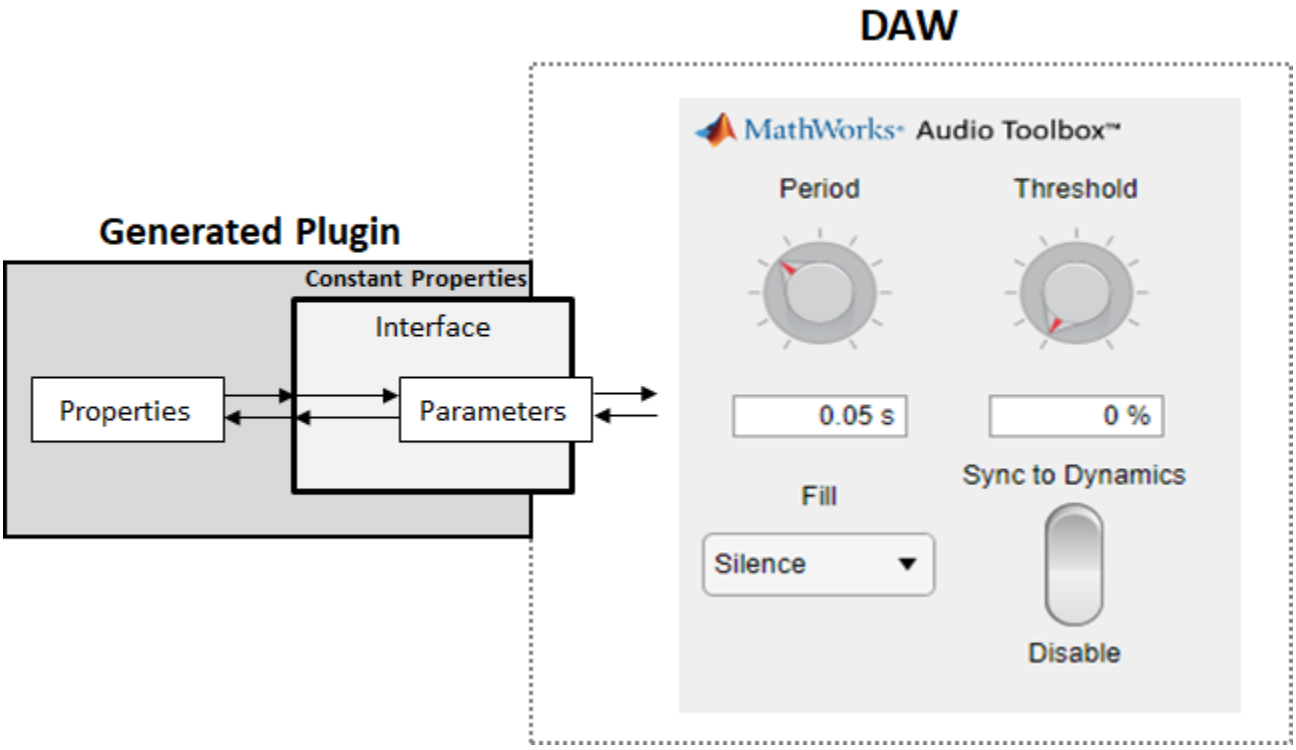


MATLAB Environment Using parameterTuner. Use `parameterTuner` to interact with plugin parameters in the MATLAB environment while developing, analyzing, or using your plugin in a programmatic workflow. The `parameterTuner` honors the graphical user interface you specified in `audioPluginParameter`, `audioPluginGridLayout`, and `audioPluginInterface` (except for filmstrips).

```
% Create test bench input and output
fileReader = dsp.AudioFileReader('RockDrums-44p1-stereo-11secs.mp3', ...
    'PlayCount',Inf, ...
    'SamplesPerFrame',256);
deviceWriter = audioDeviceWriter('SampleRate',fileReader.SampleRate);

% Set up the system under test
sut = audiopluginexample.Strobe;
setSampleRate(sut,fileReader.SampleRate);

% Open parameterTuner for interactive tuning during simulation
tuner = parameterTuner(sut);

% Stream processing loop
while ~isDone(fileReader)
    in = fileReader();

    out = process(sut,in);

    drawnow limitrate
end

% Clean up
release(fileReader)
release(deviceWriter)
```



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.



Version History

Introduced in R2016a

See Also

`audioPlugin` | `audioPluginSource` | `audioPluginInterface` | `validateAudioPlugin` | `generateAudioPlugin` | **Audio Test Bench** | `parameterTuner`

Topics

“Design an Audio Plugin”
“Design User Interface for Audio Plugin”
“Export a MATLAB Plugin to a DAW”
“Audio Plugin Example Gallery”

configureMIDI

Configure MIDI connections between audio object and MIDI controller

Syntax

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

Description

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

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

```
setpref midi DefaultDevice deviceNameValue
```

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

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

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

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

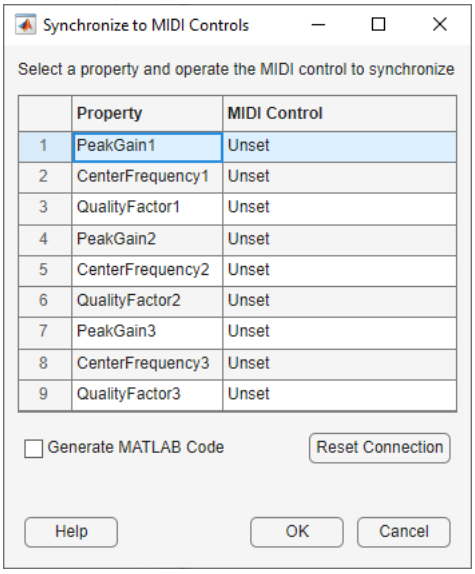
Examples

Synchronize Plugin Parameters to MIDI Controls

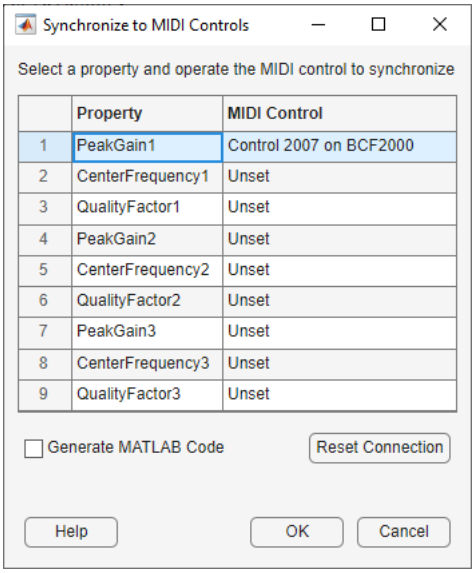
1 Open the MIDI configuration UI for a parametric equalizer plugin object.

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

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



3 On your MIDI device, operate the control that you want to synchronize to the selected plugin property. The control appears in the **MIDI Control** column in the row of the corresponding property.



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

   To disconnect the property and control currently selected on your `configureMIDI` UI, click **Reset Connection**.

5 Click **OK**.

   The specified MIDI controls and properties and now synchronized.

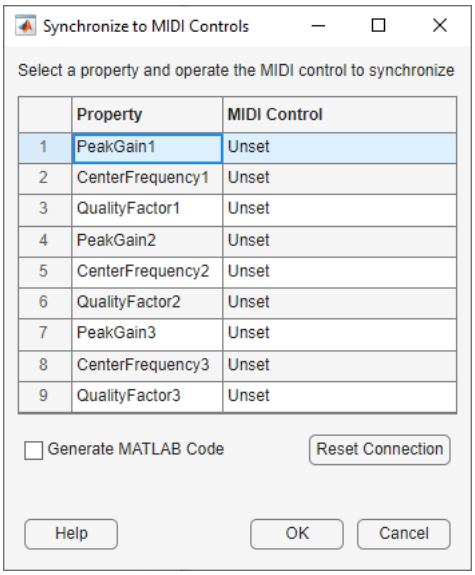
Generate MATLAB Code from `configureMIDI` UI

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.

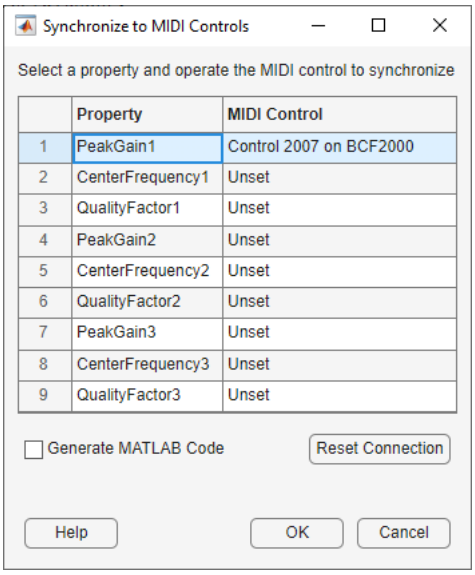
1. Open the MIDI configuration UI for a parametric equalizer plugin object.

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

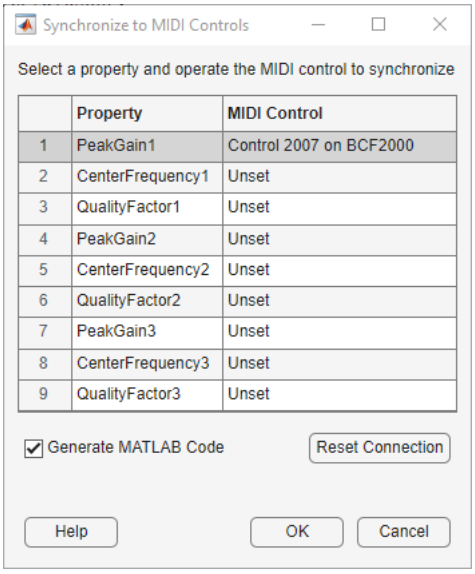
2. In the UI, select a property to synchronize with your default MIDI device.



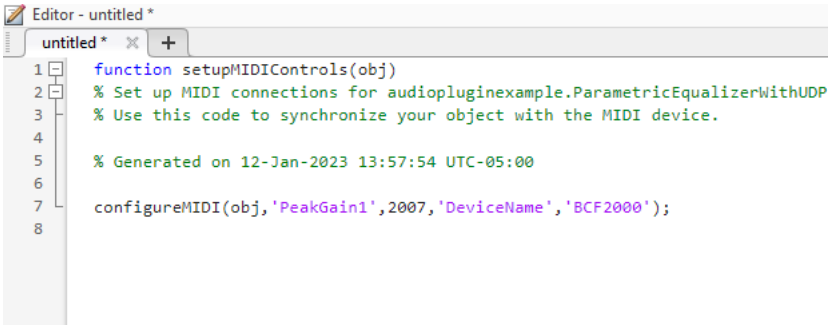
3. On your MIDI device, operate the control that you want to synchronize to the selected plugin property. The control appears in the **MIDI Control** column in the row of the corresponding property.



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



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



Make Plugin Property Respond to Any MIDI Control

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

```
parametricEQPlugin = audiopluginexample.ParametricEqualizerWithUDP;
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.ParametricEqualizerWithUDP`.

```
parametricEQPlugin = audiopluginexample.ParametricEqualizerWithUDP;
```

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 =

        1083


device =

    'BCF2000'
```

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.ParametricEqualizerWithUDP`.

```
parametricEQPlugin = audiopluginexample.ParametricEqualizerWithUDP;
```

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 =

        1087


device =

    'BCF2000'
```

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

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

Input Arguments

`audioObject` — Audio object

object

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

`propertyName` — Name of object property
character vector | string

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

`controlNumber` — MIDI device control number
integer

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 | string

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

Limitations

For MIDI connections established by `configureMIDI`, moving a MIDI control sends a callback to update the associated property values. To synchronize your MIDI device in an audio stream loop, you might need to use the `drawnow` command for the callback to process immediately. For efficiency, use the `drawnow limitrate` syntax.

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

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

configureMIDI(compressor,'Threshold')

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

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

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

Version History

Introduced in R2016a

R2023a: Updated UI

The improved UI allows you to view all properties of an object and their associated MIDI controls at once.

See Also

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

Topics

“MIDI Control for Audio Plugins”
“MIDI Control Surface Interface”

designParamEQ

Design parametric equalizer

Syntax

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

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.

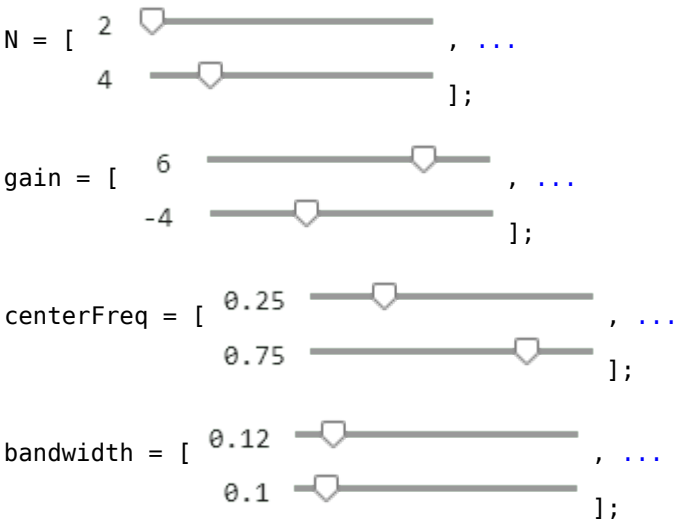
`[B,A] = designParamEQ(N,gain,centerFreq,bandwidth,mode)` specifies whether the parametric equalizer is implemented with second-order sections or fourth-order sections (FOS).

`[B,A] = designParamEQ(___,Name,Value)` specifies options using one or more `Name,Value` pair arguments.

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.

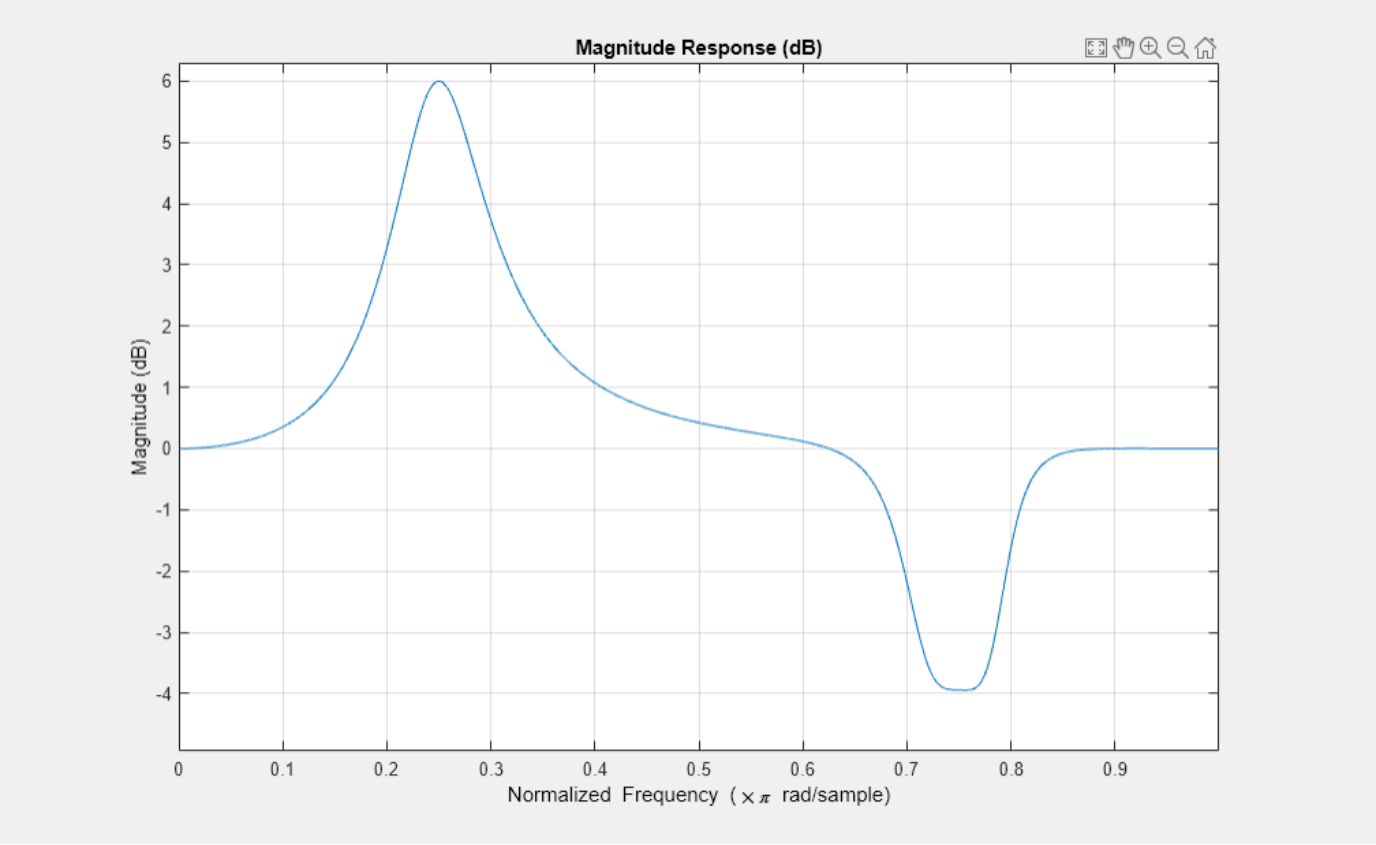


Generate the filter coefficients using the specified parameters.

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

Visualize your filter design.

```
fvtool([B,A]);
```



Filter Audio Using SOS Parametric Equalizer

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

Create audio file reader and audio device writer System objects. Use the sample rate of the reader as the sample rate of the writer.

```
frameSize = 256;

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

sampleRate = fileReader.SampleRate;

deviceWriter = audioDeviceWriter(SampleRate=sampleRate);
```

Play the audio signal through your device.

```
count = 0;
while count < 2500
    audio = fileReader();
```

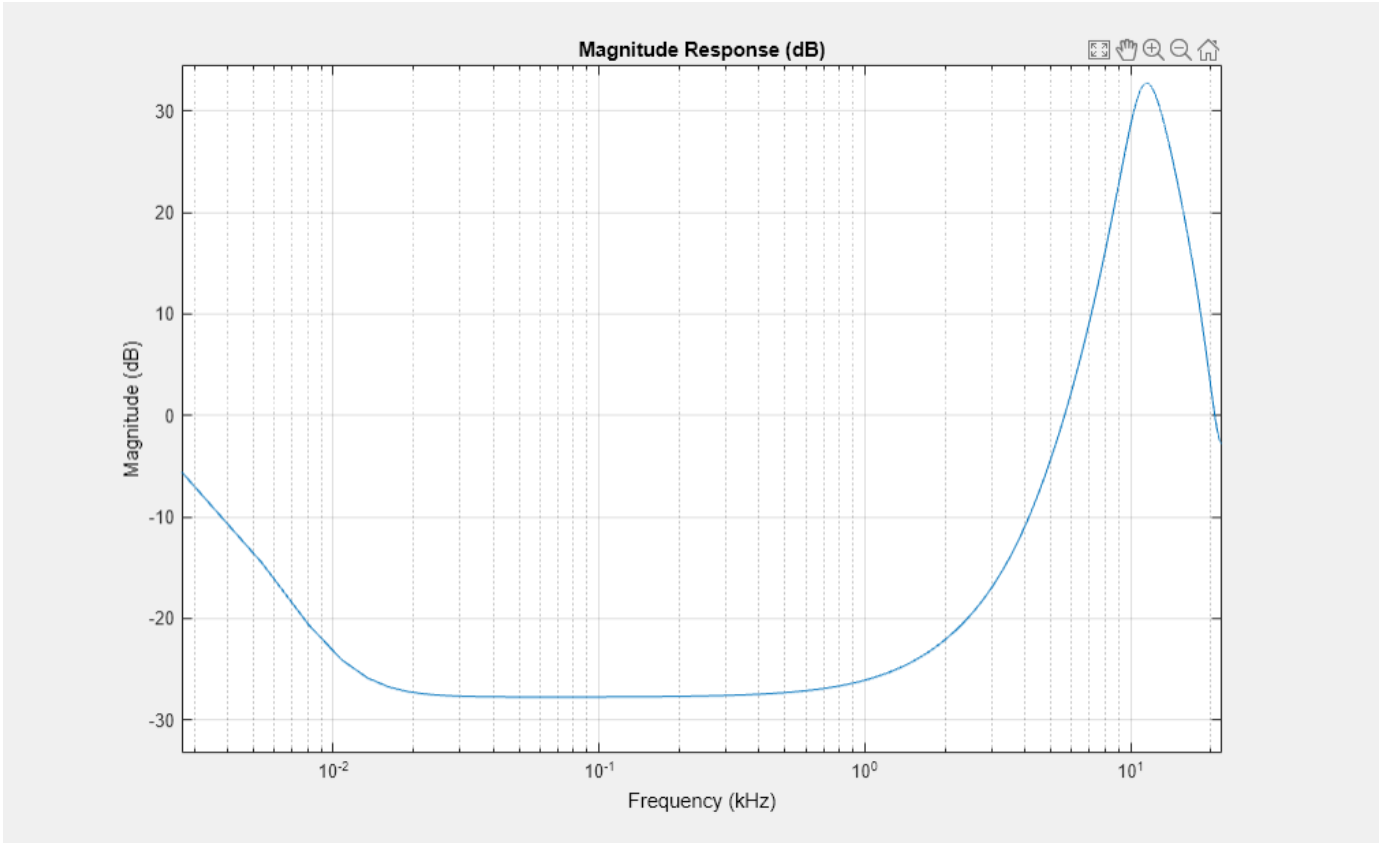
```
    deviceWriter(audio);
    count = count + 1;
end
reset(fileReader)
```

Design an SOS parametric equalizer suitable for use with `dsp.BiquadFilter`.

```
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. Call `designParamEQ` with the same design specifications. Specify the output orientation as `"row"` so that it is suitable for use with `fvtool`.

```
[Bvisualize,Avisualize] = designParamEQ(N,gain,centerFreq,bandwidth,Orientation="row");
fvtool([Bvisualize,Avisualize], ...
    Fs=fileReader.SampleRate, ...
    FrequencyScale="Log");
```



Create a biquad filter.

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

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

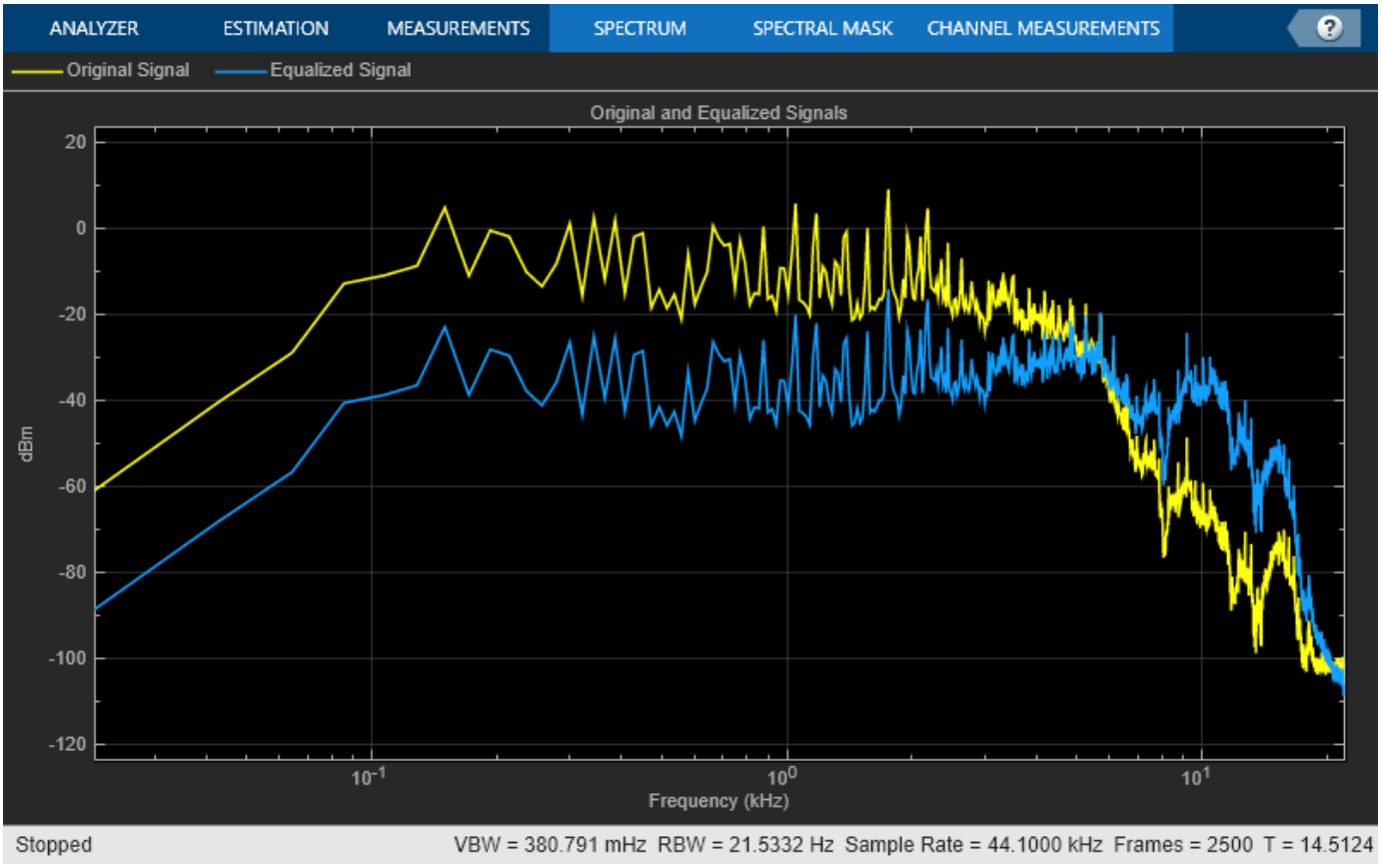
```
scope = spectrumAnalyzer( ...
    SampleRate=sampleRate, ...
    PlotAsTwoSidedSpectrum=false, ...
    FrequencyScale="log", ...
    Title="Original and Equalized Signals", ...
    ShowLegend=true, ...
    ChannelNames=["Original Signal","Equalized Signal"]);
```

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

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

As a best practice, release your objects once done.

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



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.

```
frameSize = 256;

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

sampleRate = fileReader.SampleRate;

deviceWriter = audioDeviceWriter( ...
    SampleRate=sampleRate);
```

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.

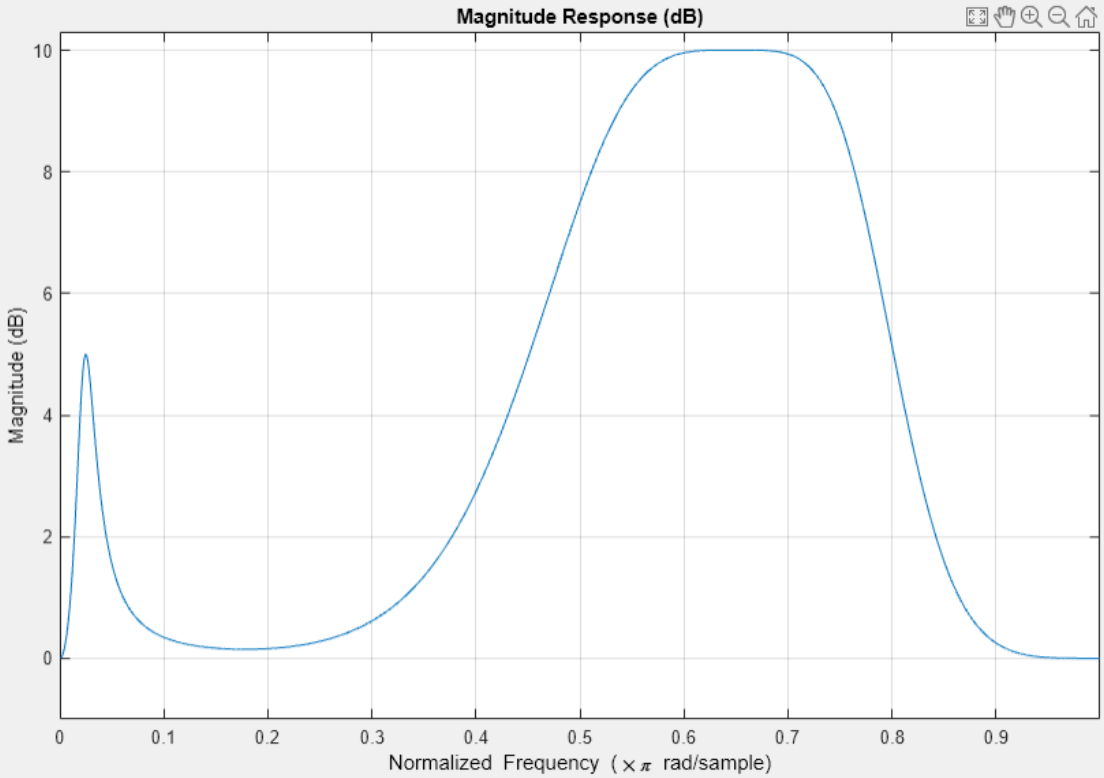
```
N = [2,4];
gain = [5,10];
centerFreq = [0.025,0.65];
bandwidth = [0.025,0.35];
mode = "fos";

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

Construct FOS IIR filters.

```
myFilter = dsp.FourthOrderSectionFilter(B,A);
```

Visualize the frequency response of your parametric equalizer.

```
fvtool(myFilter)
```



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

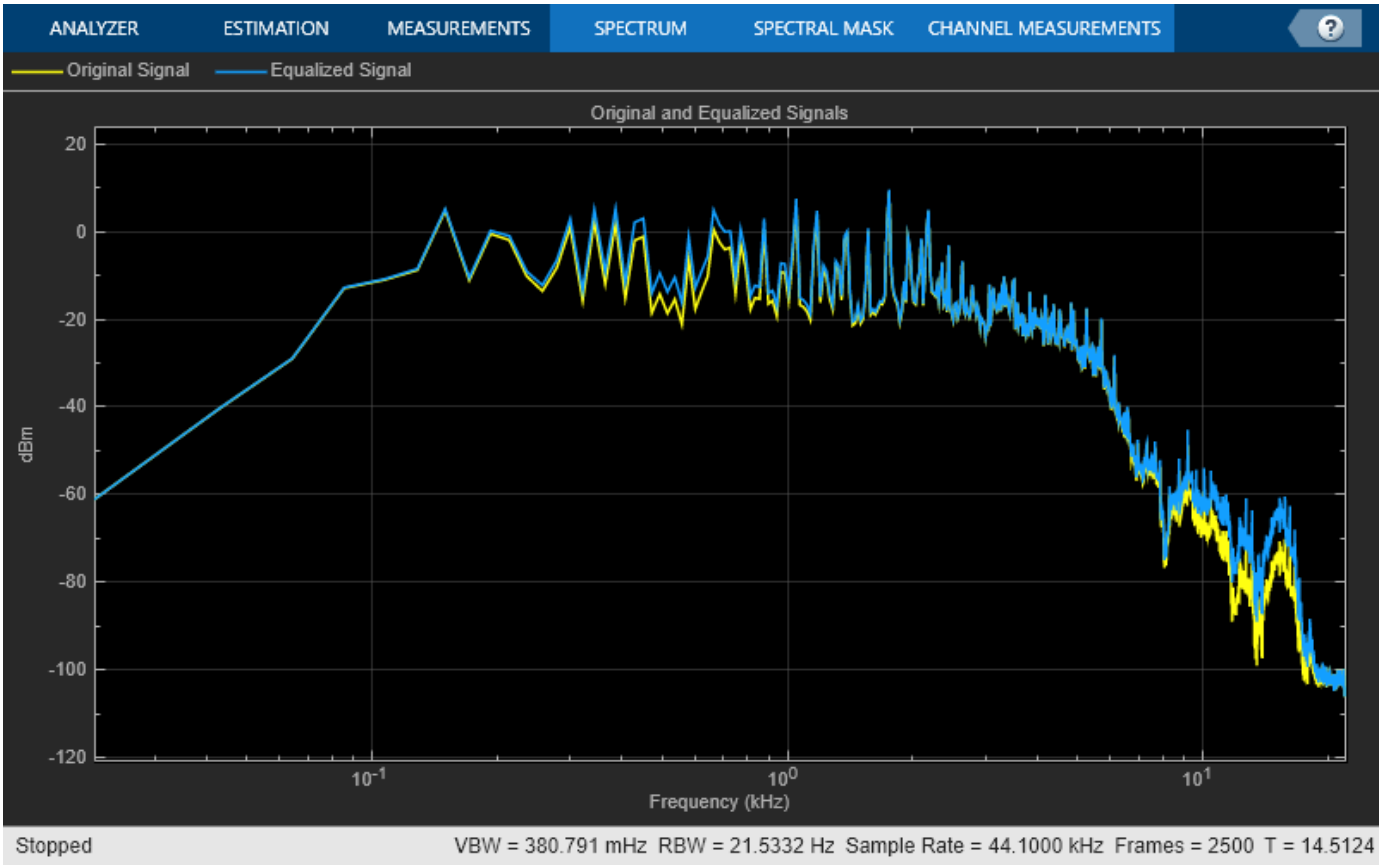
```
scope = spectrumAnalyzer( ...
    SampleRate=sampleRate, ...
    PlotAsTwoSidedSpectrum=false, ...
    FrequencyScale="log", ...
    Title="Original and Equalized Signals", ...
    ShowLegend=true, ...
    ChannelNames=["Original Signal","Equalized Signal"]);
```

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

```
count = 0;
while count < 2500
    x = fileReader();
    y = myFilter(x);

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

    deviceWriter(y);

    count = count + 1;
end
```

As a best practice, release your objects once done.

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



Input Arguments

`N` — Filter order

scalar | row vector

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

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

`gain` — Peak gain (dB)

scalar | row vector

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

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

`centerFreq` — Normalized center frequency of equalizer bands

scalar | row vector

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

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

`bandwidth` — Normalized bandwidth
scalar | row vector

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

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

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

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

Then convert to bandwidth

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

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

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

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

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

Data Types: `char` | `string`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `'Orientation',"row"`

`Orientation` — Orientation of returned filter coefficients
`"column"` (default) | `"row"`

Orientation of returned filter coefficients, specified as the comma-separated pair consisting of `'Orientation'` and `"column"` or `"row"`:

- Set `'Orientation'` to `"row"` for interoperability with **FVTool**, `dsp.DynamicFilterVisualizer`, and `dsp.FourthOrderSectionFilter`.
- Set `'Orientation'` to `"column"` for interoperability with `dsp.BiquadFilter`.

Data Types: `char` | `string`

Output Arguments

B — Numerator filter coefficients

matrix

Numerator filter coefficients, returned as a matrix. The size and interpretation of `B` depends on the `Orientation` and `mode`:

- If `'Orientation'` is set to `"column"` and `mode` is set to `"sos"`, then `B` is returned as an *L*-by-3 matrix. Each column corresponds to the numerator coefficients of your cascaded second-order sections.
- If `'Orientation'` is set to `"column"` and `mode` is set to `"fos"`, then `B` is returned as an *L*-by-5 matrix. Each column corresponds to the numerator coefficients of your cascaded fourth-order sections.
- If `'Orientation'` is set to `"row"` and `mode` is set to `"sos"`, then `B` is returned as a 3-by-*L* matrix. Each row corresponds to the numerator coefficients of your cascaded second-order sections.
- If `'Orientation'` is set to `"row"` and `mode` is set to `"fos"`, then `B` is returned as a 5-by-*L* matrix. Each row corresponds to the numerator coefficients of your cascaded fourth-order sections.

A — Denominator filter coefficients

matrix

Denominator filter coefficients, returned as a matrix. The size and interpretation of `A` depends on the `Orientation` and `mode`:

- If `'Orientation'` is set to `"column"` and `mode` is set to `"sos"`, then `A` is returned as an *L*-by-2 matrix. Each column corresponds to the denominator coefficients of your cascaded second-order sections. `A` does not include the leading unity coefficients.
- If `'Orientation'` is set to `"column"` and `mode` is set to `"fos"`, then `A` is returned as an *L*-by-4 matrix. Each column corresponds to the denominator coefficients of your cascaded fourth-order sections. `A` does not include the leading unity coefficients.
- If `'Orientation'` is set to `"row"` and `mode` is set to `"sos"`, then `A` is returned as a 3-by-*L* matrix. Each row corresponds to the denominator coefficients of your cascaded second-order sections.
- If `'Orientation'` is set to `"row"` and `mode` is set to `"fos"`, then `A` is returned as a 5-by-*L* matrix. Each row corresponds to the denominator coefficients of your cascaded fourth-order sections.

Version History

Introduced in R2016a

References

[1] Orfanidis, Sophocles J. "High-Order Digital Parametric Equalizer Design." *Journal of the Audio Engineering Society*. Vol. 53, November 2005, pp. 1026–1046.

Extended Capabilities

C/C++ Code Generation

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

See Also

`designVarSlopeFilter` | `designShelvingEQ` | `multibandParametricEQ` | `dsp.BiquadFilter`

Topics

“Parametric Equalizer Design”
“Equalization”

designShelvingEQ

Design shelving equalizer

Syntax

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

Description

`[B,A] = designShelvingEQ(gain,slope,Fc)` designs a low-shelf equalizer with the specified `gain`, `slope`, and cutoff frequency `Fc`. `B` and `A` are the numerator and denominator coefficients, respectively, of a single second-order section (biquad) IIR filter.

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

`[B,A] = designShelvingEQ( ___ ,Orientation=ornt)` specifies the orientation of the returned filter coefficients as `"column"` or `"row"`.

Examples

Filter Audio Using Low-Shelf Equalizer

Create audio file reader and audio device writer objects. Use the sample rate of the reader as the sample rate of the writer.

```
frameSize = 256;

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

deviceWriter = audioDeviceWriter(SampleRate=fileReader.SampleRate);
```

Play the audio signal through your device.

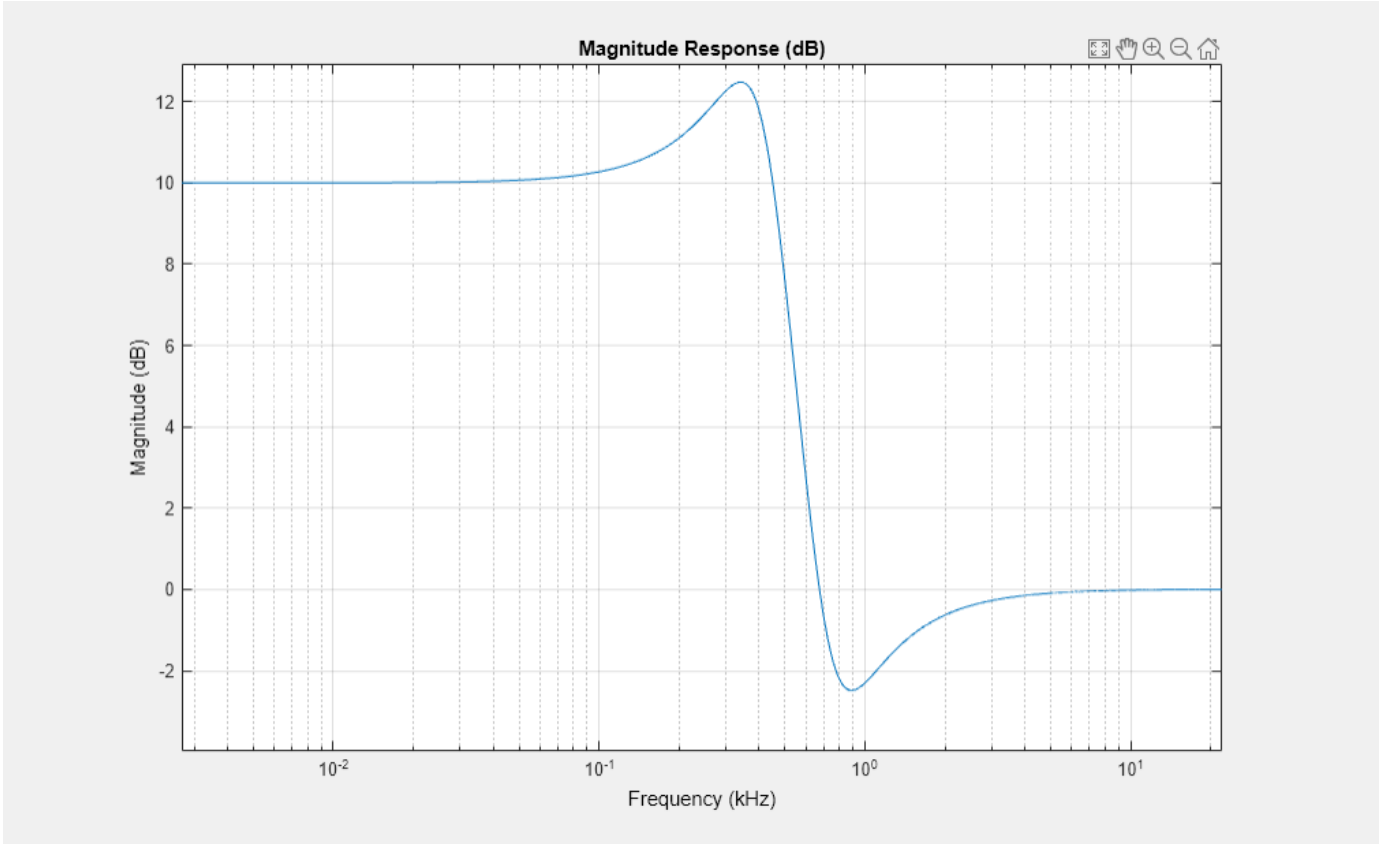
```
count = 0;
while count < 2500
    audio = step(fileReader);
    play(deviceWriter,audio);
    count = count + 1;
end
reset(fileReader)
```

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

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

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

Visualize your shelving filter design.

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



Create a biquad filter object.

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

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

```
scope = spectrumAnalyzer( ...
    SampleRate=fileReader.SampleRate, ...
    PlotAsTwoSidedSpectrum=false, ...
    FrequencyScale="log", ...
    Title="Original and Equalized Signal", ...
    ShowLegend=true, ...
    ChannelNames=["Original Signal","Equalized Signal"]);
```

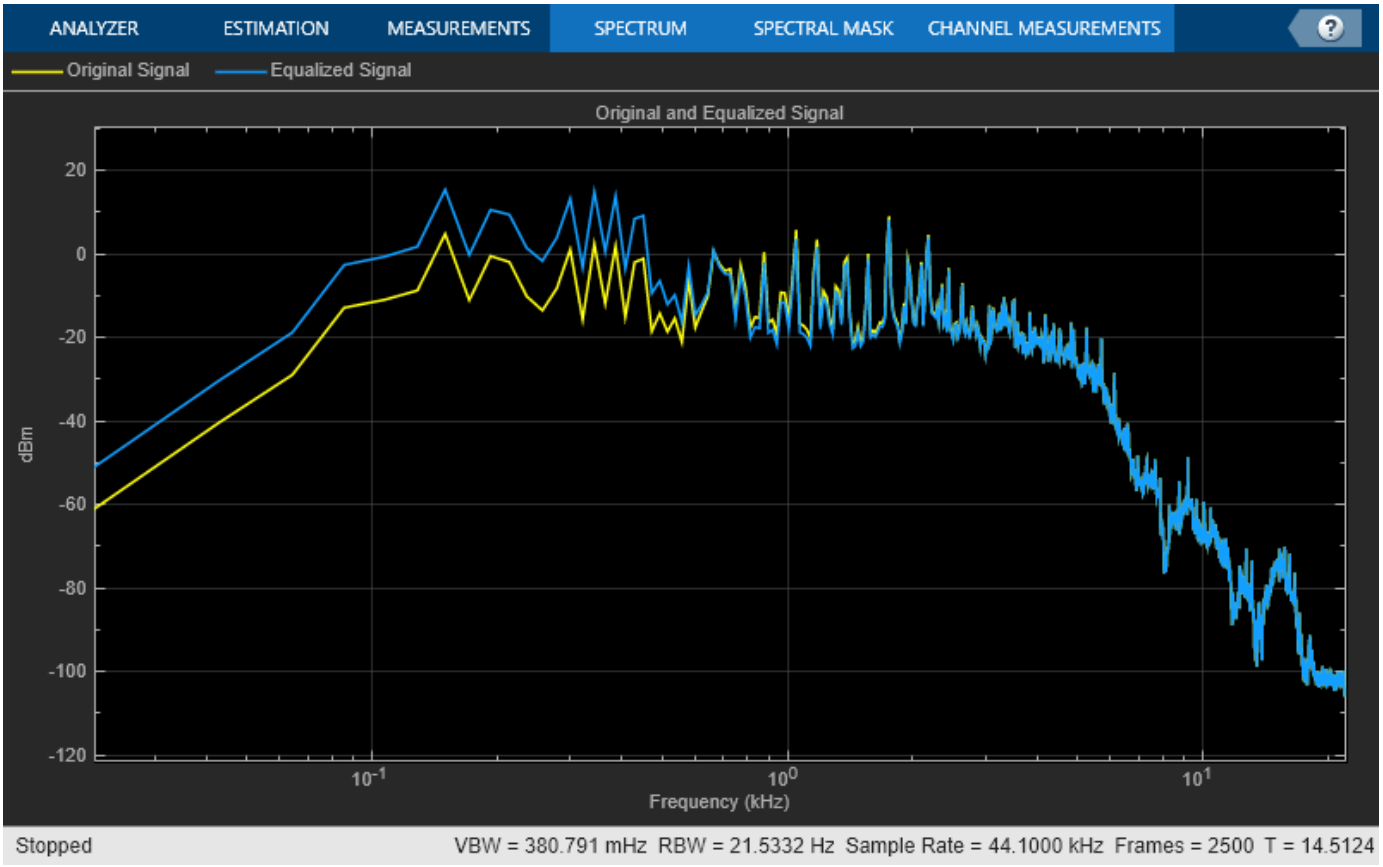
Play the equalized audio signal and visualize the original and equalized spectrums.

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

```
    originalSignal = fileReader();
    equalizedSignal = myFilter(originalSignal,B,A);
    scope([originalSignal(:,1),equalizedSignal(:,1)]);
    deviceWriter(equalizedSignal);
    count = count + 1;
end
```

As a best practice, release your objects once done.

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



Design High-Shelf Equalizer

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

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

```
Fs = 44.1e3;

gain1 = -6;
```

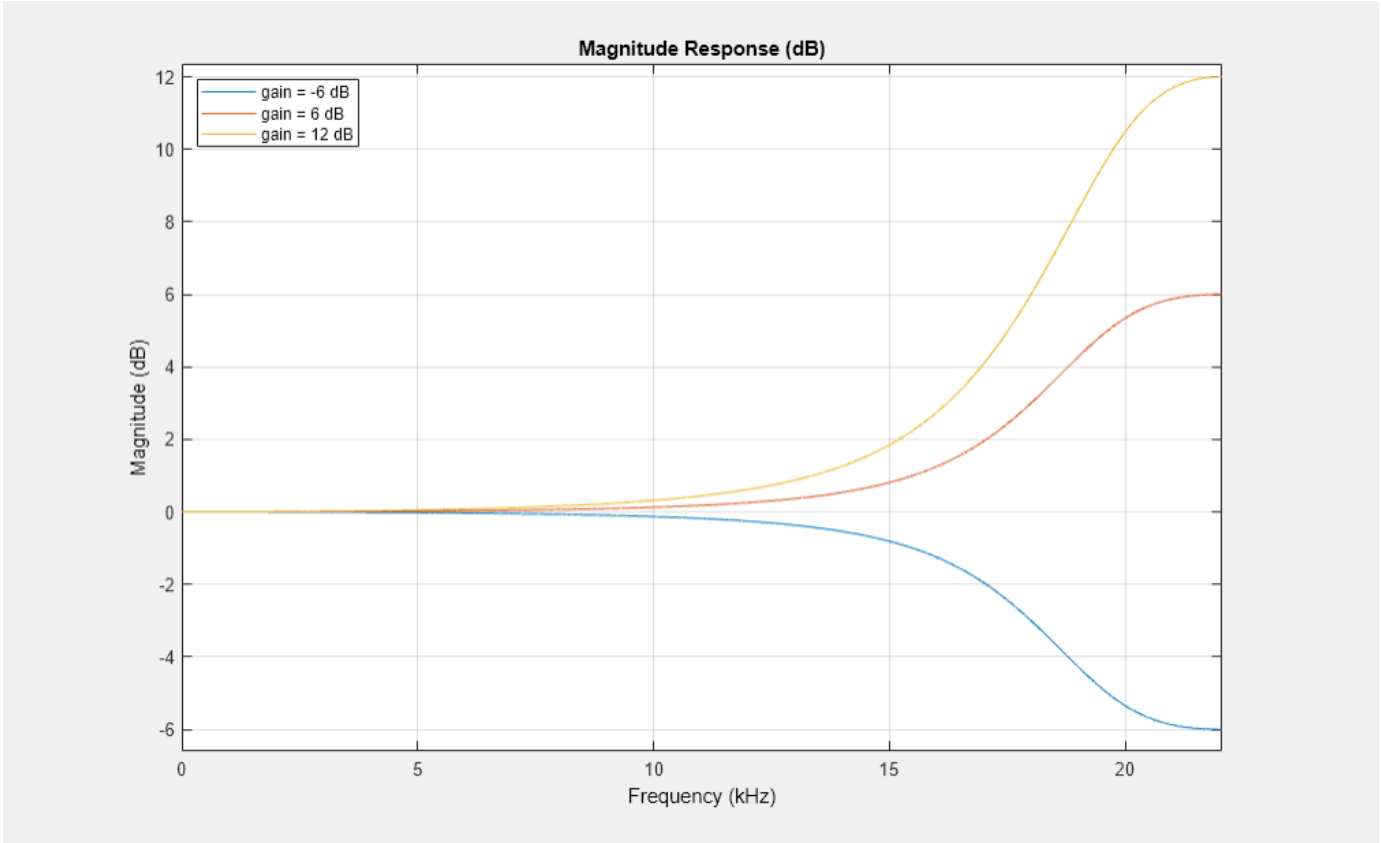
```
gain2 = 6;
gain3 = 12;

slope = 0.8;

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

Design the filter coefficients using the specified parameters.

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

Visualize your filter design.

```
fvt = fvtool([B1,A1;[1 0 0 1 0 0]], ...
    [B2,A2;[1 0 0 1 0 0]], ...
    [B3,A3;[1 0 0 1 0 0]], ...
    Fs=Fs);

legend(fvt,"gain = "+[gain1 gain2 gain3]+" dB",Location="northwest")
```



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.

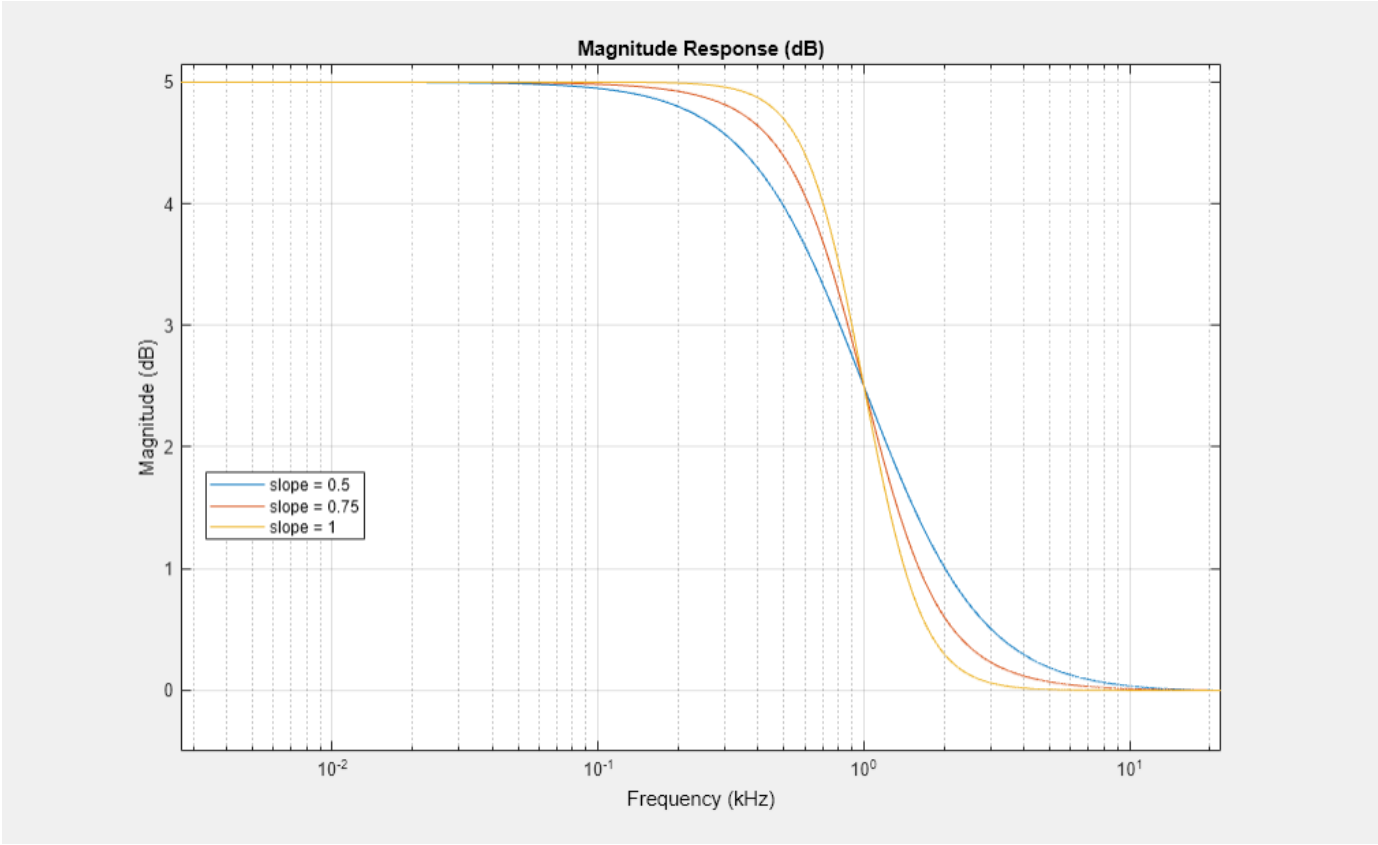
```
Fs = 44.1e3;

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,Orientation="row");
[B2,A2] = designShelvingEQ(gain,slope2,Fc,Orientation="row");
[B3,A3] = designShelvingEQ(gain,slope3,Fc,Orientation="row");
```

Visualize your filter design.

```
fvt = fvtool( ...
    dsp.BiquadFilter([B1,A1]), ...
    dsp.BiquadFilter([B2,A2]), ...
    dsp.BiquadFilter([B3,A3]), ...
    Fs=Fs, ...
    FrequencyScale="log");

legend(fvt,"slope = 0.5","slope = 0.75","slope = 1")
```



Input Arguments

`gain` — Peak gain (dB)

real scalar

Peak gain in dB, specified as a real scalar.

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

`slope` — Slope coefficient

positive scalar

Slope coefficient, specified as a positive scalar.

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

`Fc` — Normalized cutoff frequency

real scalar in the range `[0, 1]`

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

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

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

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

Filter type, specified as "lo" or "hi".

- "lo"-– Low shelving equalizer
- "hi"-– High shelving equalizer

Data Types: char | string

ornt — Orientation of returned filter coefficients
"column" (default) | "row"

Orientation of returned filter coefficients, specified as "column" or "row".

- Set ornt to "row" for interoperability with **FVTool**, dsp.DynamicFilterVisualizer, and dsp.FourthOrderSectionFilter.
- Set ornt to "column" for interoperability with dsp.BiquadFilter.

Data Types: char | string

Output Arguments

B — Numerator filter coefficients
three-element column vector | three-element row vector

Numerator filter coefficients, returned as a vector. The size and interpretation of B depend on the orientation, ornt:

- If ornt is set to "column", then B is returned as a three-element column vector.
- If ornt is set to "row", then B is returned as a three-element row vector.

.

A — Denominator filter coefficients
two-element column vector | three-element row vector

Denominator filter coefficients of the designed second-order IIR filter, returned as a vector. The size and interpretation of A depend on the orientation, ornt:

- If ornt is set to "column", then A is returned as a two-element column vector. A does not include the leading unity coefficient.
- If ornt is set to "row", then A is returned as a three-element row vector.

Version History

Introduced in R2016a

References

[1] Bristow-Johnson, Robert. "Cookbook Formulae for Audio EQ Biquad Filter Coefficients." Accessed September 13, 2021. https://webaudio.github.io/Audio-EQ-Cookbook/Audio-EQ-Cookbook.txt.

Extended Capabilities

C/C++ Code Generation

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

See Also

Blocks

Shelving Filter

Objects

`shelvingFilter` | `multibandParametricEQ`

Functions

`designParamEQ` | `designVarSlopeFilter`

Topics

“Parametric Equalizer Design”
“Equalization”

designVarSlopeFilter

Design variable slope lowpass or highpass IIR filter

Syntax

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

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.

`[B,A] = designVarSlopeFilter(___,Name,Value)` specifies options using one or more `Name,Value` pair arguments.

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.

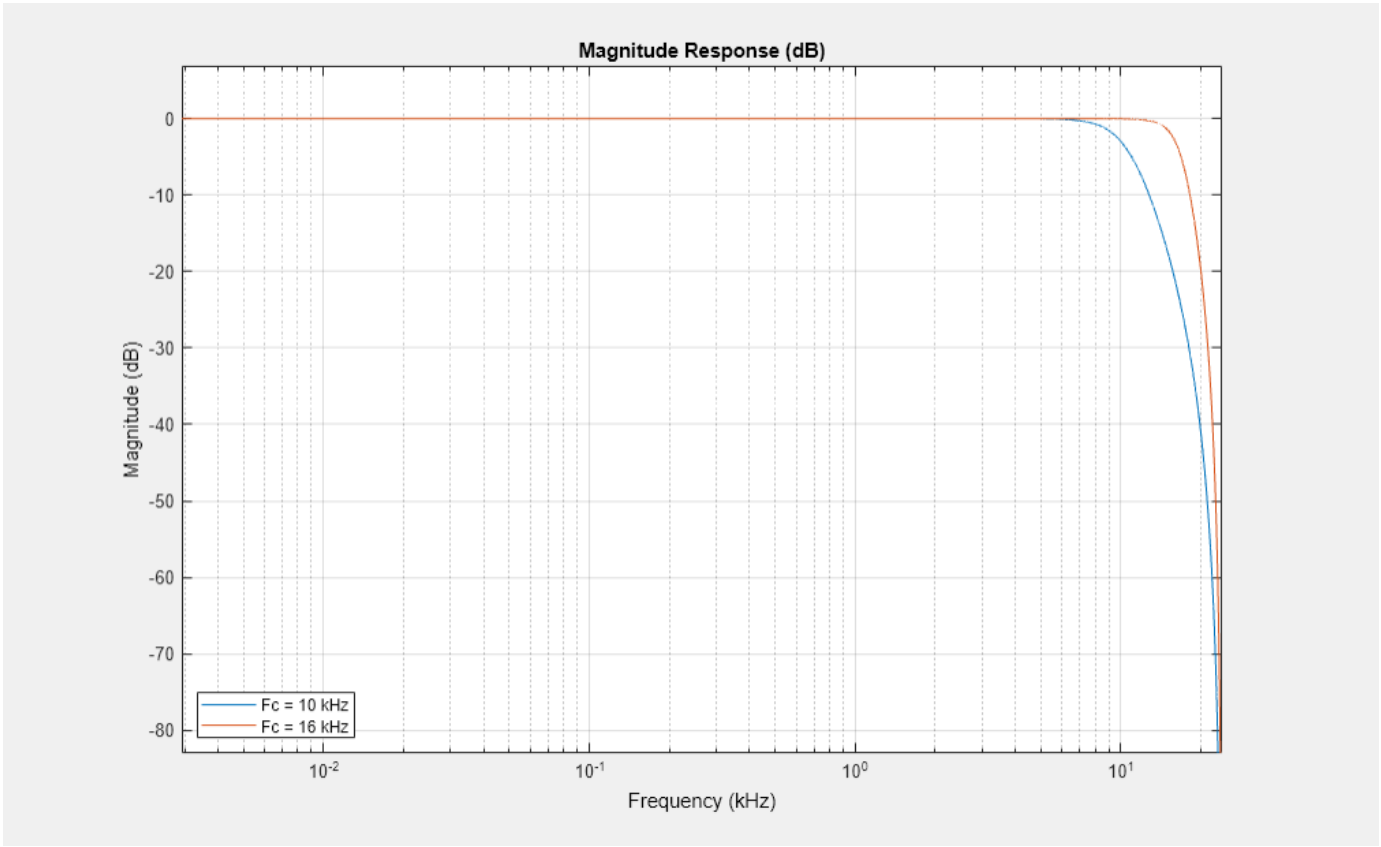
```
Fs = 48e3;

slope = 18;

Fc1 = 10e3/(Fs/2);
Fc2 = 16e3/(Fs/2);
```

Design the filter coefficients using the specified parameters.

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

Visualize your filter design.

```
fvt = fvtool([B1,A1],[B2,A2],Fs=Fs,FrequencyScale="log");

legend(fvt,"Fc = 10 kHz","Fc = 16 kHz",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.

Create audio file reader and audio device writer objects. Use the sample rate of the reader as the sample rate of the writer.

```
frameSize = 256;

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

sampleRate = fileReader.SampleRate;

deviceWriter = audioDeviceWriter( ...
    SampleRate=sampleRate);
```

Play the audio signal through your device.

```
count = 0;
while count < 2500
    audio = fileReader();
    deviceWriter(audio);
```

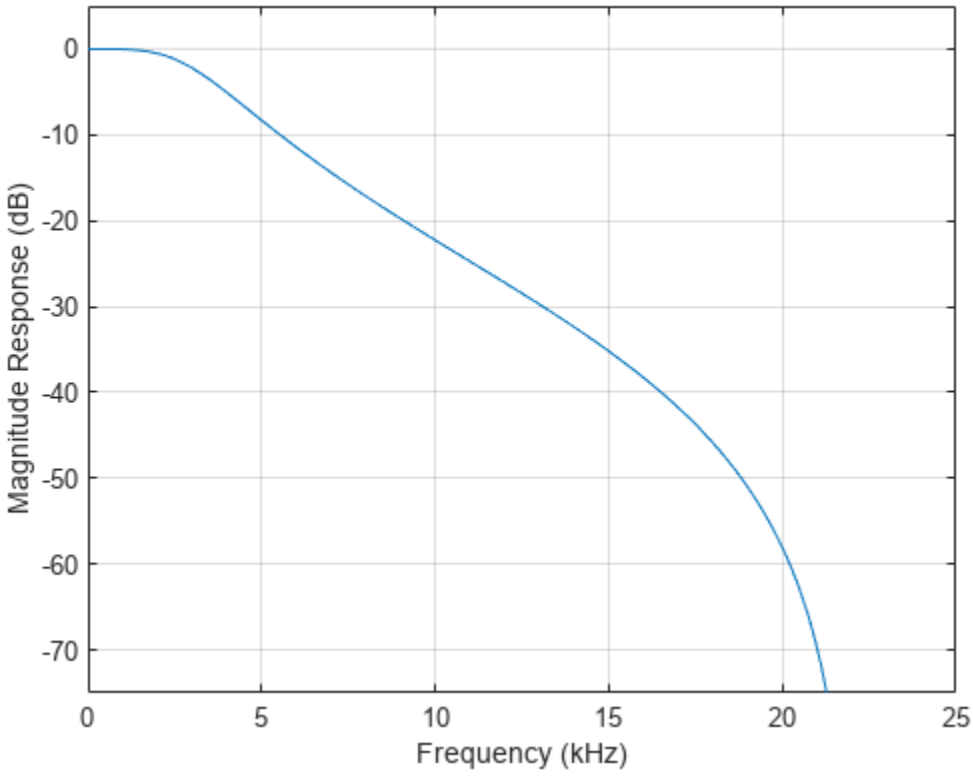
```
    count = count + 1;
end
reset(fileReader)
```

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

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

Visualize your filter design. To output filter coefficients suitable for `freqz`, call `designVarSlopeFilter` again with the same design specifications but with `Orientation` set to `"row"`.

```
[Bvisualize,Avisualize] = designVarSlopeFilter(slope,cutoff,Orientation="row");

[h,f] = freqz([Bvisualize Avisualize],[],sampleRate);
plot(f/1000,mag2db(abs(h)))
grid
ylim([-75 5])
xlabel("Frequency (kHz)")
ylabel("Magnitude Response (dB)")
```



Create a biquad filter.

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

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

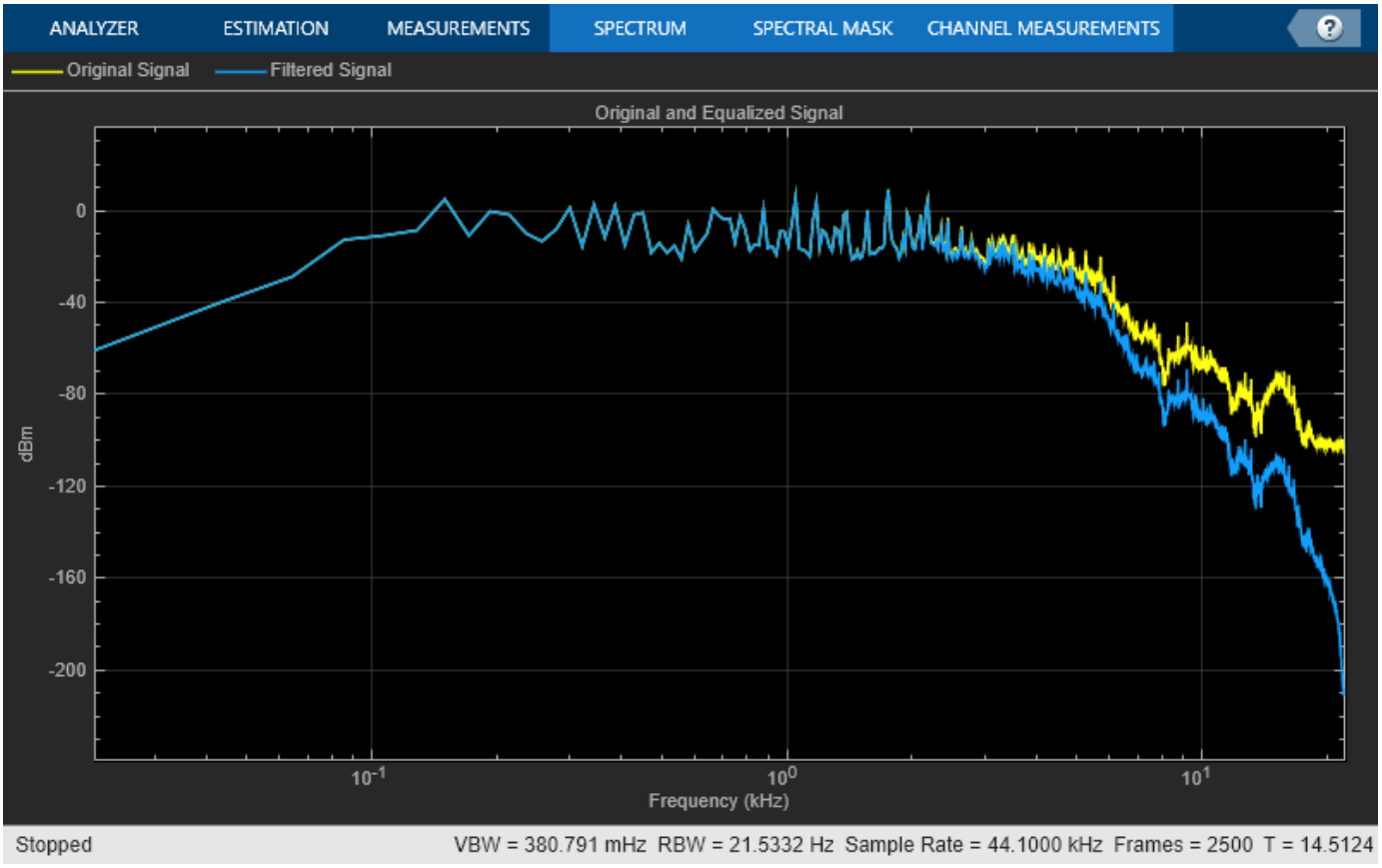
```
scope = spectrumAnalyzer( ...
    SampleRate=sampleRate, ...
    PlotAsTwoSidedSpectrum=false, ...
    FrequencyScale="log", ...
    Title="Original and Equalized Signal", ...
    ShowLegend=true, ...
    ChannelNames={'Original Signal','Filtered Signal'});
```

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

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

As a best practice, release your objects once done.

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



Design Highpass IIR Filter

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

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

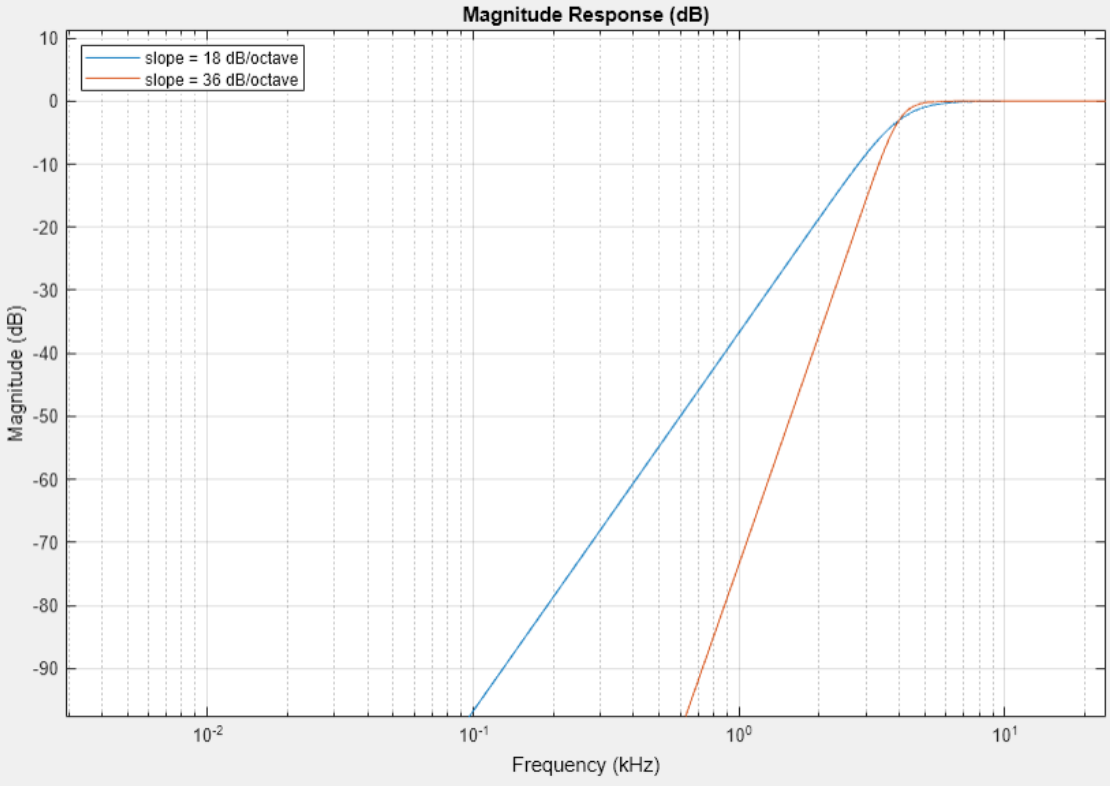
```
Fs = 48e3;
slope1 = 18;
slope2 = 36;
Fc = 4000/(Fs/2);
```

Design the filter coefficients using the specified parameters.

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

Visualize your filter design.

```
fvt = fvtool([B1,A1],[B2,A2],...
       "Fs",Fs,...
       "FrequencyScale","Log");
legend(fvt,"slope = 18 dB/octave", ...
       "slope = 36 dB/octave", ...
       "Location","NorthWest")
```



Diminish Plosives from Speech Signal

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

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

```
fileReader = dsp.AudioFileReader('audioPlosives.wav');
deviceWriter = audioDeviceWriter('SampleRate',fileReader.SampleRate);

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` 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/(fileReader.SampleRate/2),"hi");
biquadFilter = dsp.BiquadFilter( ...
    "SOSMatrixSource","Input port", ...
    "ScaleValuesInputPort",false);

crossFilt = crossoverFilter( ...
    "SampleRate",fileReader.SampleRate, ...
    "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 = timescope( ...
    "SampleRate",fileReader.SampleRate, ...
    "TimeSpanSource","property","TimeSpan",3, ...
    "BufferLength",fileReader.SampleRate*3*2, ...
    "YLimits",[-1 1], ...
    "ShowGrid",true, ...
```

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

In an audio stream loop:

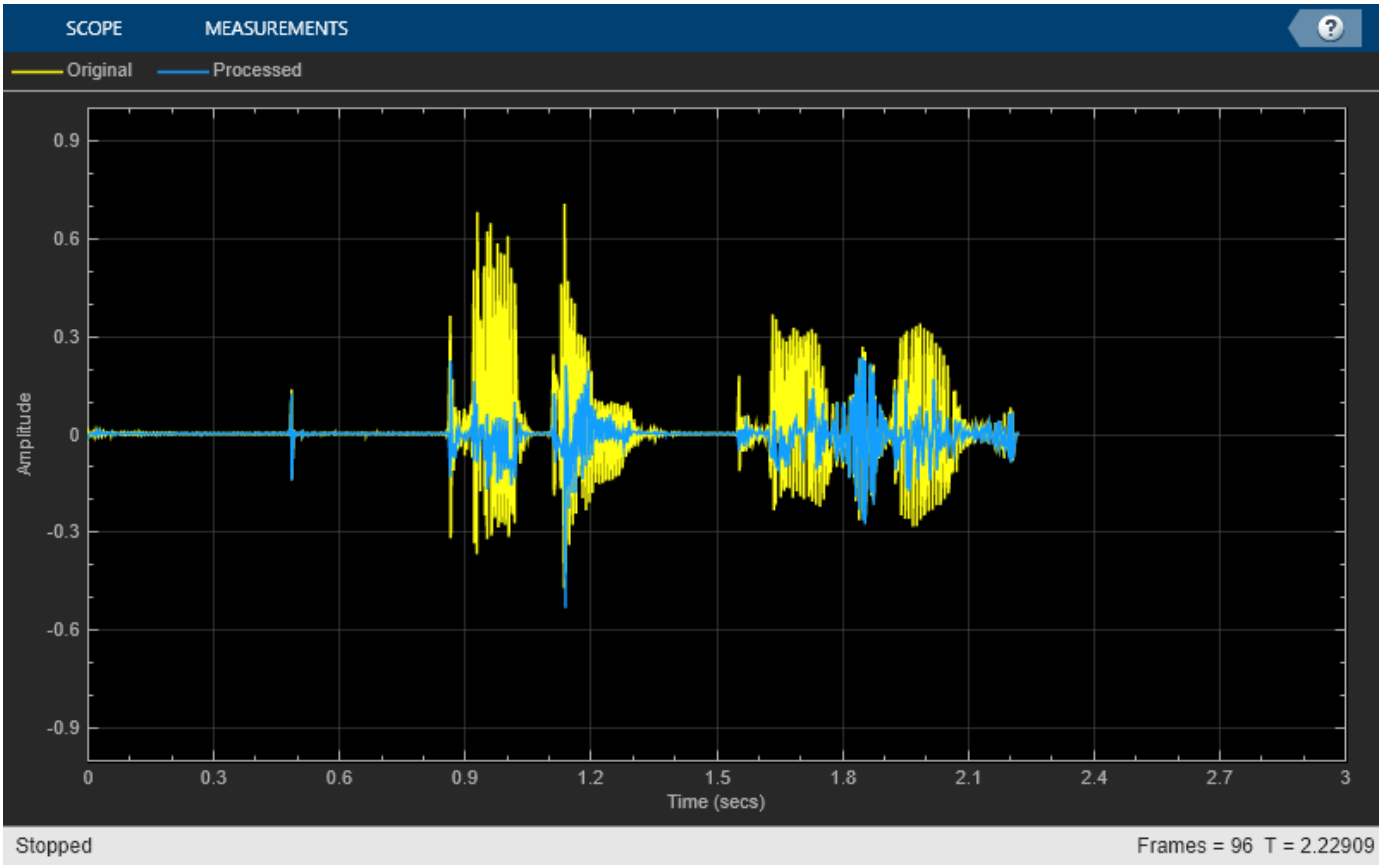
1. Read in a frame of the audio file.
2. Apply highpass filtering using your biquad filter.
3. Split the audio signal into two bands.
4. Apply dynamic range compression to the lower band.
5. Remix the channels.
6. Write the processed audio signal to your audio device for listening.
7. Visualize the processed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

As a best practice, release your objects once done.

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



Input Arguments

`slope` — Filter slope (dB/octave)

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

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

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

`Fc` — Normalized cutoff frequency

real scalar in the range 0 to 1

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

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

`type` — Filter type

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

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

- `'lo'`–– Lowpass filter

- `'hi'`-- Highpass filter

Data Types: `char` | `string`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `'Orientation',"row"`

`Orientation` — Orientation of returned filter coefficients

`"column"` (default) | `"row"`

Orientation of returned filter coefficients, specified as the comma-separated pair consisting of `'Orientation'` and `"column"` or `"row"`:

- Set `'Orientation'` to `"row"` for interoperability with **FVTool**, `dsp.DynamicFilterVisualizer`, and `dsp.FourthOrderSectionFilter`.
- Set `'Orientation'` to `"column"` for interoperability with `dsp.BiquadFilter`.

Data Types: `char` | `string`

Output Arguments

`B` — Numerator filter coefficients

3-by-4 matrix | 4-by-3 matrix

Numerator filter coefficients, returned as a matrix. The size and interpretation of `B` depends on the `Orientation`:

- If `'Orientation'` is set to `"column"`, then `B` is 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.
- If `'Orientation'` is set to `"row"`, then `B` is returned as a 4-by-3 matrix. Each row 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 | 4-by-3 matrix

Denominator filter coefficients, returned as a matrix. The size and interpretation of `A` depends on the `Orientation`:

- If `'Orientation'` is set to `"column"`, then `A` is 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.
- If `'Orientation'` is set to `"row"`, then `B` is returned as a 4-by-3 matrix. Each row of `B` corresponds to the denominator coefficients of a different second-order section of your cascaded IIR filter.

Version History

Introduced in R2016a

References

[1] Orfanidis, Sophocles J. "High-Order Digital Parametric Equalizer Design." *Journal of the Audio Engineering Society*. Vol. 53, November 2005, pp. 1026–1046.

Extended Capabilities

C/C++ Code Generation

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

See Also

`designShelvingEQ` | `designParamEQ` | `multibandParametricEQ`

Topics

“Parametric Equalizer Design”
“Equalization”

disconnectMIDI

Disconnect MIDI controls from audio object

Syntax

```
disconnectMIDI(audioObject)
```

Description

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

Examples

Disconnect MIDI Controls from Audio Plugin

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

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

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

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

```
ans =

     1
```

Add MIDI connections using `configureMIDI`.

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

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

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

```
ans =

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

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

```
disconnectMIDI(echoPlugin);
```

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

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

```
ans =

     1
```

Input Arguments

`audioObject` — Audio object

object

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

Version History

Introduced in R2016a

See Also

Classes

`audioPlugin` | `audioPluginSource`

Functions

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

Topics

“MIDI Control for Audio Plugins”
“MIDI Control Surface Interface”

fdesign.parameq

(To be removed) Parametric equalizer filter specification

Compatibility

Note The `fdesign.parameq` filter specification object will be removed in a future release. Existing instances of the object continue to run. For new code, use the `designParamEQ` function instead. For more information, see “Compatibility Considerations” on page 2-562.

Syntax

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

Description

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

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

where the parameters are defined as follows:

Parameter	Definition	Unit
`BW`	Bandwidth	
`BWp`	Passband Bandwidth	
`BWst`	Stopband Bandwidth	

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

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

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

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

Examples

Design Parametric Equalizers

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

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

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

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

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

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

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

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

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

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

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

Version History

R2022a: `fdesign.parameq` will be removed
Warns starting in R2022a

The `fdesign.parameq` filter specification object will be removed in a future release. Existing instances of the object continue to run. For new code, use the `designParamEQ` function instead.

Update Code

This table shows how the object is typically used and explains how to update the existing code to use the `designParamEQ` function.

Discouraged Usage	Recommended Replacement

Design based on Filter Bandwidth
 	
`Fs  = 48e3;`
`N   = 2;`
`Q   = 10;`
`G   = 9; % 9 dB`
 	
`% Normalized center frequency`
`Wo1 = 2000/(Fs/2);`
`Wo2 = 12000/(Fs/2);`
 	
`% Normalized bandwidth`
`BW1 = Wo1/Q;`
`BW2 = Wo2/Q;`
 	
`PEQ = fdesign.parameq('N,F0,BW,Gref,G0,GBW',N,F0,BW1,0,G,4.5);`
`BQ1 = design(PEQ,'SystemObject',true);`
 	
`PEQ.BW = BW2;`
`PEQ.F0 = Wo2;`
`BQ2 = design(PEQ,'SystemObject',true);`
 	
`% Visualize the filters`
`hfvt = fvtool(BQ1,BQ2,'Fs',Fs,'Color','white');`
`legend(hfvt,'BW1 = 200 Hz; Q = 10','BW2 = 1200 Hz; Q = 10');`
	Design based on Filter Bandwidth
	
	`Fs  = 48e3;`
	`N   = 2;`
	`Q   = 10;`
	`G   = 9; % 9 dB`
	
	`% Normalized center frequency`
	`Wo1 = 2000/(Fs/2);`
	`Wo2 = 12000/(Fs/2);`
	
	`% Normalized bandwidth`
	`BW1 = Wo1/Q;`
	`BW2 = Wo2/Q;`
	
	`[B1,A1] = designParamEQ(N,G,Wo1,BW1);`
	`[B2,A2] = designParamEQ(N,G,Wo2,BW2);`
	`BQ1 = dsp.BiquadFilter('SOSMatrix',[B1.',[1,A1.']]);`
	`BQ2 = dsp.BiquadFilter('SOSMatrix',[B2.',[1,A2.']]);`
	
	`% Visualize the filters`
	`hfvt = fvtool(BQ1,BQ2,'Fs',Fs,'Color','white');`
	`legend(hfvt,'BW1 = 200 Hz; Q = 10','BW2 = 1200 Hz; Q`

Design based on Quality factor
 	
`Fs  = 48e3;`
`N   = 2;`
`G   = 15; % 15 dB`
 	
`% Quality factor`
`Q1 = 0.48;`
`Q2 = 1/sqrt(2);`
 	
`% Normalized center frequency`
`% F0 = 1 designs a highpass filter`
`% F0 can either be 0 or 1 in this configuration`
`F0 = 1;`
 	
`% Cutoff Frequency`
`Fc = 6e3/(Fs/2);`
 	
`PEQ = fdesign.parameq('N,F0,Fc,Qa,G0',N,F0,Fc,Q1,G);`
`BQ1 = design(PEQ,'SystemObject',true);`
 	
`PEQ.Qa = Q2;`
`BQ2 = design(PEQ,'SystemObject',true);`
 	
`% Visualize the filters`
`hfvt = fvtool(BQ1,BQ2,'Fs',Fs,'Color','white');`
`legend(hfvt,'Q = 0.48','Q = 0.7071');`
	Design based on Quality factor
	
	`Fs  = 48e3;`
	`N   = 2;`
	`G   = 15; % 15 dB`
	
	`% Quality factor`
	`Q1 = 0.48;`
	`Q2 = 1/sqrt(2);`
	
	`% Normalized center frequency`
	`Wo  = 6000/(Fs/2);`
	`% Normalized bandwidth`
	`BW1 = Wo/Q1;`
	`BW2 = Wo/Q2;`
	
	`[B1,A1] = designParamEQ(N,G,Wo,BW1);`
	`[B2,A2] = designParamEQ(N,G,Wo,BW2);`
	`BQ1 = dsp.BiquadFilter('SOSMatrix',[B1.',[1,A1.']]);`
	`BQ2 = dsp.BiquadFilter('SOSMatrix',[B2.',[1,A2.']]);`
	
	`% Visualize the filters`
	`hfvt = fvtool(BQ1,BQ2,'Fs',Fs,'Color','white');`
	`legend(hfvt,'Q = 0.48','Q = 0.7071');`

Discouraged Usage	Recommended Replacement

Low shelf and high shelf filters
 	
`Fs  = 48e3;`
`N   = 4;`
`G   = 10; % 10 dB`
 	
`% Normalized center frequency`
`Wo1 = 0; % Lowpass filter`
`% Corresponds to Fs/2 (Hz) or pi (rad/sample)`
`Wo2 = 1; % Highpass filter`
 	
`% Bandwidth occurs at 7.4 dB in this case`
`BW = 1000/(Fs/2);`
 	
`PEQ = fdesign.parameq('N,F0,BW,Gref,G0,GBW');`
`BQ1 = design(PEQ,'SystemObject',true);`
 	
`PEQ.F0 = Wo2;`
`BQ2 = design(PEQ,'SystemObject',true);`
 	
`% Visualize the filters`
`hfvt = fvtool(BQ1,BQ2,'Fs',Fs,'Color','white');`
`legend(hfvt,'Low Shelf Filter','High Shelf Filter');`
	Low shelf and high shelf filters
	
	`Fs  = 48e3;`
	`N   = 4;`
	`G   = 10; % 10 dB`
	
	`% Normalized center frequency`
	`Wo1 = 0; % Lowpass filter`
	`% Corresponds to Fs/2 (Hz) or pi (rad/sample)`
	`Wo2 = 1; % Highpass filter`
	
	`% Bandwidth occurs at 7.4 dB in this case`
	`BW = 1000/(Fs/2);`
	
	`[B1,A1] = designParamEQ(N,G,Wo1,BW);`
	`[B2,A2] = designParamEQ(N,G,Wo2,BW);`
	`BQ1 = dsp.BiquadFilter('SOSMatrix',[B1.',[ones(2,1),A`
	`BQ2 = dsp.BiquadFilter('SOSMatrix',[B2.',[ones(2,1),A`
	
	`% Visualize the filters`
	`hfvt = fvtool(BQ1,BQ2,'Fs',Fs,'Color','white');`
	`legend(hfvt,'Low Shelf Filter','High Shelf Filter');`

See Also

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

Topics

“Parametric Equalizer Design”
“Equalization”

generateAudioPlugin

Generate audio plugin from MATLAB class

Syntax

```
generateAudioPlugin className
generateAudioPlugin options className
generateAudioPlugin
```

Description

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

`generateAudioPlugin options className` specifies a nondefault plugin type, output folder, file name, or file type. You can use the `-juceproject` option to create a zip file containing generated C/C++ code and a JUCER project. Options can be specified in any grouping, and in any order.

`generateAudioPlugin` with no input arguments opens a user interface (UI) to generate and validate an audio plugin. The UI provides functionality equivalent to the command-line interfaces of `generateAudioPlugin`, `audioPluginConfig`, and `validateAudioPlugin`.

- The **Audio plugin class name** corresponds to the `className` input argument.
- The **Validation options** section corresponds to the `options` argument of `validateAudioPlugin`.
- The **Generation options** section corresponds to the `options` argument of `generateAudioPlugin`.
- The **Coder configuration** section corresponds to the “Properties” on page 4-256 of `audioPluginConfig`.

Examples

Generate Audio Plugin

```
generateAudioPlugin audiopluginexample.Echo
```

```
.......
```

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

Specify Output Folder for Generated Plugin

```
mkdir(fullfile(pwd,'myPluginFolder'))
generateAudioPlugin -outdir myPluginFolder audiopluginexample.Echo
```

```
.......
```

A VST 2 plugin with file name `Echo` is saved to your specified folder, `myPluginFolder`. 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 with file name `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

```
mkdir(fullfile(pwd,'myPluginFolder'))
generateAudioPlugin -output coolEffect -outdir myPluginFolder audiopluginexample.Echo
```

```
.......
```

A VST 2 plugin with file name `coolEffect` is saved to your specified folder, `myPluginFolder`. 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 with file name `Echo.dll` is saved to your current folder.

Generate JUCE-Compatible Zip File

```
generateAudioPlugin -juceproject audiopluginexample.Echo
```

A zip file containing generated C/C++ code and a JUCER project file suitable for use with JUCE 5.3.2 to 6.0.8 is saved to your current folder.

Generate Standalone Executable

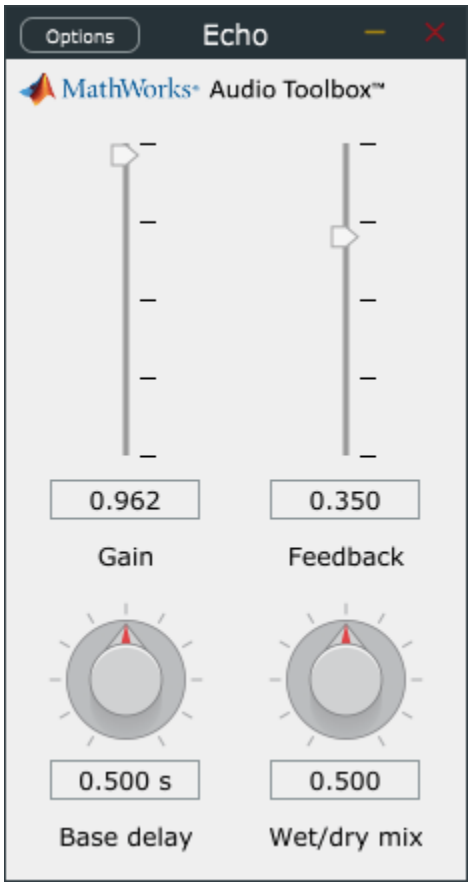
To generate a binary standalone executable, use the `-exe` option. The following command saves `Echo.exe` to your current folder.

```
generateAudioPlugin -exe audiopluginexample.Echo
```

. . . . . .

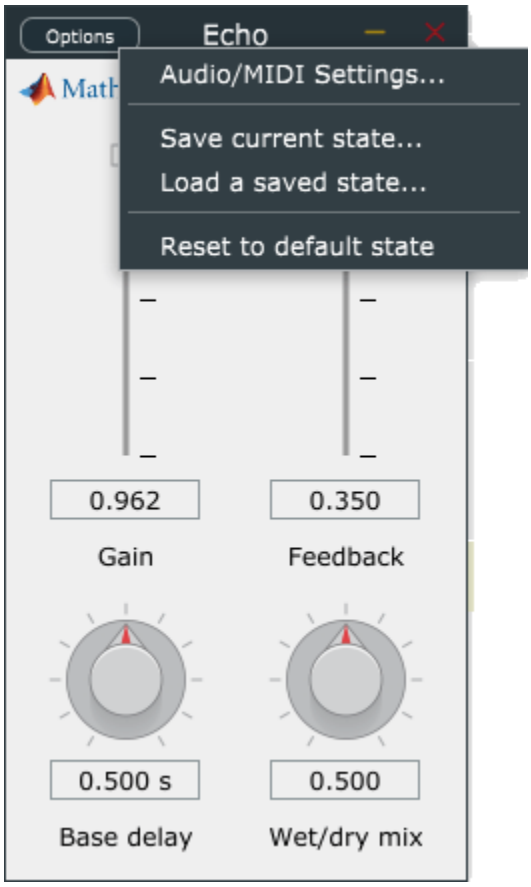
When you execute the generated code, the UI you defined in your audio plugin opens.

```
eval('!Echo.exe')
```



The standalone executable enables you to:

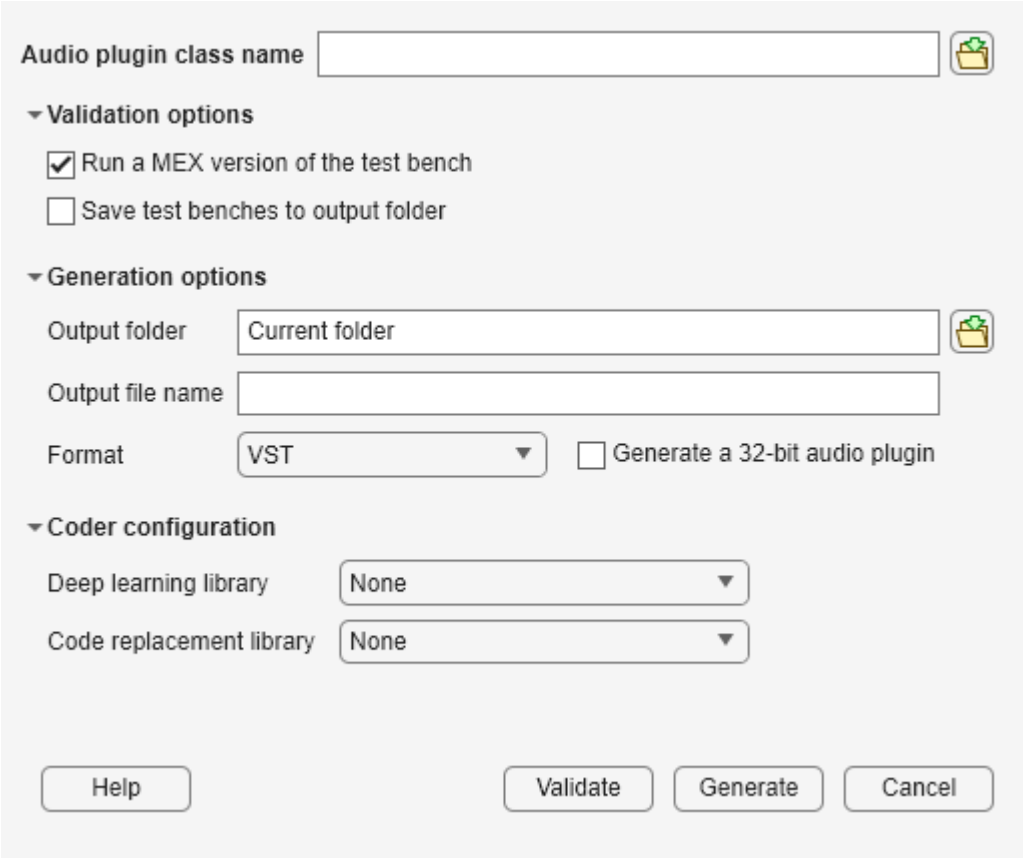
- Configure audio input and output from the plugin. Synchronizing parameters with MIDI devices is not currently supported.
- Save and load states.
- Reset states to default values.



Generate and Validate Audio Plugin Through UI

To open the UI, call `generateAudioPlugin` with no input arguments.

```
generateAudioPlugin
```



Type "`audiopluginexample.Echo`" into the **Audio plugin class name** field. Click **Validate** to validate the plugin. Click **Generate** to generate the plugin in the location specified by the **Output folder** field.

Generate and Register AUv3 Plugin on macOS

Use the `-auv3` option to generate an AUv3 plugin containing app.

```
generateAudioPlugin -auv3 audiopluginexample.Echo
```

Run the generated app to register the plugin with the macOS system. This also opens a window containing the plugin interface. You can close the app once it successfully opens.

Use the `system` function to run the `auval` macOS command and verify that the plugin was successfully registered.

```
[status,output] = system("auval -a | grep Echo")
```

```
status =

     0


output =
```

```
'aufx 4pvz Math  -  MathWorks: Echo
 '
```

Input Arguments

`options` — Options to specify output folder, plugin name, and file type

`-au` | `-auv3` | `-vst` | `-vst3` | `-exe` | `-juceproject` | `-output` *`fileName`* | `-outdir` *`folder`* | `-win32` | `-mac64universal` | `-audioconfig` *`cfg`*

Options to specify output folder, plugin name, and file type, specified as one of the values in the table. You can specify options in any order and group them.

Option	UI Setting	Description
`-au`	Set **Format** to `AU`	Generates an Audio Unit (AU) v2 audio plugin binary. This option is valid only on macOS.
`-auv3`	Set **Format** to `AUv3`	Generates a containing app for an AUv3 plugin. Run this app to register the AUv3 plugin in your macOS system. The generated plugin is ad-hoc code signed. You can use the `codesign` command in the macOS command line to manually code sign the plugin with your own certificate. Use the `--force` option to ensure that the ad-hoc signature is replaced. For more information, see the Apple documentation, Code Signing Guide. This option is valid only on macOS.
`-vst`	Set **Format** to `VST`	Generates a VST 2 audio plugin binary. By default, `generateAudioPlugin` generates a VST 2 plugin.
`-vst3`	Set **Format** to `VST3`	Generates a VST 3 audio plugin binary. This option adds a `Bypass` parameter to the plugin.
`-exe`	Set **Format** to `Standalone executable`	Generates a standalone executable for your audio plugin. When you evaluate the generated code, the UI you defined in your audio plugin opens. You can control the input to your plugin and the output from your plugin using **Options**.
`-juceproject`	Set **Format** to `JUCE project`	Creates a zip file containing generated C/C++ code and a JUCER project file suitable for use with JUCE 5.3.2 to 6.0.8. You can use the generated zip file to modify the generated plugin or compile it to a format other than VST 2.4. This option requires a MATLAB Coder™ license. To use the generated files with JUCE, you must obtain your own appropriately licensed copy of JUCE.

Option	UI Setting	Description
`-output` *`fileName`*	**Output file name**	Specifies the file name of the generated plugin or zip file. The appropriate extension is appended to the *`fileName`* based on the platform on which the plugin or zip file is generated. By default, the plugin or zip file is named after the class.
`-outdir` *`folder`*	**Output folder**	Generates a plugin or zip file to a specific folder. By default, the generated plugin is placed in the current folder. If *`folder`* is not in the current folder, specify the exact path.
`-win32`	**Generate a 32-bit audio plugin**	Creates a 32-bit audio plugin. Valid only on win64 Windows platforms. This option does not support the `coder.DeepLearningConfig("mkldnn")` deep learning library configuration, the `"Intel AVX (Windows)"` code replacement library, or the `"DSP Intel AVX2-FMA (Windows)"` code replacement library.
`-mac64universal`	**Generate a macOS universal plugin**	Creates a Mac audio plugin for use on Intel® and Apple Silicon. This option works only on Macintosh platforms. This option does not support the `coder.DeepLearningConfig("mkldnn")` deep learning library configuration or any code replacement libraries.
`-audioconfig` *`cfg`*	**Coder Configuration** section	Generates a plugin that uses a deep learning network or a code replacement library. See `audioPluginConfig` for more details.

Only the `-juceproject` option is supported in MATLAB Online.

`className` — Name of plugin class to generate

plugin class

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

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

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

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

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

Limitations

Build problems can occur when using folder names with spaces. For more information, see “Build Process Support for File and Folder Names” (Simulink Coder) and Why is the build process failing with error code: "NMAKE: fatal error U1073: don't know how to make 'C:\Program'"?.

More About

Generated VST Plugin File Extension

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

Operating System	File Extension
Windows	`.dll`
macOS	`.vst`

Version History

Introduced in R2016a

R2023a: Generate AUv3 audio plugins

Generate AUv3 plugin binaries for macOS.

R2022b: Generate and validate plugins through UI

Use a UI to configure plugin generation by calling `generateAudioPlugin` with no input arguments. The UI provides functionality equivalent to the command-line interfaces of `generateAudioPlugin`, `audioPluginConfig`, and `validateAudioPlugin`.

See Also

Audio Test Bench | `validateAudioPlugin` | `parameterTuner` | `loadAudioPlugin` | `audioPlugin` | `audioPluginSource` | `audioPluginConfig`

Topics

“Audio Plugins in MATLAB”
“Export a MATLAB Plugin to a DAW”

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)
```

```
standardLoudness = -11.6825
```

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

```
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: -22.98
Loudness range: 1.50

Beginning loudness: -23.38
Beginning loudness range: 1.18

Middle loudness: -22.97
Middle loudness range: 1.14

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

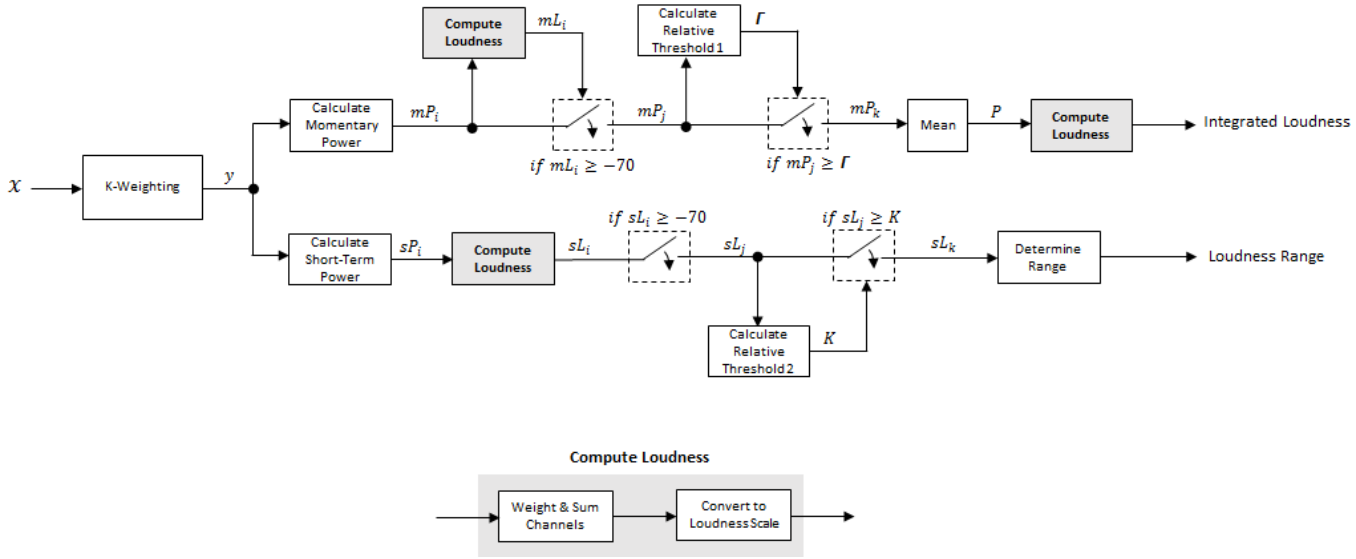
Loudness range in loudness units (LU), returned as a scalar.

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`

Algorithms

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



Integrated Loudness and Loudness Range

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

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:

$$mP_i \rightarrow mP_j$$

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

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:

$$sL_i \rightarrow sL_j$$

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

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).

Version History

Introduced in R2016b

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

`loudnessMeter` | `weightingFilter` | Loudness Meter

getMIDIConnections

Get MIDI connections of audio object

Syntax

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

Description

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

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

Examples

Get MIDI Connections of Plugin

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

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

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

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

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

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

```
connectionInfo =

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

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

```
connectionInfo.Delay1
```

```
ans =
```

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

Input Arguments

`audioObject` — Audio object

object

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

Output Arguments

`connectionInfo` — Information about MIDI connection

structure

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

Version History

Introduced in R2016a

See Also

Classes

`audioPlugin` | `audioPluginSource`

Functions

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

Topics

“MIDI Control for Audio Plugins”
“MIDI Control Surface Interface”

loadAudioPlugin

Load VST, VST 3, and AU plugins into MATLAB environment

Syntax

```
hostedPlugin = loadAudioPlugin(plugin)
```

Description

`hostedPlugin = loadAudioPlugin(plugin)` loads the 64-bit VST, VST 3, or AU audio plugin specified by `plugin`. On Windows, you can load VST and VST 3 plugins. On macOS, you can load AU, AUv3, VST, and VST 3 plugins.

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

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

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

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

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

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

Process Audio

- `audioOut = process(hostedPlugin,audioIn)`

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

Set and Get Normalized Parameter Values

- `value = getParameter(hostedPlugin,parameter)`

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

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

Get High-Level Information About the Hosted Plugin

- `dispParameter(hostedPlugin)`

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

  Returns a structure containing information about the hosted plugin.

Set the Environment in Which the Plugin Is Run

- `frameSize = getSamplesPerFrame(hostedPlugin)`

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

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

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

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

Examples

Host External Plugins in MATLAB

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

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

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

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

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

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

```
class(hostedPlugin)
```

```
ans =
'externalAudioPlugin'
```

```
class(hostedSourcePlugin)
```

```
ans =
'externalAudioPluginSource'
```

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

```
hostedSourcePlugin
```

```
hostedSourcePlugin =
  VST plugin 'oscillator'  source, 1 out, 256 samples

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

```
hostedPlugin
```

```
hostedPlugin =
  VST plugin 'ParametricEQ'  2 in, 2 out

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

Run External Plugin in MATLAB

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

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

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

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

Set the `MediumPeakGain` property to -20 dB.

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

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

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

release(fileReader)
release(deviceWriter)
```

Run External Source Plugin in MATLAB

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

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

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

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

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

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

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

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

release(deviceWriter)
```

Host AUv3 Plugin on macOS

To load an AUv3 plugin into MATLAB, you need the plugin component IDs.

Use the `auval` macOS command which lists all of the AUv3 plugins that are registered with the operating system, and use `grep` to search for the `Echo` plugin. Use the `system` function to run this command.

```
[~,out] = system("auval -a | grep Echo")
```

```
out =

    'aufx 4pvz Math  -  MathWorks: Echo
     '
```

Use the component IDs shown in the `auval` output to load the plugin into MATLAB.

```
hostedPlugin = loadAudioPlugin("AudioUnit: aufx 4pvz Math")
```

```
hostedPlugin =

  AudioUnit plugin 'Echo'  2 in, 2 out

         Gain: 0.5
     Feedback: 0.35
    BaseDelay: 0.5
   Wet_dryMix: 0.5
```

Input Arguments

`plugin` — External plugin to load

string scalar | character vector

External plugin to load, specified as a string or character vector containing the file name of the plugin. If the external plugin is not in the current folder, specify the full or relative path to the plugin file.

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

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

For AUv3 plugins on macOS, specify the plugin as a string or character vector containing the AUv3 component IDs in the format `"AudioUnit:TYPE SUBT MANU"`. `TYPE`, `SUBT`, and `MANU` are the four-character component IDs of the AUv3 plugin returned by the `auval` command in the macOS command line. For an example that shows how to load an AUv3 plugin using its component IDs, see “Host AUv3 Plugin on macOS” on page 2-584.

Example: `loadAudioPlugin("AudioUnit:aufx 4pvx Math")`

Output Arguments

`hostedPlugin` — Object of external plugin

`externalAudioPlugin` | `externalAudioSourcePlugin`

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

Limitations

- The `loadAudioPlugin` function supports 64-bit plugins only. You cannot load 32-bit plugins using the `loadAudioPlugin` function.
- Saving an external plugin as a MAT-file and then loading it preserves the external settings and parameters of the plugin but does not preserve its internal state or memory. Do not save and load your plugins when you are processing audio.

Version History

Introduced in R2016b

R2023a: Load AUv3 audio plugins

Host AUv3 plugins on macOS.

See Also

`parameterTuner` | **Audio Test Bench** | `audioPlugin` | `audioPluginSource` | `externalAudioPlugin` | `externalAudioPluginSource`

Topics

“Host External Audio Plugins”

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;
```

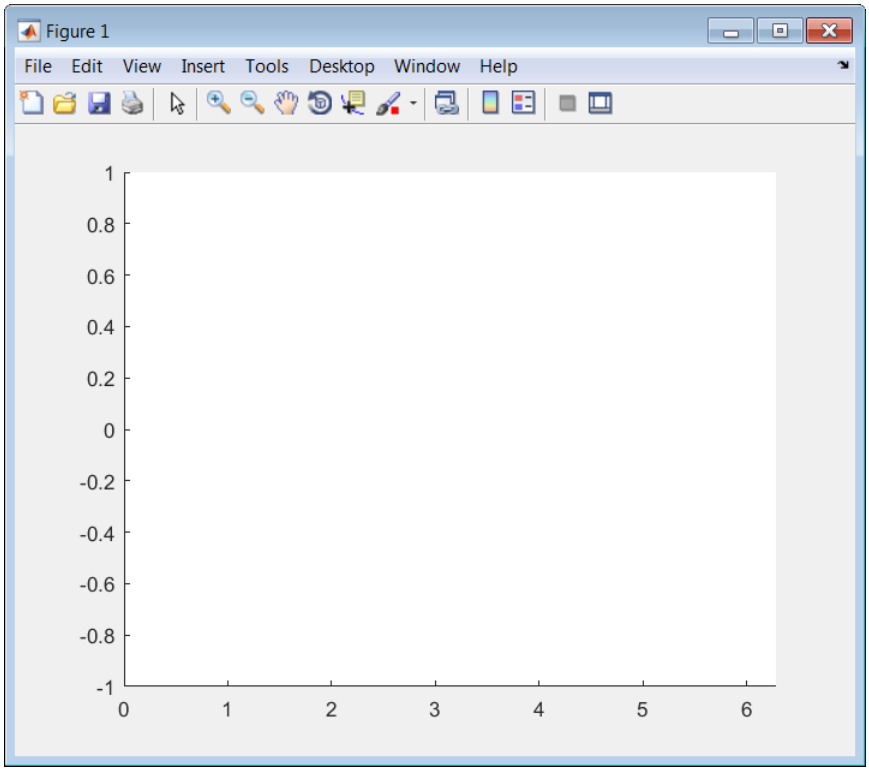
```
Move the control you wish to identify; type ^C to abort.
Waiting for control message...
```

Create an object that responds to the control you specified.

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

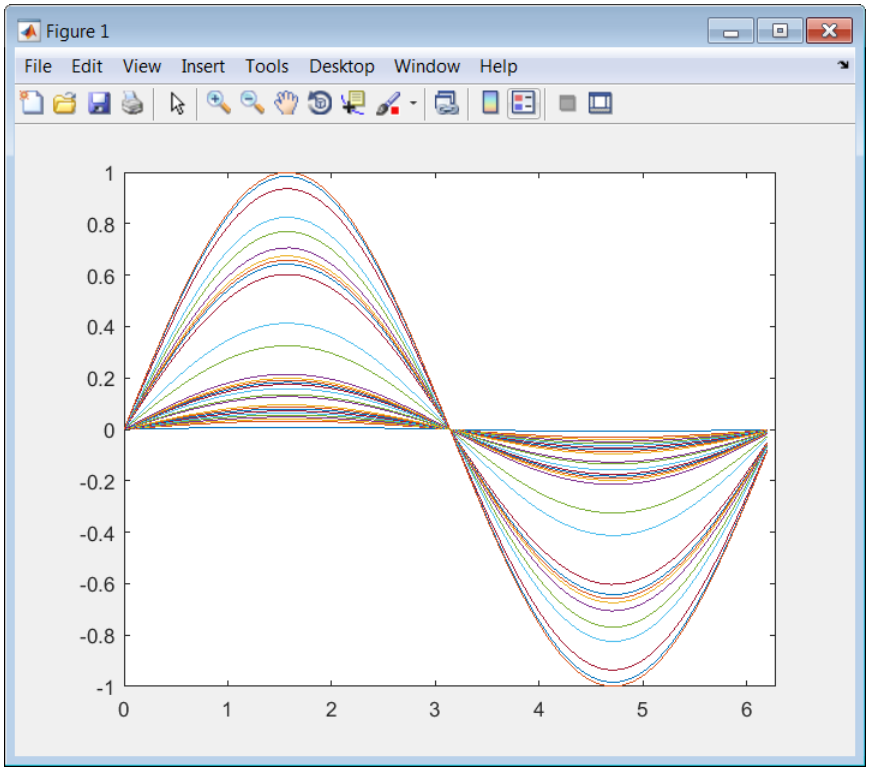
Define a function that plots a sinusoid with the amplitude set by your MIDI control. Make the axis constant.

```
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.

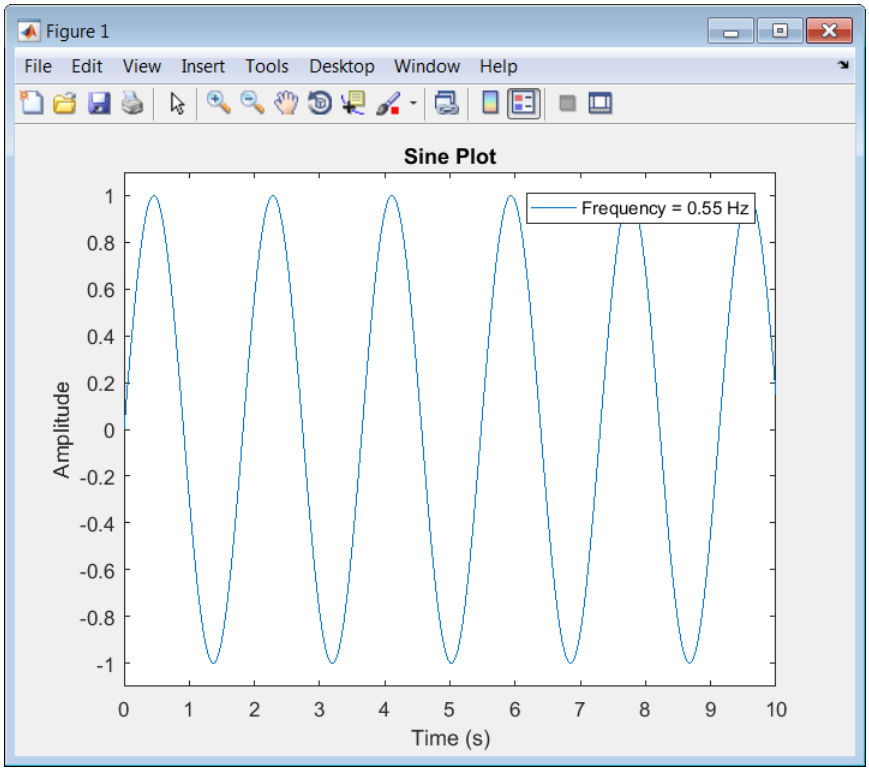
```
function plotSine(midicontrolsObject)

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”.

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.

Version History

Introduced in R2016a

See Also

`parameterTuner` | **Audio Test Bench** | `getMIDIConnections` | `configureMIDI` | `disconnectMIDI` | `midicontrols` | `midiread` | `midisync` | `midiid` | `setpref`

Topics
“MIDI Control Surface Interface”
“MIDI Control for Audio Plugins”

midicontrols

Open group of MIDI controls for reading

Syntax

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

Description

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

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

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

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

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

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

Examples

Listen to Any Control on Default Device

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

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

```
midicontrolsObject =

midicontrols object: any control on 'BCF2000'

ans =

     0
```

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

```
midiread(midicontrolsObject)
```

```
ans =

    0.3810
```

Listen to Specific Control

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

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

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

Create an object that responds to the control you specified.

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

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

```
midicontrolsObject = midiread(midicontrolsObject);
```

```
ans =

    0.4048
```

Specify Control Numbers and Initial Value

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

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

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

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

controlNumbers =

        1081        1085
        1082        1087
```

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

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

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

```
midiread(midicontrolsObject)
```

```
ans =

    0.0873    0.5000
    0.5000    0.5000
```

Specify Controls Numbers, Initial Value, and Output Mode

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

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

controlNumbers = [controlNumber1,controlNumber2];
```

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

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

```
initialValue = 12;
midicontrolsObject = midicontrols(controlNumbers,initialValue,'OutputMode','rawmidi');
```

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

```
midiread(midicontrolsObject)
```

```
ans =

    63    12
```

Set the Default MIDI Device

Assume that your MIDI device is a Behringer BCF2000. Enter this syntax at the MATLAB command line:

```
setpref midi DefaultDevice BCF2000
```

This preference persists across MATLAB sessions. You do not need to set it again unless you want to change your default device.

Specify Control Numbers and MIDI Device Name

Assume that your MIDI device is a Behringer BCF2000 and has a control with identification number 1001. Create a `midicontrols` object, which listens to control number 1001 on your Behringer BCF2000 device.

```
midicontrolsObject = midicontrols(1001,'MIDIDevice','BCF2000');
```

Input Arguments

`controlNumbers` — MIDI device control numbers

integer | array of integers

MIDI device control numbers, specified as an integer or array of integers. Use `midiid` to interactively identify the control numbers of your device. See “MIDI Device Control Numbers” on page 2-597 for an advanced explanation of how `controlNumbers` are determined.

If you specify `controlNumbers` as an empty vector, `[ ]`, then the `midicontrols` object responds to any control on your MIDI device.

Example: `1081`

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

`initialValues` — Initial values of MIDI controls

`0` (default) | scalar | array the same size as `controlNumbers`

Initial values of MIDI controls, specified as a scalar or an array the same size as `controlNumbers`. If you specify `initialValues` as a scalar, all controls specified by `controlNumbers` are assigned that value.

The value associated with your MIDI controls cannot be determined until you move a MIDI control. If you specify an initial value associated with your MIDI control, the initial value is returned by the `midiread` function until the MIDI control is moved.

- If `OutputMode` is specified as `'normalized'`, then initial values must be in the range [0,1]. Actual initial values are quantized and can be slightly different from initial values specified when your `midicontrols` object is created.
- If `OutputMode` is specified as `'rawmidi'`, then initial values must be integers in the range [0,127]

Example: `0.3`

Example: `[0,0.3,0.6]`

Example: `5`

Example: `[5;15;20]`

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

`deviceName` — MIDI device name

character vector | string

MIDI device name, assigned by the device manufacturer or host operating system, specified as a string. The specified `deviceName` can be a substring of the exact name of your device. If you do not

specify `deviceName`, the default MIDI device is used. See “Set the Default MIDI Device” on page 2-595 for an example of specifying a default MIDI device.

If you do not set a default MIDI device, the host operating system chooses the default device in an unspecified way. As a best practice, use `midiid` to identify the name of the device you want.

Example: `'MIDIDevice','BCF2000 MIDI 1'`

Data Types: `char` | `string`

`mode` — Output mode for MIDI control value
`'normalized'` (default) | `'rawmidi'`

Output mode for MIDI control value, specified as `'normalized'` or `'rawmidi'`.

- `'normalized'` — Values of your MIDI control are normalized. If your `midicontrols` object is called by `midiread`, then values in the range [0,1] are returned.
- `'rawmidi'` — Values of your MIDI control are not normalized. If your `midicontrols` object is called by `midiread`, then integer values in the range [0,127] are returned.

Example: `'OutputMode','normalized'`

Example: `'OutputMode','rawmidi'`

Data Types: `char` | `string`

Output Arguments

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

Object that listens to the controls on a MIDI device.

More About

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*.

Version History

Introduced in R2016a

See Also

`parameterTuner` | **Audio Test Bench** | `getMIDIConnections` | `configureMIDI` | `disconnectMIDI` | `midicallback` | `midiread` | `midisync` | `midiid` | `setpref`

Topics
“MIDI Control Surface Interface”
“MIDI Control for Audio Plugins”

midiid

Interactively identify MIDI control

Syntax

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

Description

`[controlNumber,deviceName] = midiid` returns the control number and device name of the MIDI control you move. Call the function and then move the control you want to identify. The function detects which control you move and returns the control number and device name that specify that control.

Examples

Identify Control Number and Device Name

Call `midiid` and then move the control you want to identify on the MIDI device you want to identify.

```
[ctl,dev] = midiid;
```

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

ctl =
1002
dev =
nanoKONTROL
```

Output Arguments

`controlNumber` — MIDI device control number
integer

MIDI device control number, specified as an integer. The device manufacturer assigns the value to the control for identification purposes.

`deviceName` — MIDI device name
string

MIDI device name assigned by the device manufacturer or host operating system, specified as a string.

Version History

Introduced in R2016a

See Also

`getMIDIConnections` | `configureMIDI` | `disconnectMIDI` | `midiread` | `midisync` | `midicallback` | `setpref` | `parameterTuner` | **Audio Test Bench**

Topics

“MIDI Control Surface Interface”
“MIDI Control for Audio Plugins”

midiread

Return most recent value of MIDI controls

Syntax

```
controlValues = midiread(midicontrolsObject)
```

Description

`controlValues = midiread(midicontrolsObject)` returns the most recent value of the MIDI controls associated with the specified `midicontrolsObject`. To create this object, use the `midicontrols` function.

Examples

Read Control Values of MIDI Device

```
midicontrolsObject = midicontrols;
controlValue = midiread(midicontrolsObject);
```

Read Multiple Control Values of MIDI Device

Identify two MIDI controls on your MIDI device.

```
[controlOne,~] = midiid
[controlTwo,~] = midiid

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

controlOne =

        1081

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

controlTwo =

        1082
```

Create a MIDI controls object that listens to both controls you identified.

```
controlNumbers = [controlOne,controlTwo];
midicontrolsObject = midicontrols(controlNumbers);
```

Move your specified MIDI controls and return their values. The values are returned as a vector that corresponds to your control numbers vector, `controlNumbers`.

```
tic
while toc < 5
    controlValues = midiread(midicontrolsObject)
end

controlValues =

    0.0397    0.0556
```

Read Control Values in an Audio Stream Loop

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

```
[controlNumber, deviceName] = midiid;

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

Create a MIDI controls object. The value associated with your MIDI controls object cannot be determined until you move the MIDI control. Specify an initial value associated with your MIDI control. The `midiread` function returns the initial value until the MIDI control is moved.

```
initialControlValue = 1;
midicontrolsObject = midicontrols(controlNumber,initialControlValue);
```

Create a `dsp.AudioFileReader` System object with default settings. Create an `audioDeviceWriter` System object and specify the sample rate.

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

In an audio stream loop, read an audio signal frame from the file, apply gain specified by the control on your MIDI device, and then write the frame to your audio output device. By default, the control value returned by `midiread` is normalized.

```
while ~isDone(fileReader)
    audioData = step(fileReader);

    controlValue = midiread(midicontrolsObject);

    gain = controlValue*2;
    audioDataWithGain = audioData*gain;

    play(deviceWriter,audioDataWithGain);
end
```

Close the input file and release your output device.

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

Input Arguments

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

object

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

Output Arguments

`controlValues` — Most recent values of MIDI controls

`[0,1]` (default) | integer values in the range `[0,127]`

Most recent values of MIDI controls, returned as normalized values in the range `[0,1]`, or as integer values in the range `[0,127]`. The output values depend on the `OutputMode` specified when your `midicontrols` object is created.

- If `OutputMode` was specified as `'normalized'`, then `midiread` returns values in the range `[0,1]`. The default `OutputMode` is `'normalized'`.
- If `OutputMode` was specified as `'rawmidi'`, then `midiread` returns integer values in the range `[0,127]`, and no quantization is required.

Version History

Introduced in R2016a

See Also

Audio Test Bench | `parameterTuner` | `getMIDIConnections` | `configureMIDI` | `disconnectMIDI` | `midicontrols` | `midicallback` | `midisync` | `midiid` | `setpref`

Topics

“MIDI Control Surface Interface”
“MIDI Control for Audio Plugins”

midisync

Send values to MIDI controls for synchronization

Syntax

```
midisync(midicontrolsObject)
midisync(midicontrolsObject,controlValues)
```

Description

`midisync(midicontrolsObject)` sends the initial values of controls to your MIDI device, as specified by your MIDI controls object. To create this object, use the `midicontrols` function. If your MIDI device can receive and respond to messages, it adjusts its controls as specified.

Note Many MIDI devices are not bidirectional. Calling `midisync` with a unidirectional device has no effect. `midisync` cannot tell whether a value is successfully sent to a device or even whether the device is bidirectional. If sending a value fails, no errors or warnings are generated.

`midisync(midicontrolsObject,controlValues)` sends `controlValues` to the MIDI controls associated with the specified `midicontrolsObject`.

Examples

Synchronize MIDI Control to Initial Value

Use `midiid` to identify a control on your default MIDI device.

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

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

Create a MIDI controls object. Specify an initial value for your control. Call `midisync` to set the specified control on your device to the initial value.

```
initialValue = 0.5;
midicontrolsObject = midicontrols(controlNumber,initialValue);
midisync(midicontrolsObject);
```

Synchronize MIDI Control to Specified Value

Use `midiid` to identify three controls on your default MIDI device.

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

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

Create a MIDI controls object. Specify initial values for your controls. Call `midisync` to set the specified control on your device to the initial value.

```
controlValues = [0,0,1];
midicontrolsObject = midicontrols(controlNumbers,controlValues);
midisync(midicontrolsObject);
```

Create a loop that updates your control values and synchronizes those values to the physical controls on your device.

```
for i = 1:100
    controlValues = controlValues + [0.006,0.008,-0.008];
    midisync(midicontrolsObject,controlValues);
    pause(0.1)
end
```

Create UI Slider and Synchronize with MIDI Control

Define this function and save it to your current folder.

```
function trivialmidigui(controlNumber,deviceName)

    slider = uicontrol('Style','slider');
    mc = midicontrols(controlNumber,'MIDIDevice',deviceName);
    midisync(mc);
    set(slider,'Callback',@slidercb);
    midicallback(mc, @mccb);

    function slidercb(slider,~)
        val = get(slider,'Value');
        midisync(mc, val);
        disp(val);
    end

    function mccb(mc)
        val = midiread(mc);
        set(slider,'Value',val);
        disp(val);
    end

end
```

Use `midiid` to identify a control number and device name. Call the function you created, specifying the control number and device name as inputs.

```
[controlNumber,deviceName] = midiid;
trivialmidigui(controlNumber,deviceName)
```

The slider on the user interface is synchronized with the specified control on your device. Move one to see the other respond.

Input Arguments

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

object

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

`controlValues` — Values sent to MIDI device

initial values specified by `midicontrolsObject` (default) | scalar | array

Values sent to MIDI device, specified as a scalar or an array the same size as `controlNumbers` of the associated `midicontrols` object. If you do not specify `controlValues`, the default value is the `initialValues` of the associated `midicontrols` object.

The possible range for `controlValues` depends on the `OutputMode` of the associated `midicontrols` object.

- If `OutputMode` is specified as `'normalized'`, then `controlValues` must consist of values in the range `[0,1]`. The default `OutputMode` is `'normalized'`.
- If `OutputMode` is specified as `'rawmidi'`, then `controlValues` must consist of integer values in the range `[0,127]`.

Example: `0.3`

Example: `[0,0.3,0.6]`

Example: `5`

Example: `[5;15;20]`

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

Version History

Introduced in R2016a

See Also

Audio Test Bench | `parameterTuner` | `getMIDIConnections` | `configureMIDI` | `disconnectMIDI` | `midicontrols` | `midiread` | `midicallback` | `midiid` | `setpref`

Topics

“MIDI Control Surface Interface”
“MIDI Control for Audio Plugins”

validateAudioPlugin

Test MATLAB source code for audio plugin

Syntax

```
validateAudioPlugin classname
validateAudioPlugin options classname
```

Description

`validateAudioPlugin classname` generates and runs a “Test Bench Procedure” on page 2-609 that exercises your audio plugin class.

`validateAudioPlugin options classname` specifies options to modify the default “Test Bench Procedure” on page 2-609.

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` | `-audioconfig cfg`

Options to modify test bench procedure, specified as `-nomex`, `-keeptestbench`, or `-audioconfig cfg`. Options can be specified together or separately, and in any order.

You can also specify these options and validate plugins using the `generateAudioPlugin` user interface (UI).

Option	`generateAudioPlugin` UI Setting	Description
`-nomex`	Clear the **Run a MEX version of the test bench** option	Do not generate and run a MEX version of the test bench file. This option significantly reduces run time of the test bench procedure.
`-keeptestbench`	**Save test benches to output folder**	Save the generated test benches to the current folder. In the `generateAudioPlugin` UI, the test benches are saved to the folder specified by **Output folder**.
`-audioconfig` *`cfg`*	**Coder Configuration** section	Specify deep learning and code replacement configuration for coder. See `audioPluginConfig` for more details

`classname` — Name of the plugin class to validate

plugin class

Name of the plugin class to validate. The plugin class must derive from either the `audioPlugin` class or the `audioPluginSource` class. The `validateAudioPlugin` function exercises an instance of the specified plugin class.

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

- `validateAudioPlugin myPlugin`
- `validateAudioPlugin myPlugin.m`

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

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

Limitations

The `validateAudioPlugin` function is compatible with Windows and Mac operating systems. It is not compatible with Linux.

More About

Test Bench Procedure

The `validateAudioPlugin` 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.

Version History

Introduced in R2016a

See Also

Audio Test Bench | `generateAudioPlugin` | `parameterTuner` | `audioPlugin` | `audioPluginSource` | `audioPluginConfig`

Topics

“Audio Plugins in MATLAB”

acousticLoudness

Perceived loudness of acoustic signal

Syntax

```
loudness = acousticLoudness(audioIn,fs)
loudness = acousticLoudness(audioIn,fs,calibrationFactor)
loudness = acousticLoudness(SPLIn)
loudness = acousticLoudness( ___ ,Name,Value)

[loudness,specificLoudness] = acousticLoudness( ___ )

[loudness,specificLoudness,perc] = acousticLoudness( ___ ,'TimeVarying',true)
[loudness,specificLoudness,perc] = acousticLoudness( ___ ,'TimeVarying',true,
'Percentiles',p)

acousticLoudness( ___ )
```

Description

`loudness = acousticLoudness(audioIn,fs)` returns loudness in sones according to ISO 532-1 (Zwicker).

`loudness = acousticLoudness(audioIn,fs,calibrationFactor)` specifies a nondefault microphone calibration factor used to compute loudness.

`loudness = acousticLoudness(SPLIn)` computes loudness using one-third-octave-band sound pressure levels (SPL).

`loudness = acousticLoudness( ___ ,Name,Value)` specifies options using one or more `Name,Value` pair arguments.

Example: `loudness = acousticLoudness(audioIn,fs,'Method','ISO 532-2')` returns loudness according to ISO 532-2 (Moore-Glasberg).

`[loudness,specificLoudness] = acousticLoudness( ___ )` also returns the specific loudness.

`[loudness,specificLoudness,perc] = acousticLoudness( ___ ,'TimeVarying',true)` also returns percentile loudness.

`[loudness,specificLoudness,perc] = acousticLoudness( ___ ,'TimeVarying',true, 'Percentiles',p)` specifies nondefault percentiles to return.

`acousticLoudness( ___ )` with no output arguments plots specific loudness and displays loudness textually. If `TimeVarying` is `true`, both loudness and specific loudness are plotted, with the latter in 3-D.

Examples

Measure Acoustic Loudness

Measure the ISO 532-1 stationary free-field loudness. Assume the recording level is calibrated such that a 1 kHz tone registers as 100 dB on a SPL meter.

```
[audioIn,fs] = audioread('WashingMachine-16-44p1-stereo-10secs.wav');

loudness = acousticLoudness(audioIn,fs)
```

```
loudness = 1×2

   28.2688   27.7643
```

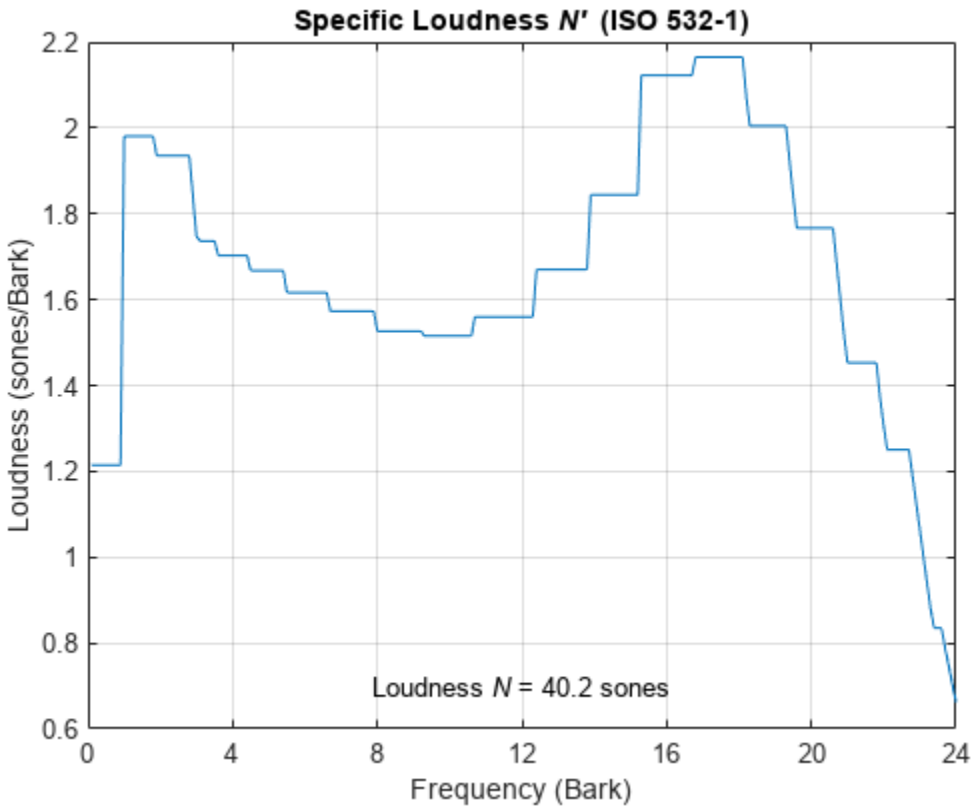
Measure Loudness and Sharpness of Stationary Signals

Create two stationary signals with equivalent power: a pink noise signal and a white noise signal.

```
fs = 48e3;
dur = 5;
pnoise = 2*pinknoise(dur*fs);
wnoise = rand(dur*fs,1) - 0.5;
wnoise = wnoise*sqrt(var(pnoise)/var(wnoise));
```

Call `acousticLoudness` using the default ISO 532-1 (Zwicker) method and no output arguments to plot the loudness of the pink noise. Call `acousticLoudness` again, this time with output arguments, to get the specific loudness.

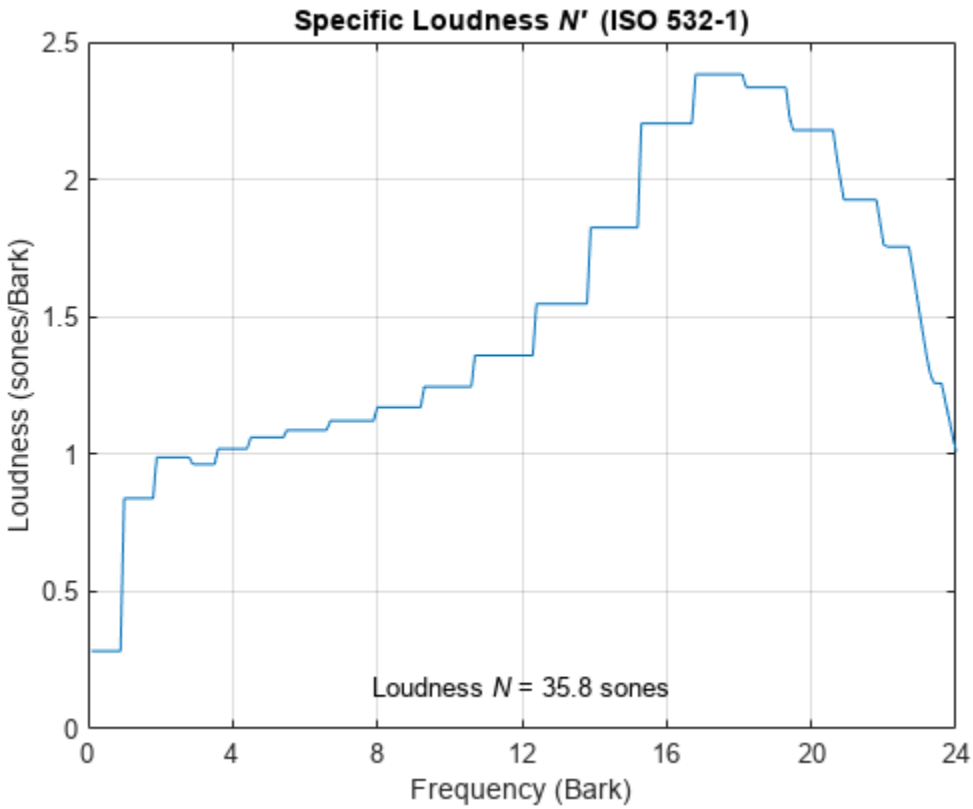
```
figure
acousticLoudness(pnoise,fs)
```




```
[~,pSpecificLoudness] = acousticLoudness(pnoise,fs);
```

Plot the loudness for the white noise signal and then get the specific loudness values.

```
figure
acousticLoudness(wnoise,fs)
```




```
[~,wSpecificLoudness] = acousticLoudness(wnoise,fs);
```

Call the `acousticSharpness` function to compare the sharpness of the pink noise and white noise.

```
pSharpness = acousticSharpness(pSpecificLoudness);
wSharpness = acousticSharpness(wSpecificLoudness);
fprintf('Sharpness of pink noise = %0.2f acum\n',pSharpness)
```

```
Sharpness of pink noise = 2.00 acum
```

```
fprintf('Sharpness of white noise = %0.2f acum\n',wSharpness)
```

```
Sharpness of white noise = 2.62 acum
```

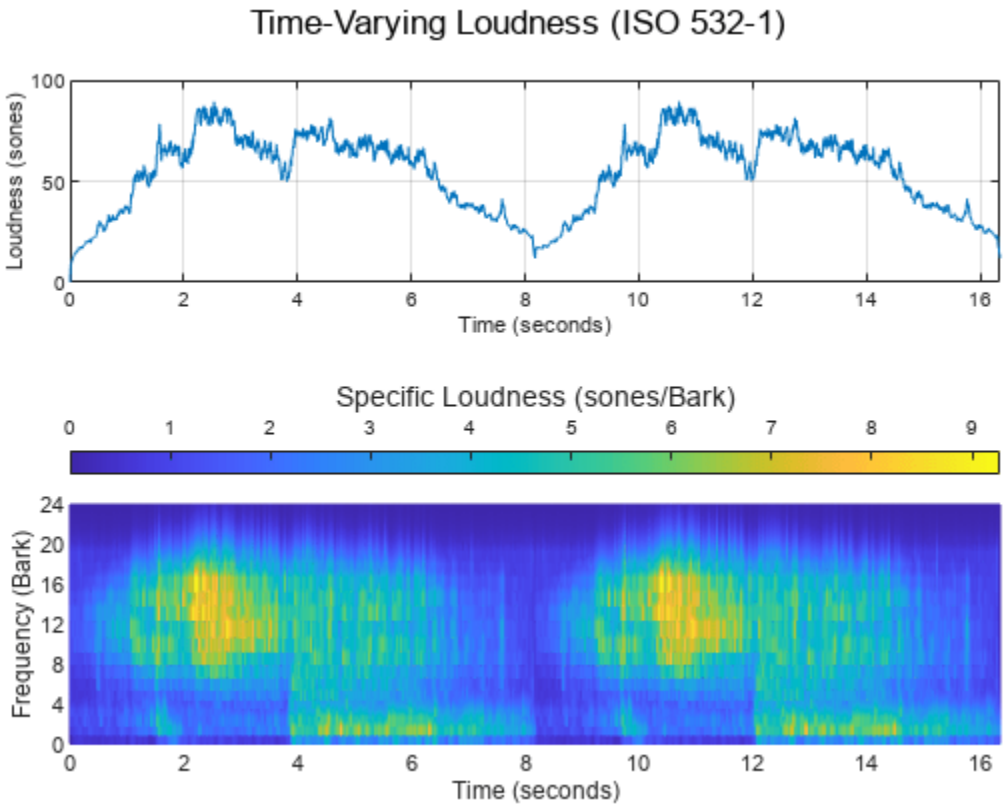
Time-Varying Loudness and Percentiles

Read in an audio file.

```
[audioIn,fs] = audioread('JetAirplane-16-11p025-mono-16secs.wav');
```

Plot the time-varying acoustic loudness in accordance with ISO 532-1 and get the percentiles. Listen to the audio signal.

```
acousticLoudness(audioIn,fs,'SoundField','diffuse','TimeVarying',true)
```




```
sound(audioIn,fs)
```

Call `acousticLoudness` again with the same inputs and get the percentiles. Print the *Nmax* and *N5* percentiles. The *Nmax* percentile is the maximum loudness reported. The *N5* percentile is the loudness below which is 95% of the reported loudness.

```
[~,~,perc] = acousticLoudness(audioIn,fs,'SoundField','diffuse','TimeVarying',true);
fprintf('Max loudness = %0.2f sones\n',perc(1))
```

```
Max loudness = 89.48 sones
```

```
fprintf('N5 loudness = %0.2f sones\n',perc(2))
```

```
N5 loudness = 81.77 sones
```

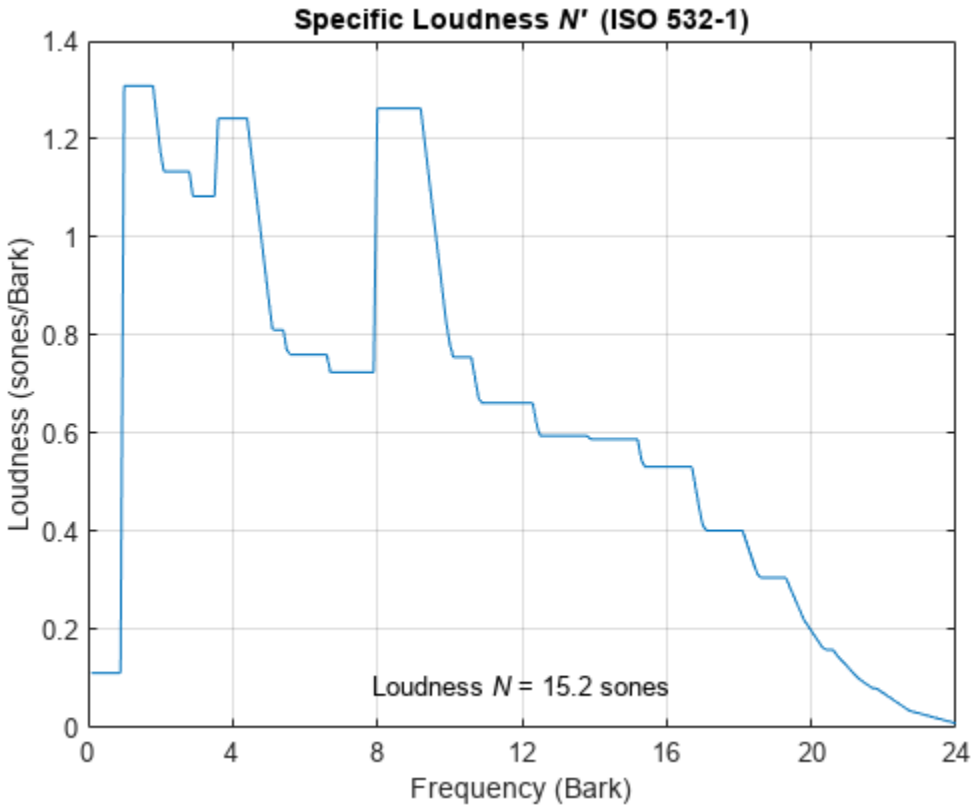
Measure Acoustic Loudness from Sound Pressure Level

Read in an audio file.

```
[audioIn,fs] = audioread('Turbine-16-44p1-mono-22secs.wav');
```

Call `acousticLoudness` with no output arguments to plot the specific loudness. Assume a calibration factor of 0.15 and a reference pressure of 21 micropascals. To determine the calibration factor specific to your audio system, use the `calibrateMicrophone` function.

```
calibrationFactor = 0.15;
refPressure = 21e-6;
acousticLoudness(audioIn,fs,calibrationFactor,'PressureReference',refPressure)
```



`acousticLoudness` enables you to specify an intermediate representation, sound pressure levels, instead of a time-domain input. This enables you to reuse intermediate SPL calculations. Another advantage is that if your physical SPL meter does not report loudness in accordance to ISO 532-1 or ISO 531-2, you can use the reported 1/3-octave SPLs to calculate standard-compliant loudness.

To calculate sound pressure levels from an audio signal, first create an `splMeter` object. Call the `splMeter` object with the audio input.

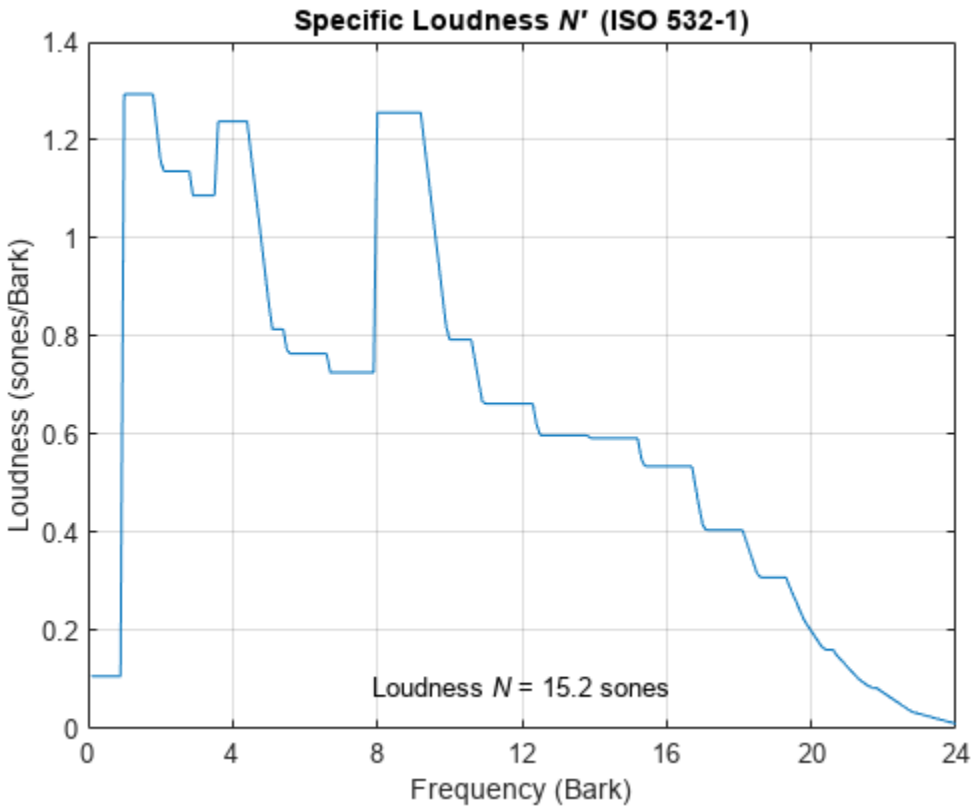
```
spl = splMeter("SampleRate",fs,"Bandwidth","1/3 octave", ...
    "CalibrationFactor",calibrationFactor,"PressureReference",refPressure, ...
    "FrequencyWeighting","Z-weighting","OctaveFilterOrder",6);

splMeasurement = spl(audioIn);
```

Compute the mean SPL level, skipping the first 0.2 seconds. Only keep the bands from 25 Hz to 12.5 kHz (the first 28 bands).

```
SPLIn = mean(splMeasurement(ceil(0.2*fs):end,1:28));
```

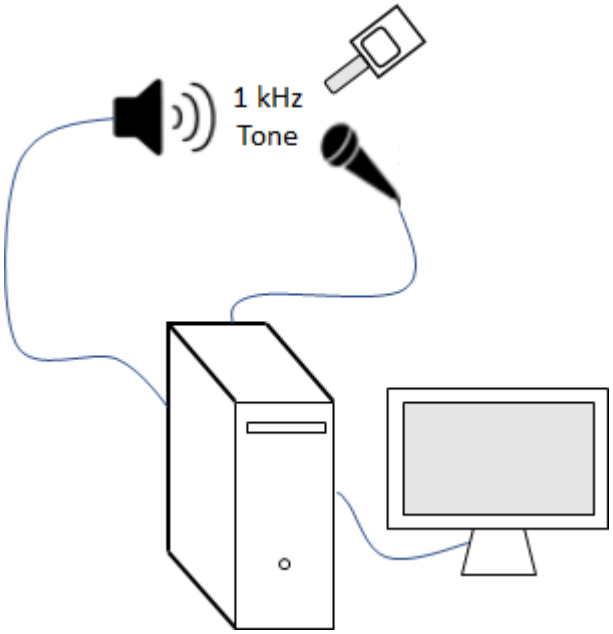
Using the SPL input, call `acousticLoudness` with no output arguments to plot the specific loudness.

```
acousticLoudness(SPLIn)
```



Loudness Measurements Using Calibrated Microphone

Set up an experiment as indicated by the diagram.



Create an `audioDeviceReader` object to read from the microphone and an `audioDeviceWriter` object to write to your speaker.

```
fs = 48e3;
deviceReader = audioDeviceReader(fs);
deviceWriter = audioDeviceWriter(fs);
```

Create an `audioOscillator` object to generate a 1 kHz sinusoid.

```
osc = audioOscillator("sine",1e3,"SampleRate",fs);
```

Create a `dsp.AsyncBuffer` object to buffer data acquired from the microphone.

```
dur = 5;
buff = dsp.AsyncBuffer(dur*fs);
```

For five seconds, play the sinusoid through your speaker and record using your microphone. While the audio streams, note the loudness as reported by your SPL meter. Once complete, read the contents of the buffer object.

```
numFrames = dur*(fs/osc.SamplesPerFrame);
for ii = 1:numFrames
    audioOut = osc();
    deviceWriter(audioOut);

    audioIn = deviceReader();
    write(buff,audioIn);
end

SPLreading = 60.4;

micRecording = read(buff);
```

To compute the calibration factor for the microphone, use the `calibrateMicrophone` function.

```
calibrationFactor = calibrateMicrophone(micRecording,deviceReader.SampleRate,SPLreading);
```

Call `acousticLoudness` with the microphone recording, sample rate, and calibration factor. The loudness reported from `acousticLoudness` is the true acoustic loudness measurement as specified by 532-1.

```
loudness = acousticLoudness(micRecording,deviceReader.SampleRate,calibrationFactor)
```

```
loudness = 14.7902
```

You can now use the calibration factor you determined to measure the loudness of any sound that is acquired through the same microphone recording chain.

Plot Specific Loudness Over Hertz

Read in an audio signal.

```
[audioIn,fs] = audioread('TrainWhistle-16-44p1-mono-9secs.wav');
```

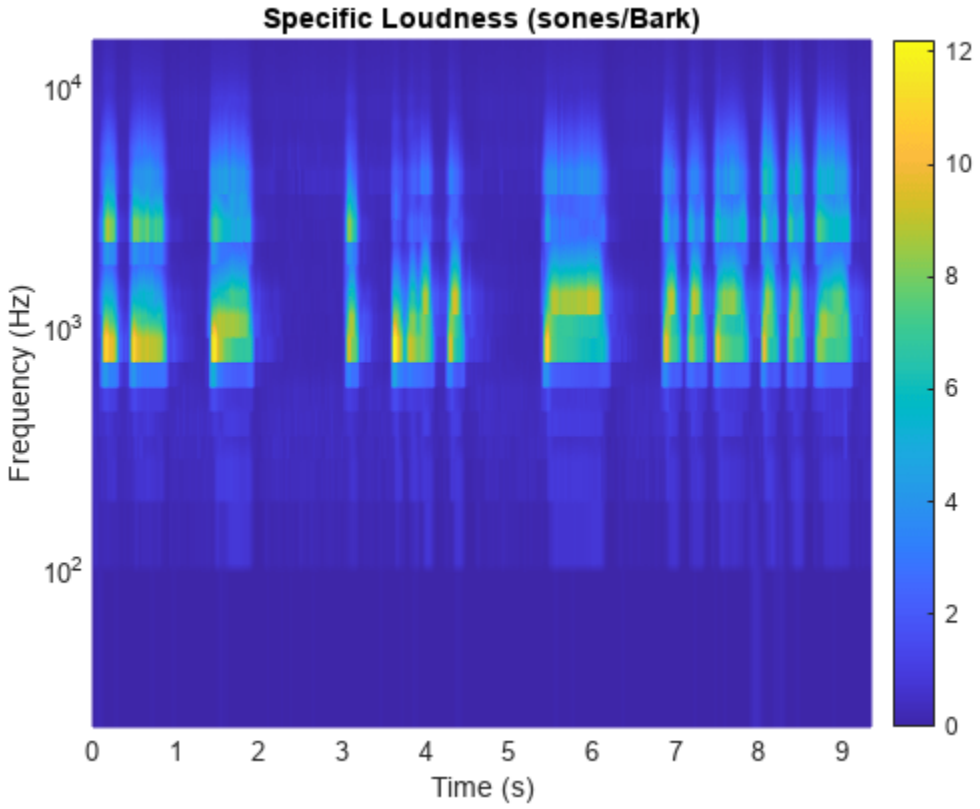
ISO 532-1

Determine the time-varying specific loudness according to the default method (ISO 532-1).

```
[~,specificLoudness] = acousticLoudness(audioIn,fs,'TimeVarying',true);
```

ISO 532-1 reports specific loudness over Bark, where the Bark bins are `0.1:0.1:24`. Convert the Bark bins to Hz and then plot the specific loudness over Hz across time.

```
barkBins = 0.1:0.1:24;
hzBins = bark2hz(barkBins);

t = 0:2e-3:2e-3*(size(specificLoudness,1)-1);
surf(t,hzBins,sum(specificLoudness,3).','EdgeColor','interp')
set(gca,'YScale','log')
view([0 90])
axis tight
xlabel('Time (s)')
ylabel('Frequency (Hz)')
colorbar
title('Specific Loudness (sones/Bark)')
```



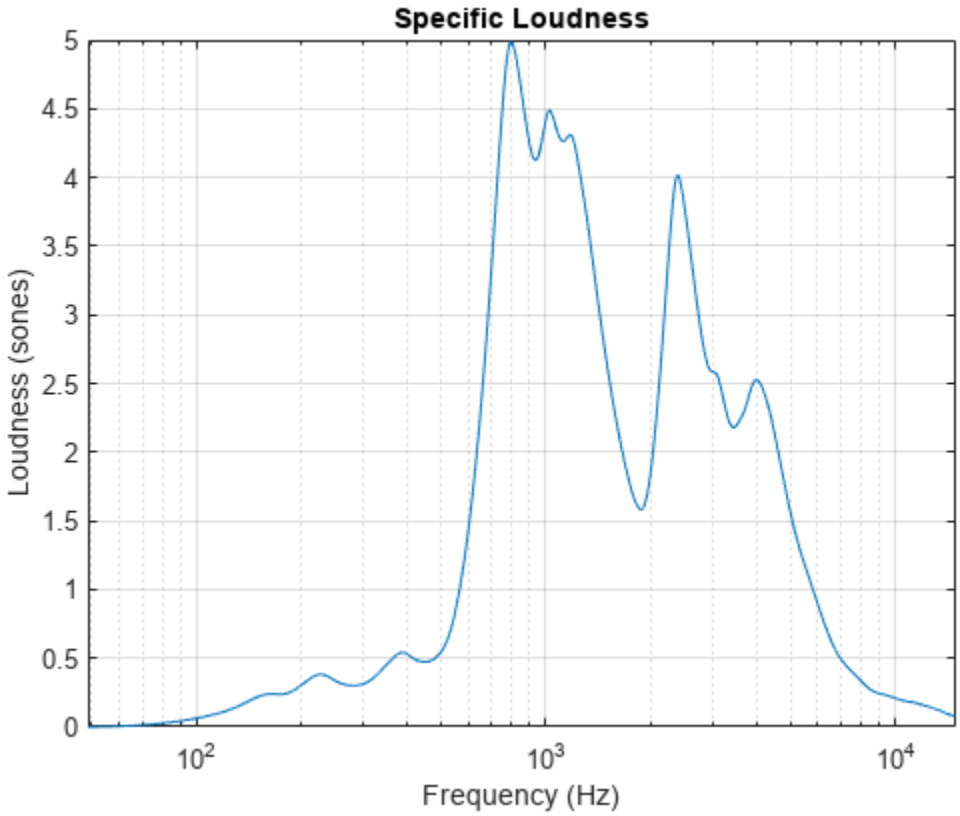
ISO 532-2

Determine the stationary specific loudness according to the Moore-Glasberg method (ISO 532-2).

```
[~,specificLoudness] = acousticLoudness(audioIn,fs,'Method','ISO 532-2');
```

ISO 532-2 reports specific loudness over the ERB scale, where the ERB bins are `1.8:0.1:38.9`. The unit of the ERB scale is sometimes referred to as Cam. Convert the ERB bins to Hz and then plot the specific loudness.

```
erbBins = 1.8:0.1:38.9;
hzBins = erb2hz(erbBins);

semilogx(hzBins,specificLoudness)
xlabel('Frequency (Hz)')
ylabel('Loudness (sones)')
title('Specific Loudness')
grid on
```



Loudness Using Custom Earphone Responses

Read in an audio file.

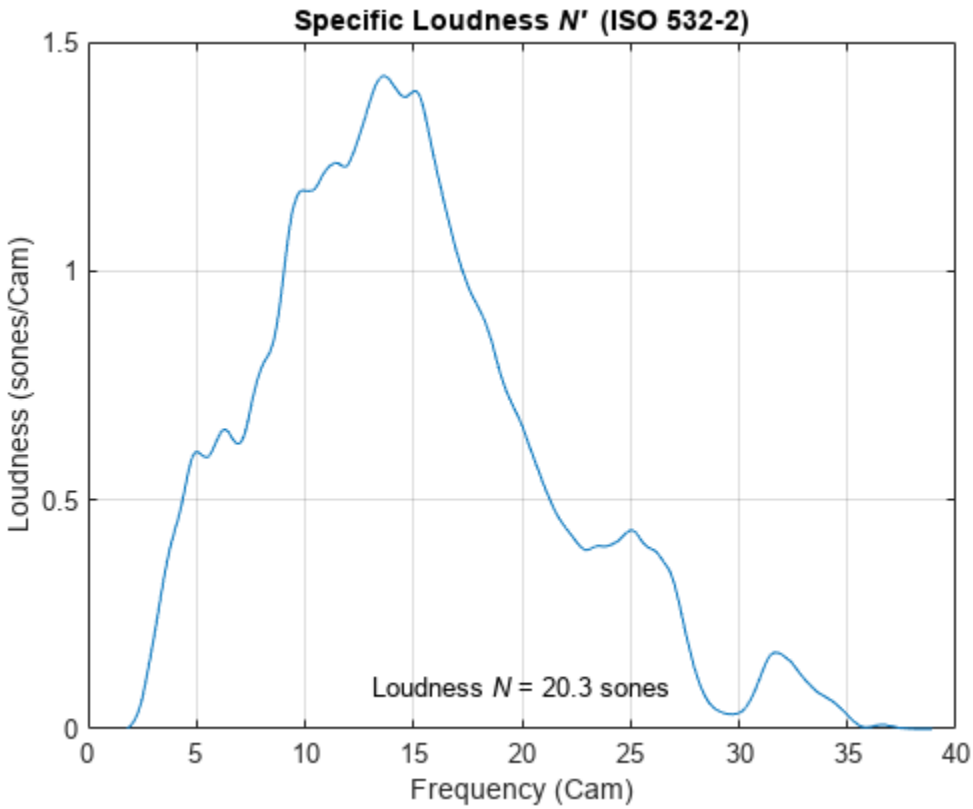
```
[x,fs] = audioread('WashingMachine-16-44p1-stereo-10secs.wav');
```

ISO 532-2 enables you to specify a custom earphone response when calculating loudness. Create a 30-by-2 matrix where the first column is the frequency and the second column is the earphone's deviation from a flat response.

```
tdh = [   0,    80,   100,  200,  500,  574,  660,  758,  871, 1000, 1149,  1320,  1516,  1741,
       2297,  2639,  3031, 3482, 4000, 4500, 5000, 5743, 6598, 7579, 8706, 10000, 12000, 16000, 2
        -50, -15.3, -13.8, -8.1, -0.5,  0.4,  0.8,  0.9,  0.5,  0.1, -0.8,  -1.5,  -2.3,  -3.2,
       -4.2,  -4.3,  -4.3, -3.9, -3.2, -2.3, -1.1, -0.3,   -2, -5.4,   -9, -12.1, -15.2,   -30,
```

Calculate the loudness using ISO 532-2. Specify `SoundField` as `earphones` and the earphone response as the matrix you just created.

```
acousticLoudness(x,fs,'Method','ISO 532-2','SoundField','earphones','EarphoneResponse',tdh)
```



Streaming Calculation of Stationary Loudness

Create a `dsp.AudioFileReader` object to read in an audio signal frame-by-frame. Specify a frame duration of 50 ms. This will be the frame duration over which you calculate stationary loudness.

```
fileReader = dsp.AudioFileReader('Engine-16-44p1-stereo-20sec.wav');

frameDur = 0.05;
fileReader.SamplesPerFrame = round(fileReader.SampleRate*frameDur);
```

Create an `audioDeviceWriter` object to write audio to your default output device.

```
deviceWriter = audioDeviceWriter('SampleRate',fileReader.SampleRate);
```

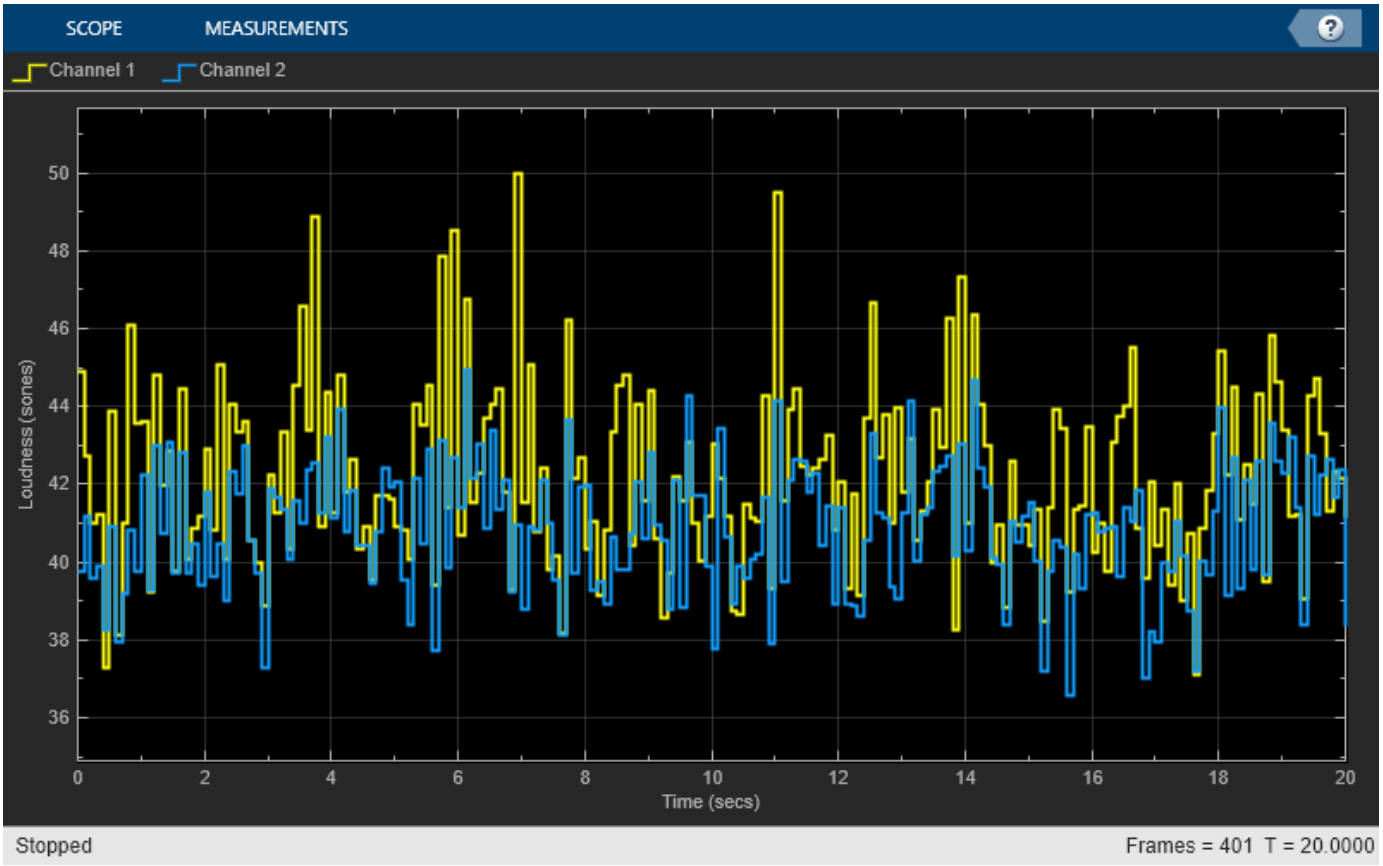
Create a `timescope` object to display stationary loudness over time.

```
scope = timescope( ...
    'SampleRate',1/frameDur, ...
    'YLabel','Loudness (sones)', ...
    'ShowGrid',true, ...
    'PlotType','Stairs', ...
    'TimeSpanSource','property', ...
    'TimeSpan',20, ...
    'AxesScaling','Auto', ...
    'ShowLegend',true);
```

In a loop:

1. Read a frame from the audio file.
2. Calculate the stationary loudness of that frame.
3. Play the sound through your output device.
4. Write the loudness to the scope.

```
while ~isDone(fileReader)
    audioIn = fileReader();
    loudness = acousticLoudness(audioIn,fileReader.SampleRate);
    deviceWriter(audioIn);
    scope(loudness)
end
release(fileReader)
release(deviceWriter)
release(scope)
```



Input Arguments

`audioIn` — Audio input

column vector | 2-column matrix

Audio input, specified as a column vector (mono) or matrix with two columns (stereo).

Data Types: `single` | `double`

`fs` — Sample rate (Hz)

positive scalar

Sample rate in Hz, specified as a positive scalar. The recommended sample rate for new recordings is 48 kHz.

Note The minimum acceptable sample rate is 8 kHz.

Data Types: `single` | `double`

`calibrationFactor` — Microphone calibration factor

`sqrt(8)` | positive scalar

Microphone calibration factor, specified as a positive scalar. The default calibration factor corresponds to a full-scale 1 kHz sine wave with a sound pressure level of 100 dB (SPL). To compute the calibration factor specific to your system, use the `calibrateMicrophone` function.

Data Types: `single` | `double`

`SPLIn` — Sound pressure level (dB)

1-by-28-by-*C* | 1-by-29-by-*C*

Sound pressure level (SPL) in dB, specified as a 1-by-28-by-*C* array or a 1-by-29-by-*C* array, depending on the `Method`:

- If `Method` is set to `'ISO 532-1'`, specify `SPLIn` as a 1-by-28-by-*C* array, where 28 corresponds to one-third-octave bands between 25 Hz and 12.5 kHz, and *C* is the number of channels.
- If `Method` is set to `'ISO 532-2'`, specify `SPLIn` as a 1-by-29-by-*C* array, where 29 corresponds to one-third-octave bands between 25 Hz and 16 kHz, and *C* is the number of channels.

For both methods, the SPL input should be measured with a flat frequency weighting (Z-weighting).

Data Types: `single` | `double`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `acousticLoudness(audioIn,fs,'Method','ISO 532-2')`

`Method` — Loudness calculation method

`'ISO 532-1'` (default) | `'ISO 532-2'`

Loudness calculation method, specified as `'ISO 532-1'` [1] or `'ISO 532-2'` [2].

Note Only in the ISO 532-1 method, output is reported for each channel independently, and stationary signals are processed after discarding up to the first 0.2 seconds of the signal at the output of the internal 1/3-octave filters.

Data Types: `char` | `string`

`TimeVarying` — Input is time-varying
`false` (default) | `true`

Input is time-varying, specified as `true` or `false`. When `true`, the `TimeResolution` argument determines the time interval.

Dependencies

To set `TimeVarying` to `true`, you must set `Method` to `'ISO 532-1'`.

Data Types: `logical`

`SoundField` — Sound field of audio recording
`'free'` (default) | `'diffuse'` | `'eardrum'` | `'earphones'`

Sound field of audio recording, specified as a character vector or scalar string. The possible values for `SoundField` depend on the `Method`:

- `'ISO 532-1'` -- `'free'`, `'diffuse'`
- `'ISO 532-2'` -- `'free'`, `'diffuse'`, `'eardrum'`, `'earphones'`

Data Types: `char` | `string`

`EarphoneResponse` — Earphone response
`[0,0]` (default) | *M*-by-2 matrix

Earphone response, specified as an *M*-by-2 matrix containing *M* frequency-amplitude pairs that describe the earphone's deviations from a flat response. The form is as specified in an ISO 11904-1:2002 earphone correction file. Specify the frequency in increasing order in Hz. Specify the amplitude deviation in decibels. Intermediate values are computed by linear interpolation. Values out of the given range are set to the nearest frequency-amplitude pair. The default value corresponds to a flat response.

Dependencies

To specify `EarphoneResponse`, you must set `SoundField` to `'earphones'`.

Data Types: `single` | `double`

`PressureReference` — Reference pressure (Pa)
`20e-6` (default) | positive scalar

Reference pressure for dB calculation in pascals, specified as a positive scalar. The default value, 20 micropascals, is the common value for air.

Dependencies

`PressureReference` is only used for time-domain input signals.

Data Types: `single` | `double`

`Percentiles` — Percentiles at which to calculate percentile loudness
`[0,5]` (default) | vector with values in the range [0, 100]

Percentiles at which to calculate percentile loudness, specified as a vector with values in the range [0, 100]. The defaults, `0` and `5`, correspond to the N_{max} and N_5 percentiles, respectively [1].

Percentile loudness refers to the loudness that is reached or exceeded in *X*% of the measured time intervals, where *X* is the specified percentile.

Data Types: `single` | `double`

`TimeResolution` — Time resolution of the output
`'standard'` (default) | `'high'`

Time resolution of the output, specified as a character vector or scalar string. The time interval is 2 ms in `'standard'` resolution, or 0.5 ms in `'high'` resolution. The default is `'standard'` (ISO 532-1 compliant).

Data Types: `char` | `string`

Output Arguments

`loudness` — Loudness (sones)
K-by-1 | *K*-by-2

Loudness in sones, returned as a *K*-by-1 column vector or *K*-by-2 matrix of independent channels. If `TimeVarying` is set to `false`, *K* is equal to `1`. If `TimeVarying` is set to `true`, then `TimeResolution` determines how many times to compute the loudness. If `Method` is set to `'ISO 532-2'`, then `loudness` is computed using a binaural model and always returned as a *K*-by-1 column vector.

`specificLoudness` — Specific loudness
K-by-240-by-*C* | *K*-by-372-by-*C*

Specific loudness, returned as a *K*-by-240-by-*C* array or a *K*-by-372-by-*C* array. The first dimension of specific loudness, *K*, matches the first dimension of `loudness`. The third dimension of specific loudness, *C*, matches the second dimension of `loudness`. The second dimension of specific loudness depends on the `Method` used to calculate loudness:

- If `Method` is set to `'ISO 532-1'`, then specific loudness is reported in sones/Bark on a scale from 0.1 to 24, inclusive, in 0.1 increments.
- If `Method` is set to `'ISO 532-2'`, then specific loudness is reported in sones/Cam on a scale from 1.8 to 38.9, inclusive, in 0.1 increments.

`perc` — Percentile loudness (sones)
p-by-1 vector (mono input) | *p*-by-2 matrix (stereo input)

Percentile loudness in sones, returned as a *p*-by-1 vector or *p*-by-2 matrix. The number of rows, *p*, is equal to the number of `Percentiles`.

Percentile loudness refers to the loudness that is reached or exceeded in *X*% of the measured time intervals, where *X* is the specified percentile.

Dependencies

The percentiles output argument is valid only if `TimeVarying` is set to `true`. If `TimeVarying` is set to `false`, the `perc` output is empty.

Algorithms

Loudness and loudness level are perceptual attributes of sound. Due to differences among people, measurements of loudness and loudness level should be considered statistical estimators. The ISO

532 series specifies procedures for estimating loudness and loudness level as perceived by persons with ontologically normal hearing under specific listening conditions.

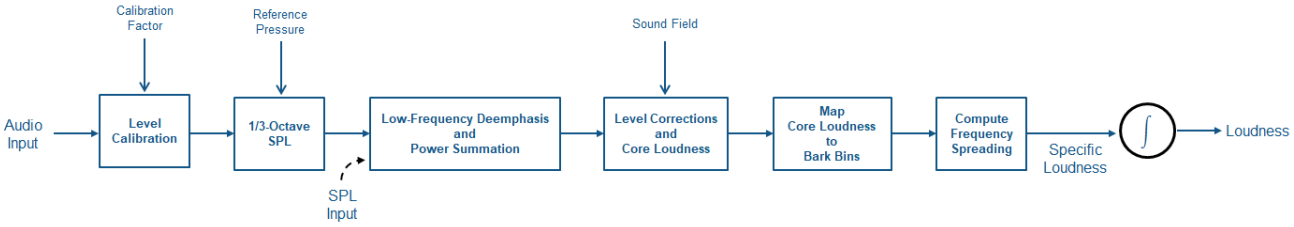
ISO 532-1 and ISO 532-2 specify two different methods for calculating loudness, but leave it to the user to select the appropriate method for a given situation.

ISO 532-1:2017(E) – Zwicker Method

ISO 532-1:2017(E) describes methods for calculating acoustic loudness of stationary and time-varying signals.

Stationary Signals

This method is based on DIN 45631:1991. The algorithm differs from ISO 532:1975, method B, by specifying corrections for low frequencies.

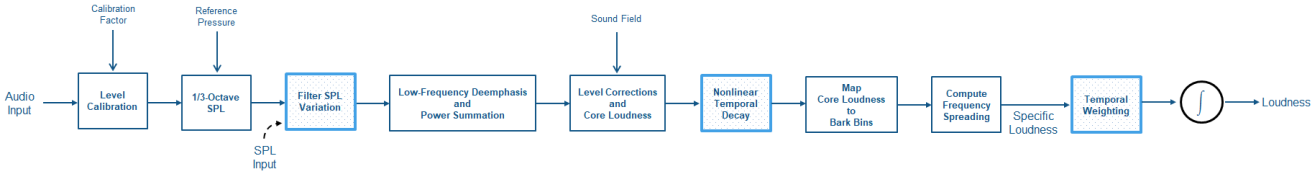


The diagram and the steps provide a high-level overview of the sequence of the method. For details, see [1].

1. The time-domain signal level is adjusted according to the `CalibrationFactor`. The following steps of the algorithm assume a true known signal level.
2. The signal is transformed to a 1/3 octave SPL representation using fractional octave band filtering. The filter bank consists of 28 filters between 25 Hz to 12.5 kHz. The output from this stage is in dB and normalized by the reference pressure.
3. Low frequency 1/3 octave bands are de-emphasized according to a fixed weighting table. Some of the low-frequency bands are combined to form a total of 20 critical bands.
4. The levels of the critical bands are corrected for filter bandwidth and the critical band level at the threshold of quiet, and then transformed to core loudness.
5. Core loudness is mapped to Bark bins.
6. Frequency spreading is computed using a table of level- and frequency-dependent slopes.
7. Loudness is calculated as the integral of specific loudness, taking into account the frequency-spreading slopes.

Time-Varying Signals

This method is based on DIN 45631/A1:2010, and is designed to properly simulate the duration-dependent behavior of loudness perception for short impulses. The method for time-varying sounds is a generalization of the Zwicker approach to stationary signals. If the generalized version is applied to stationary sounds, it gives the same loudness values as the non-generalized form for stationary signals.

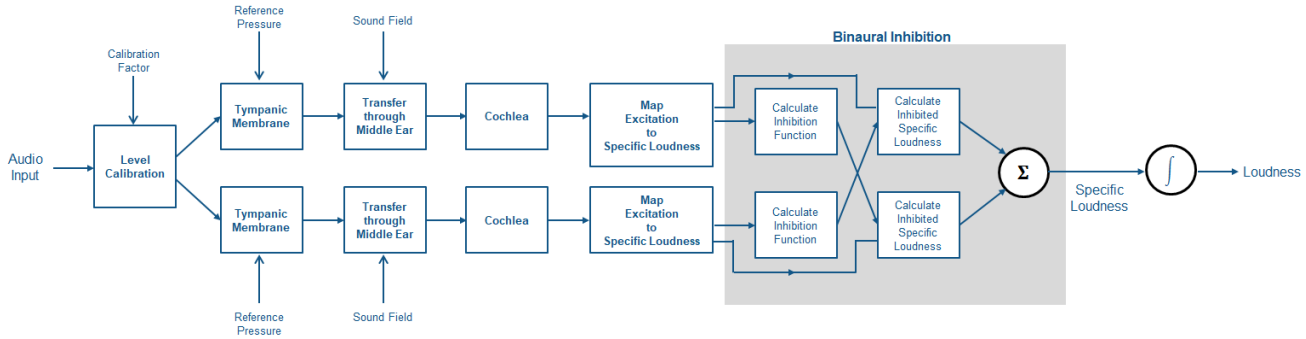


The diagram and the steps provide a high-level overview of the sequence of the method. For details, see [1].

1. The time-domain signal level is adjusted according to the `CalibrationFactor`. The following steps of the algorithm assume a true known signal level.
2. The signal is transformed to a 1/3 octave SPL representation using fractional octave band filtering. The filter bank consists of 28 filters between 25 Hz to 12.5 kHz. The output from this stage is in dB and normalized by the reference pressure.
3. The SPL bands are smoothed along time according to band-dependent filters.
4. Low frequency 1/3 octave bands are de-emphasized according to a fixed weighting table. Some of the low-frequency bands are combined to form a total of 20 critical bands.
5. The levels of the critical bands are corrected for filter bandwidth and the critical band level at the threshold of quiet, and then transformed to core loudness.
6. Nonlinear temporal decay is simulated using a diode-capacitor-resistor network. This models the steep perceptual drop after short signals when compared to long signals.
7. Core loudness is mapped to Bark bins.
8. Frequency spreading is computed using a table of level- and frequency-dependent slopes.
9. Temporal weighting is applied to simulate the duration-dependence of loudness perception.
10. Loudness is calculated as the integral of specific loudness, taking into account the frequency-spreading slopes.

ISO 532-2:2017(E) – Moore-Glasberg Method

ISO 532-2:2017(E) describes a binaural model for calculating acoustic loudness of stationary signals. The method in ISO 523-2 differs from those in ISO 532:1975: it improves the calculated loudness in the low frequency range and the binaural model allows for different sounds for each ear. ISO 532-2 provides a good match to the equal loudness level contours defined in ISO 226:2003, and the threshold of hearing defined in ISO 389-7:2005.



The diagram and the steps provide a high-level overview of the sequence of the method. For details, see [2].

1. The time-domain signal level is adjusted according to the `CalibrationFactor`. The following steps of the algorithm assume a true known signal level.
2. The signal is transformed to a spectral representation. The spectral representation is transformed according to fixed filters representing the transfer of sound through the tympanic membrane (eardrum). The spectrum is scaled according to the reference pressure.
3. The signal is transformed using a model of the inner ear. Again, the transfer function is given by a fixed filter specified in the standard. The filter choice depends on the specified sound field.
4. The signal is transformed from the sound spectrum to an excitation pattern at the basilar membrane. The transformation is accomplished using a series of rounded-exponential filters spread on the ERB scale.
5. The excitation pattern is converted to specific loudness.
6. The specific loudness is passed through a model of binary inhibition, where a signal at one ear inhibits the loudness evoked by a signal at the other ear. The output from this stage is the specific loudness in sones/ERB.
7. The specific loudness is integrated over the ERB scale to give the loudness in sones.

Version History

Introduced in R2020a

References

[1] ISO 532-1:2017(E). "Acoustics – Methods for calculating loudness – Part 1: Zwicker method." *International Organization for Standardization*.

[2] ISO 532-2:2017(E). "Acoustics – Methods for calculating loudness – Part 2: Moore-Glasberg method. *International Organization for Standardization*.

Extended Capabilities

C/C++ Code Generation

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

See Also

`splMeter` | `acousticSharpness` | `calibrateMicrophone` | `sone2phon` | `phon2sone` | `acousticFluctuation` | `acousticRoughness`

Topics

“Effect of Soundproofing on Perceived Noise Levels”

acousticSharpness

Perceived sharpness of acoustic signal

Syntax

```
sharpness = acousticSharpness(audioIn,fs)
sharpness = acousticSharpness(audioIn,fs,calibrationFactor)
sharpness = acousticSharpness(SPLIn)
sharpness = acousticSharpness(specificLoudnessIn)
sharpness = acousticSharpness( ___ ,Name,Value)
acousticSharpness( ___ ,TimeVarying,true)
```

Description

`sharpness = acousticSharpness(audioIn,fs)` returns sharpness in acum according to DIN 45692 [2] and ISO 532-1:2017(E) [1].

`sharpness = acousticSharpness(audioIn,fs,calibrationFactor)` specifies a nondefault microphone calibration factor used to compute loudness.

`sharpness = acousticSharpness(SPLIn)` computes sharpness using one-third-octave-band sound pressure levels (SPL).

`sharpness = acousticSharpness(specificLoudnessIn)` computes sharpness using specific loudness.

`sharpness = acousticSharpness( ___ ,Name,Value)` specifies options using one or more `Name,Value` pair arguments.

Example: `sharpness = acousticSharpness(audioIn,fs,calibrationFactor,'SoundField','diffuse')` returns sharpness assuming a diffuse sound field.

`acousticSharpness( ___ ,TimeVarying,true)` with no output arguments plots sharpness relative to time.

Examples

Acoustic Sharpness of Audio Signal

Compute the acoustic sharpness of turbine noise. Assume it is stationary and was recorded in a diffuse sound field.

```
[audioIn,fs] = audioread('Turbine-16-44p1-mono-22secs.wav');

sharpness = acousticSharpness(audioIn,fs,'SoundField','diffuse');

fprintf('Acoustic sharpness = %0.2f acum\n',sharpness)
```

```
Acoustic sharpness = 1.11 acum
```

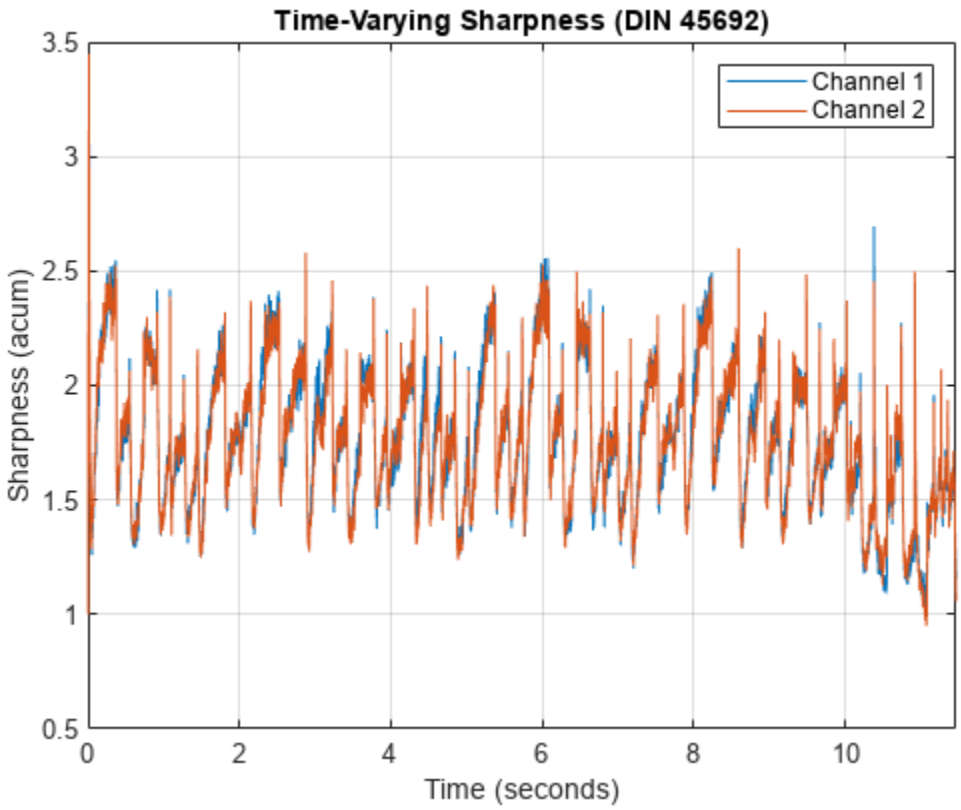
Time-Varying Sharpness

Read in an audio signal.

```
[audioIn,fs] = audioread('RockDrums-48-stereo-11secs.mp3');
```

Plot the time-varying sharpness of the signal. Listen to the signal.

```
acousticSharpness(audioIn,fs,'TimeVarying',true)
```




```
sound(audioIn,fs)
```

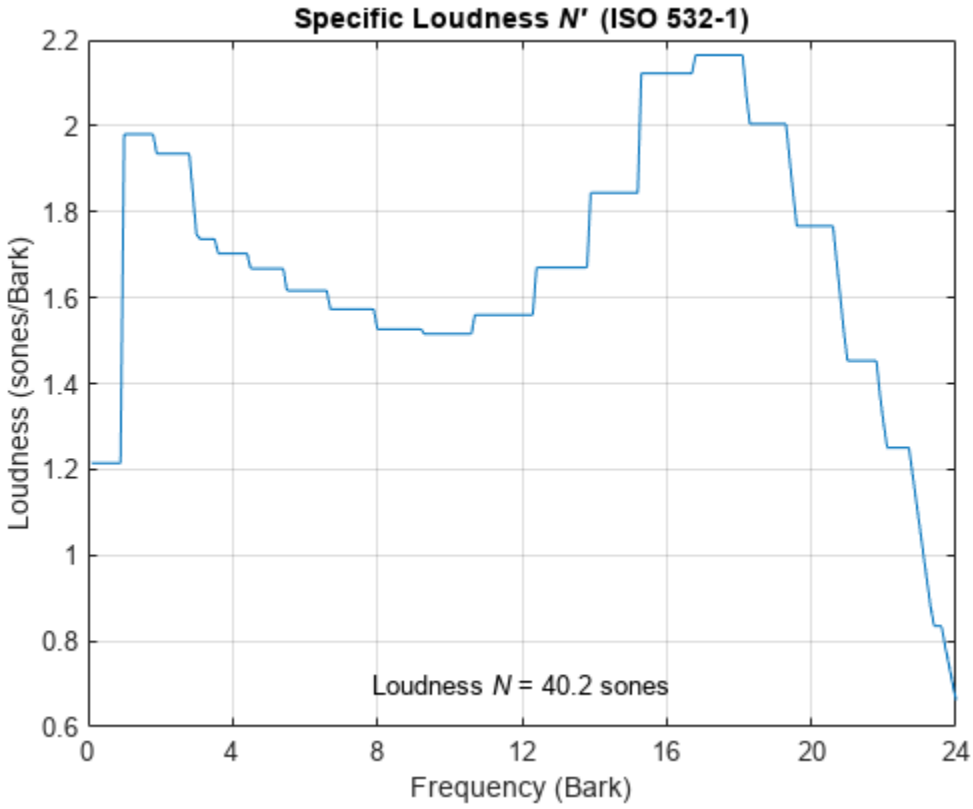
Measure Loudness and Sharpness of Stationary Signals

Create two stationary signals with equivalent power: a pink noise signal and a white noise signal.

```
fs = 48e3;
dur = 5;
pnoise = 2*pinknoise(dur*fs);
wnoise = rand(dur*fs,1) - 0.5;
wnoise = wnoise*sqrt(var(pnoise)/var(wnoise));
```

Call `acousticLoudness` using the default ISO 532-1 (Zwicker) method and no output arguments to plot the loudness of the pink noise. Call `acousticLoudness` again, this time with output arguments, to get the specific loudness.

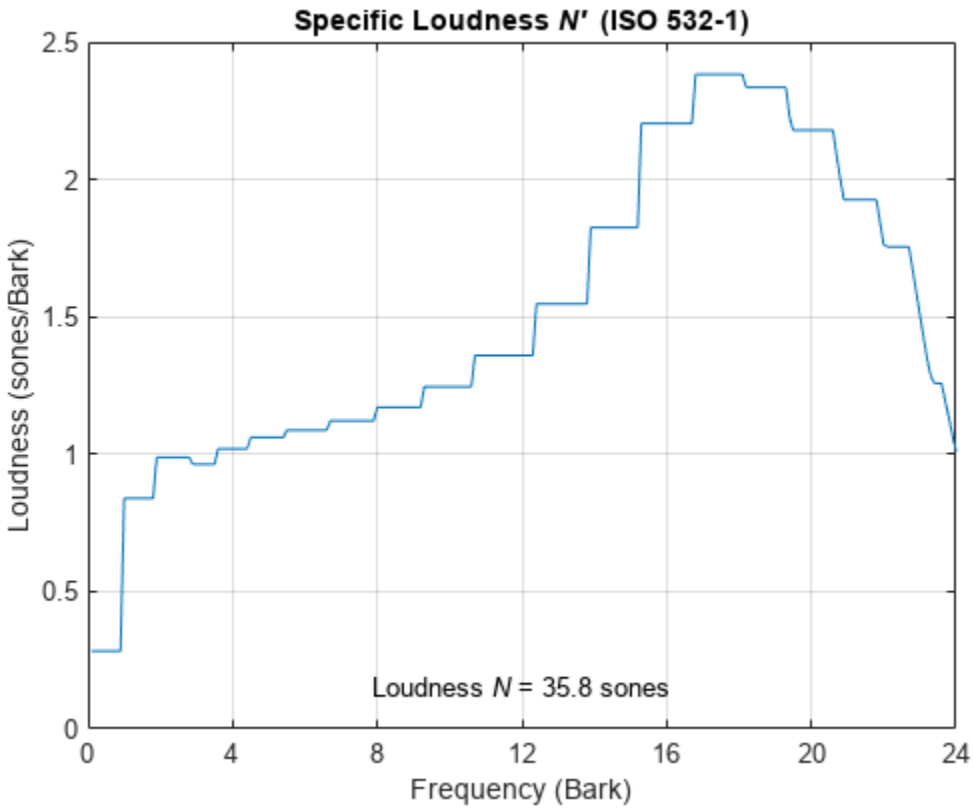
```
figure
acousticLoudness(pnoise,fs)
```




```
[~,pSpecificLoudness] = acousticLoudness(pnoise,fs);
```

Plot the loudness for the white noise signal and then get the specific loudness values.

```
figure
acousticLoudness(wnoise,fs)
```




```
[~,wSpecificLoudness] = acousticLoudness(wnoise,fs);
```

Call the `acousticSharpness` function to compare the sharpness of the pink noise and white noise.

```
pSharpness = acousticSharpness(pSpecificLoudness);
wSharpness = acousticSharpness(wSpecificLoudness);
fprintf('Sharpness of pink noise = %0.2f acum\n',pSharpness)
```

```
Sharpness of pink noise = 2.00 acum
```

```
fprintf('Sharpness of white noise = %0.2f acum\n',wSharpness)
```

```
Sharpness of white noise = 2.62 acum
```

Effect of Input Levels on Acoustic Sharpness

Create a pink noise signal with a 48 kHz sample rate and a duration of 5 seconds.

```
fs = 48e3;
n = fs*5;
pnoise = pinknoise(n);
```

Specify a vector to sweep over the dB range from -60 to 20. Create a gain vector which, when multiplied by the original signal, results in a signal with the desired output level.

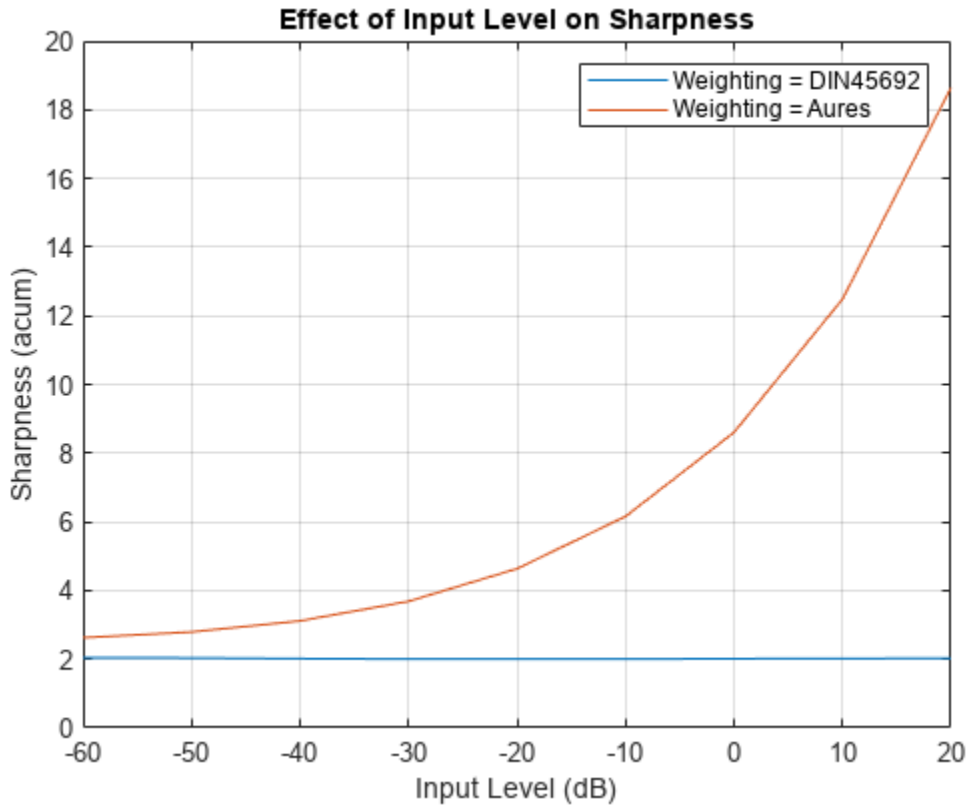
```
dBSweep = -60:10:20;
coefSweep = sqrt((10.^(dBSweep/10))/var(pnoise));
```

Call `acousticSharpness` in a loop with the different signal levels. Determine the sharpness using the default DIN 45692 frequency weighting and the Aures frequency weighting.

```
sharpnessDIN45692 = zeros(numel(dBSweep),1);
sharpnessAures = zeros(numel(dBSweep),1);
for ii = 1:numel(dBSweep)
    signal = pnoise*coefSweep(ii);
    sharpnessDIN45692(ii) = acousticSharpness(signal,fs);
    sharpnessAures(ii) = acousticSharpness(signal,fs,'Weighting','Aures');
end
```

Display the effect of the input level on the acoustic sharpness. The Aures frequency weighting method is more sensitive to the input level.

```
plot(dBSweep,sharpnessDIN45692,dBSweep,sharpnessAures)
legend('Weighting = DIN45692','Weighting = Aures')
xlabel('Input Level (dB)')
ylabel('Sharpness (acum)')
title('Effect of Input Level on Sharpness')
axis([dBSweep(1) dBSweep(end) 0 20])
grid on
```



Compare Time-Varying Sharpness of Music Genres

Read in two audio files: one of an electric guitar with distortion and one of an acoustic guitar. Both audio files have a sample rate of 44.1 kHz. For easy comparison, convert the rock guitar signal to mono and shorten the soft guitar signal to the length of the rock guitar signal.

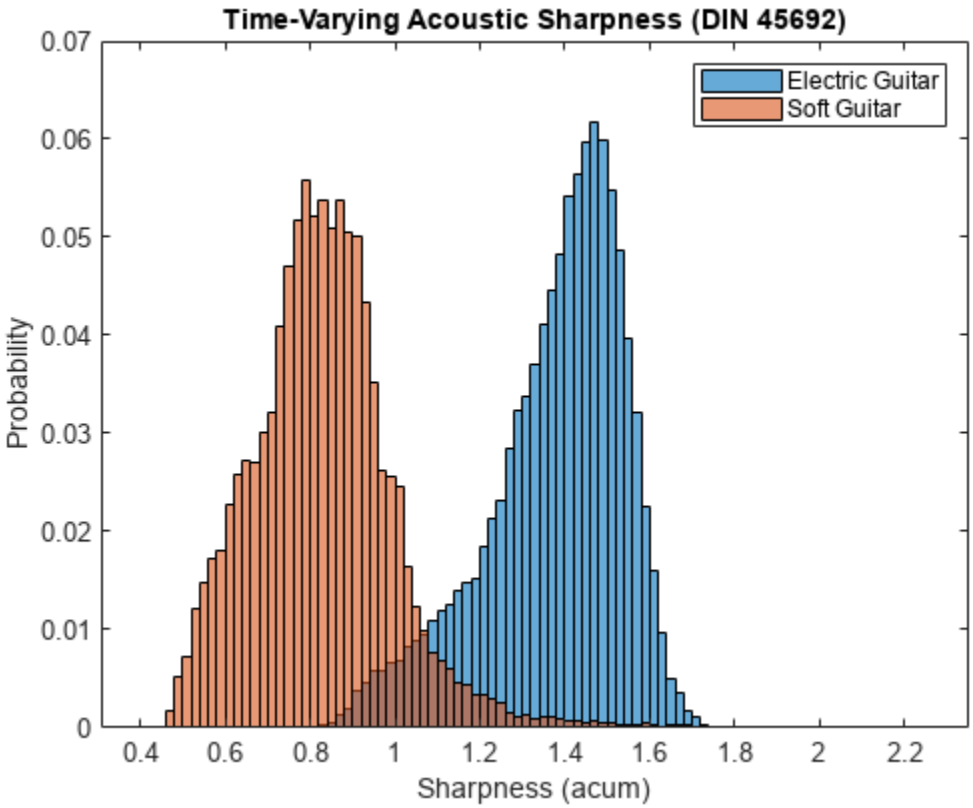
```
fs = 44.1e3;
rockGuitar = audioread('RockGuitar-16-44p1-stereo-72secs.wav');
softGuitar = audioread('SoftGuitar-44p1_mono-10mins.ogg');
rockGuitar = mean(rockGuitar,2);
softGuitar = softGuitar(1:numel(rockGuitar));
```

Calculate the time-varying sharpness for both the rock guitar and soft guitar.

```
rGSharpness = acousticSharpness(rockGuitar,fs,'TimeVarying',true);
sGSharpness = acousticSharpness(softGuitar,fs,'TimeVarying',true);
```

Plot the probability distribution based on the observed sharpness of the rock guitar and the soft guitar.

```
histogram(rGSharpness,'Normalization','probability')
hold on
histogram(sGSharpness,'Normalization','probability')
legend('Electric Guitar','Soft Guitar')
xlabel('Sharpness (acum)')
ylabel('Probability')
title('Time-Varying Acoustic Sharpness (DIN 45692)')
```



Measure Acoustic Sharpness from Sound Pressure Level

Read in an audio file.

```
[audioIn,fs] = audioread('Turbine-16-44p1-mono-22secs.wav');
```

To calculate sound pressure levels from an audio signal, first create an `splMeter` object. Call the `splMeter` object with the audio input.

```
spl = splMeter("SampleRate",fs,"Bandwidth","1/3 octave", ...
    "FrequencyWeighting","Z-weighting","OctaveFilterOrder",6);

splMeasurement = spl(audioIn);
```

Compute the mean SPL level, skipping the first 0.2 seconds. Only keep the bands from 25 Hz to 12.5 kHz (the first 28 bands).

```
SPLIn = mean(splMeasurement(ceil(0.2*fs):end,1:28));
```

To determine the acoustic sharpness of the audio signal, call `acousticSharpness` using the sound pressure level input.

```
sharpness = acousticSharpness(SPLIn)
```

```
sharpness = 1.1015
```

Input Arguments

`audioIn` — Audio input

column vector | 2-column matrix

Audio input, specified as a column vector (mono) or matrix with two columns (stereo). Sharpness is computed for each channel (column) independently.

Data Types: `single` | `double`

`fs` — Sample rate (Hz)

positive scalar

Sample rate in Hz, specified as a positive scalar. The recommended sample rate for new recordings is 48 kHz.

Note The minimum acceptable sample rate is 8 kHz.

Data Types: `single` | `double`

`calibrationFactor` — Microphone calibration factor

`sqrt(8)` | positive scalar

Microphone calibration factor, specified as a positive scalar. The default calibration factor corresponds to a full-scale 1 kHz sine wave with a sound pressure level of 100 dB (SPL). To compute the calibration factor specific to your system, use the `calibrateMicrophone` function.

Data Types: `single` | `double`

SPLIn — Sound pressure level (dB)
1-by-28-by-*C*

Sound pressure level (SPL) in dB, specified as a 1-by-28-by-*C* array. 28 corresponds to one-third-octave bands between 25 Hz and 12.5 kHz. *C* is the number of channels.

Data Types: `single` | `double`

`specificLoudnessIn` — Specific loudness (sones/Bark)
T-by-240-by-*C*

Specific loudness in sones/Bark, specified as a *T*-by-240-by-*C* array, where:

- *T* is 1 for stationary signals or one per 2 ms for time-varying signals.
- 240 is the number of Bark bins in the domain for specific loudness. The Bark bins are `0.1:0.1:24`.
- *C* is the number of channels.

You can use the `acousticLoudness` function to calculate `specificLoudnessIn` using this syntax:

```
[~,specificLoudnessIn] = acousticLoudness(audioIn,fs);
```

Data Types: `single` | `double`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `acousticSharpness(audioIn,fs,'Weighting','von Bismarck')`

`Weighting` — Frequency weighting
`'DIN 45692'` (default) | `'Aures'` | `'von Bismarck'`

Frequency weighting, specified as `'DIN 45692'`, `'Aures'`, or `'von Bismarck'`. By design, the `'Aures'` frequency weighting method is more sensitive to amplitude levels and proper calibration. For details, see “Algorithms” on page 2-637.

Data Types: `char` | `string`

`SoundField` — Sound field
`'free'` (default) | `'diffuse'`

Sound field of audio recording, specified as `'free'` or `'diffuse'`.

Data Types: `char` | `string`

`PressureReference` — Reference pressure (Pa)
`20e-6` (default) | positive scalar

Reference pressure for dB calculation in pascals, specified as a positive scalar. The default value, 20 micropascals, is the common value for air.

Data Types: `single` | `double`

`TimeVarying` — Input is time-varying
`false` (default) | `true`

Input is time-varying, specified as `true` or `false`. If `TimeVarying` is set to `true`, acoustic sharpness is calculated in 2 ms intervals.

Data Types: `logical`

Output Arguments

`sharpness` — Acoustic sharpness (acum)
scalar | vector | matrix

Acoustic sharpness in acum, returned as a scalar, vector, or matrix. Sharpness is computed according to DIN 45692 and ISO 532-1.

Data Types: `single` | `double`

Algorithms

Acoustic sharpness is a measurement derived from acoustic loudness. The acoustic loudness algorithm is described in [1] and implemented in the `acousticLoudness` function. The acoustic sharpness calculation is described in [2]. The algorithm for acoustic sharpness is outlined as follows.

$$sharpness = k\left(\frac{\int_{z=0}^{24} N'(z)\,g(z)\,z\,\mathrm{d}z}{\int_{z=0}^{24} N'(z)\,\mathrm{d}z}\right)$$

Where N' is the specific loudness in sones/Bark. The function $g(z)$ and the scaling factor k depend on the specified `Weighting` method:

`'DIN 45692'`: k is set such that a 1 kHz reference tone results in a 1 acum sharpness measurement, and

$$g(z) = 1 \quad \text{for } z \le 15.8 \text{ Bark}$$
$$g(z) = 0.15e^{0.42(z-15.8)} + 0.85 \quad \text{for } z > 15.8 \text{ Bark}$$

`'von Bismark'`: k is set to `0.11`, and

$$g(z) = 1 \quad \text{for } z \le 15 \text{ Bark}$$
$$g(z) = 0.2e^{0.308(z-15)} + 0.8 \quad \text{for } z > 15 \text{ Bark}$$

`'Aures'`: k is set to `0.11`, and

$$g(z) = 0.078\left(\frac{e^{0.171z}}{z}\right)\left(\frac{N}{\ln(0.05N + 1)}\right)$$

where

$$N = \int_{z=0}^{24} N'(z)\,\mathrm{d}z$$

Version History

Introduced in R2020a

References

[1] ISO 532-1:2017(E). "Acoustics – Methods for calculating loudness – Part 1: Zwicker method." *International Organization for Standardization*.

[2] DIN 45692:2009. "Measurement Technique for the Simulation of the Auditory Sensation of Sharpness." *German Institute for Standardization*.

Extended Capabilities

C/C++ Code Generation

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

See Also

`splMeter` | `acousticLoudness` | `calibrateMicrophone` | `sone2phon` | `phon2sone` | `acousticFluctuation` | `acousticRoughness`

Topics

“Effect of Soundproofing on Perceived Noise Levels”

detectSpeech

Detect boundaries of speech in audio signal

Syntax

```
idx = detectSpeech(audioIn,fs)
idx = detectSpeech(audioIn,fs,Name,Value)
[idx,thresholds] = detectSpeech( ___ )
detectSpeech( ___ )
```

Description

`idx = detectSpeech(audioIn,fs)` returns indices of `audioIn` that correspond to the boundaries of speech signals.

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

Example: `detectSpeech(audioIn,fs,'Window',hann(512,'periodic'),'OverlapLength',256)` detects speech using a 512-point periodic Hann window with 256-point overlap.

`[idx,thresholds] = detectSpeech( ___ )` also returns the thresholds used to compute the boundaries of speech.

`detectSpeech( ___ )` with no output arguments displays a plot of the detected speech regions in the input signal.

Examples

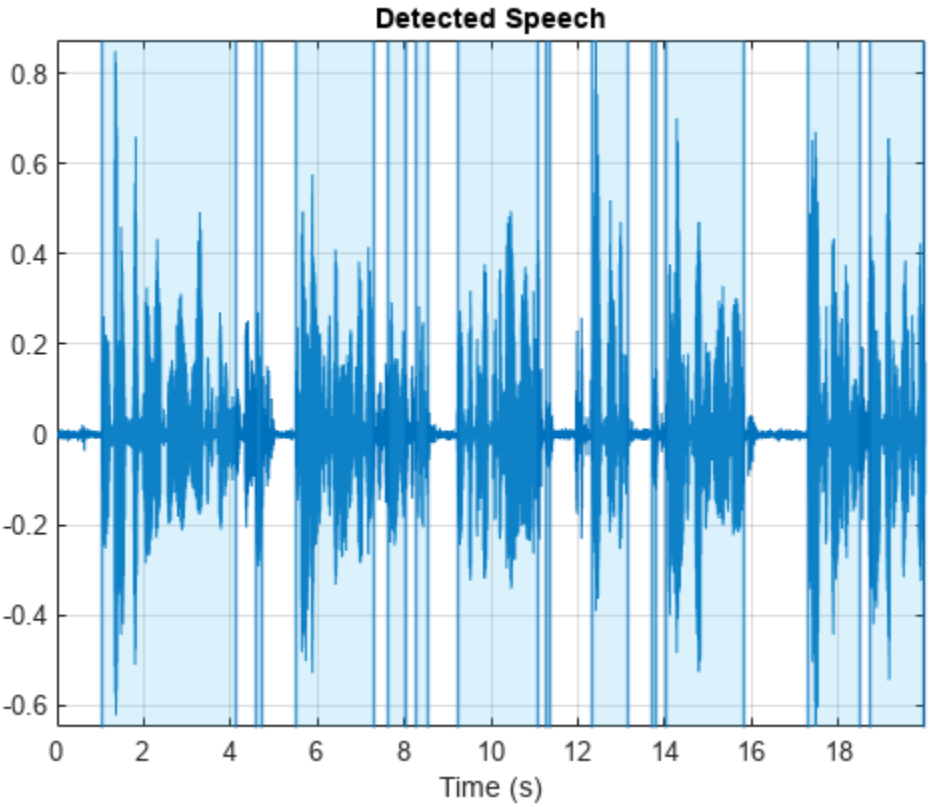
Plot Detected Regions of Speech

Read in an audio signal. Clip the audio signal to 20 seconds.

```
[audioIn,fs] = audioread('Rainbow-16-8-mono-114secs.wav');
audioIn = audioIn(1:20*fs);
```

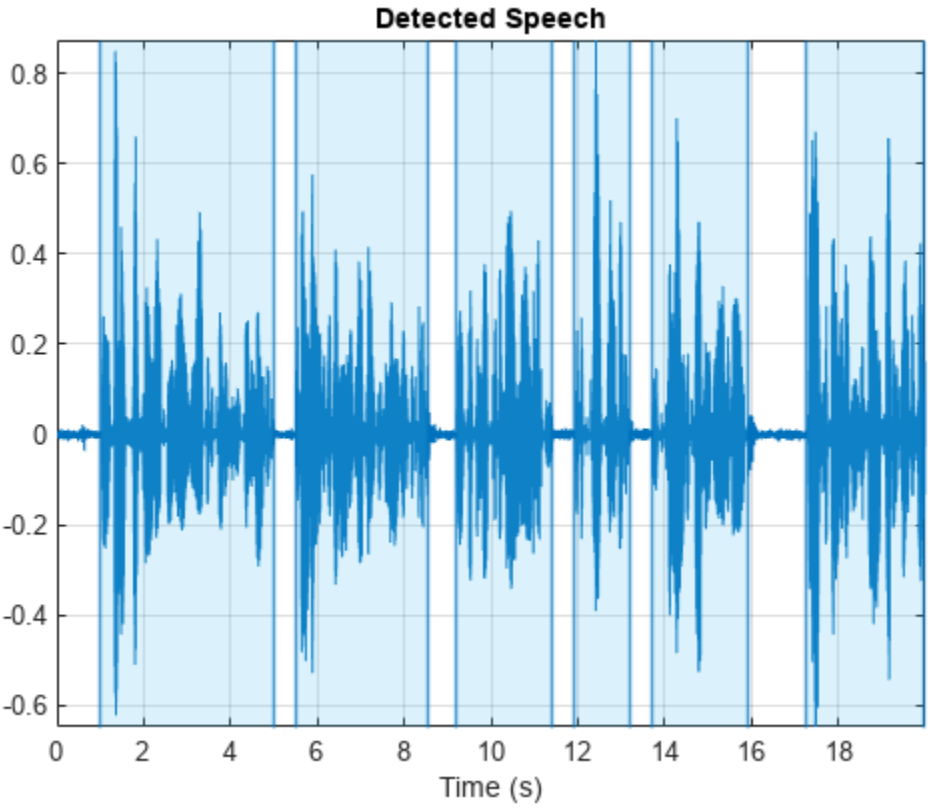
Call `detectSpeech`. Specify no output arguments to display a plot of the detected speech regions.

```
detectSpeech(audioIn,fs);
```



The `detectSpeech` function uses a thresholding algorithm based on energy and spectral spread per analysis frame. You can modify the `Window`, `OverlapLength`, and `MergeDistance` to fine-tune the algorithm for your specific needs.

```
windowDuration = 0.074 ; % seconds
numWindowSamples = round(windowDuration*fs);
win = hamming(numWindowSamples,'periodic');

percentOverlap = 35 ;
overlap = round(numWindowSamples*percentOverlap/100);

mergeDuration = 0.44 ;
mergeDist = round(mergeDuration*fs);

detectSpeech(audioIn,fs,"Window",win,"OverlapLength",overlap,"MergeDistance",mergeDist)
```



Reuse Decision Thresholds

Read in an audio file containing speech. Split the audio signal into a first half and a second half.

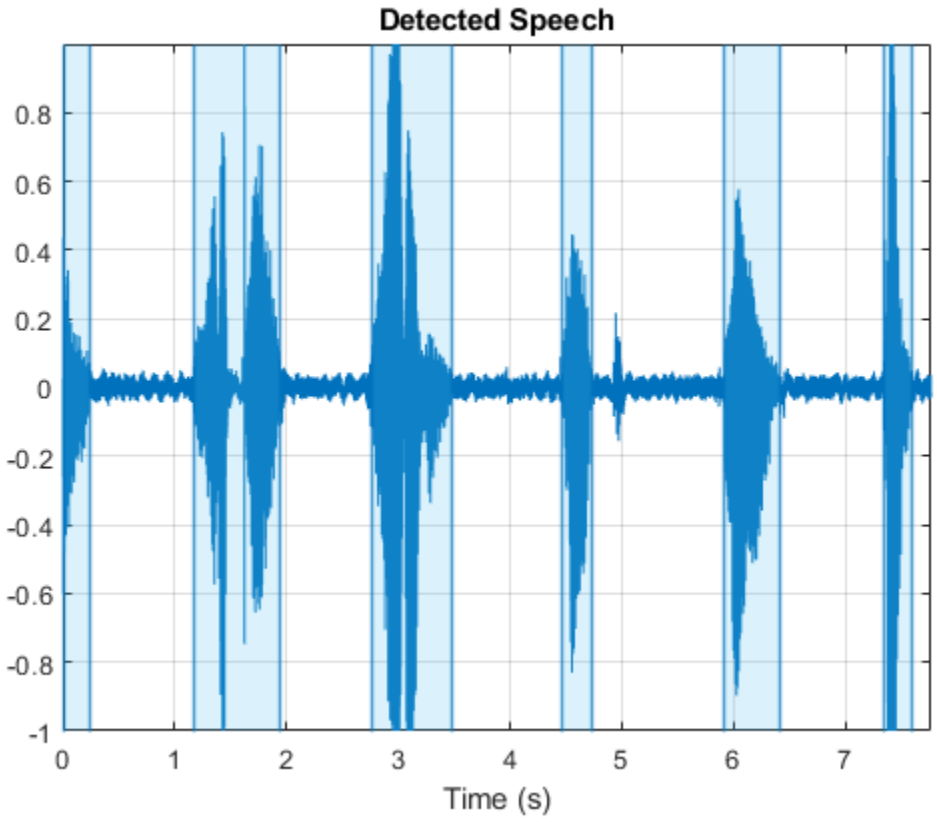
```
[audioIn,fs] = audioread('Counting-16-44p1-mono-15secs.wav');
firstHalf = audioIn(1:floor(numel(audioIn)/2));
secondHalf = audioIn(numel(firstHalf):end);
```

Call `detectSpeech` on the first half of the audio signal. Specify two output arguments to return the indices corresponding to regions of detected speech and the thresholds used for the decision.

```
[speechIndices,thresholds] = detectSpeech(firstHalf,fs);
```

Call `detectSpeech` on the second half with no output arguments to plot the regions of detected speech. Specify the thresholds determined from the previous call to `detectSpeech`.

```
detectSpeech(secondHalf,fs,'Thresholds',thresholds)
```



Working with Large Data Sets

Reusing speech detection thresholds provides significant computational efficiency when you work with large data sets, or when you deploy a deep learning or machine learning pipeline for real-time inference. Download and extract the data set [1] on page 2-645.

```
url = 'https://storage.googleapis.com/download.tensorflow.org/data/speech_commands_v0.01.tar.gz'

downloadFolder = tempdir;
datasetFolder = fullfile(downloadFolder,'google_speech');

if ~exist(datasetFolder,'dir')
    disp('Downloading data set (1.9 GB) ...')
    untar(url,datasetFolder)
end
```

Create an audio datastore to point to the recordings. Use the folder names as labels.

```
ads = audioDatastore(datasetFolder,'IncludeSubfolders',true,'LabelSource','foldernames');
```

Reduce the data set by 95% for the purposes of this example.

```
ads = splitEachLabel(ads,0.05,'Exclude','_background_noise');
```

Create two datastores: one for training and one for testing.

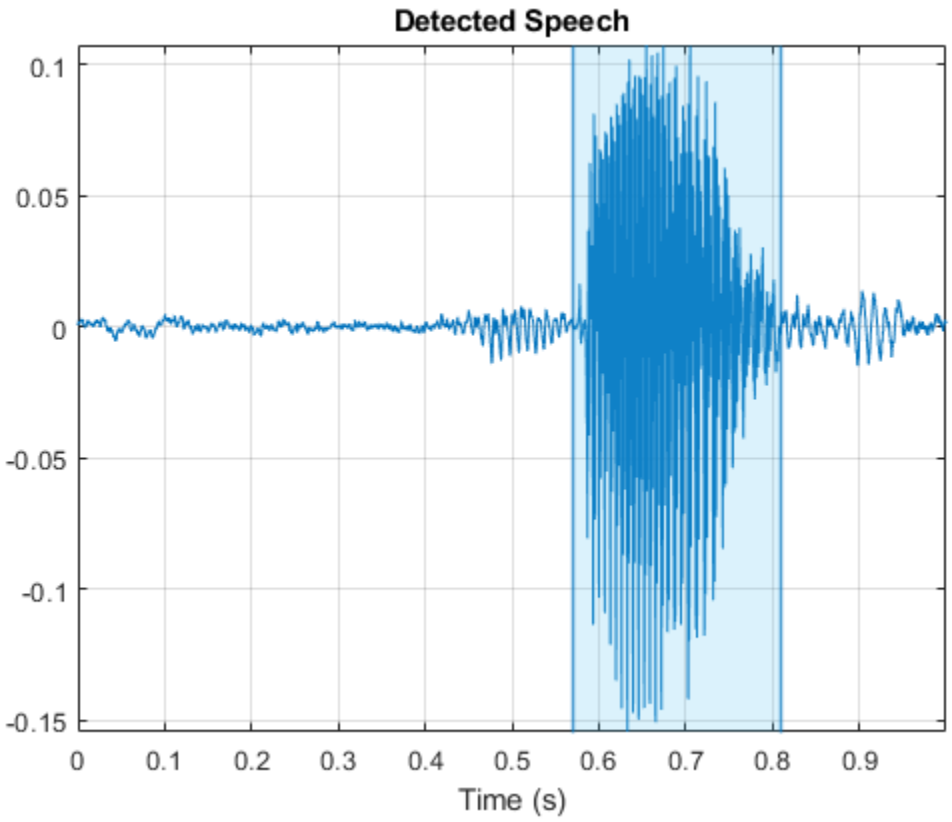
```
[adsTrain,adsTest] = splitEachLabel(ads,0.8);
```

Compute the average thresholds over the training data set.

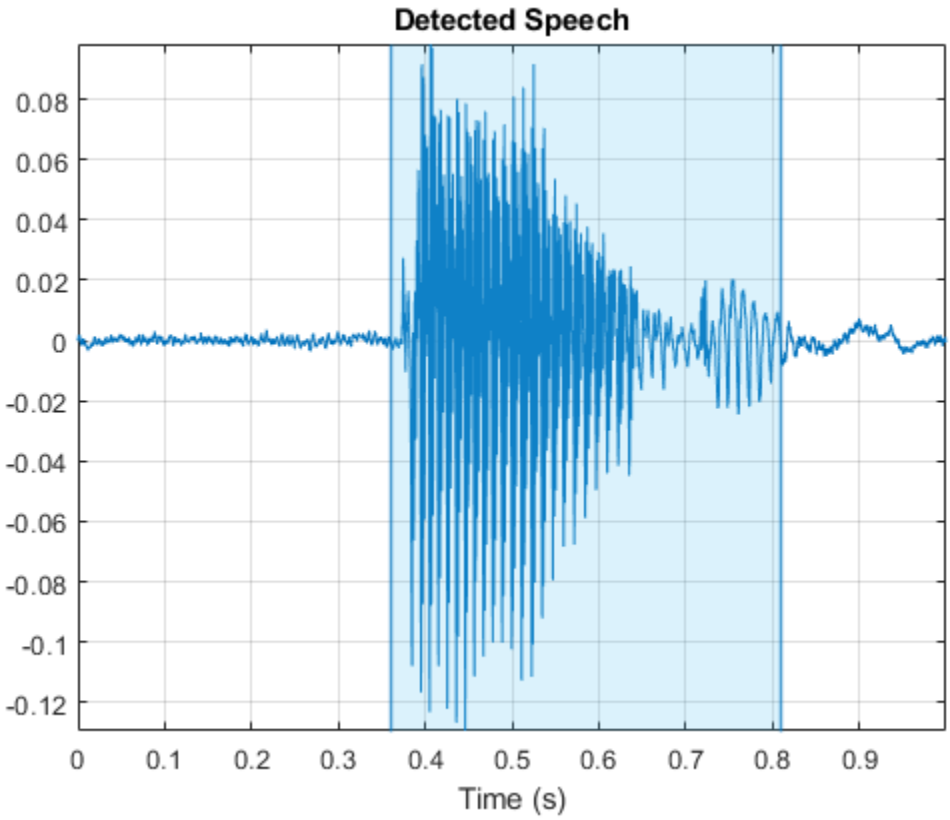
```
thresholds = zeros(numel(adsTrain.Files),2);
for ii = 1:numel(adsTrain.Files)
    [audioIn,adsInfo] = read(adsTrain);
    [~,thresholds(ii,:)] = detectSpeech(audioIn,adsInfo.SampleRate);
end
thresholdAverage = mean(thresholds,1);
```

Use the precomputed thresholds to detect speech regions on files from the test data set. Plot the detected region for three files.

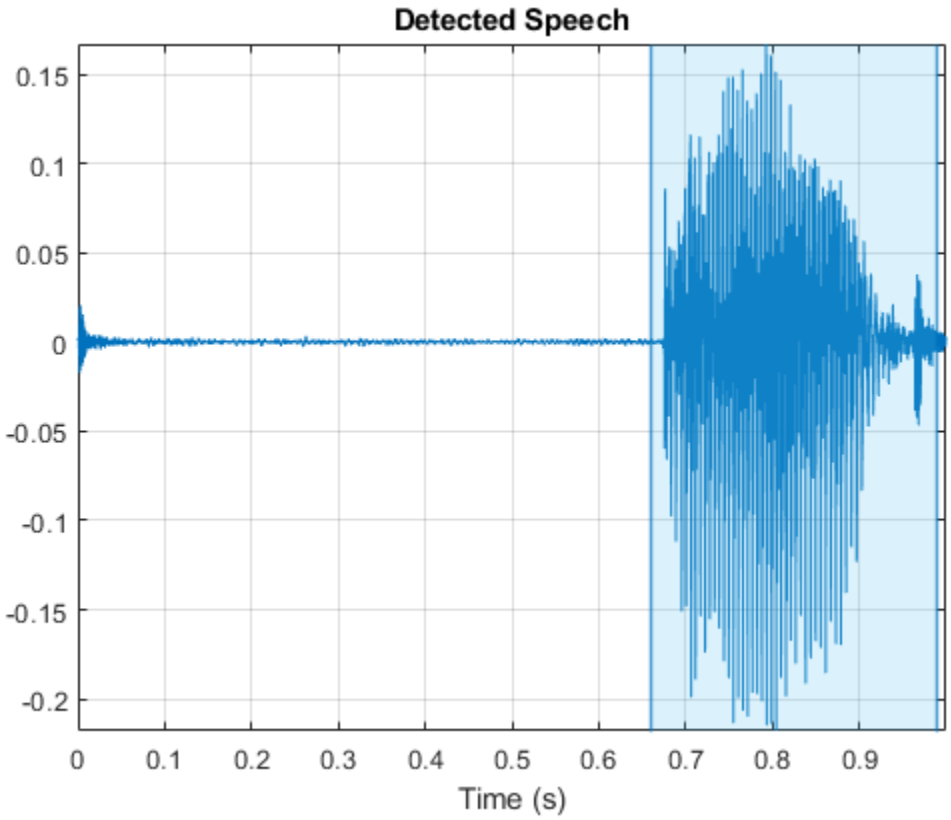
```
[audioIn,adsInfo] = read(adsTest);
detectSpeech(audioIn,adsInfo.SampleRate,'Thresholds',thresholdAverage);
```




```
[audioIn,adsInfo] = read(adsTest);
detectSpeech(audioIn,adsInfo.SampleRate,'Thresholds',thresholdAverage);
```




```
[audioIn,adsInfo] = read(adsTest);
detectSpeech(audioIn,adsInfo.SampleRate,'Thresholds',thresholdAverage);
```



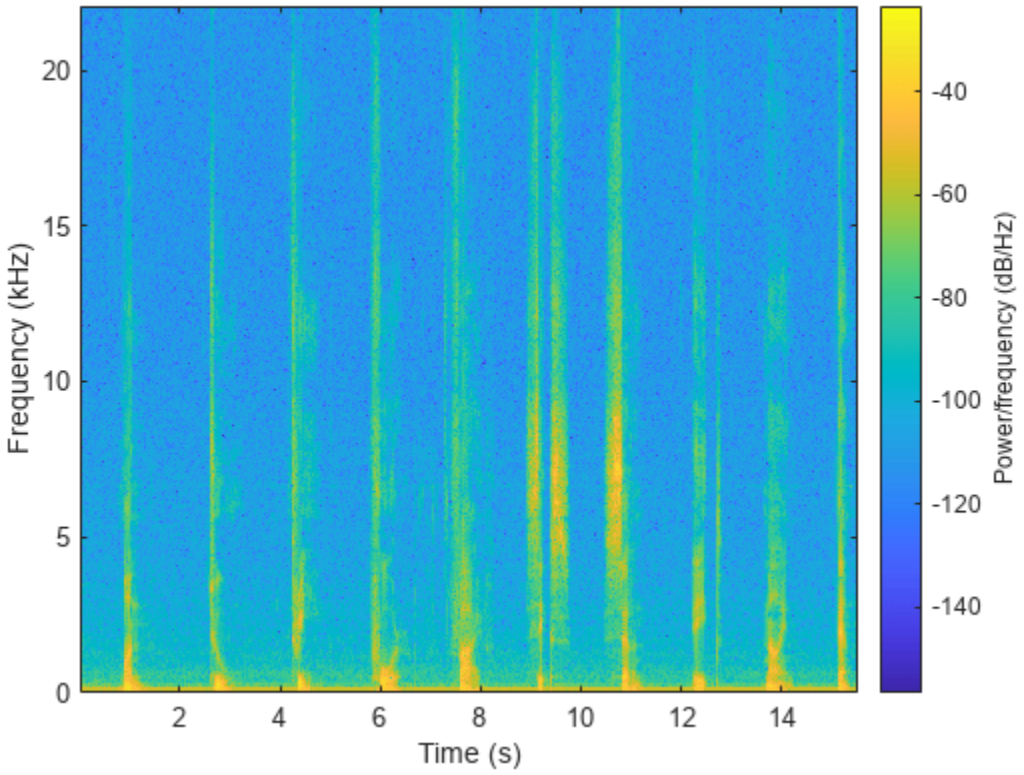
References

[1] Warden, Pete. "Speech Commands: A Public Dataset for Single Word Speech Recognition." Distributed by TensorFlow. Creative Commons Attribution 4.0 License.

Remove Silent Regions from Speech Signal

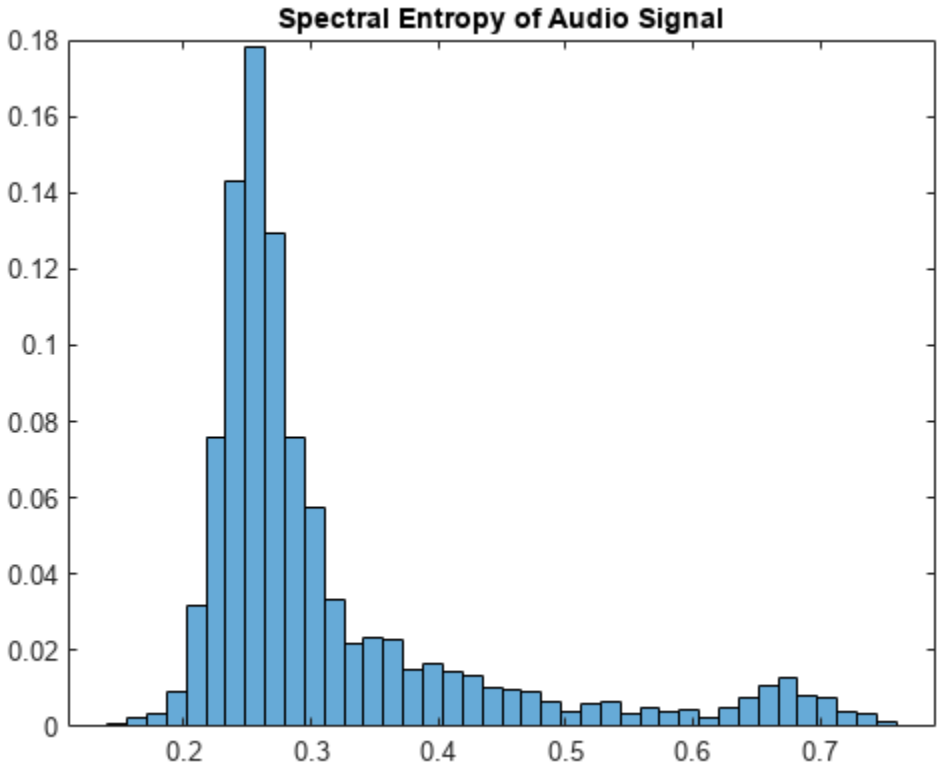
Read in an audio file and listen to it. Plot the `spectrogram`.

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

sound(audioIn,fs)

spectrogram(audioIn,hann(1024,'periodic'),512,1024,fs,'yaxis')
```



For machine learning applications, you often want to extract features from an audio signal. Call the `spectralEntropy` function on the audio signal, then plot the `histogram` to display the distribution of spectral entropy.

```
entropy = spectralEntropy(audioIn,fs);

numBins = 40;
histogram(entropy,numBins,'Normalization','probability')
title('Spectral Entropy of Audio Signal')
```



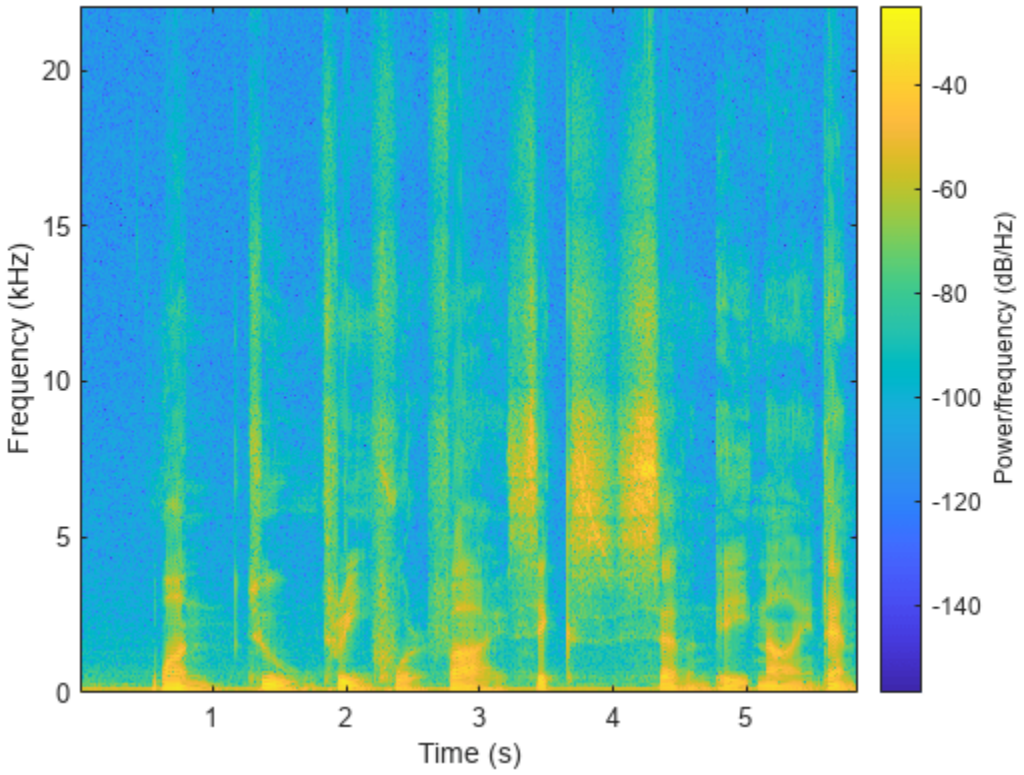
Depending on your application, you might want to extract spectral entropy from only the regions of speech. The resulting statistics are more characteristic of the speaker and less characteristic of the channel. Call `detectSpeech` on the audio signal and then create a new signal that contains only the regions of detected speech.

```
speechIndices = detectSpeech(audioIn,fs);
speechSignal = [];
for ii = 1:size(speechIndices,1)
    speechSignal = [speechSignal;audioIn(speechIndices(ii,1):speechIndices(ii,2))];
end
```

Listen to the speech signal and plot the spectrogram.

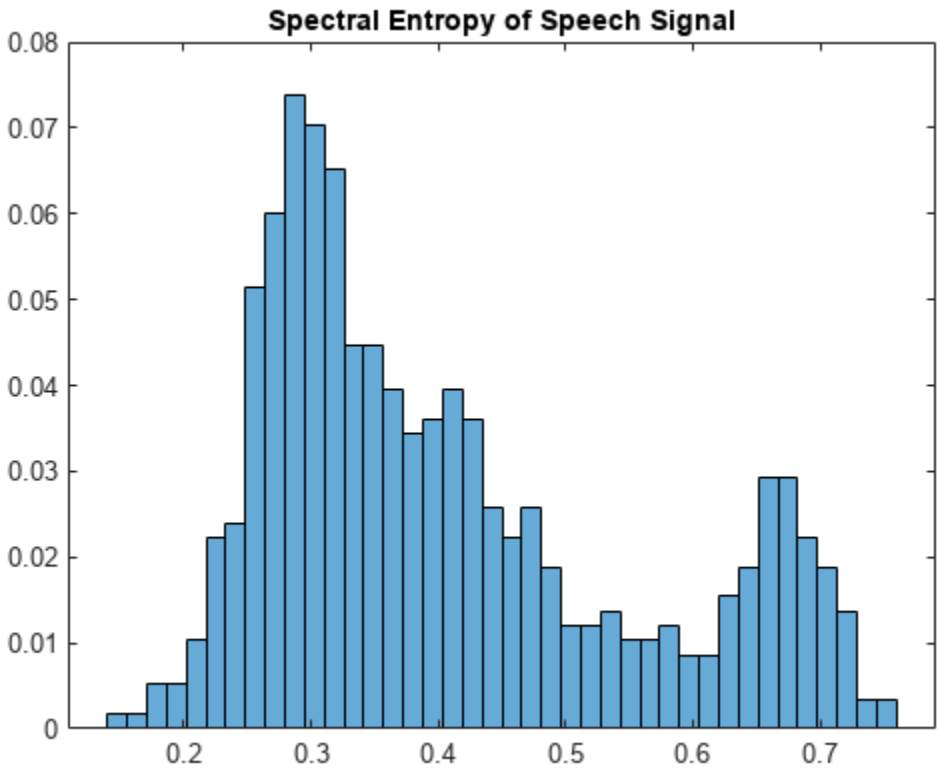
```
sound(speechSignal,fs)
```

```
spectrogram(speechSignal,hann(1024,'periodic'),512,1024,fs,'yaxis')
```



Call the `spectralEntropy` function on the speech signal and then plot the `histogram` to display the distribution of spectral entropy.

```
entropy = spectralEntropy(speechSignal,fs);

histogram(entropy,numBins,'Normalization','probability')
title('Spectral Entropy of Speech Signal')
```



Input Arguments

`audioIn` — Audio input

column vector

Audio input, specified as a column vector.

Data Types: `single` | `double`

`fs` — Sample rate (Hz)

scalar

Sample rate in Hz, specified as a scalar.

Data Types: `single` | `double`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `detectSpeech(audioIn,fs,'MergeDistance',100)`

`Window` — Window applied in time domain
`hann(round(0.03*fs),'periodic')` (default) | vector

Window applied in the time domain, specified as the comma-separated pair consisting of '`Window`' and a real vector. The number of elements in the vector must be in the range [2, `size(audioIn,1)`]. The number of elements in the vector must also be greater than `OverlapLength`.

Data Types: `single` | `double`

`OverlapLength` — Number of samples overlapping between adjacent windows
`0` (default) | scalar in the range [0, `numel(Window)−1`]

Number of samples overlapping between adjacent windows, specified as the comma-separated pair consisting of '`OverlapLength`' and an integer in the range [0, `size(Window,1)`).

Data Types: `single` | `double`

`MergeDistance` — Number of samples over which to merge positive speech detection decisions
`numel(Window)`*5 (default) | nonnegative scalar

Number of samples over which to merge positive speech detection decisions, specified as the comma-separated pair consisting of '`MergeDistance`' and a nonnegative scalar.

Note The resolution for speech detection is given by the hop length, where the hop length is equal to `numel(Window) − OverlapLength`.

Data Types: `single` | `double`

`Thresholds` — Thresholds for decision
2-element vector

Thresholds for decision, specified as the comma-separated pair consisting of '`Thresholds`' and a two-element vector.

- If you do not specify `Thresholds`, the `detectSpeech` function derives thresholds by using histograms of the features calculated over the current input frame.
- If you specify `Thresholds`, the `detectSpeech` function skips the derivation of new decision thresholds. Reusing speech decision thresholds provides significant computational efficiency when you work with large data sets, or when you deploy a deep learning or machine learning pipeline for real-time inference.

Data Types: `single` | `double`

Output Arguments

`idx` — Start and end indices of speech regions
N-by-2 matrix

Start and end indices of speech regions, returned as an *N*-by-2 matrix. *N* corresponds to the number of individual speech regions detected. The first column corresponds to start indices of speech regions and the second column corresponds to end indices of speech regions.

Data Types: `single` | `double`

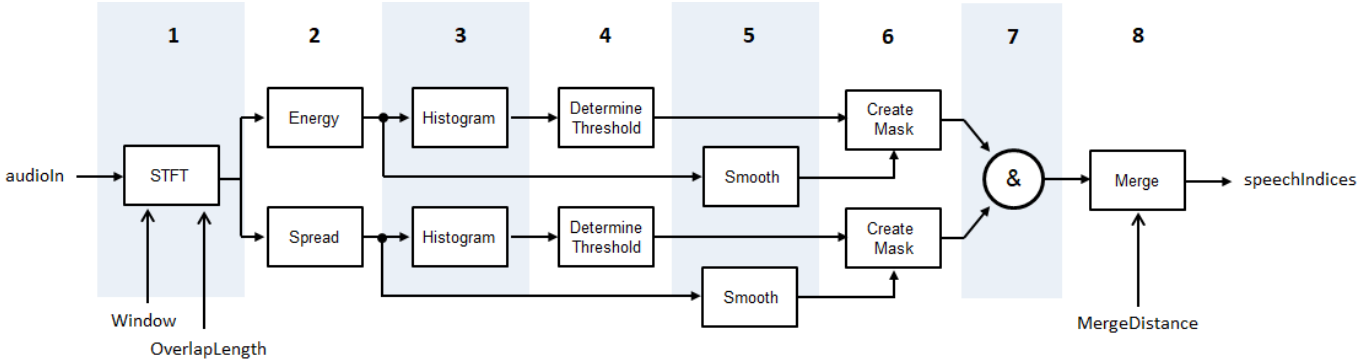
`thresholds` — Thresholds used for decision
two-element vector

Thresholds used for decision, returned as a two-element vector. The thresholds are in the order [*Energy Threshold*, *Spectral Spread Threshold*].

Data Types: `single` | `double`

Algorithms

The `detectSpeech` algorithm is based on [1], although modified so that the statistics to threshold are short-term energy and spectral spread, instead of short-term energy and spectral centroid. The diagram and steps provide a high-level overview of the algorithm. For details, see [1].



1. The audio signal is converted to a time-frequency representation using the specified `Window` and `OverlapLength`.
2. The short-term energy and spectral spread is calculated for each frame. The spectral spread is calculated according to `spectralSpread`.
3. Histograms are created for both the short-term energy and spectral spread distributions.
4. For each histogram, a threshold is determined according to $T = \frac{W \times M_1 + M_2}{W + 1}$, where M_1 and M_2 are the first and second local maxima, respectively. W is set to 5.
5. Both the spectral spread and the short-term energy are smoothed across time by passing through successive five-element moving median filters.
6. Masks are created by comparing the short-term energy and spectral spread with their respective thresholds. To declare a frame as containing speech, a feature must be above its threshold.
7. The masks are combined. For a frame to be declared as speech, both the short-term energy and the spectral spread must be above their respective thresholds.
8. Regions declared as speech are merged if the distance between them is less than `MergeDistance`.

Version History

Introduced in R2020a

References

[1] Giannakopoulos, Theodoros. "A Method for Silence Removal and Segmentation of Speech Signals, Implemented in MATLAB", (University of Athens, Athens, 2009).

Extended Capabilities

C/C++ Code Generation

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

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

See Also

`spectralSpread` | `voiceActivityDetector`

Topics

“Keyword Spotting in Noise Using MFCC and LSTM Networks”

calibrateMicrophone

Calibration factor for microphone

Syntax

```
calibrationFactor = calibrateMicrophone(micRecording,fs,SPLreading)
calibrationFactor = calibrateMicrophone(micRecording,fs,SPLreading,
Name,Value)
```

Description

`calibrationFactor = calibrateMicrophone(micRecording,fs,SPLreading)` returns the calibration factor for the microphone used to create `micRecording`.

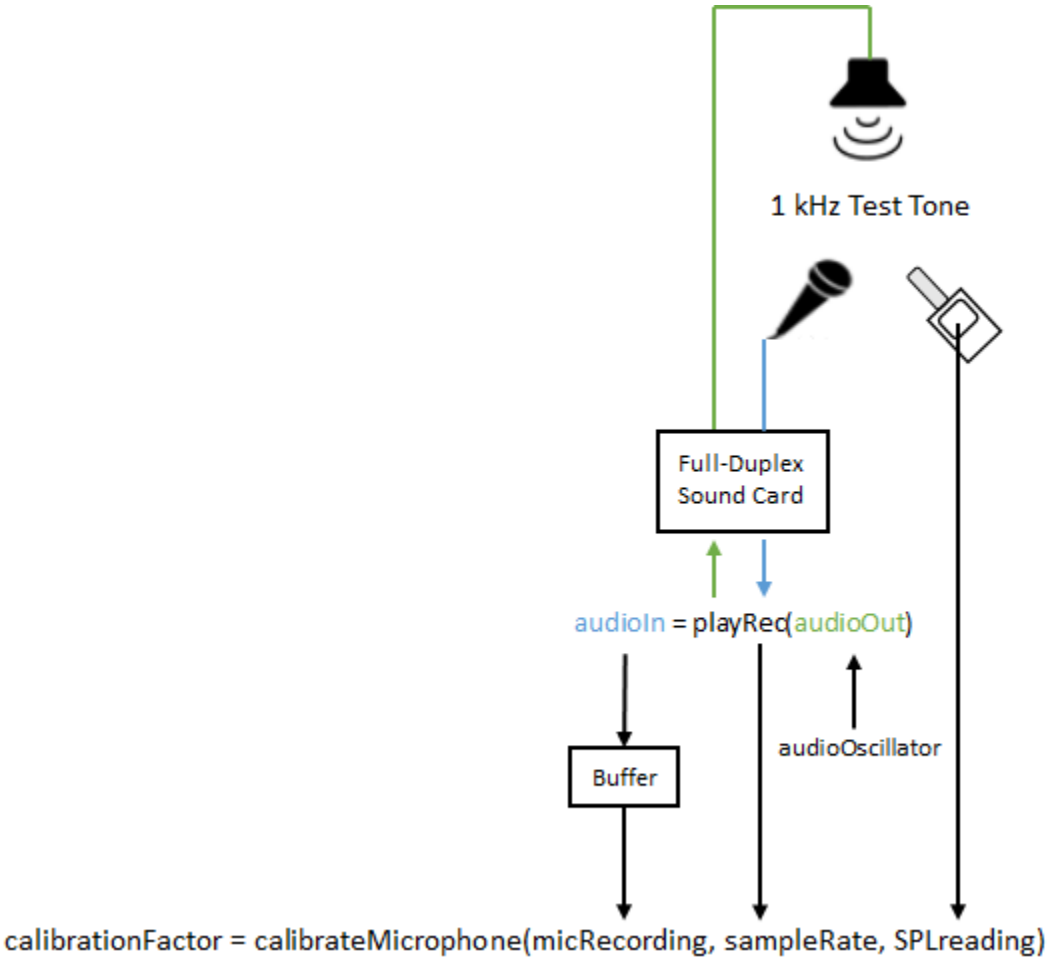
`calibrationFactor = calibrateMicrophone(micRecording,fs,SPLreading, Name,Value)` specifies options using one or more `Name,Value` pair arguments.

Example: `calibrationFactor = calibrateMicrophone(micRecording,fs,SPLreading,'FrequencyWeighting','Z-weighting')` returns the calibration factor for an SPL reading that applies Z-weighting.

Examples

Determine Microphone Calibration Factor

This diagram depicts the setup used in the example:



To run this example, you must connect a microphone and loudspeaker to a full-duplex sound card, and use an SPL meter to determine the true loudness level.

Create an `audioOscillator` object to generate a 1 kHz sine wave at a sample rate of 48 kHz.

```
fs = 48e3;
osc = audioOscillator("sine",1e3,"SampleRate",fs);
```

Create an `audioPlayerRecorder` object to write the sine wave to your loudspeaker and simultaneously read from your microphone.

```
playRec = audioPlayerRecorder(fs);
```

Create a `dsp.AsyncBuffer` object to store the audio recorded from your microphone. Specify the capacity of the buffer to hold 3 seconds worth of data.

```
dur = 3;
buff = dsp.AsyncBuffer(dur*fs);
```

In a loop, for three seconds:

- Generate a frame of a 1 kHz sinusoid.

- Write the frame to your loudspeaker and simultaneously read a frame from your microphone.
- Write the frame acquired from your microphone to the buffer.

While the loop runs, note the true SPL measurement as reported from your SPL meter. Once complete, read the contents of the buffer object.

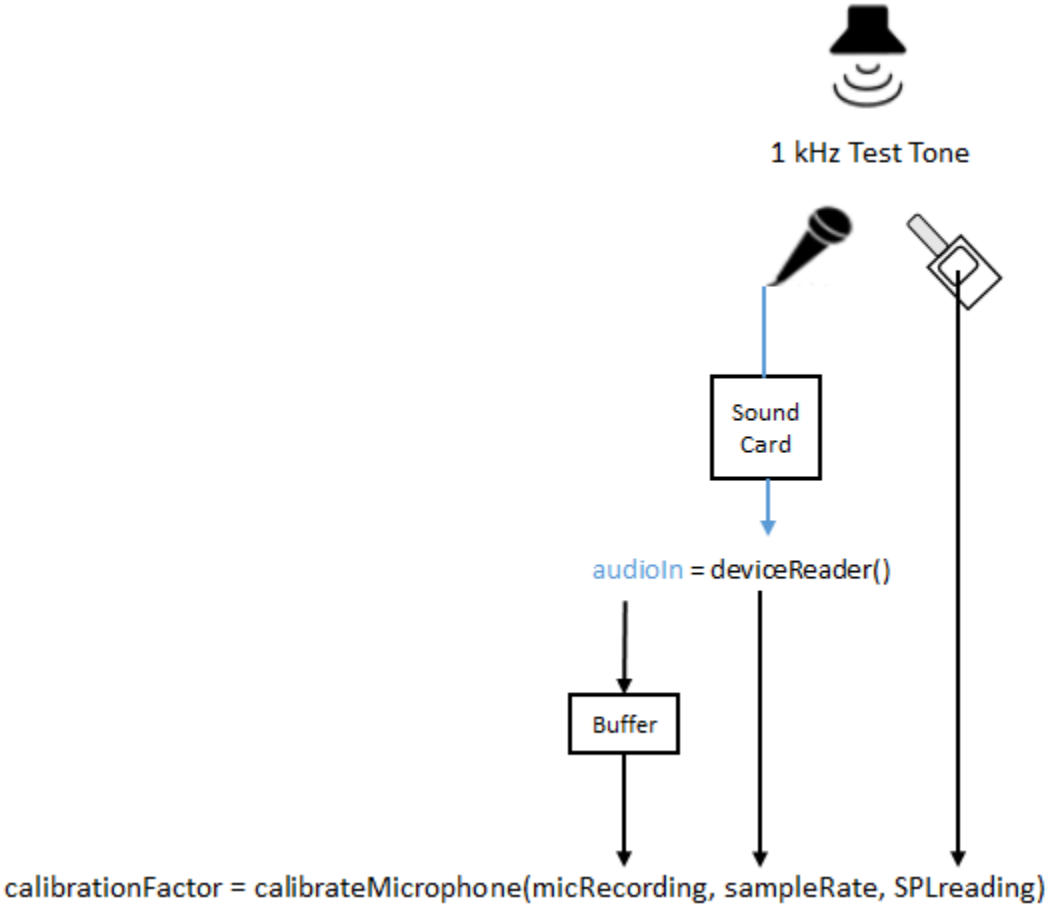
```
numFrames = dur*(fs/osc.SamplesPerFrame);
for ii = 1:numFrames
    audioOut = osc();
    audioIn = playRec(audioOut);
    write(buff,audioIn);
end
release(playRec);

SPL = 78.2; % read from physical SPL meter

micRecording = read(buff);
```

Compute the calibration factor for the microphone.

```
calibrationFactor = calibrateMicrophone(micRecording,playRec.SampleRate,SPL);
```

Calibrate Microphone Using Externally Generated Calibration Tone

The diagram depicts the example setup and data flow.



To run this example, you must connect a microphone to your audio card, generate a 1 kHz tone using an external device, and use an SPL meter to determine the true loudness level.

Specify a 48 kHz sample rate for your audio device and a 3-second duration for acquiring audio. Create an `audioDeviceReader` object to read from your audio device.

```
fs = 48e3;
dur = 3;

deviceReader = audioDeviceReader(fs);
```

Create a `dsp.AsyncBuffer` object to store the streamed audio.

```
buff = dsp.AsyncBuffer(dur*fs);
```

Start the 1 kHz test tone using an external loudspeaker. Then, in a loop, read from your audio device and then write the data to the buffer. While the loop runs, note the true SPL measurement as reported from your SPL meter. Once complete, read the contents of the buffer object.

```
N = deviceReader.SamplesPerFrame;
while buff.NumUnreadSamples+N <= buff.Capacity
    audioIn = deviceReader();
    write(buff,audioIn);
end
```

```
release(deviceReader);

SPL = 77.7; % read from physical SPL meter

micRecording = read(buff);
```

Compute the calibration factor for the microphone.

```
calibrationFactor = calibrateMicrophone(micRecording,deviceReader.SampleRate,SPL);
```

Input Arguments

`micRecording` — Audio signal used to calibrate microphone

column vector | matrix

Audio signal used to calibrate microphone, specified as a column vector (mono) or matrix of independent channels (stereo). `micRecording` must be acquired from the microphone you want to calibrate. The recording should consist of a 1 kHz test tone.

Data Types: `single` | `double`

`fs` — Sample rate of microphone recording (Hz)

positive scalar

Sample rate of microphone recording in Hz, specified as a positive scalar. The recommended sample rate for new recordings is 48 kHz.

Data Types: `single` | `double`

`SPLreading` — Sound pressure level reported from physical meter (dB)

scalar | vector

Sound pressure level reported from meter in dB, specified as a scalar or vector. If `SPLreading` is specified as a vector, it must have the same number of elements as columns in `micRecording`.

Data Types: `single` | `double`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `calibrateMicrophone(micRecording,fs,SPLReading,'PressureReference',22)`

`PressureReference` — Reference pressure (Pa)

`20e-6` (default) | positive scalar

Reference pressure for dB calculation in pascals, specified as a positive scalar. The default reference pressure (20 micropascals) is the common value for air.

Data Types: `single` | `double`

`FrequencyWeighting` — Frequency weighting used by physical meter

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

Frequency weighting used by physical meter, specified as `'A-weighting'`, `'C-weighting'`, or `'Z-weighting'`.

Data Types: `char` | `string`

Output Arguments

`calibrationFactor` — Microphone calibration factor

scalar | row vector

Microphone calibration factor, returned as a scalar or row vector with the same number of elements as `SPLreading`.

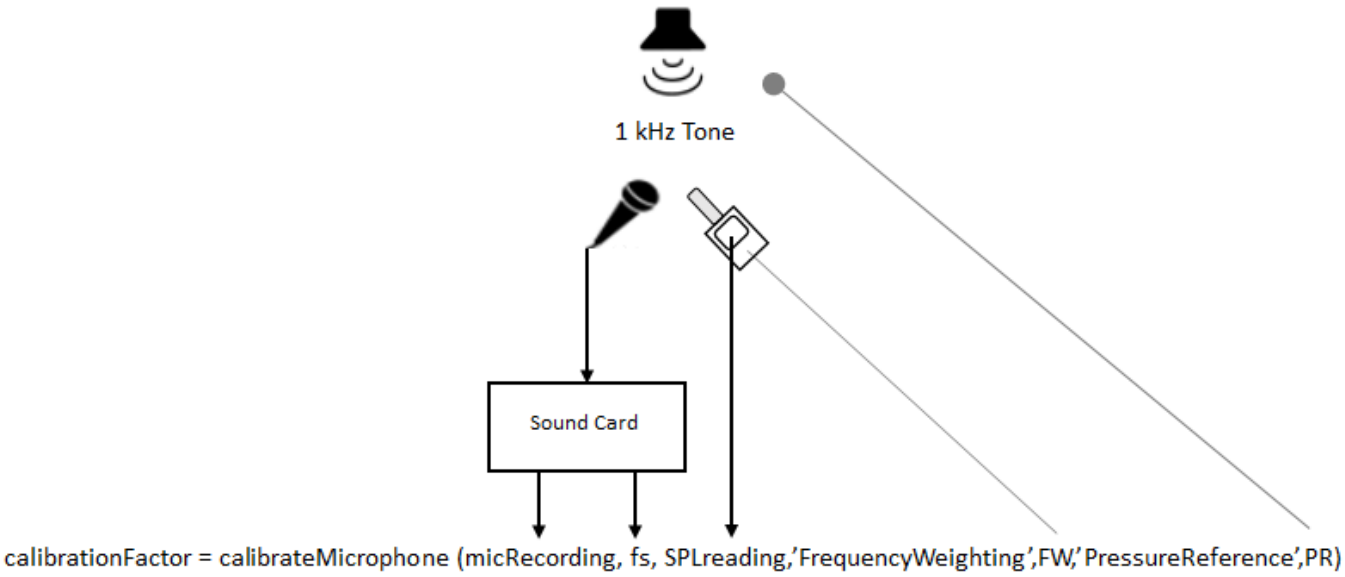
Data Types: `single` | `double`

Algorithms

To determine the calibration factor for a microphone, the `calibrateMicrophone` function uses:

- A calibration tone recorded from the microphone you want to calibrate.
- The sample rate used by your sound card for AD conversion.
- The known loudness, usually determined using a physical SPL meter.
- The frequency weighting used by your physical SPL meter.
- The atmospheric pressure at the recording location.

The diagram indicates a typical physical setup and the locations of required information.



The `calibrationFactor` is set according to the equation:

$$\text{CalibrationFactor} = \frac{10^{((\text{SPLreading}-k)/20)}}{\text{rms(x)}}$$

where `x` is the microphone recording passed through the weighting filter specified in the `FrequencyWeighting` argument. k is 1 pascal relative to the `PressureReference` calculated in dB:

$$k = 20\log_{10}\left(\frac{1}{\text{PressureReference}}\right).$$

Version History

Introduced in R2020a

Extended Capabilities

C/C++ Code Generation

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

See Also

`splMeter` | `acousticLoudness` | `acousticSharpness` | `acousticFluctuation` | `acousticRoughness`

sone2phon

Convert from sone to phon

Syntax

```
phon = sone2phon(sone)
phon = sone2phon(sone,standard)
```

Description

`phon = sone2phon(sone)` converts sone to phon, according to ISO 532-1:2017(E).

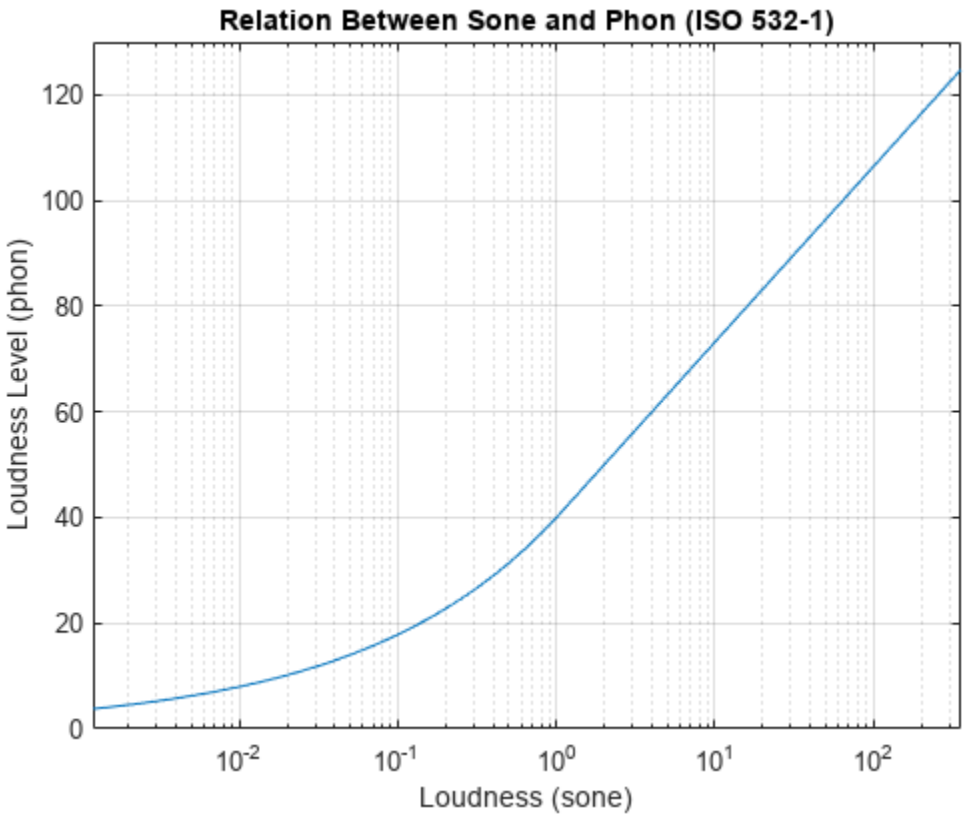
`phon = sone2phon(sone,standard)` specifies the standard used to convert sone to phon.

Examples

Convert Sone to Phon

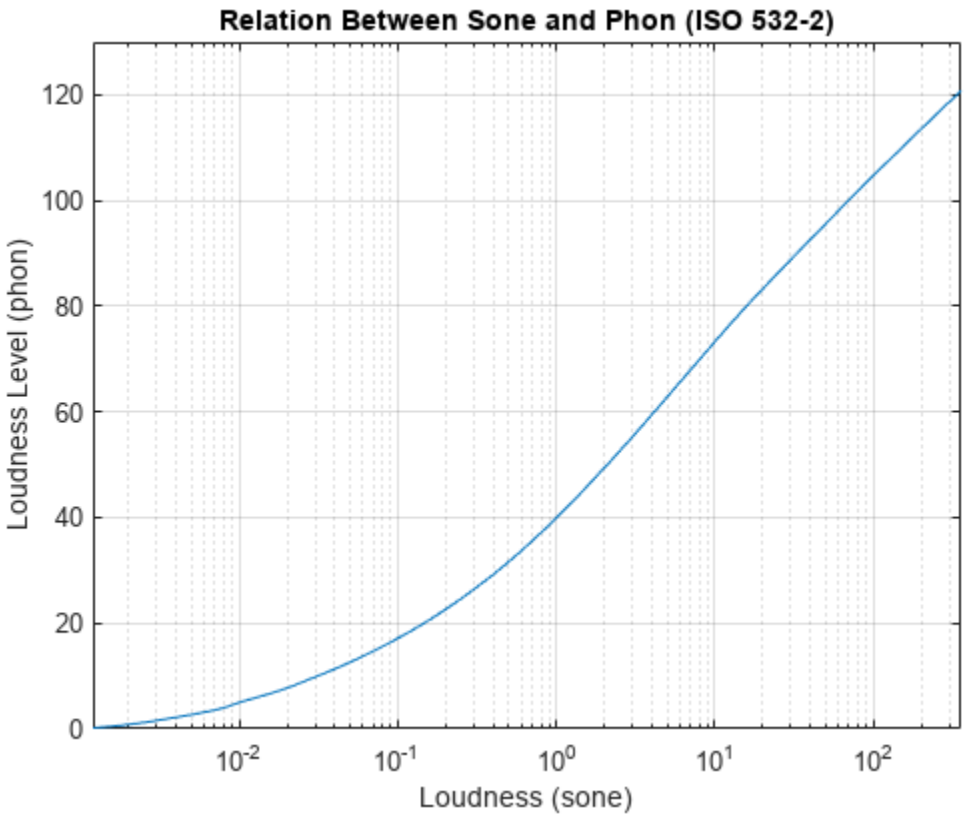
Plot the relationship between loudness (sone) and loudness levels (phon), as specified in ISO 532-1.

```
s = (0.51:0.01:1.8).^10;

p1 = sone2phon(s);

semilogx(s,p1)
xlabel('Loudness (sone)')
ylabel('Loudness Level (phon)')
title('Relation Between Sone and Phon (ISO 532-1)')
grid on
axis([0 s(end) 0 130])
```



Plot the relationship between loudness (sone) and loudness levels (phon), as specified in ISO 532-2.

```
p2 = sone2phon(s,'ISO 532-2');

semilogx(s,p2)
xlabel('Loudness (sone)')
ylabel('Loudness Level (phon)')
title('Relation Between Sone and Phon (ISO 532-2)')
grid on
axis([0 s(end) 0 130])
```



Input Arguments

sone — Input loudness in sone

nonnegative scalar | vector of nonnegative values | matrix of nonnegative values | multidimensional array of nonnegative values

Input loudness in sone, specified as a scalar, vector, matrix, or multidimensional array of nonnegative values.

Data Types: `single` | `double`

`standard` — Reference standard for unit conversion

`'ISO 532-1'` (default) | `'ISO 532-2'`

Reference standard for unit conversion, specified as `'ISO 532-1'` or `'ISO 532-2'`.

Data Types: `char` | `string`

Output Arguments

phon — Output loudness level in phon

scalar | vector | matrix | multidimensional array

Output loudness level in phon, returned as a scalar, vector, matrix, or multidimensional array the same size as `sone`.

Data Types: `single` | `double`

Algorithms

ISO 532-1: Zwicker Method

The Zwicker method of conversion from sone to phon is given by this equation in [1] on page 2-664:

$$phon = \begin{cases} 40(sone)^{0.35} & \text{if } sone < 1 \\ 40 + 10\log_2(sone) & \text{otherwise} \end{cases}$$

ISO 532-2: Moore-Glasberg Method

In the Moore-Glasberg method, conversion from sone to phon is prescribed according to this table (table 5 in [2] on page 2-664).

Loudness Level (phon)	Calculated Loudness (sone)
0.0	0.001
2.2	0.004
4.0	0.008
5.0	0.010
7.5	0.019
10.0	0.031
15.0	0.073
20.0	0.146
25.0	0.26
30.0	0.43
35.0	0.67
40.0	1.00
45.0	1.46
50.0	2.09
55.0	2.96
60.0	4.14
65.0	5.77
70.0	8.04
75.0	11.2
80.0	15.8
85.0	22.7
90.0	32.9
95.0	47.7
100.0	69.6
105.0	102.0

Loudness Level (phon)	Calculated Loudness (sone)
110.0	151.0
115.0	225.0
120.0	337.6

The `sone2phon` function uses interpolation for values not specified in the table.

Version History

Introduced in R2020a

References

[1] ISO 532-1:2017(E). "Acoustics – Methods for calculating loudness – Part 1: Zwicker method." *International Organization for Standardization*.

[2] ISO 532-2:2017(E). "Acoustics – Methods for calculating loudness – Part 2: Moore-Glasberg method." *International Organization for Standardization*.

Extended Capabilities

C/C++ Code Generation

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

See Also

`phon2sone` | `acousticLoudness`

phon2sone

Convert from phon to sone

Syntax

```
sone = phon2sone(phon)
sone = phon2sone(phon,standard)
```

Description

`sone = phon2sone(phon)` converts phon to sone, according to ISO 532-1:2017(E).

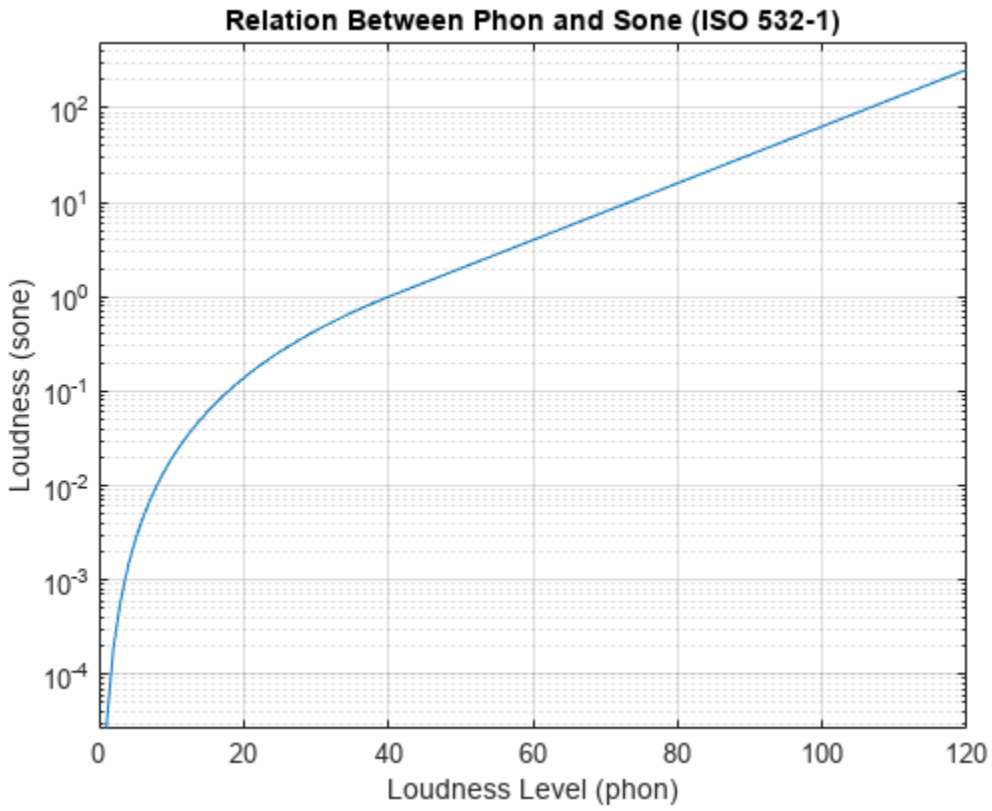
`sone = phon2sone(phon,standard)` specifies the standard used to convert phon to sone.

Examples

Convert Phon to Sone

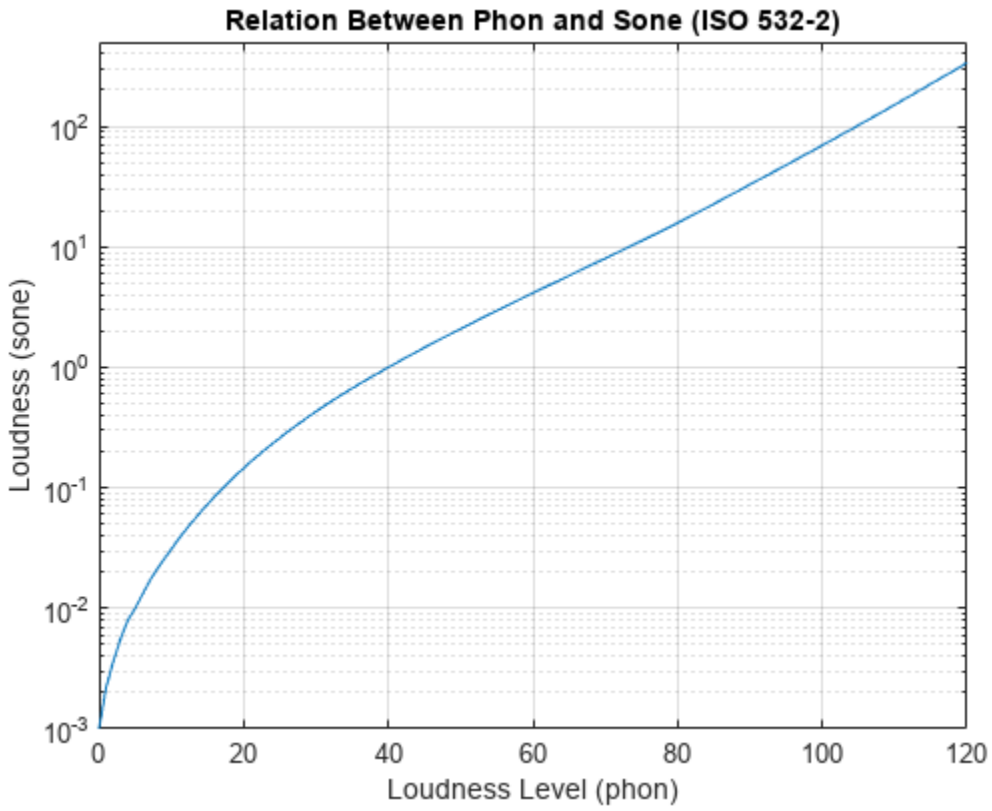
Plot the relationship between loudness level (phon) and loudness (sone), as specified in ISO 532-1.

```
p = 0:120;

s1 = phon2sone(p);

semilogy(p,s1)
xlabel('Loudness Level (phon)')
ylabel('Loudness (sone)')
title('Relation Between Phon and Sone (ISO 532-1)')
grid on
axis([0 120 0 500])
```



Plot the relationship between loudness level (phon) and loudness (sone), as specified in ISO 532-2.

```
s2 = phon2sone(p,'ISO 532-2');

semilogy(p,s2)
xlabel('Loudness Level (phon)')
ylabel('Loudness (sone)')
title('Relation Between Phon and Sone (ISO 532-2)')
grid on
axis([0 120 0 500])
```



Input Arguments

phon — Loudness level in phon

nonnegative scalar | vector of nonnegative values | matrix of nonnegative values | multidimensional array of nonnegative values

Input loudness level in phon, specified as a scalar, vector, matrix, or multidimensional array of nonnegative values.

Data Types: `single` | `double`

`standard` — Reference standard for unit conversion

`'ISO 532-1'` (default) | `'ISO 532-2'`

Reference standard for unit conversion, specified as `'ISO 532-1'` or `'ISO 532-2'`.

Data Types: `char` | `string`

Output Arguments

sone — Output loudness in sone

scalar | vector | matrix | multidimensional array

Output loudness in sone, returned as a scalar, vector, matrix, or multidimensional array the same size as phon.

Data Types: `single` | `double`

Algorithms

ISO 532-1: Zwicker Method

The Zwicker method of conversion from phon to sone is given by [1] on page 2-669:

$$sone = \begin{cases} \left(\frac{phon}{40}\right)^{1/0.35} & \text{if } phon < 1 \\ 2^{\left(\frac{phon - 40}{10}\right)} & \text{otherwise} \end{cases}$$

ISO 532-2: Moore-Glasberg Method

In the Moore-Glasberg method, conversion from phon to sone is prescribed according to this table (table 5 in [2] on page 2-669).

Loudness Level (phon)	Calculated Loudness (sone)
0.0	0.001
2.2	0.004
4.0	0.008
5.0	0.010
7.5	0.019
10.0	0.031
15.0	0.073
20.0	0.146
25.0	0.26
30.0	0.43
35.0	0.67
40.0	1.00
45.0	1.46
50.0	2.09
55.0	2.96
60.0	4.14
65.0	5.77
70.0	8.04
75.0	11.2
80.0	15.8
85.0	22.7
90.0	32.9
95.0	47.7
100.0	69.6

Loudness Level (phon)	Calculated Loudness (sone)
105.0	102.0
110.0	151.0
115.0	225.0
120.0	337.6

The `phon2sone` function uses interpolation for values not specified in the table.

Version History

Introduced in R2020a

References

[1] ISO 532-1:2017(E). "Acoustics – Methods for calculating loudness – Part 1: Zwicker method." *International Organization for Standardization*.

[2] ISO 532-2:2017(E). "Acoustics – Methods for calculating loudness – Part 2: Moore-Glasberg method." *International Organization for Standardization*.

Extended Capabilities

C/C++ Code Generation

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

See Also

`sone2phon` | `acousticLoudness`

generateSimulinkAudioPlugin

Create object class compatible with Simulink

Syntax

```
generateSimulinkAudioPlugin(plugin)
generateSimulinkAudioPlugin(plugin,fileName)
```

Description

`generateSimulinkAudioPlugin(plugin)` generates code for a System object class with the same functionality as the provided audio plugin and opens the generated file. The generated System object is compatible with Simulink® through the MATLAB System block. See Audio Plugin for a block that uses `generateSimulinkAudioPlugin` to include an audio plugin in a Simulink model.

`generateSimulinkAudioPlugin(plugin,fileName)` generates code and saves the resulting System object class to the file specified by `fileName`.

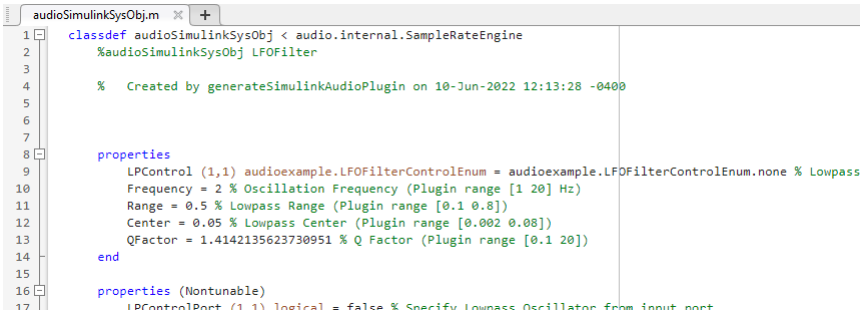
Examples

Generate Audio Plugin System object to Use in Simulink

You can include an audio plugin in your Simulink model by generating a System object with `generateSimulinkAudioPlugin` and then using that System object with the MATLAB System block.

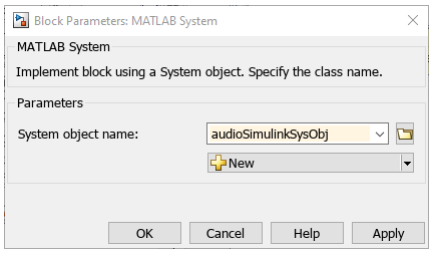
Call `generateSimulinkAudioPlugin` with the `audiopluginexample.LFOFilter` audio plugin to generate a System object class.

```
generateSimulinkAudioPlugin(audiopluginexample.LFOFilter)
```

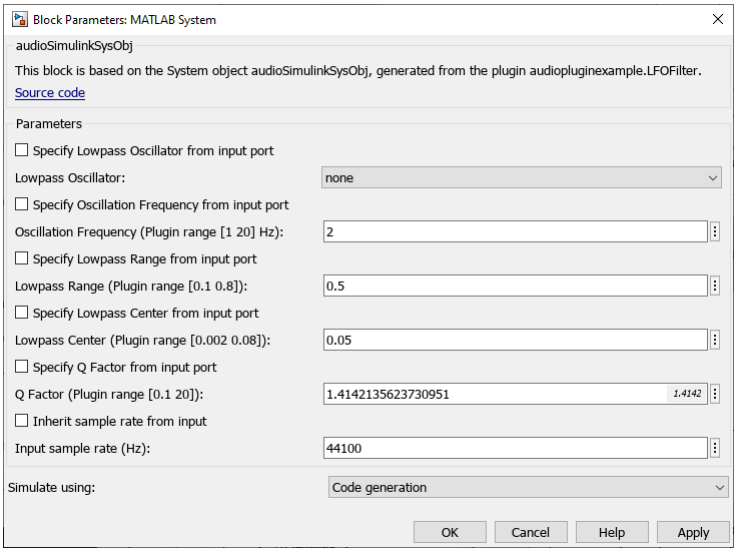


In Simulink, place the MATLAB System block in your model.

Double-click the block to open the dialog box, and specify the **System object name** as the generated class, `audioSimulinkSysObj`. Click **OK** to generate a block with the same functionality as the original plugin.



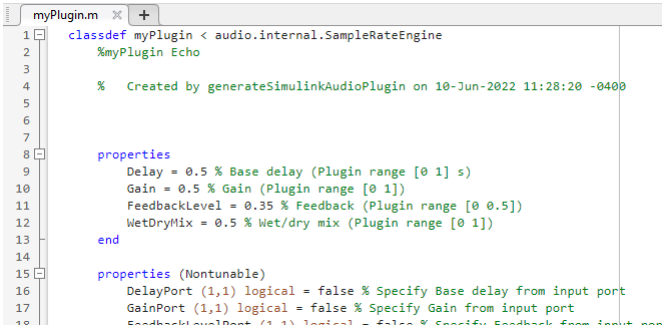
You can now use the block in your model. The block has the same parameters as the original plugin. For more information about the generated block and its parameters, see Audio Plugin.



Specify Generated Class Name

Specify the `fileName` as `"myPlugin"` to generate the System object class with that name in the current directory.

```
generateSimulinkAudioPlugin(audiopluginexample.Echo,"myPlugin")
```



Input Arguments

`plugin` — Audio plugin

audio plugin object

Audio plugin from which to generate the System object class, specified as an audio plugin object. Plugins authored in MATLAB derive from `audioPlugin` or `audioPluginSource`. Externally authored plugins derive from `externalAudioPlugin` or `externalAudioPluginSource` and are returned by `loadAudioPlugin`.

If the input plugin is an externally hosted plugin returned by `loadAudioPlugin`, the function generates additional files required by the System object. For more information, see “Code Generation” on page 2-673.

Example: `audiopluginExample.Echo`

`fileName` — File name of generated class

`"audioSimulinkSysObj"` (default) | string | character vector

File name of the generated System object class, specified as a string or character vector. You can optionally specify the path and `.m` file extension. By default, `generateSimulinkAudioPlugin` creates a class named `audioSimulinkSysObj` in the current directory.

Example: `"myLFO"`

Example: `"plugins/myEchoPlugin.m"`

Data Types: `char` | `string`

Limitations

Some Simulink functionality, such as **Step Back**, requires saving and restoring the simulation state. Blocks that use hosted external plugins do not support simulation save and restore and therefore do not support associated functionality. For tips on using simulation save and restore functionality with blocks that use plugins authored in MATLAB, see “Tips” on page 2-672.

Tips

To use Simulink functionality that requires saving and restoring the simulation state, such as **Step Back**, with a block that uses a plugin authored in MATLAB, the original plugin implementation must correctly save and load its state.

- If the original plugin is a System object, it must correctly save and load its state using the `saveObjectImpl` and `loadObjectImpl` methods.
- If the original plugin is an `audioPlugin` and not a System object plugin, it must correctly save and load its state using the `saveobj` and `loadobj` methods.

Note If the original plugin does not maintain any state, no additional considerations are necessary for the save and restore functionality.

Version History

Introduced in R2022b

Extended Capabilities

C/C++ Code Generation

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

Generating code from blocks that use external audio plugins has additional requirements. External audio plugins include plugins loaded into MATLAB with `loadAudioPlugin`.

- The `generateSimulinkAudioPlugin` function generates files in addition to the System object file to aid in code generation. These files include *`sysObjName`*`PluginLoader.m` and *`sysObjName`*`Interface.m` where *`sysObjName`* is the name of the generated System object. The function also generates *`sysObjName`*`Tables.mat` if the plugin has any parameters. These additional files are required for both code generation and running the block in simulation.
- You must select the **Support long long** parameter in the **Hardware Implementation** pane of the **Model Settings**.
- If you are using an ERT target, you must set the **Language** parameter to `C++` under the **Target selection** section of the **Code Generation** pane in the **Model Settings**. You must also select the **Dynamic memory allocation in MATLAB functions** parameter in the **Advanced parameters** section of the **Simulation Target** pane.
- To use a standalone executable generated from a block with an external plugin, you must generate the `jucehost.dll` file on Windows or the `libjucehost.dylib` file on Mac by selecting the **Package code and artifacts** parameter under the **Build process** section of the **Code Generation** pane in the **Model Settings**.
  - On Windows platforms, you must make the `jucehost.dll` file visible to the standalone executable. To do this, add the path to the `jucehost.dll` file to the `PATH` environment variable or copy the `jucehost.dll` file to the same folder as the standalone executable .
  - On Mac platforms, you must make the `libjucehost.dylib` file visible to the standalone executable. To do this, place the `libjucehost.dylib` file in the `/usr/lib` directory.

Hosted AUv3 plugins do not support code generation.

See Also

Audio Plugin | `audioPlugin` | `audioPluginSource` | `loadAudioPlugin`

speechClient

Interface with pretrained model or third-party speech service

Description

Use a `speechClient` object to interface with a wav2vec 2.0 pretrained speech-to-text model or third-party cloud-based speech services. Use the object with `speech2text` or `text2speech`.

Note To use `speechClient` to interface with third-party speech services, you must download the extended Audio Toolbox functionality from File Exchange. The File Exchange submission includes a tutorial to get started with the third-party services.

Using wav2vec 2.0 requires Deep Learning Toolbox and installing the pretrained model.

Creation

Syntax

```
clientObj = speechClient(name)
clientObj = speechClient( ___ ,Name=Value)
```

Description

`clientObj = speechClient(name)` returns a `speechClient` object that interfaces with the specified pretrained model or speech service.

`clientObj = speechClient( ___ ,Name=Value)` sets “Properties” on page 2-675 using one or more name-value arguments.

Input Arguments

name — Pretrained model or service name

`"wav2vec2.0"` | `"Google"` | `"IBM"` | `"Microsoft"` | `"Amazon"`

Name of the pretrained model or speech service, specified as `"wav2vec2.0"`, `"Google"`, `"IBM"`, `"Microsoft"`, or `"Amazon"`.

- `"wav2vec2.0"` –– Use a pretrained wav2vec 2.0 model. You can only use wav2vec 2.0 to perform speech-to-text transcription, and therefore you cannot use it with `text2speech`.
- `"Google"` –– Interface with the Google Cloud Speech-to-Text and Text-to-Speech service.
- `"IBM"` –– Interface with the IBM® Watson Speech to Text and Text to Speech service.
- `"Microsoft"` –– Interface with the Microsoft® Azure® Speech service.
- `"Amazon"` –– Interface with the Amazon Transcribe and Amazon Polly services.

Using the wav2vec 2.0 pretrained model requires Deep Learning Toolbox and installing the pretrained wav2vec 2.0 model. If the model is not installed, calling `speechClient` with `"wav2vec2.0"` provides a link to download and install the model.

To use any of the third-party speech services (Google, IBM, Microsoft, or Amazon), you must download the extended Audio Toolbox functionality from File Exchange. The File Exchange submission includes a tutorial to get started with the third-party services.

Data Types: `string` | `char`

Output Arguments

`clientObj` — Client object
`speechClient` object

Client object to use with `speech2text` to transcribe speech in audio signals to text, or with `text2speech` to synthesize speech signals from text.

Properties

`Segmentation` — Segmentation of transcript
`"word"` | `"sentence"` | `"none"`

Segmentation of the output transcript, specified as `"word"`, `"sentence"`, or `"none"`.

This property applies only to the wav2vec 2.0 pretrained model and the Amazon speech service.

- `"word"` — `speech2text` returns the transcript as a table where each word is in its own row. This is the default for the wav2vec 2.0 pretrained model.
- `"sentence"` — `speech2text` returns the transcript as a table where each sentence is in its own row. The wav2vec 2.0 pretrained model does not support this option.
- `"none"` — `speech2text` returns a string containing the entire transcript. This is the default for the Amazon speech service.

Data Types: `string` | `char`

`TimeStamps` — Include timestamps in transcript
`false` (default) | `true`

Include timestamps of transcribed speech in the transcript, specified as `true` or `false`. If you specify `TimeStamps` as `true`, `speech2text` includes an additional column in the transcript table that contains the timestamps. When using the wav2vec 2.0 pretrained model, the `speech2text` function determines the timestamps using the algorithm described in [2].

This property applies only if you set the Segmentation property to `"word"` or `"sentence"`.

Data Types: `logical`

`TimeOut` — Connection timeout
nonnegative scalar

Connection timeout, specified as a nonnegative scalar in seconds. The timeout specifies the time to wait for the initial server connection to the third-party speech service.

This property applies only to the third-party speech services.

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

Object Functions

Note For the third-party speech services, you can configure server-specific options using the following functions. See the documentation for the specific service for option names and values.

setOptions	Set server options
getOptions	Get server options
clearOptions	Remove all server options

Examples

Download wav2vec 2.0 Network

Download and install the pretrained wav2vec 2.0 model for speech-to-text transcription.

Type `speechClient("wav2vec2.0")` into the command line. If the pretrained model for wav2vec 2.0 is not installed, the function provides a download link. To install the model, click the link to download the file and unzip it to a location on the MATLAB path.

Alternatively, execute the following commands to download the wav2vec 2.0 model, unzip it to your temporary directory, and then add it to your MATLAB path.

```
downloadFile = matlab.internal.examples.downloadSupportFile("audio","wav2vec2/wav2vec2-base-960.z
wav2vecLocation = fullfile(tempdir,"wav2vec");
unzip(downloadFile,wav2vecLocation)
addpath(wav2vecLocation)
```

Check that the installation is successful by typing `speechClient("wav2vec2.0")` into the command line. If the model is installed, then the function returns a `Wav2VecSpeechClient` object.

```
speechClient("wav2vec2.0")
```

```
ans =
  Wav2VecSpeechClient with properties:

    Segmentation: 'word'
      TimeStamps: 0
```

Perform Speech-to-Text Transcription

Read in an audio file containing speech and listen to it.

```
[y,fs] = audioread("speech_dft.wav");
soundsc(y,fs)
```

Create a `speechClient` object that uses the wav2vec 2.0 pretrained network. This requires installing the pretrained network. If the network is not installed, the function provides a link with instructions to download and install the pretrained model.

```
transcriber = speechClient("wav2vec2.0");
```

Use `speech2text` to obtain a transcription of the audio signal.

```
transcript = speech2text(transcriber,y,fs)
```

```
transcript=12×2 table
    Transcript     Confidence
    ___________    __________

    "the"           0.97605
    "discreet"      0.91927
    "fourier"       0.84546
    "transform"     0.89922
    "of"            0.66676
    "a"             0.50026
    "real"          0.88796
    "valued"        0.89913
    "signal"         0.8041
    "is"            0.53891
    "conjugate"     0.98438
    "symmetric"     0.89333
```

Version History

Introduced in R2022b

References

[1] Baevski, Alexei, Henry Zhou, Abdelrahman Mohamed, and Michael Auli. “Wav2vec 2.0: A Framework for Self-Supervised Learning of Speech Representations,” 2020. https://doi.org/10.48550/ARXIV.2006.11477.

[2] Kürzinger, Ludwig, Dominik Winkelbauer, Lujun Li, Tobias Watzel, and Gerhard Rigoll. “CTC-Segmentation of Large Corpora for German End-to-End Speech Recognition.” In *Speech and Computer*, edited by Alexey Karpov and Rodmonga Potapova, 12335:267–78. Cham: Springer International Publishing, 2020. https://doi.org/10.1007/978-3-030-60276-5_27.

See Also

`speech2text` | `text2speech` | **Signal Labeler**

text2speech

Synthesize speech from text

Syntax

```
[speech,fs] = text2speech(clientObj,text)
[speech,fs] = text2speech( ___ ,HTTPTimeout=timeout)
[speech,fs,rawOutput] = text2speech( ___ )
```

Description

`[speech,fs] = text2speech(clientObj,text)` synthesizes a speech signal from the provided text. `text2speech` interfaces with third-party speech services (Google, IBM, Microsoft, or Amazon) to perform the synthesis.

Note To use `text2speech`, you must download the extended Audio Toolbox functionality from File Exchange. The File Exchange submission includes a tutorial to get started with the third-party services.

`[speech,fs] = text2speech( ___ ,HTTPTimeout=timeout)` specifies the time in seconds to wait for the initial server connection to the third-party speech service.

`[speech,fs,rawOutput] = text2speech( ___ )` also returns the unprocessed server output from the third-party speech service.

Examples

Synthesize Speech from Text

Create a `speechClient` object that interfaces with the IBM Watson Text to Speech service.

```
synthesizer = speechClient("IBM");
```

Call `text2speech` with a string to synthesize a speech signal.

```
[speech,fs] = text2speech(synthesizer,"hello world");
```

Listen to the synthesized speech.

```
soundsc(speech,fs)
```

Input Arguments

`clientObj` — Client object
`speechClient` object

Client object, specified as an object returned by `speechClient`. The object is an interface to a third-party speech service.

You cannot use `text2speech` with a `speechClient` object that interfaces with the wav2vec 2.0 pretrained model.

To use the third-party speech services, you must download the extended Audio Toolbox functionality from File Exchange. The File Exchange submission includes a tutorial to get started with the third-party services.

Example: `speechClient("IBM")`

`text` — Text
string | character array

Text to synthesize into speech, specified as a string or character array.

Example: "Hello world"

Data Types: `char` | `string`

`timeout` — Time to wait for server connection in seconds
positive scalar

Time to wait for initial server connection in seconds, specified as a positive scalar. This sets the `TimeOut` property of `clientObj`.

Output Arguments

`speech` — Synthesized speech
column vector

Synthesized speech signal, returned as a column vector (single channel).

Data Types: `double`

`fs` — Sample rate (Hz)
positive double

Sample rate of speech signal in Hz, returned as a positive double. The sample rate depends on the third-party service and the server options set through the `clientObj`. See the documentation for the specific speech service for more information.

Data Types: `double`

`rawOutput` — Unprocessed server output
`ResponseMessage` | structure

Unprocessed server output, returned as a `matlab.net.http.ResponseMessage` object containing the HTTP response from the third-party speech service. If the third-party speech service is Amazon, `text2speech` returns the server output as a structure.

Version History

Introduced in R2022b

See Also

`speechClient` | `speech2text`

detectspeechnn

Detect boundaries of speech in audio signal using AI

Syntax

```
roi = detectspeechnn(audioIn,fs)
roi = detectspeechnn(audioIn,fs,Name=Value)
detectspeechnn( ___ )
```

Description

`roi = detectspeechnn(audioIn,fs)` returns indices corresponding to the beginning and end of speech within the audio signal.

`roi = detectspeechnn(audioIn,fs,Name=Value)` specifies options using one or more name-value arguments. For example, `detectspeechnn(audioIn,fs,MergeThreshold=0.5)` merges speech regions that are separated by 0.5 seconds or less.

`detectspeechnn( ___ )` with no output arguments plots the input signal and the detected speech regions.

This function requires both Audio Toolbox and Deep Learning Toolbox.

Examples

Detect Speech in Audio Signal

Read in an audio signal containing speech and music and listen to the sound.

```
[audioIn,fs] = audioread("MusicAndSpeech-16-mono-14secs.ogg");
sound(audioIn,fs)
```

Call `detectspeechnn` on the signal to obtain the regions of interest (ROIs), in samples, containing speech.

```
roi = detectspeechnn(audioIn,fs)
```

```
roi = 2×2

           1       63120
       83600      150000
```

Convert the ROIs from samples to seconds.

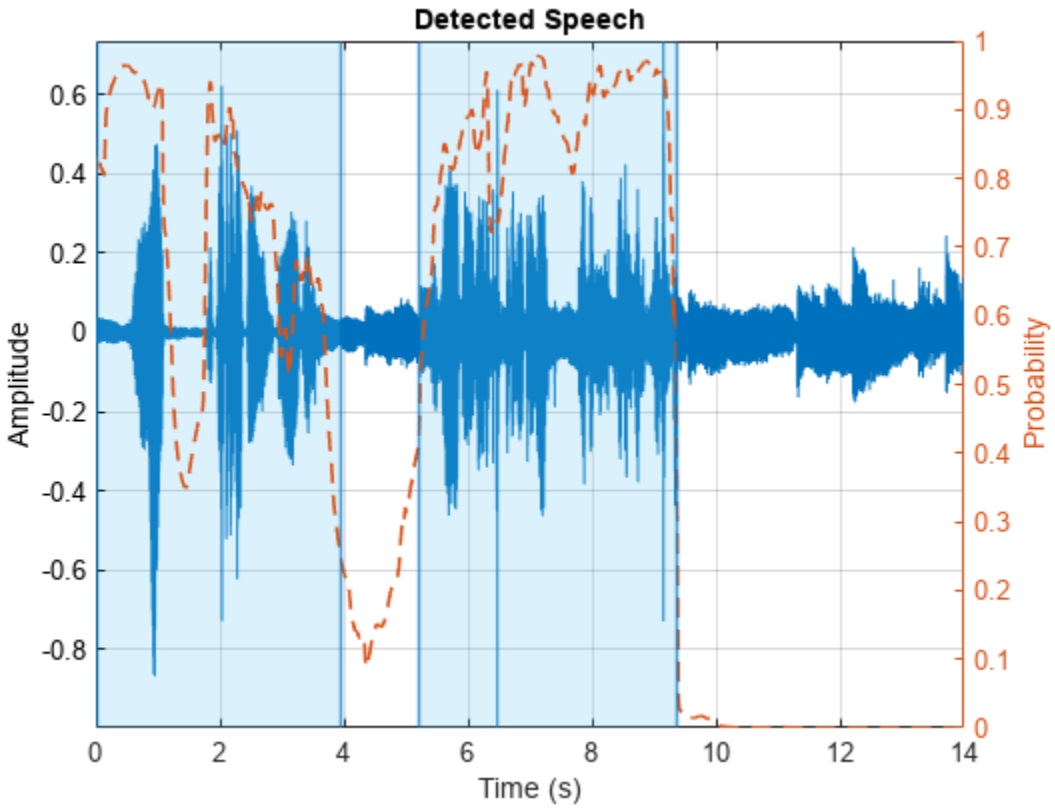
```
roiSeconds = (roi-1)/fs
```

```
roiSeconds = 2×2

         0    3.9449
```

```
    5.2249    9.3749
```

Plot the audio waveform with the speech regions.

```
detectspeechnn(audioIn,fs)
```



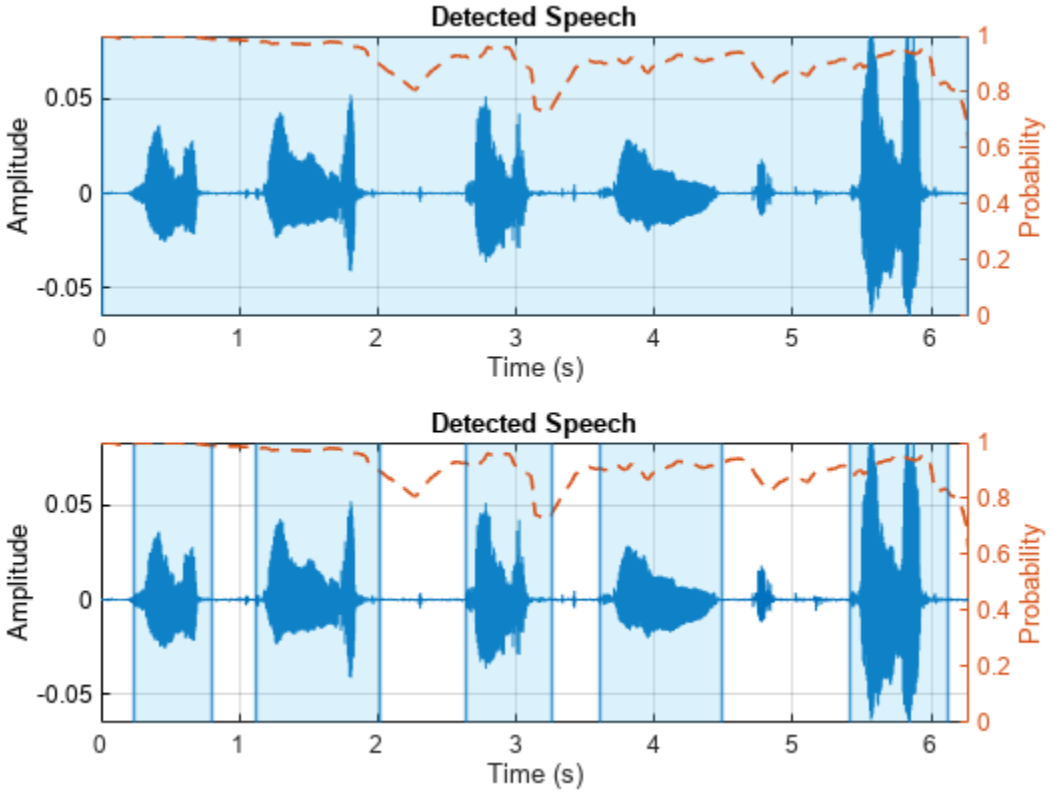
Refine Speech Regions with Energy-Based VAD

Read in an audio signal containing a speaker repeating the phrase "volume up".

```
[audioIn,fs] = audioread("MaleVolumeUp-16-mono-6secs.ogg");
```

Compare detected speech regions by calling `detectspeechnn` with and without the application of an energy-based voice activity detector (VAD) in postprocessing.

```
tiledlayout(2,1)
nexttile()
detectspeechnn(audioIn,fs)
nexttile()
detectspeechnn(audioIn,fs,ApplyEnergyVAD=true)
```



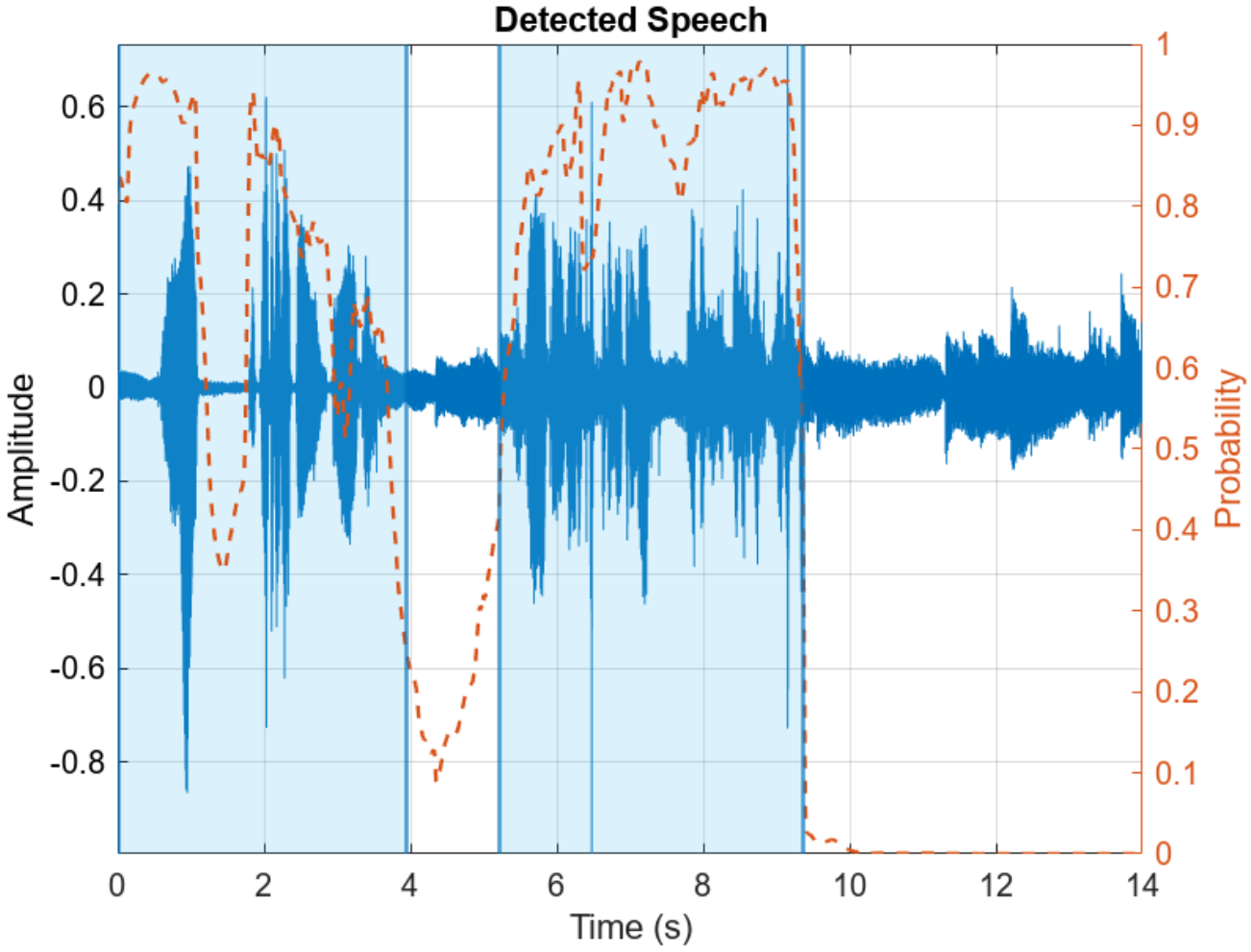
Adjust Postprocessing Parameters for Detecting Speech

Read in an audio signal.

```
[audioIn,fs] = audioread("MusicAndSpeech-16-mono-14secs.ogg");
```

Call `detectspeechnn` with no output arguments to display a plot of the detected speech regions.

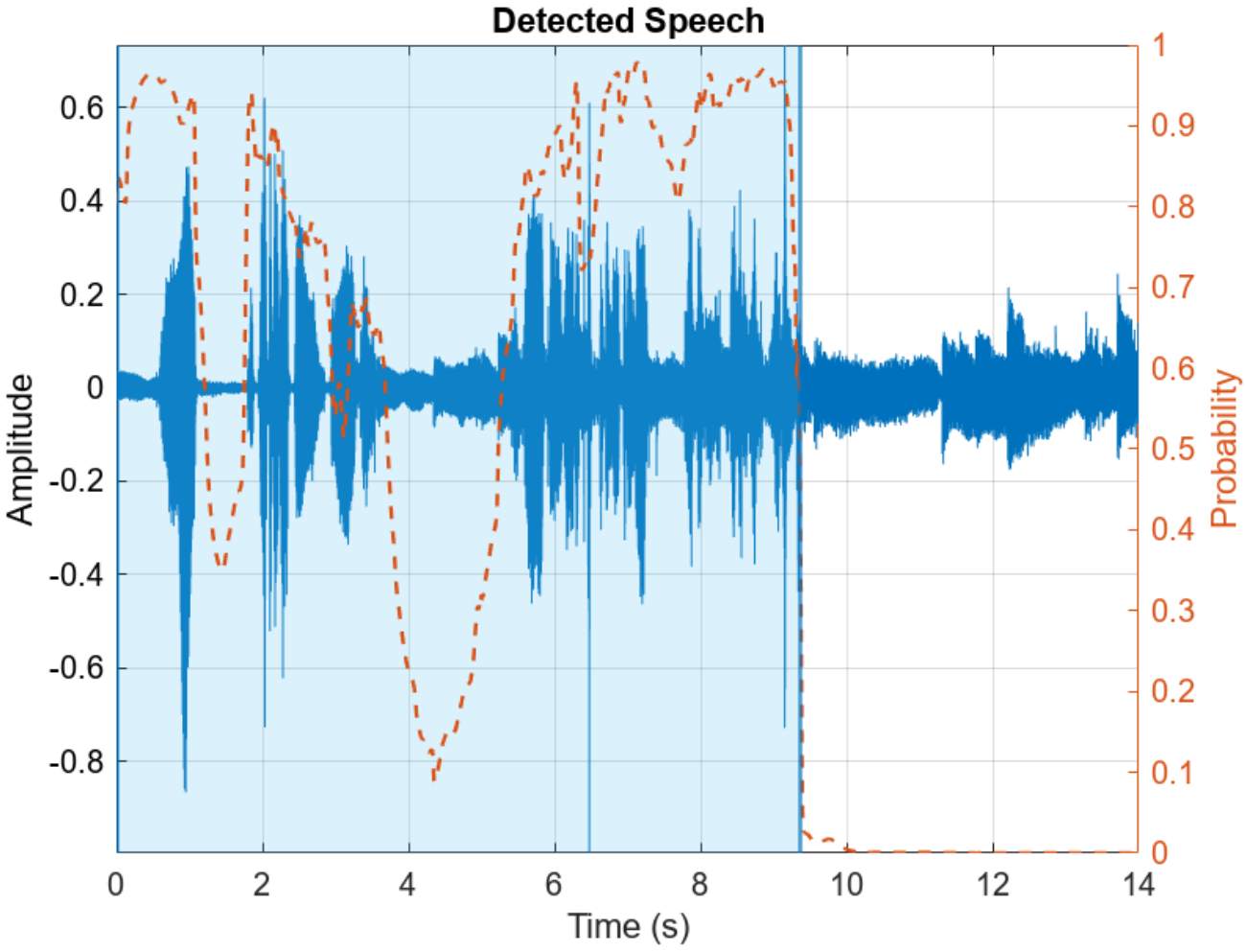
```
detectspeechnn(audioIn,fs);
```



Modify the parameters used in the postprocessing algorithm and see how they affect the detected speech regions. For more information about the VAD postprocessing algorithm, see “Postprocessing” on page 2-688.

```
mergeThreshold = 1.3  ; % seconds

lengthThreshold = 0.25  ; % seconds

activationThreshold = 0.5  ; % probability

deactivationThreshold = 0.25  ; % probability

applyEnergyVAD =  ;

detectspeechnn(audioIn,fs,MergeThreshold=mergeThreshold, ...
    LengthThreshold=lengthThreshold, ...
```

```
ActivationThreshold=activationThreshold, ...
DeactivationThreshold=deactivationThreshold)
```



Detect Speech in Streaming Audio

Use `detectspeechnn` to detect the presence of speech in a streaming audio signal.

Create a `dsp.AudioFileReader` object to stream an audio file for processing. Set the `SamplesPerFrame` property to read 100 ms nonoverlapping chunks from the signal.

```
afr = dsp.AudioFileReader("MaleVolumeUp-16-mono-6secs.ogg");
analysisDuration = 0.1; % seconds
afr.SamplesPerFrame = floor(analysisDuration*afr.SampleRate);
```

The neural network architecture of `detectspechnn` does not retain state between calls, and it performs best when analyzing larger chunks of audio signals. When you use `detectspeechnn` in a streaming scenario, specific application requirements of accuracy, computational efficiency, and latency dictate the analysis duration and whether to overlap analysis chunks.

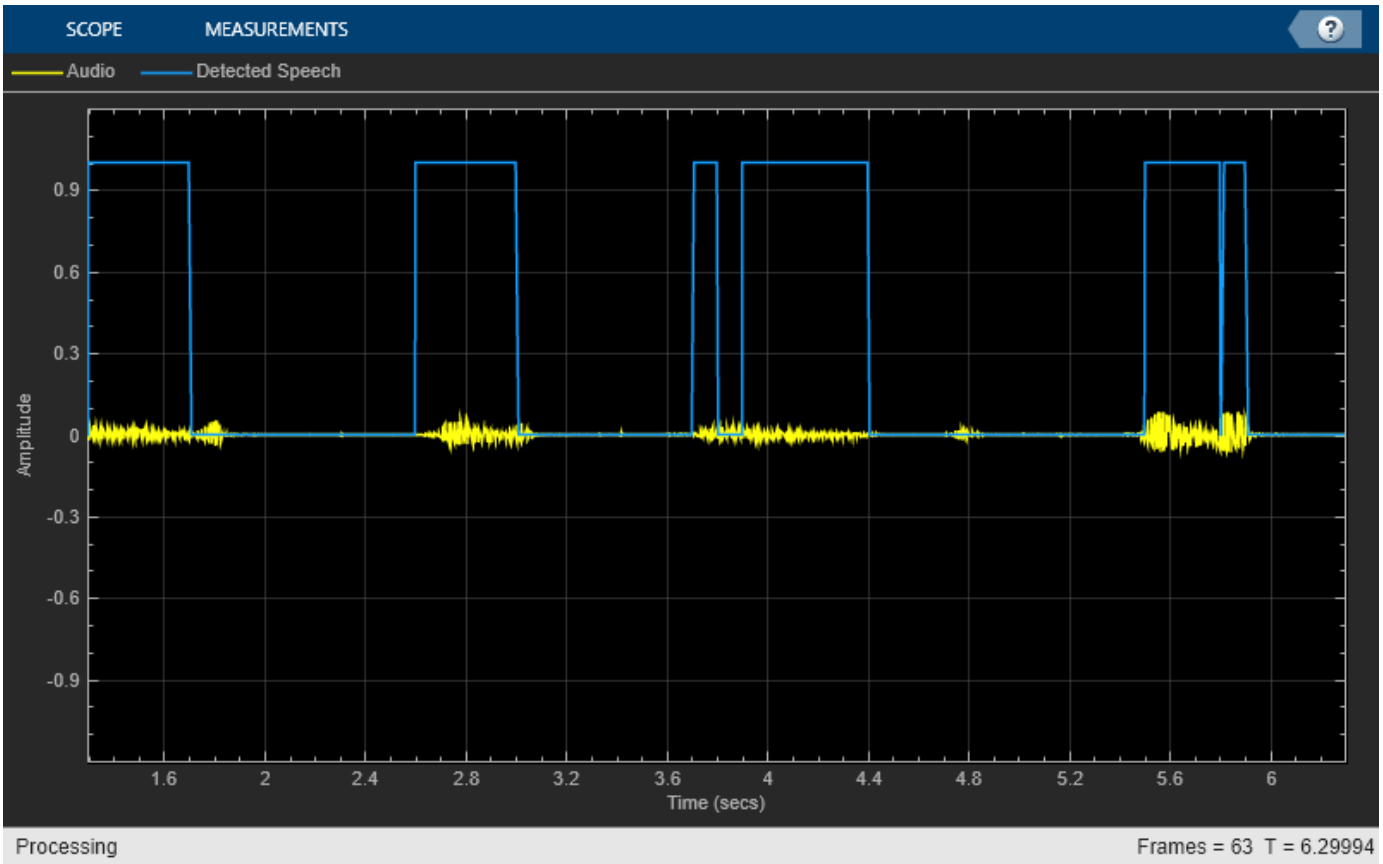
Create a `timescope` object to plot the audio signal and the detected speech regions. Create an `audioDeviceWriter` to play the audio as you stream it.

```
scope = timescope(NumInputPorts=2, ...
    SampleRate=afr.SampleRate, ...
    TimeSpanSource="property",TimeSpan=5, ...
    YLimits=[-1.2,1.2], ...
    ShowLegend=true,ChannelNames=["Audio","Detected Speech"]);
adw = audioDeviceWriter(afr.SampleRate);
```

In a streaming loop:

1. Read in a 100 ms chunk from the audio file.
2. Use `detectspeechnn` to detect any regions of speech in the frame. Use `sigroi2binmask` to convert the region indices to a binary mask.
3. Plot the audio signal and the detected speech.
4. Play the audio with the device writer.

```
while ~isDone(afr)
    audioIn = afr();
    segments = detectspeechnn(audioIn,afr.SampleRate,LengthThreshold=0.01);
    mask = sigroi2binmask(segments,afr.SamplesPerFrame);
    scope(audioIn,mask)
    adw(audioIn);
end
```



Input Arguments

`audioIn` — Audio input

column vector

Audio input signal, specified as a column vector (single channel).

Data Types: `single` | `double`

`fs` — Sample rate (Hz)

positive scalar

Sample rate in Hz, specified as a positive scalar.

Data Types: `single` | `double`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `detectspeechnn(audioIn,fs,ApplyEnergyVAD=true)`

`MergeThreshold` — Merge threshold

`0.25` (default) | nonnegative scalar

Merge threshold in seconds, specified as a nonnegative scalar. The function merges speech regions that are separated by a duration less than or equal to the specified threshold. Set the threshold to `Inf` to not merge any detected regions.

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

`LengthThreshold` — Length threshold

`0.25` (default) | nonnegative scalar

Length threshold in seconds, specified as a nonnegative scalar. The function does not return speech regions that have a duration less than or equal to the specified threshold.

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

`ActivationThreshold` — Probability threshold to start a speech segment

`0.5` (default) | scalar in the range [0, 1]

Probability threshold to start a speech segment, specified as a scalar in the range [0, 1].

Data Types: `single` | `double`

`DeactivationThreshold` — Probability threshold to end a speech segment

`0.25` (default) | scalar in the range [0, 1]

Probability threshold to end a speech segment, specified as a scalar in the range [0, 1].

Data Types: `single` | `double`

`ApplyEnergyVAD` — Apply energy-based voice activity detector

`false` (default) | `true`

Apply energy-based voice activity detector (VAD) to the speech regions detected by the neural network, specified as `true` or `false`.

Data Types: `logical`

Output Arguments

`roi` — Speech regions

N-by-2 matrix

Speech regions, returned as an *N*-by-2 matrix of indices into the input signal, where *N* is the number of individual speech regions detected. The first column contains the index of the start of a speech region, and the second column contains the index of the end of a region.

Algorithms

Preprocessing

The `detectspeechnn` function preprocesses the audio data using the following steps.

1. Resample the audio to 16kHz.
2. Compute a centered short-time Fourier transform (STFT) using a 25 ms periodic Hamming window and 10 ms hop length. Pad the signal so that the first window is centered at 0 s.
3. Convert the STFT to a power spectrogram.
4. Apply a mel filter bank with 40 bands to obtain a mel spectrogram.
5. Convert the mel spectrogram to a log scale.
6. Standardize each of the mel bands to have zero mean and standard deviation of 1.

Neural Network Inference

The preprocessed data is passed to a pretrained VAD neural network. The network outputs represent the probability of speech in each frame of audio in the input spectrogram.

The neural network is a ported version of the `vad-crdnn-libriparty` pretrained model provided by SpeechBrain[1], which combines convolutional, recurrent, and fully connected layers.

Postprocessing

The `detectspeechnn` function postprocesses the VAD network output using the following steps.

1. Apply activation and deactivation thresholds to posterior probabilities to determine candidate speech regions.
2. Optionally, apply energy-based VAD to refine the detected speech regions.
3. Merge speech regions that are close to each other according to the merge threshold.
4. Remove speech regions that are shorter than or equal to the length threshold.

Version History

Introduced in R2023a

References

[1] Ravanelli, Mirco, et al. *SpeechBrain: A General-Purpose Speech Toolkit*. arXiv, 8 June 2021. *arXiv.org*, http://arxiv.org/abs/2106.04624

See Also

Functions
`vadnet` | `vadnetPreprocess` | `vadnetPostprocess` | `detectSpeech`

Objects
`voiceActivityDetector`

Blocks
Voice Activity Detector

Topics
“Voice Activity Detection in Noise Using Deep Learning”
“Train Voice Activity Detection in Noise Model Using Deep Learning”

vadnet

Voice activity detection (VAD) neural network

Syntax

```
net = vadnet()
```

Description

`net = vadnet()` returns a pretrained VAD model.

This function requires both Audio Toolbox and Deep Learning Toolbox.

Examples

Detect Speech with Pretrained VAD Model

Read in an audio signal containing speech and music and listen to the sound.

```
[audioIn,fs] = audioread("MusicAndSpeech-16-mono-14secs.ogg");
sound(audioIn,fs)
```

Use `vadnetPreprocess` to preprocess the audio by computing a mel spectrogram.

```
features = vadnetPreprocess(audioIn,fs);
```

Call `vadnet` to obtain a pretrained VAD neural network.

```
net = vadnet;
```

Pass the preprocessed audio through the network to obtain the probability of speech in each frame.

```
probs = predict(net,features);
```

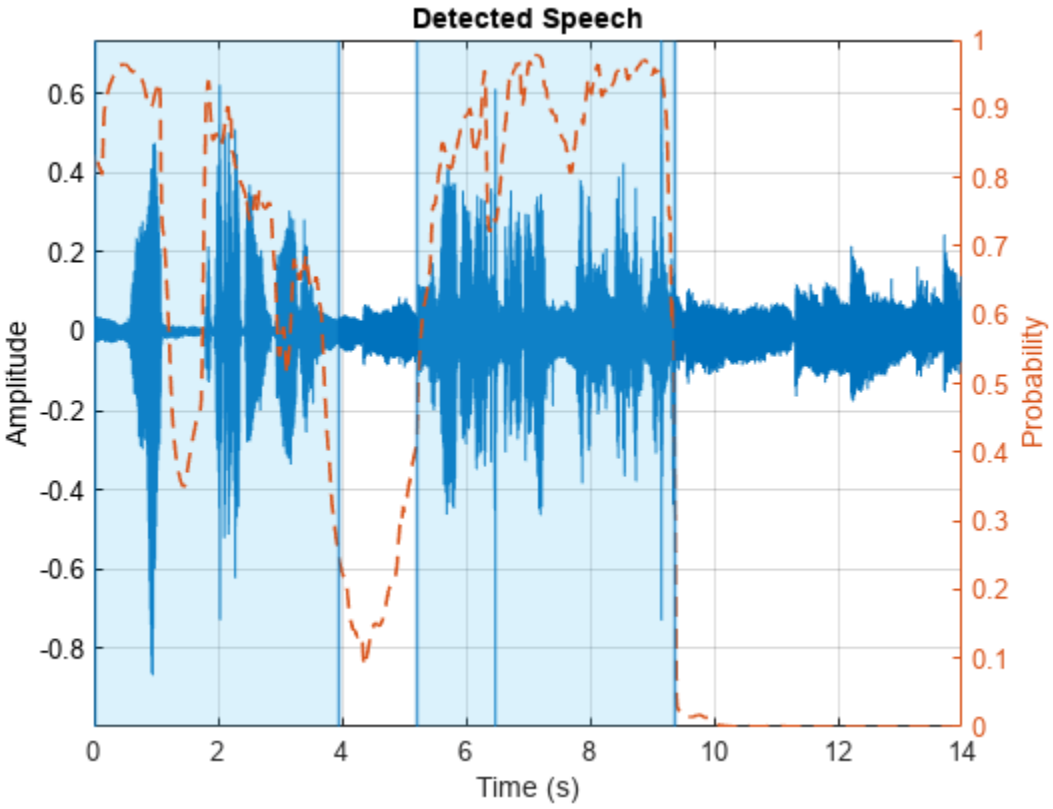
Use `vadnetPosprocess` to postprocess the network output and determine the boundaries of the speech regions in the signal.

```
roi = vadnetPostprocess(audioIn,fs,probs)
```

```
roi = 2×2

           1       63120
       83600      150000
```

Plot the audio with the detected speech regions.

```
vadnetPostprocess(audioIn,fs,probs)
```



Use VAD Neural Network on Streaming Audio

Create a `dsp.AudioFileReader` object to stream an audio file for processing. Set the `SamplesPerFrame` property to read 100 ms nonoverlapping chunks from the signal.

```
afr = dsp.AudioFileReader("MaleVolumeUp-16-mono-6secs.ogg");
analysisDuration = 0.1; % seconds
afr.SamplesPerFrame = floor(analysisDuration*afr.SampleRate);
```

The `vadnet` architecture does not retain state between calls, and it performs best when analyzing larger chunks of audio signals. When you use `vadnet` in a streaming scenario, specific application requirements of accuracy, computational efficiency, and latency dictate the analysis duration and whether to overlap analysis chunks.

Create a `timescope` object to plot the audio signal and the corresponding speech probabilities. Create an `audioDeviceWriter` to play the audio as you stream it.

```
scope = timescope(NumInputPorts=2, ...
    SampleRate=afr.SampleRate, ...
    TimeSpanSource="property",TimeSpan=5, ...
    YLimits=[-1.2,1.2], ...
    ShowLegend=true,ChannelNames=["Audio","Speech Probability"]);
adw = audioDeviceWriter(afr.SampleRate);
```

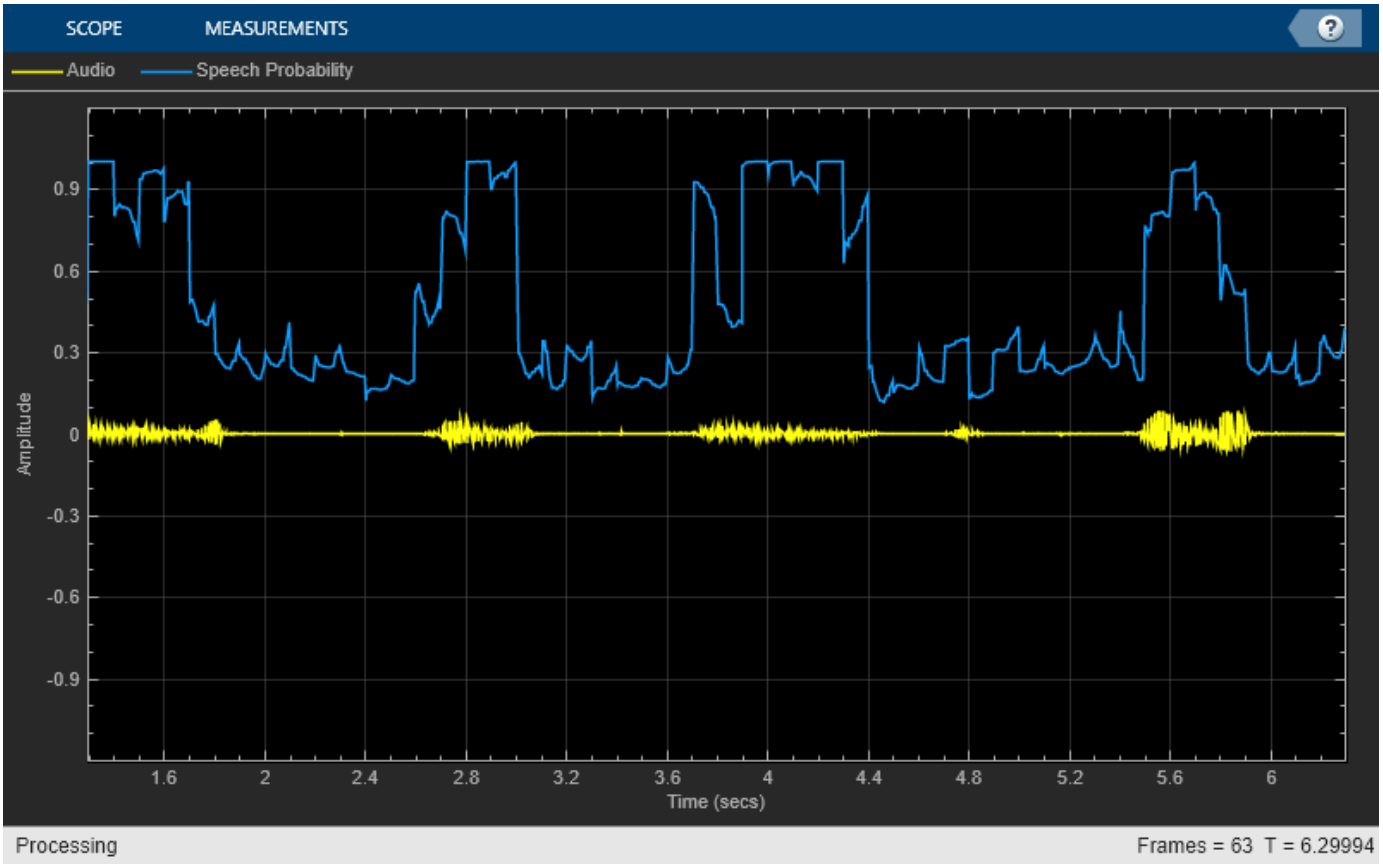
Call `vadnet` to obtain a pretrained VAD neural network.

```
net = vadnet();
```

In a streaming loop:

1. Read in a 100 ms chunk from the audio file.
2. Preprocess the audio into a mel spectrogram using `vadnetPreprocess`.
3. Use the VAD network to predict the probability of speech in each frame of the spectrogram. Replicate the probabilities to correspond to each sample in the audio signal.
4. Plot the audio signal and the probabilities of speech.
5. Play the audio with the device writer.

```
hop = 0.01 * afr.SampleRate;
while ~isDone(afr)
    audioIn = afr();

    features = vadnetPreprocess(audioIn,afr.SampleRate);
    probs = predict(net,features);
    % Replicate probs to correspond to samples in audioIn
    probs = repelem(probs,hop)';
    probs = probs((hop/2)+1:end-hop/2);

    scope(audioIn,probs)
    adw(audioIn);
end
```



Output Arguments

net — Pretrained VAD model
`DAGNetwork` object

Pretrained VAD neural network, returned as a `DAGNetwork` object.

Algorithms

The neural network is a ported version of the `vad-crdnn-libriparty` pretrained model provided by SpeechBrain[1], which combines convolutional, recurrent, and fully connected layers.

Version History

Introduced in R2023a

References

[1] Ravanelli, Mirco, et al. *SpeechBrain: A General-Purpose Speech Toolkit*. arXiv, 8 June 2021. *arXiv.org*, http://arxiv.org/abs/2106.04624

See Also

Functions
`vadnetPreprocess` | `vadnetPostprocess` | `detectspeechnn` | `detectSpeech`

Objects
`voiceActivityDetector`

Blocks
Voice Activity Detector

Topics
“Voice Activity Detection in Noise Using Deep Learning”
“Train Voice Activity Detection in Noise Model Using Deep Learning”

vadnetPreprocess

Preprocess audio for voice activity detection (VAD) network

Syntax

```
features = vadnetPreprocess(audioIn,fs)
```

Description

`features = vadnetPreprocess(audioIn,fs)` returns a mel spectrogram from the audio input that you can feed to the pretrained network returned by `vadnet`.

Examples

Detect Speech with Pretrained VAD Model

Read in an audio signal containing speech and music and listen to the sound.

```
[audioIn,fs] = audioread("MusicAndSpeech-16-mono-14secs.ogg");
sound(audioIn,fs)
```

Use `vadnetPreprocess` to preprocess the audio by computing a mel spectrogram.

```
features = vadnetPreprocess(audioIn,fs);
```

Call `vadnet` to obtain a pretrained VAD neural network.

```
net = vadnet;
```

Pass the preprocessed audio through the network to obtain the probability of speech in each frame.

```
probs = predict(net,features);
```

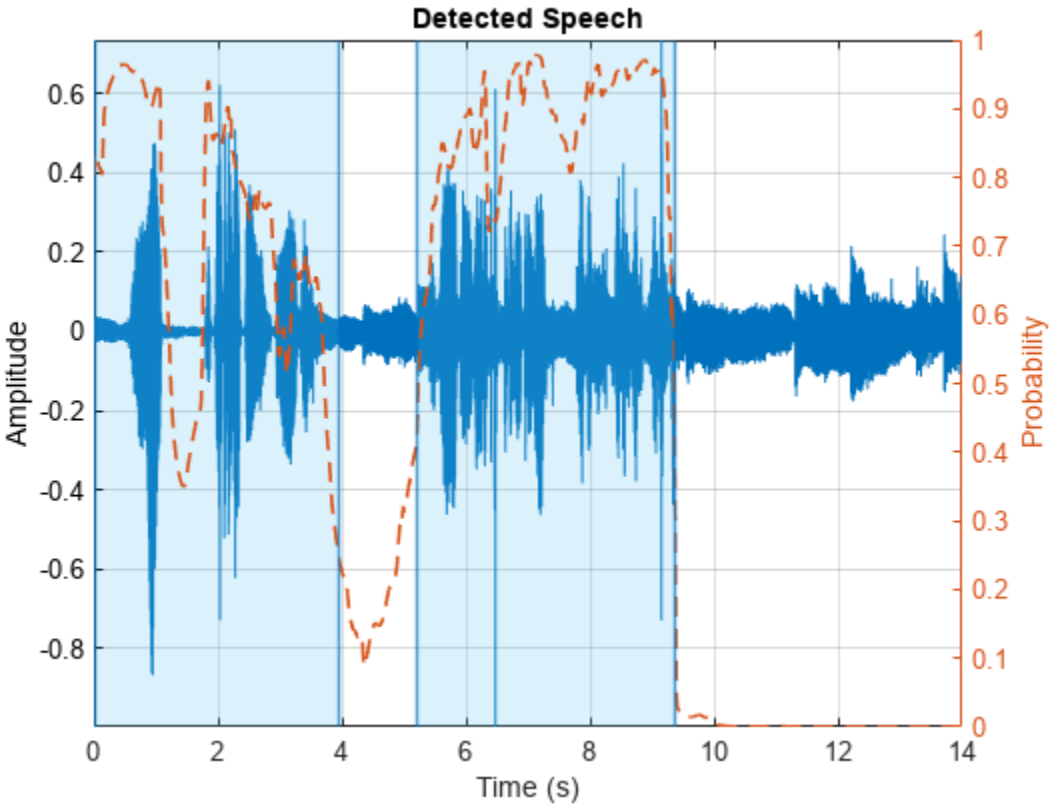
Use `vadnetPosprocess` to postprocess the network output and determine the boundaries of the speech regions in the signal.

```
roi = vadnetPostprocess(audioIn,fs,probs)
```

```
roi = 2×2

           1       63120
       83600      150000
```

Plot the audio with the detected speech regions.

```
vadnetPostprocess(audioIn,fs,probs)
```



Use VAD Neural Network on Streaming Audio

Create a `dsp.AudioFileReader` object to stream an audio file for processing. Set the `SamplesPerFrame` property to read 100 ms nonoverlapping chunks from the signal.

```
afr = dsp.AudioFileReader("MaleVolumeUp-16-mono-6secs.ogg");
analysisDuration = 0.1; % seconds
afr.SamplesPerFrame = floor(analysisDuration*afr.SampleRate);
```

The `vadnet` architecture does not retain state between calls, and it performs best when analyzing larger chunks of audio signals. When you use `vadnet` in a streaming scenario, specific application requirements of accuracy, computational efficiency, and latency dictate the analysis duration and whether to overlap analysis chunks.

Create a `timescope` object to plot the audio signal and the corresponding speech probabilities. Create an `audioDeviceWriter` to play the audio as you stream it.

```
scope = timescope(NumInputPorts=2, ...
    SampleRate=afr.SampleRate, ...
    TimeSpanSource="property",TimeSpan=5, ...
    YLimits=[-1.2,1.2], ...
    ShowLegend=true,ChannelNames=["Audio","Speech Probability"]);
adw = audioDeviceWriter(afr.SampleRate);
```

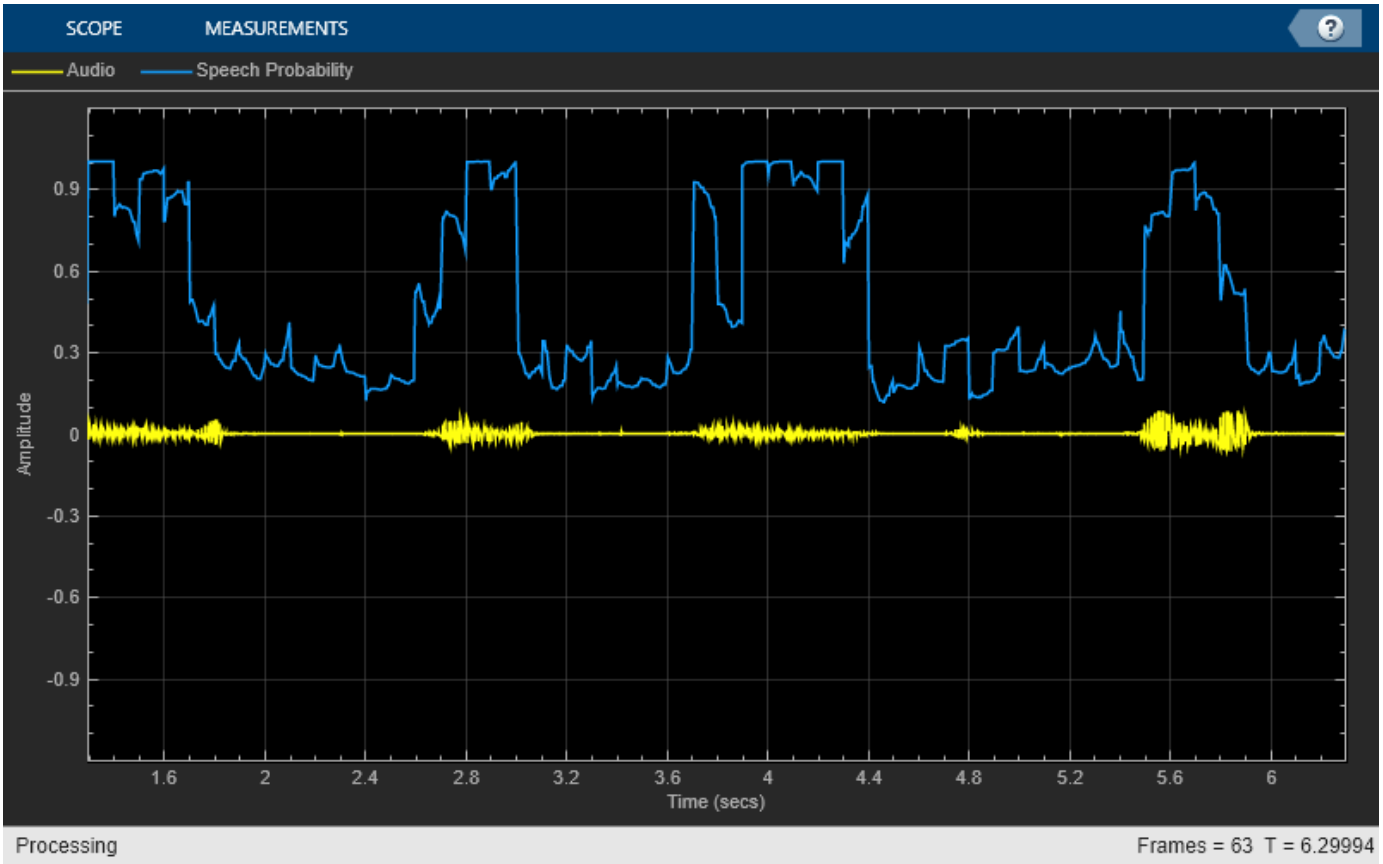
Call `vadnet` to obtain a pretrained VAD neural network.

```
net = vadnet();
```

In a streaming loop:

1. Read in a 100 ms chunk from the audio file.
2. Preprocess the audio into a mel spectrogram using `vadnetPreprocess`.
3. Use the VAD network to predict the probability of speech in each frame of the spectrogram. Replicate the probabilities to correspond to each sample in the audio signal.
4. Plot the audio signal and the probabilities of speech.
5. Play the audio with the device writer.

```
hop = 0.01 * afr.SampleRate;
while ~isDone(afr)
    audioIn = afr();

    features = vadnetPreprocess(audioIn,afr.SampleRate);
    probs = predict(net,features);
    % Replicate probs to correspond to samples in audioIn
    probs = repelem(probs,hop)';
    probs = probs((hop/2)+1:end-hop/2);

    scope(audioIn,probs)
    adw(audioIn);
end
```



Input Arguments

`audioIn` — Audio input

column vector

Audio input signal, specified as a column vector (single channel).

Data Types: `single` | `double`

`fs` — Sample rate (Hz)

positive scalar

Sample rate in Hz, specified as a positive scalar.

Data Types: `single` | `double`

Output Arguments

`features` — Mel spectrogram

40-by-*T* matrix

Mel spectrogram, returned as a 40-by-*T* matrix, where *T* is the number of spectra in the spectrogram.

Algorithms

The `vadnetPreprocess` function preprocesses the audio data using the following steps.

1. Resample the audio to 16kHz.
2. Compute a centered short-time Fourier transform (STFT) using a 25 ms periodic Hamming window and 10 ms hop length. Pad the signal so that the first window is centered at 0 s.
3. Convert the STFT to a power spectrogram.
4. Apply a mel filter bank with 40 bands to obtain a mel spectrogram.
5. Convert the mel spectrogram to a log scale.
6. Standardize each of the mel bands to have zero mean and standard deviation of 1.

Version History

Introduced in R2023a

See Also

Functions

`vadnet` | `vadnetPostprocess` | `detectspeechnn` | `detectSpeech`

Objects

`voiceActivityDetector`

Blocks

Voice Activity Detector

Topics

“Voice Activity Detection in Noise Using Deep Learning”

“Train Voice Activity Detection in Noise Model Using Deep Learning”

vadnetPostprocess

Postprocess frame-based VAD probabilities

Syntax

```
roi = vadnetPostprocess(audioIn,fs,probs)
roi = vadnetPostprocess( ___ ,Name=Value)
vadnetPostprocess( ___ )
```

Description

`roi = vadnetPostprocess(audioIn,fs,probs)` postprocesses the speech probabilities output by a voice activity detection (VAD) network and returns indices corresponding to the beginning and end of speech within the audio signal.

`roi = vadnetPostprocess( ___ ,Name=Value)` specifies options using one or more name-value arguments. For example, `vadnetPostprocess(audioIn,fs,MergeThreshold=0.5)` merges speech regions that are separated by 0.5 seconds or less.

`vadnetPostprocess( ___ )` with no output arguments plots the input signal and the detected speech regions.

Examples

Detect Speech with Pretrained VAD Model

Read in an audio signal containing speech and music and listen to the sound.

```
[audioIn,fs] = audioread("MusicAndSpeech-16-mono-14secs.ogg");
sound(audioIn,fs)
```

Use `vadnetPreprocess` to preprocess the audio by computing a mel spectrogram.

```
features = vadnetPreprocess(audioIn,fs);
```

Call `vadnet` to obtain a pretrained VAD neural network.

```
net = vadnet;
```

Pass the preprocessed audio through the network to obtain the probability of speech in each frame.

```
probs = predict(net,features);
```

Use `vadnetPosprocess` to postprocess the network output and determine the boundaries of the speech regions in the signal.

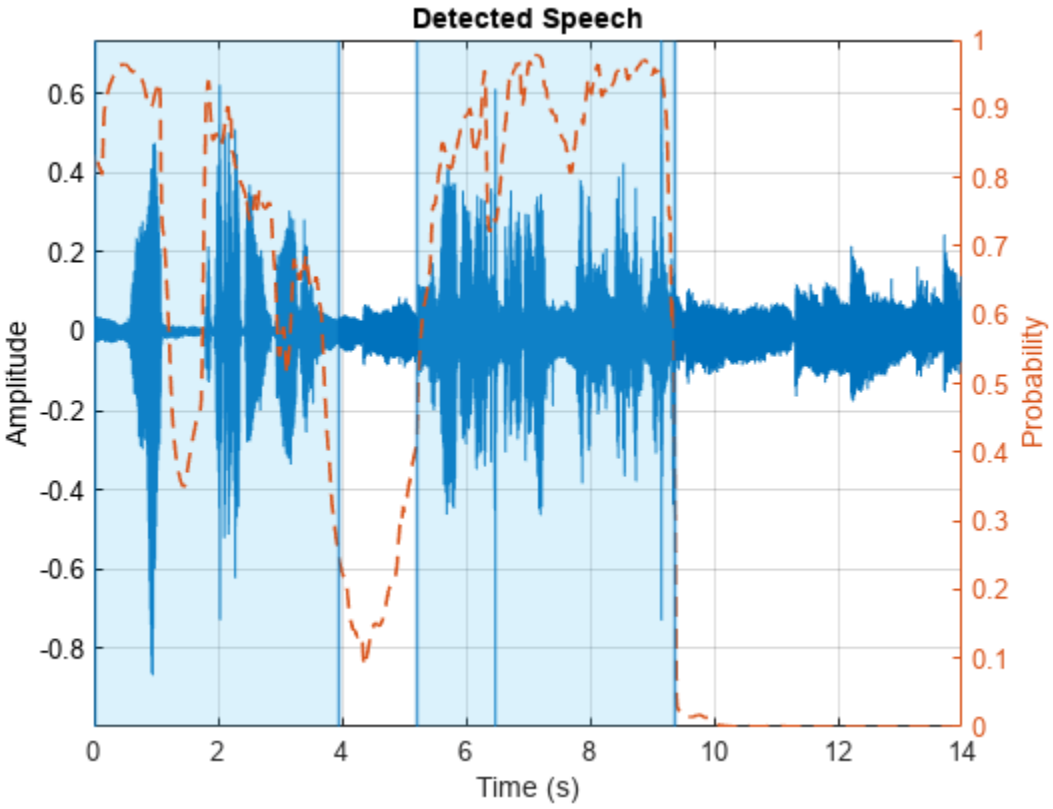
```
roi = vadnetPostprocess(audioIn,fs,probs)
```

```
roi = 2×2

           1       63120
```

```
        83600      150000
```

Plot the audio with the detected speech regions.

```
vadnetPostprocess(audioIn,fs,probs)
```



Customize VAD Postprocessing

Read in an audio signal containing speech and music and listen to the sound.

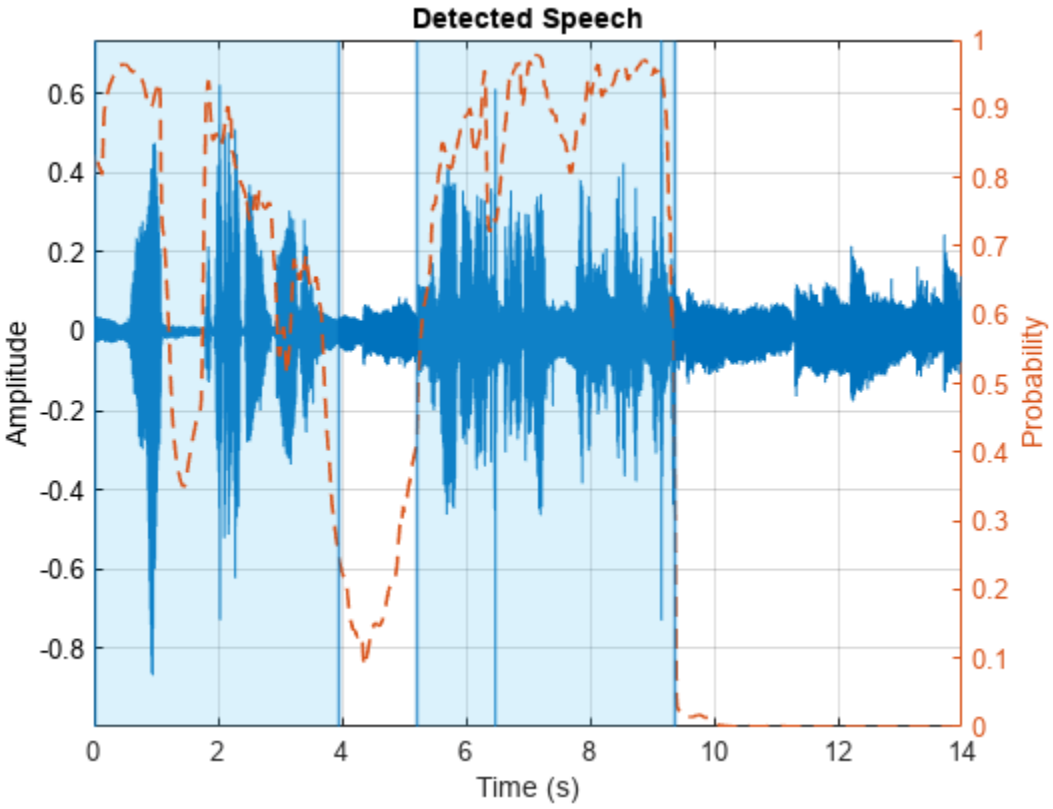
```
[audioIn,fs] = audioread("MusicAndSpeech-16-mono-14secs.ogg");
sound(audioIn,fs)
```

Preprocess the audio and pass it through the pretrained `vadnet` model.

```
features = vadnetPreprocess(audioIn,fs);
net = vadnet;
probs = predict(net,features);
```

Call `vadnetPostprocess` with the merge threshold set to 1 to merge detected speech regions that are separated by 1 second or less.

```
vadnetPostprocess(audioIn,fs,probs,MergeThreshold=1)
```



Input Arguments

`audioIn` — Audio input

column vector

Audio input signal, specified as a column vector (single channel).

Data Types: `single` | `double`

`fs` — Sample rate (Hz)

positive scalar

Sample rate in Hz, specified as a positive scalar.

Data Types: `single` | `double`

`probs` — VAD probabilities

vector

VAD probabilities of speech in each audio frame, specified as a vector. These probabilities are the output of a `vadnet` model.

Data Types: `single` | `double`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `vadnetPostprocess(audioIn,fs,probs,ApplyEnergyVAD=true)`

`MergeThreshold` — Merge threshold
`0.25` (default) | nonnegative scalar

Merge threshold in seconds, specified as a nonnegative scalar. The function merges speech regions that are separated by a duration less than or equal to the specified threshold. Set the threshold to `Inf` to not merge any detected regions.

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

`LengthThreshold` — Length threshold
`0.25` (default) | nonnegative scalar

Length threshold in seconds, specified as a nonnegative scalar. The function does not return speech regions that have a duration less than or equal to the specified threshold.

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

`ActivationThreshold` — Probability threshold to start a speech segment
`0.5` (default) | scalar in the range [0, 1]

Probability threshold to start a speech segment, specified as a scalar in the range [0, 1].

Data Types: `single` | `double`

`DeactivationThreshold` — Probability threshold to end a speech segment
`0.25` (default) | scalar in the range [0, 1]

Probability threshold to end a speech segment, specified as a scalar in the range [0, 1].

Data Types: `single` | `double`

`ApplyEnergyVAD` — Apply energy-based VAD
`false` (default) | `true`

Apply energy-based VAD to the speech regions detected by the neural network, specified as `true` or `false`.

Data Types: `logical`

Output Arguments

`roi` — Speech regions
N-by-2 matrix

Speech regions, returned as an *N*-by-2 matrix of indices into the input signal, where *N* is the number of individual speech regions detected. The first column contains the index of the start of a speech region, and the second column contains the index of the end of a region.

Algorithms

The `vadnetPostprocess` function postprocesses the VAD network output using the following steps.

1. Apply activation and deactivation thresholds to posterior probabilities to determine candidate speech regions.
2. Optionally, apply energy-based VAD to refine the detected speech regions.
3. Merge speech regions that are close to each other according to the merge threshold.
4. Remove speech regions that are shorter than or equal to the length threshold.

Version History

Introduced in R2023a

See Also

Functions
`vadnet` | `vadnetPreprocess` | `detectspeechnn` | `detectSpeech`

Objects
`voiceActivityDetector`

Blocks
Voice Activity Detector

Topics
“Voice Activity Detection in Noise Using Deep Learning”
“Train Voice Activity Detection in Noise Model Using Deep Learning”

audioEnvelope

Compute envelope of an audio file

Syntax

```
[minEnv,maxEnv] = audioEnvelope(audioIn)
[minEnv,maxEnv] = audioEnvelope(filename)
[minEnv,maxEnv] = audioEnvelope( ___ ,Name=Value)

[minEnv,maxEnv,loc] = audioEnvelope( ___ )
[minEnv,maxEnv,loc,fs] = audioEnvelope( ___ )
audioEnvelope( ___ )
```

Description

`[minEnv,maxEnv] = audioEnvelope(audioIn)` returns the envelope of the input audio signal. The audio envelope contains maximum and minimum values over nonoverlapping frames of the input signal, and it approximates the shape of the waveform.

`[minEnv,maxEnv] = audioEnvelope(filename)` returns the envelope of the audio file.

`[minEnv,maxEnv] = audioEnvelope( ___ ,Name=Value)` specifies options using one or more name-value arguments. For example, `audioEnvelope(filename,Range=[1000, 5000])` computes the envelope of the signal from samples 1000 through 5000 in the audio file.

`[minEnv,maxEnv,loc] = audioEnvelope( ___ )` also returns the locations of the envelope frames within the original signal.

`[minEnv,maxEnv,loc,fs] = audioEnvelope( ___ )` also returns the sample rate of the audio signal.

`audioEnvelope( ___ )` with no output arguments plots the audio envelope.

Examples

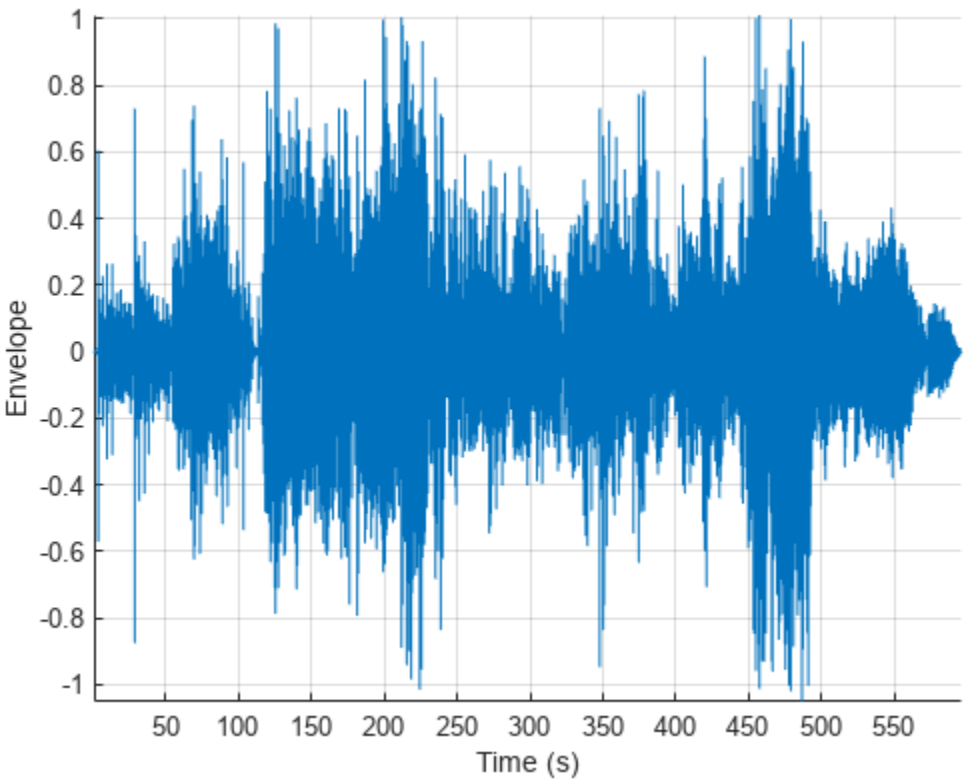
Envelope of Audio File

Compute the envelope of a file containing a 10-minute audio signal.

```
[envMin,envMax] = audioEnvelope("SoftGuitar-44p1_mono-10mins.ogg");
```

Plot the envelope by calling `audioEnvelope` with no output arguments.

```
audioEnvelope("SoftGuitar-44p1_mono-10mins.ogg")
```



Envelope of Audio Signal

Read in a 10-minute audio signal from a file.

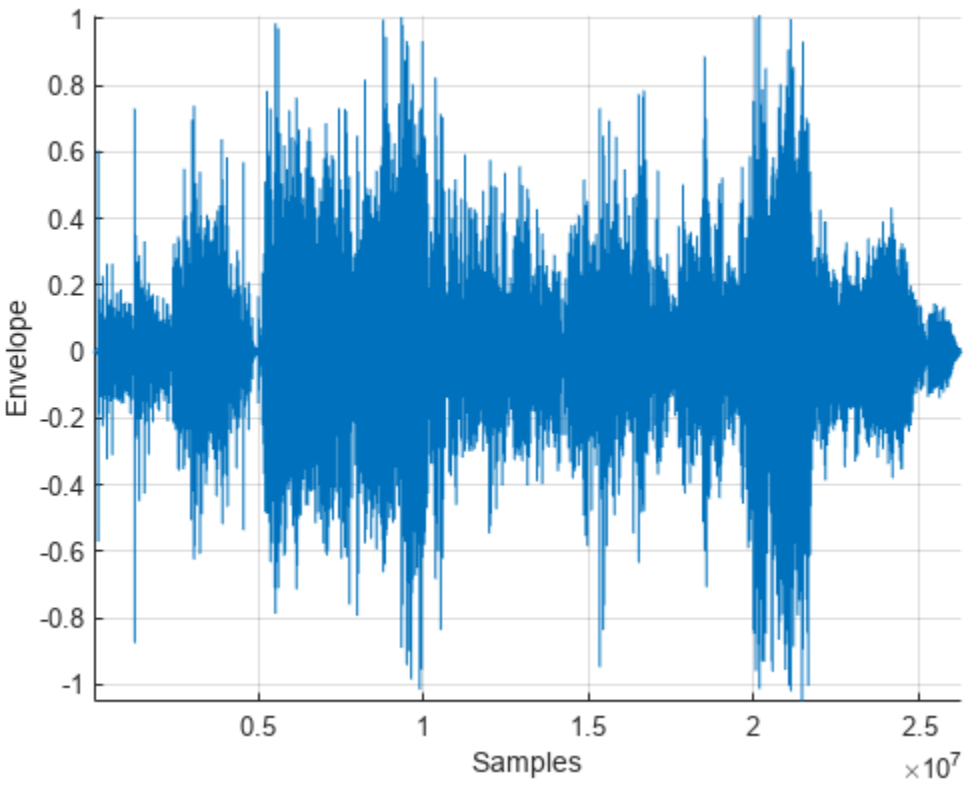
```
[audioIn,fs] = audioread("SoftGuitar-44p1_mono-10mins.ogg");
```

Compute the audio envelope of the signal.

```
[envMin,envMax] = audioEnvelope(audioIn);
```

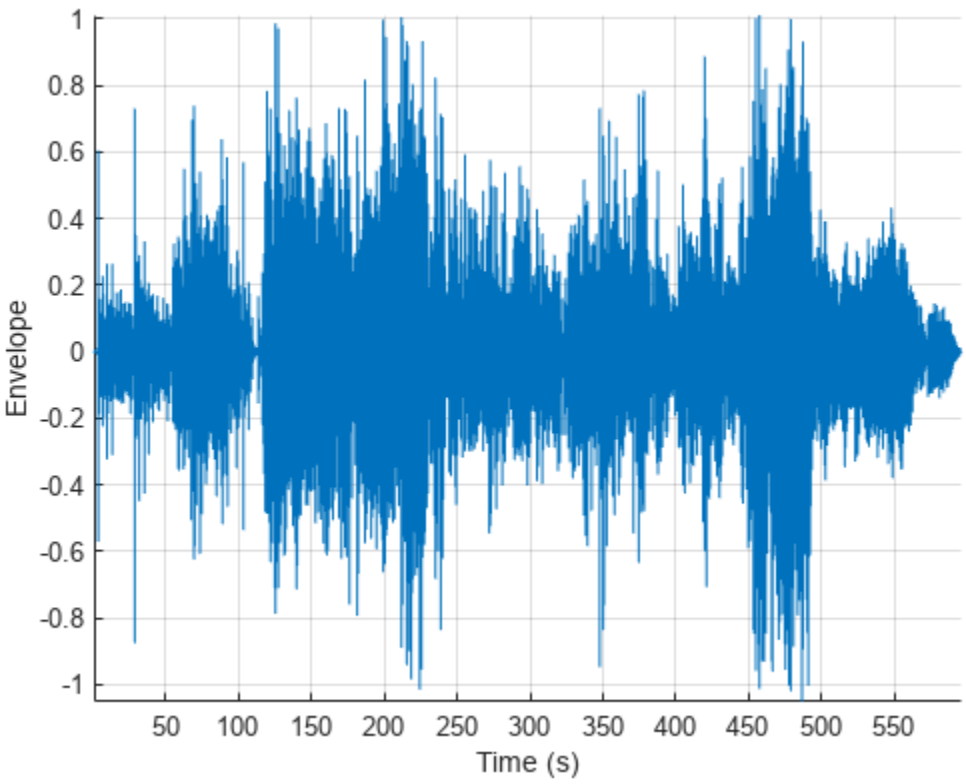
Call `audioEnvelope` with no output arguments to plot the envelope.

```
audioEnvelope(audioIn)
```



Specify the `SampleRate` of the signal to see time on the *x*-axis of the convenience plot instead of samples.

```
audioEnvelope(audioIn,SampleRate=fs)
```



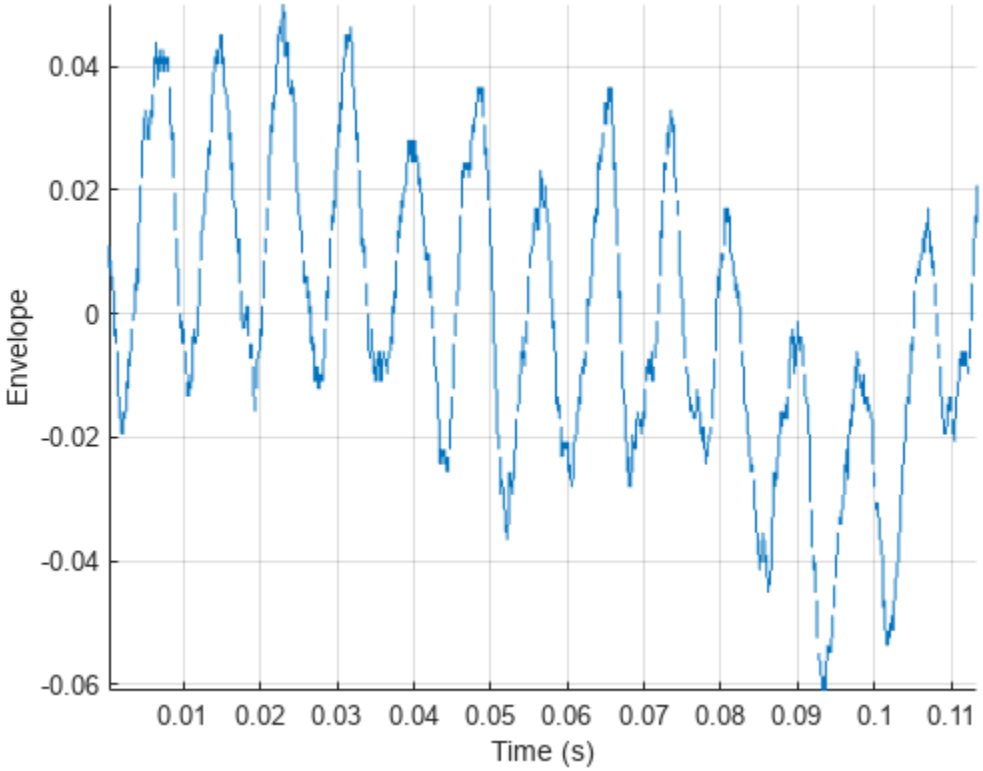
Specify Range and Size of Audio Envelope

Compute the envelope from samples 5000 to 10,000 of the audio file. Set the number of points in the envelope to be 500.

```
[envMin,envMax] = audioEnvelope("Counting-16-44p1-mono-15secs.wav", ...
                                Range=[5000, 10000],NumPoints=500);
```

Plot the envelope of the specified size and range.

```
audioEnvelope("Counting-16-44p1-mono-15secs.wav", ...
    Range=[5000, 10000],NumPoints=500);
```



Audio Envelope with Frame Locations

Compute the envelope of an audio file. Specify additional output arguments to obtain the locations of the envelope frames and the sample rate of the audio signal.

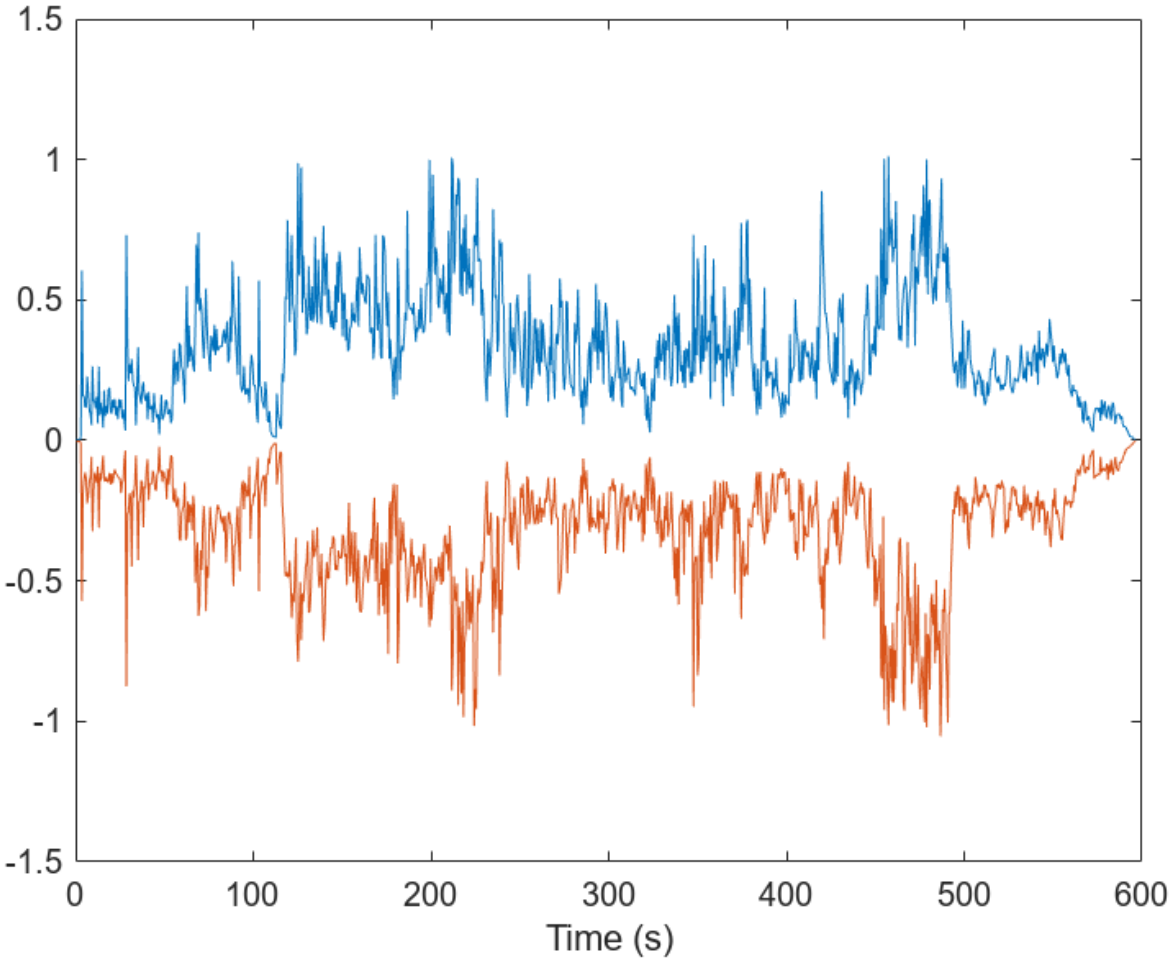
```
[minEnv,maxEnv,loc,fs] = audioEnvelope("SoftGuitar-44p1_mono-10mins.ogg");
```

Use the sample rate to convert the frame locations from samples to seconds.

```
loc = loc./fs;
```

Plot the maximum and minimum values of the envelope with the time of the original audio signal on the *x*-axis.

```
plot(loc,maxEnv,loc,minEnv)
xlabel("Time (s)")
```



Input Arguments

`audioIn` — Audio input

column vector | matrix

Audio input signal, specified as a column vector or matrix. If the input is a matrix, the columns are treated as individual channels.

Data Types: `single` | `double`

`filename` — Name of audio file

string scalar | character vector

Name of the audio file, specified as a string scalar or character vector. `audioEnvelope` accepts the same file formats as `audioread`.

Data Types: `char` | `string`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Example: `audioEnvelope(audioIn,NumPoints=5000)`

`NumPoints` — Number of points in envelope

`1000` (default) | positive integer

Number of points in the audio envelope, specified as a positive integer.

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

`Range` — Range of envelope

row vector of two positive integers

Range of the envelope in samples, specified as a row vector of two positive integers. The range specifies the start and end indices into the input signal that define the region over which to compute the envelope. The default range is the entire input signal.

Example: Range=[1000, 2000]

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

`SampleRate` — Sample rate in Hz

positive scalar

Sample rate in Hz, specified as a positive scalar.

You can use this name-value argument only when the input is a numeric array. If the input is a file name, the function derives the sample rate from the audio file information.

Data Types: `single` | `double`

Output Arguments

`minEnv` — Minimum values of envelope

matrix

Minimum values of the envelope, returned as a `NumPoints`-by-*C* matrix, where *C* is the number of channels in the input signal. The values are the minimums over nonoverlapping frames in the input signal. The frame size is equal to `floor(`*L*`/NumPoints)`, where *L* is the length of the signal.

`maxEnv` — Maximum values of envelope

matrix

Maximum values of the envelope, returned as a `NumPoints`-by-*C* matrix, where *C* is the number of channels in the input signal. The values are the maximums over nonoverlapping frames in the input signal. The frame size is equal to `floor(`*L*`/NumPoints)`, where *L* is the length of the signal.

`loc` — Frame locations

row vector

Frame locations, returned as a row vector of length `NumPoints`. The locations are indices into the input signal of the most recent sample of each frame.

`fs` — Sample rate in Hz
positive scalar

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

If you specify the input signal as a numeric array instead of a file and do not specify the `SampleRate` argument, then the sample rate is returned as 1.

Version History

Introduced in R2023a

See Also

`audioread`

Topics
“Plot Large Audio Files”

getOptions

Get server options

Syntax

```
options = getOptions(clientObj)
```

Description

`options = getOptions(clientObj)` returns the server options for the `speechClient` object.

Examples

Configure Server Options for Third-Party Speech Service

Create a `speechClient` object that interfaces with the Google speech service.

```
clientObj = speechClient("Google");
```

Set the `languageCode` server option for speech-to-text transcription.

```
setOptions(clientObj,"languageCode","en-US");
```

Use `getOptions` to view the server options in the `speechClient` object.

```
options = getOptions(clientObj)
```

```
options=1×2 cell array
    {'languageCode'}    {["en-US"]}
```

Use `clearOptions` to remove the server option from the `speechClient` object.

```
clearOptions(clientObj)
```

Input Arguments

`clientObj` — Client object

`speechClient` object

Client object, specified as an object returned by `speechClient`. Server options are only valid for client objects that interface with third-party speech services (Google, IBM, Microsoft, or Amazon).

Output Arguments

`options` — Server options

cell array

Server options, returned as a cell array of name-value pairs in the format `{Name1,Value1,...,NameN,ValueN}`, where `Name` is the server option name and `Value` is the

value of the option. See the documentation for the third-party speech service for information about valid server options.

Version History

Introduced in R2023a

See Also

`speechClient` | `setOptions` | `clearOptions`

setOptions

Set server options

Syntax

```
setOptions(clientObj,Name,Value)
```

Description

`setOptions(clientObj,Name,Value)` sets one or more server options for the `speechClient` object.

Examples

Configure Server Options for Third-Party Speech Service

Create a `speechClient` object that interfaces with the Google speech service.

```
clientObj = speechClient("Google");
```

Set the `languageCode` server option for speech-to-text transcription.

```
setOptions(clientObj,"languageCode","en-US");
```

Use `getOptions` to view the server options in the `speechClient` object.

```
options = getOptions(clientObj)
```

```
options=1×2 cell array
    {'languageCode'}    {["en-US"]}
```

Use `clearOptions` to remove the server option from the `speechClient` object.

```
clearOptions(clientObj)
```

Input Arguments

`clientObj` — Client object

`speechClient` object

Client object, specified as an object returned by `speechClient`. You can only set server options for client objects that interface with third-party speech services (Google, IBM, Microsoft, or Amazon).

Name-Value Pair Arguments

Specify server options as `Name1,Value1,...,NameN,ValueN`, where `Name` is a string containing the server option and `Value` is the corresponding value.

For example, `setOptions(clientObj,"languageCode","en-US")` sets the `languageCode` server option to `"en-US"`.

Valid server options and values depend on the specific third-party API. See the documentation for the specific service for option names and values.

- Google Cloud Speech-to-Text documentation
- Google Cloud Text-to-Speech documentation
- IBM Watson Speech to Text documentation
- IBM Watson Text to Speech documentation
- Microsoft Azure Speech service documentation
- Amazon Transcribe documentation
- Amazon Polly documentation

Version History

Introduced in R2023a

See Also

`speechClient` | `getOptions` | `clearOptions`

clearOptions

Remove all server options

Syntax

```
clearOptions(clientObj)
```

Description

`clearOptions(clientObj)` removes all user-specified server options from the `speechClient` object.

Examples

Configure Server Options for Third-Party Speech Service

Create a `speechClient` object that interfaces with the Google speech service.

```
clientObj = speechClient("Google");
```

Set the `languageCode` server option for speech-to-text transcription.

```
setOptions(clientObj,"languageCode","en-US");
```

Use `getOptions` to view the server options in the `speechClient` object.

```
options = getOptions(clientObj)
```

```
options=1×2 cell array
    {'languageCode'}    {["en-US"]}
```

Use `clearOptions` to remove the server option from the `speechClient` object.

```
clearOptions(clientObj)
```

Input Arguments

`clientObj` — Client object

`speechClient` object

Client object, specified as an object returned by `speechClient`. Server options are valid only for client objects that interface with third-party speech services (Google, IBM, Microsoft, or Amazon).

Version History

Introduced in R2023a

See Also

`speechClient` | `setOptions` | `getOptions`

System Objects

audioTimeScaler

Apply time scaling to streaming audio

Description

The `audioTimeScaler` object performs audio time scale modification (TSM) independently across each input channel.

To modify the time scale of streaming audio:

1. Create the `audioTimeScaler` object and set its properties.
2. Call the object with arguments, as if it were a function.

To learn more about how System objects work, see What Are System Objects?

Creation

Syntax

```
aTS = audioTimeScaler
aTS = audioTimeScaler(speedupFactor)
aTS = audioTimeScaler( ___ ,'Name',Value)
```

Description

`aTS = audioTimeScaler` creates an object, `aTS`, that performs audio time scale modification independently across each input channel over time.

`aTS = audioTimeScaler(speedupFactor)` sets the `SpeedupFactor` property to `speedupFactor`.

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

Example: `aTS = audioTimeScaler(1.2,'Window',sqrt(hann(1024,'periodic')),'OverlapLength',768)` creates an object, `aTS`, that increases the tempo of audio by 1.2 times its original speed using a periodic 1024-point Hann window and a 768-point overlap.

Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see System Design in MATLAB Using System Objects.

`SpeedupFactor` — Speedup factor

`1.1` (default) | positive real scalar

Speedup factor, specified as a positive real scalar.

Tunable: Yes

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

`InputDomain` — Domain of input signal

`"Time"` (default) | `"Frequency"`

Domain of the input signal, specified as `"Time"` or `"Frequency"`.

Data Types: `char` | `string`

`Window` — Analysis window

`sqrt(hann(512,'periodic'))` (default) | real vector

Analysis window, specified as a real vector.

Note If using `audioTimeScaler` with frequency-domain input, you must specify `Window` as the same window used to transform `audioIn` to the frequency domain.

Data Types: `single` | `double`

`OverlapLength` — Overlap length of adjacent analysis windows

`384` (default) | nonnegative integer

Overlap length of adjacent analysis windows, specified as a nonnegative integer.

Note If using `audioTimeScaler` with frequency-domain input, you must specify `OverlapLength` as the same overlap length used to transform `audioIn` to a time-frequency representation.

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

`FFTLength` — FFT length

`[]` (default) | positive scalar integer

FFT length, specified as a positive integer. The default, `[]`, means that the FFT length is equal to the number of rows in the input signal.

Dependencies

To enable this property, set `InputDomain` to `'Time'`.

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

`LockPhase` — Apply identity phase locking

`false` (default) | `true`

Apply identity phase locking, specified as `true` or `false`.

Data Types: `logical`

Usage

Syntax

```
audioOut = aTS(audioIn)
```

Description

`audioOut = aTS(audioIn)` applies time-scale modification to the input, `audioIn`, and returns the time-scaled output, `audioOut`.

Input Arguments

`audioIn` — Input audio
column vector | matrix

Input audio, specified as a column vector or matrix. How `audioTimeScaler` interprets `audioIn` depends on the InputDomain property.

- If `InputDomain` is set to `"Time"`, `audioIn` must be a real *N*-by-1 column vector or *N*-by-*C* matrix. The number of rows, *N*, must be equal to or less than the hop length (`size(audioIn,1) <= numel(Window)-OverlapLength`). Columns of a matrix are interpreted as individual channels.
- If `InputDomain` is set to `"Frequency"`, specify `audioIn` as a real or complex *NFFT*-by-1 column vector or *NFFT*-by-*C* matrix. The number of rows, *NFFT*, is the number of points in the DFT calculation, and is set on the first call to the audio time scaler. *NFFT* must be greater than or equal to the window length (`size(audioIn,1) >= numel(Window)`). Columns of a matrix are interpreted as individual channels.

Data Types: `single` | `double`
Complex Number Support: Yes

Output Arguments

`audioOut` — Time-stretched audio
column vector | matrix

Time-stretched audio, returned as a column vector or matrix.

Data Types: `single` | `double`

Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

Common to All System Objects

step	Run System object algorithm
release	Release resources and allow changes to System object property values and input characteristics

reset Reset internal states of System object

Examples

Apply Time Scale Modification to Streaming Audio

To minimize artifacts caused by windowing, create a square root Hann window capable of perfect reconstruction. Use `iscola` to verify the design.

```
win = sqrt(hann(1024,'periodic'));
overlapLength = 896;
iscola(win,overlapLength)
```

```
ans = logical
   1
```

Create an `audioTimeScaler` with a speedup factor of `1.5`. Change the value of `alpha` to hear the effect of the speedup factor.

```
alpha = 1.5                                    ;
aTS = audioTimeScaler( ...
    'SpeedupFactor',alpha, ...
    'Window',win, ...
    'OverlapLength',overlapLength);
```

Create a `dsp.AudioFileReader` object to read frames from an audio file. The length of frames input to the audio time scaler must be less than or equal to the analysis hop length defined in `audioTimeScaler`. To minimize buffering, set the samples per frame of the file reader to the analysis hop length.

```
hopLength = numel(aTS.Window) - overlapLength;
fileReader = dsp.AudioFileReader('Counting-16-44p1-mono-15secs.wav', ...
    'SamplesPerFrame',hopLength);
```

Create an `audioDeviceWriter` to write frames to your audio device. Use the same sample rate as the file reader.

```
deviceWriter = audioDeviceWriter('SampleRate',fileReader.SampleRate);
```

In an audio stream loop, read a frame the file, apply time scale modification, and then write a frame to the device.

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

As a best practice, release your objects once done.

```
release(deviceWriter)
release(fileReader)
release(aTS)
```

Process Frequency-Domain Input

Create a window capable of perfect reconstruction. Use `iscola` to verify the design.

```
win = kbdwin(512);
overlapLength = 256;
iscola(win,overlapLength)
```

```
ans = logical
   1
```

Create an `audioTimeScaler` with a speedup factor of `0.8`. Set `InputDomain` to `"Frequency"` and specify the window and overlap length used to transform time-domain audio to the frequency domain. Set `LockPhase` to `true` to increase the fidelity in the time-scaled output.

```
alpha = 0.8;
timeScaleModification = audioTimeScaler( ...
    "SpeedupFactor",alpha, ...
    "InputDomain","Frequency", ...
    "Window",win, ...
    "OverlapLength",overlapLength, ...
    "LockPhase",true);
```

Create a `dsp.AudioFileReader` object to read frames from an audio file. Create a `dsp.STFT` object to perform a short-time Fourier transform on streaming audio. Specify the same window and overlap length you used to create the `audioTimeScaler`. Create an `audioDeviceWriter` object to write frames to your audio device.

```
fileReader = dsp.AudioFileReader('RockDrums-44p1-stereo-11secs.mp3','SamplesPerFrame',numel(win)

shortTimeFourierTransform = dsp.STFT('Window',win,'OverlapLength',overlapLength,'FFTLength',nume

deviceWriter = audioDeviceWriter('SampleRate',fileReader.SampleRate);
```

In an audio stream loop:

1. Read a frame from the file.
2. Input the frame to the STFT. The `dsp.STFT` object performs buffering.
3. Apply time scale modification.
4. Write the modified audio to your audio device.

```
while ~isDone(fileReader)
    x = fileReader();
    X = shortTimeFourierTransform(x);
    y = timeScaleModification(X);
    deviceWriter(y);
end
```

As a best practice, release your objects once done.

```
release(fileReader)
release(shortTimeFourierTransform)
release(timeScaleModification)
release(deviceWriter)
```

Algorithms

`audioTimeScaler` uses the same phase vocoder algorithm as `stretchAudio` and is based on the descriptions in [1] and [2].

Version History

Introduced in R2019b

References

[1] Driedger, Johnathan, and Meinard Müller. "A Review of Time-Scale Modification of Music Signals." *Applied Sciences*. Vol. 6, Issue 2, 2016.

[2] Driedger, Johnathan. "Time-Scale Modification Algorithms for Music Audio Signals." Master's thesis, Saarland University, 2011.

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

`shiftPitch` | `stretchAudio` | `audioDataAugmenter`

parameterTuner

Tune object parameters while streaming

Syntax

```
H = parameterTuner(obj)
```

Description

`H = parameterTuner(obj)` creates a parameter tuning UI and returns a figure handle, `H`.

Examples

Tune Parameters of Multiple Objects

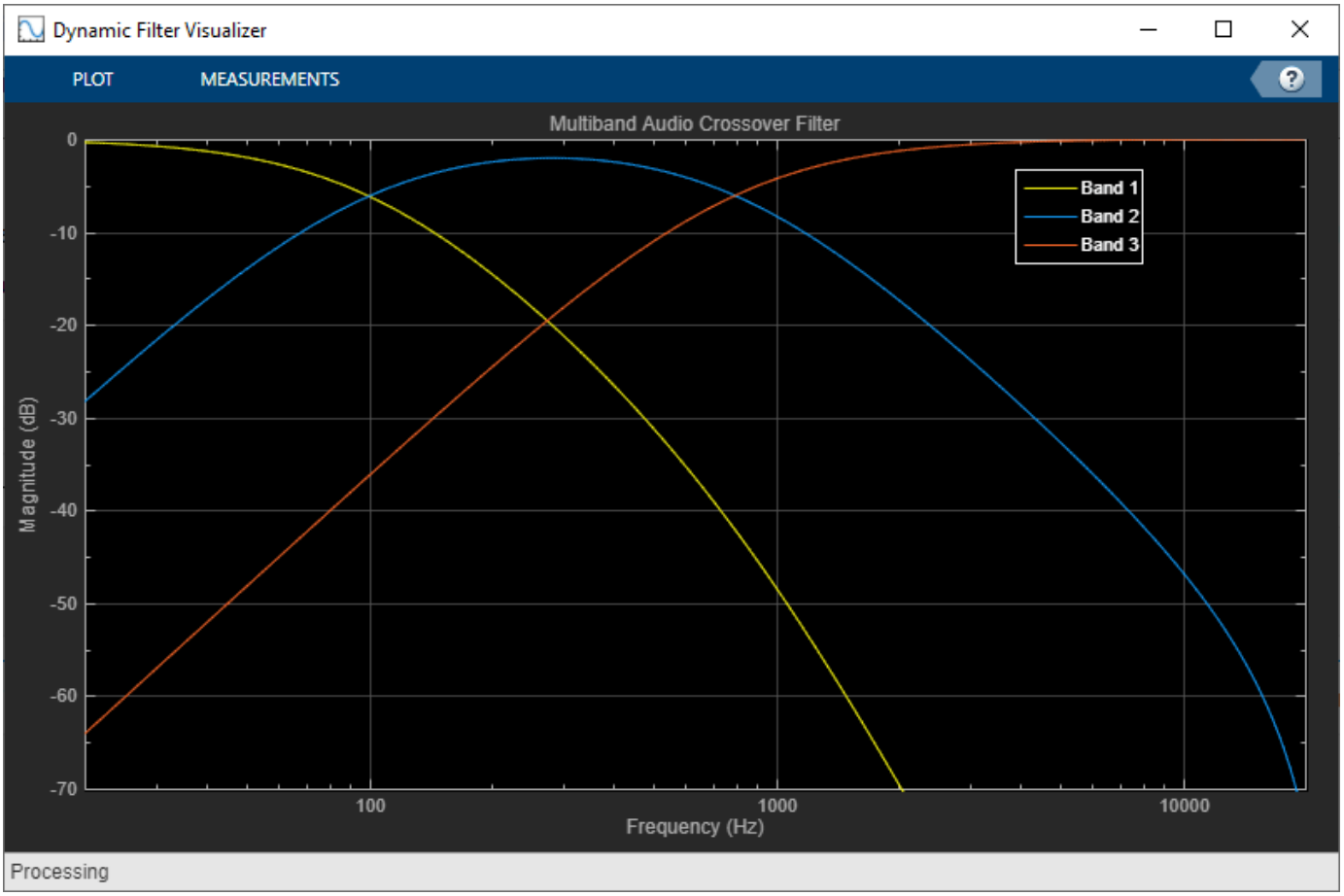
`parameterTuner` enables you to graphically tune parameters of multiple objects. In this example, you use a crossover filter to split a signal into multiple subbands and then apply different effects to the subbands.

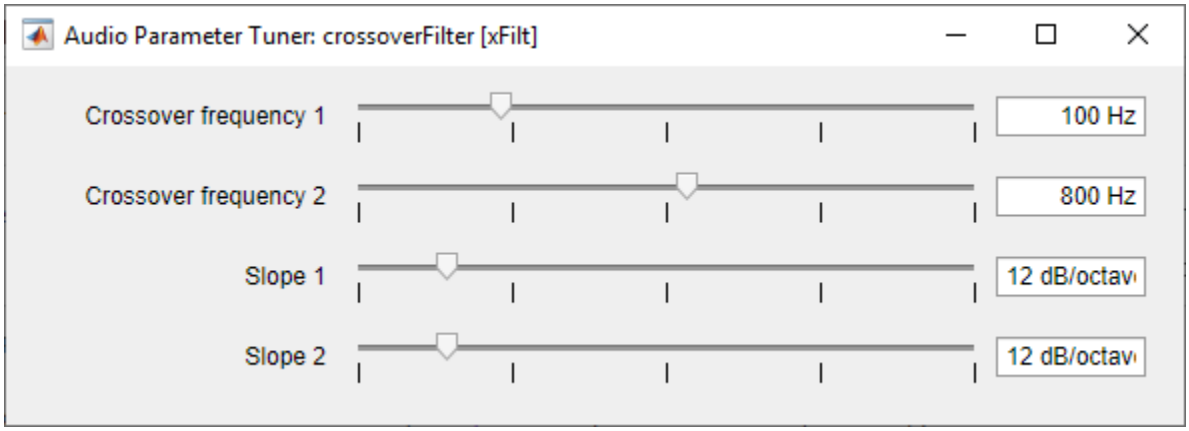
Create a `dsp.AudioFileReader` to read in audio frame-by-frame. Create an `audioDeviceWriter` to write audio to your sound card.

```
fileReader = dsp.AudioFileReader('FunkyDrums-48-stereo-25secs.mp3', ...
                                 'PlayCount',2);
deviceWriter = audioDeviceWriter('SampleRate',fileReader.SampleRate);
```

Create a `crossoverFilter` with two crossovers to split the audio into three bands. Call `visualize` to plot the frequency responses of the filters. Call `parameterTuner` to open a UI to tune the crossover frequencies while streaming.

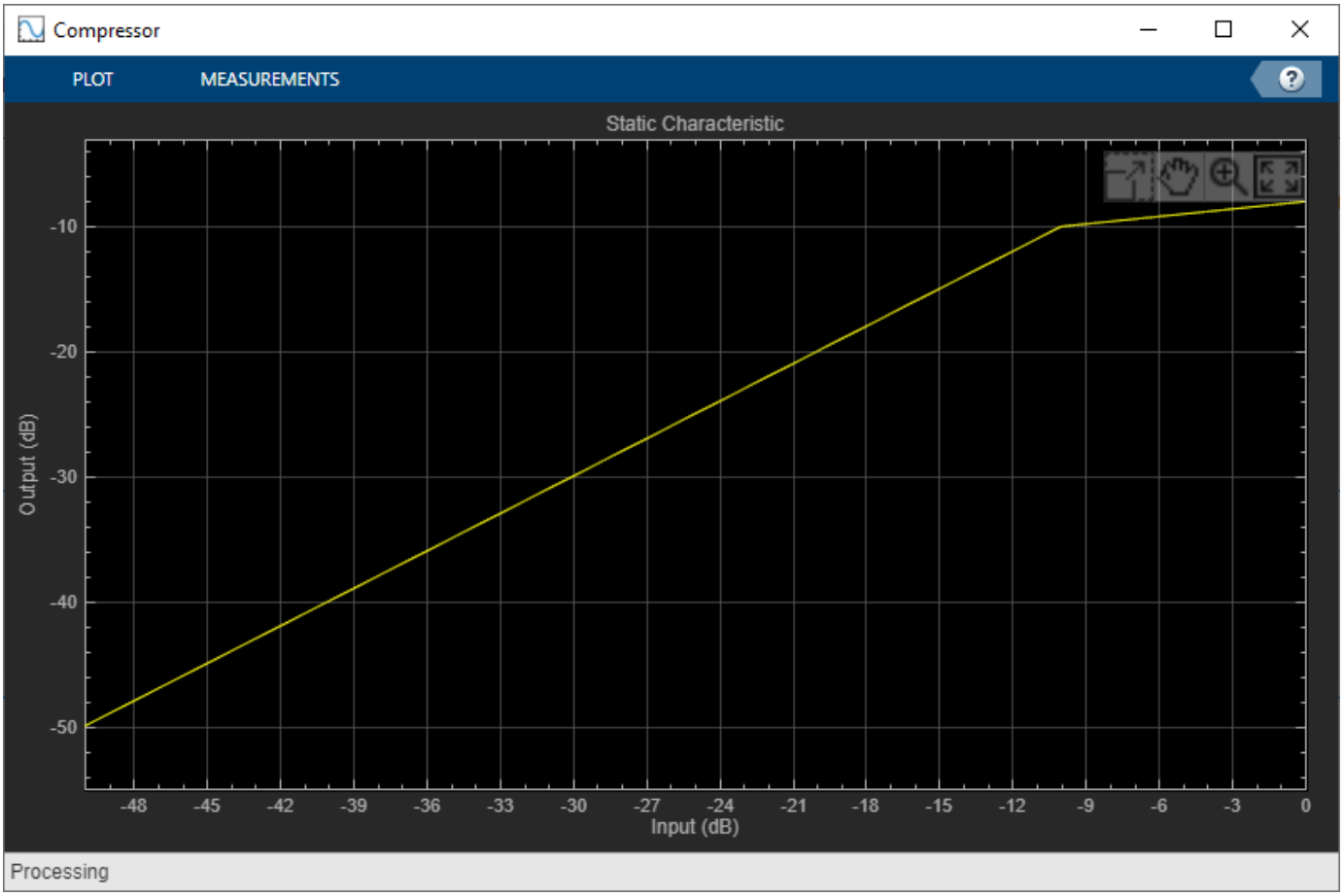
```
xFilt = crossoverFilter('SampleRate',fileReader.SampleRate,'NumCrossovers',2);
visualize(xFilt)
parameterTuner(xFilt)
```

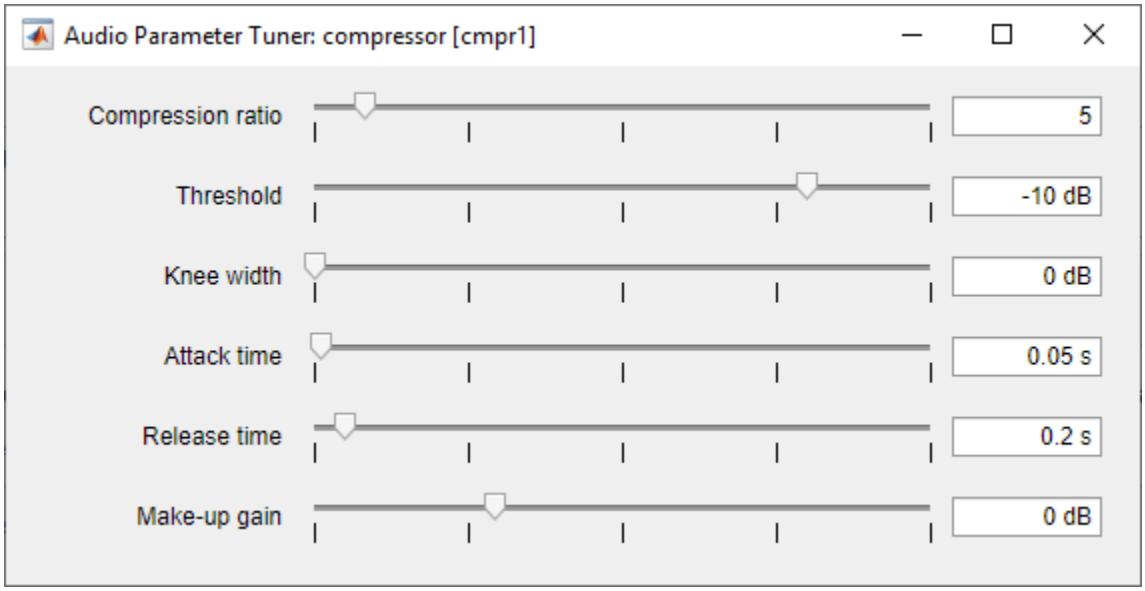




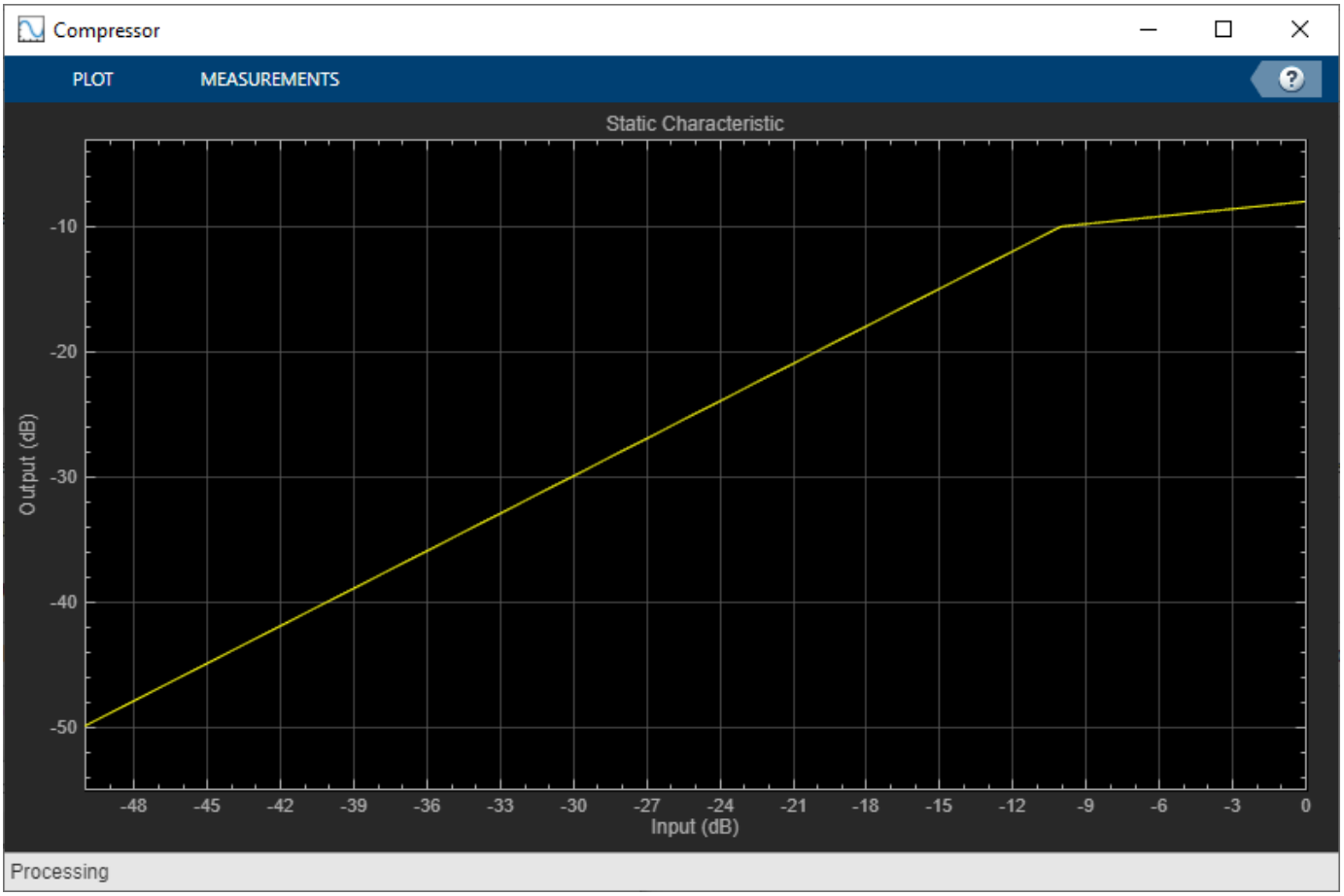
Create two `compressor` objects to apply dynamic range compression on two of the subbands. Call `visualize` to plot the static characteristic of both of the compressors. Call `parameterTuner` to open UIs to tune the static characteristics.

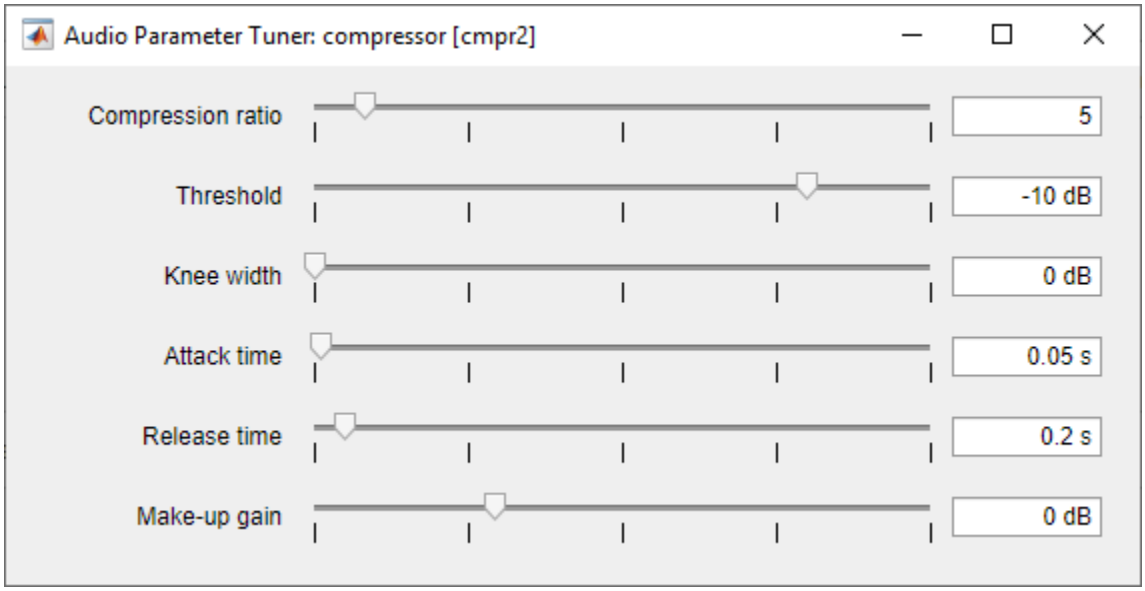
```
cmpr1 = compressor('SampleRate',fileReader.SampleRate);
visualize(cmpr1)
parameterTuner(cmpr1)
```






```
cmpr2 = compressor('SampleRate',fileReader.SampleRate);
visualize(cmpr2)
parameterTuner(cmpr2)
```





Create an `audiopluginexample.Chorus` to apply a chorus effect to one of the bands. Call `parameterTuner` to open a UI to tune the chorus plugin parameters.

```
chorus = audiopluginexample.Chorus;
setSampleRate(chorus,fileReader.SampleRate);
parameterTuner(chorus)
```

In an audio stream loop:

1. Read in a frame of audio from the file.
2. Split the audio into three bands using the crossover filter.
3. Apply dynamic range compression to the first and second bands.
4. Apply a chorus effect to the third band.
5. Sum the audio bands.
6. Write the frame of audio to your audio device for listening.

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

    [b1,b2,b3] = xFilt(audioIn);

    b1 = cmpr1(b1);
    b2 = cmpr2(b2);
    b3 = process(chorus,b3);

    audioOut = b1+b2+b3;

    deviceWriter(audioOut);

    drawnow limitrate % Process parameterTuner callbacks
end
```

As a best practice, release your objects once done.

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

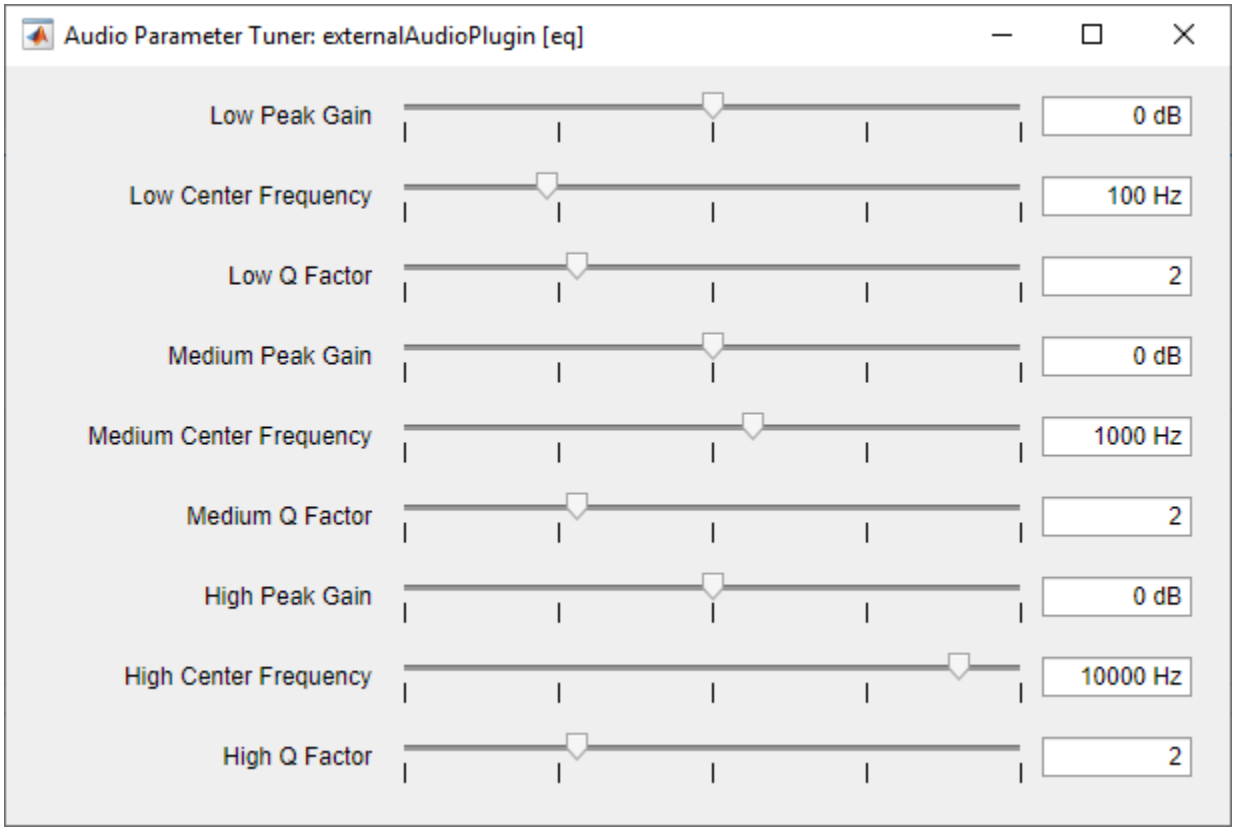
Tune Hosted Audio Plugin Parameters

Create a `dsp.AudioFileReader` to read in audio frame-by-frame. Create an `audioDeviceWriter` to write audio to your sound card. Use `loadAudioPlugin` to load an equalizer plugin. If you are using a Mac, replace the `.dll` file extension with `.vst`.

```
fileReader = dsp.AudioFileReader('FunkyDrums-48-stereo-25secs.mp3');
deviceWriter = audioDeviceWriter('SampleRate',fileReader.SampleRate);

pluginPath = fullfile(matlabroot,'toolbox/audio/samples/ParametricEqualizer.dll');
eq = loadAudioPlugin(pluginPath);
setSampleRate(eq,fileReader.SampleRate);
```

Call `parameterTuner` to open a UI to tune parameters of the equalizer while streaming.

```
parameterTuner(eq)
```



In an audio stream loop:

1. Read in a frame of audio from the file.
2. Apply equalization.
3. Write the frame of audio to your audio device for listening.

```
while ~isDone(fileReader)
    audioIn = fileReader();
    audioOut = process(eq,audioIn);
    deviceWriter(audioOut);

    drawnow limitrate % Process parameterTuner callbacks
end
```

As a best practice, release your objects once done.

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

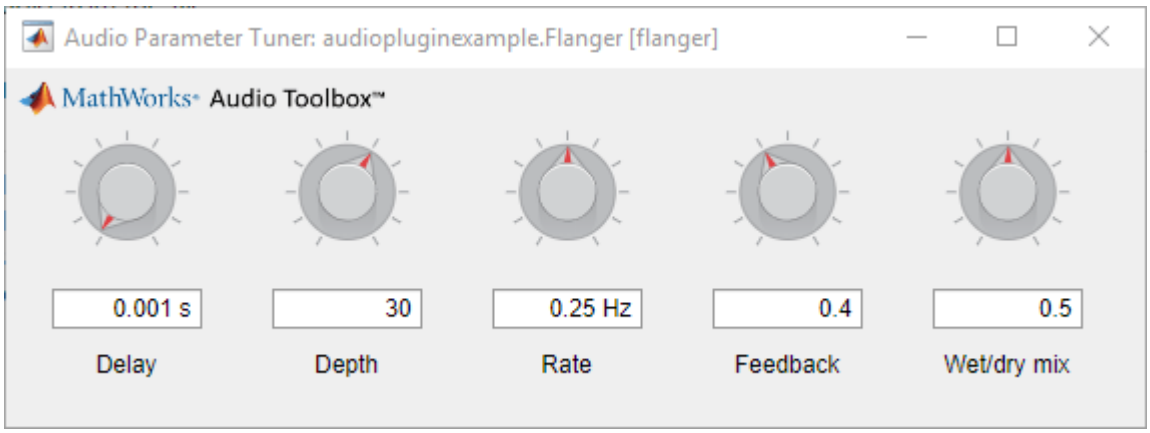
Tune MATLAB Audio Plugin Parameters

Create a `dsp.AudioFileReader` to read in audio frame-by-frame. Create an `audioDeviceWriter` to write audio to your sound card. Create an `audiopluginexample.Flanger` to process the audio data and set the sample rate.

```
fileReader = dsp.AudioFileReader('RockGuitar-16-96-stereo-72secs.flac');
deviceWriter = audioDeviceWriter('SampleRate',fileReader.SampleRate);

flanger = audiopluginexample.Flanger;
setSampleRate(flanger,fileReader.SampleRate);
```

Call `parameterTuner` to open a UI to tune parameters of the flanger while streaming.

```
parameterTuner(flanger)
```



In an audio stream loop:

1. Read in a frame of audio from the file.
2. Apply flanging.
3. Write the frame of audio to your audio device for listening.

```
while ~isDone(fileReader)
    audioIn = fileReader();
    audioOut = process(flanger,audioIn);
    deviceWriter(audioOut);

    drawnow limitrate % Process parameterTuner callbacks
end
```

As a best practice, release your objects once done.

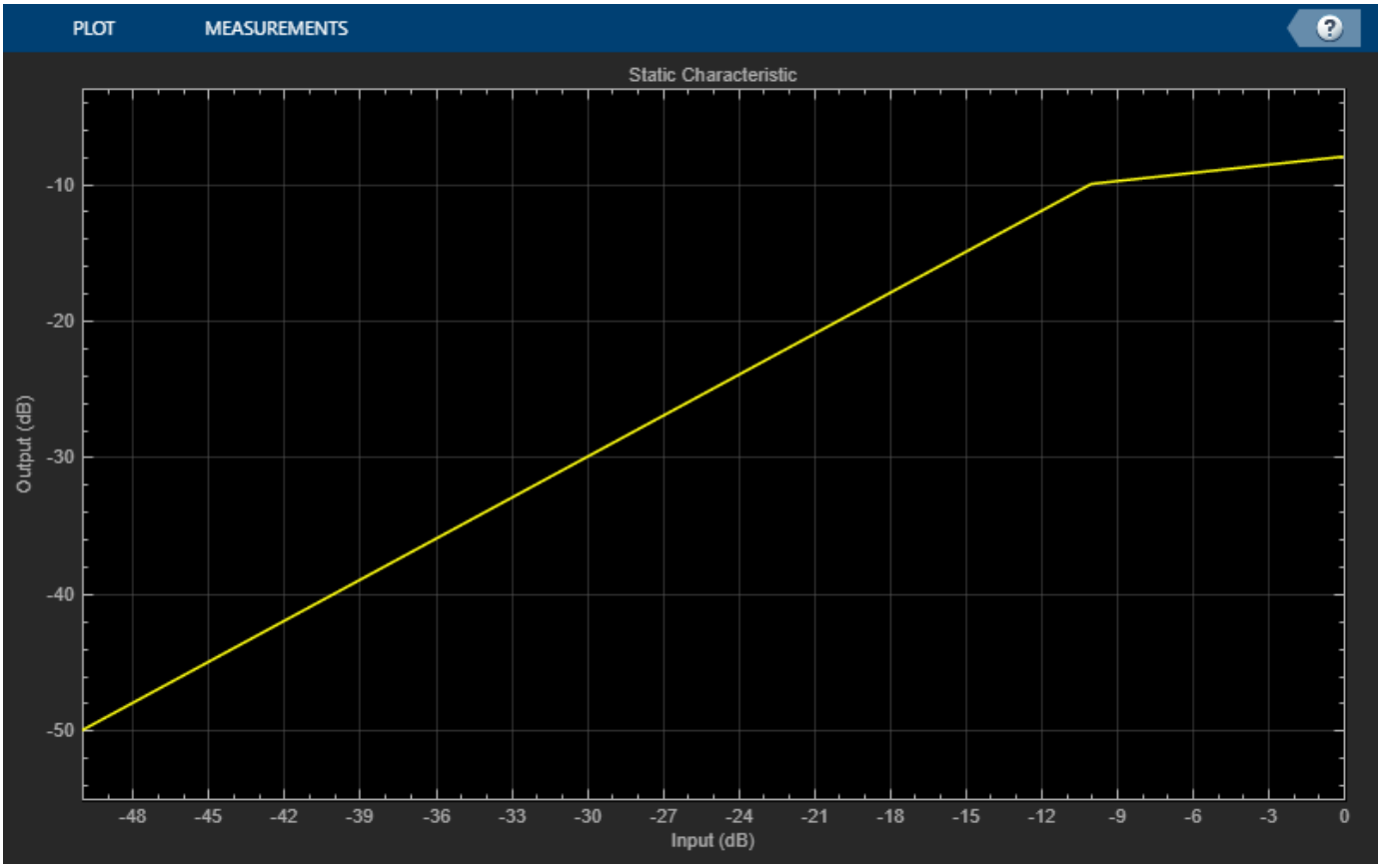
```
release(fileReader)
release(deviceWriter)
```

Tune Compressor Parameters

Create a `dsp.AudioFileReader` to read in audio frame-by-frame. Create an `audioDeviceWriter` to write audio to your sound card. Create a `compressor` to process the audio data. Call `visualize` to plot the static characteristic of the `compressor`.

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

```
dRC = compressor('SampleRate',fileReader.SampleRate);
visualize(dRC)
```

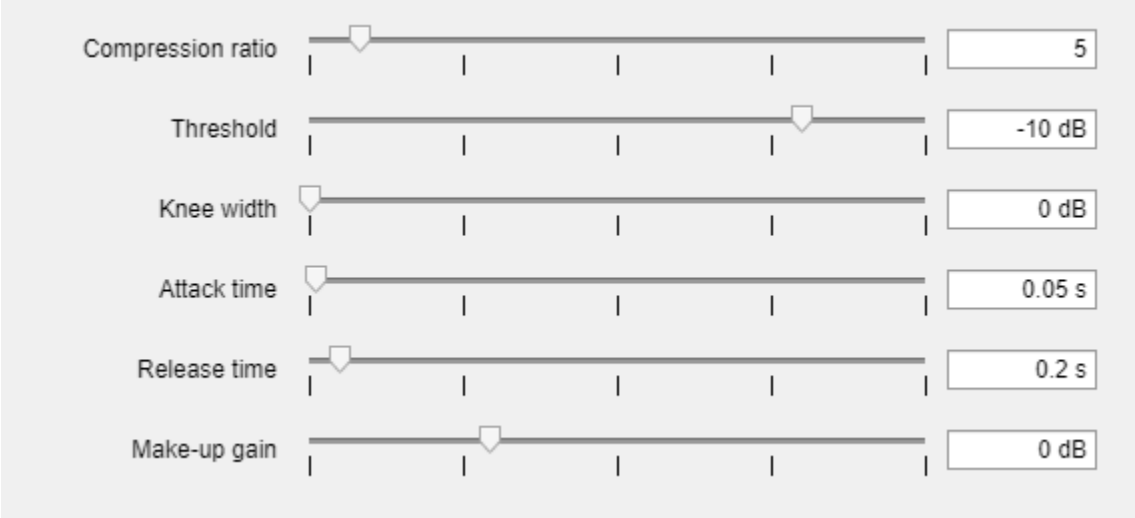


Create a `timescope` to visualize the original and processed audio.

```
scope = timescope( ...
    'SampleRate',fileReader.SampleRate, ...
    'TimeSpanSource','property',...
    'TimeSpan',1, ...
    'BufferLength',fileReader.SampleRate*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)';
```

Call `parameterTuner` to open a UI to tune parameters of the compressor while streaming.

```
parameterTuner(dRC)
```



In an audio stream loop:

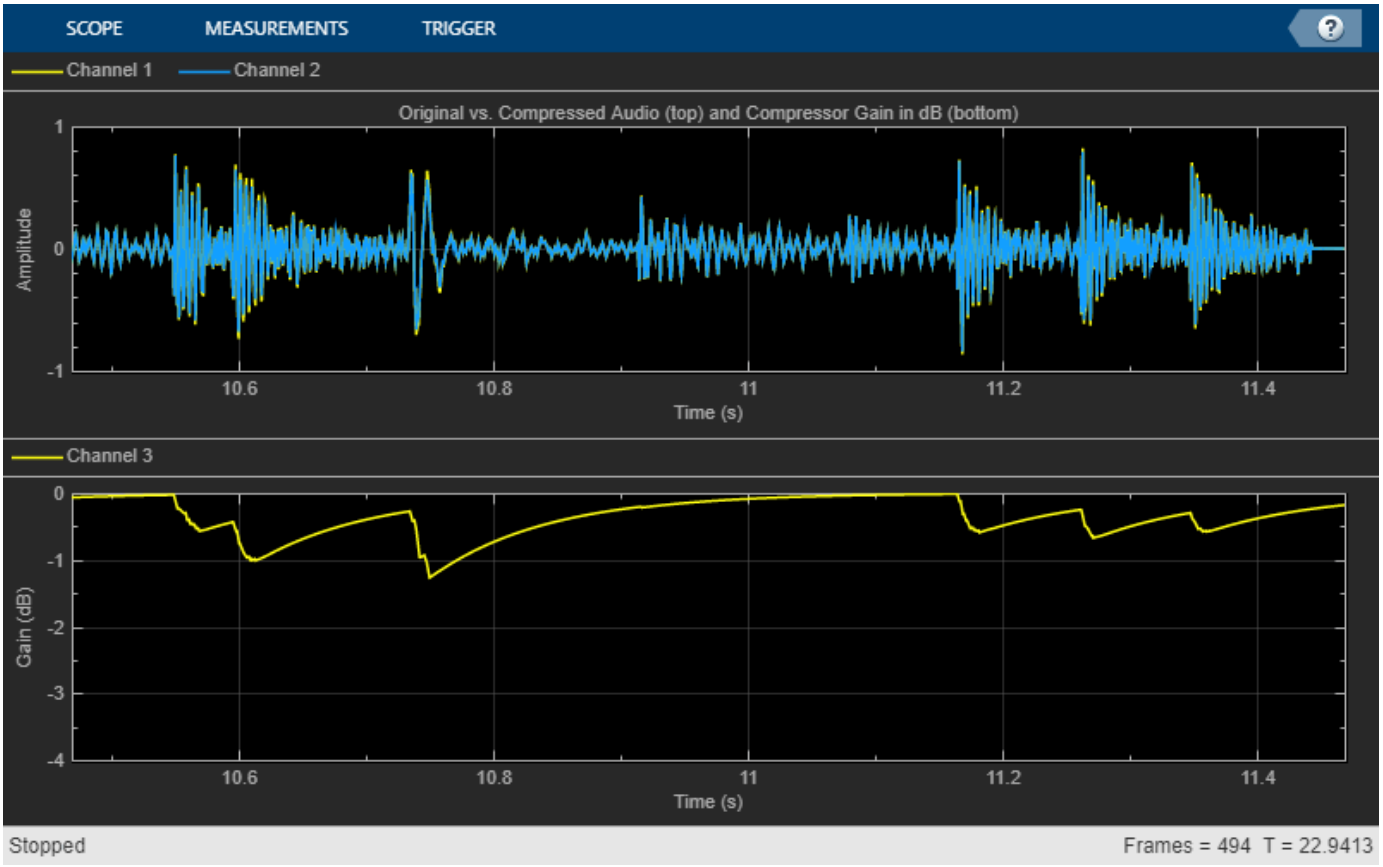
1. Read in a frame of audio from the file.
2. Apply dynamic range compression.
3. Write the frame of audio to your audio device for listening.
4. Visualize the original audio, the processed audio, and the gain applied.

While streaming, tune parameters of the dynamic range compressor and listen to the effect.

```
while ~isDone(fileReader)
    audioIn = fileReader();
    [audioOut,g] = dRC(audioIn);
    deviceWriter(audioOut);
    scope([audioIn(:,1),audioOut(:,1)],g(:,1));
    drawnow limitrate % required to update parameter
end
```

As a best practice, release your objects once done.

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



Input Arguments

`obj` — Object to tune

`audioPlugin` object | `compressor` | `expander` | `limiter` | `noiseGate` | `octaveFilter` | `crossoverFilter` | `multibandParametericEQ` | `graphicEQ` | `audioOscillator` | `wavetableSynthesizer` | `reverberator` | `shelvingFilter`

Object to tune, specified as an object that inherits from `audioPlugin` or one of the following Audio Toolbox objects:

- `compressor`
- `expander`
- `limiter`
- `noiseGate`
- `octaveFilter`
- `crossoverFilter`
- `multibandParametricEQ`
- `graphicEQ`
- `audioOscillator`
- `wavetableSynthesizer`

- `reverberator`
- `shelvingFilter`

Output Arguments

`H` — Target figure

`Figure` object

Target figure, returned as a `Figure` object.

Version History

Introduced in R2019a

See Also

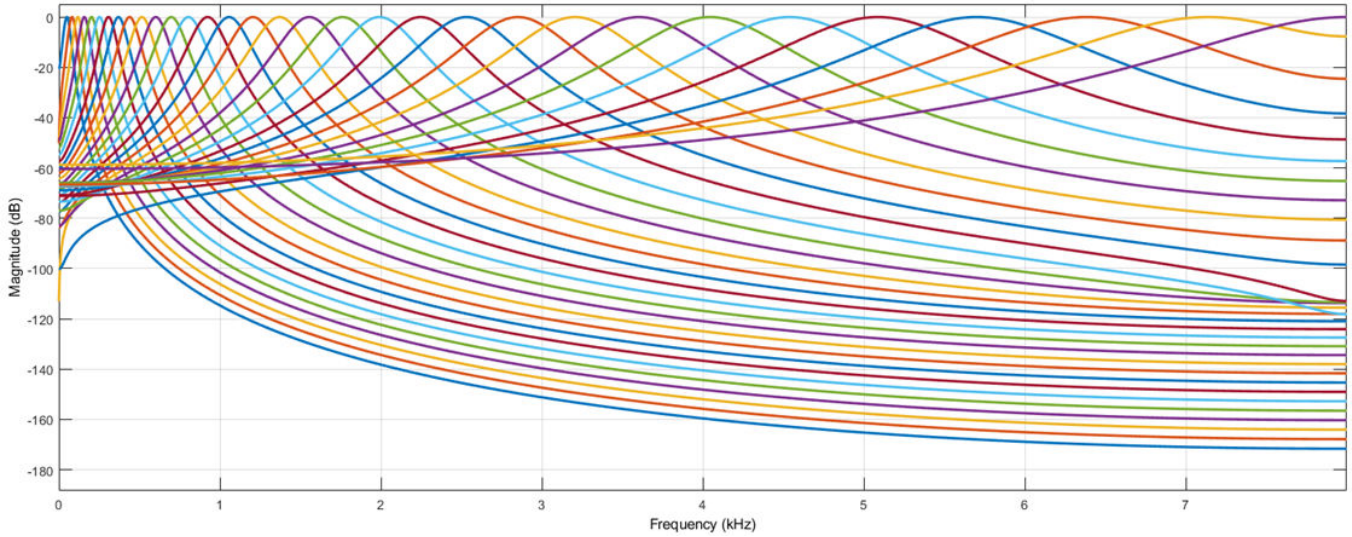
Audio Test Bench | `audioPlugin`

gammatoneFilterBank

Gammatone filter bank

Description

`gammatoneFilterBank` decomposes a signal by passing it through a bank of gammatone filters equally spaced on the ERB scale. Gammatone filter banks were designed to model the human auditory system.



To model the human auditory system:

1. Create the `gammatoneFilterBank` object and set its properties.
2. Call the object with arguments, as if it were a function.

To learn more about how System objects work, see What Are System Objects?

Creation

Syntax

```
gammaFiltBank = gammatoneFilterBank
gammaFiltBank = gammatoneFilterBank(range)
gammaFiltBank = gammatoneFilterBank(range,numFilts)
gammaFiltBank = gammatoneFilterBank(range,numFilts,fs)
gammaFiltBank = gammatoneFilterBank(___,Name,Value)
```

Description

`gammaFiltBank = gammatoneFilterBank` returns a gammatone filter bank. The object filters data independently across each input channel over time.

`gammaFiltBank = gammatoneFilterBank(range)` sets the `Range` property to `range`.

`gammaFiltBank = gammatoneFilterBank(range,numFilts)` sets the `NumFilters` property to `numFilts`.

`gammaFiltBank = gammatoneFilterBank(range,numFilts,fs)` sets the `SampleRate` property to `fs`.

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

Example: `gammaFiltBank = gammatoneFilterBank([62.5,12e3],'SampleRate',24e3)` creates a gammatone filter bank, `gammaFiltBank`, with bandpass filters placed between 62.5 Hz and 12 kHz. `gammaFiltBank` operates at a sample rate of 24 kHz.

Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see System Design in MATLAB Using System Objects.

`FrequencyRange` — Frequency range of filter bank (Hz)

`[50 8000]` (default) | two-element row vector of monotonically increasing values

Frequency range of the filter bank in Hz, specified as a two-element row vector of monotonically increasing values.

Tunable: No

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

`NumFilters` — Number of filters

`32` (default) | positive integer scalar

Number of filters in the filter bank, specified as a positive integer scalar.

Tunable: No

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

`SampleRate` — Input sample rate (Hz)

`16000` (default) | positive scalar

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

Tunable: Yes

Data Types: `single` | `double`

Usage

Syntax

```
audioOut = gammaFiltBank(audioIn)
```

Description

`audioOut = gammaFiltBank(audioIn)` applies the gammatone filter bank on the input and returns the filtered output.

Input Arguments

`audioIn` — Audio input to filter bank

scalar | vector | matrix

Audio input to the filter bank, specified as a scalar, vector, or matrix. If specified as a matrix, the columns are treated as independent audio channels.

Data Types: `single` | `double`

Output Arguments

`audioOut` — Audio output from filter bank

scalar | vector | matrix | 3-D array

Audio output from the filter bank, returned as a scalar, vector, matrix, or 3-D array. The shape of `audioOut` depends on the shape of `audioIn` and `NumFilters`. If `audioIn` is an *M*-by-*N* matrix, then `audioOut` is returned as an *M*-by-`NumFilters`-by-*N* array. If *N* is 1, then `audioOut` is returned as a matrix.

Data Types: `single` | `double`

Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

Specific to `gammatoneFilterBank`

fvtool	Visualize filter bank
freqz	Compute frequency response
getCenterFrequencies	Center frequencies of filters
getBandwidths	Get filter bandwidths
coeffs	Get filter coefficients
info	Get filter information

Common to All System Objects

step	Run System object algorithm
release	Release resources and allow changes to System object property values and input characteristics

reset Reset internal states of System object

Examples

Apply Gammatone Filter Bank

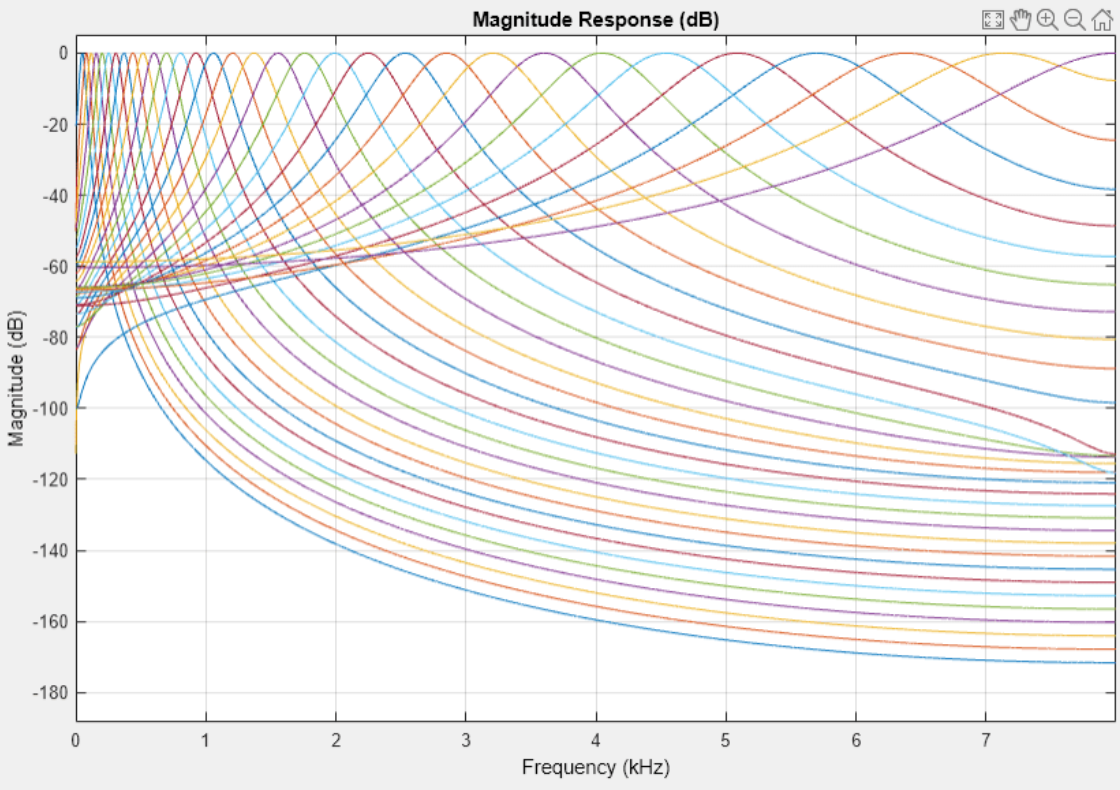
Create a default gammatone filter bank for a 16 kHz sample rate.

```
fs = 16e3;
gammaFiltBank = gammatoneFilterBank(SampleRate=fs)
```

```
gammaFiltBank =

  gammatoneFilterBank with properties:

    FrequencyRange: [50 8000]
        NumFilters: 32
        SampleRate: 16000
```

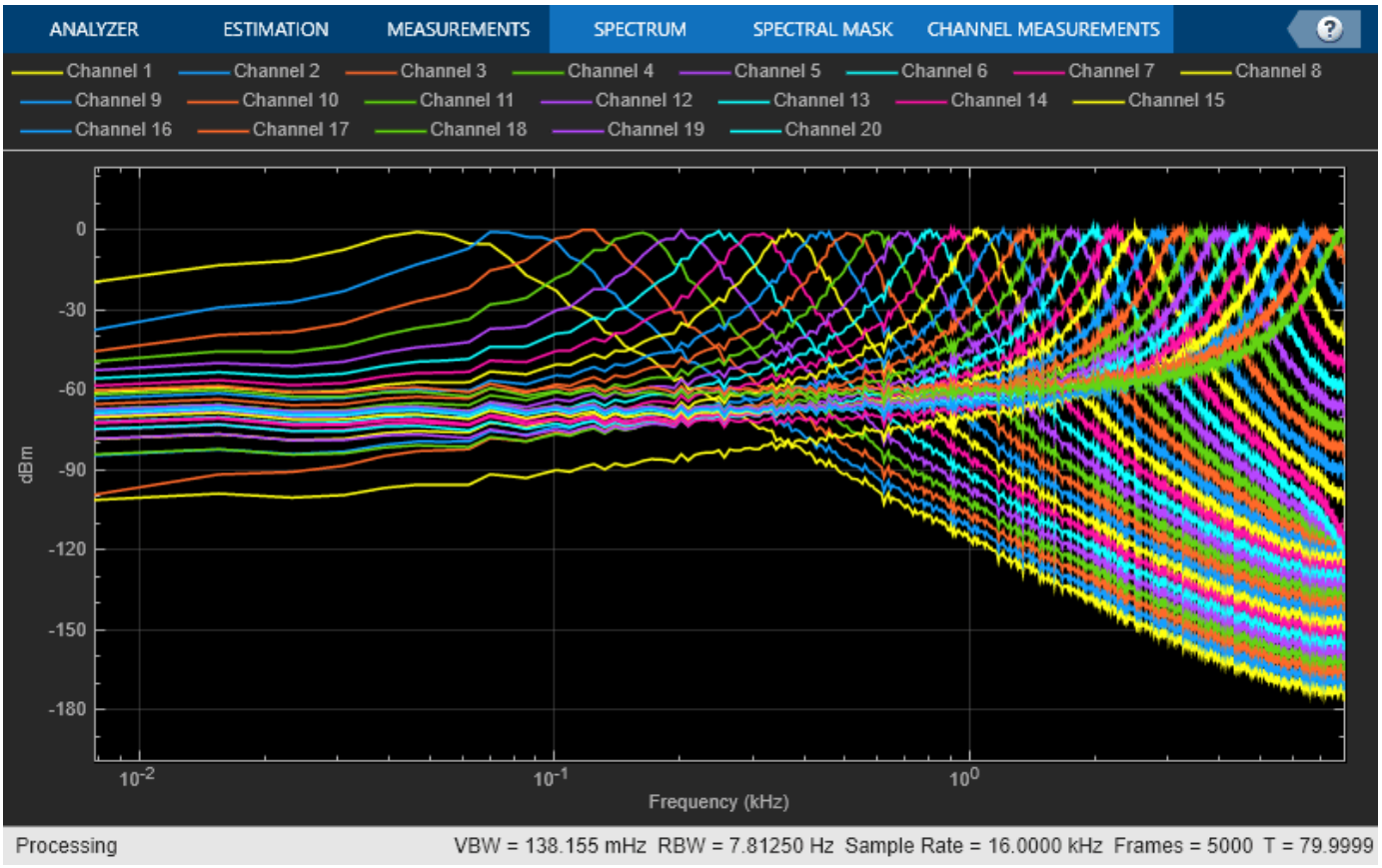
Use `fvtool` to visualize the response of the filter bank.

```
fvtool(gammaFiltBank)
```



Process white Gaussian noise through the filter bank. Use a spectrum analyzer to view the spectrum of the filter outputs.

```
sa = spectrumAnalyzer(SampleRate=16e3,...
    PlotAsTwoSidedSpectrum=false,...
    FrequencyScale="log");

for i = 1:5000
    x = randn(256,1);
    y = gammaFiltBank(x);
    sa(y);
end
```



Analysis and Synthesis

This example illustrates a nonoptimal but simple approach to analysis and synthesis using `gammatoneFilterBank`.

Read in an audio file and listen to its contents.

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

Create a default `gammatoneFilterBank`. The default frequency range of the filter bank is 50 to 8000 Hz. Frequencies outside of this range are attenuated in the reconstructed signal.

```
gammaFiltBank = gammatoneFilterBank('SampleRate',fs)

gammaFiltBank =
  gammatoneFilterBank with properties:

    FrequencyRange: [50 8000]
        NumFilters: 32
        SampleRate: 44100
```

Pass the audio signal through the gammatone filter bank. The output is 32 channels, where the number of channels is set by the `NumFilters` property of the `gammatoneFilterBank`.

```
audioOut = gammaFiltBank(audioIn);

[N,numChannels] = size(audioOut)

N = 685056

numChannels = 32
```

To reconstruct the original signal, sum the channels. Listen to the result.

```
reconstructedSignal = sum(audioOut,2);
sound(reconstructedSignal,fs)
```

The gammatone filter bank introduced various group delays for the output channels, which results in poor reconstruction. To compensate for the group delay, remove the beginning delay from the individual channels and zero-pad the ends of the channels. Use `info` to get the group delays. Listen to the group delay-compensated reconstruction.

```
infoStruct = info(gammaFiltBank);
groupDelay = round(infoStruct.GroupDelays); % round for simplicity

audioPadded = [audioOut;zeros(max(groupDelay),gammaFiltBank.NumFilters)];

for i = 1:gammaFiltBank.NumFilters
    audioOut(:,i) = audioPadded(groupDelay(i)+1:N+groupDelay(i),i);
end

reconstructedSignal = sum(audioOut,2);
sound(reconstructedSignal,fs)
```

Create Gammatone Spectrogram

Read in an audio signal and convert it to mono for easy visualization.

```
[audio,fs] = audioread('WaveGuideLoopOne-24-96-stereo-10secs.aif');
audio = mean(audio,2);
```

Create a `gammatoneFilterBank` with 64 filters that span the range 62.5 to 20,000 Hz. Pass the audio signal through the filter bank.

```
gammaFiltBank = gammatoneFilterBank('SampleRate',fs, ...
                                    'NumFilters',64, ...
                                    'FrequencyRange',[62.5,20e3]);
```

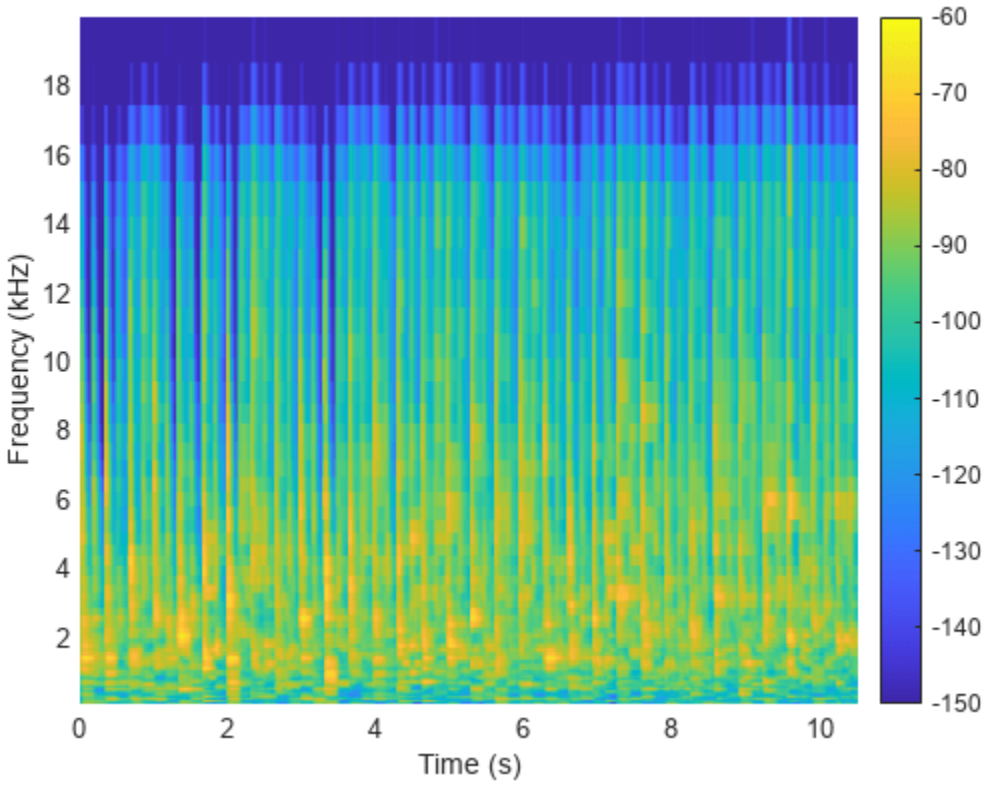
```
audioOut = gammaFiltBank(audio);
```

Calculate the energy-per-band using 50 ms windows with 25 ms overlap. Use `dsp.AsyncBuffer` to divide the signals into overlapped windows and then to log the RMS value of each window for each channel.

```
samplesPerFrame = round(0.05*fs);
samplesOverlap = round(0.025*fs);

buff = dsp.AsyncBuffer(numel(audio));
write(buff,audioOut.^2);

sink = dsp.AsyncBuffer(numel(audio));

while buff.NumUnreadSamples > 0
    currentFrame = read(buff,samplesPerFrame,samplesOverlap);
    write(sink,mean(currentFrame,1));
end
```

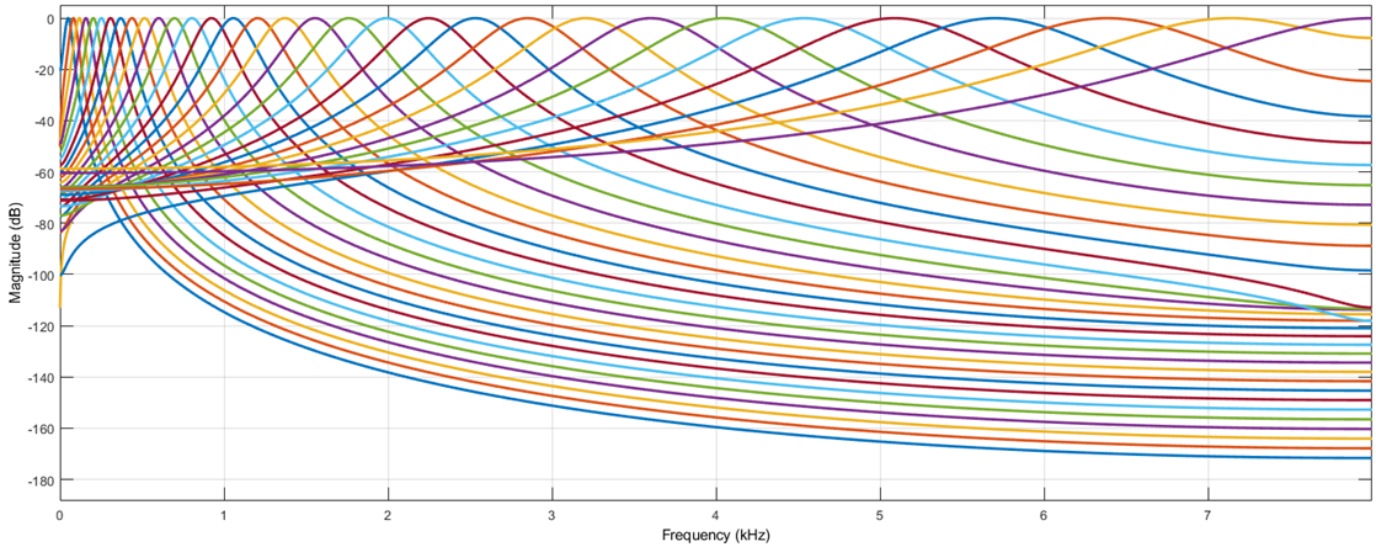
Convert the energy values to dB. Plot the energy-per-band over time.

```
gammatoneSpec = read(sink);
D = 20*log10(gammatoneSpec');

timeVector = ((samplesPerFrame-samplesOverlap)/fs)*(0:size(D,2)-1);
cf = getCenterFrequencies(gammaFiltBank)./1e3;

surf(timeVector,cf,D,'EdgeColor','none')
axis([timeVector(1) timeVector(end) cf(1) cf(end)])
view([0 90])
caxis([-150,-60])
colorbar
xlabel('Time (s)')
ylabel('Frequency (kHz)')
```



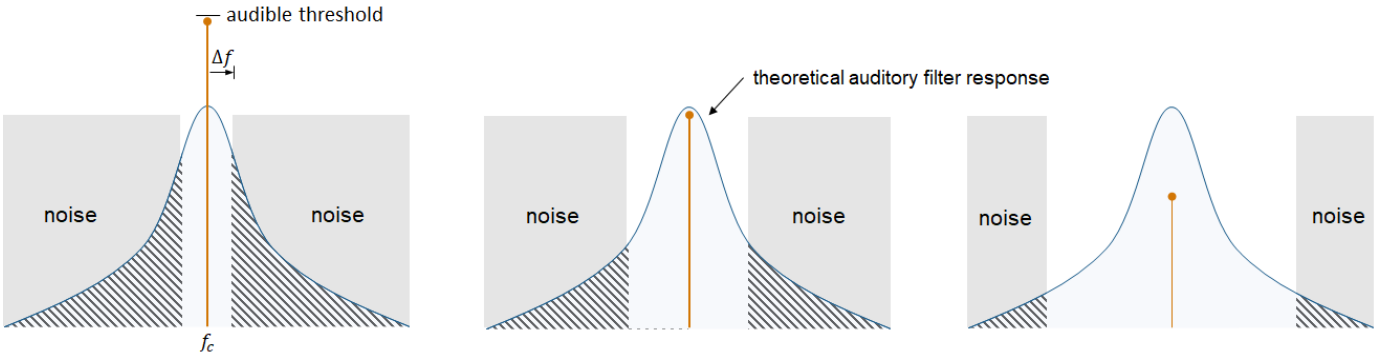
Algorithms

A gammatone filter bank is often used as the front end of a cochlea simulation, which transforms complex sounds into a multichannel activity pattern like that observed in the auditory nerve.[2] The `gammatoneFilterBank` follows the algorithm described in [1]. The algorithm is an implementation of an idea proposed in [2]. The design of the gammatone filter bank can be described in two parts: the filter shape (gammatone) and the frequency scale. The equivalent rectangular bandwidth (ERB) scale defines the relative spacing and bandwidth of the gammatone filters. The derivation of the ERB scale also provides an estimate of the auditory filter response which closely resembles the gammatone filter.

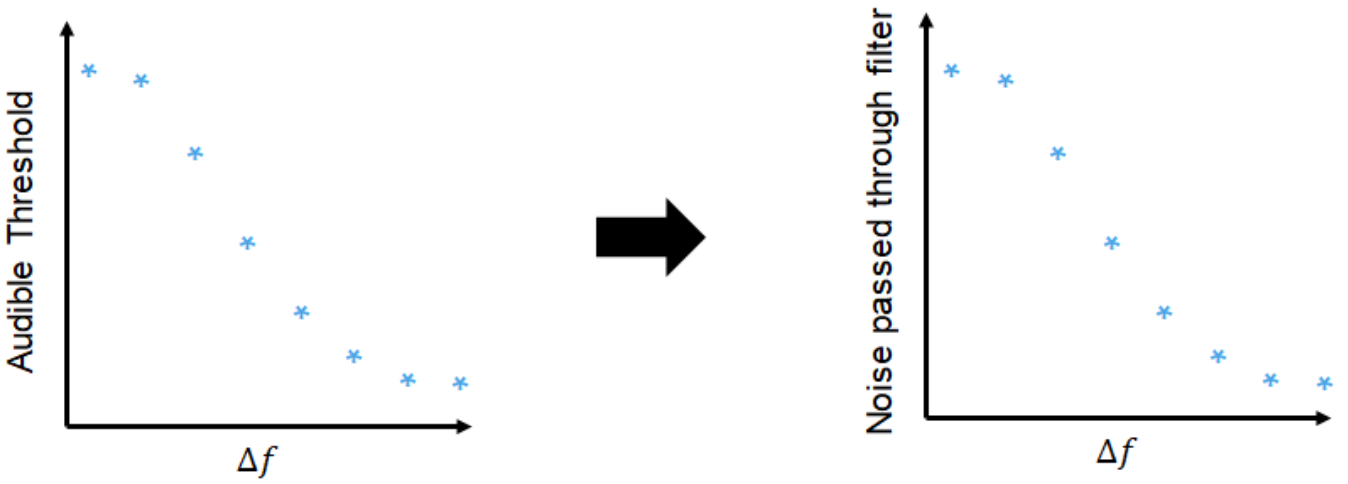


Frequency Scale

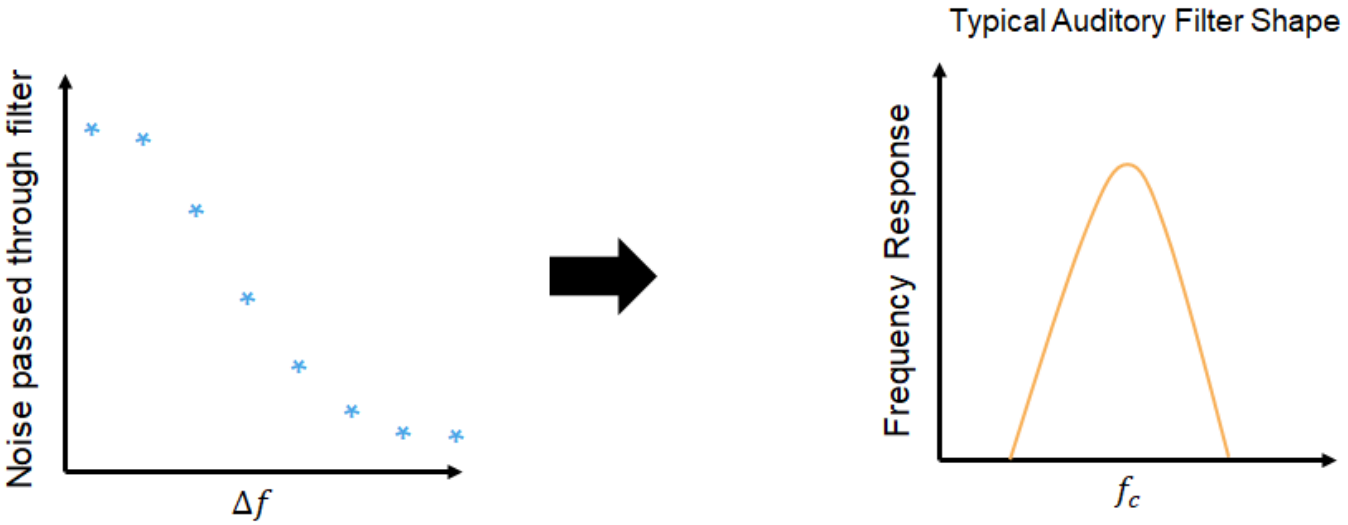
The ERB scale was determined using the notched-noise masking method. This method involves a listening test wherein notched noise is centered on a tone. The power of the tone is tuned, and the audible threshold (the power required for the tone to be heard) is recorded. The experiment is repeated for different notch widths and center frequencies.



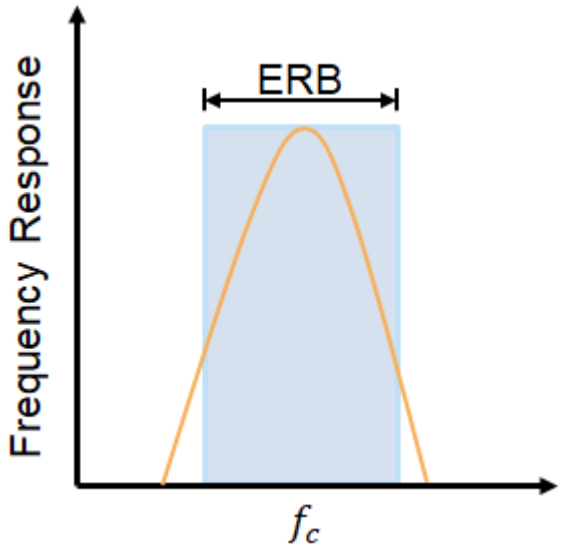
The notched-noise method assumes the audible threshold corresponds to a constant signal-to-masker ratio at the output of the theoretical auditory filter. That is, the ratio of the power of the f_c tone and the shaded area is constant. Therefore, the relationship between the audible threshold and $2\Delta f$ (the notch bandwidth) is linearly related to the relationship between the noise passed through the filter and $2\Delta f$.



The derivative of the function relating Δf to the noise passed through the filter estimates the auditory filter shape. Because Δf has an inverse relationship with the noise power passed through the filter, the derivative of the function must be multiplied by –1. The resulting auditory filter shape is usually approximated as a roex filter.



The equivalent rectangular bandwidth of the auditory filter is defined as the width of a rectangular filter required to pass the same noise power as the auditory filter.



[4] defines ERB as a function of center frequency for young listeners with normal hearing and a moderate noise level:

$$\text{ERB} = 24.7(0.00437f_c + 1)$$

The ERB scale (ERBs) is an extension of the relationship between ERB and center frequency, derived by integrating the reciprocal of the ERB function:

$$\text{ERBs} = 21.4\log_{10}(0.00437f + 1)$$

To design a gammatone filter bank, [2] suggests distributing the center frequencies of the filters in proportion to their bandwidth. To accomplish this, `gammatoneFilterBank` defines the center frequencies as linearly spaced on the ERB scale, covering the specified frequency range with the desired number of filters. You can specify the frequency range and desired number of filters using the `FrequencyRange` and `NumFilters` properties.

Gammatone Filter

The gammatone filter was introduced in [3]. The continuous impulse response is:

$$g(t) = at^{n-1}e^{-2\pi bt}\cos(2\pi f_c t + \phi)$$

where

- a –- amplitude factor
- t –- time in seconds
- n –- filter order (set to four to model human hearing)
- f_c–- center frequency
- b –- bandwidth, set to 1.019*`hz2erb(`f_c`)`.
- ϕ –- phase factor

The gammatone filter is similar to the roex filter derived from the notched-noise experiment. `gammatoneFilterBank` implements the digital filter as a cascade of four second-order sections, as described in [1].

Version History

Introduced in R2019a

References

[1] Slaney, Malcolm. "An Efficient Implementation of the Patterson-Holdsworth Auditory Filter Bank." Apple Computer Technical Report 35, 1993.

[2] Patterson, R. D., K. Robinson, J. Holdsworth, D. McKeown, C. Zhang, and M. Allerhand. "Complex Sounds and Auditory Images." *Auditory Physiology and Perception*. 1992, pp. 429–446.

[3] Aertsen, A. M. H. J., and P. I. M. Johannesma. "Spectro-temporal Receptive Fields of Auditory Neurons in the Grassfrog." *Biological Cybernetics*. Vol. 38, Issue 4, 1980, pp. 223–234.

[4] Glasberg, Brian R., and Brian C. J. Moore. "Derivation of Auditory Filter Shapes from Notched-Noise Data." *Hearing Research*. Vol. 47. Issue 1-2, 1990, pp. 103 –138.

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

`octaveFilterBank` | `crossoverFilter`

coeffs

Get filter coefficients

Syntax

```
[B,A] = coeffs(obj)
```

Description

`[B,A] = coeffs(obj)` returns the coefficients of the filters created by `obj`.

Examples

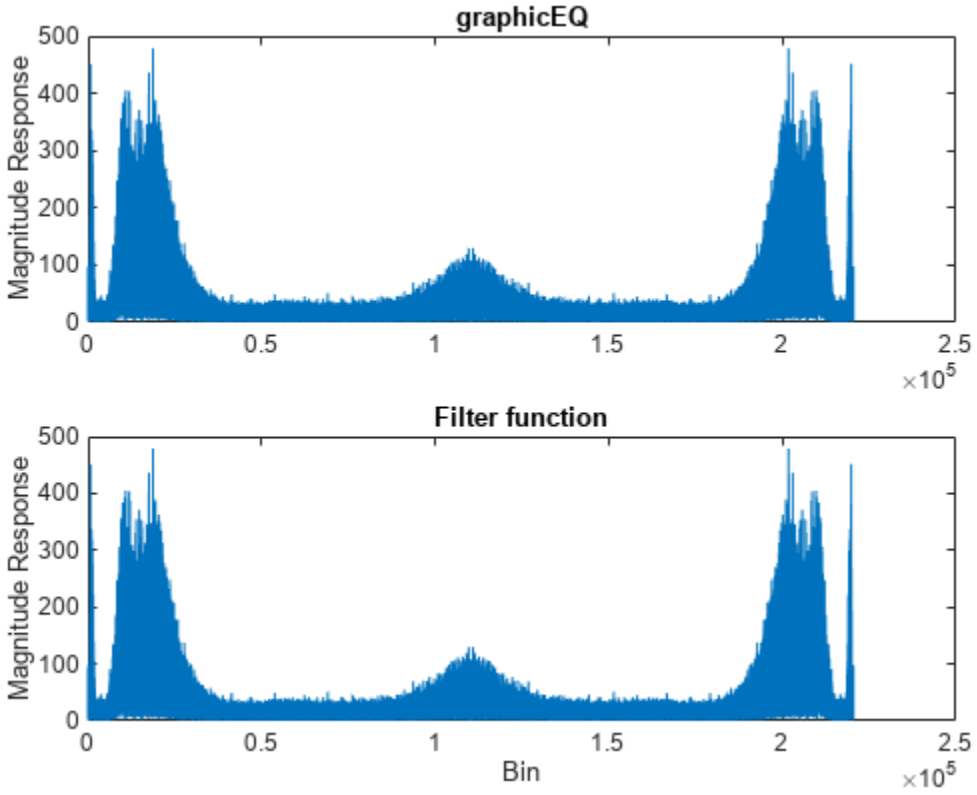
Get `graphicEQ` Coefficients

Create a `graphicEQ` and then call `coeffs` to get its coefficients. The coefficients are returned as second-order sections. The dimensions of `B` are 3-by-(`M` * `EQOrder` / 2), where `M` is the number of bandpass equalizers. The dimensions of `A` are 2-by-(`M` * `EQOrder` / 2).

```
fs = 44.1e3;
x = 0.1*randn(fs*5,1);
equalizer = graphicEQ('SampleRate',fs, ...
                      'Gains',[-10,-10,10,10,-10,-10,10,10,-10,-10], ...
                      'EQOrder',2);

[B,A] = coeffs(equalizer);
```

Compare the output of the `filter` function using coefficients `B` and `A` with the output of `graphicEQ`.

```
y = x;
for section = 1:equalizer.EQOrder/2
     for i = 1:numel(equalizer.Gains)
        y = filter(B(:,i*section),A(:,i*section),y);
     end
end
audioOut_filter = y;

audioOut = equalizer(x);

subplot(2,1,1)
plot(abs(fft(audioOut)))
title('graphicEQ')
ylabel('Magnitude Response')

subplot(2,1,2)
plot(abs(fft(audioOut_filter)))
title('Filter function')
xlabel('Bin')
ylabel('Magnitude Response')
```



Get gammatoneFilterBank Coefficients

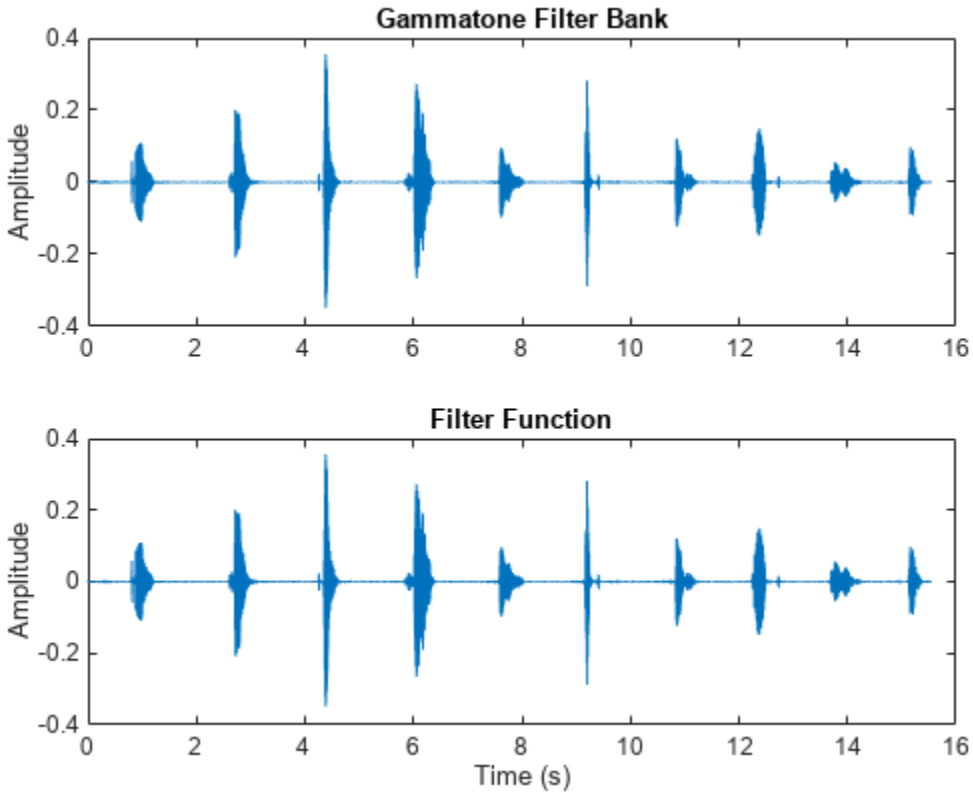
Create the default `gammatoneFilterBank`, and then call `coeffs` to get its coefficients. Each gammatone filter is an eighth-order IIR filter composed of a cascade of four second-order sections. The size of `B` is 4-by-3-by-`NumFilters`. The size of `A` is 4-by-2-by-`NumFilters`.

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

gammaFiltBank = gammatoneFilterBank('SampleRate',fs);

[B,A] = coeffs(gammaFiltBank);
```

Compare the output of the `filter` function using coefficients `B` and `A` with the output of `gammaFiltBank`. For simplicity, compare output from channel eight only.

```
channelToCompare = 8;
y1 = filter(B(1,:,channelToCompare),[1,A(1,:,channelToCompare)],audioIn);
y2 = filter(B(2,:,channelToCompare),[1,A(2,:,channelToCompare)],y1);
y3 = filter(B(3,:,channelToCompare),[1,A(3,:,channelToCompare)],y2);
audioOut_filter = filter(B(4,:,channelToCompare),[1,A(4,:,channelToCompare)],y3);

audioOut = gammaFiltBank(audioIn);

t = (0:(size(audioOut,1)-1))'/fs;
```

```
subplot(2,1,1)
plot(t,audioOut(:,channelToCompare))
title('Gammatone Filter Bank')
ylabel('Amplitude')

subplot(2,1,2)
plot(t,audioOut_filter)
title('Filter Function')
xlabel('Time (s)')
ylabel('Amplitude')
```



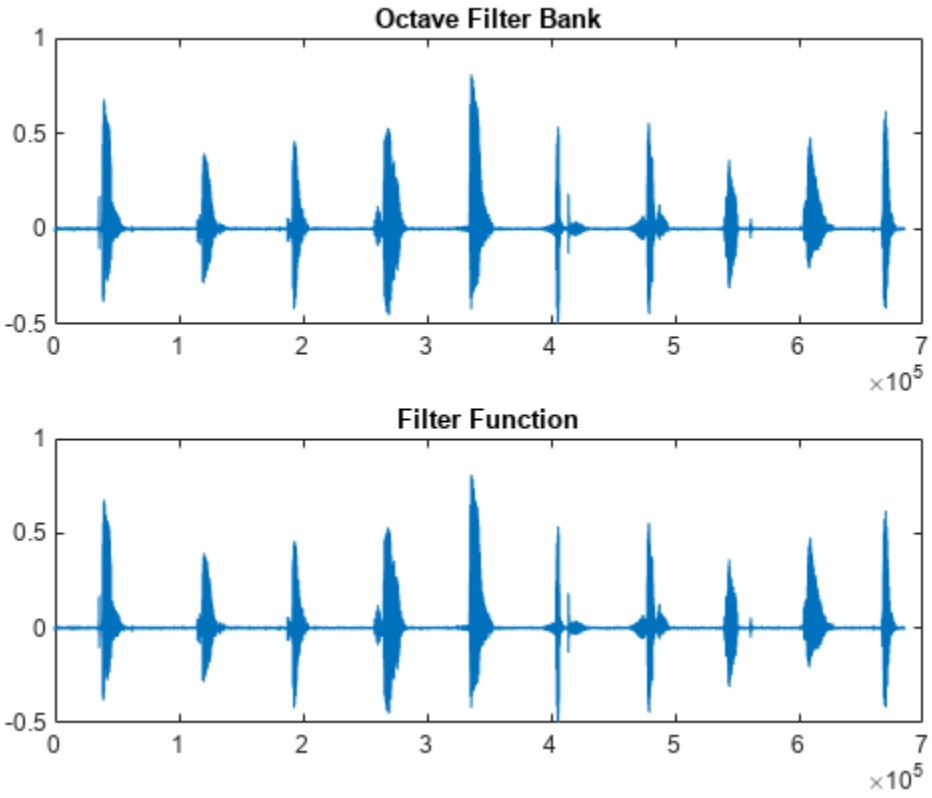
Get `octaveFilterBank` Coefficients

Create the default `octaveFilterBank`, and then call `coeffs` to get its coefficients. The coefficients are returned as second-order sections. The dimensions of B and A are *T*-by-3-by-*M*, where *T* is the number of sections and *M* is the number of filters.

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

octFiltBank = octaveFilterBank('SampleRate',fs);

[B,A] = coeffs(octFiltBank);
```

Compare the output of the `filter` function using coefficients `B` and `A` with the output of `octaveFilterBank`. For simplicity, compare output from channel five only.

```
channelToCompare = 5;

audioOut_filter = filter(B(1,:,channelToCompare),A(1,:,channelToCompare),audioIn);
audioOut = octFiltBank(audioIn);

subplot(2,1,1)
plot(audioOut(:,channelToCompare))
title('Octave Filter Bank')

subplot(2,1,2)
plot(audioOut_filter)
title('Filter Function')
```



Input Arguments

`obj` — Object to get filter coefficients from

`gammatoneFilterBank` | `octaveFilterBank` | `graphicEQ`

Object to get filter coefficients from, specified as an object of `gammatoneFilterBank`, `octaveFilterBank`, `graphicEQ`, or `shelvingFilter`.

Output Arguments

`B` — Numerator filter coefficients

matrix | 3-D array

Numerator filter coefficients, returned as a 2-D matrix or 3-D array, depending on `obj`.

Data Types: `single` | `double`

`A` — Denominator filter coefficients
matrix | 3-D array

Numerator filter coefficients, returned as a 2-D matrix or 3-D array, depending on `obj`.

Data Types: `single` | `double`

Version History

Introduced in R2019a

R2020b: SOS returned instead of FOS from `octaveFilterBank`
Behavior changed in R2020b

The `coeffs` function of `octaveFilterBank` now returns the filter in second-order sections (SOS) instead of fourth-order sections (FOS). This new format reflects an updated internal representation, which has been enhanced to remain stable at very low frequencies.

See Also

`gammatoneFilterBank` | `octaveFilterBank` | `graphicEQ`

freqz

Compute frequency response

Syntax

```
[H,f] = freqz(obj)
[H,f] = freqz(obj,ind)
[H,f] = freqz(___,N=n)
freqz(___)
```

Description

`[H,f] = freqz(obj)` returns a matrix of complex frequency responses for each filter designed by `obj`.

`[H,f] = freqz(obj,ind)` returns the frequency response of filters with indices corresponding to the elements in vector `ind`.

`[H,f] = freqz(___,N=n)` returns the *N*-point complex frequency response.

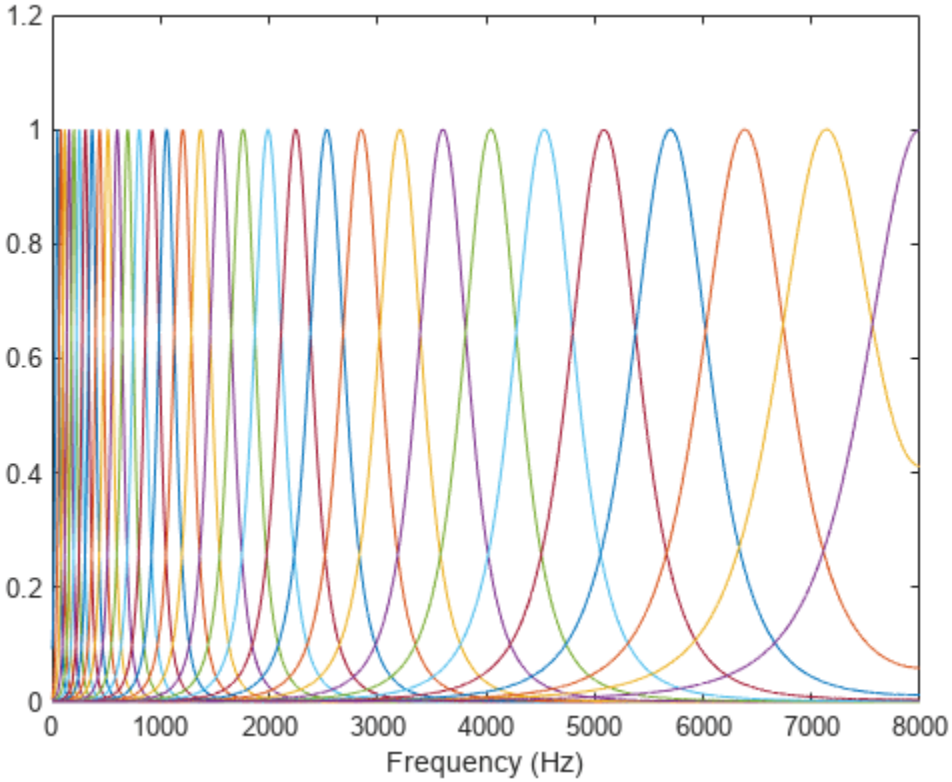
`freqz(___)` with no output arguments plots the frequency response of the filter bank.

Examples

Frequency Response of gammatoneFilterBank

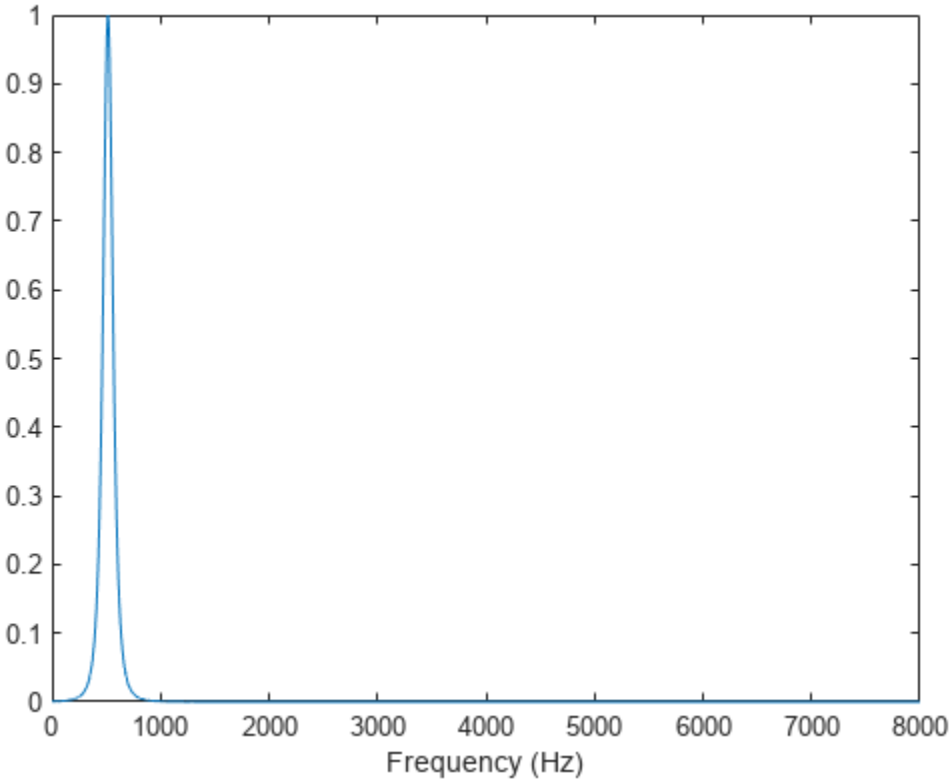
Create a `gammatoneFilterBank` object. Call `freqz` to get the complex frequency response, `H`, of the filter bank and a vector of frequencies, `f`, at which the response is calculated. Plot the magnitude frequency response of the filter bank.

```
gammaFiltBank = gammatoneFilterBank;
[H,f] = freqz(gammaFiltBank);

plot(f,abs(H))
xlabel("Frequency (Hz)")
```



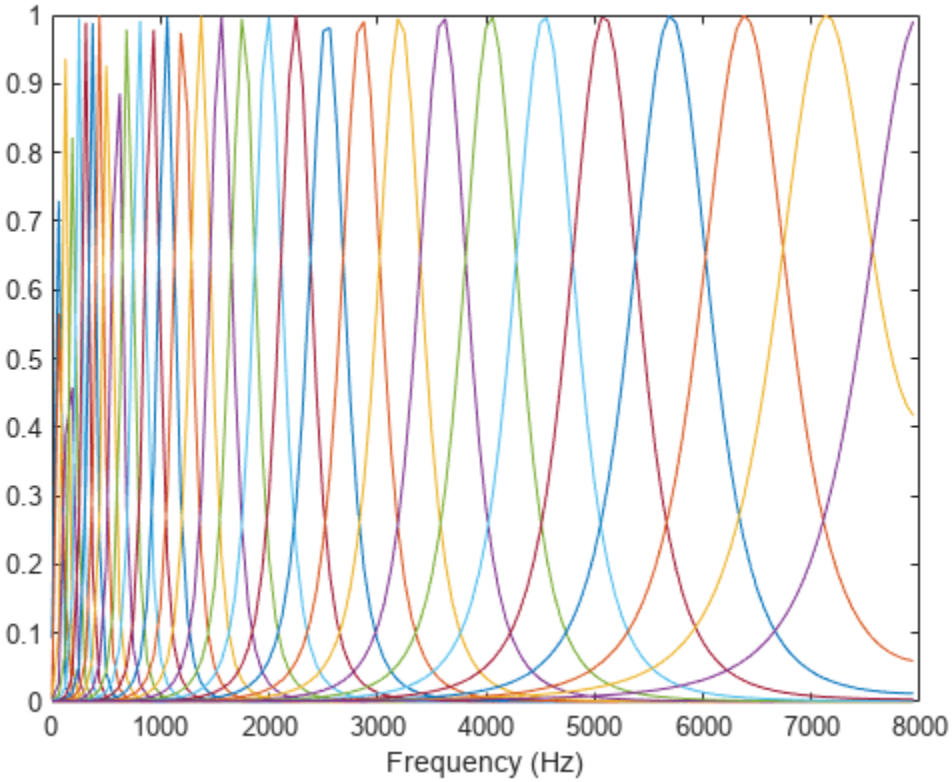
To get the frequency response of a subset of filters in the filter bank, specify the second argument as a row vector of indices between one and the number of filters in the filter bank. Get the frequency response of the 10th filter in the filter bank and plot the magnitude frequency response.

```
[H,f] = freqz(gammaFiltBank,10);

plot(f,abs(H))
xlabel("Frequency (Hz)")
```



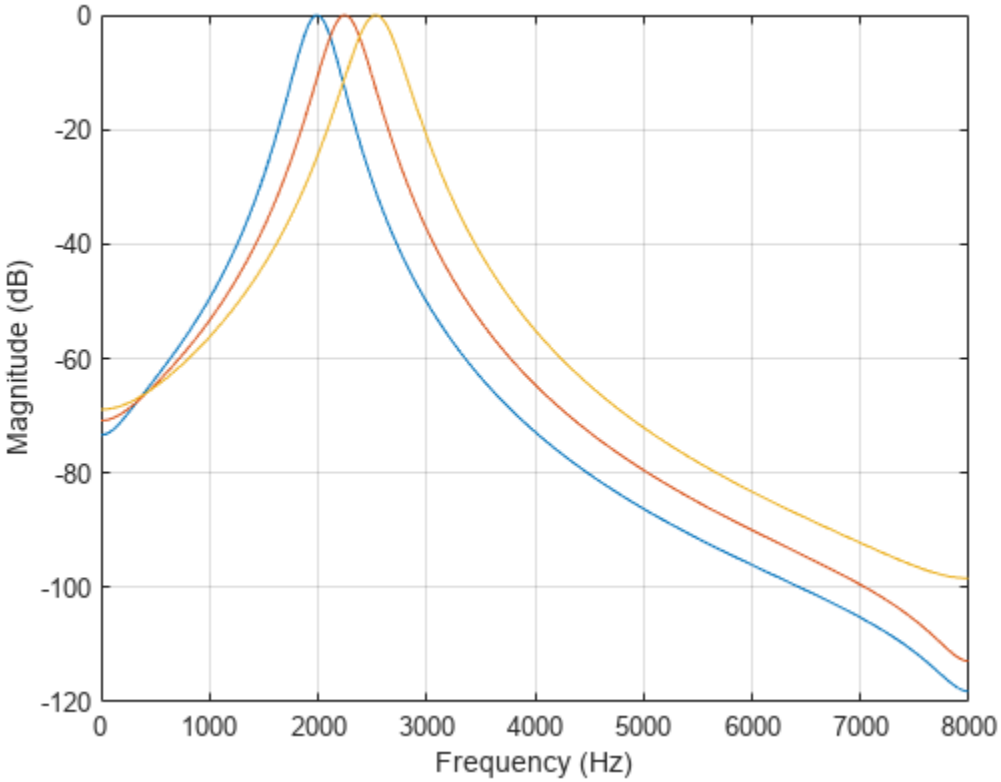
To specify the number of points in the frequency response, use the `N` name-value argument. Specify that the frequency response contains 128 points. Plot the magnitude frequency response.

```
[H,f] = freqz(gammaFiltBank,N=128);

plot(f,abs(H))
xlabel("Frequency (Hz)")
```



To visualize the magnitude frequency response only, call `freqz` without any output arguments. Plot the magnitude frequency response, in dB, of filters 20, 21, and 22 with 1024 points.

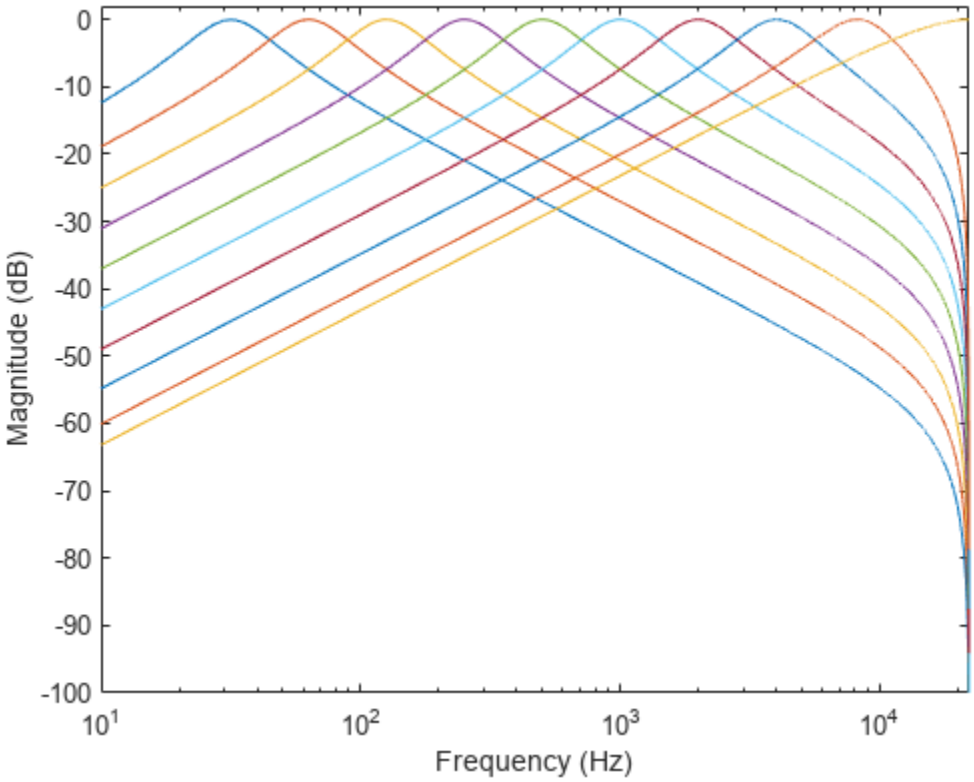
```
freqz(gammaFiltBank,[20,21,22],N=1024)
```



Frequency Response of `octaveFilterBank`

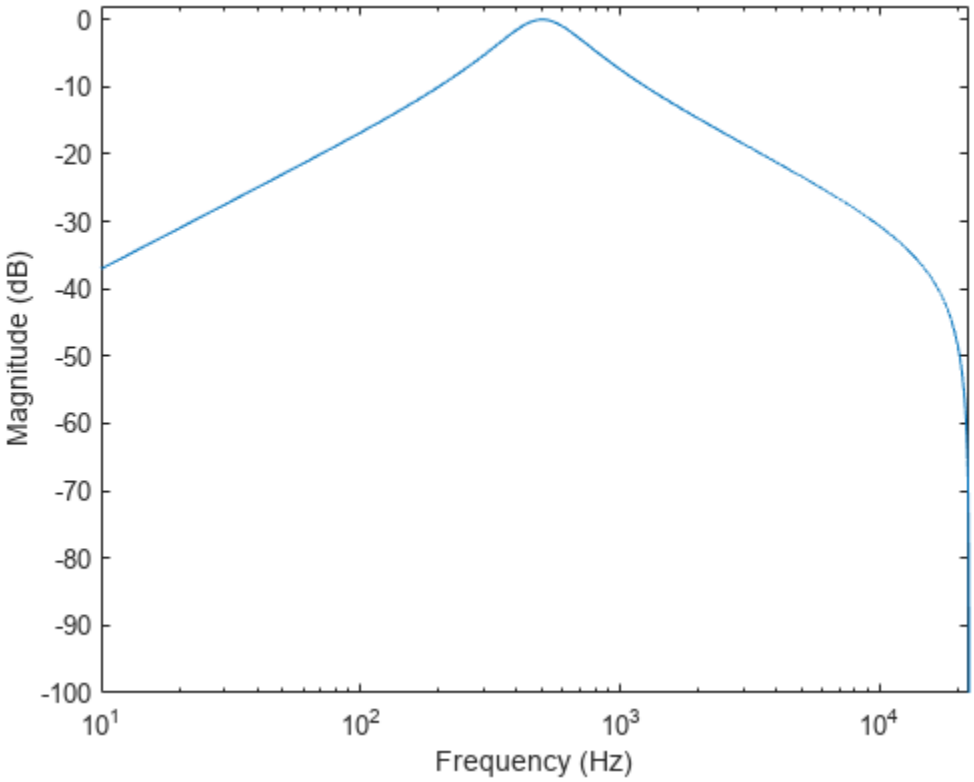
Create an `octaveFilterBank` object. Call `freqz` to get the complex frequency response, `H`, of the filter bank and a vector of frequencies, `f`, at which the response is calculated. Plot the magnitude frequency response in dB.

```
octFiltBank = octaveFilterBank;
[H,f] = freqz(octFiltBank);

plot(f,20*log10(abs(H)))
xlabel("Frequency (Hz)")
ylabel("Magnitude (dB)")
set(gca,XScale="log")
axis([10 octFiltBank.SampleRate/2 -100 2])
```



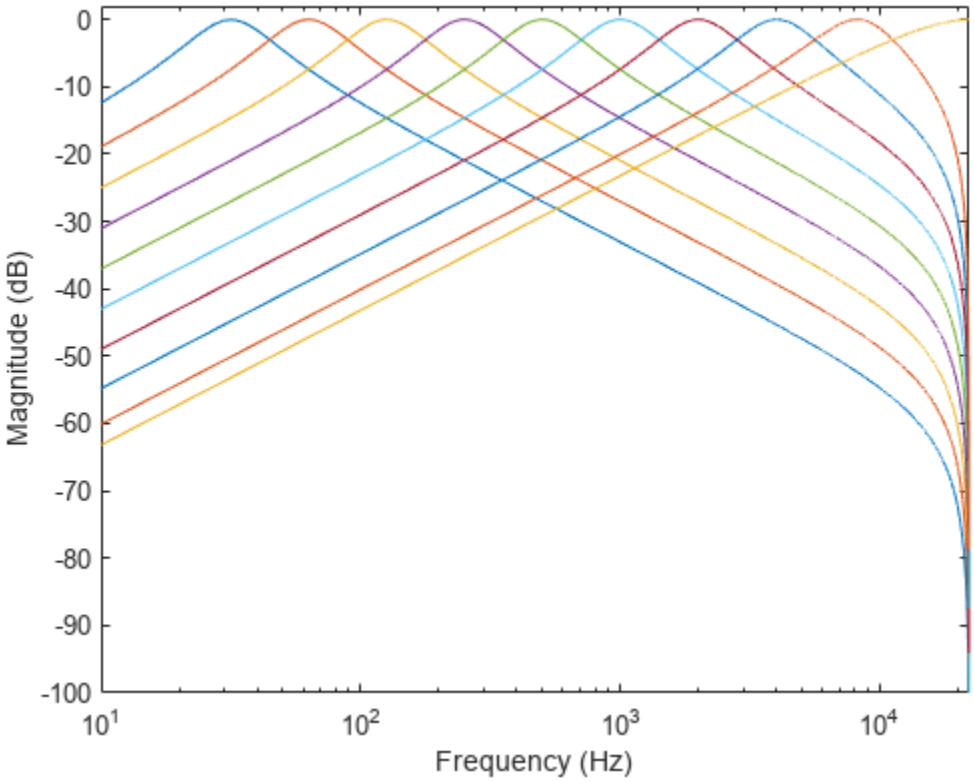
To get the frequency response of a subset of filters in the filter bank, specify the second argument as a row vector of indices between one and the number of filters in the filter bank. Get the frequency response of the 5th filter in the filter bank and plot the magnitude frequency response in dB.

```
[H,f] = freqz(octFiltBank,5);

plot(f,20*log10(abs(H)))
xlabel("Frequency (Hz)")
ylabel("Magnitude (dB)")
set(gca,XScale="log")
axis([10 octFiltBank.SampleRate/2 -100 2])
```



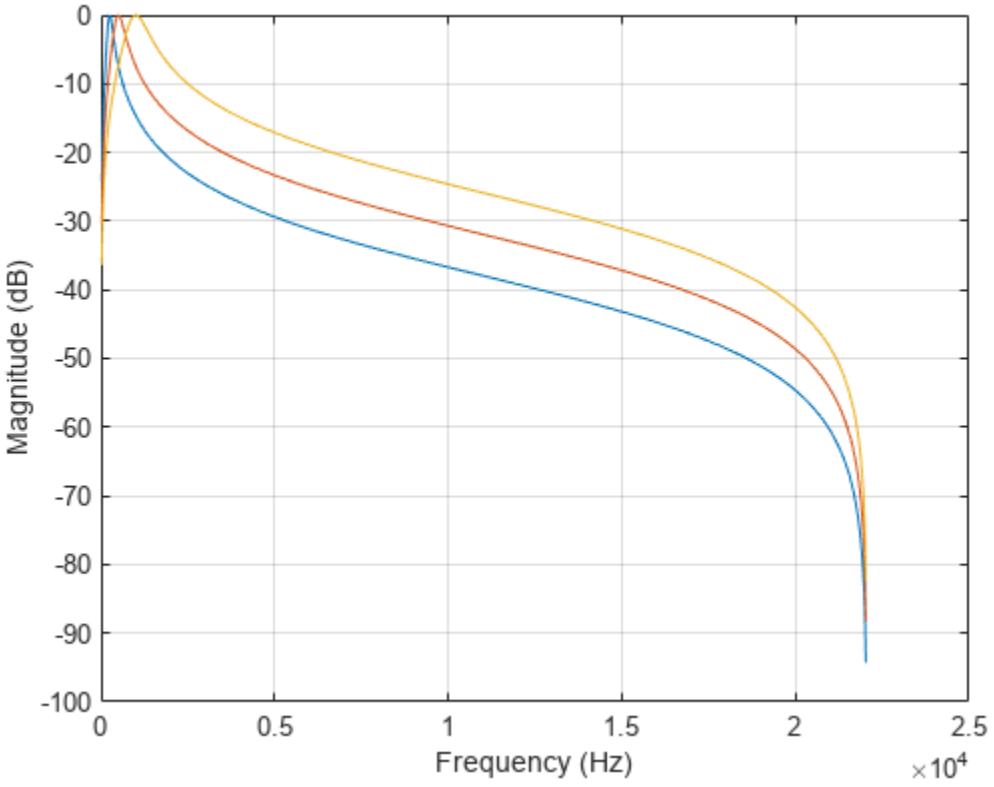
To specify the number of points in the frequency response, use the `N` name-value argument. Specify that the frequency response contains 8192 points. Plot the magnitude frequency response in dB.

```
[H,f] = freqz(octFiltBank,N=8192);

plot(f,20*log10(abs(H)))
xlabel("Frequency (Hz)")
ylabel("Magnitude (dB)")
set(gca,XScale="log")
axis([10 octFiltBank.SampleRate/2 -100 2])
```



To visualize the magnitude frequency response only, call `freqz` without any output arguments. Plot the magnitude frequency response, in dB, of filters 4, 5, and 6 with 1024 points.

```
freqz(octFiltBank,[4,5,6],N=1024)
```



Input Arguments

`obj` — Object to get filter frequency responses from

`gammatoneFilterBank` | `octaveFilterBank`

Object to get filter frequency responses from, specified as an object of `gammatoneFilterBank` or `octaveFilterBank`.

`ind` — Indices of filters to calculate frequency responses from

1:*N* (default) | row vector of integers with values in the range [1, *N*]

Indices of filters to calculate frequency responses from, specified as a row vector of integers with values in the range [1, *N*]. *N* is the total number of filters designed by `obj`.

`n` — Number of points in frequency response

8192 (default) | positive integer

Number of points in the frequency response, specified as a positive integer.

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

Output Arguments

`H` — Complex frequency response of each filter

matrix

Complex frequency response of each filter, returned as an *M*-by-*N* matrix. *M* is the number of points, specified by `n`. *N* is the number of filters, which is either `length(ind)` or, if `ind` is not specified, the total number of filters in the filter bank.

Data Types: `double`

`f` — Frequencies at which response is computed (Hz)
column vector

Frequencies at which the response is computed in Hz, returned as a column vector.

Data Types: `double`

Version History

Introduced in R2019a

See Also

`gammatoneFilterBank` | `octaveFilterBank` | `fvtool`

fvtool

Visualize filter bank

Syntax

```
fvtool(obj)
fvtool(obj,ind)
fvtool( ___ ,N=n)
```

Description

`fvtool(obj)` visualizes the filters in the filter bank using the Filter Visualization Tool (`FVTool`).

`fvtool(obj,ind)` visualizes the filters corresponding to the elements in the vector `ind`.

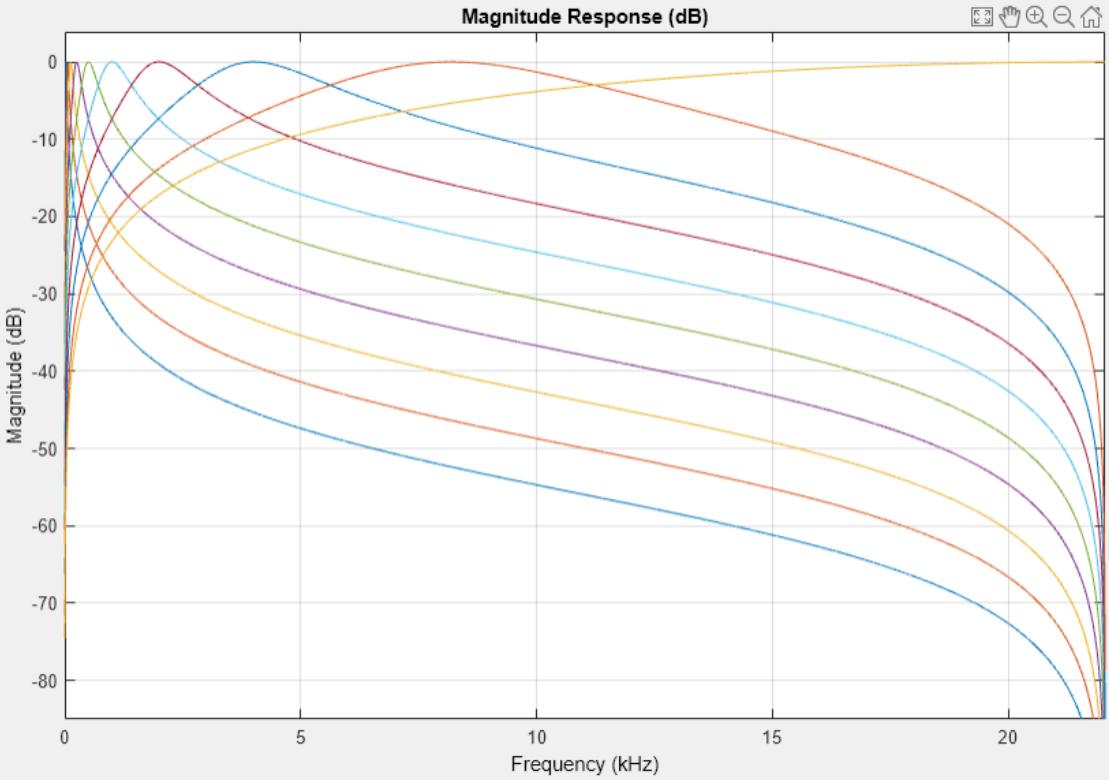
`fvtool( ___ ,N=n)` specifies the number of points used to visualize the filters.

Examples

View `octaveFilterBank` in `FVTool`

Create an `octaveFilterBank` object. Call `fvtool` to visualize the filter bank.

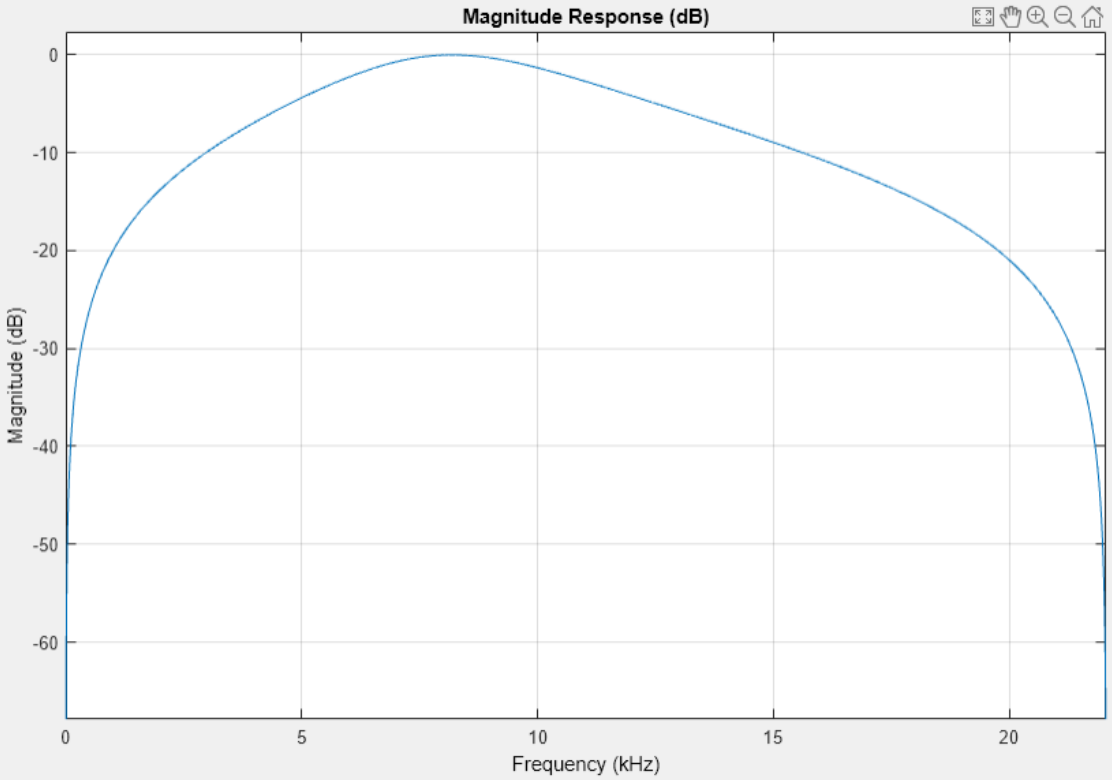
```
octFiltBank = octaveFilterBank;
fvtool(octFiltBank)
```




```
ans =
  Figure (filtervisualizationtool) with properties:

      Number: []
        Name: 'Figure 1: Magnitude Response (dB)'
       Color: [0.9400 0.9400 0.9400]
    Position: [360 402 560 420]
       Units: 'pixels'

  Use get to show all properties
```

To visualize a subset of filters in the filter bank, specify the second argument as a row vector of indices between one and the number of filters in the filter bank. If not specified, `fvtool` visualizes 1 to *N* filters of the filter bank, where *N* is the smallest of `octFiltBank.NumFilters` and `64`. Visualize the ninth filter.

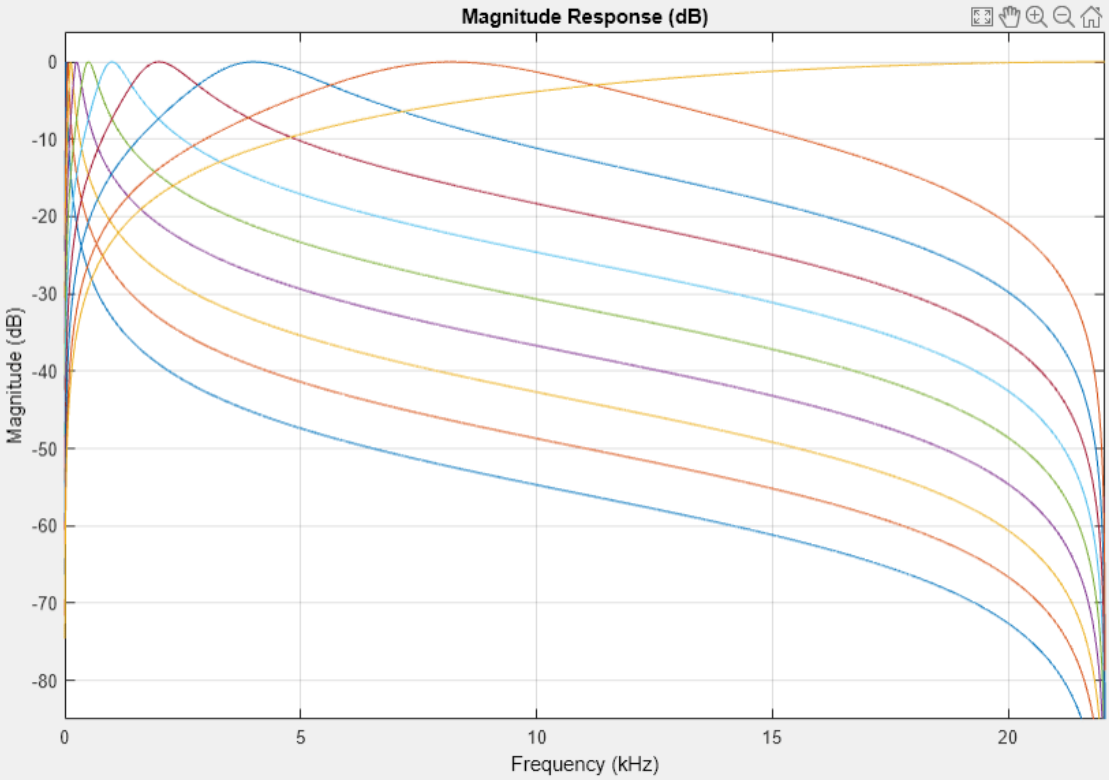
```
fvtool(octFiltBank,9)
```




```
ans =
  Figure (filtervisualizationtool) with properties:

      Number: []
        Name: 'Figure 2: Magnitude Response (dB)'
       Color: [0.9400 0.9400 0.9400]
    Position: [360 402 560 420]
       Units: 'pixels'

  Use get to show all properties
```

To specify the number of points in the frequency response, use the `N` name-value argument. Specify that the frequency response contains 8192 points.

```
fvtool(octFiltBank,N=8192)
```

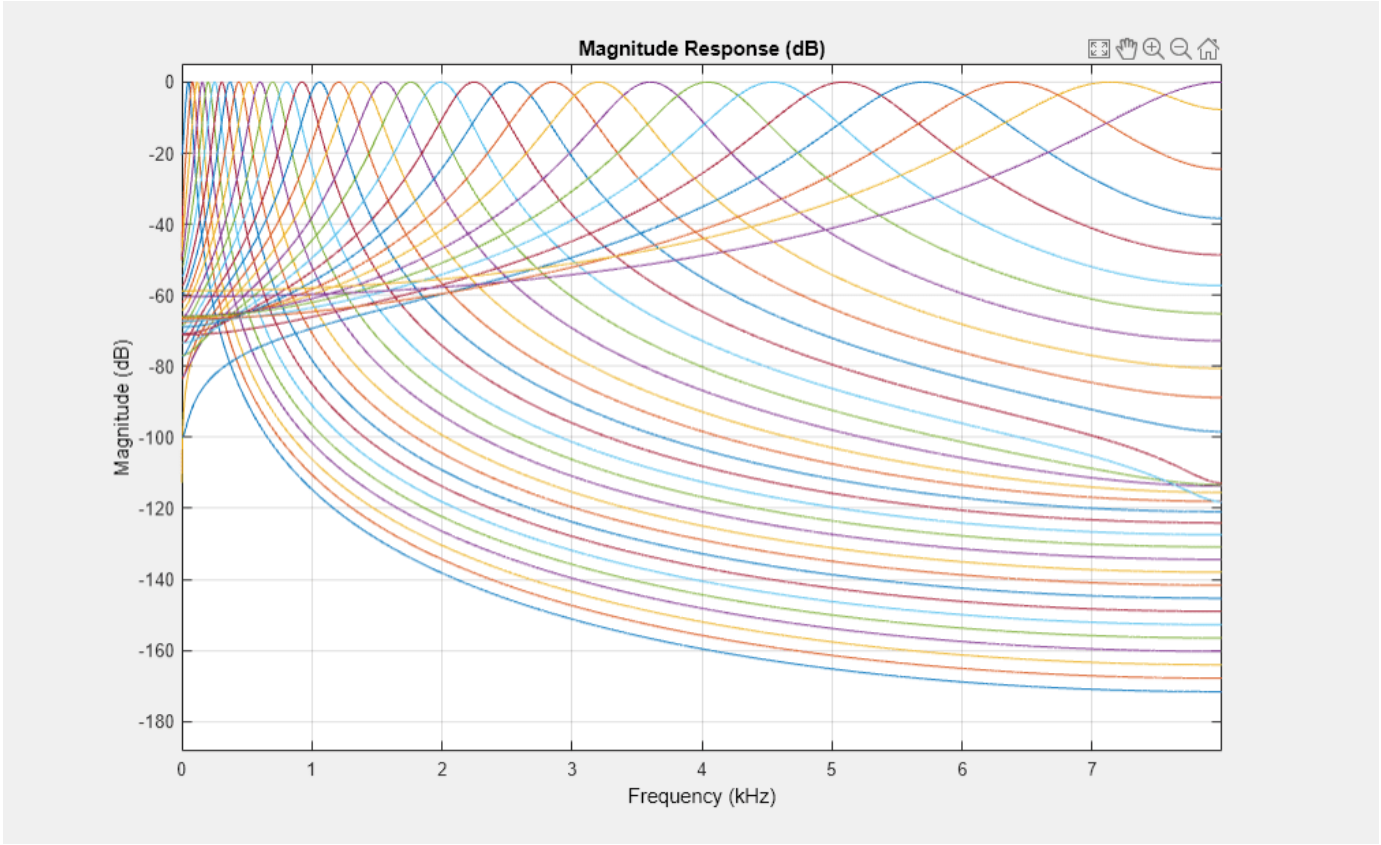



```
ans =
  Figure (filtervisualizationtool) with properties:

      Number: []
        Name: 'Figure 3: Magnitude Response (dB)'
       Color: [0.9400 0.9400 0.9400]
    Position: [360 402 560 420]
       Units: 'pixels'

  Use get to show all properties
```

View `gammatoneFilterBank` in FVTool

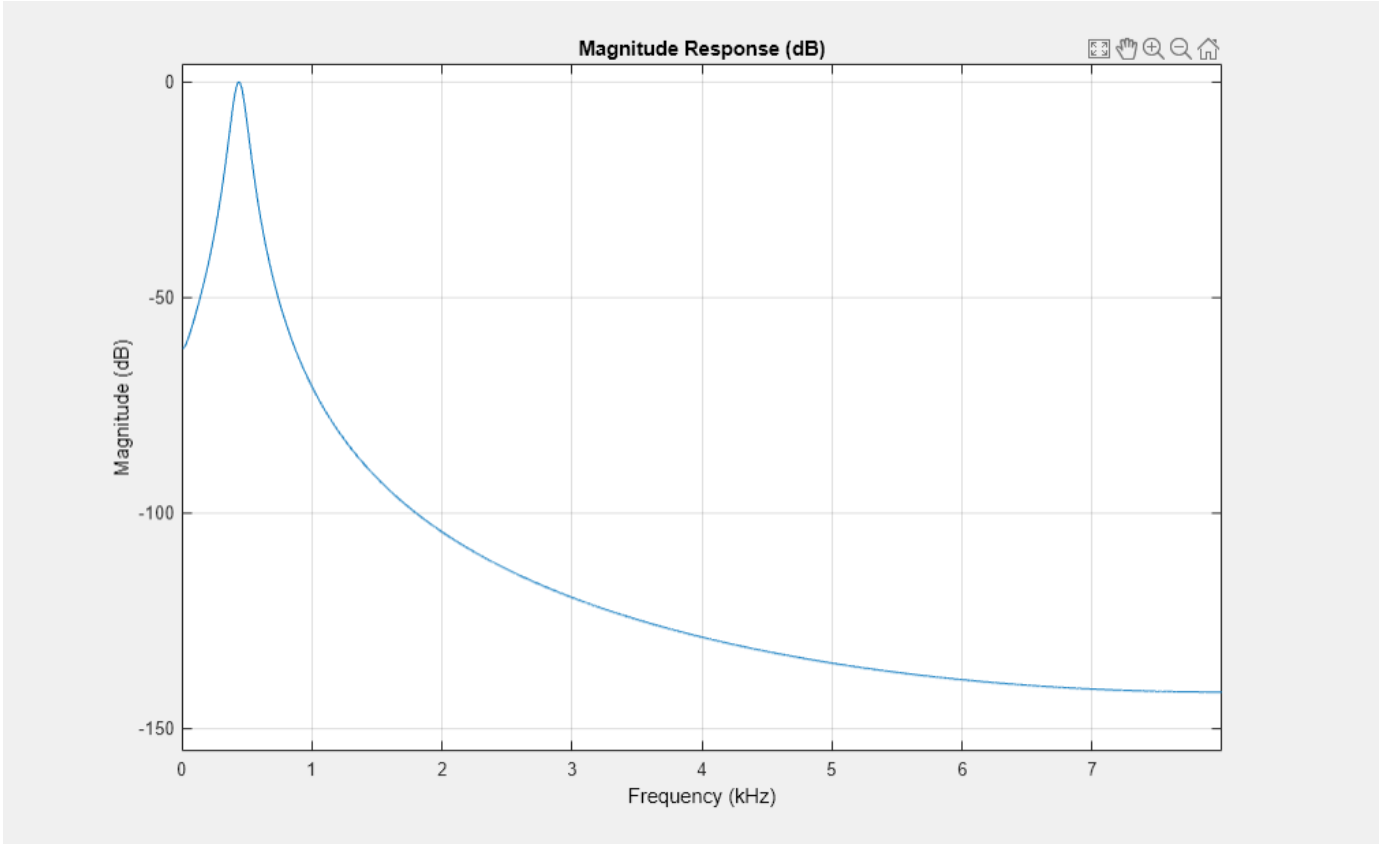
Create a `gammatoneFilterBank` object. Call `fvtool` to visualize the filter bank.

```
gammaFiltBank = gammatoneFilterBank;
fvtool(gammaFiltBank);
```



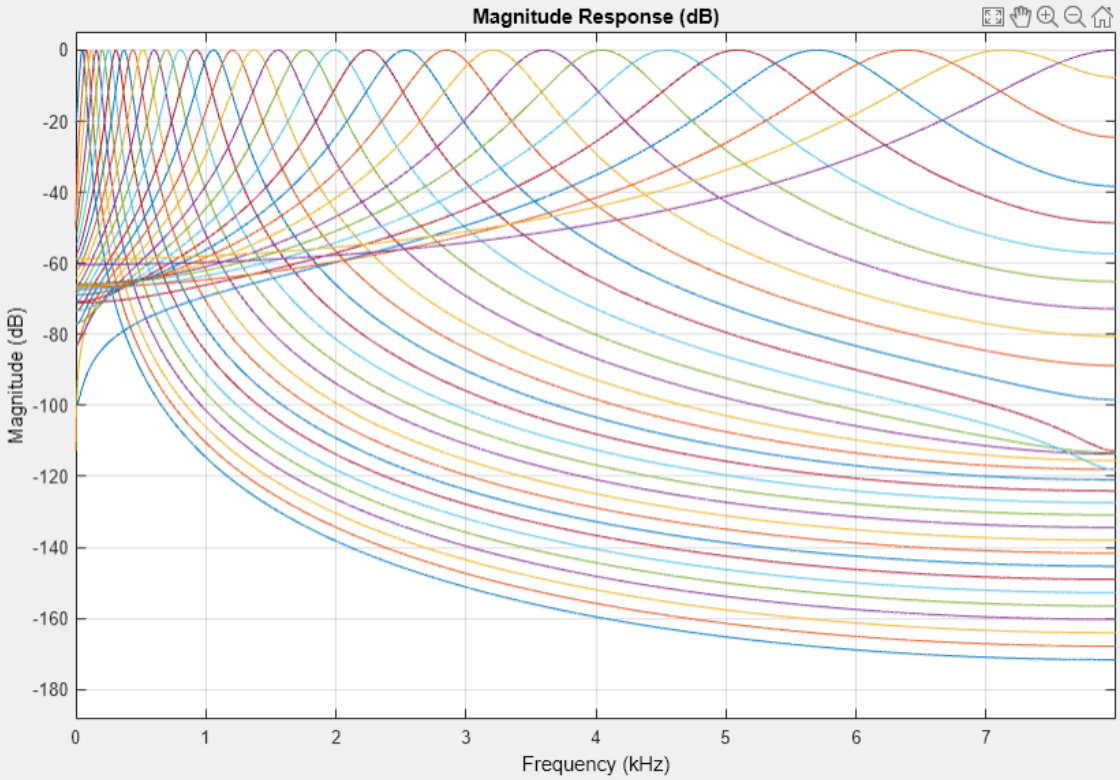
To visualize a subset of filters in the filter bank, specify the second argument as a row vector of indices between one and the number of filters in the filter bank. If not specified, `fvtool` visualizes 1 to *N* filters of the filter bank, where *N* is the smallest of `gammaFiltBank.NumFilters` and `64`. Visualize the ninth filter.

```
fvtool(gammaFiltBank,9);
```



To specify the number of points in the frequency response, use the `N` name-value argument. Specify that the frequency response contains 8192 points.

```
fvtool(gammaFiltBank,N=8192);
```



Input Arguments

`obj` — Object to get filter frequency responses from

`gammatoneFilterBank` | `octaveFilterBank`

Object to get filter frequency responses from, specified as an object of `gammatoneFilterBank` or `octaveFilterBank`.

`ind` — Indices of filters to calculate frequency responses from

`1:max(N,64)` (default) | row vector of integers with values in the range [1, *N*]

Indices of filters to calculate frequency responses from, specified as a row vector of integers with values in the range [1, *N*]. *N* is the total number of filters designed by `obj`.

`n` — Number of points

`8192` (default) | positive integer

Number of points used to visualize the filters, specified as a positive integer.

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

Version History

Introduced in R2019a

See Also

`gammatoneFilterBank` | `octaveFilterBank`

getBandedgeFrequencies

Get filter bandedges

Syntax

```
bandEdges = getBandedgeFrequencies(obj)
[bandEdges,centerFrequencies] = getBandedgeFrequencies(obj)
```

Description

`bandEdges = getBandedgeFrequencies(obj)` returns the bandedge frequencies of the filters designed by `obj`. If there are M filters, then there are M center frequencies and $M+1$ band edge frequencies.

`[bandEdges,centerFrequencies] = getBandedgeFrequencies(obj)` returns the center frequencies of the filters designed by `obj`.

Examples

Get Bandedge Frequencies

Create a default `octaveFilterBank` object.

```
octFiltBank = octaveFilterBank;
```

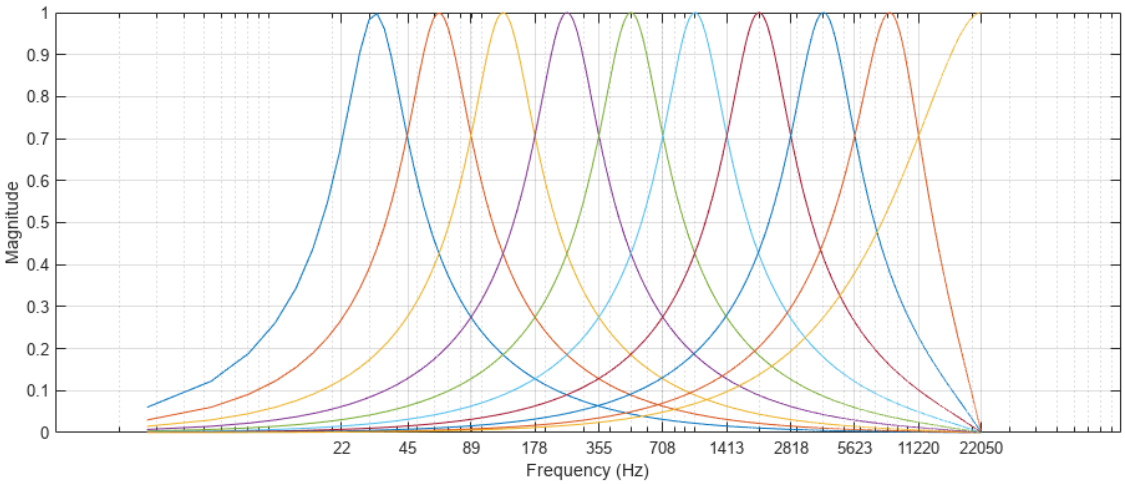
Call `getBandedgeFrequencies` to return a vector of bandedge frequencies.

```
bE = getBandedgeFrequencies(octFiltBank)
```

```
bE = 1×11
10^4 ×

    0.0022    0.0045    0.0089    0.0178    0.0355    0.0708    0.1413    0.2818    0.5623    1.1
```

Call `freqz` to get the frequency response of the filter bank. Plot the magnitude frequency response. Use the bandedge frequencies to label the frequency axis.

```
[H,f] = freqz(octFiltBank);
semilogx(f,abs(H))
xticks(round(bE))
xlabel('Frequency (Hz)')
ylabel('Magnitude')
grid on
h = gcf;
set(h,'Position',[h.Position(1) h.Position(2) h.Position(3)*2 h.Position(4)])
```



Input Arguments

`obj` — Object to get filter information from

`octaveFilterBank` object

Object to get filter information from, specified as an object of `octaveFilterBank`.

Output Arguments

`bandEdges` — Bandedges of filters (Hz)

row vector

Bandedges of filters designed by `obj` in Hz, returned as a row vector.

Data Types: `double` | `single`

`centerFrequencies` — Center frequencies of filters (Hz)

row vector

Center frequencies of filters designed by `obj` in Hz, returned as a row vector.

Data Types: `double` | `single`

Version History

Introduced in R2019a

See Also

`octaveFilterBank`

getCenterFrequencies

Center frequencies of filters

Syntax

```
cf = getCenterFrequencies(obj)
```

Description

`cf = getCenterFrequencies(obj)` returns the center frequencies of the filters created by `obj`, in Hz.

Examples

Center Frequencies of gammatoneFilterBank

Create a `gammatoneFilterBank` and get the center frequencies of the filters in the filter bank.

```
gammaFiltBank = gammatoneFilterBank;

cf = getCenterFrequencies(gammaFiltBank)
```

```
cf = 1×32
10^3 ×

    0.0500    0.0822    0.1181    0.1581    0.2027    0.2525    0.3081    0.3700    0.4391    0.5
```

Center frequencies of a gammatone filter bank are spaced evenly on the ERB scale. Convert the center frequencies vector to the ERB scale and calculate the differences between center frequencies.

```
diff(hz2erb(cf))
```

```
ans = 1×31

    1.0130    1.0130    1.0130    1.0130    1.0130    1.0130    1.0130    1.0130    1.0130    1.0
```

Center Frequencies of octaveFilterBank

Create an `octaveFilterBank` and get the center frequencies of the filters in the filter bank.

```
octFiltBank = octaveFilterBank;

cf = getCenterFrequencies(octFiltBank)
```

```
cf = 1×10
10^4 ×
```

```
    0.0032    0.0063    0.0126    0.0251    0.0501    0.1000    0.1995    0.3981    0.7943    1.5
```

Center frequencies of an octave filter bank are spaced evenly on a logarithmic scale. Convert the center frequencies vector to the log scale and calculate the differences between center frequencies.

```
diff(log10(cf))
```

```
ans = 1×9

    0.3000    0.3000    0.3000    0.3000    0.3000    0.3000    0.3000    0.3000    0.3000
```

Get Center Frequencies of Octave Filter Bank Used in `splMeter`

Create an octave bandwidth `splMeter` and get the center frequencies of the octave filter bank. Round the center frequencies to two significant digits for display purposes.

```
SPL = splMeter('SampleRate',44100,'Bandwidth','1 octave');
cf = getCenterFrequencies(SPL);
round(cf,2,'significant')
```

```
ans = 1×10

          32          63         130         250         500        1000        2000        4000
```

Input Arguments

`obj` — Object to get filter bank center frequencies from

`gammatoneFilterBank` | `octaveFilterBank` | `splMeter`

Object to get filter bank center frequencies from, specified as an object of `gammatoneFilterBank`, `octaveFilterBank`, or `splMeter`.

Output Arguments

`cf` — Filter bank center frequencies (Hz)

scalar | vector

Filter bank center frequencies in Hz, returned a scalar or vector.

Version History

Introduced in R2019a

See Also

`gammatoneFilterBank` | `octaveFilterBank` | `splMeter`

getBandwidths

Get filter bandwidths

Syntax

```
bw = getBandwidths(obj)
```

Description

`bw = getBandwidths(obj)` returns the bandwidths of the filters created by `obj`, in Hz.

Examples

Get Filter Bandwidths of gammatoneFilterBank

Create a default `gammatoneFilterBank`. Call `getBandwidths` to get the bandwidths of the filters, in Hz.

```
gammaFiltBank = gammatoneFilterBank;

bw = getBandwidths(gammaFiltBank)
```

```
bw = 1×32

   30.6688   34.2080   38.1555   42.5583   47.4691   52.9463   59.0554   65.8692   73.4690   81.9
```

Input Arguments

obj — Object to get filter bandwidth from

`gammatoneFilterBank`

Object to get filter bandwidth from, specified as an object of `gammatoneFilterBank`.

Output Arguments

bw — Filter bandwidths (Hz)

scalar | vector

Filter bandwidths in Hz, returned a scalar or vector.

Version History

Introduced in R2019a

See Also

`gammatoneFilterBank`

getGroupDelays

Get group delays

Syntax

```
groupDelays = getGroupDelays(obj)
[groupDelays,centerFrequencies] = getGroupDelays(obj)
```

Description

`groupDelays = getGroupDelays(obj)` returns the group delay of each filter at its center frequency.

`[groupDelays,centerFrequencies] = getGroupDelays(obj)` returns the center frequency of each filter.

Examples

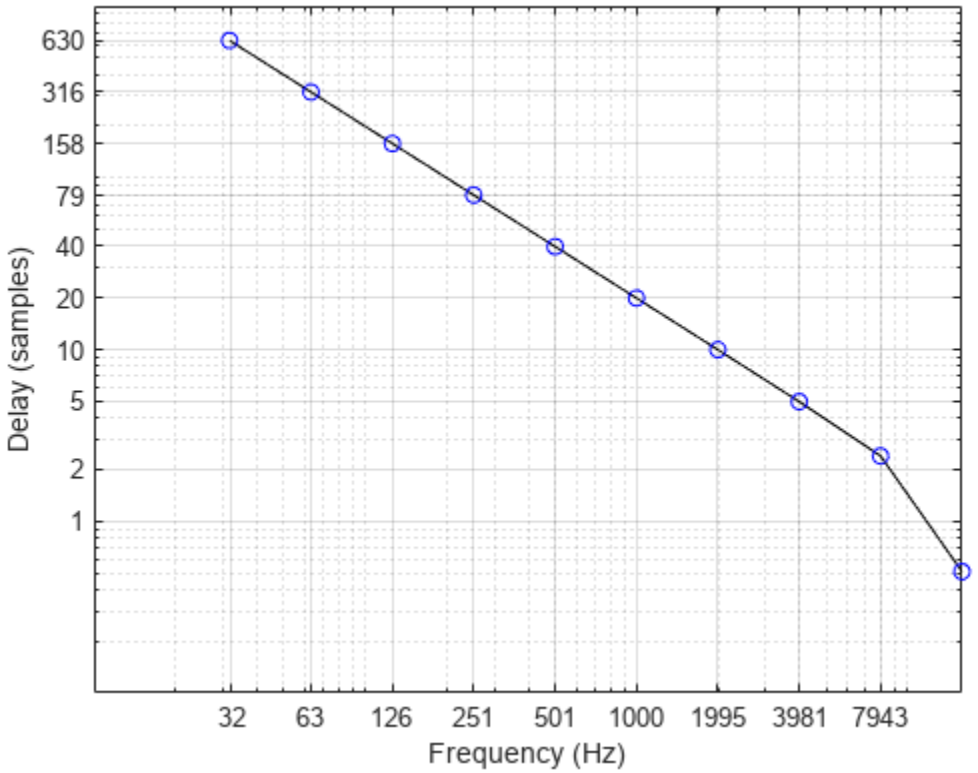
Get Group Delays

Create a default `octaveFilterBank` object. Call `getGroupDelays` to get the group delay of each octave filter at its center frequency.

```
octFiltBank = octaveFilterBank;
[gd,cf] = getGroupDelays(octFiltBank);
```

Plot the group delay as a function of filter center frequency.

```
loglog(cf,gd,'k-',cf,gd,'bo')
grid on
xlabel('Frequency (Hz)')
ylabel('Delay (samples)')
xticks(round(cf))
yticks(round(fliplr(gd)))
```



Input Arguments

`obj` — Object to get group delays from

`octaveFilterBank`

Object to get group delays from, specified as an object of `octaveFilterBank`.

Output Arguments

`groupDelays` — Group delays (samples)

row vector

Group delay of each filter at its center frequency in samples, returned as a row vector.

`centerFrequencies` — Center frequencies of filters (Hz)

row vector

Center frequencies of filters designed by `obj` in Hz, returned as a row vector.

Data Types: `double` | `single`

Version History

Introduced in R2019a

See Also

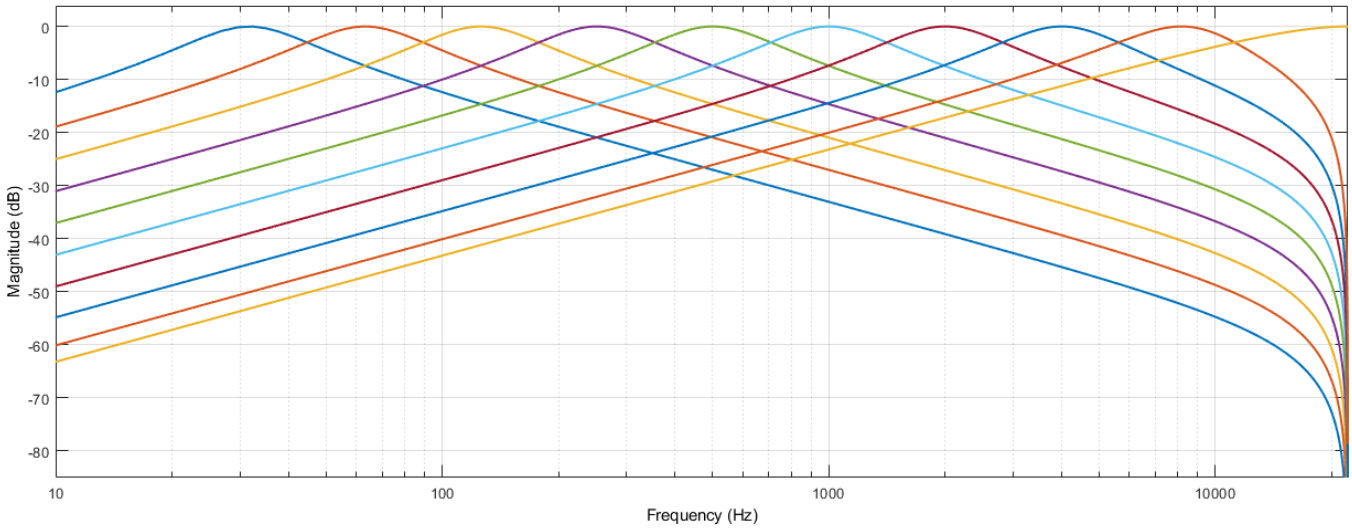
`octaveFilterBank`

octaveFilterBank

Octave and fractional-octave filter bank

Description

`octaveFilterBank` decomposes a signal into octave or fractional-octave subbands. 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.



To apply a bank of octave-band or fractional octave-band filters:

1. Create the `octaveFilterBank` object and set its properties.
2. Call the object with arguments, as if it were a function.

To learn more about how System objects work, see What Are System Objects?

Creation

Syntax

```
octFiltBank = octaveFilterBank
octFiltBank = octaveFilterBank(bandwidth)
octFiltBank = octaveFilterBank(bandwidth,fs)
octFiltBank = octaveFilterBank( ___ ,Name,Value)
```

Description

`octFiltBank = octaveFilterBank` returns an octave filter bank. The objects filters data independently across each input channel over time.

`octFiltBank = octaveFilterBank(bandwidth)` sets the `Bandwidth` property to `bandwidth`.

`octFiltBank = octaveFilterBank(bandwidth,fs)` sets the `SampleRate` property to `fs`.

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

Example: `octFiltBank = octaveFilterBank('1/2 octave','FrequencyRange', [62.5,12000])` creates a ½ octave-band filter bank, `octFiltBank`, with bandpass filters placed between 62.5 Hz and 12,000 Hz.

Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see System Design in MATLAB Using System Objects.

`Bandwidth` — Filter bandwidth (octave)

`'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: No

Data Types: `char` | `string`

`SampleRate` — Input sample rate (Hz)

`44100` (default) | positive scalar

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

Tunable: Yes

Data Types: `single` | `double`

`FrequencyRange` — Frequency range of filter bank (Hz)

`[22 22050]` (default) | two-element row vector of positive monotonically increasing values

Frequency range of the filter bank in Hz, specified as a two-element row vector of positive monotonically increasing values. The filter bank center frequencies are placed according to the `Bandwidth`, `RefererenceFrequency`, and `OctaveRatioBase` properties. Filters that have a center frequency outside `FrequencyRange` are ignored.

Tunable: No

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

`ReferenceFrequency` — Reference frequency (Hz)

`1000` (default) | positive integer scalar

Reference frequency of the filter bank in Hz, specified as a positive integer scalar. The reference frequency defines one of the center frequencies. All other center frequencies are set relative to the reference frequency.

Tunable: No

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

`FilterOrder` — Order of octave filters
2 (default) | even integer

Order of the octave filters, specified as an even integer. The filter order applies to each individual filter in the filter bank.

Tunable: No

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

`OctaveRatioBase` — Octave ratio base
`10` (default) | 2

Octave ratio base, specified as `10` or 2. The octave ratio base determines the distribution of the center frequencies of the octave filters. The ANSI S1.11 standard recommends base 10. Base 2 is popular for music applications. Base 2 defines an octave as a factor of 2, and base 10 defines an octave as a factor of $10^{0.3}$.

Tunable: No

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

Usage

Syntax

```
audioOut = octFiltBank(audioIn)
```

Description

`audioOut = octFiltBank(audioIn)` applies the octave filter bank on the input and returns the filtered output.

Input Arguments

`audioIn` — Audio input to octave filter bank
scalar | vector | matrix

Audio input to the octave filter bank, specified as a scalar, vector, or matrix. If specified as a matrix, the columns are treated as independent audio channels.

Data Types: `single` | `double`

Output Arguments

`audioOut` — Audio output from octave filter bank
matrix | 3-D array

Audio output from octave filter bank, returned as a scalar, vector, matrix, or 3-D array. The shape of `audioOut` depends on the shape of `audioIn` and the number of filters in the filter bank. If *M* is the number of filters, and `audioIn` is an *L*-by-*N* matrix, then `audioOut` is returned as an *L*-by-*M*-by-*N* array. If *N* is 1, then `audioOut` is a matrix.

Data Types: `single` | `double`

Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

Specific to `octaveFilterBank`

coeffs	Get filter coefficients
freqz	Compute frequency response
fvtool	Visualize filter bank
getBandedgeFrequencies	Get filter bandedges
getCenterFrequencies	Center frequencies of filters
getGroupDelays	Get group delays
info	Get filter information
isStandardCompliant	Verify octave filter bank is ANSI S1.11-2004 compliant

Common to All System Objects

step	Run System object algorithm
release	Release resources and allow changes to System object property values and input characteristics
reset	Reset internal states of System object

Examples

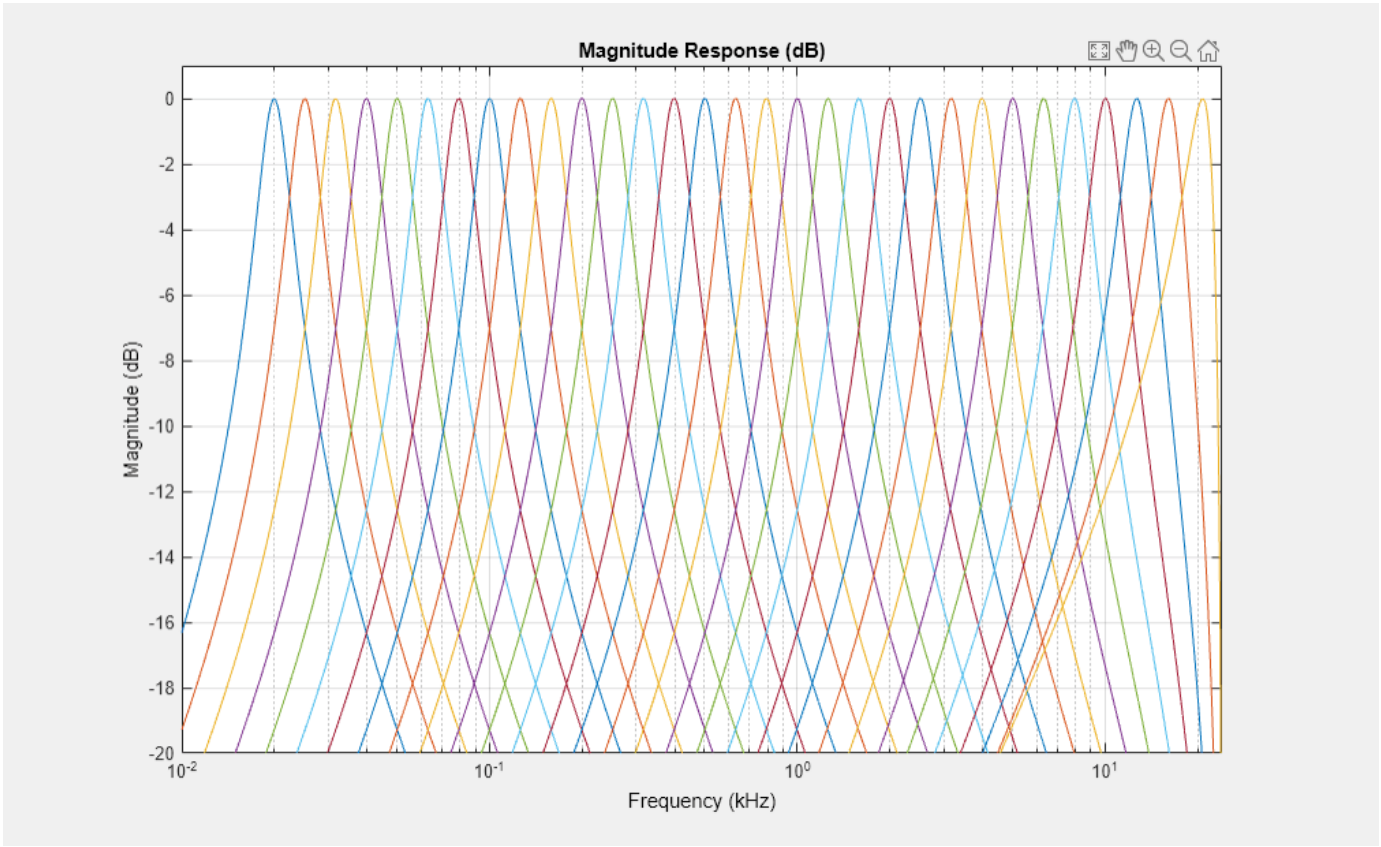
Apply Octave Filter Bank

Create a 1/3-octave filter bank for a signal sampled at 48 kHz. Set the frequency range to `[18 22000]` Hz.

```
octFilBank = octaveFilterBank("1/3 octave",48000, ...
                              FrequencyRange=[18 22000]);
```

Use `fvtool` to visualize the response of the filter bank. To get a high-resolution view on the lower frequencies, set the scale of the *x*-axis to `log` and `NFFT` to `2^16`.

```
fvt = fvtool(octFilBank,"NFFT",2^16);
set(fvt,FrequencyScale="log")
zoom(fvt,[.01 24 -20 1])
```



Display the filter bank center frequencies.

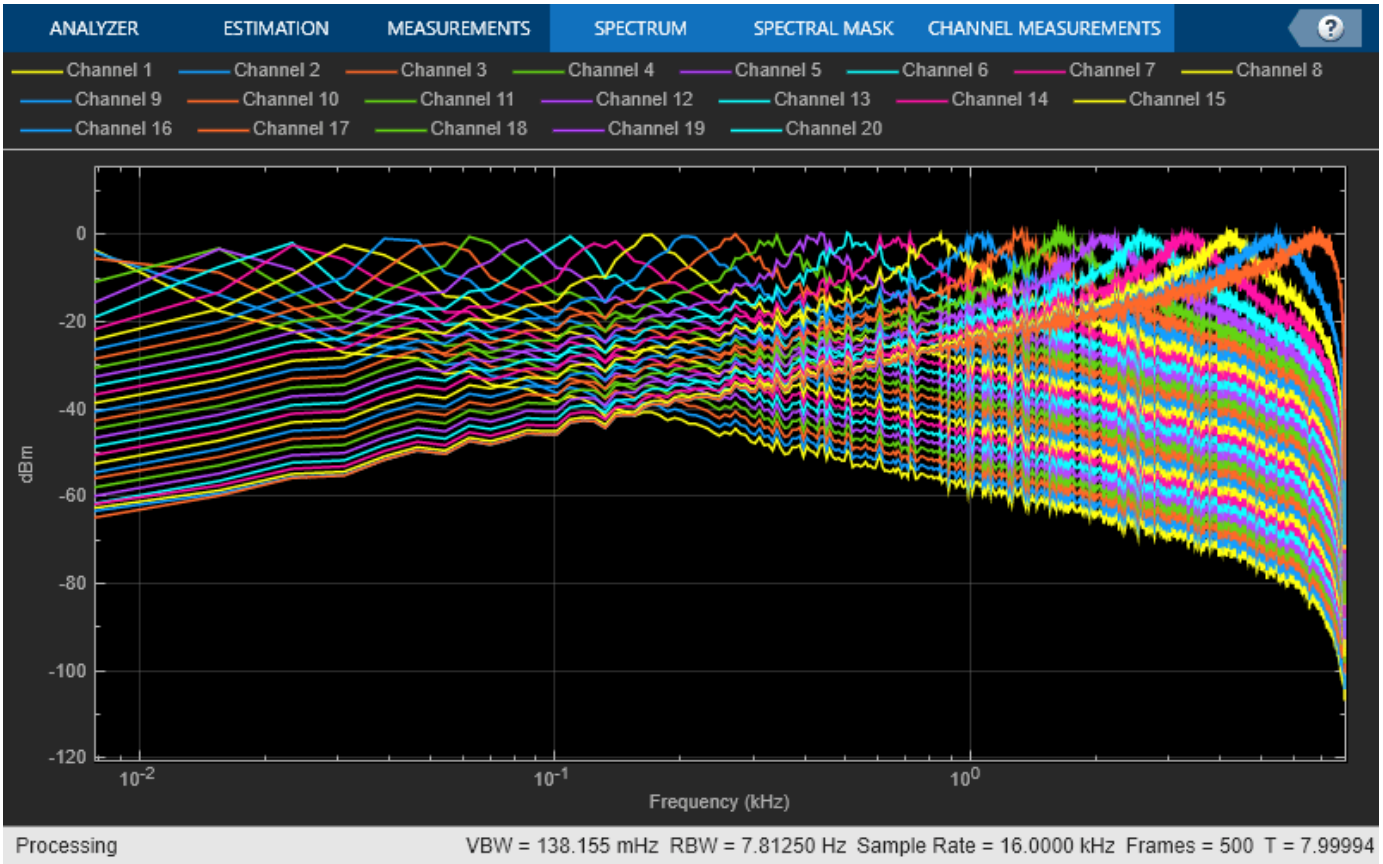
```
fc = getCenterFrequencies(octFilBank);
cf = string(size(fc));
for ii = find(fc<1000)
    cf(ii) = sprintf("%.0f Hz",round(fc(ii),2,"significant"));
end
for ii = find(fc>=1000)
    cf(ii) = sprintf("%.1f kHz",fc(ii)/1000);
end
disp(cf)
```

```
  Columns 1 through 7

    "20 Hz"    "25 Hz"    "32 Hz"    "40 Hz"    "50 Hz"    "63 Hz"    "79 Hz"

  Columns 8 through 13

    "100 Hz"    "130 Hz"    "160 Hz"    "200 Hz"    "250 Hz"    "320 Hz"

  Columns 14 through 19

    "400 Hz"    "500 Hz"    "630 Hz"    "790 Hz"    "1.0 kHz"    "1.3 kHz"

  Columns 20 through 25

    "1.6 kHz"    "2.0 kHz"    "2.5 kHz"    "3.2 kHz"    "4.0 kHz"    "5.0 kHz"
```

```
  Columns 26 through 30

    "6.3 kHz"    "7.9 kHz"    "10.0 kHz"    "12.6 kHz"    "15.8 kHz"

  Column 31

    "20.0 kHz"
```

Process white Gaussian noise through the filter bank. Use a spectrum analyzer to view the spectrum of the filter outputs.

```
sa = spectrumAnalyzer(SampleRate=16e3,...
    PlotAsTwoSidedSpectrum=false,...
    FrequencyScale="log");

for index = 1:500
    x = randn(256,1);
    y = octFilBank(x);
    sa(y);
end
```



Analysis and Synthesis

The `octaveFilterBank` enables good reconstruction of a signal after analyzing or modifying its subbands.

Read in an audio file and listen to its contents.

```
[audioIn,fs] = audioread('RandomOscThree-24-96-stereo-13secs.aif');
sound(audioIn,fs)
```

Create a default `octaveFilterBank`. The default frequency range of the filter bank is 22 to 22,050 Hz. Frequencies outside of this range are attenuated in the reconstructed signal.

```
octFiltBank = octaveFilterBank('SampleRate',fs);
```

Pass the audio signal through the octave filter bank. The number of outputs depends on the `FrequencyRange`, `ReferenceFrequency`, `OctaveRatioBase`, and `Bandwidth` properties of the octave filter bank. Each channel of the input is passed through a filter bank independently and is returned as a separate page in the output.

```
audioOut = octFiltBank(audioIn);

[N,numFilters,numChannels] = size(audioOut)
```

```
N = 1265935

numFilters = 10

numChannels = 2
```

The octave filter bank introduces various group delays. To compensate for the group delay, remove the beginning delay from the individual filter outputs and zero-pad the ends of the signals so that they are all the same size. Use `getGroupDelays` to get the group delays. Listen to the group delay-compensated reconstruction.

```
groupDelay = round(getGroupDelays(octFiltBank)); % round for simplicity

audioPadded = [audioOut;zeros(max(groupDelay),numFilters,numChannels)];

for i = 1:numFilters
    audioOut(:,i,:) = audioPadded(groupDelay(i)+1:N+groupDelay(i),i,:);
end
```

To reconstruct the original signal, sum the outputs of the filter banks for each channel. Use `squeeze` to move the second channel from the third dimension to the second in the reconstructed signal.

```
reconstructedSignal = squeeze(sum(audioOut,2));
sound(reconstructedSignal,fs)
```

Algorithms

The `octaveFilterBank` is implemented as a parallel structure of octave filters. Individual octave filters are designed as described by `octaveFilter`. By default, the octave filter bank center frequencies are placed as specified by the ANSI S1.11-2004 standard. You can modify the filter placements using the `Bandwidth`, `FrequencyRange`, `ReferenceFrequency`, and `OctaveRatioBase` properties.

Version History

Introduced in R2019a

References

[1] Orfanidis, Sophocles J. *Introduction to Signal Processing*. Englewood Cliffs, NJ: Prentice Hall, 2010.

[2] Acoustical Society of America. *American National Standard Specification for Octave-Band and Fractional-Octave-Band Analog and Digital Filters*. ANSI S1.11-2004. Melville, NY: Acoustical Society of America, 2009.

Extended Capabilities

C/C++ Code Generation

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

Usage notes and limitations:

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

See Also

Octave Filter Bank | `gammatoneFilterBank` | `octaveFilter` | `graphicEQ` | `splMeter`

Topics

"Octave-Band and Fractional Octave-Band Filters"

isStandardCompliant

Verify octave filter bank is ANSI S1.11-2004 compliant

Syntax

```
[status,cf] = isStandardCompliant(ofb,ctype)
[statusall,cfref] = isStandardCompliant(ofb,ctype,'all')
```

Description

`[status,cf] = isStandardCompliant(ofb,ctype)` indicates whether each filter in `ofb` is compliant with the minimum and maximum attenuation specifications of the `ctype` design specified in the ANSI S1.11-2004 standard. The function also returns a vector of center frequencies.

`[statusall,cfref] = isStandardCompliant(ofb,ctype,'all')` returns a scalar that is true only if all the filters in the filter bank are compliant.

Examples

Verify Standard Compliance

Use `octaveFilterBank` to design an octave filter bank. Use `isStandardCompliant` to verify if the designed octave filter bank is compliant with the ANSI S1.11-2004 standard.

Create an `octaveFilterBank` object composed of 12th-order 1-octave filters.

Call `isStandardCompliant`, specifying the compliance class type as `'class 0'`. Display the compliance status and reference frequency for each filter.

```
ofb = octaveFilterBank('FilterOrder',12,'Bandwidth','1 octave');
[status,reffreq] = isStandardCompliant(ofb,'class 0')
```

```
status = 1x10 logical array

   1   1   1   1   1   1   1   1   1   1


reffreq = 1×10
10^4 ×

    0.0032    0.0063    0.0126    0.0251    0.0501    0.1000    0.1995    0.3981    0.7943    1.5
```

Verify Octave Filter Bank Compliance

Use the '`all`' option to return compliance verification of the complete filter bank. The '`all`' option also returns the ANSI reference frequency against which the function checks the mask.

```
ofb = octaveFilterBank('FilterOrder',12,'Bandwidth','1/3 octave');
[status,reffreq] = isStandardCompliant(ofb,'class 0','all')
```

```
status = logical
   1


reffreq = 1000
```

Decrease the filter order to produce a noncompliant filter in the bank. Verify the last filter in the bank is noncompliant.

```
ofb = octaveFilterBank('FilterOrder',8);
status = isStandardCompliant(ofb,'class 1')
```

```
status = 1x10 logical array

   1   1   1   1   1   1   1   1   1   0
```

Use the 'all' option to verify noncompliance of the bank.

```
isStandardCompliant(ofb,'class 1','all')
```

```
ans = logical
   0
```

Input Arguments

ofb — Input octave filter bank

octaveFilterBank object

Input filter bank, specified as an octaveFilterBank object.

ctype — Compliance class type

'class 0' | 'class 1' | 'class 2'

Compliance class type to verify, specified as 'class 0', 'class 1', or 'class 2'. For more information on ANSI S1.11-2004 compliant filter classes, see “Octave-Band and Fractional Octave-Band Filters”.

Data Types: char | string

Output Arguments

status — Compliance status

vector

Compliance status, returned as a logical vector. The compliance status indicates whether each filter in the object ofb is compliant with the ANSI S1.11-2004 standard for ctype.

Data Types: logical

cf — Standard-compliant center frequencies

vector

Standard-compliant center frequencies, returned as a vector. The center frequencies are used to set the class attenuation limits.

`statusall` — Aggregate compliance status
scalar

Aggregate compliance status, returned as a logical scalar. `statusall` is true only if all the filters in the filter bank are compliant.

Data Types: `logical`

`cfref` — Standard-compliant reference frequency
scalar

Standard-compliant reference frequency, returned as a scalar. The reference frequency is used to set the class attenuation limits.

Tips

- To meet compliance, set the reference frequency of the octave filter bank to one of the values returned by the `getANSICenterFrequencies(octaveFilter)` method, increase the filter order, limit the frequency range, or increase the sample rate.

Version History

Introduced in R2021a

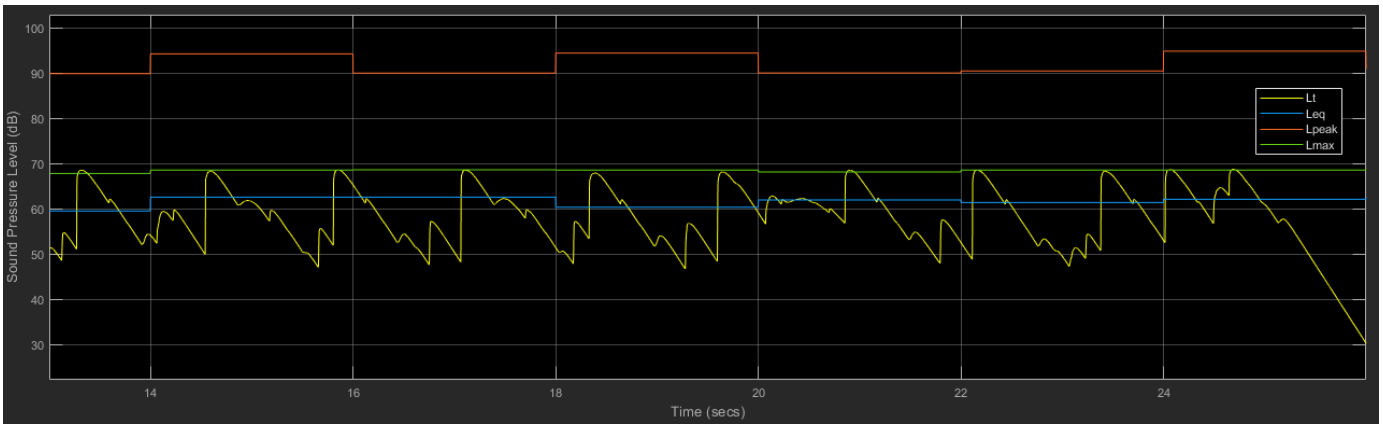
splMeter

Measure sound pressure level of audio signal

Description

The `splMeter` System object computes sound pressure level measurements. The object returns measurements for:

- frequency-weighted sound levels
- fast or slow time-weighted sound levels
- equivalent-continuous sound levels
- peak sound levels
- maximum sound levels



To implement SPL metering:

1. Create the `splMeter` object and set its properties.
2. Call the object with arguments, as if it were a function.

To learn more about how System objects work, see What Are System Objects?

Creation

Syntax

```
SPL = splMeter
SPL = splMeter(Name,Value)
```

Description

`SPL = splMeter` creates a System object, `SPL`, that performs SPL metering.

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

Example: `SPL = splMeter('FrequencyWeighting','C-weighting','SampleRate',12000)` creates a System object, `SPL`, that performs C-weighting and operates at 12 kHz.

Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see System Design in MATLAB Using System Objects.

`Bandwidth` — Width of analysis bands

`'Full band'` (default) | `'1 octave'` | `'2/3 octave'` | `'1/3 octave'`

Width of analysis bands, specified as `'Full band'`, `'1 octave'`, `'2/3 octave'`, or `'1/3 octave'`. If `Bandwidth` is specified as `'Full band'`, the SPL meter returns one set of measurements for the whole frequency band. If `Bandwidth` is specified as `'1 octave'`, `'2/3 octave'`, or `'1/3 octave'`, the SPL meter returns one set of measurements per octave or fractional-octave band.

Tunable: No

Data Types: `char` | `string`

`FrequencyRange` — Frequency range of filter bank (Hz)

`[22 22050]` (default) | two-element row vector of positive monotonically increasing values

Frequency range of the filter bank in Hz, specified as a two-element row vector of positive monotonically increasing values. Frequency bands centered above `SampleRate`/2 are excluded.

Tunable: No

Dependencies

To enable this property, set `Bandwidth` to `'1 octave'`, `'2/3 octave'`, or `'1/3 octave'`.

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

`OctaveFilterOrder` — Order of octave filter

2 (default) | even integer

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

Tunable: No

Dependencies

To enable this property, set `Bandwidth` to `'1 octave'`, `'2/3 octave'`, or `'1/3 octave'`.

Data Types: `single` | `double`

`FrequencyWeighting` — Frequency weighting applied to input

`'A-weighting'` (default) | `'C-weighting'` | `'Z-weighting'` (no weighting)

Frequency weighting applied to input, specified as `'A-weighting'`, `'C-weighting'`, or `'Z-weighting'`, where Z-weighting corresponds to no weighting. The frequency weighting is designed and implemented using the `weightingFilter` System object.

Tunable: No

Data Types: `char` | `string`

`TimeWeighting` — Time weighting (s)

`'Fast'` (default) | `'Slow'`

Time weighting, in seconds, for calculation of time-weighted sound level and maximum time-weighted sound level, specified as `'Fast'` or `'Slow'`. The `TimeWeighting` property is used to specify the coefficient of a lowpass filter.

- `'Fast'` – 1/8
- `'Slow'` – 1

Tunable: Yes

Data Types: `char` | `string`

`PressureReference` — Reference pressure for dB calculations (Pa)

`2e-5` (default) | positive scalar

Reference pressure for dB calculations in Pa, specified as a positive scalar.

Tunable: Yes

Data Types: `single` | `double`

`TimeInterval` — Time interval for reporting level measurements (s)

`1` (default) | positive scalar

Time interval, in seconds, to report equivalent-continuous, peak, and maximum time-weighted sound levels, specified as a positive scalar integer.

Tunable: No

Data Types: `single` | `double`

`CalibrationFactor` — Calibration factor multiplied by input

`1` (default) | positive finite scalar or vector

Scalar (mono input) or vector (multichannel input) calibration factor multiplied by input.

To set the calibration factor using a reference tone, use `calibrate`.

Tunable: No

Data Types: `single` | `double`

`SampleRate` — Input sample rate (Hz)

`44100` (default) | positive scalar

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

Tunable: No

Data Types: `single` | `double`

Usage

Syntax

```
[Lt,Leq,Lpeak,Lmax] = SPL(audioIn)
```

Description

`[Lt,Leq,Lpeak,Lmax] = SPL(audioIn)` returns measurement values for the time-weighted (`Lt`) sound level of the current input frame, `audioIn`. The object also returns the equivalent-continuous (`Leq`), peak (`Lpeak`), and maximum time-weighted (`Lmax`) sound levels of the input to your SPL meter.

Input Arguments

`audioIn` — Audio input to SPL meter

column vector | matrix

Audio input to the SPL meter, specified as a column vector or matrix. The columns of the matrix are treated as independent audio channels.

Data Types: `single` | `double`

Output Arguments

`Lt` — Time-weighted sound level (dB)

column vector | matrix | 3-D array

Time-weighted sound level in dB, returned as a column vector, matrix, or 3-D array the same type as `audioIn`.

Size and interpretation of the outputs depend on what the Bandwidth property is set to:

- `'Full band'` (default) –– `Lt`, `Leq`, `Lpeak`, and `Lmax` are returned as column vectors or matrices the same size as `audioIn`.
- `'1 octave'`, `'2/3 octave'`, or `'1/3 octave'` –– `Lt`, `Leq`, `Lpeak`, and `Lmax` are returned as *L*-by-*B*-by-*C* arrays.
  - *L* –– Number of rows in `audioIn`
  - *B* –– Number of octave bands
  - *C* –– Number of columns in `audioIn`

Data Types: `single` | `double`

`Leq` — Equivalent-continuous sound level (dB)

column vector | matrix | 3-D array

Equivalent-continuous sound level in dB, returned as a column vector, matrix, or 3-D array the same type as `audioIn`.

Size and interpretation of the outputs depend on what the Bandwidth property is set to:

- `'Full band'` (default) –– `Lt`, `Leq`, `Lpeak`, and `Lmax` are returned as column vectors or matrices the same size as `audioIn`.
- `'1 octave'`, `'2/3 octave'`, or `'1/3 octave'` –– `Lt`, `Leq`, `Lpeak`, and `Lmax` are returned as *L*-by-*B*-by-*C* arrays.
  - *L* –– Number of rows in `audioIn`
  - *B* –– Number of octave bands
  - *C* –– Number of columns in `audioIn`

Data Types: `single` | `double`

Lpeak — Peak sound level (dB)
column vector | matrix | 3-D array

Peak sound level in dB, returned as a column vector, matrix, or 3-D array the same type as `audioIn`.

Size and interpretation of the outputs depend on what the Bandwidth property is set to:

- `'Full band'` (default) –– `Lt`, `Leq`, `Lpeak`, and `Lmax` are returned as column vectors or matrices the same size as `audioIn`.
- `'1 octave'`, `'2/3 octave'`, or `'1/3 octave'` –– `Lt`, `Leq`, `Lpeak`, and `Lmax` are returned as *L*-by-*B*-by-*C* arrays.
  - *L* –– Number of rows in `audioIn`
  - *B* –– Number of octave bands
  - *C* –– Number of columns in `audioIn`

Data Types: `single` | `double`

Lmax — Maximum time-weighted sound level (dB)
column vector | matrix | 3-D array

Maximum time-weighted sound level in dB, returned as a column vector, matrix, or 3-D array the same type as `audioIn`.

Size and interpretation of the outputs depend on what the Bandwidth property is set to:

- `'Full band'` (default) –– `Lt`, `Leq`, `Lpeak`, and `Lmax` are returned as column vectors or matrices the same size as `audioIn`.
- `'1 octave'`, `'2/3 octave'`, or `'1/3 octave'` –– `Lt`, `Leq`, `Lpeak`, and `Lmax` are returned as *L*-by-*B*-by-*C* arrays.
  - *L* –– Number of rows in `audioIn`
  - *B* –– Number of octave bands
  - *C* –– Number of columns in `audioIn`

Data Types: `single` | `double`

Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

Specific to splMeter

calibrate	Calibrate meter using calibration tone with known level
getCenterFrequencies	Center frequencies of filters

Common to All System Objects

step	Run System object algorithm
release	Release resources and allow changes to System object property values and input characteristics
reset	Reset internal states of System object

Examples

Measure SPL of Audio Signal

Use the `splMeter` System object™ to measure the A-weighted sound pressure level of a streaming audio signal. Specify a two second time-interval for reporting and a fast time-weighting. Visualize the SPL measurements using the `timescope` object.

Create a `dsp.AudioFileReader` object to read in an audio file frame by frame. Create an `audioDeviceWriter` object to listen to the audio signal. Create a `timescope` object to visualize SPL measurements. Create an `splMeter` to measure the sound pressure level of the audio file. Use the default calibration factor of 1.

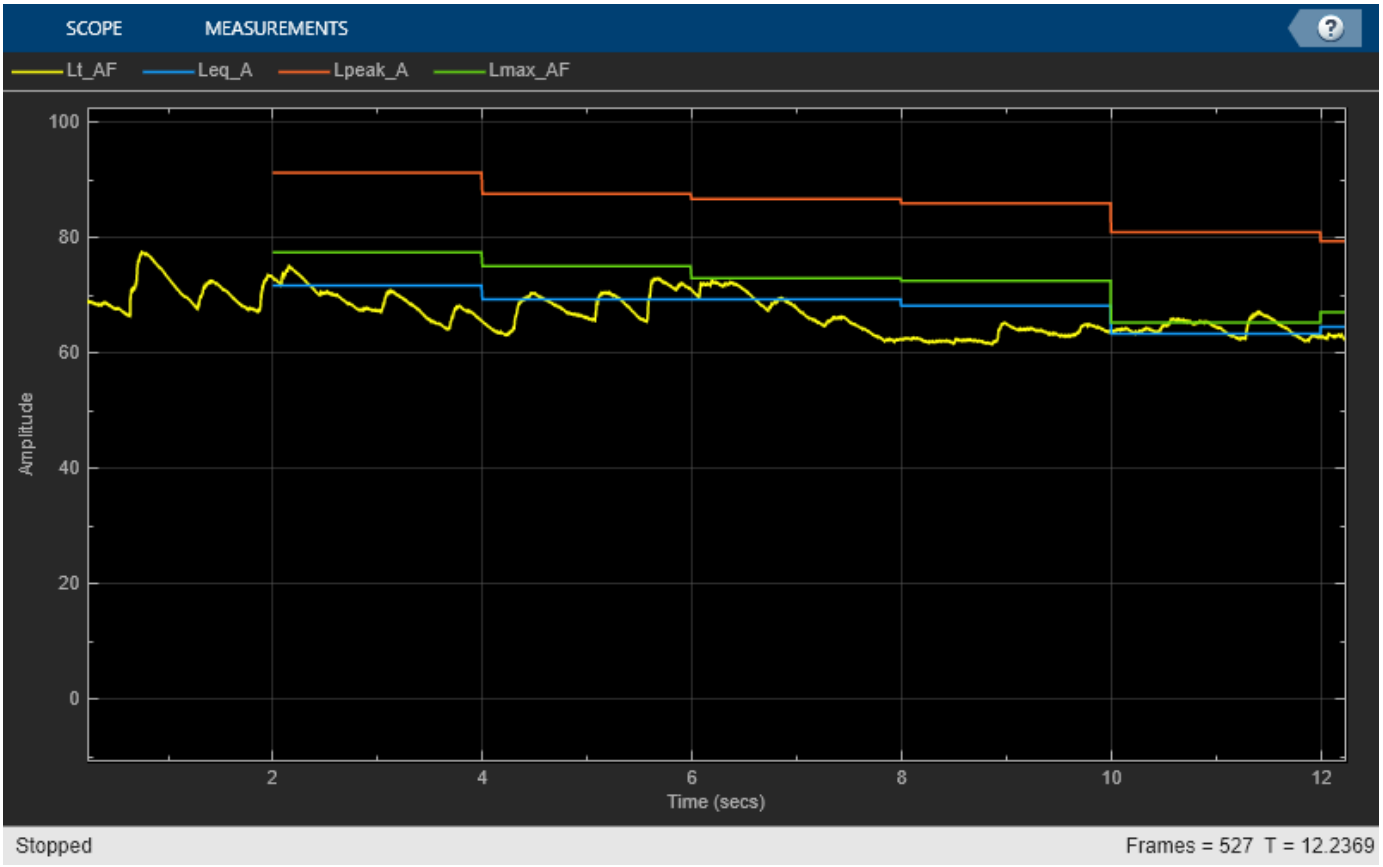
```
source = dsp.AudioFileReader('Ambiance-16-44p1-mono-12secs.wav');
fs = source.SampleRate;

player = audioDeviceWriter('SampleRate',fs);

scope  = timescope('SampleRate',fs, ...
    'TimeSpanOverrunAction','Scroll', ...
    'TimeSpanSource','Property','TimeSpan',3,'ShowGrid',true, ...
    'YLimits',[20 110],'AxesScaling','Auto', ...
    'ShowLegend',true,'BufferLength',4*3*fs, ...
    'ChannelNames', ...
    {'Lt_AF','Leq_A','Lpeak_A','Lmax_AF'}, ...
    'Name','Sound Pressure Level Meter');

SPL = splMeter('TimeWeighting','Fast', ...
    'FrequencyWeighting','A-weighting', ...
    'SampleRate',fs, ...
    'TimeInterval',2);
```

In an audio stream loop:

1. Read in the audio signal frame.
2. Play the audio signal to your output device.
3. Call the SPL meter to return the time-weighted, equivalent-continuous, peak, and maximum time-weighted sound levels in dB.
4. Display the sound levels using the scope.

As a best practice, release your objects once complete.

```
while ~isDone(source)
    x = source();
    player(x);
    [Lt,Leq,Lpeak,Lmax] = SPL(x);
    scope([Lt,Leq,Lpeak,Lmax])
end

release(source)
release(player)
release(SPL)
release(scope)
```



Octave SPL Metering

The `splMeter` enables you to monitor sound pressure level for octave and fractional-octave bands. In this example, you monitor the equivalent-continuous sound pressure level of 1/3-octave bands.

Create a `dsp.AudioFileReader` object to read in an audio file frame by frame. Create an `audioDeviceWriter` object so you can listen to the audio signal. Create an `splMeter` to measure the octave sound pressure level of the audio file. Use the default calibration factor of 1. Create a `dsp.ArrayPlot` object to visualize the equivalent-continuous SPL for each octave band.

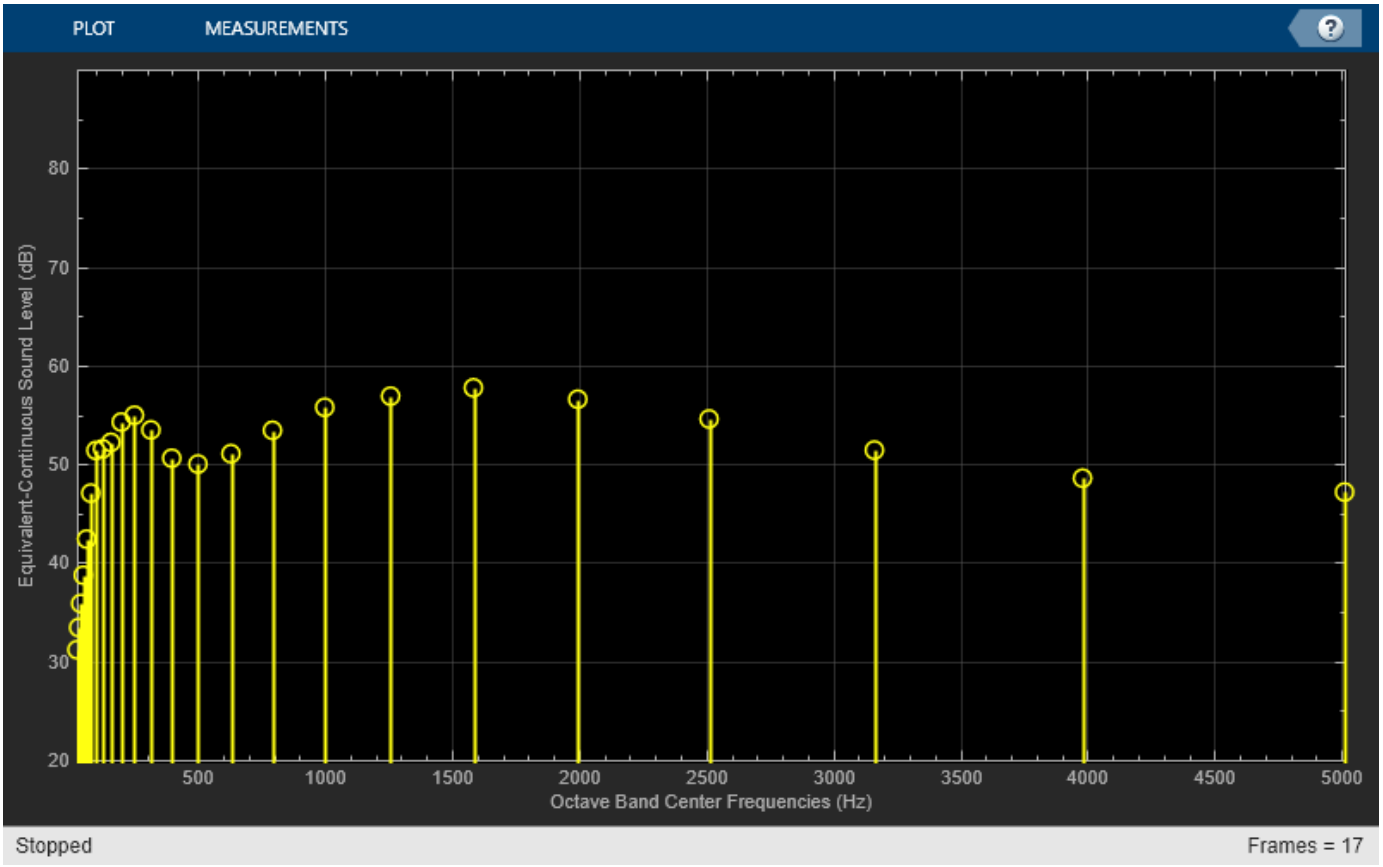
```
source = dsp.AudioFileReader('JetAirplane-16-11p025-mono-16secs.wav');
fs = source.SampleRate;

player = audioDeviceWriter('SampleRate',fs);

SPL = splMeter( ...
    'Bandwidth','1/3 octave', ...
    'SampleRate',fs);
centerFrequencies = getCenterFrequencies(SPL);

scope  = dsp.ArrayPlot(...
    'XDataMode','Custom', ...
    'CustomXData',centerFrequencies, ...
    'XLabel','Octave Band Center Frequencies (Hz)', ...
    'YLabel','Equivalent-Continuous Sound Level (dB)', ...
    'YLimits',[20 90], ...
    'ShowGrid',true, ...
    'Name','Sound Pressure Level Meter');
```

In an audio stream loop:

1. Read in the audio signal frame.
2. Play the audio signal to your output device.
3. Call the SPL meter to return the equivalent-continuous sound pressure level in dB.
4. Display the sound levels using the scope. Update the scope only when the equivalent-continuous sound pressure level has changed.

As a best practice, release your objects once complete.

```
LeqPrevious = zeros(size(centerFrequencies));
while ~isDone(source)
    x = source();
    player(x);
    [~,Leq] = SPL(x);

    for i = 1:size(Leq,1)
        if LeqPrevious ~= Leq(i,:)
            scope(Leq(i,:)')
            LeqPrevious = Leq(i,:);
        end
    end

end

release(source)
release(player)
release(SPL)
release(scope)
```



Algorithms

Sound pressure level calculations follow the algorithms described in [1]. You can specify property values to conform to standards [2] and [3].

Calibration

To account for environmental and input device effects in SPL measurements, the audio input is multiplied by a calibration factor:

$$x = \text{audioIn} \times \text{CalibrationFactor}$$

The `CalibrationFactor` property can be set directly, or by using the `calibrate` function, which compares a known level with acquired data. The known level is determined using a physical calibrator.

Frequency Weighting

A-, C-, or Z-frequency weighting is applied. The frequency weighting is implemented using the `weightingFilter` System object.

Analysis Bands

If you specify the `Bandwidth` property as `'1 octave'`, `'2/3 octave'` or `'1/3 octave'`, then the SPL calculations are applied to each octave or fractional-octave band. These analysis bands are determined after frequency weighting.

Time-Weighted Sound Level

Time-weighted sound level is defined as the ratio of the time-weighted root mean squared sound pressure to the reference sound pressure, converted to dB. That is,

$$Lt = 10\log_{10}\left\{\frac{\left(1/\tau\right)\int_s^t y(\xi)^2 e^{-(t-\xi)/\tau}d\xi}{p_o^2}\right\}$$
$$= 10\log_{10}\left\{\frac{h(y^2)}{p_o^2}\right\}$$

$h(y^2)$ can be interpreted as the convolution of y^2 with a filter with impulse response $\left(1/\tau\right)e^{-t/\tau}$. y is the output of the frequency-weighting filter. The impulse response corresponds to a lowpass filter of the form $H(s) = \frac{1/\tau}{s + 1/\tau}$. Using impulse invariance, the discrete filter can be interpreted as,

$$H(z) = \frac{1/(\tau \times fs)}{1 - e^{-1/(\tau \times fs)}z^{-1}} .$$

- τ is specified by the time-weighting coefficient as 0.125 (if `TimeWeighting` is set to `'Fast'`) or 1 (if `TimeWeighting` is set to`'Slow'`).
- *fs* is the sample rate specified by the `SampleRate` property.

Equivalent-Continuous Sound Level

Equivalent-continuous sound level is also called time-average sound level. It is defined as the ratio of root mean squared sound pressure to the reference sound pressure, converted to dB. That is,

$$Leq = 10\log_{10}\left\{\frac{\left(1/T\right)\int_1^{t_2} y^2 dt}{p_o^2}\right\}$$
$$= 20\log_{10}(\mathrm{rms}(y)/p_o)$$

where

- y is the output of the frequency-weighting filter.
- p_o is the reference sound pressure, specified by the `PressureReference` property.

Peak Sound Level

Peak sound level is defined as the ratio of peak sound pressure to the reference sound pressure, converted to dB. That is,

$$Lpeak = 20\log_{10}(\max(|y|)/p_o)$$

where

- y is the output of the frequency-weighting filter.
- p_o is the reference sound pressure, specified by the `PressureReference` property.

Max Time-Weighted Sound Level

Maximum time-weighted sound level is defined as the greatest time-weighted sound level within a stated time interval.

Version History

Introduced in R2018a

References

[1] Harris, Cyril M. *Handbook of Acoustical Measurements and Noise Control*. 3rd ed. American Institute of Physics, 1998.

[2] International Electrotechnical Commission. Electroacoustics - Sound level meters - Part 1: Specifications. IEC 61672-1:2013.

[3] American National Standards Institute. ANSI S1.4: Specification for Sound Level Meters. 1983.

Extended Capabilities

C/C++ Code Generation

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

Usage notes and limitations:

“System Objects in MATLAB Code Generation” (MATLAB Coder)

See Also

`loudnessMeter` | Loudness Meter | `integratedLoudness`

Topics

“Sound Pressure Measurement of Octave Frequency Bands”

calibrate

Calibrate meter using calibration tone with known level

Syntax

```
calibrate(SPL,micRecording,SPLreading)
```

Description

`calibrate(SPL,micRecording,SPLreading)` sets the `CalibrationFactor` property of the `splMeter` object. The calibration factor is based on the computed sound pressure level (SPL) of `micRecording` and the known `SPLreading`.

To calibrate, first set the SampleRate property of the `splMeter` object to match the `micRecording`, and the PressureReference and FrequencyWeighting properties to match the values from the physical SPL meter.

Input Arguments

`SPL` — `splMeter` System object

object

`splMeter` System object to be calibrated.

`micRecording` — Audio signal used to calibrate microphone

column vector

Audio signal used to calibrate microphone, specified as a column vector. `micRecording` must be acquired from the microphone you want to calibrate. The recording should consist of a 1 kHz test tone.

Data Types: `single` | `double`

`SPLreading` — Sound pressure level reported from physical meter (dB)

scalar

Sound pressure level (SPL) reported from physical meter in dB, specified as a scalar.

Data Types: `single` | `double`

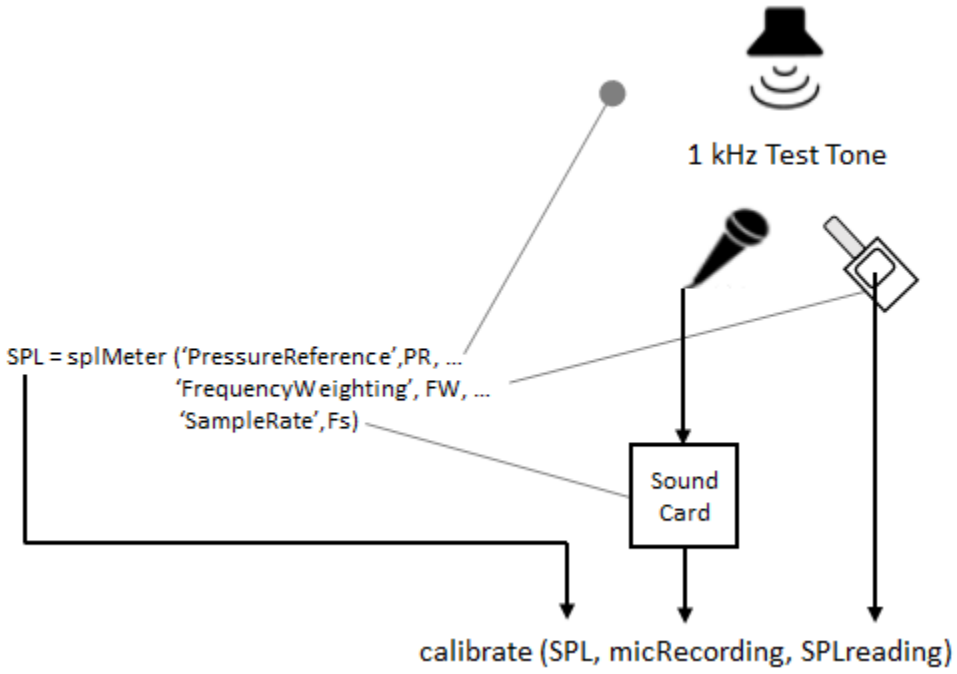
Algorithms

To set the `CalibrationFactor` property on an `splMeter` object, the `calibrate` function uses:

- A calibration tone recorded from the microphone you want to calibrate
- The sample rate used by your sound card for AD conversion.
- The known loudness, usually determined using a physical SPL meter.
- The frequency weighting used by your physical SPL meter.

- The atmospheric pressure at the recording location.

The diagram indicates a typical physical setup and the locations of required information.



The `CalibrationFactor` property is set according to the equation:

$$\text{CalibrationFactor} = \frac{10^{((\text{SPLreading}-k)/20)}}{\text{rms(x)}}$$

where `x` is the microphone recording passed through the weighting filter specified by the FrequencyWeighting property of the `splMeter` object. k is 1 pascal relative to the reference pressure calculated in dB:

$$k = 20\log_{10}\left(\frac{1}{\text{PressureReference}}\right).$$

Version History

Introduced in R2018a

See Also

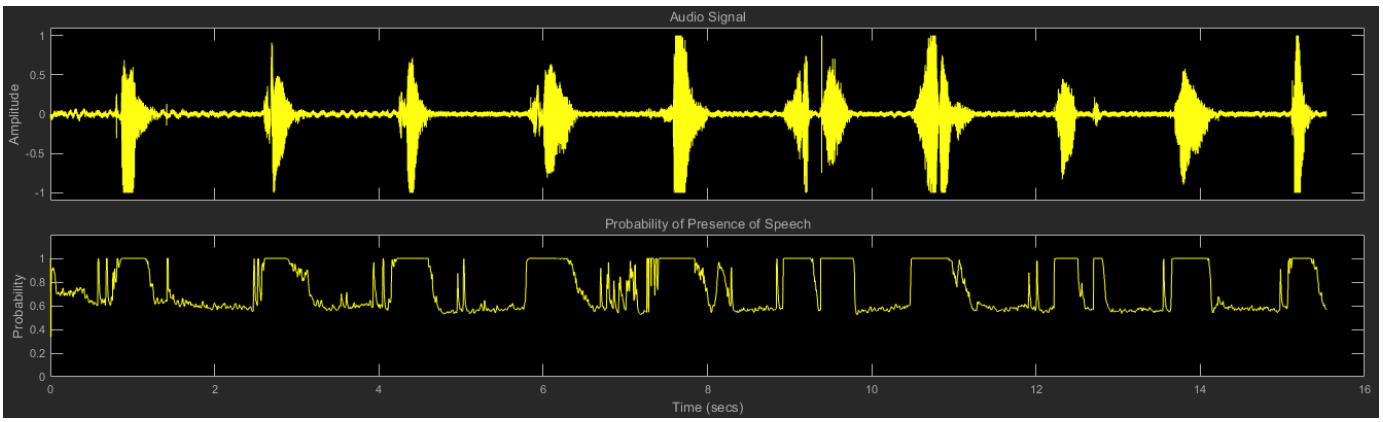
`splMeter` | `calibrateMicrophone`

voiceActivityDetector

Detect presence of speech in audio signal

Description

The `voiceActivityDetector` System object detects the presence of speech in an audio segment. You can also use the `voiceActivityDetector` System object to output an estimate of the noise variance per frequency bin.



To detect the presence of speech:

1. Create the `voiceActivityDetector` object and set its properties.
2. Call the object with arguments, as if it were a function.

To learn more about how System objects work, see What Are System Objects?

Creation

Syntax

```
VAD = voiceActivityDetector
VAD = voiceActivityDetector(Name,Value)
```

Description

`VAD = voiceActivityDetector` creates a System object, `VAD`, that detects the presence of speech independently across each input channel.

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

Example: `VAD = voiceActivityDetector('InputDomain','Frequency')` creates a System object, `VAD`, that accepts frequency-domain input.

Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see System Design in MATLAB Using System Objects.

`InputDomain` — Domain of input signal

`'Time'` (default) | `'Frequency'`

Domain of the input signal, specified as `'Time'` or `'Frequency'`.

Tunable: No

Data Types: `char` | `string`

`FFTLength` — FFT length

`[]` (default) | positive scalar

FFT length, specified as a positive scalar. The default is `[]`, which means that the `FFTLength` is equal to the number of rows of the input.

Tunable: No

Dependencies

To enable this property, set InputDomain to `'Time'`.

Data Types: `single` | `double`

`Window` — Window function for FFT

`'Hann'` (default) | `'Chebyshev'` | `'Flat Top'` | `'Hamming'` | `'Kaiser'` | `'Rectangular'`

Time-domain window function applied before calculating the discrete-time Fourier transform (DTFT), specified as `'Hann'`, `'Rectangular'`, `'Flat Top'`, `'Hamming'`, `'Chebyshev'`, or `'Kaiser'`.

The window function is designed using the algorithms of the following functions:

- Hann –– `hann`
- Chebyshev –– `chebwin`
- Flat Top –– `flattopwin`
- Hamming –– `hamming`
- Kaiser –– `kaiser`

Tunable: No

Dependencies

To enable this property, set InputDomain to `'Time'`.

Data Types: `char` | `string`

`SidelobeAttenuation` — Sidelobe attenuation of window (dB)

`60` (default) | real positive scalar

Sidelobe attenuation of the window in dB, specified as a real positive scalar.

Tunable: No

Dependencies

To enable this property, set InputDomain to `'Time'` and Window to `'Chebyshev'` or `'Kaiser'`.

Data Types: `single` | `double`

`SilenceToSpeechProbability` — Probability of transition from a frame of silence to a frame of speech

`0.2` (default) | scalar in the range [0,1]

Probability of transition from a frame of silence to a frame of speech, specified as a scalar in the range [0,1].

Tunable: Yes

Data Types: `single` | `double`

`SpeechToSilenceProbability` — Probability of transition from a frame of speech to a frame of silence

`0.1` (default) | scalar in the range [0,1]

Probability of transition from a frame of speech to a frame of silence, specified as a scalar in the range [0,1].

Tunable: Yes

Data Types: `single` | `double`

Usage

Syntax

```
[probability,noiseEstimate] = VAD(audioIn)
```

Description

`[probability,noiseEstimate] = VAD(audioIn)` applies a voice activity detector on the input, `audioIn`, and returns the probability that speech is present. It also returns the estimated noise variance per frequency bin.

Input Arguments

`audioIn` — Audio input to voice activity detector

scalar | vector | matrix

Audio input to the voice activity detector, specified as a scalar, vector, or matrix. If `audioIn` is a matrix, the columns are treated as independent audio channels.

The size of the audio input is locked after the first call to the `voiceActivityDetector` object. To change the size of `audioIn`, call `release` on the object.

If InputDomain is set to `'Time'`, `audioIn` must be real-valued. If `InputDomain` is set to `'Frequency'`, `audioIn` can be real-valued or complex-valued.

Data Types: `single` | `double`
Complex Number Support: Yes

Output Arguments

`probability` — Probability that speech is present
scalar | row vector

Probability that speech is present, returned as a scalar or row vector with the same number of columns as `audioIn`.

Data Types: `single` | `double`

`noiseEstimate` — Estimate of noise variance per frequency bin
column vector | matrix

Estimate of the noise variance per frequency bin, returned as a column vector or matrix with the same number of columns as `audioIn`.

Data Types: `single` | `double`

Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

Common to All System Objects

clone	Create duplicate System object
isLocked	Determine if System object is in use
release	Release resources and allow changes to System object property values and input characteristics
reset	Reset internal states of System object
step	Run System object algorithm

Examples

Detect Voice Activity

Use the default `voiceActivityDetector` System object™ to detect the presence of speech in a streaming audio signal.

Create an audio file reader to stream an audio file for processing. Define parameters to chunk the audio signal into 10 ms non-overlapping frames.

```
fileReader = dsp.AudioFileReader('Counting-16-44p1-mono-15secs.wav');
fs = fileReader.SampleRate;
fileReader.SamplesPerFrame = ceil(10e-3*fs);
```

Create a default `voiceActivityDetector` System object to detect the presence of speech in the audio file.

```
VAD = voiceActivityDetector;
```

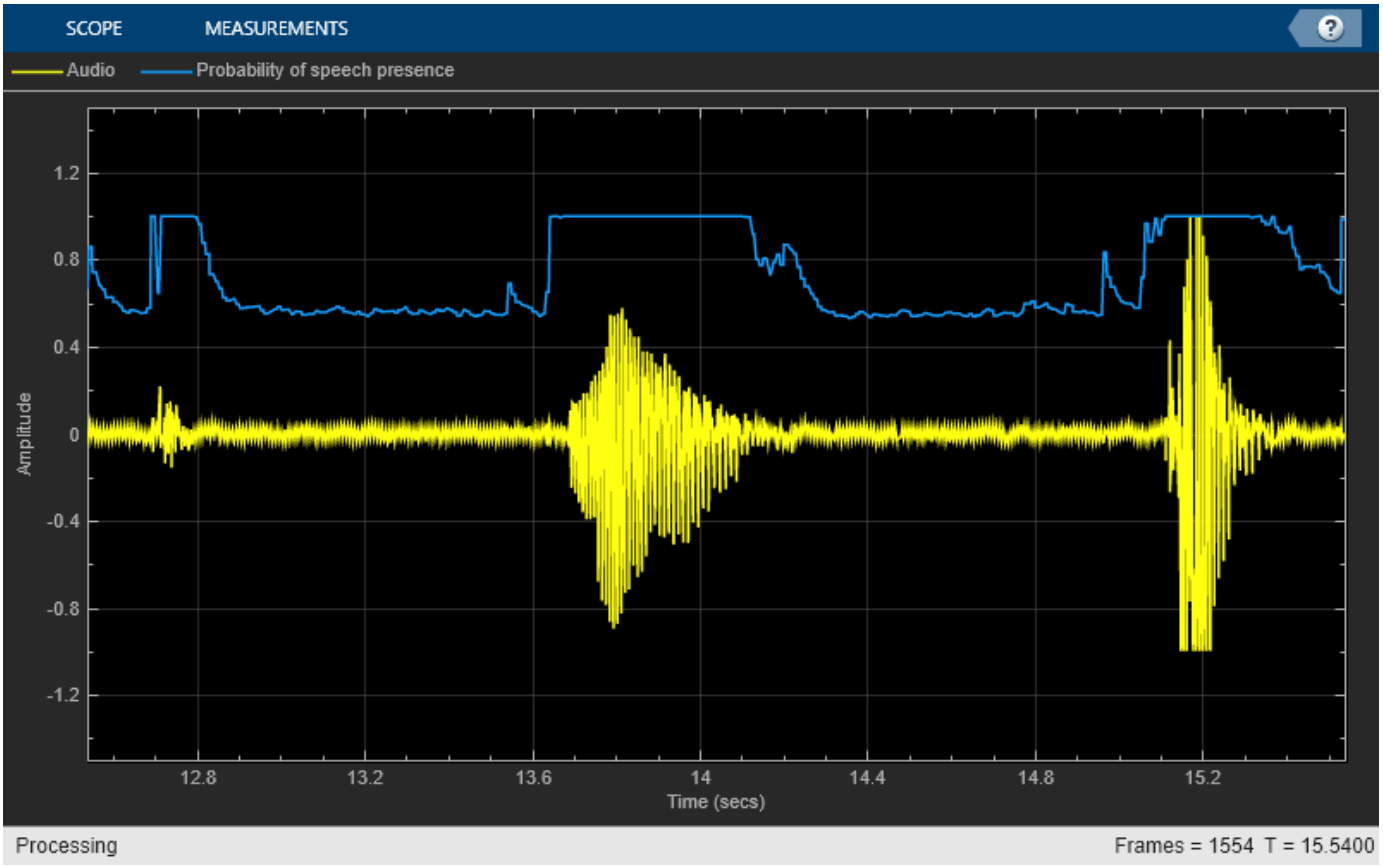
Create a scope to plot the audio signal and corresponding probability of speech presence as detected by the voice activity detector. Create an audio device writer to play the audio through your sound card.

```
scope = timescope( ...
    'NumInputPorts',2, ...
    'SampleRate',fs, ...
    'TimeSpanSource','Property','TimeSpan',3, ...
    'BufferLength',3*fs, ...
    'YLimits',[-1.5 1.5], ...
    'TimeSpanOverrunAction','Scroll', ...
    'ShowLegend',true, ...
    'ChannelNames',{'Audio','Probability of speech presence'});
deviceWriter = audioDeviceWriter('SampleRate',fs);
```

In an audio stream loop:

1. Read from the audio file.
2. Calculate the probability of speech presence.
3. Visualize the audio signal and speech presence probability.
4. Play the audio signal through your sound card.

```
while ~isDone(fileReader)
    audioIn = fileReader();
    probability = VAD(audioIn);
    scope(audioIn,probability*ones(fileReader.SamplesPerFrame,1))
    deviceWriter(audioIn);
end
```



Detect Voice Activity Using Overlapped Frames

Use a voice activity detector to detect the presence of speech in an audio signal. Plot the probability of speech presence along with the audio samples.

Create a `dsp.AudioFileReader` System object™ to read a speech file.

```
afr = dsp.AudioFileReader('Counting-16-44p1-mono-15secs.wav');
fs = afr.SampleRate;
```

Chunk the audio into 20 ms frames with 75% overlap between successive frames. Convert the frame time in seconds to samples. Determine the hop size (the increment of new samples). In the audio file reader, set the samples per frame to the hop size. Create a default `dsp.AsyncBuffer` object to manage overlapping between audio frames.

```
frameSize = ceil(20e-3*fs);
overlapSize = ceil(0.75*frameSize);
hopSize = frameSize - overlapSize;
afr.SamplesPerFrame = hopSize;

inputBuffer = dsp.AsyncBuffer('Capacity',frameSize);
```

Create a `voiceActivityDetector` System object. Specify an FFT length of 1024.

```
VAD = voiceActivityDetector('FFTLength',1024);
```

Create a scope to plot the audio signal and corresponding probability of speech presence as detected by the voice activity detector. Create an `audioDeviceWriter` System object to play audio through your sound card.

```
scope = timescope('NumInputPorts',2, ...
    'SampleRate',fs, ...
    'TimeSpanSource','Property','TimeSpan',3, ...
    'BufferLength',3*fs, ...
    'YLimits',[-1.5,1.5], ...
    'TimeSpanOverrunAction','Scroll', ...
    'ShowLegend',true, ...
    'ChannelNames',{'Audio','Probability of speech presence'});

player = audioDeviceWriter('SampleRate',fs);
```

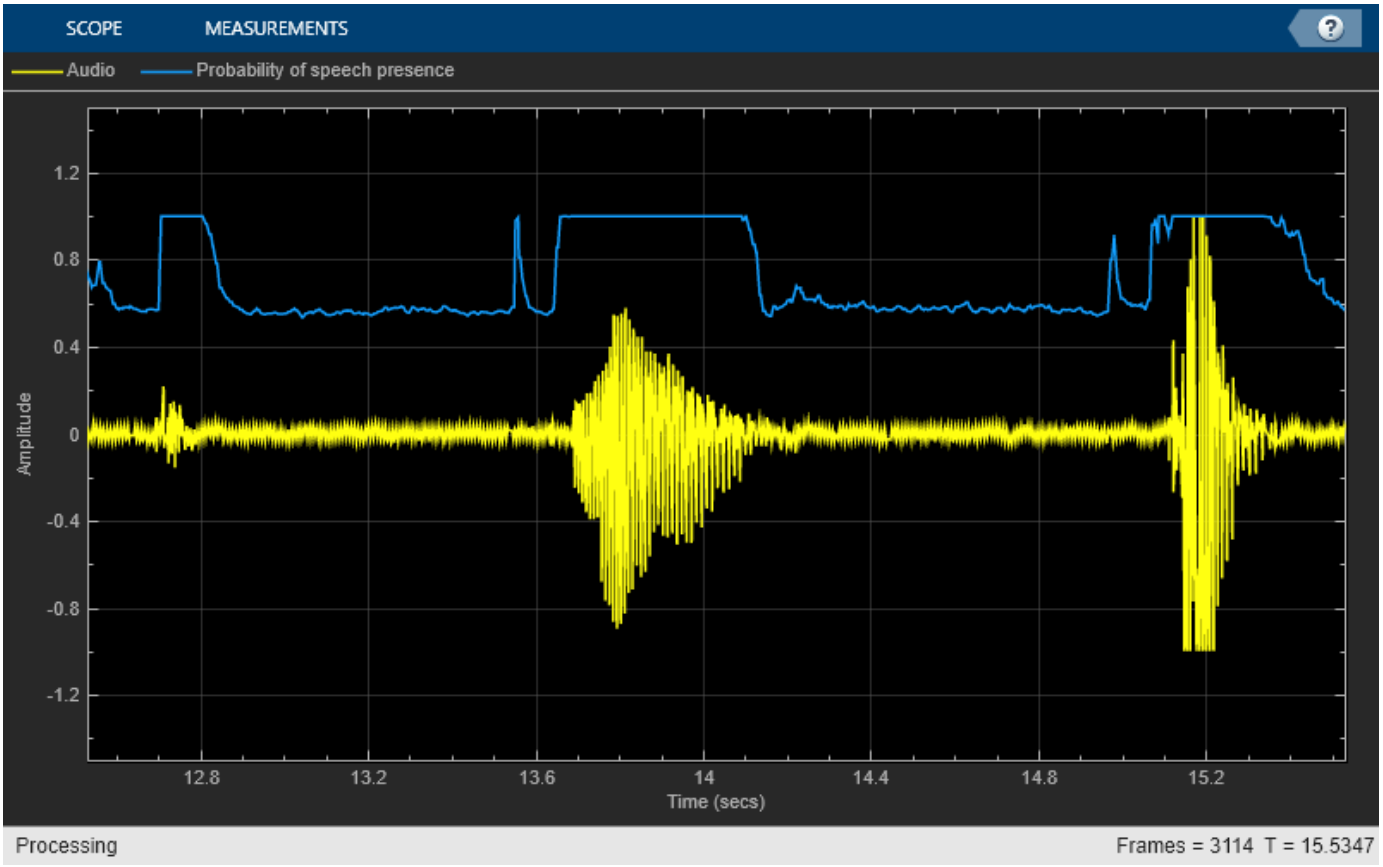
Initialize a vector to hold the probability values.

```
pHold = ones(hopSize,1);
```

In an audio stream loop:

1. Read a hop worth of samples from the audio file and save the samples into the buffer.
2. Read a frame from the buffer with specified overlap from the previous frame.
3. Call the voice activity detector to get the probability of speech for the frame under analysis.
4. Set the last element of the probability vector to the new probability decision. Visualize the audio and speech presence probability using the time scope.
5. Play the audio through your sound card.
6. Set the probability vector to the most recent result for plotting in the next loop.

```
while ~isDone(afr)
    x = afr();
    n = write(inputBuffer,x);

    overlappedInput = read(inputBuffer,frameSize,overlapSize);

    p = VAD(overlappedInput);

    pHold(end) = p;
    scope(x,pHold)

    player(x);

    pHold(:) = p;
end
```



Release the `player` once the audio finishes playing.

```
release(player)
```

Determine Pitch Contour of Streaming Audio

Create a `dsp.AudioFileReader` object to read in audio frame-by-frame.

```
fileReader = dsp.AudioFileReader("SingingAMajor-16-mono-18secs.ogg");
```

Create a `voiceActivityDetector` object to detect the presence of voice in streaming audio.

```
VAD = voiceActivityDetector;
```

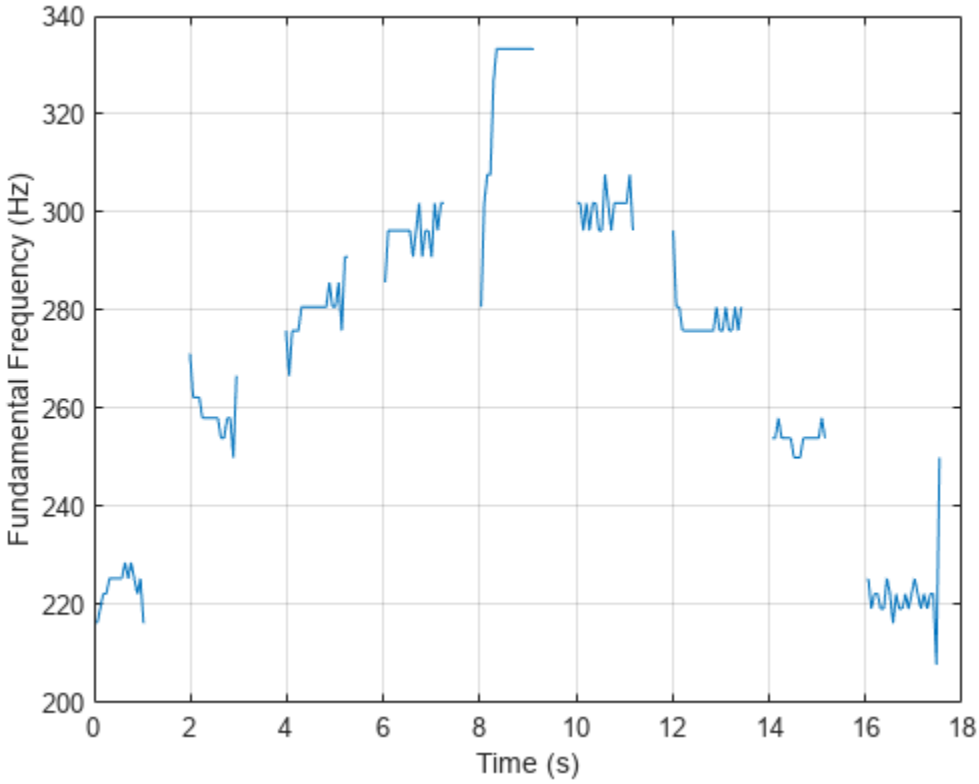
While there are unread samples, read from the file and determine the probability that the frame contains voice activity. If the frame contains voice activity, call `pitch` to estimate the fundamental frequency of the audio frame. If the frame does not contain voice activity, declare the fundamental frequency as `NaN`.

```
f0 = [];
while ~isDone(fileReader)
    x = fileReader();

    if VAD(x) > 0.99
        decision = pitch(x,fileReader.SampleRate, ...
```

```
            WindowLength=size(x,1), ...
            OverlapLength=0, ...
            Range=[200,340]);
    else
        decision = NaN;
    end
    f0 = [f0;decision];
end
```

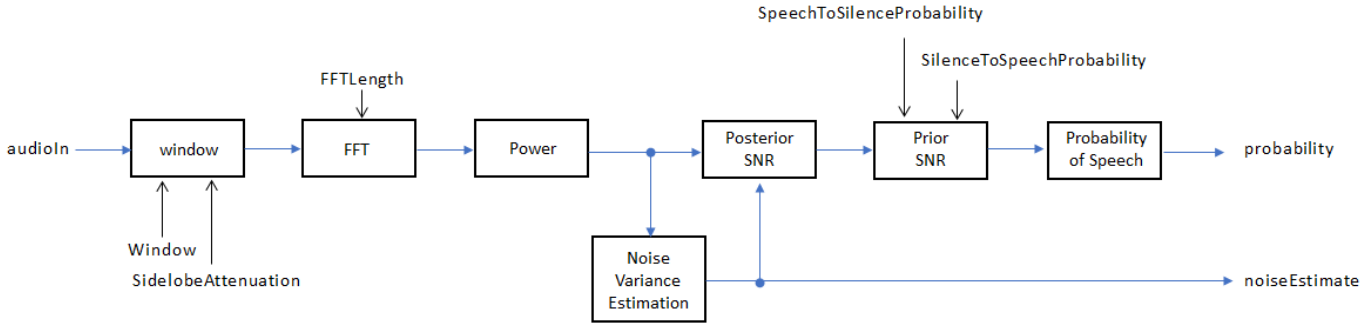
Plot the detected pitch contour over time.

```
t = linspace(0,(length(f0)*fileReader.SamplesPerFrame)/fileReader.SampleRate,length(f0));
plot(t,f0)
ylabel("Fundamental Frequency (Hz)")
xlabel("Time (s)")
grid on
```



Algorithms

The `voiceActivityDetector` implements the algorithm described in [1].



If `InputDomain` is specified as `'Time'`, the input signal is windowed and then converted to the frequency domain according to the `Window`, `SidelobeAttenuation`, and `FFTLength` properties. If `InputDomain` is specified as frequency, the input is assumed to be a windowed discrete time Fourier transform (DTFT) of an audio signal. The signal is then converted to the power domain. Noise variance is estimated according to [2]. The posterior and prior SNR are estimated according to the Minimum Mean-Square Error (MMSE) formula described in [3]. A log likelihood ratio test and Hidden Markov Model (HMM)-based hang-over scheme determine the probability that the current frame contains speech, according to [1].

Version History

Introduced in R2018a

References

[1] Sohn, Jongseo., Nam Soo Kim, and Wonyong Sung. "A Statistical Model-Based Voice Activity Detection." *Signal Processing Letters IEEE*. Vol. 6, No. 1, 1999.

[2] Martin, R. "Noise Power Spectral Density Estimation Based on Optimal Smoothing and Minimum Statistics." *IEEE Transactions on Speech and Audio Processing*. Vol. 9, No. 5, 2001, pp. 504–512.

[3] Ephraim, Y., and D. Malah. "Speech Enhancement Using a Minimum Mean-Square Error Short-Time Spectral Amplitude Estimator." *IEEE Transactions on Acoustics, Speech, and Signal Processing*. Vol. 32, No. 6, 1984, pp. 1109–1121.

Extended Capabilities

C/C++ Code Generation

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

Usage notes and limitations:

“System Objects in MATLAB Code Generation” (MATLAB Coder)

See Also

`audioFeatureExtractor` | `mfcc` | `pitch` | `cepstralCoefficients` | Voice Activity Detector

cepstralFeatureExtractor

(To be removed) Extract cepstral features from audio segment

Note The `cepstralFeatureExtractor` System object™ will be removed in a future release. For more information, see “Version History”.

Description

The `cepstralFeatureExtractor` System object extracts cepstral features from an audio segment. Cepstral features are commonly used to characterize speech and music signals.

To extract cepstral features:

1. Create the `cepstralFeatureExtractor` object and set its properties.
2. Call the object with arguments, as if it were a function.

To learn more about how System objects work, see What Are System Objects?

Creation

Syntax

```
cepFeatures = cepstralFeatureExtractor
cepFeatures = cepstralFeatureExtractor(Name,Value)
```

Description

`cepFeatures = cepstralFeatureExtractor` creates a System object, `cepFeatures`, that calculates cepstral features independently across each input channel. Columns of the input are treated as individual channels.

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

Example: `cepFeatures = cepstralFeatureExtractor('InputDomain','Frequency','SampleRate',fs,'LogEnergy','Replace')` accepts a signal in the frequency domain, sampled at `fs` Hz. The first element of the coefficients vector is replaced by the log energy value.

Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see System Design in MATLAB Using System Objects.

`FilterBank` — Type of filter bank
`'Mel'` (default) | `'Gammatone'`

Type of filter bank, specified as either `'Mel'` or `'Gammatone'`. When `FilterBank` is set to `Mel`, the object computes the mel frequency cepstral coefficients (MFCC). When `FilterBank` is set to `Gammatone`, the object computes the gammatone cepstral coefficients (GTCC).

Data Types: `char` | `string`

`InputDomain` — Domain of input signal
`'Time'` (default) | `'Frequency'`

Domain of the input signal, specified as either `'Time'` or `'Frequency'`.

Data Types: `char` | `string`

`NumCoeffs` — Number of coefficients to return
`13` (default) | positive integer

Number of coefficients to return, specified as an integer in the range [2, *v*], where *v* is the number of valid passbands. The number of valid passbands depends on the type of filter bank:

- `Mel` –– The number of valid passbands is defined as `sum(BandEdges <= floor(SampleRate/2))-2`.
- `Gammatone` –– The number of valid passbands is defined as `ceil(hz2erb(FrequencyRange(2))-hz2erb(FrequencyRange(1)))`.

Data Types: `single` | `double`

`Rectification` — Nonlinear rectification type
`'Log'` (default) | `'Cubic-Root'`

Nonlinear rectification type, specified as `'Log'` or `'Cubic-Root'`.

Data Types: `char` | `string`

`FFTLength` — FFT length
`[]` (default) | positive integer

FFT length, specified as a positive integer. The default, `[]`, means that the FFT length is equal to the number of rows in the input signal.

Dependencies

To enable this property, set `InputDomain` to `'Time'`.

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

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

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

- `'Append'` –– The object prepends the log energy to the coefficients vector. The length of the coefficients vector is 1 + `NumCoeffs`.

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

Data Types: `char` | `string`

`SampleRate` — Input sample rate (Hz)

`16000` (default) | positive scalar

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

Data Types: `single` | `double`

Advanced properties

`BandEdges` — Band edges of mel filter bank (Hz)

row vector

Band edges of the filter bank in Hz, specified as a nonnegative monotonically increasing row vector in the range [0, ∞). The maximum bandedge frequency can be any finite number. The number of bandedges must be in the range [4, 80].

The default band edges are spaced linearly for the first ten and then logarithmically after. The default band edges are set as recommended by [1].

Dependencies

To enable this property, set `FilterBank` to `Mel`.

Data Types: `single` | `double`

`FrequencyRange` — Frequency range of gammatone filter bank (Hz)

`[50 8000]` (default) | two-element row vector

Frequency range of the filter bank in Hz, specified as a positive, monotonically increasing two-element row vector. The maximum frequency can be any finite number. The center frequencies of the filter bank are equally spaced between `hz2erb(FrequencyRange(1))` and `hz2erb(FrequencyRange(2))` on the ERB scale.

Dependencies

To enable this property, set `FilterBank` to `Gammatone`.

Data Types: `single` | `double`

`FilterBankDesignDomain` — Domain for mel filter bank design

`'Hz'` (default) | `'Bin'`

Domain for filter bank design, specified as either `'Hz'` or `'Bin'`. The filter bank is designed as overlapped triangles with band edges specified by the `BandEdges` property.

The `BandEdges` property is specified in Hz. When you set the design domain to:

- `'Hz'` –– Filter bank triangles are drawn in Hz and are mapped onto bins.

  Here is an example that plots the filter bank in bins when the `FilterBankDesignDomain` is set to `'Hz'`:

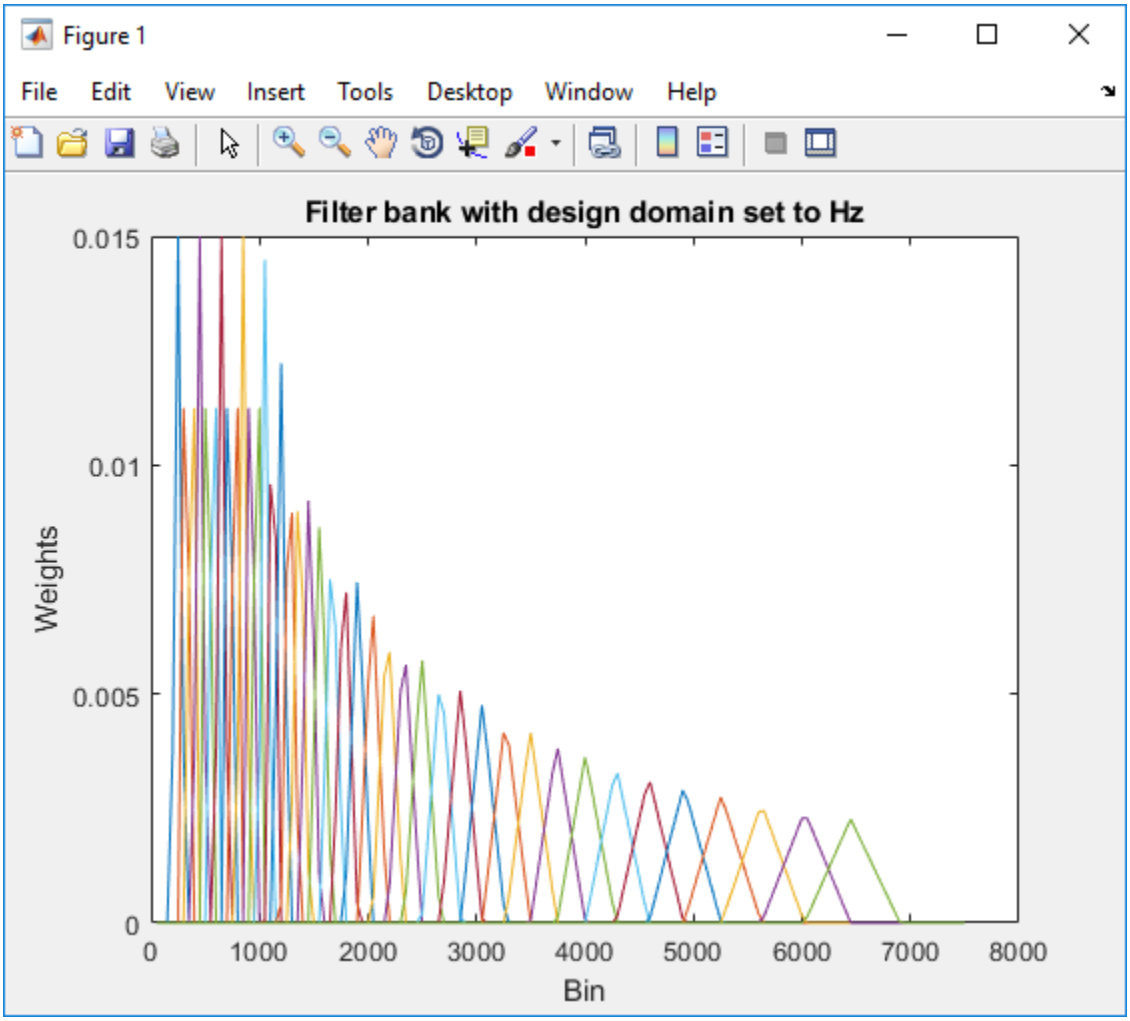
```
[audioFile, fs] = audioread('NoisySpeech-16-22p5-mono-5secs.wav');
duration = round(0.02*fs); % 20 ms audio segment
audioSegment = audioFile(5500:5500+duration-1);
cepFeatures = cepstralFeatureExtractor('SampleRate',fs)

cepFeatures =
 cepstralFeatureExtractor with properties:

   Properties
               InputDomain: 'Time'
                 NumCoeffs: 13
                 FFTLength: []
                 LogEnergy: 'Append'
                SampleRate: 22500

   Advanced Properties
                 BandEdges: [1×42 double]
     FilterBankDesignDomain: 'Hz'
    FilterBankNormalization: 'Bandwidth'
```

Pass the audio segment as an input to the cepstral feature extractor algorithm to lock the object.

```
[coeffs,delta,deltaDelta] = cepFeatures(audioSegment);
```

Use the `getFilters` function to get the filter bank. Plot the filter bank.

```
[filterbank, freq] = getFilters(cepFeatures);
plot(freq(1:150),filterbank(1:150,:))
```

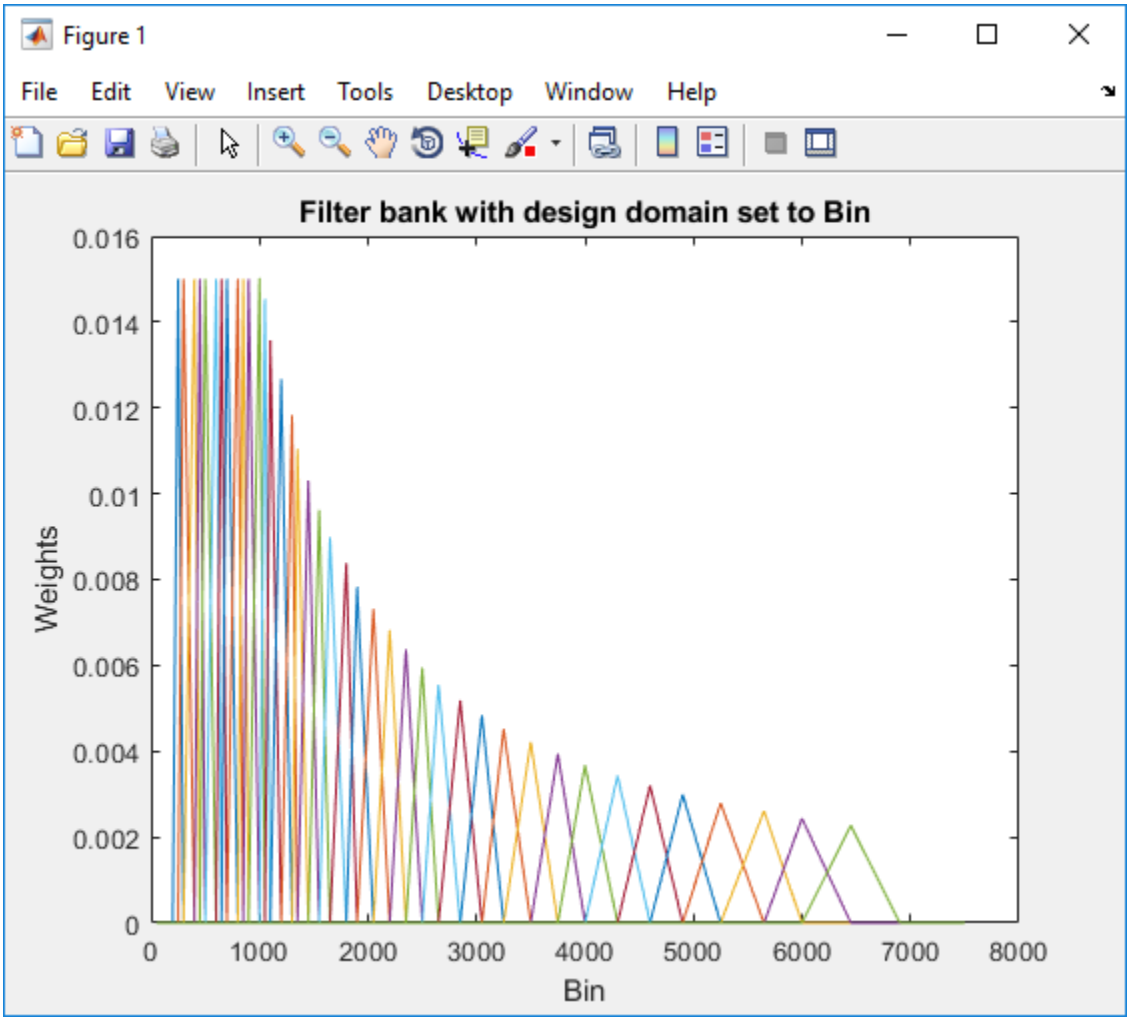


For details, see [1].

- `'Bin'` –– The bandedge frequencies in `'Hz'` are converted to bins. The filter bank triangles are drawn symmetrically in bins.

  Change the `FilterBankDesignDomain` property to `'Bin'`:

```
release(cepFeatures);
cepFeatures.FilterBankDesignDomain = 'Bin';
[coeffs,delta,deltaDelta] = cepFeatures(audioSegment);
[filterbank, freq] = getFilters(cepFeatures);
plot(freq(1:150),filterbank(1:150,:))
```



For details, see [2].

Dependencies

To enable this property, set `FilterBank` to `Mel`.

Data Types: `char` | `string`

`FilterBankNormalization` — Normalize filter bank

`'Bandwidth'` (default) | `'Area'` | `'None'`

Normalization technique used on the weights of the filter bank, specified as:

- `'Bandwidth'` -- The weights of each bandpass filter are normalized by the corresponding bandwidth of the filter.
- `'Area'` -- The weights of each bandpass filter are normalized by the corresponding area of the bandpass filter.
- `'None'` -- The weights of the filter are not normalized.

Data Types: `char` | `string`

Usage

Syntax

```
[coeffs,delta,deltaDelta] = cepFeatures(audioIn)
```

Description

`[coeffs,delta,deltaDelta] = cepFeatures(audioIn)` returns the cepstral coefficients, the log energy, the delta, and the delta-delta.

The log energy value prepends the coefficient vector or replaces the first element of the coefficients vector based on whether you set the `LogEnergy` property to `'Append'` or `'Replace'`. For details, see “coeffs” on page 3-0 .

Input Arguments

`audioIn` — Input signal

column vector | matrix

Input signal, specified as a column vector or matrix. If `InputDomain` is set to `'Time'`, specify `audioIn` as a real-valued frame of audio data. If `InputDomain` is set to `'Frequency'`, specify `audioIn` as a real- or complex-valued discrete Fourier transform. If specified as a matrix, the columns are treated as independent audio channels.

Data Types: `single` | `double`
Complex Number Support: Yes

Output Arguments

`coeffs` — Cepstral coefficients

column vector | matrix

Cepstral coefficients, returned as a column vector or a matrix. If the coefficients matrix is an *N*-by-*M* matrix, *N* is determined by the values you specify in `NumCoeffs` and `LogEnergy` properties. *M* equals the number of input audio channels.

When the `LogEnergy` property is set to:

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

Data Types: `single` | `double`

`delta` — Change in coefficients

column vector | matrix

Change in coefficients over consecutive calls to the algorithm, returned as a vector or a matrix. The `delta` array is of the same size and data type as the `coeffs` array.

In this example, `cepFeatures` is the cepstral feature extractor that accepts audio input signal sampled at 12 kHz. Stream in three segments of audio signal on three consecutive calls to the object algorithm. Return the cepstral coefficients of the filter bank and the corresponding `delta` values.

```
cepFeatures = cepstralFeatureExtractor('SampleRate',12000);
[coeff1,delta1] = cepFeatures(audioIn);
[coeff2,delta2] = cepFeatures(audioIn);
[coeff3,delta3] = cepFeatures(audioIn);
```

`delta2` is computed as `coeff2-coeff1`, while `delta3` is computed as `coeff3-coeff2`. The initial array, `delta1`, is an array of zeros.

Data Types: `single` | `double`

`deltaDelta` — Change in delta values

column vector | matrix

Change in `delta` values over consecutive calls to the algorithm, returned as a vector or a matrix. The `deltaDelta` array is the same size and data type as the `coeffs` and `delta` arrays.

In this example, consecutive calls to the cepstral feature extractor algorithm return the `deltaDelta` values in addition to the coefficients and the `delta` values.

```
cepFeatures = cepstralFeatureExtractor('SampleRate',12000);
[coeff1,delta1,deltaDelta1] = cepFeatures(audioIn);
[coeff2,delta2,deltaDelta2] = cepFeatures(audioIn);
[coeff3,delta3,deltaDelta3] = cepFeatures(audioIn);
```

`deltaDelta2` is computed as `delta2-delta1`, while `deltaDelta3` is computed as `delta3-delta2`. The initial array, `deltaDelta1`, is an array of zeros.

Data Types: `single` | `double`

Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

Specific to cepstralFeatureExtractor

getFilters	Get auditory filter bank

Common to All System Objects

clone	Create duplicate System object
isLocked	Determine if System object is in use
release	Release resources and allow changes to System object property values and input characteristics
reset	Reset internal states of System object
step	Run System object algorithm

Examples

Get MFCC Data for Speech Segment

Extract the mel frequency cepstral coefficients and the log energy values of segments in a speech file. Return `delta`, the difference between current and the previous cepstral coefficients, and

`deltaDelta`, the difference between the current and the previous `delta` values. The log energy value the object computes can prepend the coefficients vector or replace the first element of the coefficients vector. This is done based on whether you set the `LogEnergy` property of the `cepstralFeatureExtractor` object to `'Replace'` or `'Append'`.

Read an audio signal from `'Counting-16-44p1-mono-15secs.wav'` file. Extract a 40 ms segment from the audio data. Create a `cepstralFeatureExtractor` object. The cepstral coefficients computed by the default object are the mel frequency coefficients. In addition, the object computes the log energy, delta, and delta-delta values of the audio segment.

```
[audioFile, fs] = audioread('Counting-16-44p1-mono-15secs.wav');
duration = round(0.04*fs); % 40 ms
audioSegment = audioFile(40000:40000+duration-1);
cepFeatures = cepstralFeatureExtractor('SampleRate',fs)
```

```
cepFeatures =
  cepstralFeatureExtractor with properties:

   Properties
       FilterBank: 'Mel'
      InputDomain: 'Time'
        NumCoeffs: 13
    Rectification: 'Log'
        FFTLength: []
        LogEnergy: 'Append'
       SampleRate: 44100

  Show all properties
```

The `LogEnergy` property is set to `'Append'`. The first element in the coefficients vector is the log energy value and the remaining elements are the 13 cepstral coefficients computed by the object. The number of cepstral coefficients is determined by the value you specify in the `NumCoeffs` property.

```
[coeffs,delta,deltaDelta] = cepFeatures(audioSegment)
```

```
coeffs = 14×1

    5.2999
   -4.9406
    3.6130
    0.4397
   -0.2280
   -1.1068
    0.6679
    0.6367
   -0.3869
    0.6127
       ⋮


delta = 14×1

     0
     0
     0
     0
     0
```

```
     0
     0
     0
     0
     0
       ⋮
```

```
deltaDelta = 14×1

     0
     0
     0
     0
     0
     0
     0
     0
     0
     0
       ⋮
```

The initial values for the `delta` and `deltaDelta` arrays are always zero. Consider another 40 ms audio segment in the file and extract the cepstral features from this segment.

```
audioSegmentTwo = audioFile(5820:5820+duration-1);
[coeffsTwo,deltaTwo,deltaDeltaTwo] = cepFeatures(audioSegmentTwo)
```

```
coeffsTwo = 14×1

   -0.1582
  -15.9507
    2.4295
    0.2835
    0.4345
    0.4382
    0.6040
    0.4168
    0.1846
    0.2636
       ⋮
```

```
deltaTwo = 14×1

   -5.4581
  -11.0101
   -1.1836
   -0.1561
    0.6625
    1.5449
   -0.0639
   -0.2199
    0.5715
   -0.3491
       ⋮
```

```
deltaDeltaTwo = 14×1

   -5.4581
  -11.0101
   -1.1836
   -0.1561
    0.6625
    1.5449
   -0.0639
   -0.2199
    0.5715
   -0.3491
       ⋮
```

Verify that the difference between `coeffsTwo` and `coeffs` vectors equals `deltaTwo`.

```
isequal(coeffsTwo-coeffs,deltaTwo)
```

```
ans = logical
   1
```

Verify that the difference between `deltaTwo` and `delta` vectors equals `deltaDeltaTwo`.

```
isequal(deltaTwo-delta,deltaDeltaTwo)
```

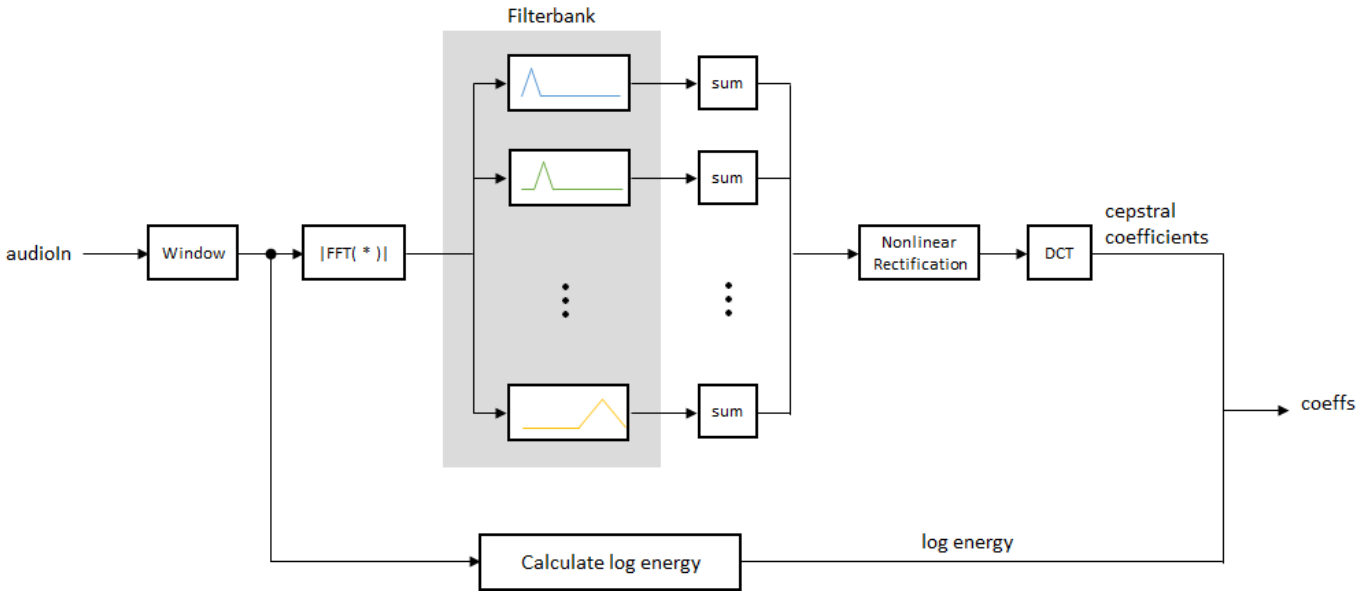
```
ans = logical
   1
```

Algorithms

Auditory Cepstrum Coefficients

Auditory cepstrum coefficients are popular features extracted from speech signals for use in recognition tasks. In the source-filter model of speech, cepstral coefficients are understood to represent the filter (vocal tract). The vocal tract frequency response is relatively smooth, whereas the source of voiced speech can be modeled as an impulse train. As a result, the vocal tract can be estimated by the spectral envelope of a speech segment.

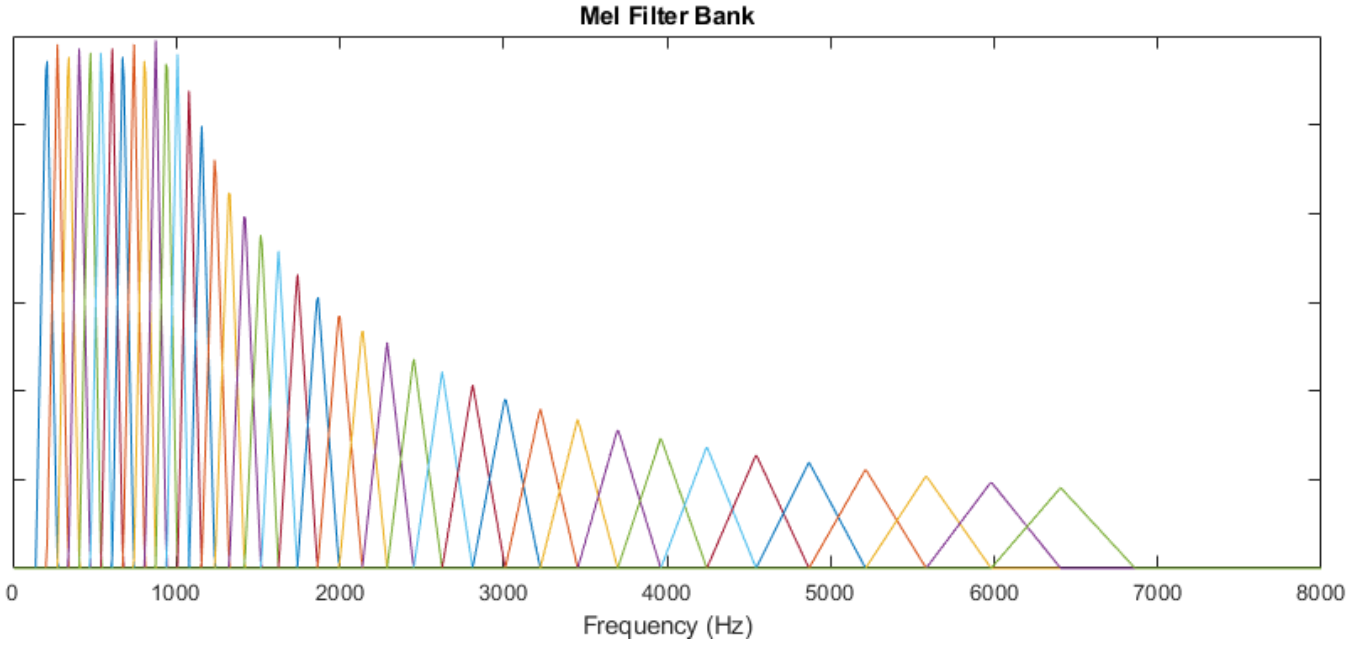
The motivating idea of cepstral coefficients is to compress information about the vocal tract (smoothed spectrum) into a small number of coefficients based on an understanding of the cochlea. Although there is no hard standard for calculating the coefficients, the basic steps are outlined by the diagram.



Two popular implementations of the filter bank are the mel filter bank and the gammatone filter bank.

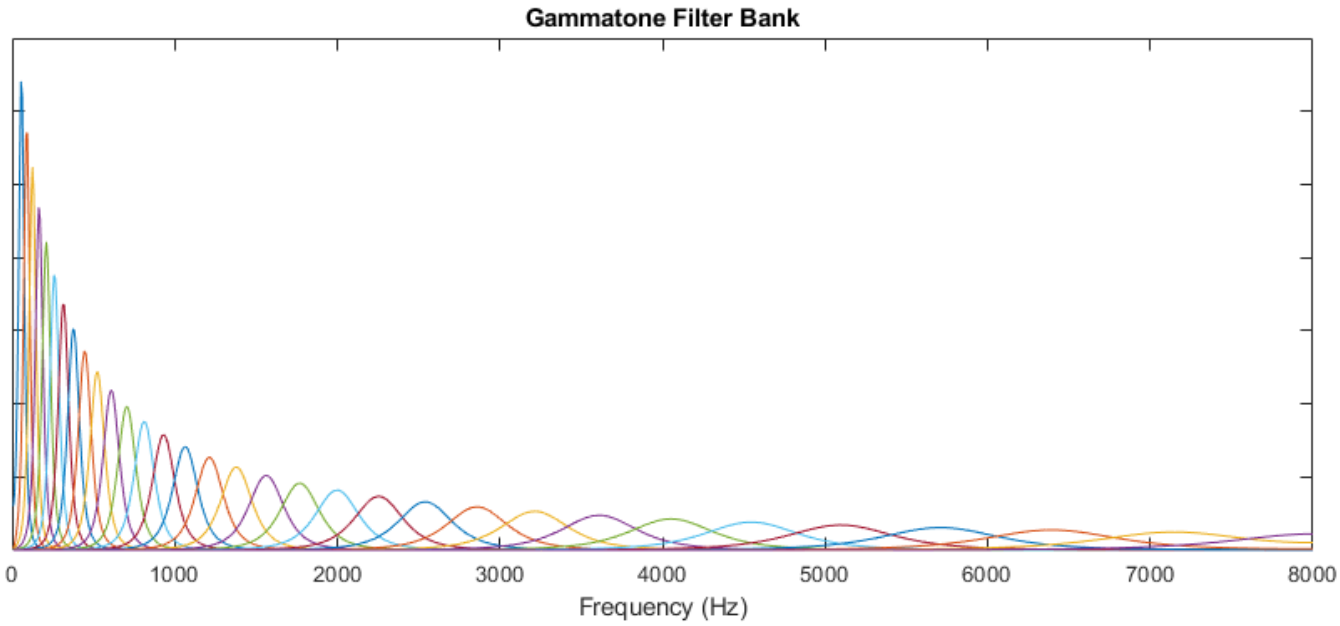
Mel Filter Bank

The default mel filter bank linearly spaces the first 10 triangular filters and logarithmically spaces the remaining filters.



Gammatone Filter Bank

The default gammatone filter bank is composed of gammatone filters spaced linearly on the ERB scale between 50 and 8000 Hz. The filter bank is designed by `gammatoneFilterBank`.



Log Energy

If the input (x) is a time-domain signal, the log energy is computed using the following equation:

$$\log E = \log(\text{sum}(x^2))$$

If the input (x) is a frequency-domain signal, the log energy is computed using the following equation:

$$\log E = \log\left(\text{sum}\left(|x|^2\right)/FFTLength\right)$$

Version History

Introduced in R2018a

R2022b: Warns

Warns starting in R2022b

The `cepstraFeatureExtractor` object issues a warning that it will be removed in a future release. Use the `mfcc` and `gtcc` functions to compute the same features for batch signals. For streaming applications, improve performance by designing the filter bank once with `designAuditoryFilterBank`, and then apply the filter bank and extract the same features with `cepstralCoefficients` and `audioDelta` in the streaming loop. If you are extracting multiple audio features, use the `audioFeatureExtractor` object.

cepstralFeatureExtractor Configuration	Recommended Replacement

`FilterBank` property set to `"Mel"`	Use the `mfcc` function.
	
	`[audioIn,fs] = audioread("Counting-16-44p1-mono-15secs.wav");`
	`[coeffs,delta,deltaDelta] = mfcc(audioIn,fs);`
	
	Alternatively, use a combination of `designAuditoryFilterBank` and `cepstralCoefficients`. See "Mel Frequency Cepstral Coefficients" on page 2-192 for an example. To extract delta features with this approach, use `audioDelta`.

`FilterBank` property set to `"Gammatone"`	Use the `gtcc` function.
	
	`[audioIn,fs] = audioread("Counting-16-44p1-mono-15secs.wav");`
	`[coeffs,delta,deltaDelta] = gtcc(audioIn,fs);`
	
	Alternatively, use a combination of `designAuditoryFilterBank` and `cepstralCoefficients`. See "Gammatone Frequency Cepstral Coefficients" on page 2-192 for an example. To extract delta features with this approach, use `audioDelta`.

`FilterBankDesignDomain` property set to `"Bin"`	No replacement
`FilterBankNormalization` property set to `"Area"` or `"None"`	Use the `designAuditoryFilterBank` function, with the `Normalization` name-value argument set to `"area"` or `"none"`, combined with the `cepstralCoefficients` function.

R2020b: To be removed
Not recommended starting in R2020b

The `cepstraFeatureExtractor` object runs without warning, but it will be removed in a future release.

References

[1] Auditory Toolbox. https://engineering.purdue.edu/~malcolm/interval/1998-010/AuditoryToolboxTechReport.pdf

[2] ETSI ES 201 108 V1.1.3 (2003-09). https://www.etsi.org/deliver/etsi_es/201100_201199/201108/01.01.03_60/es_201108v010103p.pdf

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

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

See Also

`mfcc` | `gtcc` | `gammatoneFilterBank` | `cepstralCoefficients` | `audioFeatureExtractor`

getFilters

Get auditory filter bank

Note The `cepstralFeatureExtractor` System object™ and `getFilters` object function will be removed in a future release.

Syntax

```
[filterbank,freq] = getFilters(cepFeatures)
```

Description

`[filterbank,freq] = getFilters(cepFeatures)` returns the filter bank and the corresponding frequency bins in Hz. Each column of the filter bank corresponds to a single bandpass filter. The filter bank is undefined until the object is locked.

Examples

Get Auditory Filter Bank

The auditory filter bank contains a set of bandpass filters. The `getFilters` function returns the auditory filter bank and the corresponding frequency bins.

Read an audio signal from `'SpeechDFT-16-8-mono-5secs.wav'` file. Extract a 40 ms segment from the audio data. Create a `cepstralFeatureExtractor` System object™ that accepts a time-domain audio input signal sampled at 8 kHz.

```
[audioFile, fs] = audioread('SpeechDFT-16-8-mono-5secs.wav');
duration = round(0.04*fs); % 40 ms
audioSegment = audioFile(5500:5500+duration-1);
cepFeatures = cepstralFeatureExtractor('SampleRate',fs)
```

```
cepFeatures =
  cepstralFeatureExtractor with properties:

   Properties
       FilterBank: 'Mel'
      InputDomain: 'Time'
        NumCoeffs: 13
    Rectification: 'Log'
        FFTLength: []
        LogEnergy: 'Append'
       SampleRate: 8000

  Show all properties
```

Pass the 40 ms audio segment as an input to the `cepstralFeatureExtractor` algorithm. The algorithm computes the mel frequency coefficients, log energy, delta, and delta-delta values of the audio segment.

```
[coeffs,delta,deltaDelta] = cepFeatures(audioSegment);
```

Input Arguments

`cepFeatures` — Input cepstral feature extractor System object

`cepstralFeatureExtractor` System object

Input cepstral feature extractor, specified as a `cepstralFeatureExtractor` System object. To use the `getFilters` function, the object must be locked. The filter bank is defined only when the object is locked. The object is locked when you call the object algorithm.

Output Arguments

`filterbank` — Auditory filter bank

matrix

Filter bank used to calculate cepstral features, returned as a matrix. Each column of the matrix corresponds to a single bandpass filter in the filter bank. The number of columns in the matrix is given by $m - 2$, where m is the length of the vector you specify in the `BandEdges` property of the System object. The number of rows in the matrix corresponds to the FFT length. By default, the FFT length equals the number of rows in the input signal. You can also specify the FFT length through the `FFTLength` property of the System object.

Data Types: `single` | `double`

`freq` — Frequency bins corresponding to filter bank (Hz)

row vector

Frequency bins corresponding to the filter bank in Hz, returned as a row vector. The length of the vector equals the FFT length.

Data Types: `single` | `double`

Version History

Introduced in R2018a

R2022b: To be removed

Warns starting in R2022b

The `cepstralFeatureExtractor` object and `getFilters` object function will be removed in a future release.

See Also

`cepstralFeatureExtractor`

staticCharacteristic

Return static characteristic of dynamic range controller

Syntax

```
outputLevel = staticCharacteristic(dynamicRangeController)
outputLevel = staticCharacteristic(dynamicRangeController,inputRange)
```

Description

`outputLevel = staticCharacteristic(dynamicRangeController)` returns the static characteristic of the dynamic range control object.

`outputLevel = staticCharacteristic(dynamicRangeController,inputRange)` enables you to specify the input range.

Examples

Get Output Level From Static Characteristic

Create a `limiter` System object™. Get the output level of the static characteristic over a specified range.

```
dynamicRangeLimiter = limiter;
inputLevel = -15:1:-5
```

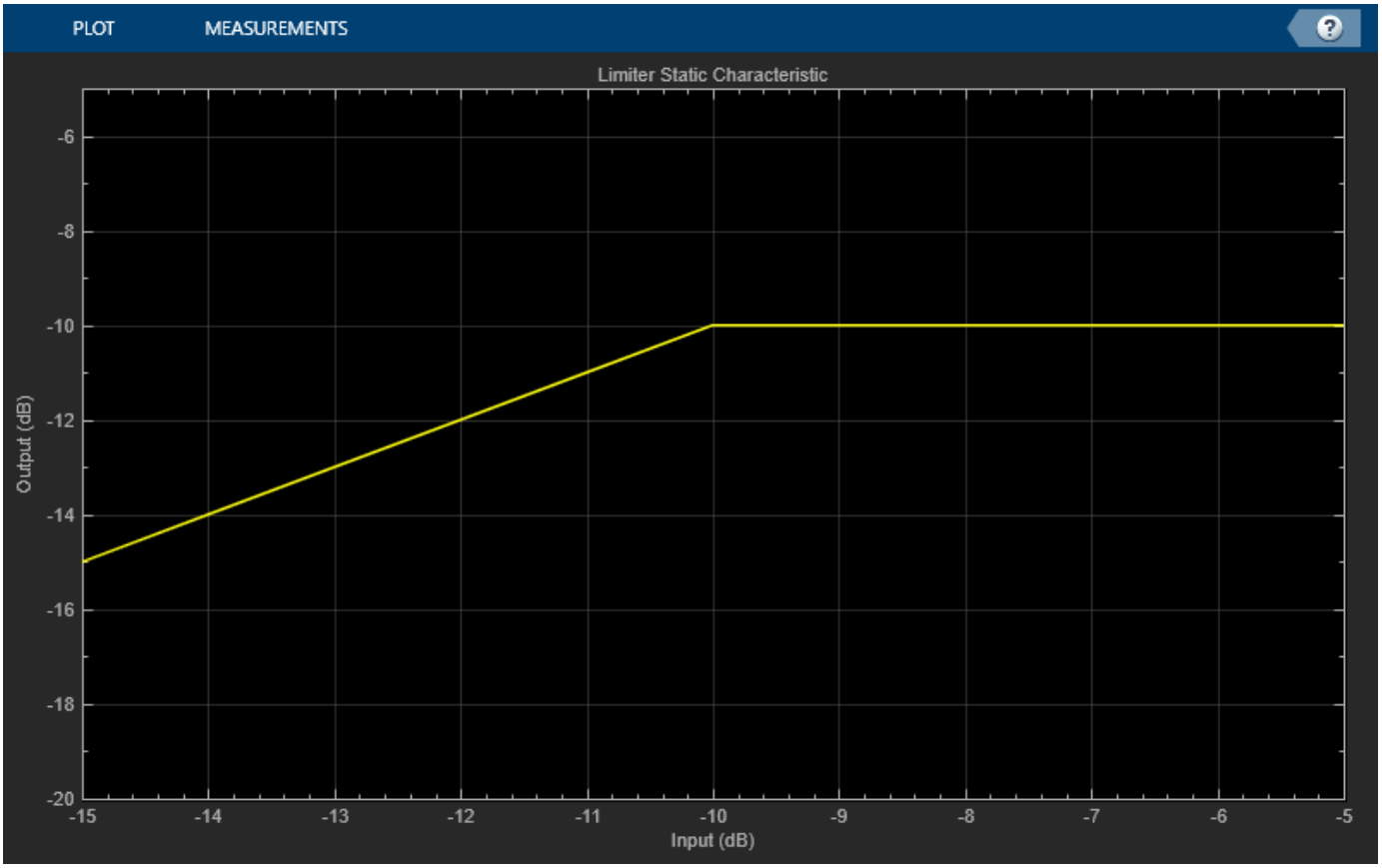
```
inputLevel = 1×11

   -15   -14   -13   -12   -11   -10    -9    -8    -7    -6    -5
```

```
outputLevel = staticCharacteristic(dynamicRangeLimiter,inputLevel)
```

```
outputLevel = 1×11

   -15   -14   -13   -12   -11   -10   -10   -10   -10   -10   -10
```

Plot the static characteristic. Modify the title to state that the object is a limiter.

```
hvsz = visualize(dynamicRangeLimiter,inputLevel);
hvsz.Title = "Limiter Static Characteristic";
```



Input Arguments

`dynamicRangeController` — Dynamic range control object

object

Dynamic range control object, specified as a `compressor`, `expander`, or `limiter` System object.

`inputRange` — Range to calculate static characteristic output

`[-50:0.01:0]` (default) | vector of monotonically increasing values

Range over which to calculate the output of the static characteristic, specified as a vector of monotonically increasing values expressed in dB. The default input range is `[-50:0.01:0]` dB.

Output Arguments

`outputLevel` — Output level (dB)

vector

Output level in dB, returned as a vector the same size as `inputRange`.

Version History

Introduced in R2022a

See Also

`compressor` | `expander` | `limiter`

Topics

“Dynamic Range Control”

visualize

Visualize static characteristic of dynamic range controller

Syntax

```
visualize(dynamicRangeController)
visualize(dynamicRangeController,inputRange)
hvsz = visualize( ___ )
```

Description

`visualize(dynamicRangeController)` plots the static characteristic of the dynamic range control object. The plot is updated automatically when properties of the object change.

`visualize(dynamicRangeController,inputRange)` enables you to specify the input range.

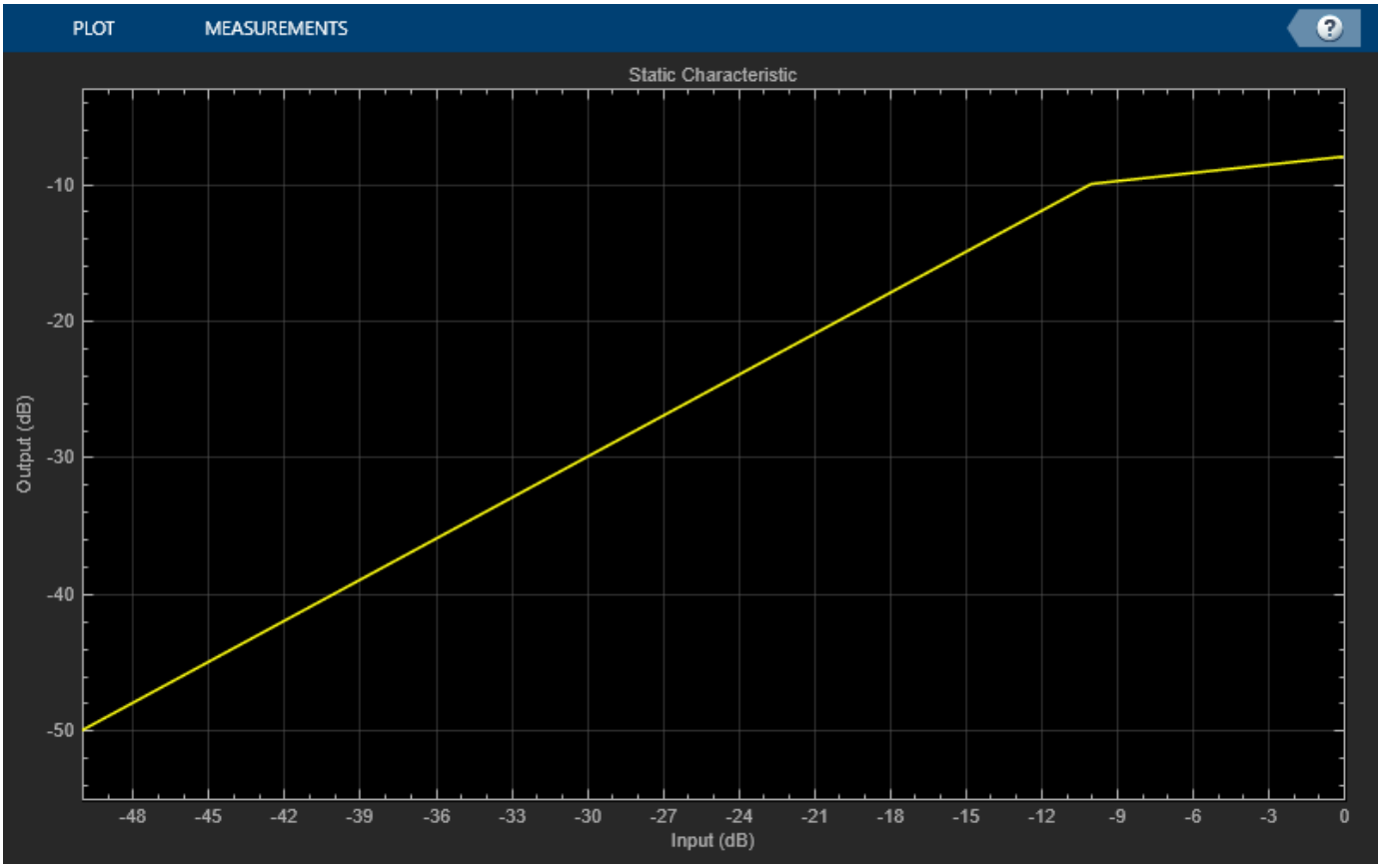
`hvsz = visualize( ___ )` returns a handle to the visualizer when called with any of the previous syntaxes.

Examples

Plot Static Characteristic

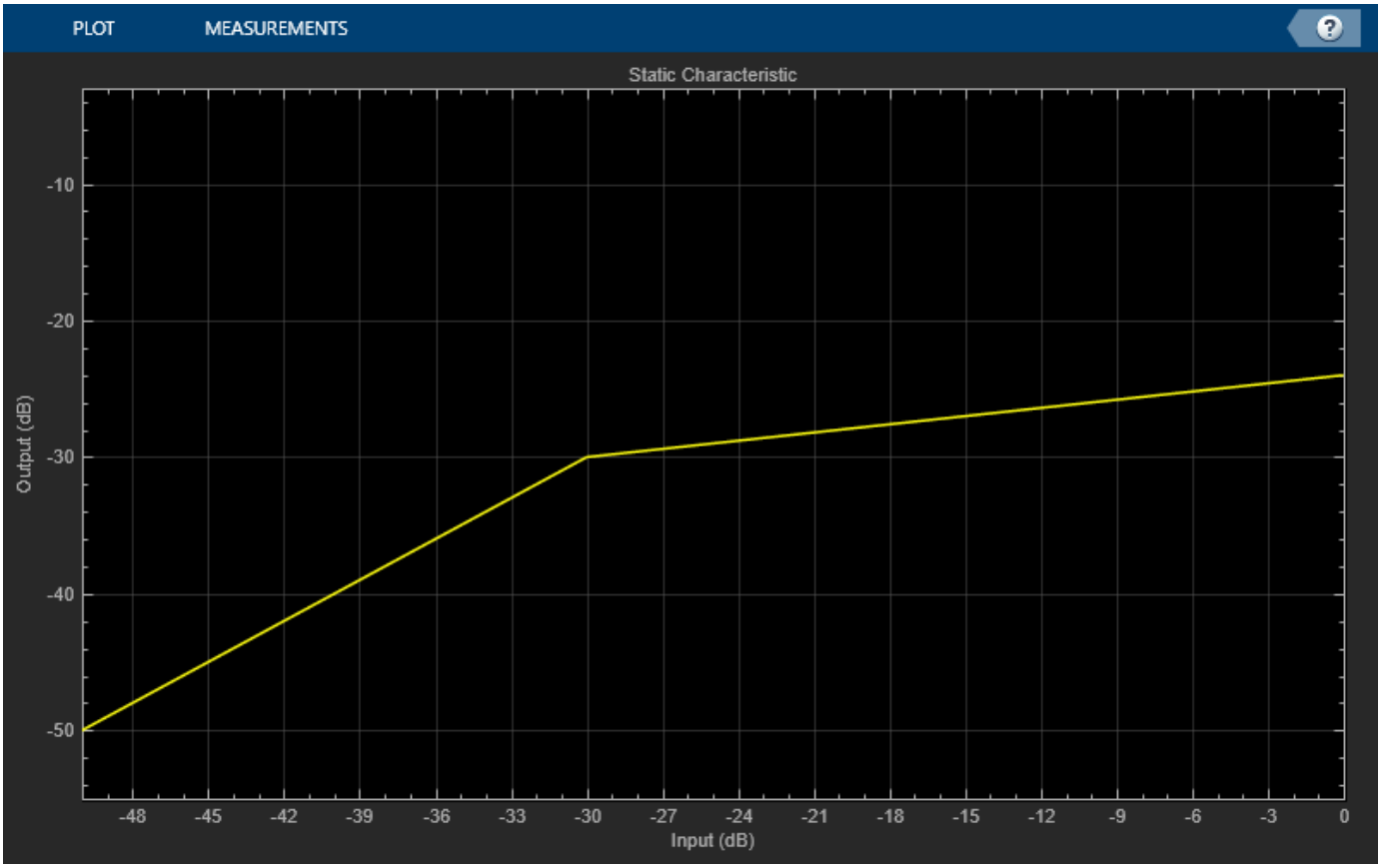
Create a `compressor` System object™, and then plot the static characteristic.

```
dynamicRangeCompressor = compressor;
visualize(dynamicRangeCompressor)
```



The static characteristic plot updates automatically if you modify a property of the object.

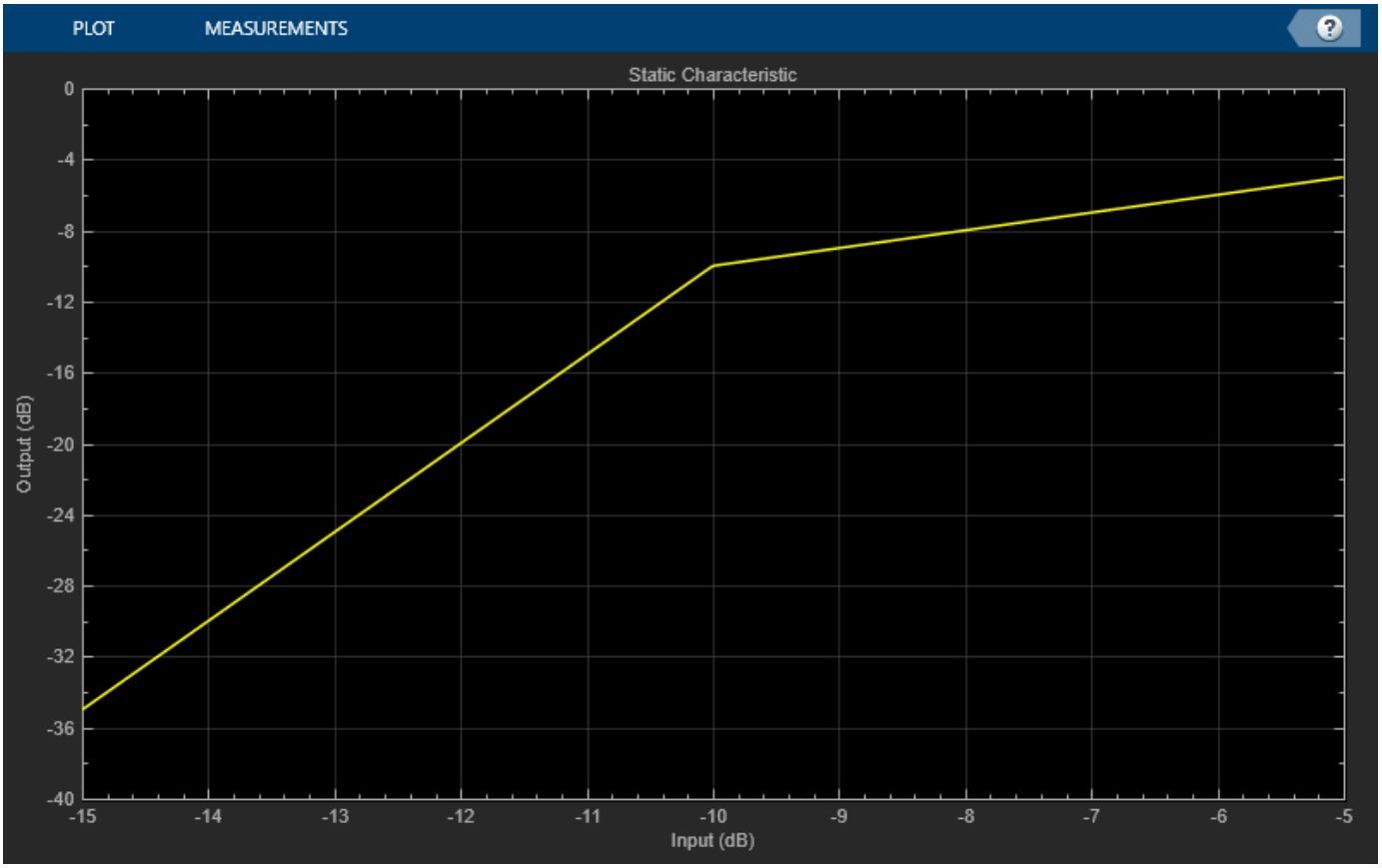
```
dynamicRangeCompressor.Threshold = -30;
```



Specify Range of Static Characteristic Plot

Create an `expander` System object™. Plot the static characteristic over the range –15 to –5, in 0.001 dB increments.

```
dynamicRangeExpander = expander;
visualize(dynamicRangeExpander,-15:0.001:-5)
```



Get Output Level From Static Characteristic

Create a `limiter` System object™. Get the output level of the static characteristic over a specified range.

```
dynamicRangeLimiter = limiter;
inputLevel = -15:1:-5
```

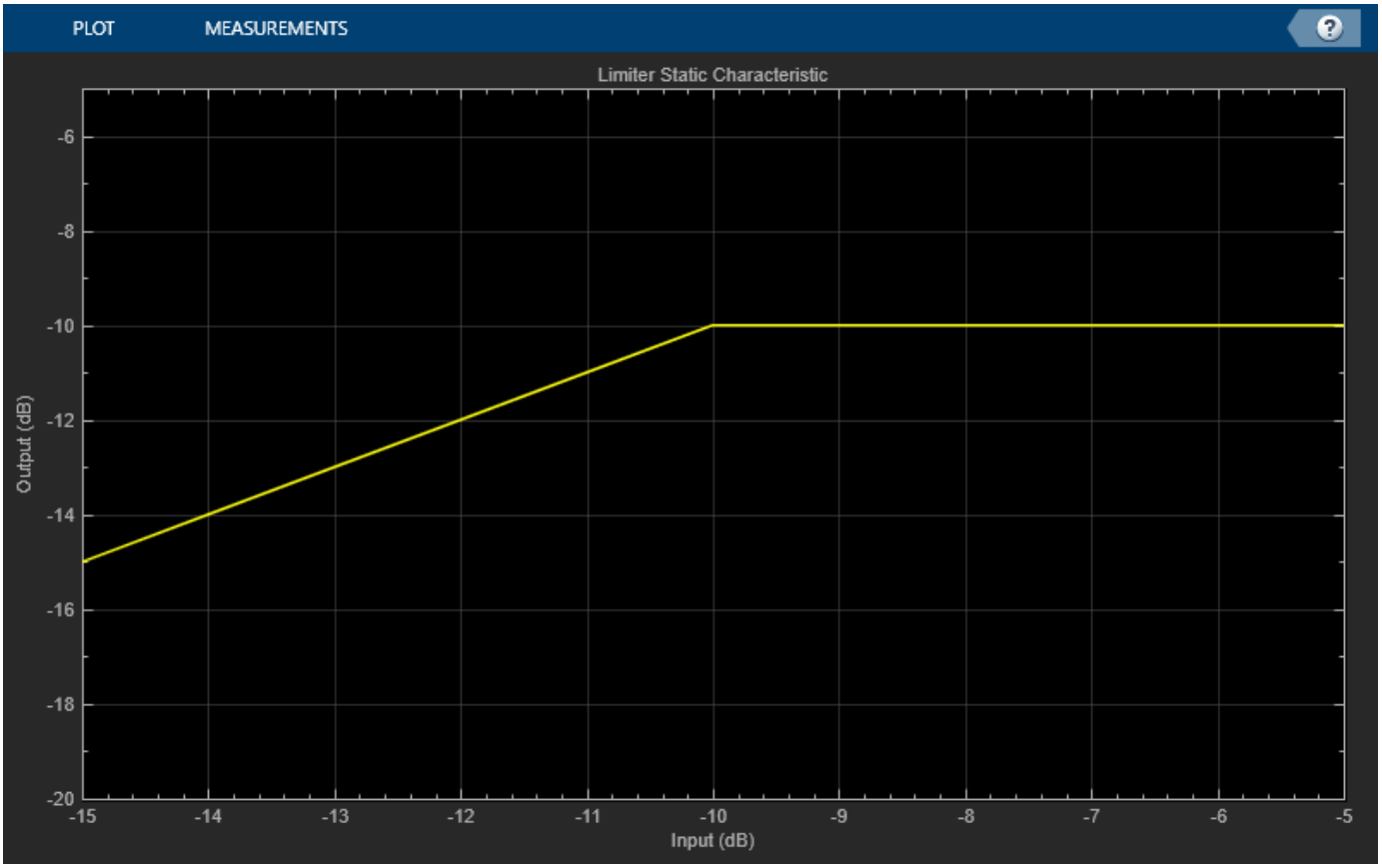
```
inputLevel = 1×11

   -15   -14   -13   -12   -11   -10    -9    -8    -7    -6    -5
```

```
outputLevel = staticCharacteristic(dynamicRangeLimiter,inputLevel)
```

```
outputLevel = 1×11

   -15   -14   -13   -12   -11   -10   -10   -10   -10   -10   -10
```

Plot the static characteristic. Modify the title to state that the object is a limiter.

```
hvsz = visualize(dynamicRangeLimiter,inputLevel);
hvsz.Title = "Limiter Static Characteristic";
```



Input Arguments

`dynamicRangeController` — Dynamic range control object

object

Dynamic range control object, specified as a `compressor`, `expander`, `limiter`, or `noiseGate` System object.

`inputRange` — Range to calculate static characteristic output

`[-50:0.01:0]` (default) | vector of monotonically increasing values

Range over which to calculate the output of the static characteristic, specified as a vector of monotonically increasing values expressed in dB. The default input range is `[-50:0.01:0]` dB.

Output Arguments

`hvsz` — Visualizer handle

`dsp.ArrayPlot` object

Visualizer handle, returned as a `dsp.ArrayPlot` object.

Version History

Introduced in R2016a

R2022a: `visualize` object function outputs figure handles

Behavior changed in R2022a

In previous releases, calling the `visualize` function of the dynamic range control System objects with an output argument returned static characteristic data. Starting in R2022a, that syntax returns a figure handle instead. Use the new `staticCharacteristic` function to compute static dynamic range characteristics.

R2022a: `visualize` plots static characteristics of `noiseGate` System objects in dB

Behavior changed in R2022a

In previous releases, the `visualize` function plotted the static characteristic of a `noiseGate` System object in linear units. Starting in R2022a, the function plots all static characteristics in dB.

See Also

`compressor` | `expander` | `limiter` | `noiseGate`

Topics

“Dynamic Range Control”

createAudioPluginClass

Create audio plugin class that implements functionality of System object

Syntax

```
createAudioPluginClass(obj)
createAudioPluginClass(obj,pluginName)
```

Description

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

If the object is locked, the number of input and output channels of the plugin is equal to the number of channels of the object. Otherwise, the number of channels is equal to 2.

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

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

Examples

Create an Audio Plugin Class From a System Object

Create a `compressor` object. Call `createAudioPluginClass` to create a System object™ plugin class that implements the functionality of the compressor object.

```
cmpr = compressor;
createAudioPluginClass(cmpr)
```

Specify Name of Created Plugin Class

Create an object of the `reverberator` System object™. Call `createAudioPluginClass` to create a System object™ plugin class that implements the functionality of the `reverberator` object, specifying the plugin class name as the second argument.

```
reverb = reverberator;
createAudioPluginClass(reverb,'Garage');
```

Input Arguments

`obj` — System object to create plugin class from

Audio Toolbox System object

System object from which to create a plugin class.

`pluginName` — Name of created plugin class
character vector

Name of created plugin class, specified as a character vector with fewer than 64 elements.

Data Types: `char`

Version History

Introduced in R2016a

See Also

Objects
`compressor` | `audioOscillator` | `crossoverFilter` | `expander` | `graphicEQ` | `limiter` | `multibandParametricEQ` | `noiseGate` | `octaveFilter` | `reverberator` | `wavetableSynthesizer` | `weightingFilter`

Apps
Audio Test Bench

Functions
`parameterTuner`

Topics
“Audio Plugins in MATLAB”
“Export a MATLAB Plugin to a DAW”

getFilter

Return biquad filter object with design parameters set

Syntax

```
biquad = getFilter(obj)
```

Description

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

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

Examples

Get Biquad Filter for Octave Filter Design

Create an `octaveFilter` System object™. Call `getFilter` on your object to return a `dsp.BiquadFilter` object with design parameters specified by your `octaveFilter` System object.

```
octFilt = octaveFilter;
biquad = getFilter(octFilt)
```

```
biquad =
  dsp.SOSFilter with properties:

            Structure: 'Direct form II transposed'
    CoefficientSource: 'Property'
            Numerator: [3x3 double]
          Denominator: [3x3 double]
       HasScaleValues: false

  Show all properties
```

Get Biquad Filter for Weighting Filter Design

Create a `weightingFilter` System object™.

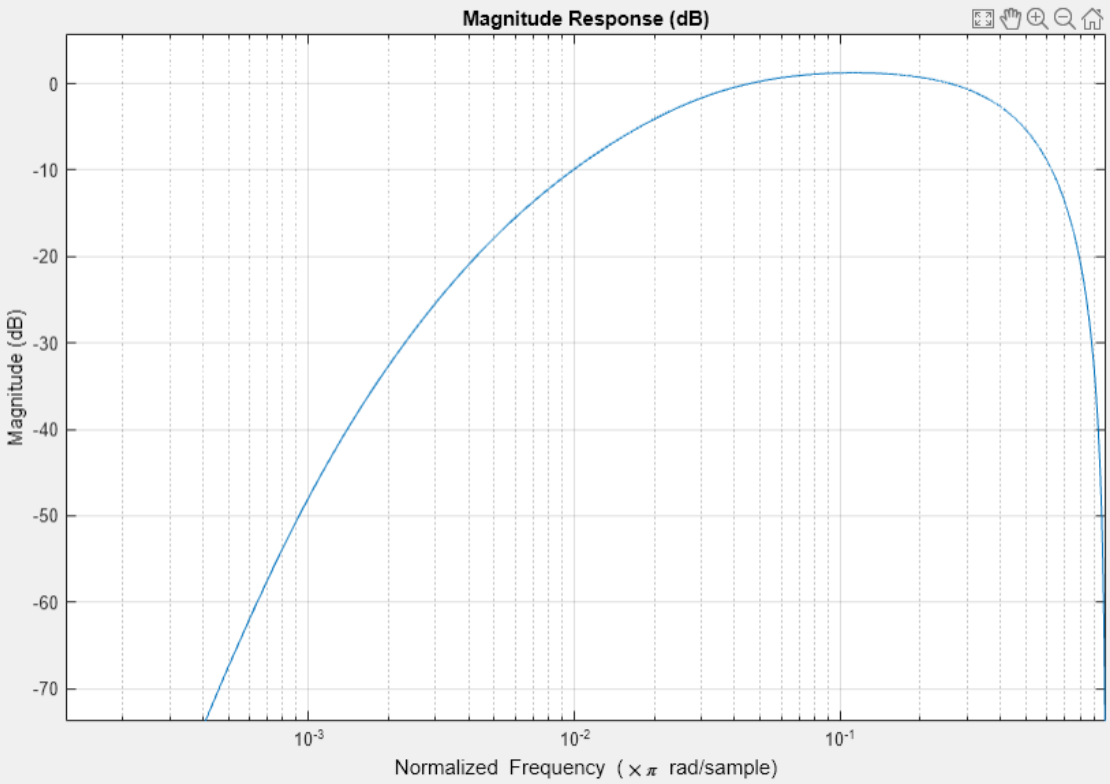
```
weightFilt = weightingFilter;
```

Call `getFilter` on your object to return a `dsp.BiquadFilter` object with design parameters specified by your `weightingFilter` System object. Use `fvtool` to visualize the biquad filter.

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

```
biquad =
  dsp.SOSFilter with properties:

            Structure: 'Direct form II transposed'
    CoefficientSource: 'Property'
            Numerator: [3x3 double]
          Denominator: [3x3 double]
       HasScaleValues: true
          ScaleValues: [4x1 double]

  Show all properties
```

```
fvtool(biquad,'FrequencyScale','log')
```



Input Arguments

`obj` — System object to get filter from

System object

System object that you want to get a biquad filter object from.

Output Arguments

`biquad` — Object of `dsp.BiquadFilter`
object

`dsp.BiquadFilter` object.

Version History

Introduced in R2016b

See Also

`weightingFilter` | `octaveFilter` | `dsp.BiquadFilter`

Topics
“Audio Weighting Filters”
“Octave-Band and Fractional Octave-Band Filters”
“Sound Pressure Measurement of Octave Frequency Bands”

info

Get audio device information

Syntax

```
infoStruct = info(obj)
```

Description

`infoStruct = info(obj)` returns a structure, `infoStruct`, containing information about the System object, `obj`.

Examples

Get Input Audio Device Information

Create an object of the `audioDeviceReader` System object™ and then call `info` to return a structure containing information about the selected driver, device name, and the maximum number of input channels.

```
deviceReader = audioDeviceReader;
info(deviceReader)
```

Get Output Audio Device Information

Create an object of the `audioDeviceWriter` System object™ and then call `info` to return a structure containing information about the selected driver, device name, and the maximum number of output channels.

```
deviceWriter = audioDeviceWriter;
info(deviceWriter)
```

Get Audio I/O Device Information

Create an object of the `audioPlayerRecorder` System object™ and then call `info` to return a structure containing information about the selected driver, device name, and the maximum number of input and output channels.

```
playRec = audioPlayerRecorder;
info(playRec)
```

Input Arguments

`obj` — System object to get information from

System object

System object to get information from.

Output Arguments

`infoStruct` — Struct containing object information

struct

Struct containing information about the System object, `obj`. Fields of the struct depend on the System object.

Version History

Introduced in R2016a

See Also

`audioDeviceWriter` | `audioDeviceReader` | `audioPlayerRecorder`

cost

Estimate implementation cost of audio System objects

Syntax

```
implementationCost = cost(audioObj)
```

Description

`implementationCost = cost(audioObj)` returns a structure, `implementationCost`, whose fields contain information about the computation cost of implementing the audio System object, `audioObj`.

Examples

Estimate Implementation Cost of Crossover Filter

Create a crossover filter with 2 crossovers with 48 dB/octave slopes. Call `cost` to get an estimate of the implementation cost.

```
crossFilt = crossoverFilter('NumCrossovers',2,'CrossoverSlopes',48);
cost1 = cost(crossFilt)
```

```
cost1 = struct with fields:
                  NumCoefficients: 120
                        NumStates: 48
    MultiplicationsPerInputSample: 120
          AdditionsPerInputSample: 97
```

Reduce the crossover slopes for both crossovers to 12 dB/octave. Call `cost` to get an estimate of the new implementation cost.

```
crossFilt.CrossoverSlopes = 12;
cost2 = cost(crossFilt)
```

```
cost2 = struct with fields:
                  NumCoefficients: 36
                        NumStates: 12
    MultiplicationsPerInputSample: 36
          AdditionsPerInputSample: 25
```

Input Arguments

audioObj — Audio System object

`crossoverFilter` object

Specify the input as a supported audio System object.

Data Types: `object`

Output Arguments

`implementationCost` — Estimate of implementation cost

struct

Estimate of the implementation cost of a filter, returned as struct:

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

Version History

Introduced in R2016a

See Also

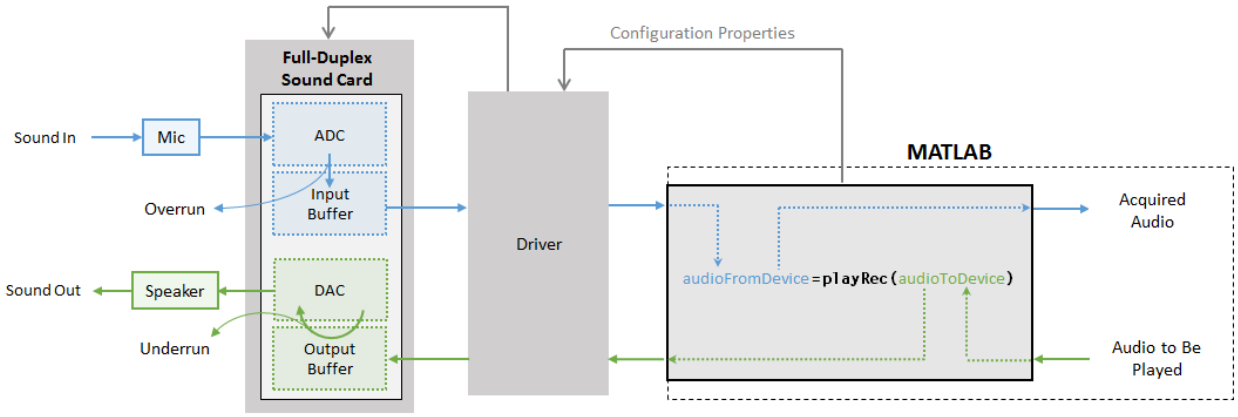
`crossoverFilter`

audioPlayerRecorder

Simultaneously play and record using an audio device

Description

The `audioPlayerRecorder` System object reads and writes audio samples using your computer's audio device. To use `audioPlayerRecorder`, you must have an audio device and driver capable of simultaneous playback and record.



See "Audio I/O: Buffering, Latency, and Throughput" for a detailed explanation of the data flow.

To simultaneously play and record:

1. Create the `audioPlayerRecorder` object and set its properties.
2. Call the object with arguments, as if it were a function.

To learn more about how System objects work, see What Are System Objects?

Creation

Syntax

```
playRec = audioPlayerRecorder
playRec = audioPlayerRecorder(sampleRateValue)
playRec = audioPlayerRecorder(___,Name,Value)
```

Description

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

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

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

Example: `playRec = audioPlayerRecorder(48000,'BitDepth','8-bit integer')` creates a System object, `playRec`, that operates at a 48 kHz sample rate and an 8-bit integer bit depth.

Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see System Design in MATLAB Using System Objects.

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

default audio device (default) | character vector | string

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

Supported drivers for `audioPlayerRecorder` are platform-specific:

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

Note The default audio device is the default device of your machine only if it supports full-duplex mode. If your machine's default audio device does not support full-duplex mode, `audioPlayerRecorder` specifies as the default device the first available device it detects that is capable of full-duplex mode. Use the `info` method to get the device name associated with your `audioPlayerRecorder` object.

Data Types: `char` | `string`

`SampleRate` — Sample rate used by device to record and play audio data (Hz)

`44100` (default) | positive integer

Sample rate used by device to record and play audio data, in Hz, specified as a positive integer. The range of `SampleRate` depends on your audio hardware.

Data Types: `single` | `double`

`BitDepth` — Data type used by device

`'16-bit integer'` (default) | `'8-bit integer'` | `'32-bit float'` | `'24-bit integer'`

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

Data Types: `char` | `string`

`SupportVariableSize` — Support variable frame size
`false` (default) | `true`

Option to support variable frame size, specified as `false` or `true`.

- `false` -- If the `audioPlayerRecorder` object is locked, the input must have the same frame size at each call. The buffer size of your audio device is the same as the input frame size. If you are using the object on Windows, open the ASIO UI to set the sound card buffer to the frame size value.
- `true` -- If the `audioPlayerRecorder` object is locked, the input frame size can change at each call. The buffer size of your audio device is specified through the BufferSize property.

To minimize latency, set `SupportVariableSize` to `false`. If variable-size input is required by your audio system, set `SupportVariableSize` to `true`.

Data Types: `logical`

`BufferSize` — Buffer size of audio device
`1024` (default) | positive integer

Buffer size of audio device, specified as a positive integer.

Note If you are using the object on a Windows machine, use `asiosettings` to set the sound card buffer size to the `BufferSize` value of your `audioPlayerRecorder` System object.

Dependencies

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

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

`PlayerChannelMapping` — Mapping between columns of played data and channels of device
`[]` (default) | scalar | vector

Mapping between columns of played data and channels of output device, specified as a scalar or as a vector of valid channel indices. The default value of this property is `[]`, which means that the default channel mapping is used.

Note To ensure mono output on only one channel of a stereo device, use the default `PlayerChannelMapping` setting and provide a stereo signal where one channel is all zeros.

Example: `outputLeftOnly = [x(:,1) zeros(size(x,1),1)];`

Example: `outputRightOnly = [zeros(size(x,1),1) x(:,1)];`

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

`RecorderChannelMapping` — Mapping between channels of device and columns of recorded data
`1` (default) | scalar | vector

Mapping between channels of your audio device and columns of recorded data, specified as a scalar or as a vector of valid channel indices. The default value is `1`, which means that the first recording channel on the device is used to acquire data and is mapped to a single-column matrix.

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

Usage

Syntax

```
audioFromDevice = playRec(audioToDevice)
[audioFromDevice,numUnderrun] = playRec(audioToDevice)
[audioFromDevice,numUnderrun,numOverrun] = playRec(audioToDevice)
```

Description

`audioFromDevice = playRec(audioToDevice)` writes one frame of audio samples, `audioToDevice`, to the selected audio device, and returns one frame of audio, `audioFromDevice`.

`[audioFromDevice,numUnderrun] = playRec(audioToDevice)` returns the number of samples overrun since the last call to `playRec`.

`[audioFromDevice,numUnderrun,numOverrun] = playRec(audioToDevice)` returns the number of samples underrun since the last call to `playRec`.

Note: When you call the `audioPlayerRecorder` System object, the audio device specified by the `Device` property is locked. An audio device can be locked by only one `audioPlayerRecorder` at a time. To release the audio device, call `release` on the `audioPlayerRecorder` System object.

Input Arguments

`audioToDevice` — Audio to device

matrix

Audio signal to write to device, specified as a matrix. The columns of the matrix are treated as independent audio channels.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8`

Output Arguments

`audioFromDevice` — Audio from device

matrix

Audio signal read from device, returned as a matrix the same size and data type as `audioToDevice`.

Data Types: `single` | `double` | `int16` | `int32` | `uint8`

`numUnderrun` — Number of samples underrun

scalar

Number of samples by which the player queue was underrun since the last call to `playRec`. Underrun refers to output signal silence. Output signal silence occurs if the device buffer is empty when it is time for digital-to-analog conversion. This results when the processing loop in MATLAB does not supply samples at the rate the sound card demands.

Data Types: `uint32`

`numOverrun` — Number of samples overrun
scalar

Number of samples by which the recorder queue was overrun since the last call to `playRec`. Overrun refers to input signal drops. Input signal drops occur when the processing stage does not keep pace with the acquisition of samples.

Data Types: `uint32`

Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

Specific to audioPlayerRecorder

getAudioDevices	List available audio devices
info	Get audio device information

Common to All System Objects

clone	Create duplicate System object
isLocked	Determine if System object is in use
release	Release resources and allow changes to System object property values and input characteristics
reset	Reset internal states of System object
step	Run System object algorithm
setup	One-time set up tasks for System objects

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',512);
fs = fileReader.SampleRate;

fileWriter = dsp.AudioFileWriter('Counting-PlaybackRecorded.wav', ...
    'SampleRate',fs);

aPR = audioPlayerRecorder('SampleRate',fs);
```

In a frame-based loop:

1. Read an audio signal from your file.
2. Play the audio signal to your device and simultaneously record audio from your device. Use the optional `nUnderruns` and `nOverruns` output arguments to track any loss of synchronization.
3. Write your recorded audio to a file.

Once the loop is completed, release the objects to free devices and resources.

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

    [audioRecorded,nUnderruns,nOverruns] = aPR(audioToPlay);

    fileWriter(audioRecorded)

    if nUnderruns > 0
        fprintf('Audio player queue was underrun by %d samples.\n',nUnderruns);
    end
    if nOverruns > 0
        fprintf('Audio recorder queue was overrun by %d samples.\n',nOverruns);
    end
end
```

```
Audio player queue was underrun by 512 samples.
```

```
release(fileReader)
release(fileWriter)
release(aPR)
```

Specify Nondefault Channel Mapping

The `audioPlayerRecorder` System object™ enables you to specify a nondefault mapping between the channels of your audio device and the data sent to and received from your audio device. To run this example, your audio device must have at least two channels and be capable of full-duplex mode.

Using Default Settings

Create an `audioPlayerRecorder` object with default settings. The `audioPlayerRecorder` is automatically configured to a compatible device and driver.

```
aPR = audioPlayerRecorder;
```

The `audioPlayerRecorder` combines reading from your device and writing to your device in a single call: `audioFromDevice = aPR(audioToDevice)`. Calling the `audioPlayerRecorder` with default settings:

- Maps columns of `audioToDevice` to output channels of your device
- Maps input channels of your device to columns of `audioFromDevice`

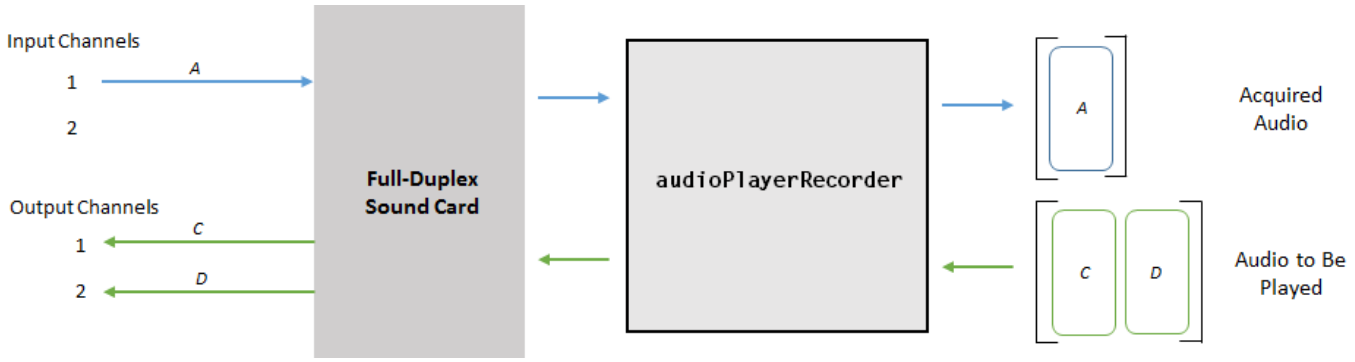
By default, `audioFromDevice` is a one-column matrix corresponding to channel 1 of your audio device. To view the maximum number of input and output channels of your device, use the `info` method.

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

`aPRInfo` is returned as a structure with fields containing information about your selected driver, audio device, and the maximum number of input and output channels in your configuration.

Call the `audioPlayerRecorder` with a two-column matrix. By default, column 1 is mapped to output channel 1, and column 2 is mapped to output channel 2. The `audioPlayerRecorder` returns a one-column matrix with the same number of rows as the `audioToDevice` matrix.

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

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

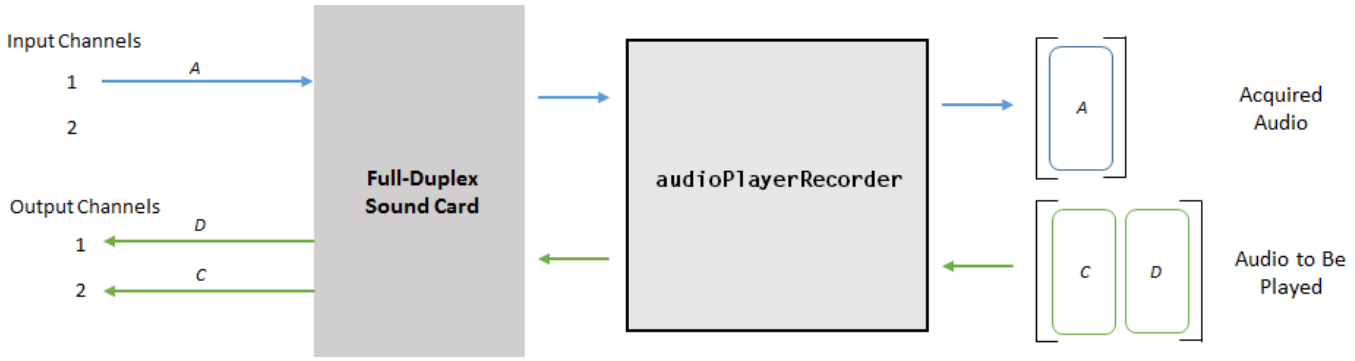


Nondefault Channel Mapping for Audio Output

Specify a nondefault channel mapping for your audio output. Specify that column 1 of `audioToDevice` maps to channel 2, and that column 2 of `audioToDevice` maps to channel 1. To modify the channel mapping, the `audioPlayerRecorder` object must be unlocked.

Run the `audioPlayerRecorder` object. If you are using headphones or stereo speakers, notice that the high frequency and low frequency tones have switched speakers.

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

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

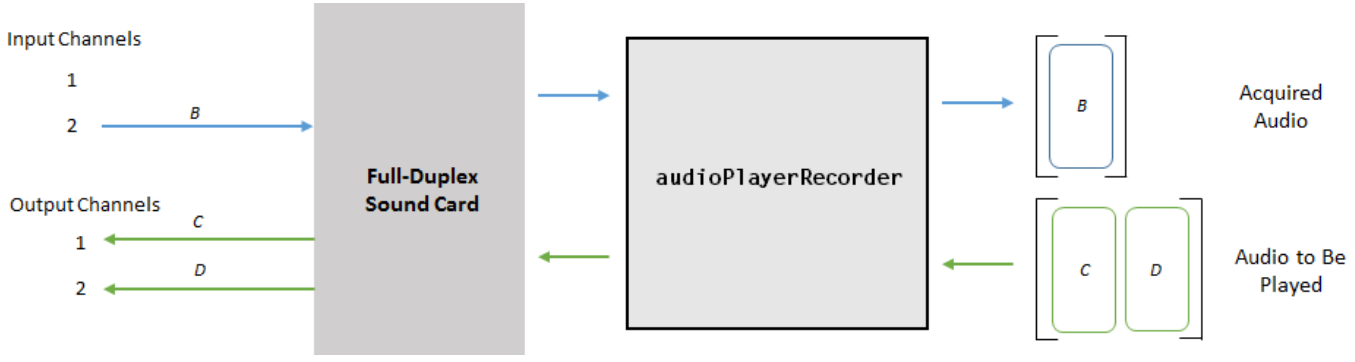


Nondefault Channel Mapping for Audio Input

Specify a nondefault channel mapping for your audio input. Record data from only channel two of your device. In this case, channel 2 is mapped to a one-column matrix. Use `size` to verify that `audioFromDevice` is a 256-by-1 matrix.

```
release(aPR)
aPR.RecorderChannelMapping = 2;

audioFromDevice = aPR(audioToDevice);

[rows,col] = size(audioFromDevice)
```

```
rows =

   256


col =

     1
```



As a best practice, release your audio device once complete.

```
release(aPR)
```

Version History

Introduced in R2017a

Extended Capabilities

C/C++ Code Generation

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

Usage notes and limitations:

- “System Objects in MATLAB Code Generation” (MATLAB Coder)
- The executable generated from this System object relies on prebuilt dynamic library files (`.dll` files) included with MATLAB. Use the `packNGo` function to package the code generated from this object and all the relevant files in a compressed zip file. Using this zip file, you can relocate, unpack, and rebuild your project in another development environment where MATLAB is not installed. For more details, see “Run Audio I/O Features Outside MATLAB and Simulink”.

See Also

Functions

`asiosettings` | `getAudioDevices` | `audioDeviceWriter` | `audioDeviceReader` | `dsp.AudioFileReader`

Blocks

Audio Device Reader | Audio Device Writer

Topics

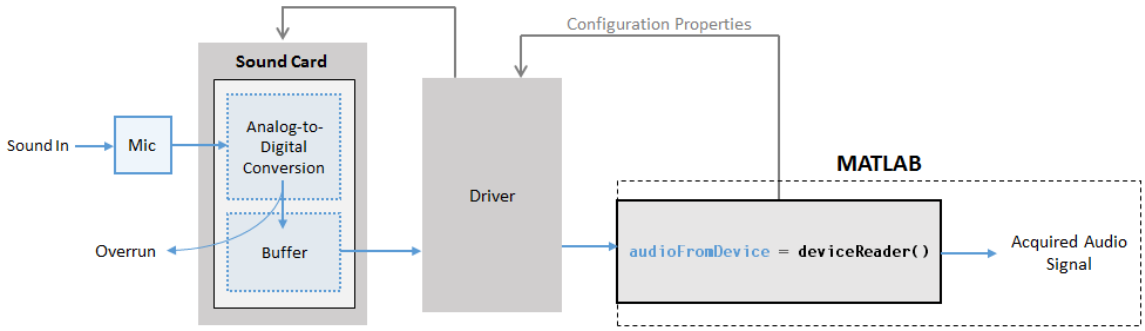
“Audio I/O: Buffering, Latency, and Throughput”
“Run Audio I/O Features Outside MATLAB and Simulink”
“Real-Time Audio in MATLAB”

audioDeviceReader

Record from sound card

Description

The `audioDeviceReader` System object reads audio samples using your computer's audio input device.



See "Audio I/O: Buffering, Latency, and Throughput" for a detailed explanation of the audio device reader data flow.

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

To stream data from an audio device:

1. Create the `audioDeviceReader` object and set its properties.
2. Call the object with arguments, as if it were a function.

To learn more about how System objects work, see What Are System Objects?

Creation

Syntax

```
deviceReader = audioDeviceReader
deviceReader = audioDeviceReader(sampleRateValue)
deviceReader = audioDeviceReader(sampleRateValue,sampPerFrameValue)
deviceReader = audioDeviceReader( ___ ,Name,Value)
```

Description

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

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

`deviceReader = audioDeviceReader(sampleRateValue,sampPerFrameValue)` sets the SamplesPerFrame property to `sampPerFrameValue`.

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

Example: `deviceReader = audioDeviceReader(16000,'BitDepth','8-bit integer')` creates a System object, `deviceReader`, that operates at a 16 kHz sample rate and an 8-bit integer bit depth.

Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see System Design in MATLAB Using System Objects.

`Driver` — Driver used to access audio device (Windows only)

`'DirectSound'` (default) | `'ASIO'` | `'WASAPI'`

Driver used to access your audio device, specified as `'DirectSound'`, `'ASIO'`, or `'WASAPI'`.

- ASIO drivers do not come pre-installed on Windows machines. To use the `'ASIO'` driver option, install an ASIO driver outside of MATLAB.

  Note If `Driver` is specified as `'ASIO'`, use `asiosettings` to set the sound card buffer size to the `SamplesPerFrame` value of your `audioDeviceReader` System object.

- WASAPI drivers are supported for exclusive-mode only.

ASIO and WASAPI drivers do not provide sample rate conversion. For ASIO and WASAPI drivers, set `SampleRate` to a sample rate supported by your audio device.

This property applies only on Windows machines. Linux machines always use the ALSA driver. Mac machines always use the CoreAudio driver.

Data Types: `char` | `string`

`Device` — Device used to acquire audio samples

default audio device (default) | character vector | string

Device used to acquire audio samples, specified as a character vector or string. Use `getAudioDevices` to list available devices for the selected driver.

Data Types: `char` | `string`

`NumChannels` — Number of input channels acquired by audio device

`1` (default) | integer

Number of input channels acquired by audio device, specified as an integer. The range of `NumChannels` depends on your audio hardware.

Dependencies

To enable this property, set ChannelMappingSource to `'Auto'`.

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

`SamplesPerFrame` — Frame size read from audio device
`1024` (default) | integer

Frame size read from audio device, specified as a positive integer. `SamplesPerFrame` is also the size of your device buffer and the number of columns of the output matrix returned by your `audioDeviceReader` object.

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

`SampleRate` — Sample rate used by device to acquire audio data (Hz)
`44100` (default) | positive integer

Sample rate used by device to acquire audio data, in Hz, specified as a positive integer. The range of `SampleRate` depends on your audio hardware.

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

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

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

Data Types: `char` | `string`

`ChannelMappingSource` — Source of mapping between device channels and output matrix
`'Auto'` (default) | `'Property'`

Source of mapping between the channels of your audio input device and columns of the output matrix, specified as `'Auto'` or `'Property'`.

- `'Auto'` –– The default settings determine the mapping between device channels and output matrix. For example, suppose that your audio device has six channels available, and you set `NumChannels` to 6. The output from a call to your audio device reader is a six-column matrix. Column 1 corresponds to channel 1, column 2 corresponds to channel 2, and so on.
- `'Property'` –– The ChannelMapping property determines the mapping between channels of your audio device and columns of the output matrix.

Data Types: `char` | `string`

`ChannelMapping` — Nondefault mapping between device channels and output matrix
`[1:MaximumInputChannels]` (default) | scalar | vector

Nondefault mapping between channels of your audio input device and columns of the output matrix, specified as a vector of valid channel indices. See “Specify Channel Mapping for audioDeviceReader” on page 3-148 for more information.

Dependencies

To enable this property, set ChannelMappingSource to `'Property'`.

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

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

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

Note If `OutputDataType` is specified as `'double'` or `'single'`, the audio device reader outputs data in the range [–1, 1]. For other data types, the range is [min, max] of the specified data type.

Data Types: `char` | `string`

Usage

Syntax

```
audioFromDevice = deviceReader()
[audioFromDevice,numOverrun] = deviceReader()
```

Description

`audioFromDevice = deviceReader()` returns one frame of audio samples from the selected audio input device.

`[audioFromDevice,numOverrun] = deviceReader()` returns the number of samples by which the audio reader's queue was overrun since the last call to `deviceReader`.

Note: When you call the `audioDeviceReader` System object, the audio device specified by the Device property is locked. An audio device can be locked by only one `audioDeviceReader` at a time. To release the audio device, call `release` on your `audioDeviceReader` object.

Output Arguments

`audioFromDevice` — Audio from device
matrix

Audio signal read from device, returned as a matrix. The specified number of channels and the SamplesPerFrame property determine the matrix size. The data type of the matrix depends on the OutputDataType property.

Data Types: `single` | `double` | `int16` | `int32` | `uint8`

`numOverrun` — Number of samples overrun
scalar

Number of samples by which the audio reader's queue was overrun since the last call to `deviceReader`.

Data Types: `uint32`

Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

Specific to audioDeviceReader

getAudioDevices	List available audio devices
info	Get audio device information

Common to All System Objects

clone	Create duplicate System object
isLocked	Determine if System object is in use
release	Release resources and allow changes to System object property values and input characteristics
reset	Reset internal states of System object
step	Run System object algorithm
setup	One-time set up tasks for System objects

Examples

Read from Microphone and Write to Audio File

Record 10 seconds of speech with a microphone and send the output to a WAV file.

Create an `audioDeviceReader` 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.')
```

```
Speak into microphone now.
```

```
tic
while toc < 10
    acquiredAudio = deviceReader();
    fileWriter(acquiredAudio);
end
disp('Recording complete.')
```

```
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` object to reduce latency.

Create an `audioDeviceReader` object with default settings.

```
deviceReader = audioDeviceReader
```

```
deviceReader =
  audioDeviceReader with properties:

            Driver: 'DirectSound'
            Device: 'Default'
       NumChannels: 1
    SamplesPerFrame: 1024
         SampleRate: 44100

  Show all properties
```

Calculate the latency due to your device buffer.

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

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

Set the `SamplesPerFrame` property of your `audioDeviceReader` object to `64`. Calculate the latency.

```
deviceReader.SamplesPerFrame = 64;
fprintf('Latency due to device buffer: %f seconds.\n',deviceReader.SamplesPerFrame/deviceReader.S
```

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

Set the `SampleRate` property of your `audioDeviceReader` System object to `96000`. Calculate the latency.

```
deviceReader.SampleRate = 96000;
fprintf('Latency due to device buffer: %f seconds.\n',deviceReader.SamplesPerFrame/deviceReader.S
```

```
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` object to decrease overrun. Your results depend on your computer.

Create an `audioDeviceReader` System object with `SamplesPerFrame` set to `256` and `SampleRate` set to `44100`. 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` object. Specify the file name and data type to write.

```
fileWriter = dsp.AudioFileWriter('mySpeech.wav','FileFormat','WAV');
```

Record 5 seconds of speech. In an audio stream loop, read an audio signal frame from your device, and write the audio signal frame to a specified file.

```
totalOverrun = 0;
disp('Speak into microphone now.')
```

```
Speak into microphone now.
```

```
tic
while toc < 5
    [input,numOverrun] = deviceReader();
    totalOverrun = totalOverrun + numOverrun;
    fileWriter(input);
end
fprintf('Recording complete.\n')
```

```
Recording complete.
```

```
fprintf('Total number of samples overrun: %d.\n',totalOverrun)
```

```
Total number of samples overrun: 0.
```

```
fprintf('Total seconds overrun: %d.\n',double(totalOverrun)/double(deviceReader.SampleRate))
```

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

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

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

Use `pause` to add an artificial computational load to your audio stream loop. The computational load causes the audio stream loop to go slower than the device, which causes acquired samples to be dropped.

```
disp('Speak into microphone now.')
```

```
Speak into microphone now.
```

```
tic
while toc < 5
    [input,numOverrun] = deviceReader();
    totalOverrun = totalOverrun + numOverrun;
    fileWriter(input);
    pause(0.01)
```

```
end
fprintf('Recording complete.\n')
```

```
Recording complete.
```

```
fprintf('Total number of samples overrun: %d.\n',totalOverrun)
```

```
Total number of samples overrun: 97536.
```

```
fprintf('Total seconds overrun: %d.\n',double(totalOverrun)/double(deviceReader.SampleRate))
```

```
Total seconds overrun: 2.211701e+00.
```

Release your `audioDeviceReader` and `dsp.AudioFileWriter` 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.')
```

```
Speak into microphone now.
```

```
tic
while toc < 5
    [input,numOverrun] = deviceReader();
    totalOverrun = totalOverrun + numOverrun;
    fileWriter(input);
    pause(0.01)
end
fprintf('Recording complete.\n')
```

```
Recording complete.
```

```
fprintf('Total number of samples overrun: %d.\n',totalOverrun)
```

```
Total number of samples overrun: 0.
```

```
fprintf('Total seconds overrun: %f.\n',totalOverrun/deviceReader.SampleRate)
```

```
Total seconds overrun: 0.000000.
```

Specify Channel Mapping for `audioDeviceReader`

Specify nondefault channel mapping for an `audioDeviceReader` object. This example is hardware specific. It assumes that your computer has a default audio input device with two available channels.

Create an `audioDeviceReader` object with default settings.

```
deviceReader = audioDeviceReader;
```

The default number of channels is 1. Call your `audioDeviceReader` 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` object must be unlocked to change this property.

```
release(deviceReader)
deviceReader.ChannelMappingSource = 'Property'
```

```
deviceReader =
  audioDeviceReader with properties:

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

  Show all properties
```

By default, if `ChannelMappingSource` is set to `'Property'`, all available channels are mapped to the output. Call your `audioDeviceReader` 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` object.

```
release(deviceReader)
deviceReader.ChannelMapping = [2,1];
```

Now when you call your `audioDeviceReader`:

- Input channel 1 of your device maps to the second column of your output matrix.
- Input channel 2 of your device maps to the first column of your output matrix.

Acquire a specific channel from your input device.

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

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

Version History

Introduced in R2016a

Extended Capabilities

C/C++ Code Generation

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

Usage notes and limitations:

- “System Objects in MATLAB Code Generation” (MATLAB Coder)
- The executable generated from this System object relies on prebuilt dynamic library files (`.dll` files) included with MATLAB. Use the `packNGo` function to package the code generated from this object and all the relevant files in a compressed zip file. Using this zip file, you can relocate, unpack, and rebuild your project in another development environment where MATLAB is not installed. For more details, see “Run Audio I/O Features Outside MATLAB and Simulink”.

See Also

Functions

`asiosettings` | `getAudioDevices` | `audioDeviceWriter` | `audioPlayerRecorder` | `dsp.AudioFileReader`

Blocks

Audio Device Reader

Topics

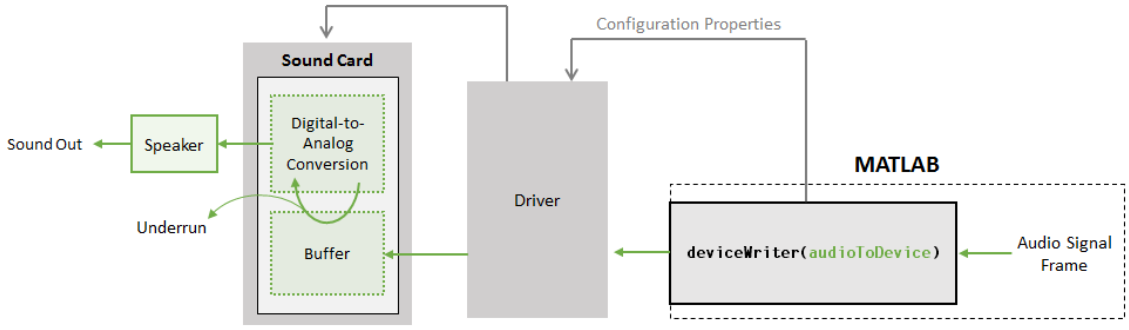
“Audio I/O: Buffering, Latency, and Throughput”
“Run Audio I/O Features Outside MATLAB and Simulink”
“Real-Time Audio in MATLAB”

audioDeviceWriter

Play to sound card

Description

The `audioDeviceWriter` System object writes audio samples to an audio output device. Properties of the audio device writer specify the driver, the device, and device attributes such as sample rate, bit depth, and buffer size.



See “Audio I/O: Buffering, Latency, and Throughput” for a detailed explanation of the audio device writer data flow.

To stream data to an audio device:

1. Create the `audioDeviceWriter` object and set its properties.
2. Call the object with arguments, as if it were a function.

To learn more about how System objects work, see What Are System Objects?

Creation

Syntax

```
deviceWriter = audioDeviceWriter
deviceWriter = audioDeviceWriter(sampleRateValue)
deviceWriter = audioDeviceWriter(___,Name,Value)
```

Description

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

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

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

Example: `deviceWriter = audioDeviceWriter(48000,'BitDepth','8-bit integer')` creates a System object, `deviceWriter`, that operates at a 48 kHz sample rate and an 8-bit integer bit depth.

Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see System Design in MATLAB Using System Objects.

`Driver` — Driver used to access audio device (Windows only)

`'DirectSound'` (default) | `'ASIO'` | `'WASAPI'`

Driver used to access your audio device, specified as `'DirectSound'`, `'ASIO'`, or `'WASAPI'`.

- ASIO drivers do not come pre-installed on Windows machines. To use the `'ASIO'` driver option, install an ASIO driver outside of MATLAB.

  Note If `Driver` is specified as `'ASIO'`, use `asiosettings` to set the sound card buffer size to the buffer size of your `audioDeviceWriter` System object.

- WASAPI drivers are supported for exclusive-mode only.

ASIO and WASAPI drivers do not provide sample rate conversion. For ASIO and WASAPI drivers, set `SampleRate` to a sample rate supported by your audio device.

This property applies only on Windows machines. Linux machines always use the ALSA driver. Mac machines always use the CoreAudio driver.

To specify nondefault `Driver` values, you must have an Audio Toolbox license. If the toolbox is not installed, specifying nondefault `Driver` values returns an error.

Data Types: `char` | `string`

`Device` — Device used to play audio samples

default audio device (default) | character vector | string scalar

Device used to play audio samples, specified as a character vector or string scalar. Use `getAudioDevices` to list available devices for the selected driver.

Data Types: `char` | `string`

`SampleRate` — Sample rate of signal sent to audio device (Hz)

`44100` (default) | positive integer

Sample rate of signal sent to audio device, in Hz, specified as a positive integer. The range of `SampleRate` depends on your audio hardware.

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

`BitDepth` — Data type used by the device

`'16-bit integer'` (default) | `'8-bit integer'` | `'24-bit integer'` | `'32-bit float'`

Data type used by the device, specified as a character vector or string scalar. Before performing digital-to-analog conversion, the input data is cast to a data type specified by `BitDepth`.

To specify a nondefault `BitDepth`, you must have an Audio Toolbox license. If the toolbox is not installed, specifying a nondefault `BitDepth` returns an error.

Data Types: `char` | `string`

`SupportVariableSizeInput` — Support variable frame size

`false` (default) | `true`

Option to support variable frame size, specified as `true` or `false`.

- `false` –– If the `audioDeviceWriter` object is locked, the input must have the same frame size at each call. The buffer size of your audio device is the same as the input frame size.
- `true` –– If the `audioDeviceWriter` object is locked, the input frame size can change at each call. The buffer size of your audio device is specified through the BufferSize property.

Data Types: `char`

`BufferSize` — Buffer size of audio device

`4096` (default) | positive integer

Buffer size of audio device, specified as a positive integer.

Note If `Driver` is specified as `'ASIO'`, open the ASIO UI to set the sound card buffer size to the `BufferSize` value of your `audioDeviceWriter` System object.

Dependencies

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

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

`ChannelMappingSource` — Source of mapping between input matrix and device channels

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

Source of mapping between columns of input matrix and channels of audio output device, specified as `'Auto'` or `'Property'`.

- `'Auto'` –– Default settings determine the mapping between columns of input matrix and channels of audio output device. For example, suppose that your input is a matrix with four columns, and your audio device has four channels available. Column 1 of your input data writes to channel 1 of your device, column 2 of your input data writes to channel 2 of your device, and so on.
- `'Property'` –– The ChannelMapping property determines the mapping between columns of input matrix and channels of audio output device.

Data Types: `char` | `string`

`ChannelMapping` — Nondefault mapping between input matrix and device channels

`[1:MaximumOutputChannels]` (default) | scalar | vector

Nondefault mapping between columns of input matrix and channels of output device, specified as a scalar or vector of valid channel indices. See the “Specify Channel Mapping for audioDeviceWriter” on page 3-159 example for more information.

To selectively map between columns of the input matrix and your sound card's output channels, you must have an Audio Toolbox license. If the toolbox is not installed, specifying a nondefault `ChannelMapping` returns an error.

Note To ensure mono output on only one channel of a stereo device, use the default `ChannelMapping` setting and provide a stereo signal where one channel is all zeros.

Example: `outputLeftOnly = [x(:,1) zeros(size(x,1),1)];`

Example: `outputRightOnly = [zeros(size(x,1),1) x(:,1)];`

Dependencies

To enable this property, set ChannelMappingSource to `'Property'`.

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

Usage

Syntax

```
numUnderrun = deviceWriter(audioToDevice)
```

Description

`numUnderrun = deviceWriter(audioToDevice)` writes one frame of audio samples, `audioToDevice`, to the selected audio device and returns the number of audio samples underrun since the last call to `deviceWriter`.

Note: When you call the `audioDeviceWriter` System object, the audio device specified by the `Device` property is locked. An audio device can be locked by only one `audioDeviceWriter` at a time. To release the audio device, call `release` on your `audioDeviceWriter` System object.

Input Arguments

`audioToDevice` — Audio to device
matrix

Audio signal to write to device, specified as a matrix. The columns of the matrix are treated as independent audio channels.

If `audioToDevice` is of data type `'double'` or `'single'`, the audio device writer clips values outside the range [–1, 1]. For other data types, the allowed input range is [min, max] of the specified data type.

Data Types: `single` | `double` | `int16` | `int32` | `uint8`

Output Arguments

`numUnderrun` — Number of samples underrun
scalar

Number of samples by which the audio device writer queue was underrun since the last call to `deviceWriter`.

Data Types: `uint32`

Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

Specific to audioDeviceWriter

getAudioDevices	List available audio devices
info	Get audio device information

Common to All System Objects

clone	Create duplicate System object
isLocked	Determine if System object is in use
release	Release resources and allow changes to System object property values and input characteristics
reset	Reset internal states of System object
step	Run System object algorithm
setup	One-time set up tasks for System objects

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` 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')
```

```
fileInfo = struct with fields:
             Filename: 'B:\matlab\toolbox\dsp\samples\speech_dft.mp3'
    CompressionMethod: 'MP3'
          NumChannels: 1
           SampleRate: 22050
         TotalSamples: 112893
             Duration: 5.1199
                Title: []
              Comment: []
               Artist: []
```

```
               BitRate: 64
```

Create an `audioDeviceWriter` object and specify the sample rate.

```
deviceWriter = audioDeviceWriter('SampleRate',fileInfo.SampleRate);
```

Call `setup` to reduce the computational load of initialization in an audio stream loop.

```
setup(deviceWriter,zeros(fileReader.SamplesPerFrame,...
    fileInfo.NumChannels))
```

Use the `info` function to obtain the characteristic information about the device writer.

```
info(deviceWriter)
```

```
ans = struct with fields:
                    Driver: 'DirectSound'
                DeviceName: 'Primary Sound Driver'
    MaximumOutputChannels: 2
```

In an audio stream loop, read an audio signal frame from the file, and write the frame to your device.

```
while ~isDone(fileReader)
    audioData = fileReader();
    deviceWriter(audioData);
end
```

Close the input file and release the device.

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

Reduce Latency due to Output Device Buffer

Latency due to the output device buffer is the time delay of writing one frame of data. Modify default properties of your `audioDeviceWriter` System object™ to reduce latency due to device buffer size.

Create a `dsp.AudioFileReader` System object to read an audio file with default settings.

```
fileReader = dsp.AudioFileReader('speech_dft.mp3');
```

Create an `audioDeviceWriter` System object and specify the sample rate to match that of the audio file reader.

```
deviceWriter = audioDeviceWriter(...
    'SampleRate',fileReader.SampleRate);
```

Calculate the latency due to your device buffer, in seconds.

```
bufferLatency = fileReader.SamplesPerFrame/deviceWriter.SampleRate %#ok
```

```
bufferLatency = 0.0464
```

Set the `SamplesPerFrame` property of your `dsp.AudioFileReader` System object to 256. Calculate the buffer latency in seconds.

```
fileReader.SamplesPerFrame = 256;
bufferLatency = fileReader.SamplesPerFrame/deviceWriter.SampleRate
```

```
bufferLatency = 0.0116
```

Determine and Decrease Underrun

Underrun refers to output signal silence, which occurs when the audio stream loop does not keep pace with the output device. Determine the underrun of an audio stream loop, add artificial computational load to the audio stream loop, and then modify properties of your `audioDeviceWriter` object to decrease underrun. Your results depend on your computer.

Create a `dsp.AudioFileReader` 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` 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)
```

```
Total samples underrun: 0.
```

```
fprintf('Total seconds underrun: %d.\n',double(totalUnderrun)/double(deviceWriter.SampleRate))
```

```
Total seconds underrun: 0.
```

Release your `dsp.AudioFileReader` and `audioDeviceWriter` objects and set your counter variable to zero.

```
release(fileReader)
release(deviceWriter)
totalUnderrun = 0;
```

Use `pause` to mimic an algorithm that takes 0.075 seconds to process. The pause causes the audio stream loop to go slower than the device, which results in periods of silence in the output audio signal.

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

```
    numUnderrun = deviceWriter(input);
    totalUnderrun = totalUnderrun + numUnderrun;
    pause(0.075)
end
fprintf('Total samples underrun: %d.\n',totalUnderrun)

Total samples underrun: 68608.

fprintf('Total seconds underrun: %d.\n',double(totalUnderrun)/double(deviceWriter.SampleRate))

Total seconds underrun: 3.111474e+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)

Total samples underrun: 0.

fprintf('Total seconds underrun: %d.\n',double(totalUnderrun)/double(deviceWriter.SampleRate))

Total seconds underrun: 0.
```

The increased frame size reduces the total underrun of your audio stream loop. However, increasing the frame size also increases latency. Other approaches to reduce underrun include:

- Increasing the buffer size independent of input frame size. To increase buffer size independent of input frame size, you must first set `SupportVariableSizeInput` to `true`. This approach also increases latency.
- Decreasing the sample rate. Decreasing the sample rate reduces both latency and underrun at the cost of signal resolution.
- Choosing an optimal driver and device for your system.

Specify Channel Mapping for `audioDeviceWriter`

Specify nondefault channel mapping for an `audioDeviceWriter` object. This example is hardware specific. It assumes that your computer has a default audio output device with two available channels.

Create an `audioDeviceWriter` object with default settings.

```
deviceWriter = audioDeviceWriter;
```

By default, the `audioDeviceWriter` object writes the maximum number of channels available, corresponding to the columns of the input matrix. Use `info` to get the maximum number of channels of your device.

```
info(deviceWriter)
```

```
ans = struct with fields:
                   Driver: 'DirectSound'
               DeviceName: 'Primary Sound Driver'
    MaximumOutputChannels: 2
```

If `deviceWriter` is called with one column of data, two channels are written to your audio output device. Both channels correspond to the one column of data.

Use the `audioOscillator` object to output a tone to your `audioDeviceWriter` 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` object is called with two columns of data, two channels are written to your audio output device. The first column corresponds to channel 1 of your audio output device, and the second column corresponds to channel 2 of your audio output device.

Write a two-column matrix to your audio output device. Column 1 corresponds to the sine tone, and column 2 corresponds to a static signal. If you are using headphones, you can hear the tone from one speaker and the static from the other speaker.

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

Specify alternative mappings between channels of your device and columns of the output matrix by indicating the output channel number at an index corresponding to the input column. Set `ChannelMappingSource` to `'Property'`. Indicate that the first column of your input data writes to channel 2 of your output device, and that the second column of your input data writes to channel 1 of your output device. To modify the channel mapping, you must first unlock the `audioDeviceReader` 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
```

Version History

Introduced in R2016a

Extended Capabilities

C/C++ Code Generation

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

Usage notes and limitations:

- “System Objects in MATLAB Code Generation” (MATLAB Coder)
- The executable generated from this System object relies on prebuilt dynamic library files (`.dll` files) included with MATLAB. Use the `packNGo` function to package the code generated from this object and all the relevant files in a compressed zip file. Using this zip file, you can relocate, unpack, and rebuild your project in another development environment where MATLAB is not installed. For more details, see “Run Audio I/O Features Outside MATLAB and Simulink”.

See Also

`asiosettings` | `getAudioDevices` | Audio Device Writer | `audioDeviceReader` | `audioPlayerRecorder` | `dsp.AudioFileWriter` | `dsp.AudioFileReader`

Topics

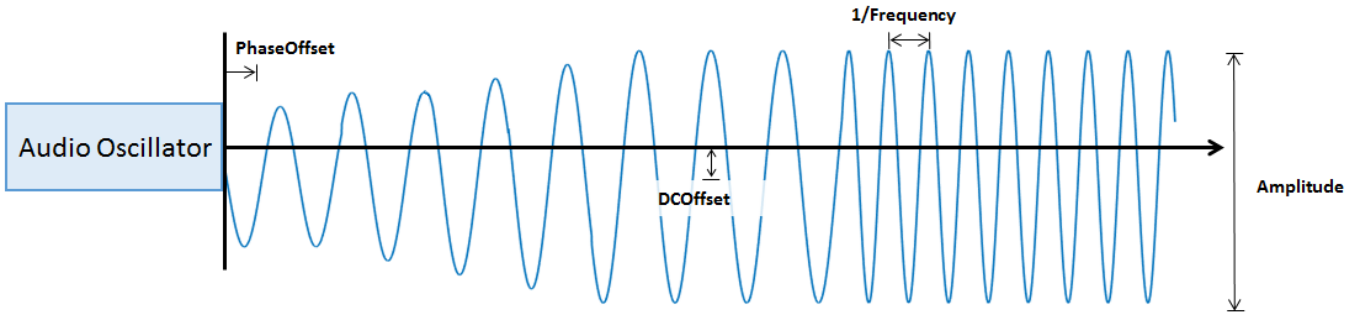
“Run Audio I/O Features Outside MATLAB and Simulink”
“Audio I/O: Buffering, Latency, and Throughput”
“Measure Audio Latency”
“Real-Time Audio in MATLAB”

audioOscillator

Generate sine, square, and sawtooth waveforms

Description

The `audioOscillator` System object generates tunable waveforms. Typical uses include the generation of test signals for test benches, and the generation of control signals for audio effects. Properties of the `audioOscillator` System object specify the type of waveform generated.



To generate tunable waveforms:

1. Create the `audioOscillator` object and set its properties.
2. Call the object with arguments, as if it were a function.

To learn more about how System objects work, see What Are System Objects?

Creation

Syntax

```
osc = audioOscillator
osc = audioOscillator(signalTypeValue)
osc = audioOscillator(signalTypeValue,frequencyValue)
osc = audioOscillator( ___ ,Name,Value)
```

Description

`osc = audioOscillator` creates an audio oscillator System object, `osc`, with default property values.

`osc = audioOscillator(signalTypeValue)` sets the SignalType property to `signalTypeValue`.

`osc = audioOscillator(signalTypeValue,frequencyValue)` sets the Frequency property to `frequencyValue`.

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

Example: `osc = audioOscillator('SignalType','sine','Frequency',8000,'DCOffset',1)` creates a System object, `osc`, which generates 8 kHz sinusoids with a DC offset of one.

Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see System Design in MATLAB Using System Objects.

`SignalType` — Type of generated waveform

`'sine'` (default) | `'square'` | `'sawtooth'`

Type of waveform generated by your `audioOscillator` object, specified as `'sine'`, `'square'`, or `'sawtooth'`.

The waveforms are generated using the algorithms specified by the `sin`, `square`, and `sawtooth` functions.

Tunable: No

Data Types: `char` | `string`

`Frequency` — Frequency of generated waveform (Hz)

`100` (default) | real scalar | vector of real scalars

Frequency of generated waveform in Hz, specified as a real scalar or vector of real scalars greater than or equal to 0.

- For sine waveforms, specify `Frequency` as a scalar or as a vector of length NumTones.
- For square waveforms, specify `Frequency` as a scalar.
- For sawtooth waveforms, specify `Frequency` as a scalar.

Tunable: Yes

Data Types: `single` | `double`

`Amplitude` — Amplitude of generated waveform

`1` (default) | real scalar | vector of real scalars

Amplitude of generated waveform, specified as a real scalar or vector of real scalars greater than or equal to 0.

- For sine waveforms, specify `Amplitude` as a vector of length NumTones.
- For square waveforms, specify `Amplitude` as a scalar.
- For sawtooth waveforms, specify `Amplitude` as a scalar.

The generated waveform is multiplied by the value specified by `Amplitude` at the output, before DCOffset is applied.

Tunable: Yes

Data Types: `single` | `double`

`PhaseOffset` — Normalized phase offset of generated waveform

`0` (default) | real scalar | vector of real scalars

Normalized phase offset of generated waveform, specified as a real scalar or vector of real scalars with values in the range [0, 1]. The range is a normalized 2π-radian interval.

- For sine waveforms, specify `PhaseOffset` as a vector of length NumTones.
- For square waveforms, specify `PhaseOffset` as a scalar.
- For sawtooth waveforms, specify `PhaseOffset` as a scalar.

Tunable: No

Data Types: `single` | `double`

`DCOffset` — Value added to each element of generated waveform

`0` (default) | real scalar | vector of real scalars

Value added to each element of generated waveform, specified as a real scalar or vector of real scalars.

- For sine waveforms, specify `DCOffset` as a vector of length NumTones.
- For square waveforms, specify `DCOffset` as a scalar.
- For sawtooth waveforms, specify `DCOffset` as a scalar.

Tunable: Yes

Data Types: `single` | `double`

`NumTones` — Number of pure sine waveform tones

`1` (default) | positive integer

Number of pure sine waveform tones summed and then generated by the audio oscillator.

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

Tunable: No

Dependencies

To enable this property, set SignalType to `'sine'`.

Data Types: `single` | `double`

`DutyCycle` — Square waveform duty cycle

`0.5` (default) | scalar in the range [0, 1]

Square waveform duty cycle, specified as a scalar in the range [0, 1].

Square waveform duty cycle is the percentage of one period in which the waveform is above the median amplitude. A `DutyCycle` of 1 or 0 is equivalent to a DC offset.

Tunable: Yes

Dependencies

To enable this property, set SignalType to `'square'`.

Data Types: `single` | `double`

`Width` — Sawtooth width

1 (default) | scalar in the range [0, 1]

Sawtooth width, specified as a scalar in the range [0, 1].

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

Tunable: Yes

Dependencies

To enable this property, set SignalType to `'sawtooth'`.

Data Types: `single` | `double`

`SamplesPerFrame` — Number of samples per frame

512 (default) | positive integer

Number of samples per frame, specified as a positive integer in the range [1, `MaxSamplesPerFrame`].

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

Tunable: Yes

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

`MaxSamplesPerFrame` — Maximum number of samples per frame

192000 (default) | positive integer

Maximum number of samples per frame, specified as a positive integer. Setting this property to a lower value can save memory when using code generation.

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

`SampleRate` — Sample rate of generated waveform (Hz)

44100 (default) | positive scalar

Sample rate of generated waveform in Hz, specified as a positive scalar greater than twice the value specified by Frequency.

Tunable: Yes

Data Types: `single` | `double`

`OutputDataType` — Data type of generated waveform

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

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

Tunable: Yes

Data Types: `char` | `string`

Usage

Syntax

```
waveform = osc()
```

Description

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

Output Arguments

`waveform` — Waveform output from oscillator
column vector

Waveform output from the audio oscillator, returned as a column vector with length specified by the `SamplesPerFrame` property.

Data Types: `single` | `double`

Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

Specific to audioOscillator

createAudioPluginClass	Create audio plugin class that implements functionality of System object
parameterTuner	Tune object parameters while streaming

MIDI

configureMIDI	Configure MIDI connections between audio object and MIDI controller
disconnectMIDI	Disconnect MIDI controls from audio object
getMIDIConnections	Get MIDI connections of audio object

Common to All System Objects

clone	Create duplicate System object
isLocked	Determine if System object is in use
release	Release resources and allow changes to System object property values and input characteristics
reset	Reset internal states of System object
step	Run System object algorithm

The `createAudioPluginClass` and `configureMIDI` functions map tunable properties of the `audioOscillator` System object to user-facing parameters:

Property	Range	Mapping	Units
`Frequency`	[0.1, 20000]	log	Hz
`Amplitude`	[0, 10]	linear	no units
`DCOffset`	[–10, 10]	linear	no units
`DutyCycle` (available when you set `SignalType` to `'square'`)	[0, 1]	linear	no units
`Width` (available when you set `SignalType` to `'sawtooth'`)	[0, 1]	linear	no units

Examples

Generate Variable-Frequency Sine Wave

Use the `audioOscillator` to generate a variable-frequency sine wave.

Create an audio oscillator to generate a sine wave. Use the default settings.

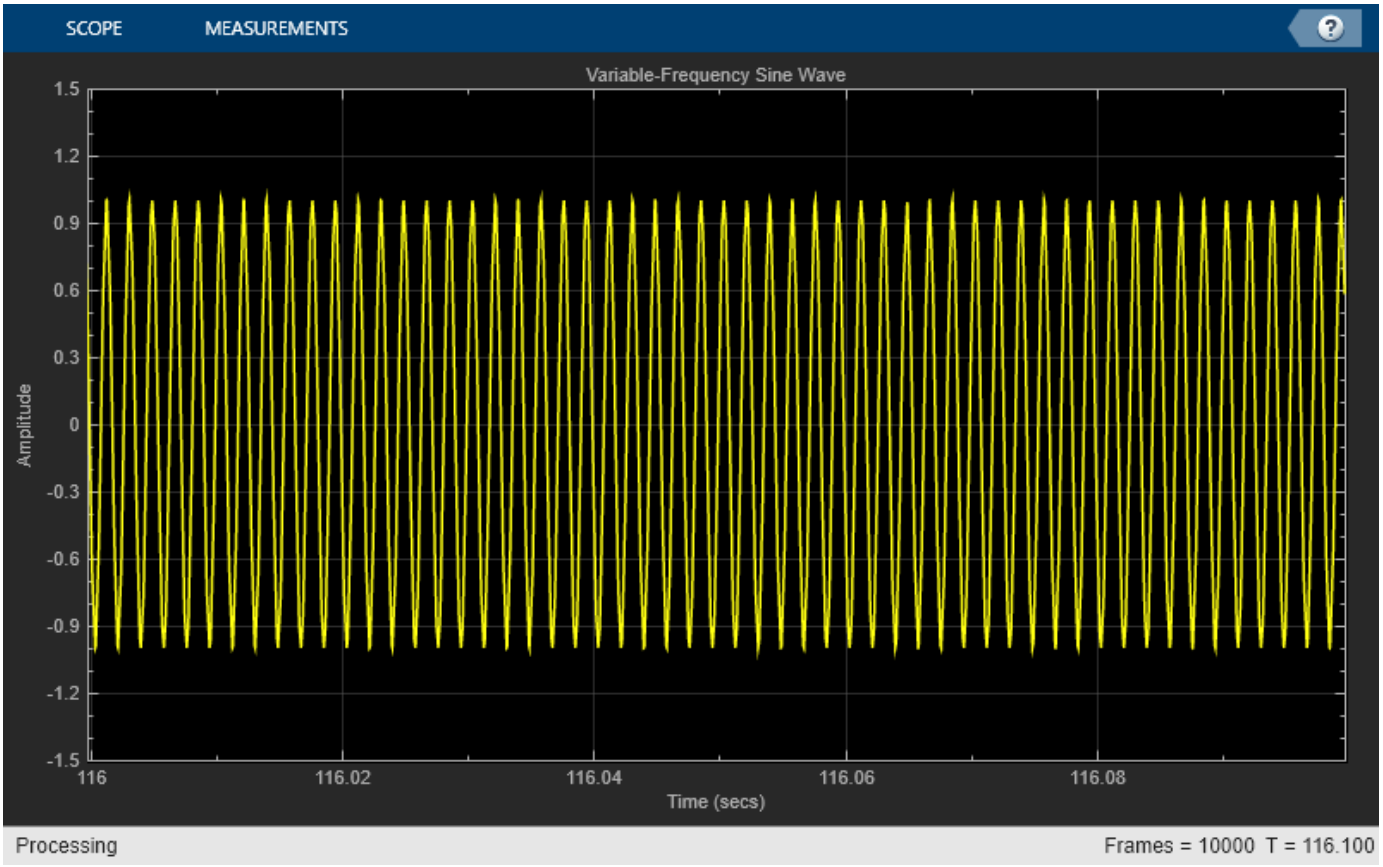
```
osc = audioOscillator;
```

Create a time scope to visualize the variable-frequency sine wave generated by the audio oscillator.

```
scope = timescope( ...
    'SampleRate',osc.SampleRate, ...
    'TimeSpanSource','Property','TimeSpan',0.1, ...
    'YLimits',[-1.5,1.5], ...
    'TimeSpanOverrunAction','Scroll', ...
    'ShowGrid',true, ...
    'Title','Variable-Frequency Sine Wave');
```

Place the audio oscillator in an audio stream loop. Increase the frequency of your sine wave in 50-Hz increments.

```
counter = 0;
while (counter < 1e4)
    counter = counter + 1;
    sineWave = osc();
    scope(sineWave);
    if mod(counter,1000)==0
        osc.Frequency = osc.Frequency + 50;
    end
end
```



Create a Melody by Tuning Oscillation Frequency

Tune the frequency of an audio oscillator at regularly spaced intervals to create a melody. Play the melody to your audio output device.

Create a structure to hold the frequency values of notes in a melody.

```
notes = struct('C4',261.63,'E4',329.63,'G4sharp',415.30,'A4',440,'B4',493.88, ...
    'C5',523.25,'D5',587.25,'D5sharp',622.25,'E5',659.25,'Silence',0);
```

Create `audioOscillator` and `audioDeviceWriter` objects. Use the default settings.

```
osc = audioOscillator;
aDW = audioDeviceWriter;
```

Create a vector with the initial melody of Fur Elise.

```
melody = [notes.Silence notes.Silence,...
    notes.E5 notes.D5sharp notes.E5 notes.D5sharp notes.E5 notes.B4 ...
        notes.D5 notes.C5 notes.A4 notes.A4 notes.Silence ...
    notes.C4 notes.E4 notes.A4 notes.B4 notes.B4 notes.Silence ...
    notes.E4 notes.G4sharp notes.B4 notes.C5 notes.C5 notes.Silence];
```

Specify the note duration in seconds. In an audio stream loop, call your audio oscillator and write the sound to your audio device. Update the frequency of the audio oscillator in `noteDuration` time steps to follow the melody. As a best practice, release your objects once complete.

```
noteDuration = 0.3;

i = 1;
tic
while i < numel(melody)
    tone = osc();
    aDW(tone);
    if toc >= noteDuration
        i = i + 1;
        osc.Frequency = melody(i);
        tic
    end
end

release(osc)
release(aDW)
```

Control Cutoff Frequency of Lowpass Filter

Create a low-frequency oscillator (LFO) lowpass filter, using the `audioOscillator` as a control signal.

Create `dsp.AudioFileReader` and `audioDeviceWriter` System objects to read from an audio file and write to your audio device. Create a biquad filter object to apply lowpass filtering to your audio signal.

```
fileReader = dsp.AudioFileReader('Filename','Engine-16-44p1-stereo-20sec.wav');
deviceWriter = audioDeviceWriter('SampleRate',fileReader.SampleRate);
lowpassFilter = dsp.BiquadFilter( ...
    'SOSMatrixSource','Input port', ...
    'ScaleValuesInputPort',false);
```

Create an audio oscillator object. Your audio oscillator controls the cutoff frequency of the lowpass filter in an audio stream loop.

```
osc = audioOscillator('SignalType','sawtooth', ...
    'DCOffset',0.05, ...
    'Amplitude',0.03, ...
    'SamplesPerFrame',fileReader.SamplesPerFrame, ...
    'SampleRate',fileReader.SampleRate, ...
    'Frequency',5);
```

In a loop, filter the audio signal through the lowpass filter. Write the output signal to your audio device.

```
while ~isDone(fileReader)
    audioIn    = fileReader();
    ctrlSignal = osc();
    [B,A]      = designVarSlopeFilter(48,ctrlSignal(end));
    audioOut   = lowpassFilter(audioIn,B,A);
```

```
    deviceWriter(audioOut);
end
```

As a best practice, release objects once complete.

```
release(osc)
release(fileReader)
release(deviceWriter)
```

For a more complete implementation of an LFO Filter, see audiopluginexample.LFOFilter in the “Audio Plugin Example Gallery”.

Version History

Introduced in R2016a

R2022b: New `MaxSamplesPerFrame` property

Use the `MaxSamplesPerFrame` property to specify the maximum number of samples per frame. Setting the property to a lower value can save memory when using code generation.

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

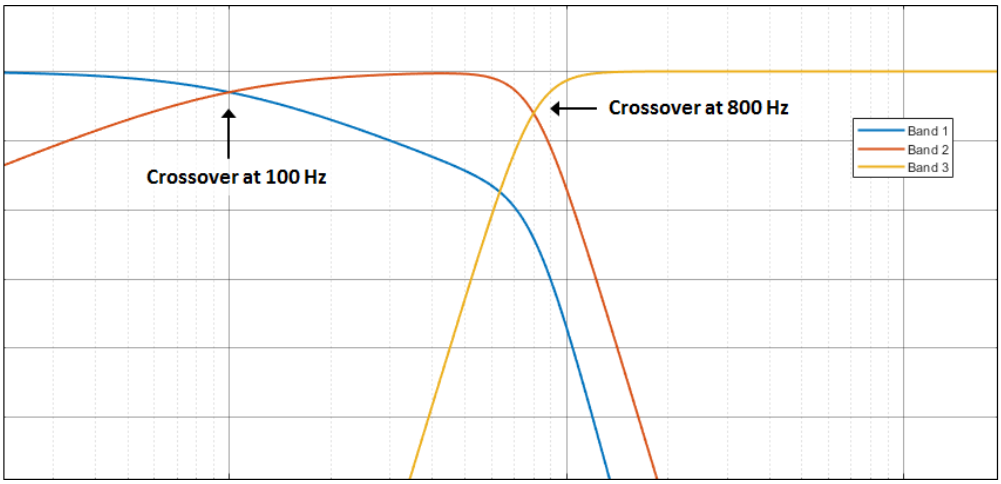
`wavetableSynthesizer` | Audio Oscillator

crossoverFilter

Audio crossover filter

Description

The `crossoverFilter` System object implements an audio crossover filter, which is used to split an audio signal into two or more frequency bands. Crossover filters are multiband filters whose overall magnitude frequency response is flat.



To implement an audio crossover filter:

1. Create the `crossoverFilter` object and set its properties.
2. Call the object with arguments, as if it were a function.

To learn more about how System objects work, see What Are System Objects?

Creation

Syntax

```
crossFilt = crossoverFilter
crossFilt = crossoverFilter(nCrossovers)
crossFilt = crossoverFilter(nCrossovers,xFrequencies)
crossFilt = crossoverFilter(nCrossovers,xFrequencies,xSlopes)
crossFilt = crossoverFilter(nCrossovers,xFrequencies,xSlopes,Fs)
crossFilt = crossoverFilter(___,Name,Value)
```

Description

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

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

`crossFilt = crossoverFilter(nCrossovers,xFrequencies)` sets the CrossoverFrequencies property to `xFrequencies`.

`crossFilt = crossoverFilter(nCrossovers,xFrequencies,xSlopes)` sets the CrossoverSlopes property to `xSlopes`.

`crossFilt = crossoverFilter(nCrossovers,xFrequencies,xSlopes,Fs)` sets the SampleRate property to `Fs`.

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

Example: `crossFilt = crossoverFilter(2,'CrossoverFrequencies', [100,800],'CrossoverSlopes',[6,48])` creates a System object, `crossFilt`, with two crossovers located at 100 Hz and 800 Hz, and crossover slopes of 6 dB/octave and 48 dB/octave, respectively.

Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see System Design in MATLAB Using System Objects.

`NumCrossovers` — Number of magnitude response band crossings

1 (default) | 2 | 3 | 4

Number of magnitude response band crossings, specified as a scalar integer in the range 1 to 4.

The number of bands output when implementing crossover filtering is one more than the `NumCrossovers` value.

Number of magnitude response band crossings	Number of bands output
1	two-band
2	three-band
3	four-band
4	five-band

Tunable: No

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

`CrossoverFrequencies` — Crossover frequencies (Hz)

`100` (default) | scalar | vector

Crossover frequencies in Hz, specified as a scalar or vector of real values of length `NumCrossovers`.

Crossover frequencies are the intersections of magnitude response bands of the individual two-band crossover filters used in the multiband crossover filter.

Tunable: Yes

Data Types: `single` | `double`

`CrossoverSlopes` — Crossover slopes (dB/octave)
`12` (default) | scalar | vector

Crossover slopes in dB/octave, specified as a scalar or vector of real values in the range [6:6:48]. If a specified crossover slope is not inside the range, the slope is rounded to the nearest allowed value.

- If `CrossoverSlopes` is a scalar, all two-band component crossover slopes take that value.
- If `CrossoverSlopes` is a vector of length NumCrossovers, the respective two-band component crossover slopes take those values.

Crossover slopes are the slopes of individual bands at the associated crossover frequency, as specified in the two-band component crossover.

Tunable: Yes

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

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

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

Tunable: Yes

Data Types: `single` | `double`

Usage

Syntax

```
[band1,...,bandN] = crossFilt(audioIn)
```

Description

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

Input Arguments

`audioIn` — Audio input to crossover filter
matrix

Audio input to the crossover filter, specified as a matrix. The columns of the matrix are treated as independent audio channels.

Data Types: `single` | `double`

Output Arguments

`[band1,...,bandN]` — Audio bands output from crossover filter
set of matrices

Audio bands output from the crossover filter, returned as a set of `N` bands. The NumCrossovers property determines the number of return arguments: `N = NumCrossovers + 1`. The size of each output argument is the same size as `audioIn`.

Data Types: `single` | `double`

Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

Specific to crossoverFilter

visualize	Visualize magnitude response of crossover filter
cost	Estimate implementation cost of audio System objects
createAudioPluginClass	Create audio plugin class that implements functionality of System object
parameterTuner	Tune object parameters while streaming

MIDI

configureMIDI	Configure MIDI connections between audio object and MIDI controller
disconnectMIDI	Disconnect MIDI controls from audio object
getMIDIConnections	Get MIDI connections of audio object

Common to All System Objects

clone	Create duplicate System object
isLocked	Determine if System object is in use
release	Release resources and allow changes to System object property values and input characteristics
reset	Reset internal states of System object
step	Run System object algorithm

The `createAudioPluginClass` and `configureMIDI` functions map tunable properties of the `crossoverFilter` System object to user-facing parameters:

Property	Range	Mapping	Unit
`CrossoverFrequencies`	[20, 20000]	linear	Hz
`CrossoverSlopes`	[6, 48]	linear	dB/octave

Examples

Pass Noise Signal Through Crossover Filter

Use the `crossoverFilter` object to split Gaussian noise into three separate frequency bands.

Create a 5 second noise signal that assumes a 24 kHz sample rate.

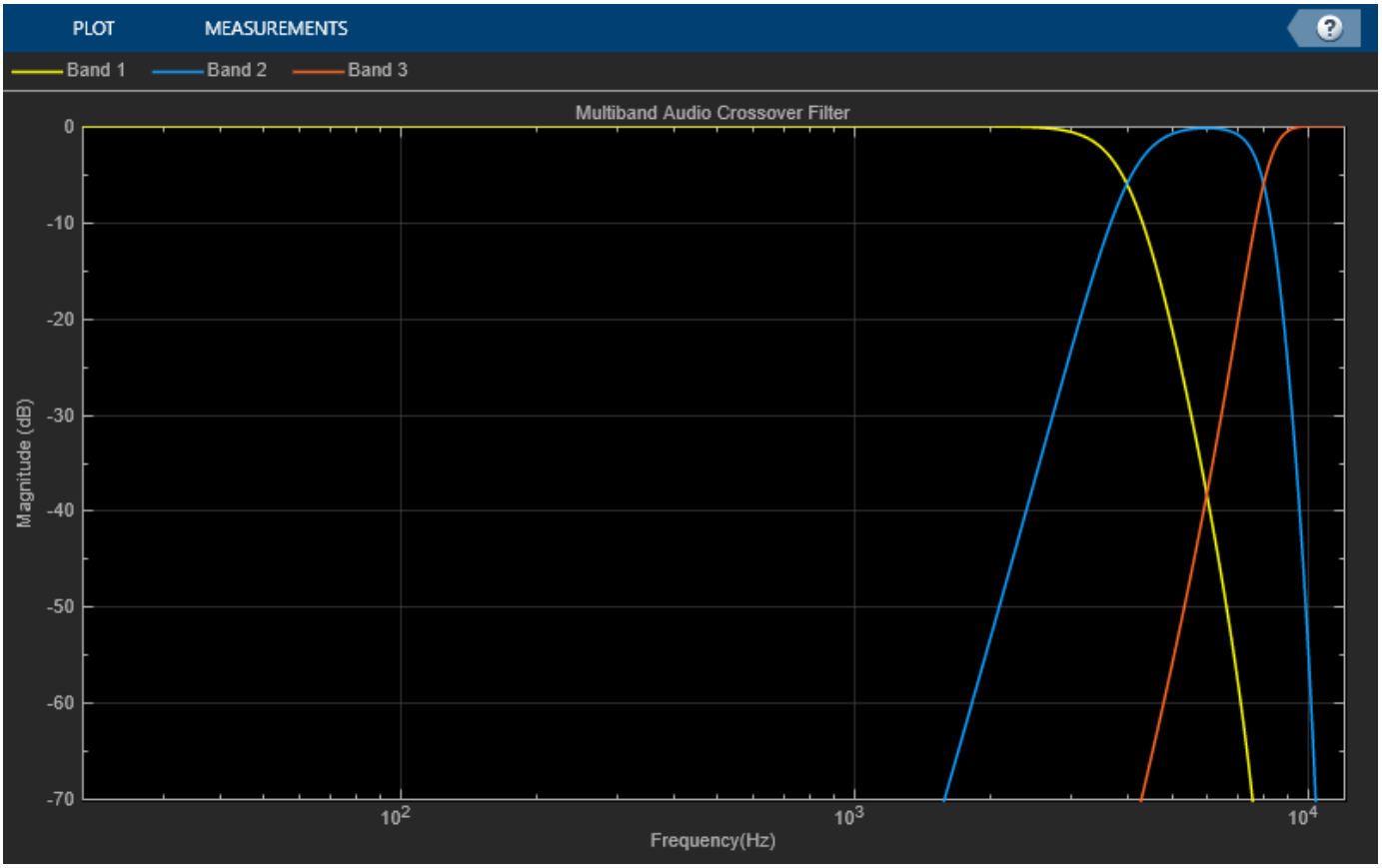
```
fs = 24e3;
noise = randn(fs*5,1);
```

Create a `crossoverFilter` 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=fs);
```

Visualize the magnitude response of your crossover filter object.

```
visualize(crossFilt)
```



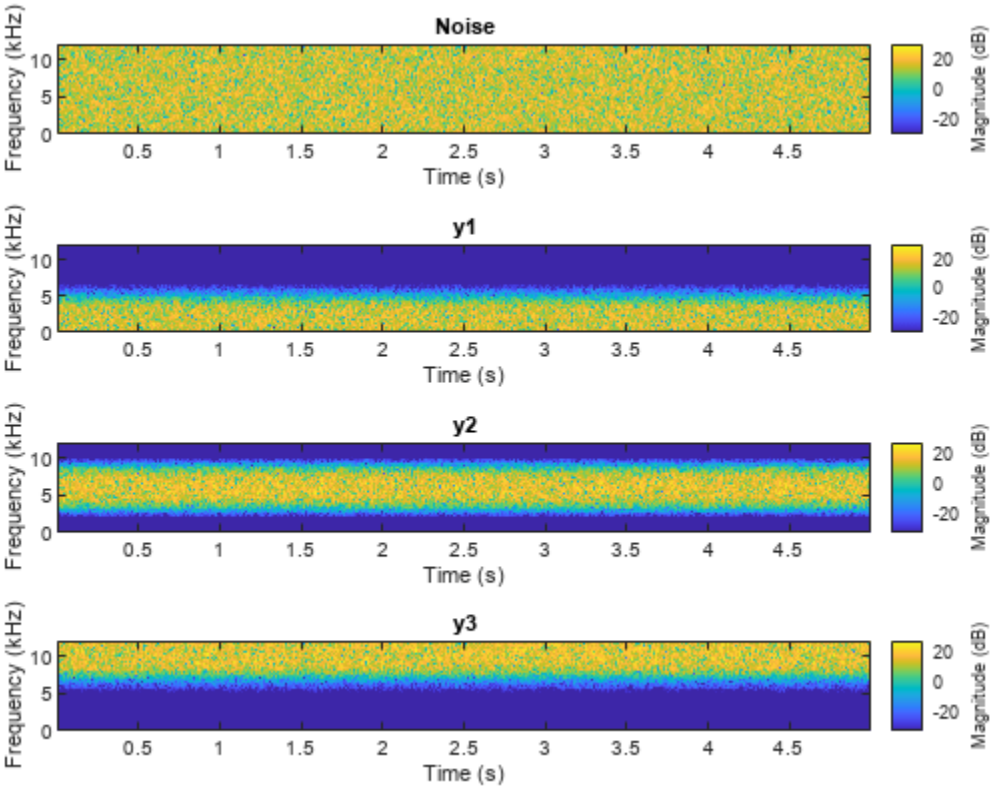
Call your crossover filter like a function with the noise signal as the argument.

```
[y1,y2,y3] = crossFilt(noise);
```

Visualize the results using a spectrogram.

```
tiledlayout(4,1)

nexttile
```

```
stft(noise,fs,FrequencyRange="onesided")
title("Noise")

nexttile
stft(y1,fs,FrequencyRange="onesided")
title("y1")

nexttile
stft(y2,fs,FrequencyRange="onesided")
title("y2")

nexttile
stft(y3,fs,FrequencyRange="onesided")
title("y3")
```



Split Audio Signal into Three Bands

Use the `crossoverFilter` object to split an audio signal into three frequency bands.

Create the `dsp.AudioFileReader` and `audioDeviceWriter` objects. Use the sample rate of the reader as the sample rate of the writer.

```
samplesPerFrame = 256;

fileReader = dsp.AudioFileReader( ...
```

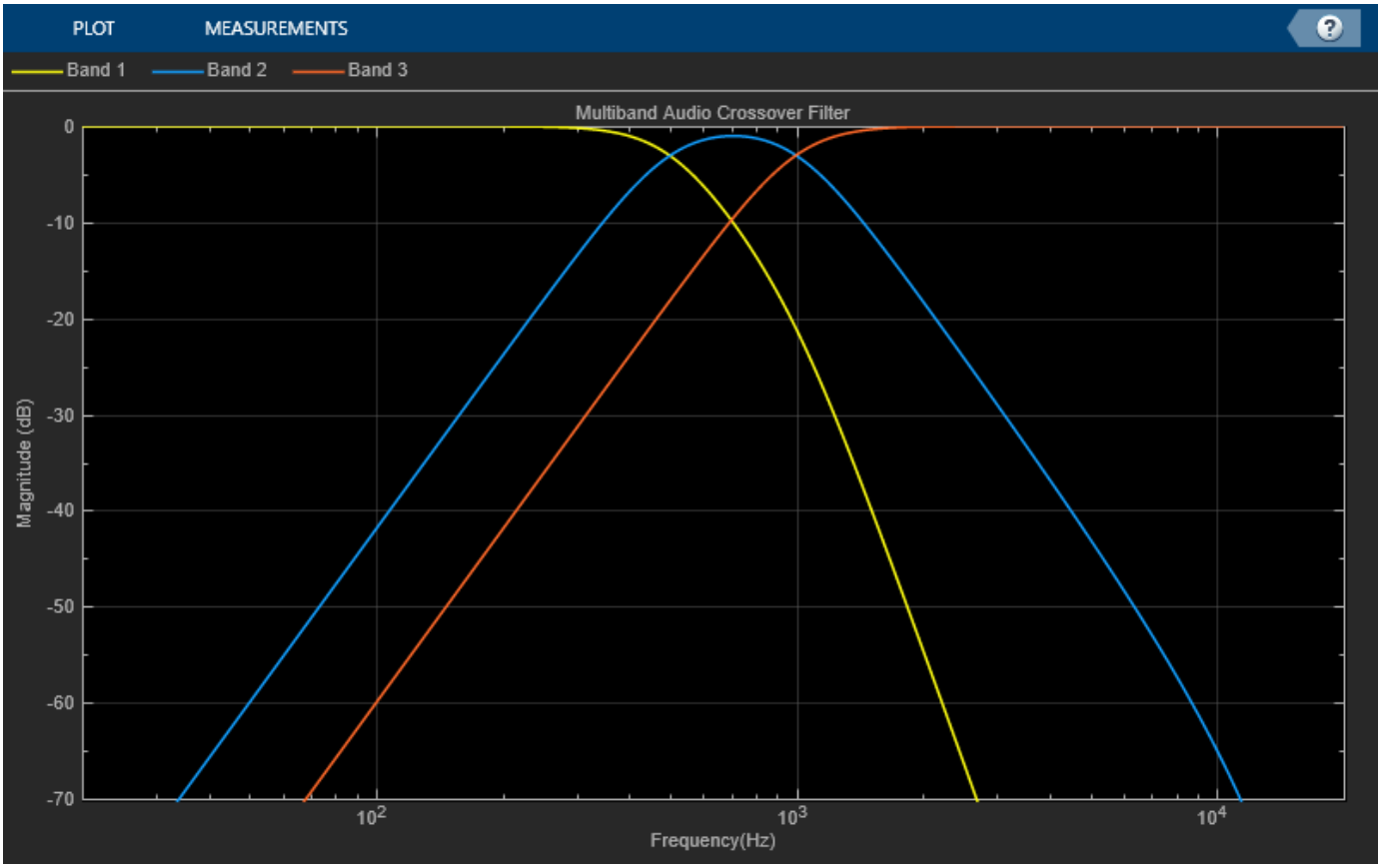
```
    "RockGuitar-16-44p1-stereo-72secs.wav", ...
    SamplesPerFrame=samplesPerFrame);
deviceWriter = audioDeviceWriter( ...
    SampleRate=fileReader.SampleRate);
```

Create a `crossoverFilter` 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);
```

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
```

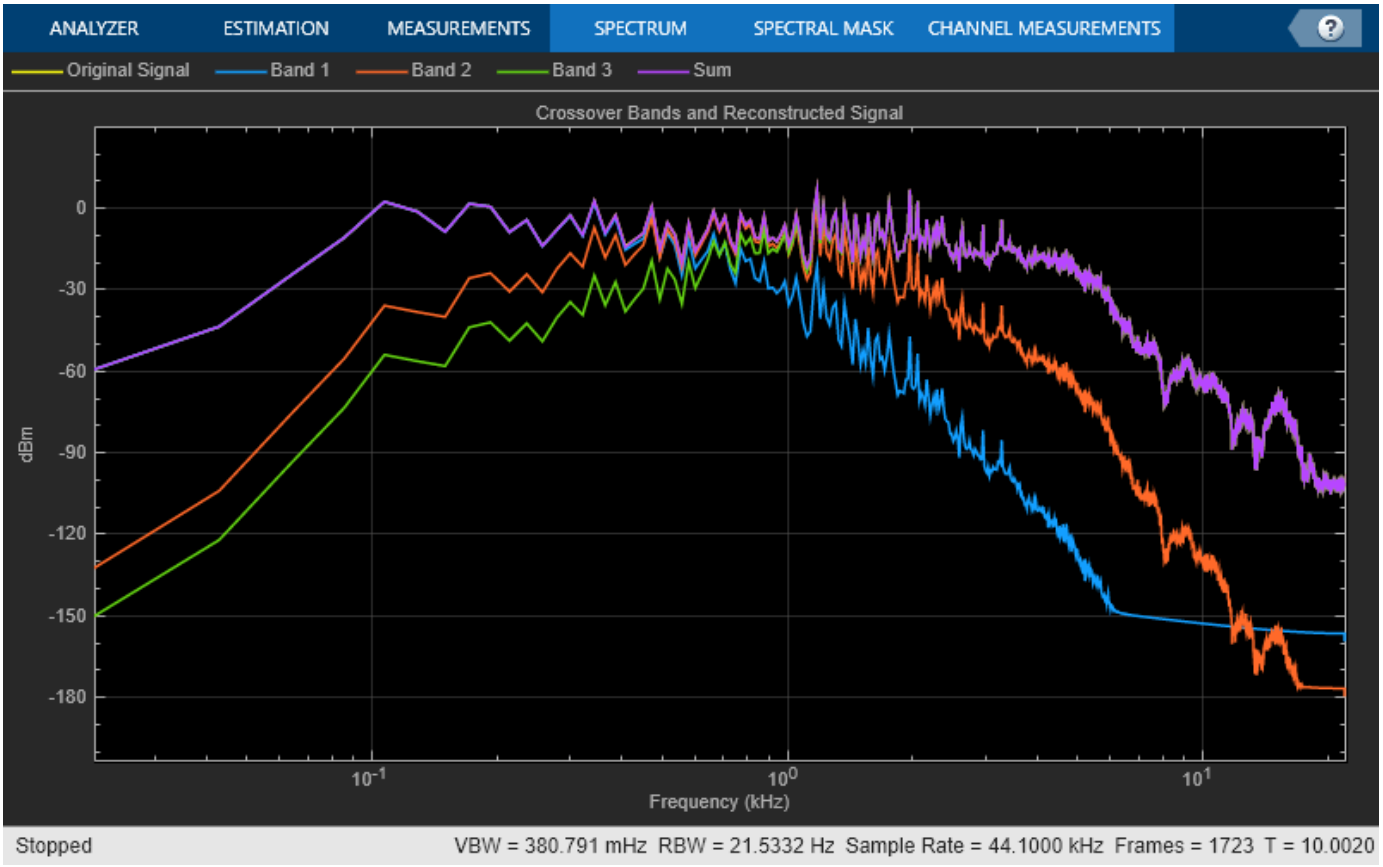
```
          AdditionsPerInputSample: 37
```

Create a spectrum analyzer to visualize the effect of the crossover filter.

```
scope = spectrumAnalyzer( ...
    SampleRate=fileReader.SampleRate, ...
    PlotAsTwoSidedSpectrum=false, ...
    FrequencyScale="log", ...
    Title="Crossover Bands and Reconstructed Signal", ...
    ShowLegend=true, ...
    ChannelNames=["Original Signal","Band 1","Band 2","Band 3","Sum"]);
```

Play 10 seconds of the audio signal. Visualize the spectrum of the original audio, the crossover bands, and the reconstructed signal (sum of bands).

```
setup(scope,ones(samplesPerFrame,5))
count = 0;
while count < (fileReader.SampleRate/samplesPerFrame)*10
    originalSignal = fileReader();
    [band1,band2,band3] = crossFilt(originalSignal);
    sumOfBands = band1 + band2 + band3;
    scope([originalSignal(:,1), ...
           band1(:,1), ...
           band2(:,1), ...
           band3(:,1), ...
           sumOfBands(:,1)])
    deviceWriter(sumOfBands);
    count = count + 1;
end

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



Apply Split-Band De-Essing

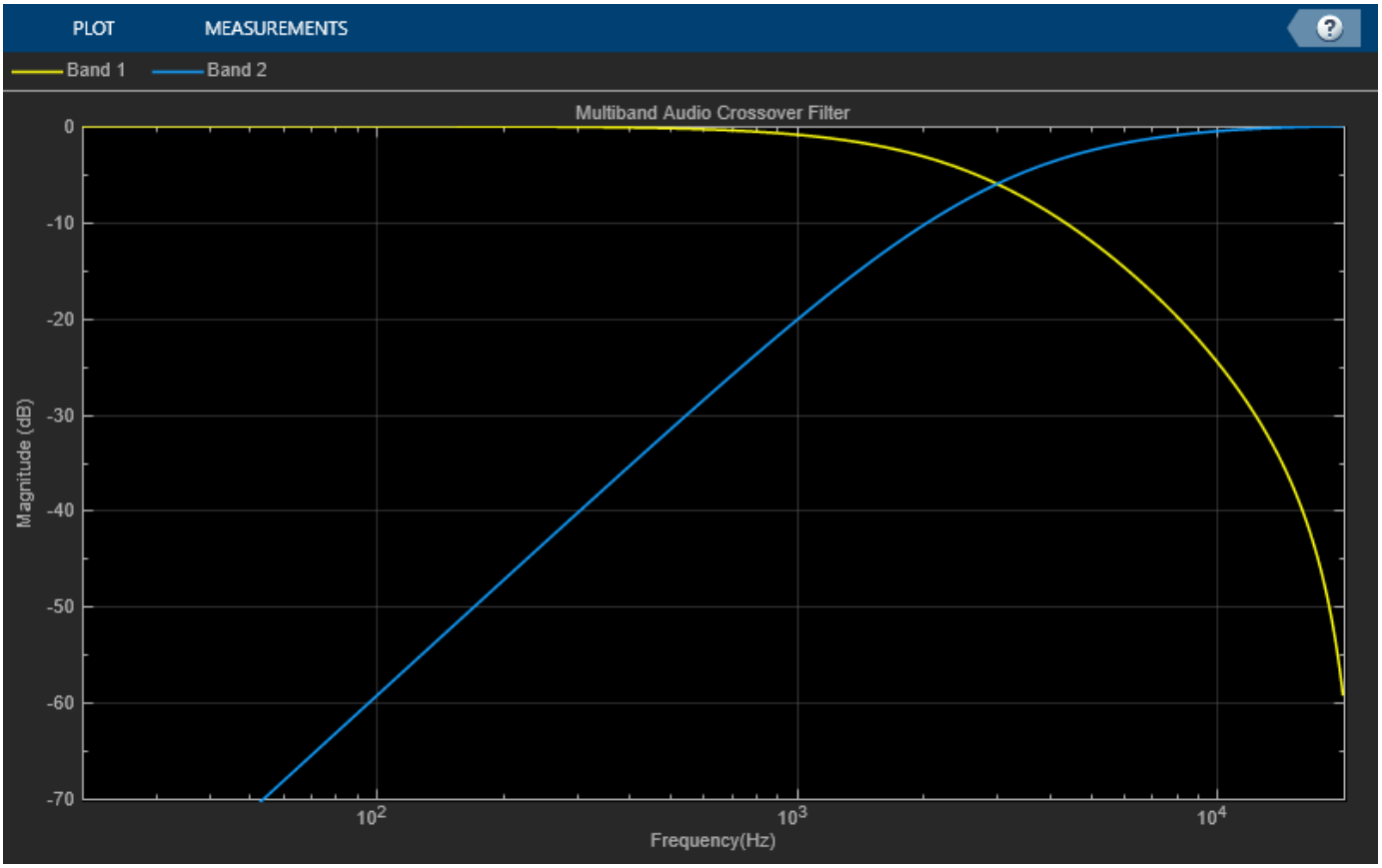
De-essing is the process of diminishing sibilant sounds in an audio signal. Sibilance refers to the *s*, *z*, and *sh* sounds in speech, which can be disproportionately emphasized during recording. *es* sounds fall under the category of unvoiced speech with all consonants and have a higher frequency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

```
scope = timescope( ...
    'SampleRate',fileReader.SampleRate, ...
    'TimeSpanOverrunAction','Scroll', ...
    'TimeSpanSource','Property','TimeSpan',4, ...
    'BufferLength',fileReader.SampleRate*8, ...
    'YLimits',[-1 1], ...
    'ShowGrid',true, ...
```

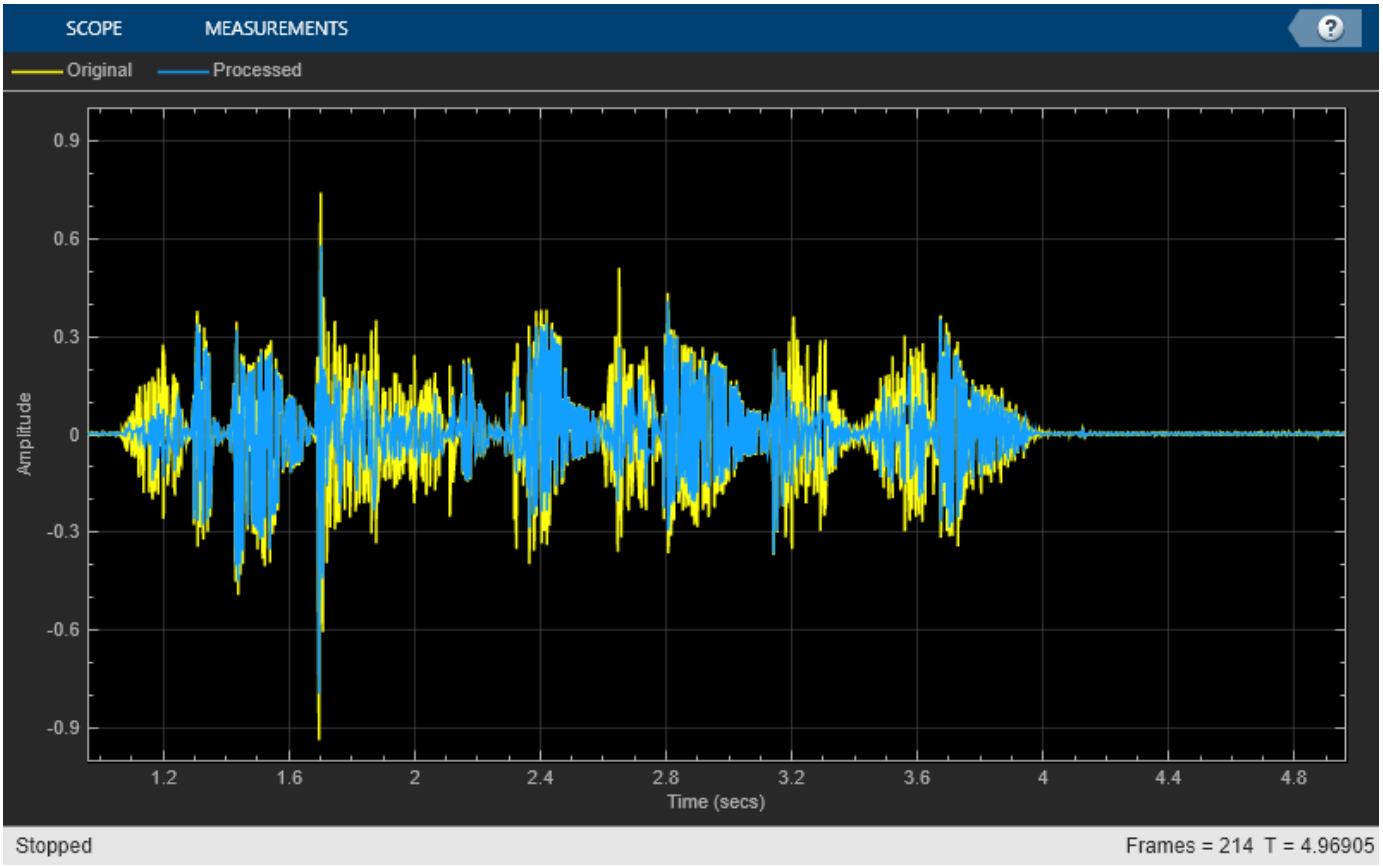
```
    'ShowLegend',true, ...
    'ChannelNames',{'Original','Processed'});
```

In an audio stream loop:

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

As a best practice, release your objects once done.

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

    [band1,band2] = crossFilt(audioIn);

    band2processed = dRExpander(band2);

    procAudio  = band1 + band2processed;

    deviceWriter(procAudio);

    scope([audioIn procAudio]);
end

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




```
release(crossFilt)
release(dRExpander)
```

Diminish Plosives from Speech Signal

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

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

```
fileReader = dsp.AudioFileReader('audioPlosives.wav');
deviceWriter = audioDeviceWriter('SampleRate',fileReader.SampleRate);

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` 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/(fileReader.SampleRate/2),"hi");
biquadFilter = dsp.BiquadFilter( ...
    "SOSMatrixSource","Input port", ...
    "ScaleValuesInputPort",false);

crossFilt = crossoverFilter( ...
    "SampleRate",fileReader.SampleRate, ...
    "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 = timescope( ...
    "SampleRate",fileReader.SampleRate, ...
    "TimeSpanSource","property","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.

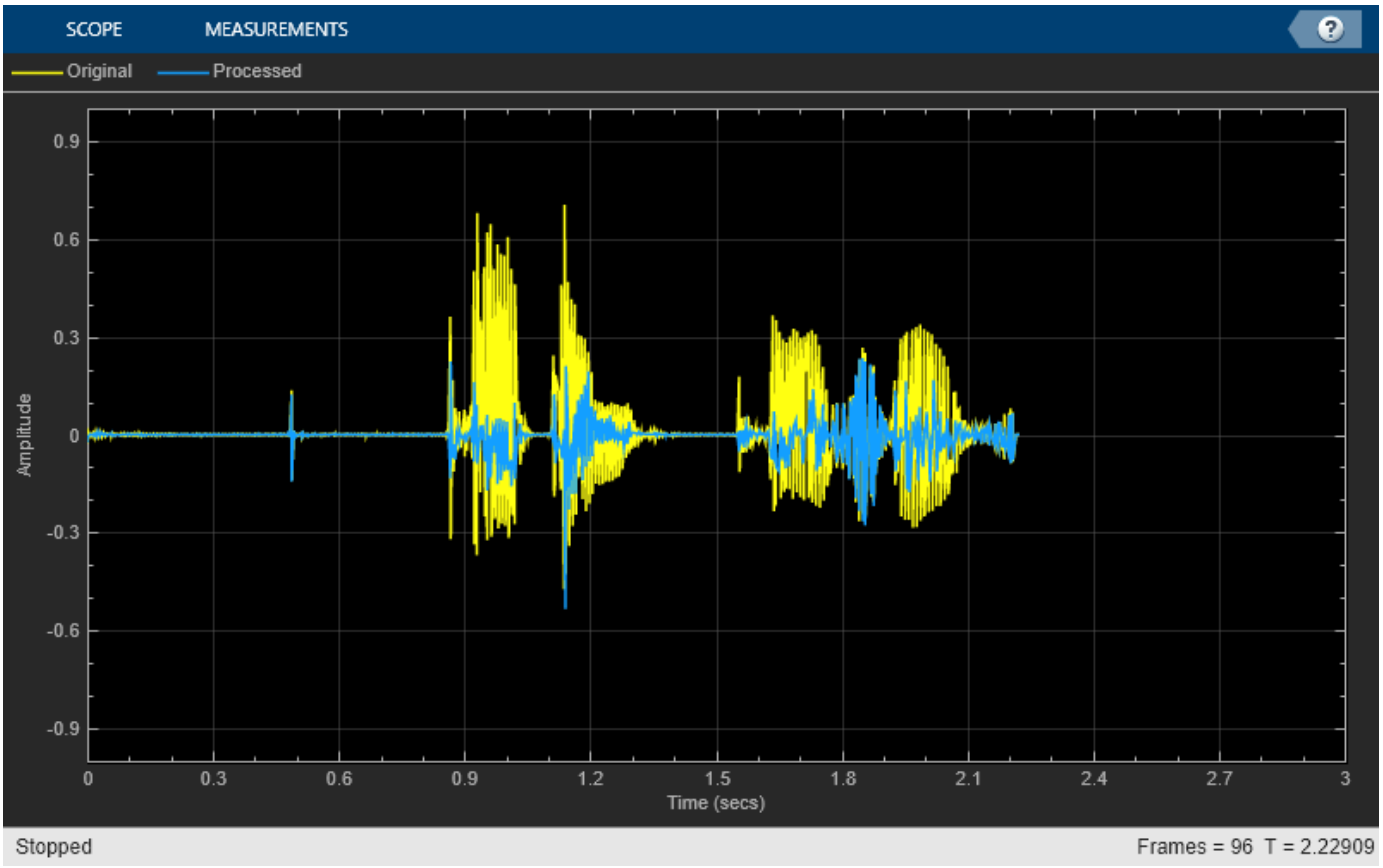
As a best practice, release your objects once done.

```
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
```

As a best practice, release your objects once done.

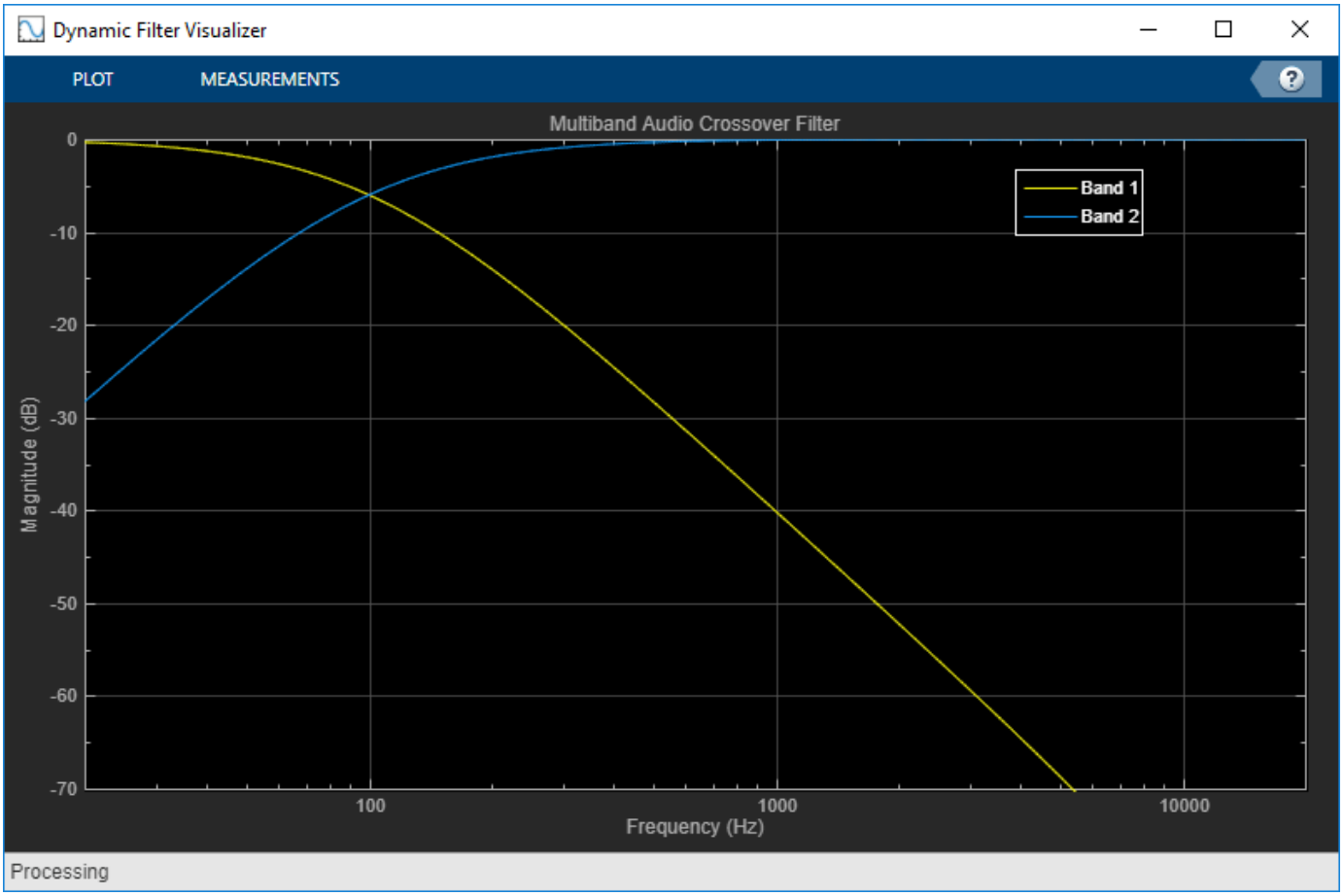
```
release(deviceWriter)
release(fileReader)
release(crossFilt)
release(dRCompressor)
release(scope)
```



Tune Crossover Filter Parameters

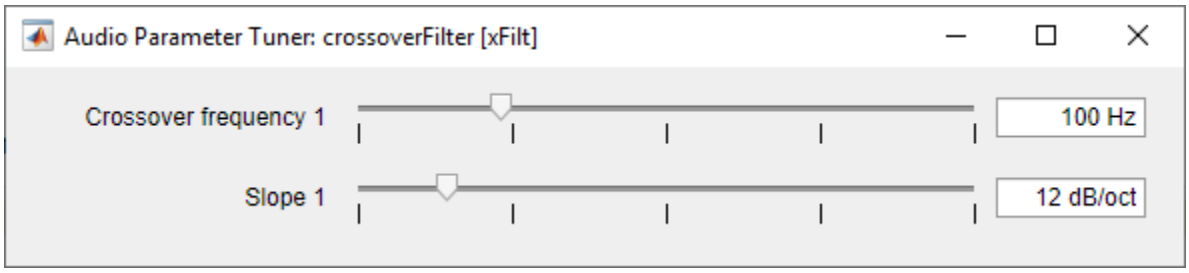
Create a `dsp.AudioFileReader` to read in audio frame-by-frame. Create a `audioDeviceWriter` to write audio to your sound card. Create a `crossoverFilter` to process the audio data. Call visualize to plot the frequency responses of the filters.

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

xFilt = crossoverFilter('SampleRate',fileReader.SampleRate);
visualize(xFilt)
```



Call `parameterTuner` to open a UI to tune parameters of the crossover filter while streaming.

```
parameterTuner(xFilt)
```



In an audio stream loop:

1. Read in a frame of audio from the file.
2. Apply crossover filtering.
3. Write the frame of audio to your audio device for listening.

While streaming, tune parameters of the crossover filter and listen to the effect.

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

```
    [low,high] = xFilt(audioIn);
    deviceWriter([low(:,1),high(:,1)]);
    drawnow limitrate % required to update parameter
end
```

As a best practice, release your objects once done.

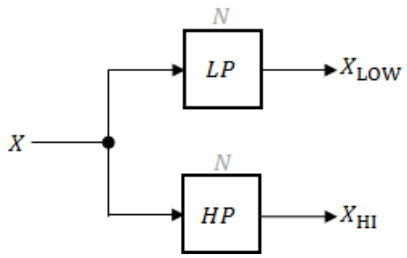
```
release(deviceWriter)
release(fileReader)
release(xFilt)
```

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

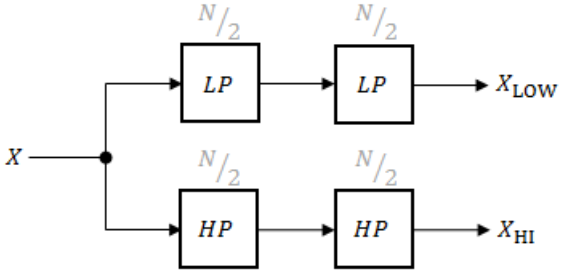
Odd-order two-band (one crossover) filters are implemented as parallel complementary highpass and lowpass filters.



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.

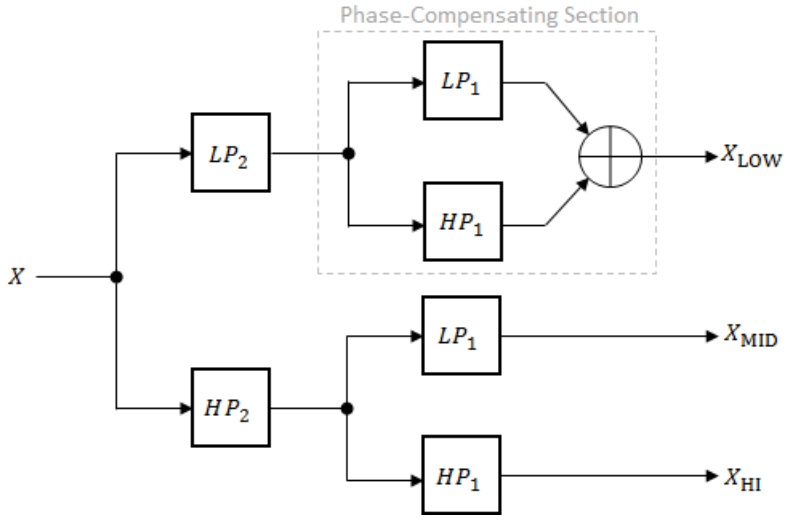


LP and *HP* are Butterworth filters of order *N*/2, where *N* is the order of the overall filter. The filters are implemented as direct-form II transposed second-order sections.

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.

Version History

Introduced in R2016a

References

[1] D’Appolito, Joseph A. "Active Realization of Multiway All-Pass Crossover Systems." *Journal of Audio Engineering Society*. Vol. 35, Issue 4, 1987, pp. 239–245.

Extended Capabilities

C/C++ Code Generation

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

Usage notes and limitations:

“System Objects in MATLAB Code Generation” (MATLAB Coder)

See Also

Objects
`multibandParametricEQ`

Blocks
Crossover Filter

visualize

Visualize magnitude response of crossover filter

Syntax

```
visualize(crossFilt)
visualize(crossFilt,NFFT)
hvsz = visualize( ___ )
```

Description

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

`visualize(crossFilt,NFFT)` specifies an N-point FFT used to calculate the magnitude response.

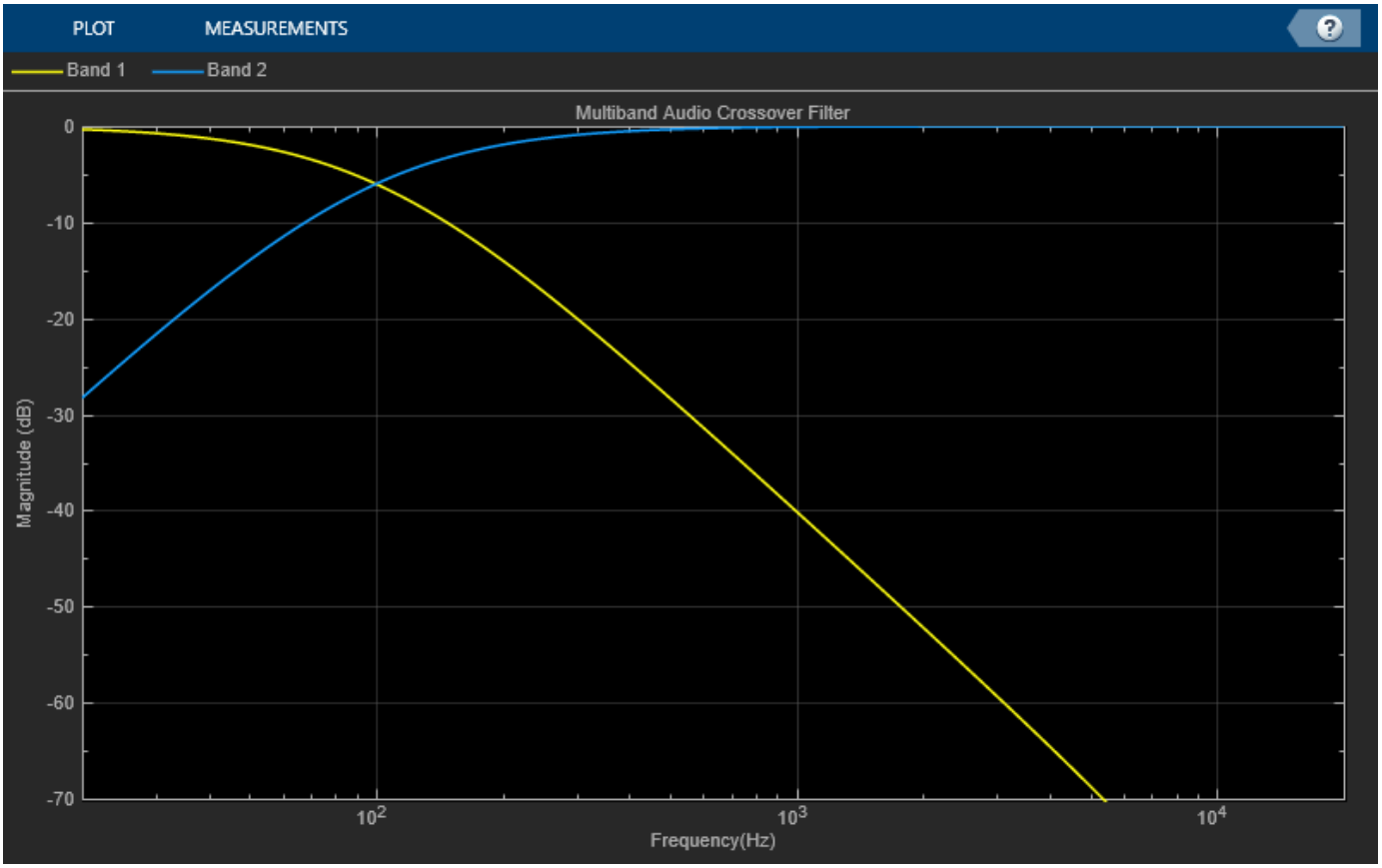
`hvsz = visualize( ___ )` returns a handle to the visualizer as a `dsp.DynamicFilterVisualizer` object when called with any of the previous syntaxes.

Examples

Visualize Magnitude Response of Crossover Filter

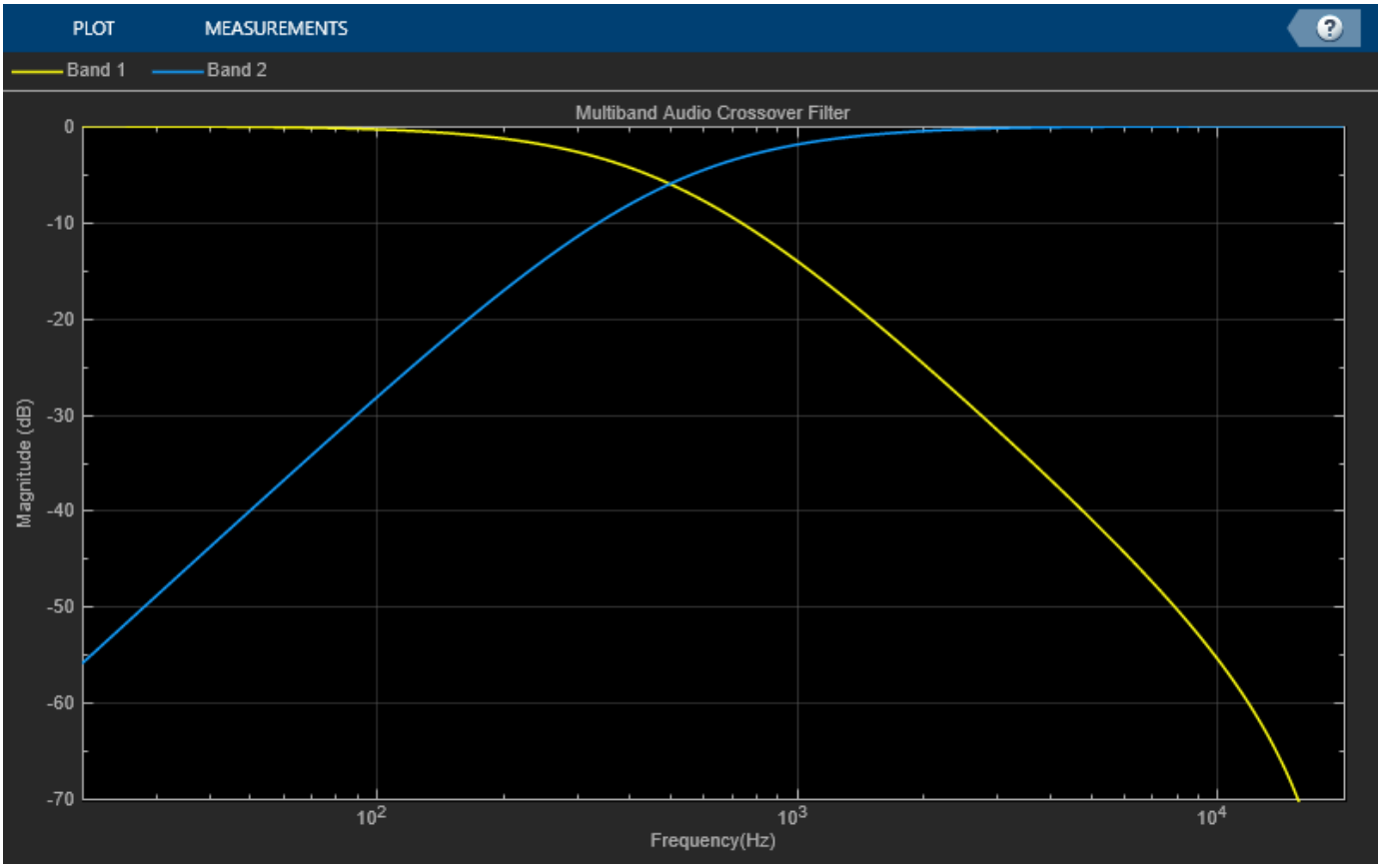
Create a `crossoverFilter` object, and then call `visualize` to plot the magnitude response of the filter.

```
crossFilt = crossoverFilter;
visualize(crossFilt)
```



Modify the crossover frequency and observe that the plot is updated automatically.

```
crossFilt.CrossoverFrequencies = 500;
```



Input Arguments

`crossFilt` — Crossover filter to visualize

object of `crossoverFilter` System object

Crossover filter whose magnitude response you want to plot.

NFFT — N-point FFT

2048 (default) | positive scalar

Number of bins used to calculate the DFT, specified as a positive scalar.

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

Version History

Introduced in R2016a

See Also

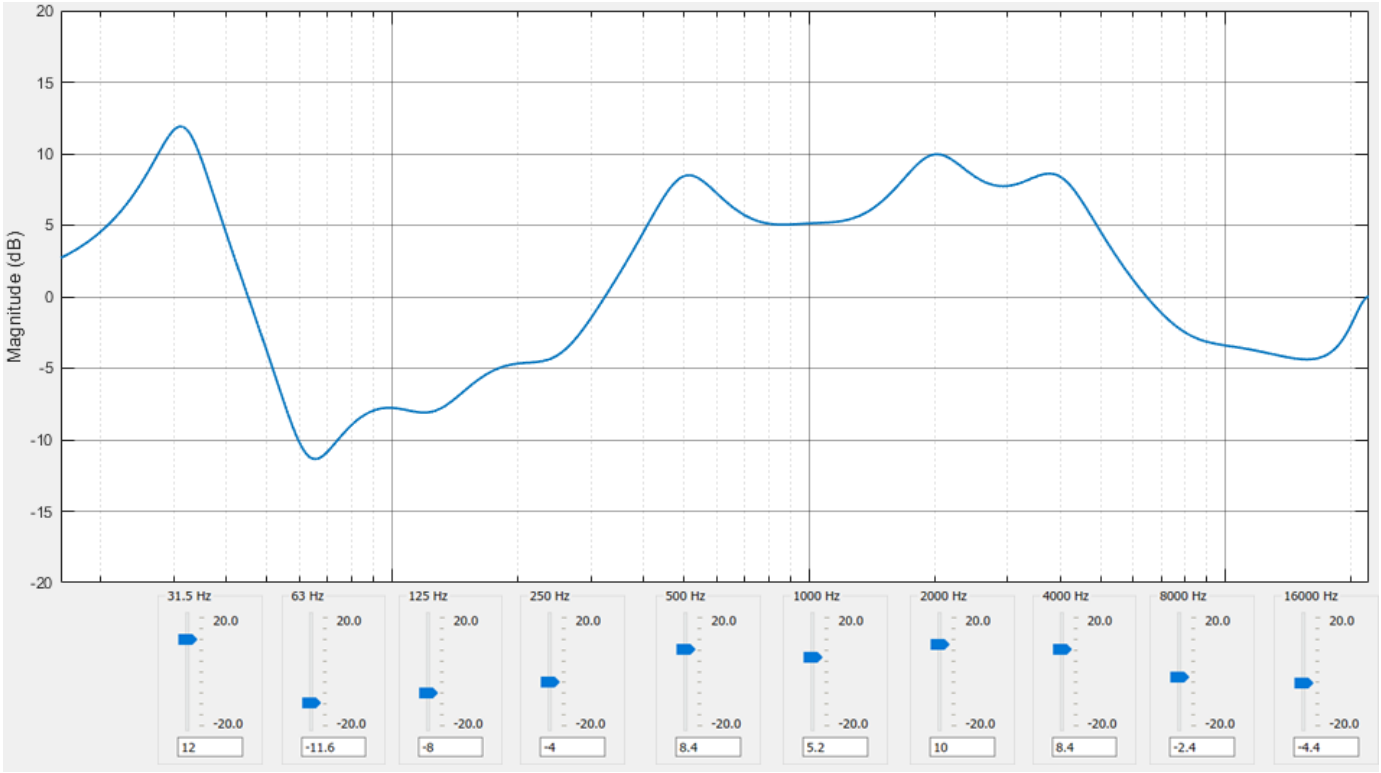
`crossoverFilter`

graphicEQ

Standards-based graphic equalizer

Description

The `graphicEQ` System object implements a graphic equalizer that can tune the gain on individual octave or fractional octave bands. The object filters the data independently across each input channel over time using the filter specifications. Center and edge frequencies of the bands are based on the ANSI S1.11-2004 standard.



To equalize an audio signal:

1. Create the `graphicEQ` object and set its properties.
2. Call the object with arguments, as if it were a function.

To learn more about how System objects work, see What Are System Objects?

Creation

Syntax

```
equalizer = graphicEQ
```

```
equalizer = graphicEQ(Name,Value)
```

Description

`equalizer = graphicEQ` creates a graphic equalizer with default values.

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

Example: `equalizer = graphicEQ('Structure','Parallel','EQOrder','1/3 octave')` creates a System object, `equalizer`, which implements filtering using a parallel structure and one-third octave filter bandwidth.

Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see System Design in MATLAB Using System Objects.

`Gains` — Gain of each octave or fractional octave band (dB)
`[0 0 0 0 0 0 0 0 0 0]` (default) | 10-, 15-, or 30-element row vector

Gain of each octave of fractional octave band in dB, specified as a row vector with a length determined by the Bandwidth property:

- `'1 octave'` –– Specify gains as a 10-element row vector.
- `'2/3 octave'` –– Specify gains as a 15-element row vector.
- `'1/3 octave'` –– Specify gains as a 30-element row vector.

Example: `equalizer = graphicEQ('Bandwidth','2/3 octave','Gains', [5,5,5,5,5,0,0,0,0,0,-5,-5,-5,-5,-5])` creates a two-third octave graphic equalizer with specified gains.

You can tune the gains of your graphic equalizer when the object is locked. However, you cannot tune the length of the gains when the object is locked.

Tunable: Yes

Data Types: `single` | `double`

`EQOrder` — Order of individual equalizer bands
2 (default) | positive even integer

Order of individual equalizer bands, specified as a positive even integer. All equalizer bands have the same order.

Tunable: No

Data Types: `single` | `double`

`Bandwidth` — Filter bandwidth (octaves)
`'1 octave'` (default) | `'2/3 octave'` | `'1/3 octave'`

Filter bandwidth in octaves, specified as `'1 octave'`, `'2/3 octave'`, or `'1/3 octave'`.

The ANSI S1.11-2004 standard defines the center and edge frequencies of your equalizer. The ISO 266:1997(E) standard specifies corresponding preferred frequencies for labeling purposes.

1-Octave Bandwidth

Center frequencies	`32 63 126 251 501 1000 1995 3981 7943 15849`
Edge frequencies	`22 45 89 178 355 708 1413 2818 5623 1122 22387`
Preferred frequencies	`31.5 63 125 250 500 1000 2000 4000 8000 16000`

2/3-Octave Bandwidth

Center frequencies	`25 40 63 100 158 251 398 631 1000 1585 2512 3981 6310 10000 15849`
Edge frequencies	`20 32 50 79 126 200 316 501 794 1259 1995 3162 5012 7943 12589 19953`
Preferred frequencies	`25 40 63 100 160 250 400 630 1000 1600 2500 4000 6300 10000 16000`

1/3-Octave Bandwidth

Center frequencies	`25 32 40 50 63 79 100 126 158 200 251 316 398 501 631 794 1000 1259 1585 1995 2512 3162 3981 5012 6310 7943 10000 12589 15849 19953`
Edge frequencies	`22 28 35 45 56 71 89 112 141 178 224 282 355 447 562 708 891 1122 1413 1778 2239 2818 3548 4467 5623 7079 8913 11220 14125 17783 22387`
Preferred frequencies	`25 31.5 40 50 63 80 100 125 160 200 250 315 400 500 630 800 1000 1250 1600 2000 2500 3150 4000 5000 6300 8000 10000 12500 16000 20000`

Tunable: No

Data Types: `char` | `string`

`Structure` — Type of implementation

`'Cascade'` (default) | `'Parallel'`

Type of implementation, specified as `'Cascade'` or `'Parallel'`. See “Algorithms” on page 3-199 and “Graphic Equalization” for information about these implementation structures.

Tunable: No

Data Types: `char` | `string`

SampleRate — Input sample rate (Hz)

44100 (default) | positive scalar

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

Tunable: Yes

Data Types: single | double

Usage

Syntax

```
audioOut = equalizer(audioIn)
```

Description

`audioOut = equalizer(audioIn)` performs graphic equalization on the input signal, `audioIn`, and returns the equalized signal, `audioOut`. The type of equalization is specified by the algorithm and properties of the `graphicEQ` System object, `equalizer`.

Input Arguments

audioIn — Audio input to graphic equalizer

matrix

Audio input to the graphic equalizer, specified as a matrix. The columns of the matrix are treated as independent audio channels.

Data Types: single | double

Output Arguments

audioOut — Audio output from graphic equalizer

matrix

Audio output from the graphic equalizer, returned as a matrix the same size as `audioIn`.

Data Types: single | double

Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

Specific to graphicEQ

createAudioPluginClass	Create audio plugin class that implements functionality of System object
coeffs	Get filter coefficients
info	Get filter information
visualize	Visualize magnitude response of graphic equalizer
parameterTuner	Tune object parameters while streaming

MIDI

configureMIDI	Configure MIDI connections between audio object and MIDI controller
disconnectMIDI	Disconnect MIDI controls from audio object
getMIDIConnections	Get MIDI connections of audio object

Common to All System Objects

clone	Create duplicate System object
isLocked	Determine if System object is in use
release	Release resources and allow changes to System object property values and input characteristics
reset	Reset internal states of System object
step	Run System object algorithm

The `createAudioPluginClass` and `configureMIDI` functions map tunable properties of the `graphicEQ` System object to user-facing parameters:

Property	Range	Mapping	Unit
`Gains`	[–20, 20]	linear	dB

Examples

Perform Graphic Equalization

Create objects to read from an audio file and write to your audio device. Use the sample rate of the reader as the sample rate of the writer.

```
frameLength = 512;
reader = dsp.AudioFileReader('RockDrums-48-stereo-11secs.mp3','SamplesPerFrame',frameLength);
player = audioDeviceWriter('SampleRate',reader.SampleRate);
```

In an audio stream loop, read audio from a file and play the audio through your audio device.

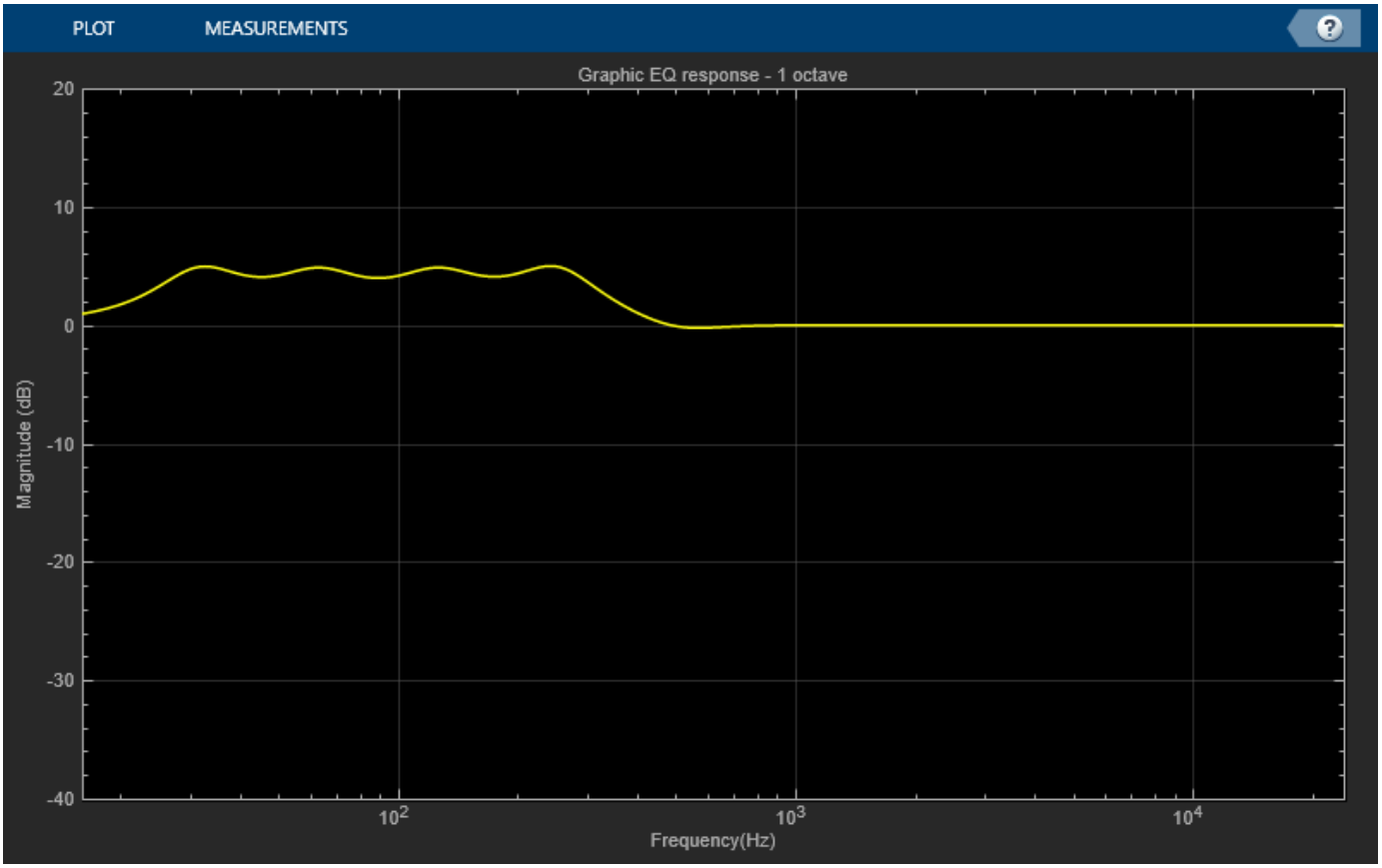
```
while ~isDone(reader)
    x = reader();
    player(x);
end
release(reader)
release(player)
```

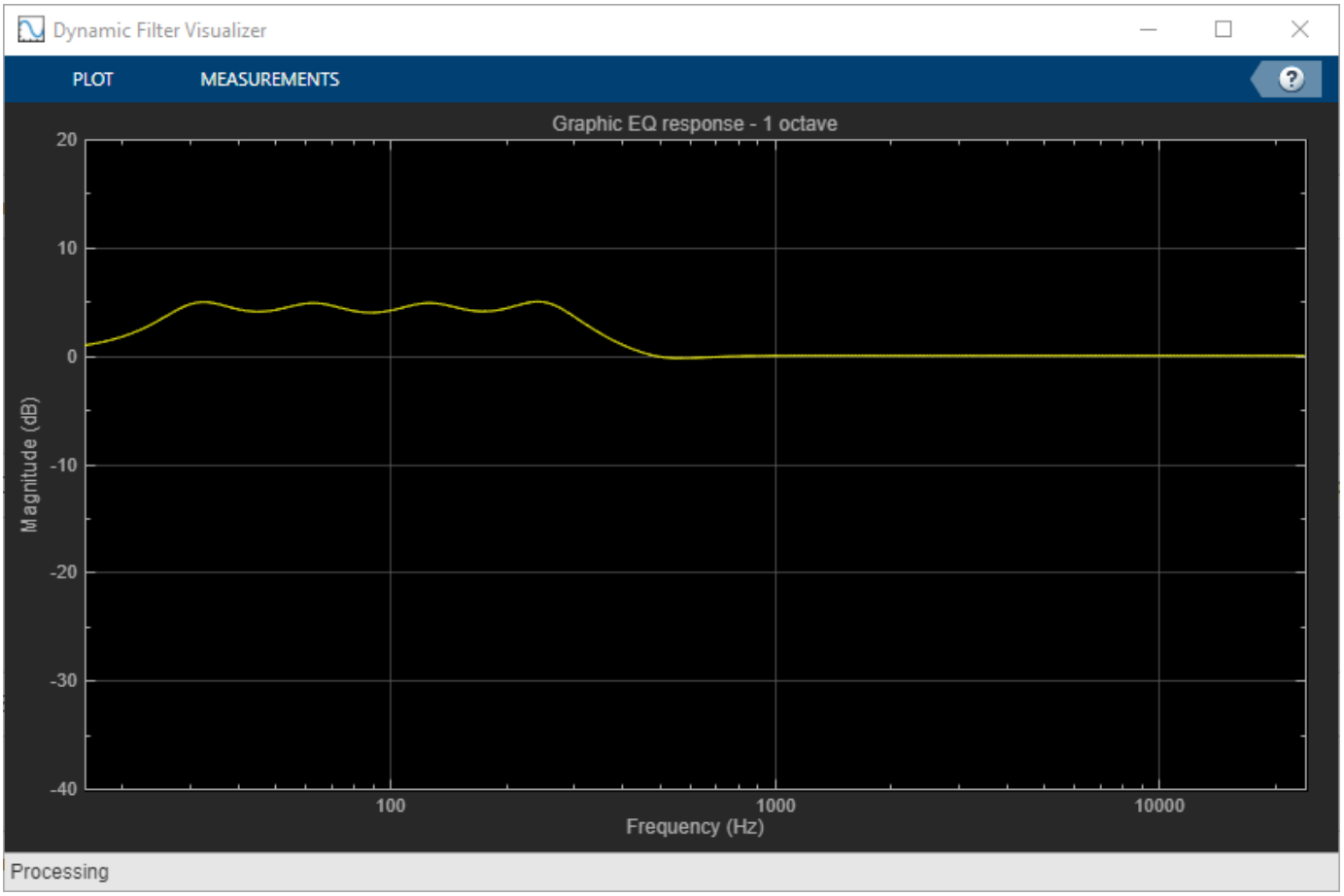
Create a one-octave graphic equalizer implemented with a cascade structure. Use the sample rate of the reader as the sample rate of the equalizer.

```
equalizer = graphicEQ( ...
    'Bandwidth','1 octave', ...
    'Structure','Cascade', ...
    'SampleRate',reader.SampleRate);
```

Specify to increase the gain on low frequencies and then visualize the equalizer.

```
equalizer.Gains = [5,5,5,5,0,0,0,0,0,0];
visualize(equalizer)
```


PLOT
MEASUREMENTS
Graphic EQ response - 1 octave
20
10
0
-10
-20
-30
-40
Magnitude (dB)
10^2
10^3
10^4
Frequency(Hz)



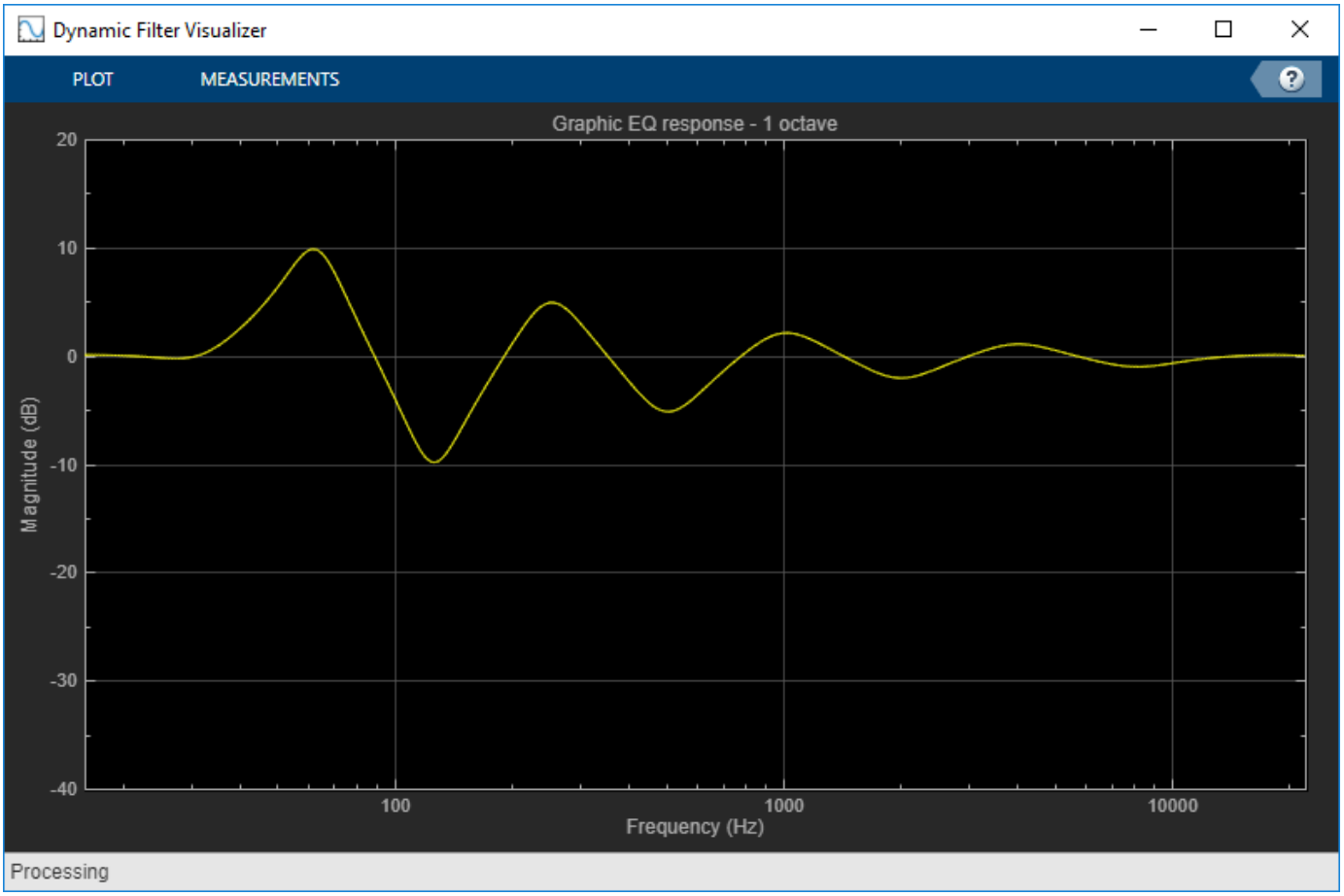
In an audio stream loop, read audio from a file, apply equalization, and then play the equalized audio through your audio device.

```
while ~isDone(reader)
    x = reader();
    y = equalizer(x);
    player(y);
end
release(reader)
release(player)
```

Tune Graphic EQ Parameters

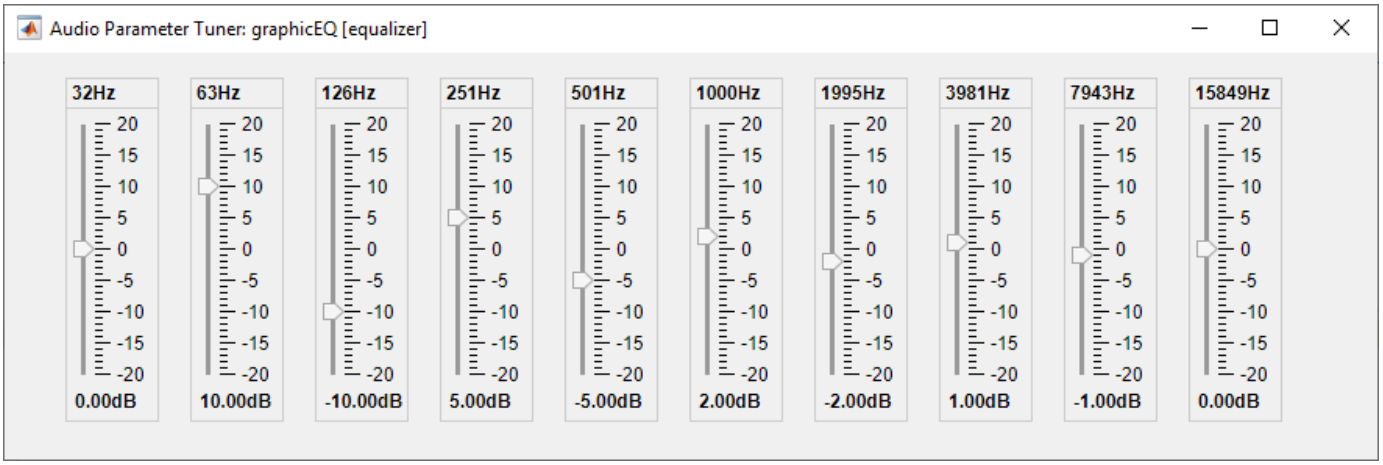
Create a `dsp.AudioFileReader` to read in audio frame-by-frame. Create an `audioDeviceWriter` to write audio to your sound card. Create a `graphicEQ` to process the audio data. Call `visualize` to plot the frequency response of the graphic equalizer.

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

equalizer = graphicEQ('SampleRate',fileReader.SampleRate,'Gains',[0,10,-10,5,-5,2,-2,1,-1,0]);
visualize(equalizer)
```



Call `parameterTuner` to open a UI to tune parameters of the equalizer while streaming.

```
parameterTuner(equalizer)
```



In an audio stream loop:

1. Read in a frame of audio from the file.

2 Apply equalization.

3 Write the frame of audio to your audio device for listening.

While streaming, tune parameters of the equalizer and listen to the effect.

```
while ~isDone(fileReader)
    audioIn = fileReader();
    audioOut = equalizer(audioIn);
    deviceWriter(audioOut);
    drawnow limitrate % required to update parameter
end
```

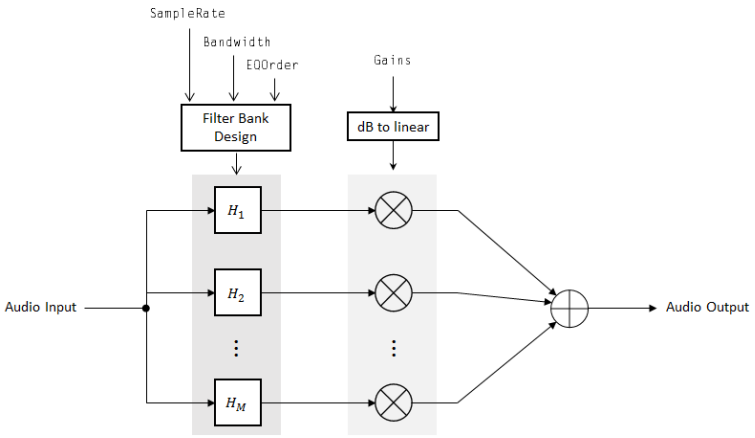
As a best practice, release your objects once done.

```
release(deviceWriter)
release(fileReader)
release(equalizer)
```

Algorithms

The implementation of your graphic equalizer depends on the `Structure` property. See “Graphic Equalization” for a discussion of the pros and cons of the parallel and cascade implementations. Refer to the following sections to understand how these algorithms are implemented in Audio Toolbox.

Parallel Structure



Filter Bank Design

The parallel implementation designs the individual equalizers using the `octaveFilter design method` and spaces them on the spectrum according to the ANSI S1.11-2004 standard.

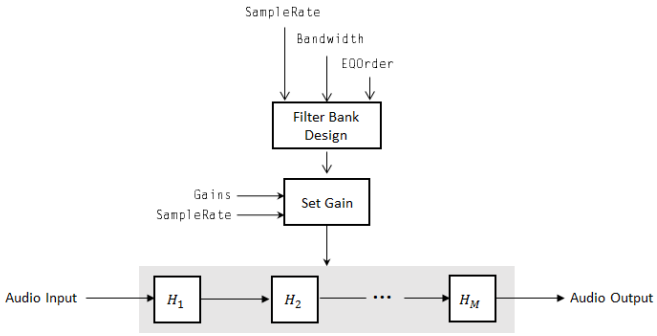
If you set the SampleRate property so that the Nyquist frequency (`SampleRate/2`) is less than the final bandpass edge defined by the ANSI S1.11-2004 standard, then:

- The final bandpass filter is the one whose upper bandpass edge is less than the Nyquist frequency.
- The final filter is implemented as a highpass filter designed by the `designParamEQ` function.

Real-Time Computation

1. The input signal is fed into a filterbank of *M* filters, where *M* depends on the specified `Bandwidth` and `SampleRate` properties.
2. Each branch of the filterbank is multiplied by the linear form of the corresponding element of the `Gains` property.
3. The branches are summed and the output signal is returned.

Cascade Structure



Filter Bank Design

The cascade implementation designs the graphic equalizer filter bank using the `multibandParametricEQ` System object.

Gain Setting

If the EQOrder property is set to 2, then a gain correction is calculated according to [1]. The gain correction is independent of the requested gains. The gain correction is recomputed during the real-time processing only if the `SampleRate` property is modified.

If the `EQOrder` property is not set to 2, no gain correction is applied, and the requested gains are passed on to the `multibandParametricEQ` object.

Real-Time Computation

The input signal is fed into a cascade of *M* biquad filters, where *M* depends on the specified `Bandwidth` and `SampleRate` properties.

Version History

Introduced in R2017b

References

[1] Oliver, Richard J., and Jean-Marc Jot. "Efficient Multi-Band Digital Audio Graphic Equalizer with Accurate Frequency Response Control." Presented at the 139th Convention of the AES, New York, October 2015.

[2] Acoustical Society of America. *American National Standard Specification for Octave-Band and Fractional-Octave-Band Analog and Digital Filters*. ANSI S1.11-2004. Melville, NY: Acoustical Society of America, 2009.

[3] International Organization for Standardization. *Acoustics –– Preferred frequencies*. ISO 266:1997(E). Second Edition. 1997.

Extended Capabilities

C/C++ Code Generation

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

Usage notes and limitations:

“System Objects in MATLAB Code Generation” (MATLAB Coder)

See Also

Blocks

Graphic EQ | Single-Band Parametric EQ | Multiband Parametric EQ

Functions

`multibandParametricEQ` | `designParamEQ` | `designShelvingEQ` | `designVarSlopeFilter`

Topics

“Graphic Equalization”
“Equalization”

info

Get filter information

Syntax

```
infoStruct = info(obj)
```

Description

`infoStruct = info(obj)` returns a structure, `infoStruct`, containing information about `obj`.

Examples

Get Graphic Equalizer Standards-Based Frequencies

Create a `graphicEQ` System object™. Call `info` to return a structure containing standards-based center, edge, and preferred frequencies.

```
equalizer = graphicEQ;
info(equalizer)
```

```
ans = struct with fields:
       CenterFrequencies: [31.6228 63.0957 125.8925 251.1886 501.1872 1000 1.9953e+03 3.9811e+03
         EdgeFrequencies: [22.3872 44.6684 89.1251 177.8279 354.8134 707.9458 1.4125e+03 2.8184e+
    PreferredFrequencies: [31.5000 63 125 250 500 1000 2000 4000 8000 16000]
```

`octaveFilterBank` Info

Create a default `octaveFilterBank`. Call `info` to return a struct containing information about the octave filter bank.

```
octFiltBank = octaveFilterBank;

infoStruct = info(octFiltBank)
```

```
infoStruct = struct with fields:
      CenterFrequencies: [31.6228 63.0957 125.8925 251.1886 501.1872 1000 1.9953e+03 3.9811e+03 7
    BandedgeFrequencies: [22.3872 44.6684 89.1251 177.8279 354.8134 707.9458 1.4125e+03 2.8184e+0
            GroupDelays: [630.0160 315.7551 158.2517 79.3121 39.7471 19.9144 9.9678 4.9661 2.3963
```

`gammatoneFilterBank` Info

Create a default `gammatoneFilterBank`. Call `info` to return a struct containing information about the octave filter bank.

```
gammaFiltBank = gammatoneFilterBank;

infoStruct = info(gammaFiltBank)

infoStruct = struct with fields:
    CenterFrequencies: [50.0000 82.1776 118.0670 158.0966 202.7439 252.5416 308.0839 370.0333 439
           Bandwidths: [30.6688 34.2080 38.1555 42.5583 47.4691 52.9463 59.0554 65.8692 73.4690 8
         GroupDelays: [330.1033 295.5650 264.8688 237.2999 212.5619 190.3751 170.4802 152.6421 1
```

Input Arguments

`obj` — Object to get information from

`graphicEQ` | `gammatoneFilterBank` | `octaveFilterBank`

Object to get information from, specified as an object of `gammatoneFilterBank`, `octaveFilterBank`, or `graphicEQ`.

Output Arguments

`infoStruct` — Struct containing object information

`struct`

Struct containing information about the input `obj`.

Version History

Introduced in R2017b

See Also

`graphicEQ` | `octaveFilterBank` | `gammatoneFilterBank`

visualize

Visualize magnitude response of graphic equalizer

Syntax

```
visualize(equalizer)
visualize(equalizer,NFFT)
hvsz = visualize( ___ )
```

Description

`visualize(equalizer)` plots the magnitude response of the `graphicEQ` object, `equalizer`. The plot is updated automatically when properties of the object change.

`visualize(equalizer,NFFT)` specifies an N-point FFT used to calculate the magnitude response.

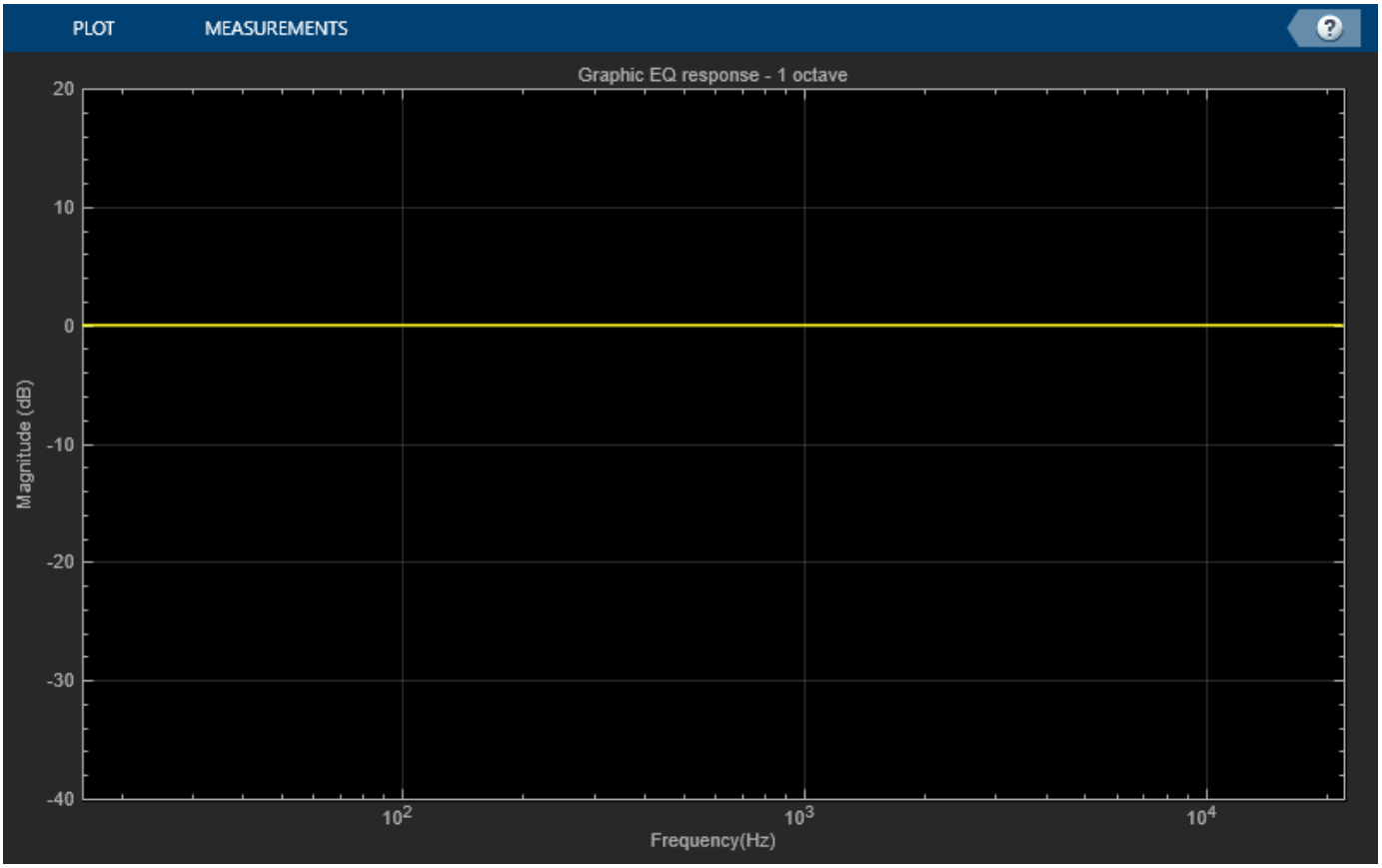
`hvsz = visualize( ___ )` returns a handle to the visualizer as a `dsp.DynamicFilterVisualizer` object when called with any of the previous syntaxes.

Examples

Visualize Magnitude Response of Graphic Equalizer

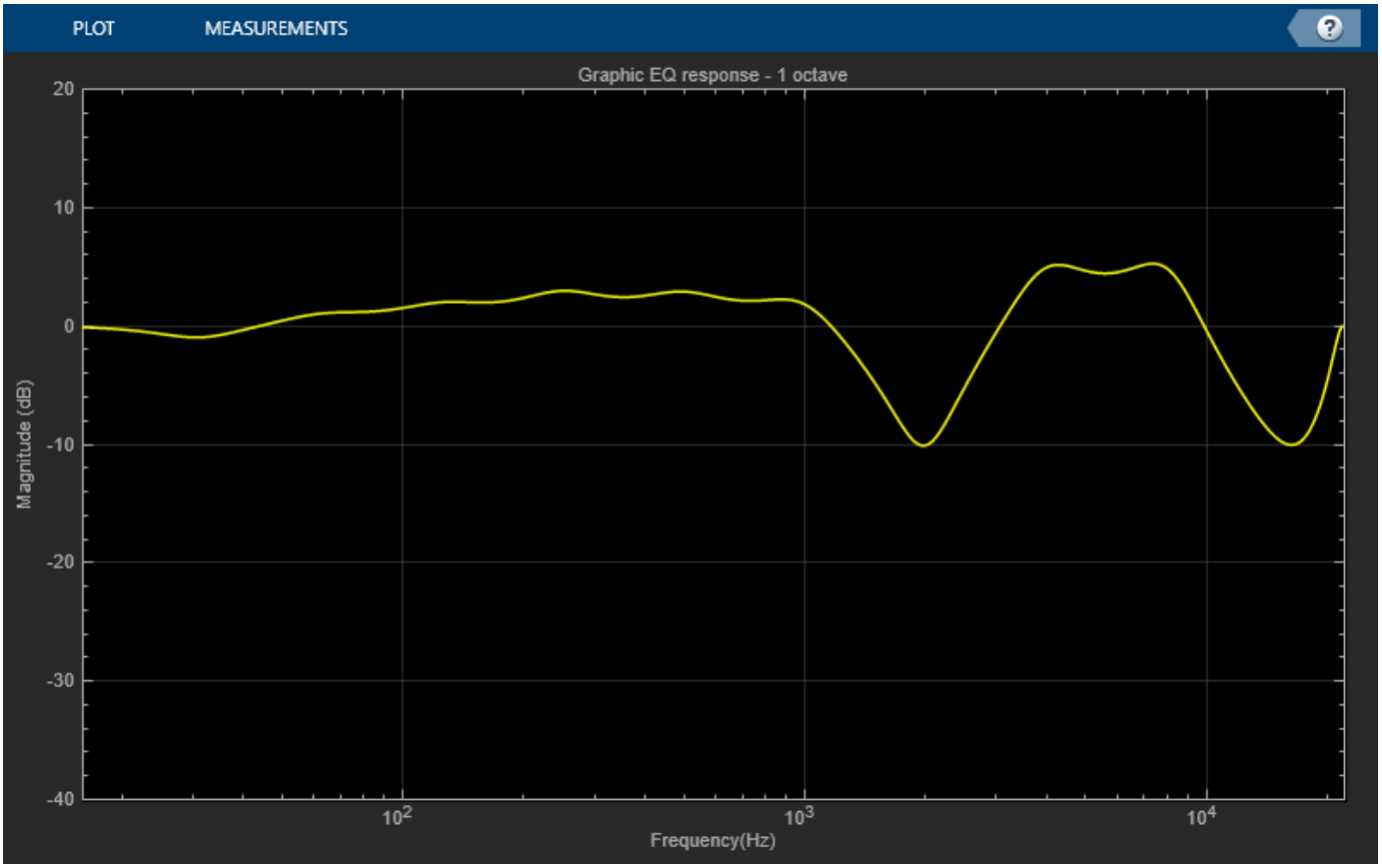
Create a default `graphicEQ` System object™ and then call `visualize`.

```
equalizer = graphicEQ;
visualize(equalizer)
```



Set the gains of the graphic equalizer to new values. The visualization of the magnitude response updates automatically.

```
equalizer.Gains = [-1,1,2,3,3,2,-10,5,5,-10];
```



Input Arguments

`equalizer` — Graphic equalizer to visualize

object of `graphicEQ` System object

Graphic equalizer whose magnitude response you want to plot.

`NFFT` — N-point FFT

2048 (default) | positive scalar

Number of bins used to calculate the DFT, specified as a positive scalar.

Data Types: `single` | `double`

Version History

Introduced in R2017b

See Also

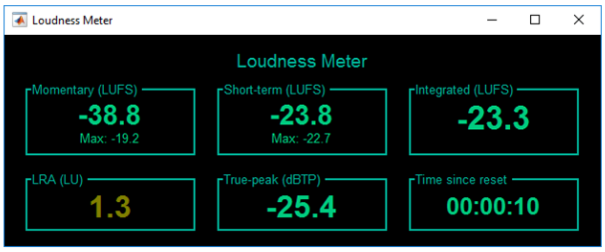
`graphicEQ`

loudnessMeter

Standard-compliant loudness measurements

Description

The `loudnessMeter` System object computes the loudness, loudness range, and true-peak of an audio signal in accordance with EBU R 128 and ITU-R BS.1770-4 standards.



To implement loudness metering:

1. Create the `loudnessMeter` object and set its properties.
2. Call the object with arguments, as if it were a function.

To learn more about how System objects work, see What Are System Objects?

Creation

Syntax

```
loudMtr = loudnessMeter
loudMtr = loudnessMeter(Name,Value)
```

Description

`loudMtr = loudnessMeter` creates a System object, `loudMtr`, that performs loudness metering independently across each input channel.

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

Example: `loudMtr = loudnessMeter('ChannelWeights',[1.2, 0.8],'SampleRate',12000)` creates a System object, `loudMtr`, with channel weights of 1.2 and 0.8, and a sample rate of 12 kHz.

Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see System Design in MATLAB Using System Objects.

`ChannelWeights` — Linear weighting applied to each input channel

`[1, 1, 1, 1.41, 1.41]` (default) | nonnegative row vector

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

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

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

Tunable: Yes

Data Types: `single` | `double`

`UseRelativeScale` — Use relative scale for loudness measurements

`false` (default) | `true`

Use relative scale for loudness measurements, specified as a logical scalar.

- `false` –– The loudness measurements are absolute and returned in loudness units full scale (LUFS).
- `true` –– The loudness measurements are relative to the `TargetLoudness` value and returned in loudness units (LU).

Tunable: No

Data Types: `logical`

`TargetLoudness` — Target loudness level for relative scale (LUFS)

`-23` (default) | real scalar

Target loudness level for relative scale in LUFS, specified as a real scalar.

For example, if the `TargetLoudness` is –23 LUFS, then a loudness value of –23 LUFS is reported as 0 LU.

Tunable: Yes

Dependencies

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

Data Types: `single` | `double`

`SampleRate` — Input sample rate (Hz)

`44100` (default) | positive scalar

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

Tunable: Yes

Data Types: `single` | `double`

Usage

Syntax

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

Description

`[momentary,shortTerm,integrated,range,peak] = loudMtr(audioIn)` returns measurement values for momentary and short-term loudness of the input to your loudness meter, and the true-peak value of the current input frame, `audioIn`. It also returns the integrated loudness and loudness range of the input to your loudness meter since the last time `reset` was called.

Input Arguments

`audioIn` — Audio input to loudness meter

matrix

Audio input to the loudness meter, specified as a matrix. The columns of the matrix are treated as independent audio channels.

Note If you use the default `ChannelWeights` of the `loudnessMeter`, as a best practice, specify the input channels in this order: [Left, Right, Center, Left surround, Right surround].

Data Types: `single` | `double`

Output Arguments

`momentary` — Momentary loudness (LUFS)

column vector

Momentary loudness in loudness units relative to full scale (LUFS), returned as a column vector with the same number of rows as `audioIn`.

By default, loudness measurements are returned in LUFS. If you set the `UseRelativeScale` property to `true`, loudness measurements are returned in loudness units (LU).

Data Types: `single` | `double`

`shortTerm` — Short-term loudness (LUFS)

column vector

Short-term loudness in loudness units relative to full scale (LUFS), returned as a column vector with the same number of rows as `audioIn`.

By default, loudness measurements are returned in LUFS. If you set the `UseRelativeScale` property to `true`, loudness measurements are returned in loudness units (LU).

Data Types: `single` | `double`

`integrated` — Integrated loudness (LUFS)
column vector

Integrated loudness in loudness units relative to full scale (LUFS), returned as a column vector with the same number of rows as `audioIn`.

By default, loudness measurements are returned in LUFS. If you set the `UseRelativeScale` property to `true`, loudness measurements are returned in loudness units (LU).

Data Types: `single` | `double`

`range` — Loudness range (LU)
column vector

Loudness range in loudness units (LU), returned as a column vector with the same number of rows as `audioIn`.

Data Types: `single` | `double`

peak — True-peak loudness (dB-TP)
scalar

True-peak loudness in dB-TP, returned as a column vector with the same number of rows as `audioIn`.

Data Types: `single` | `double`

Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

Specific to `loudnessMeter`

visualize	Open 'EBU Mode' meter display

Common to All System Objects

clone	Create duplicate System object
isLocked	Determine if System object is in use
release	Release resources and allow changes to System object property values and input characteristics
reset	Reset internal states of System object
step	Run System object algorithm

Examples

Loudness of Audio Signal

Create a `dsp.AudioFileReader` System object™ to read in an audio file. Create a `loudnesMeter` System object. Use the sample rate of the audio file as the sample rate of the `loudnessMeter`.

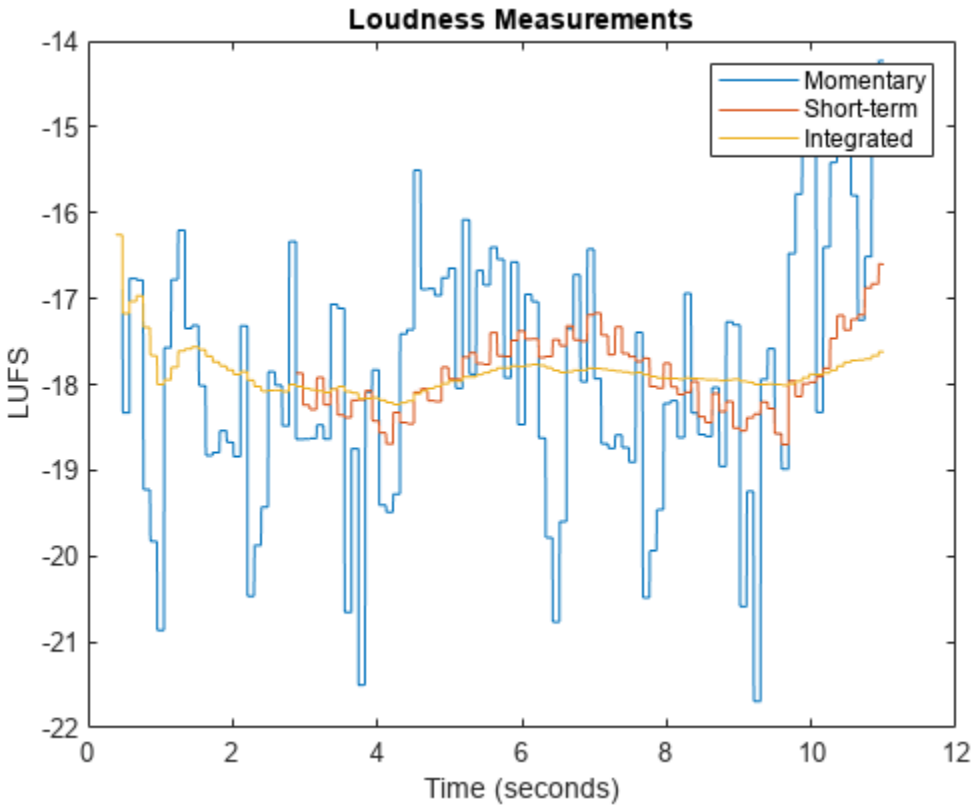
```
fileReader = dsp.AudioFileReader('RockDrums-44p1-stereo-11secs.mp3');
loudMtr = loudnessMeter('SampleRate',fileReader.SampleRate);
```

Read in the audio file in an audio stream loop. Use the loudness meter to determine the momentary, short-term, and integrated loudness of the audio signal. Cache the loudness measurements for analysis.

```
momentary = [];
shortTerm = [];
integrated = [];

while ~isDone(fileReader)
    x = fileReader();
    [m,s,i] = loudMtr(x);
    momentary = [momentary;m];
    shortTerm = [shortTerm;s];
    integrated = [integrated;i];
end

release(fileReader)
```

Plot the momentary, short-term, and integrated loudness of the audio signal.

```
t = linspace(0,11,length(momentary));
plot(t,[momentary,shortTerm,integrated])
title('Loudness Measurements')
legend('Momentary','Short-term','Integrated')
xlabel('Time (seconds)')
ylabel('LUFS')
```



Plot Momentary Loudness and Loudness Range of Audio Stream

Create an audio file reader and an audio device writer.

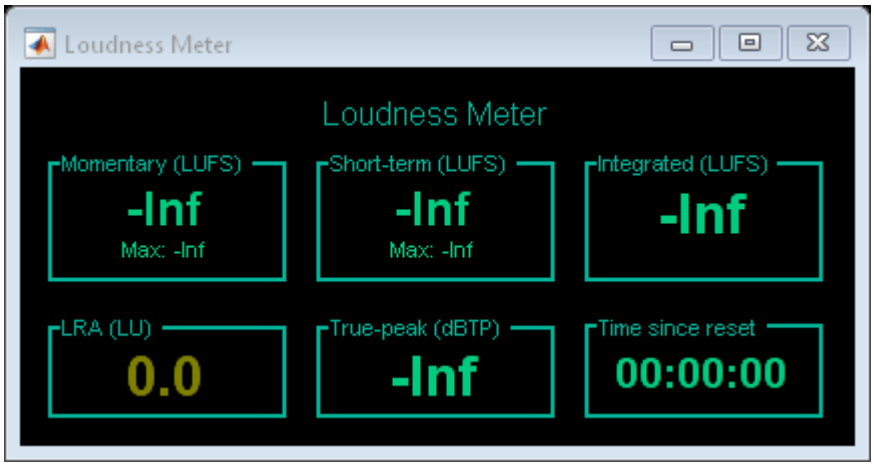
```
fileReader = dsp.AudioFileReader('FunkyDrums-44p1-stereo-25secs.mp3', ...
    'SamplesPerFrame',1024);
fs = fileReader.SampleRate;
deviceWriter = audioDeviceWriter('SampleRate',fs);
```

Create a time scope to visualize your audio stream loop.

```
timeScope = timescope('NumInputPorts',2, ...
    'SampleRate',fs, ...
    'TimeSpanOverrunAction','Scroll', ...
    'LayoutDimensions',[2,1], ...
    'TimeSpanSource','Property','TimeSpan',5, ...
    'BufferLength',5*fs);

% Top subplot of scope
timeScope.Title = 'Momentary Loudness';
timeScope.YLabel = 'LUFS';
timeScope.YLimits = [-40, 0];

% Bottom subplot of scope
timeScope.ActiveDisplay = 2;
timeScope.Title = 'Loudness Range';
timeScope.YLabel = 'LU';
timeScope.YLimits = [-1, 2];
```

Create a loudness meter. Use the sample rate of your input file as the sample rate of your loudness meter. Call `visualize` to open an 'EBU-mode' visualization for your loudness meter.

```
loudMtr = loudnessMeter('SampleRate',fs);
visualize(loudMtr)
```



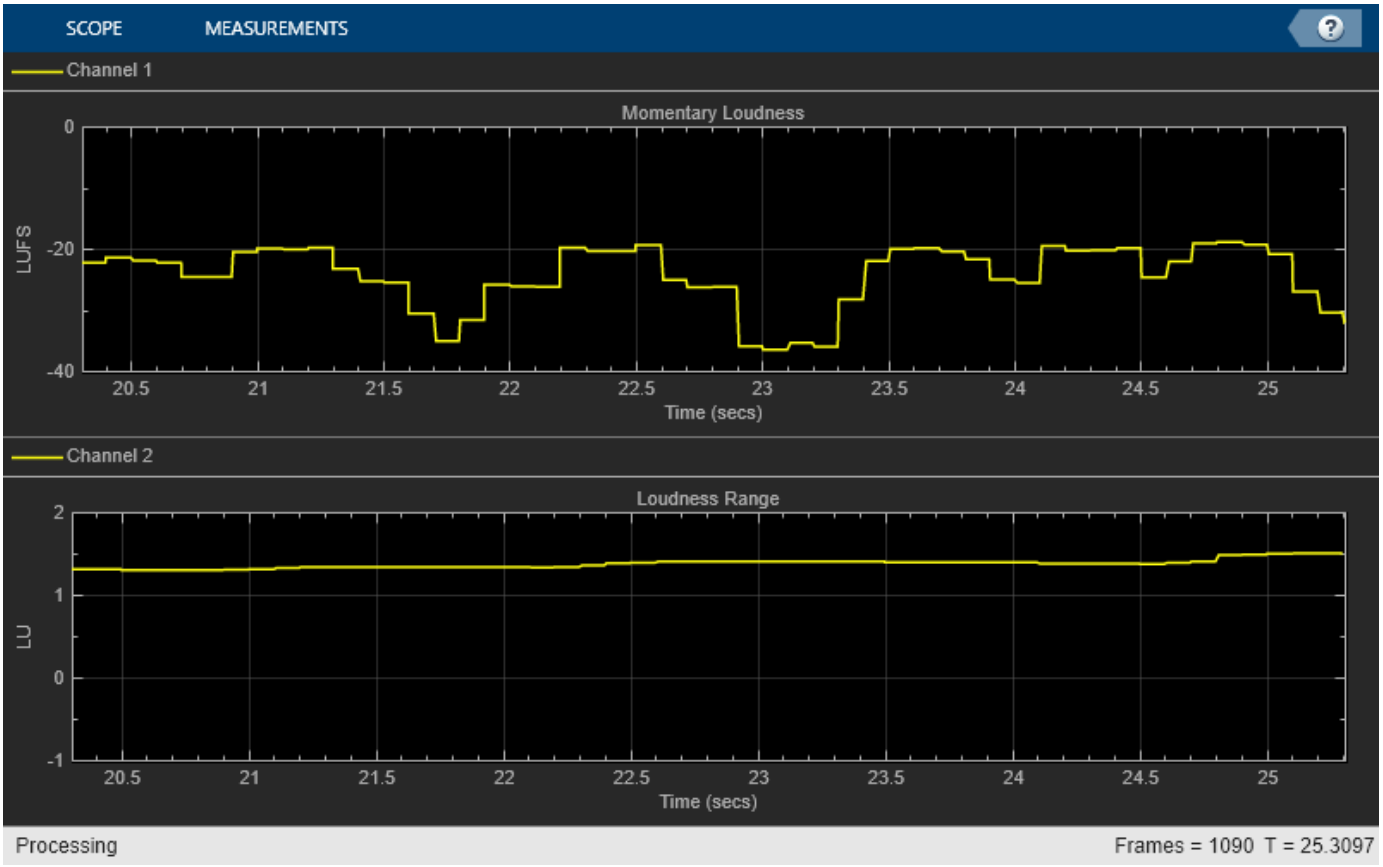
In an audio stream loop:

- Read in your audio file.

- Compute the momentary loudness and loudness range.
- Visualize the momentary loudness and loudness range on your time scope.
- Play the audio signal.

The 'EBU-mode' loudness meter visualization updates automatically while it is open. As a best practice, release your file reader and device writer once the loop is completed.

```
while ~isDone(fileReader)
    audioIn = fileReader();
    [momentaryLoudness,~,~,LRA] = loudMtr(audioIn);
    timeScope(momentaryLoudness,LRA);
    deviceWriter(audioIn);
end

release(fileReader)
release(deviceWriter)
```



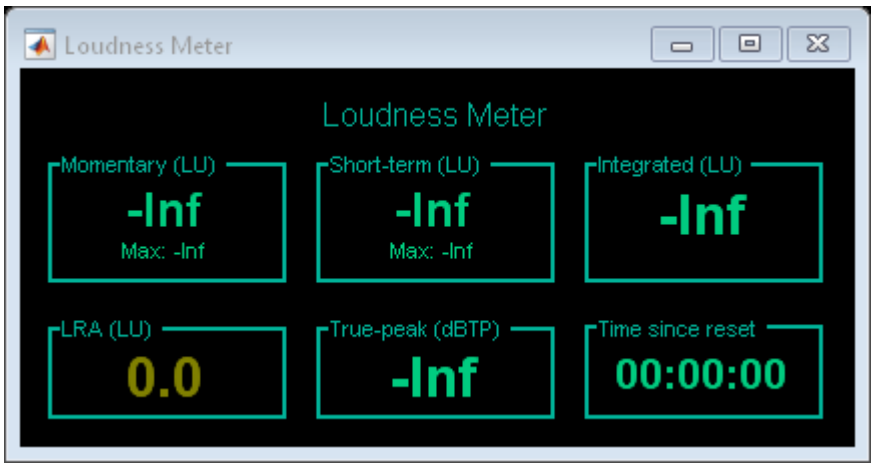
Relative Scale for Loudness Measurements

Create an audio file reader to read in an audio file. Create an audio device writer to write the audio file to your audio device. Use the sample rate of your file reader as the sample rate of your device writer.

```
fileReader = dsp.AudioFileReader('Counting-16-44p1-mono-15secs.wav',...
    'SamplesPerFrame',1024);
fs = fileReader.SampleRate;
deviceWriter = audioDeviceWriter('SampleRate',fs);
```

Create a loudness meter with the target loudness set to the default `-23` LUFS. Open the 'EBU-mode' loudness meter visualization.

```
loudMtr = loudnessMeter('UseRelativeScale',true);
visualize(loudMtr)
```



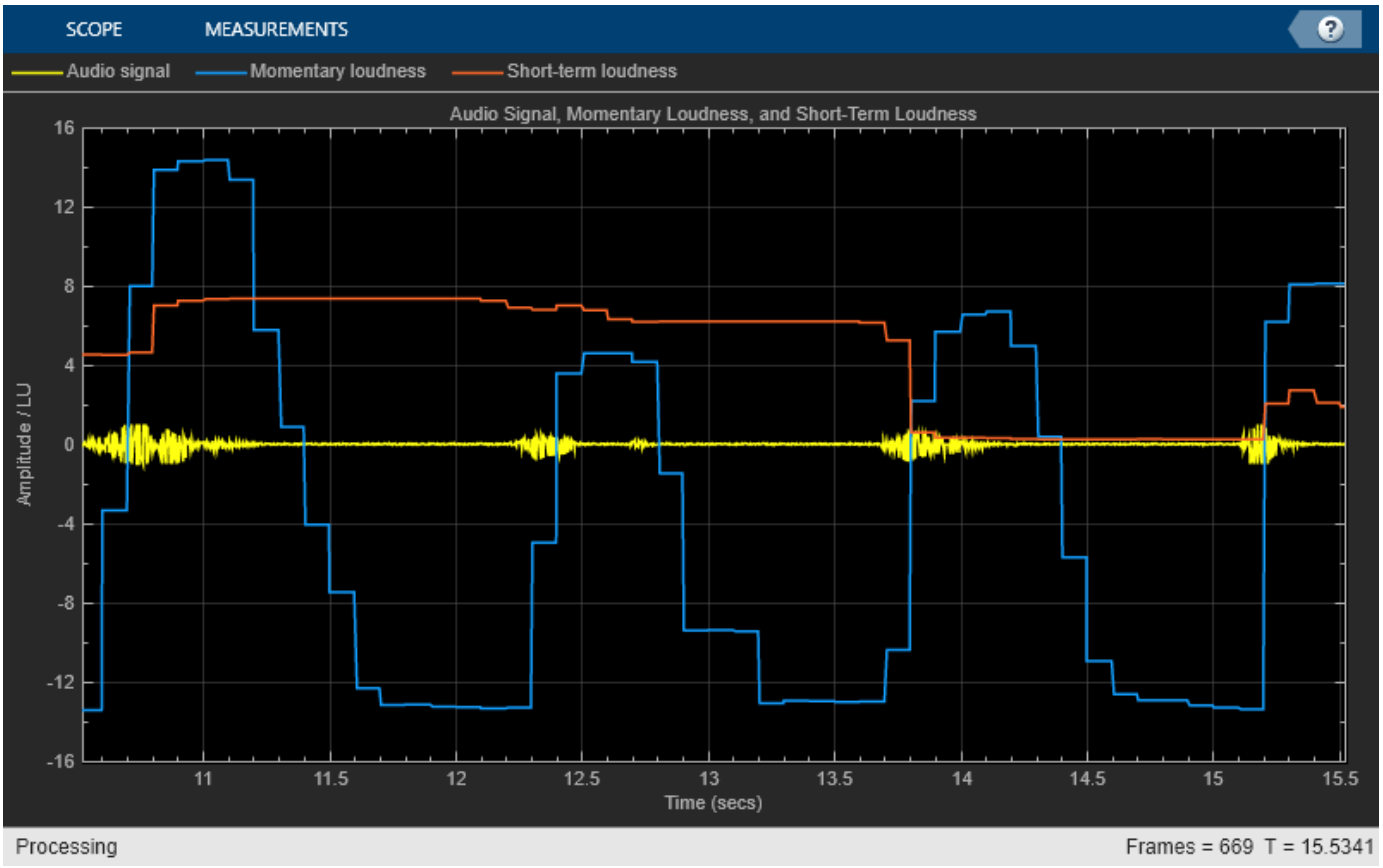
Create a time scope to visualize your audio signal and its measured relative momentary and short-term loudness.

```
scope = timescope( ...
    'NumInputPorts',3, ...
```

```
    'SampleRate',fs, ...
    'TimeSpanOverrunAction','Scroll', ...
    'TimeSpanSource','Property','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);
```

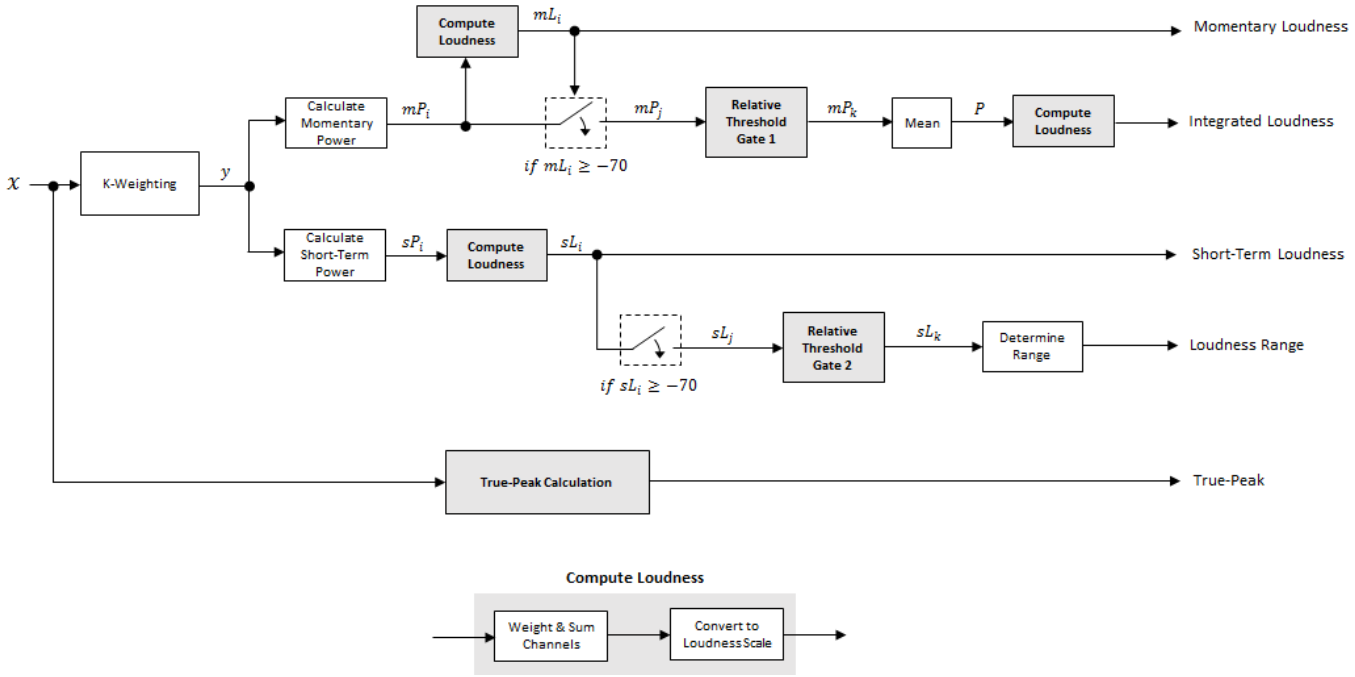
In an audio stream loop, listen to and visualize the audio signal.

```
while ~isDone(fileReader)
    x = fileReader();
    [momentary,shortTerm] = loudMtr(x);
    scope(x,momentary,shortTerm)
    deviceWriter(x);
end

release(deviceWriter)
release(fileReader)
```



Algorithms

The `loudnessMeter` System object calculates the momentary loudness, short-term loudness, integrated loudness, loudness range (LRA), and true-peak value of an audio signal. You can specify any number of channels and nondefault channel weights used for loudness measurements. The `loudnessMeter` algorithm is described for the general case of n channels with default channel weights.



Loudness Measurements

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

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` System object returns the last computed values for momentary and integrated loudness. If enough samples have been collected, then the power (mean square) of each segment of the K-weighted channels is calculated:

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

   - mP_i is the momentary power of the ith segment.
   - w is the segment length in samples.

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

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

   - G_c is the weighting for channel c.

   mL is the momentary loudness returned by your `loudnessMeter` System object. It is also used internally to calculate the integrated loudness (steps 3–6).

3. The *integrated loudness* measurement considers the audio signal since the last reset of your loudness meter. To calculate integrated loudness, the momentary power is passed through a gating system. The gate system pauses the measurement during periods of low sound, such as stretches of silence in a movie.

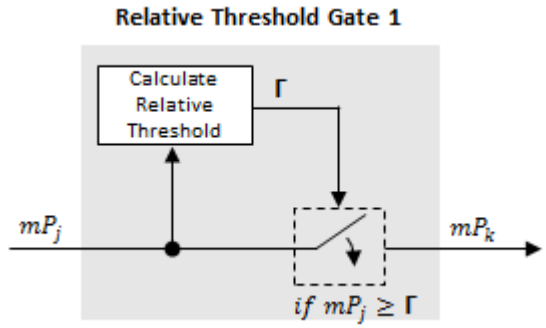
   The momentary power segment is gated using the corresponding momentary loudness segment calculation:

$$mP_i \rightarrow mP_j$$

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

   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:

$$mP_j \rightarrow mP_k$$

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

The relative threshold is recomputed during each call to your `loudnessMeter` object. The cached values of mP_j are gated again depending on the updated value of Γ.

5 The momentary power segments are averaged:

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

6 The integrated loudness is computed in LUFS by passing the mean momentary power, P, through the Compute Loudness system:

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

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` System object returns the last computed values for short-term loudness and loudness range. If enough samples have been collected, then the power (mean square) of each K-weighted channel is calculated:

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

- sP_i is the short-term power of the ith segment of a channel.
- w is the segment length in samples.

2 The short-term loudness, sL, is computed in LUFS for each segment:

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

- G_c is the weighting for channel c.

sL is the short-term loudness returned by your `loudnessMeter` System object. It is also used internally to calculate the loudness range (steps 3–5).

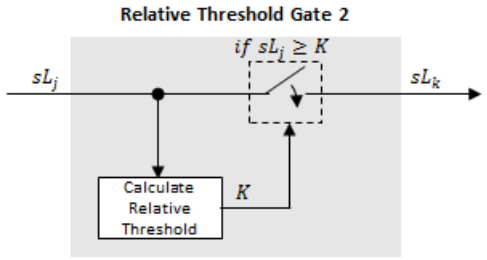
3 The short-term loudness is gated using an absolute threshold:

$$sL_i \rightarrow sL_j$$

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

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:

$$sL_j \rightarrow sL_k$$

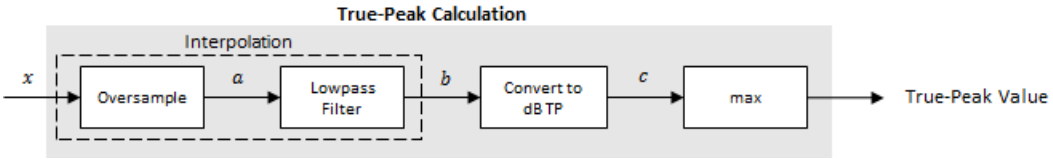
$$k = \left\{ j \,|\, sL_j \geq K \right\}$$

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. The algorithm does not consider an input sample rate below 750 Hz.

Input Sample Rate (kHz)	Upsample Factor
[0.75, 1.5)	256
[1.5, 3)	128
[3, 6)	64
[6,12)	32

Input Sample Rate (kHz)	Upsample Factor
[12, 24)	16
[24, 48)	8
[48, 96)	4
[96,192)	2
[192, ∞)	Not required

2. The oversampled signal *a* passes through a lowpass filter with a half-polyphase length of 12 and stopband attenuation of 80 dB. The filter design uses `designMultirateFIR`.
3. The filtered signal *b* is rectified and converted to the dB TP scale:

   $c = 20 \times \log_{10}(|b|)$
4. The true-peak is determined as the maximum of the converted signal *c*.

Version History

Introduced in R2016b

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

Blocks

Loudness Meter

Functions
`octaveFilter` | `weightingFilter` | `integratedLoudness`

visualize

Open 'EBU Mode' meter display

Syntax

```
visualize(loudMtr)
hvsz = 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`.

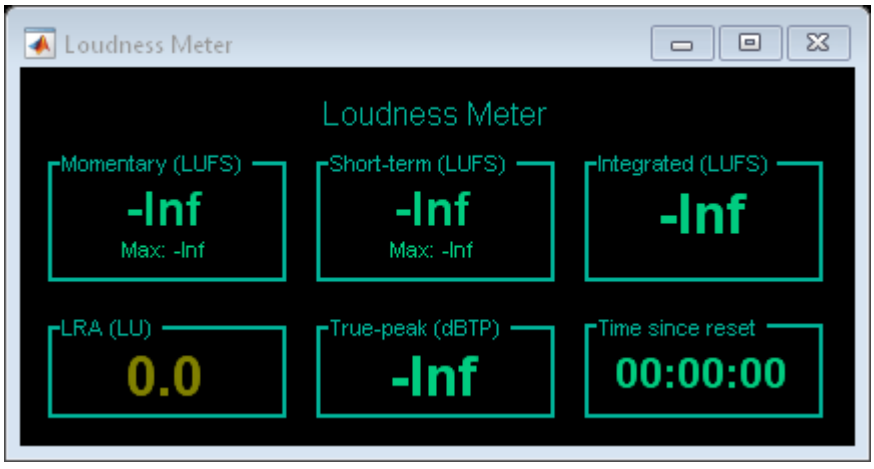
`hvsz = visualize(loudMtr)` returns a handle to the display.

Examples

Open an 'EBU Mode' Loudness Meter Display

Create a `loudnessMeter` System object™, and then call `visualize` to open an 'EBU Mode' loudness meter display.

```
loudMtr = loudnessMeter;
visualize(loudMtr)
```

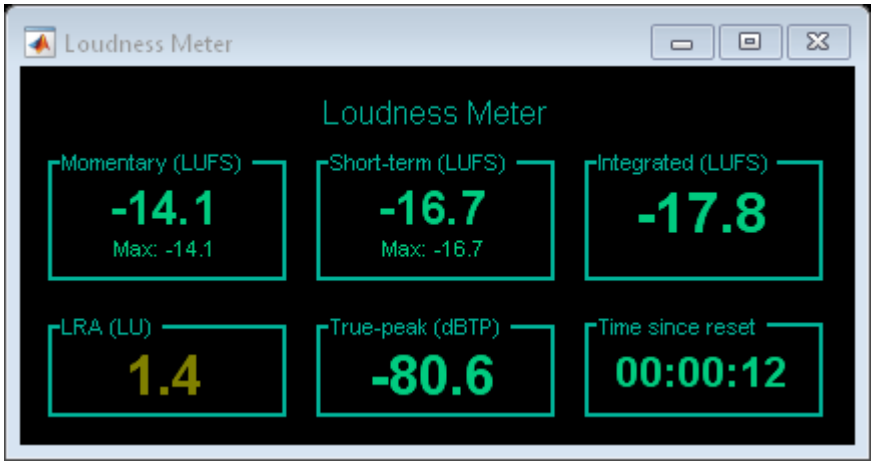


Create an audio file reader System object and specify the audio file to analyze. Create an audio device writer System object to play the audio to your output device.

```
fileReader = dsp.AudioFileReader('RockDrums-48-stereo-11secs.mp3');
deviceWriter = audioDeviceWriter('SampleRate',fileReader.SampleRate);
```

In an audio stream loop, read the audio from the file and play it to your device. The loudness meter visualization updates at each call.

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



Input Arguments

`loudMtr` — Object of `loudnessMeter`
object

Object of the `loudnessMeter` System object.

Version History

Introduced in R2016b

See Also

Blocks
Loudness Meter

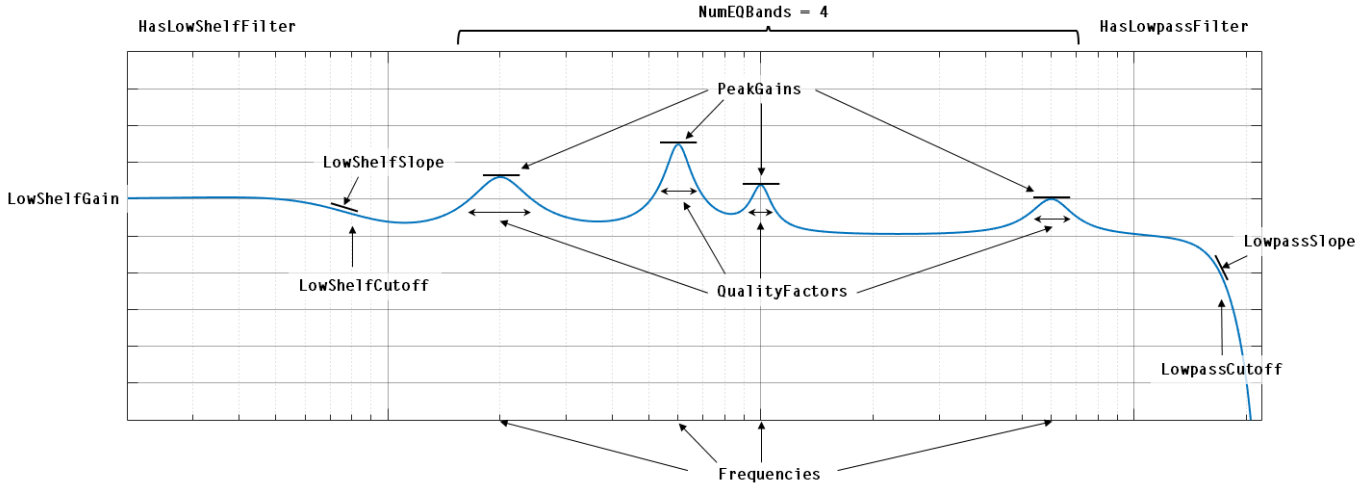
Functions
`integratedLoudness`

multibandParametricEQ

Multiband parametric equalizer

Description

The `multibandParametricEQ` System object performs multiband parametric equalization independently across each channel of input using specified center frequencies, gains, and quality factors. You can configure the System object with up to 10 bands. You can add low-shelf and high-shelf filters, as well as highpass (low-cut) and lowpass (high-cut) filters.



To implement a multiband parametric equalizer:

1. Create the `multibandParametricEQ` object and set its properties.
2. Call the object with arguments, as if it were a function.

To learn more about how System objects work, see What Are System Objects?

Creation

Syntax

```
mPEQ = multibandParametricEQ
mPEQ = multibandParametricEQ(Name,Value)
```

Description

`mPEQ = multibandParametricEQ` creates a System object, `mPEQ`, that performs multiband parametric equalization.

`mPEQ = multibandParametricEQ(Name,Value)` sets each construction argument or property `Name` to the specified `Value`. Unspecified properties and creation arguments have default values.

Example: `mPEQ = multibandParametricEQ('NumEQBands',3,'Frequencies', [300,1200,5000])` creates a multiband parametric equalizer System object, `mPEQ`, with `NumEQBands` set to 3 and the `Frequencies` property set to `[300,1200,5000]`.

Note The value specified by NumEQBands must be the length of the row vectors specified by Frequencies, QualityFactors, and PeakGains. During creation of the System object, the first property you specify locks the value.

Creation Arguments

Creation arguments are properties which are set during creation of the System object and cannot be modified later. If you do not explicitly set a creation argument value, the property takes a default value.

`NumEQBands` — Number of equalizer bands

3 (default) | integer in the range [1, 10]

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

`NumEQBands` is set during creation of the System object and cannot be modified later. If you do not explicitly set its value, the property takes the default value.

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

Data Types: `single` | `double`

`EQOrder` — Order of individual equalizer bands

2 (default) | even integer

Order of individual equalizer bands, specified as an even integer. All equalizer bands have the same order.

`EQOrder` is set during creation of the System object and cannot be modified later. If you do not explicitly set its value, the property takes the default value.

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

Data Types: `single` | `double`

`HasLowShelfFilter` — Low-shelf filter toggle

`false` (default) | `true`

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

- `false` –- Do not enable low-shelf filtering in multiband parametric equalizer implementation.
- `true` –- Enable low-shelf filtering in multiband parametric equalizer implementation.

`HasLowShelfFilter` is set during creation of the System object and cannot be modified later. If you do not explicitly set its value, the property takes the default value.

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

Data Types: `logical`

`HasHighShelfFilter` — High-shelf filter toggle
`false` (default) | `true`

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

- `false` –– Do not enable high-shelf filtering in multiband parametric equalizer implementation.
- `true` –– Enable high-shelf filtering in multiband parametric equalizer implementation.

`HasHighShelfFilter` is set during creation of the System object and cannot be modified later. If you do not explicitly set its value, the property takes the default value.

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

Data Types: `logical`

`HasLowpassFilter` — Lowpass filter toggle
`false` (default) | `true`

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

- `false` –– Do not enable lowpass filtering in multiband parametric equalizer implementation.
- `true` –– Enable lowpass filtering in multiband parametric equalizer implementation.

`HasLowpassFilter` is set during creation of the System object and cannot be modified later. If you do not explicitly set its value, the property takes the default value.

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

Data Types: `logical`

`HasHighpassFilter` — Highpass filter toggle
`false` (default) | `true`

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

- `false` –– Do not enable highpass filtering in multiband parametric equalizer implementation.
- `true` –– Enable highpass filtering in multiband parametric equalizer implementation.

`HasHighpassFilter` is set during creation of the System object and cannot be modified later. If you do not explicitly set its value, the property takes the default value.

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

Data Types: `logical`

`Oversample` — Oversample toggle
`false` (default) | `true`

Oversample toggle, specified as `false` or `true`.

- `false` –– Runs the multiband parametric equalizer at the input sample rate.

- `true` –– Runs the multiband parametric equalizer at two times the input sample rate. Oversampling minimizes the frequency-warping effects introduced by the bilinear transformation.

  A halfband interpolator implements oversampling before equalization. A halfband decimator reduces the sample rate back to the input sampling rate after equalization.

`Oversample` is set during creation of the System object and cannot be modified later. If you do not explicitly set its value, the property takes the default value.

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

Data Types: `logical`

Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see System Design in MATLAB Using System Objects.

Multiband Equalizer

`Frequencies` — Center frequencies of equalizer bands (Hz)

`[100, 181, 325]` (default) | row vector of length `NumEQBands`

Center frequencies of equalizer bands in Hz, specified as a row vector of length `NumEQBands`. The vector consists of real scalars in the range 0 to SampleRate/2.

Tunable: Yes

Data Types: `single` | `double`

`QualityFactors` — Quality factors of equalizer bands

`[1.6,1.6,1.6]` (default) | row vector of length `NumEQBands`

Quality factors of equalizer bands, specified as a row vector of length NumEQBands.

Tunable: Yes

Data Types: `single` | `double`

`PeakGains` — Peak or dip filter gains (dB)

`[0,0,0]` (default) | row vector of length `NumEQBands`

Peak or dip filter gains in dB, specified as a row vector of length NumEQBands. The vector consists of real scalars in the range [–inf, 20].

Tunable: Yes

Data Types: `single` | `double`

Low-Shelf Filter

`LowShelfCutoff` — Low-shelf filter cutoff (Hz)

`200` (default) | scalar

Low-shelf filter cutoff in Hz, specified as a scalar greater than or equal to 0.

Tunable: Yes

Dependencies

To enable this property, set HasLowShelfFilter to `true` during creation.

Data Types: `single` | `double`

`LowShelfSlope` — Low-shelf filter slope coefficient

`1.5` (default) | positive scalar

Low-shelf filter slope coefficient, specified as a positive scalar.

Tunable: Yes

Dependencies

To enable this property, set HasLowShelfFilter to `true` during creation.

Data Types: `single` | `double`

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

`0` (default) | real scalar

Low-shelf filter gain in dB, specified as a real scalar.

Tunable: Yes

Dependencies

To enable this property, set HasLowShelfFilter to `true` during creation.

Data Types: `single` | `double`

High-Shelf Filter

`HighShelfCutoff` — High-shelf filter cutoff (Hz)

`15000` (default) | nonnegative real scalar

High-shelf filter cutoff in Hz, specified as a real scalar greater than or equal to 0.

Tunable: Yes

Dependencies

To enable this property, set HasHighShelfFilter to `true` during creation.

Data Types: `single` | `double`

`HighShelfSlope` — High-shelf slope coefficient

`1.5` (default) | positive scalar

High-shelf filter slope coefficient, specified as a positive scalar.

Tunable: Yes

Dependencies

To enable this property, set HasHighShelfFilter to `true` during creation.

Data Types: `single` | `double`

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

`0` (default) | real scalar

High-shelf filter gain in dB, specified as a real scalar.

Tunable: Yes

Dependencies

To enable this property, set HasHighShelfFilter to `true` during creation.

Data Types: `single` | `double`

Lowpass Filter

`LowpassCutoff` — Lowpass filter cutoff frequency (Hz)

`18000` (default) | nonnegative real scalar

Lowpass filter cutoff frequency in Hz, specified as a real scalar greater than or equal to 0.

Tunable: Yes

Dependencies

To enable this property, set HasLowpassFilter to `true` during creation.

Data Types: `single` | `double`

`LowpassSlope` — Lowpass filter slope (dB/octave)

`12` (default) | real scalar in the range `[0:6:48]`

Lowpass filter slope in dB/octave, specified as a real scalar in the range `[0:6:48]`. Values that are not multiples of 6 are rounded to the nearest multiple of 6.

Tunable: Yes

Dependencies

To enable this property, set HasLowpassFilter to `true` during creation.

Data Types: `single` | `double`

Highpass Filter

`HighpassCutoff` — Highpass filter cutoff frequency (Hz)

`20` (default) | nonnegative real scalar

Highpass filter cutoff in Hz, specified as a real scalar greater than or equal to 0.

Tunable: Yes

Dependencies

To enable this property, set HasHighpassFilter to `true` during creation.

Data Types: `single` | `double`

`HighpassSlope` — Highpass filter slope (dB/octave)

`30` (default) | real scalar in the range `[0:6:48]`

Highpass filter slope in dB/octave, specified as a real scalar in the range `[0:6:48]`. Values that are not multiples of 6 are rounded to the nearest multiple of 6.

Tunable: Yes

Dependencies

To enable this property, set HasHighpassFilter to `true` during creation.

Data Types: `single` | `double`

Sampling

`SampleRate` — Input sample rate (Hz)

`44100` (default) | positive scalar

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

Tunable: Yes

Data Types: `single` | `double`

Usage

Syntax

```
audioOut = mPEQ(audioIn)
```

Description

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

Input Arguments

`audioIn` — Audio input to equalizer

matrix

Audio input to the equalizer, specified as a matrix. The columns of the matrix are treated as independent audio channels.

Data Types: `single` | `double`

Output Arguments

`audioOut` — Audio output from equalizer

matrix

Audio output from the equalizer, returned as a matrix the same size as `audioIn`.

Data Types: `single` | `double`

Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

Specific to `multibandParametricEQ`

createAudioPluginClass	Create audio plugin class that implements functionality of System object
visualize	Visualize magnitude response of multiband parametric equalizer
parameterTuner	Tune object parameters while streaming

MIDI

configureMIDI	Configure MIDI connections between audio object and MIDI controller
disconnectMIDI	Disconnect MIDI controls from audio object
getMIDIConnections	Get MIDI connections of audio object

Common to All System Objects

clone	Create duplicate System object
isLocked	Determine if System object is in use
release	Release resources and allow changes to System object property values and input characteristics
reset	Reset internal states of System object
step	Run System object algorithm

The `createAudioPluginClass` and `configureMIDI` functions map tunable properties of the `multibandParametricEQ` System object to user-facing parameters:

Property	Range	Mapping	Unit
`Frequencies`	[20, 20000]	log	Hz
`QualityFactors`	[0.2, 700]	linear	none
`PeakGains`	[–50, 20]	linear	dB
`LowShelfCutoff`	[20, 20000]	log	Hz
`LowShelfSlope`	[0.1, 5]	linear	none
`LowShelfGain`	[–12, 12]	linear	dB
`HighShelfCutoff`	[20, 20000]	log	Hz
`HighShelfSlope`	[0.1, 5]	linear	none
`HighShelfGain`	[–12, 12]	linear	dB
`LowpassCutoff`	[20, 20000]	log	Hz
`LowpassSlope`	[0, 48]	linear	dB/octave
`HighpassCutoff`	[20, 20000]	log	Hz
`HighpassSlope`	[0, 48]	linear	dB/octave

Examples

Multiband Parametric Equalization

Create `dsp.AudioFileReader` and `audioDeviceWriter` 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.

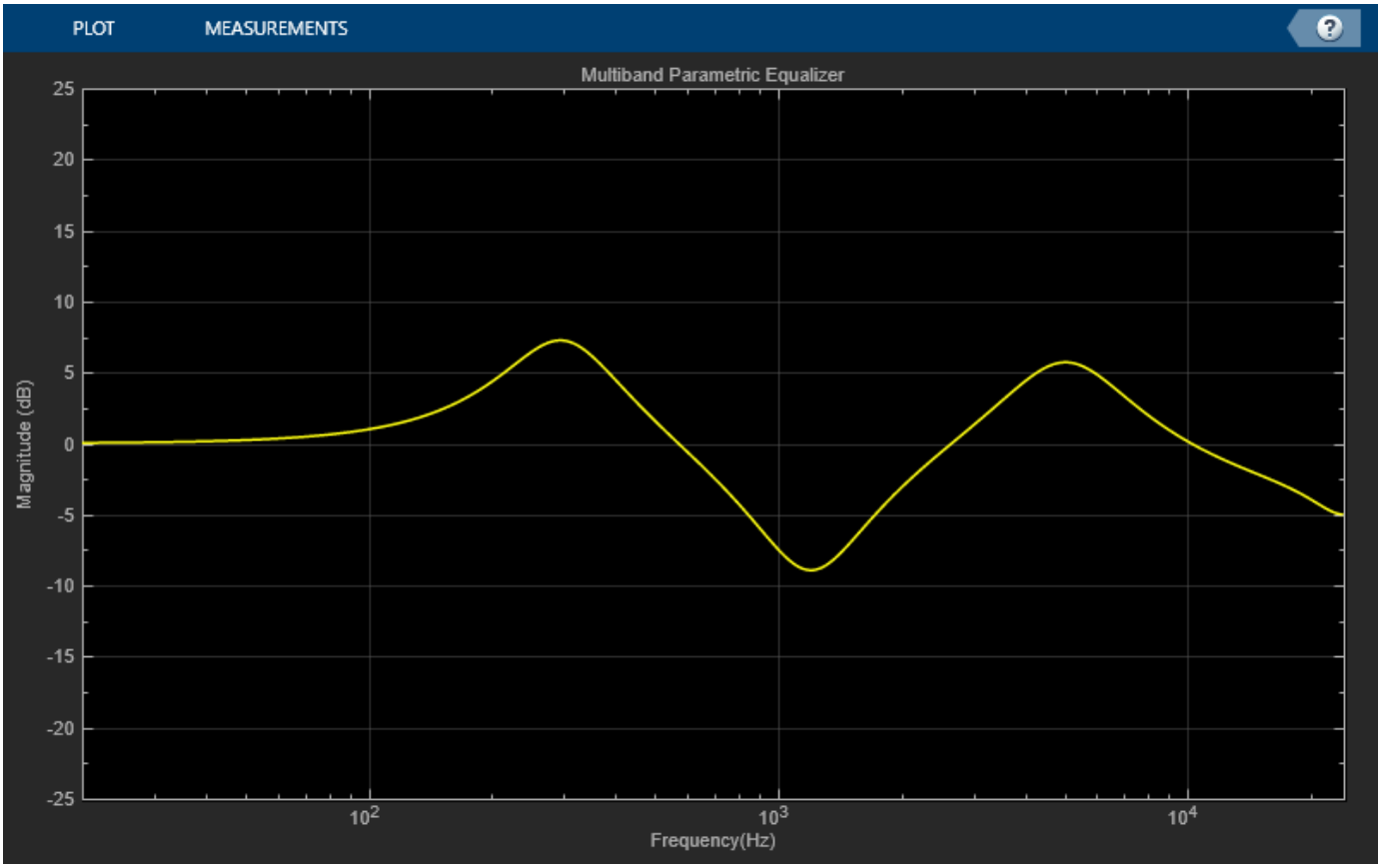
```
frameLength = 512;

fileReader = dsp.AudioFileReader( ...
    'Filename','RockDrums-48-stereo-11secs.mp3', ...
    'SamplesPerFrame',frameLength);
deviceWriter = audioDeviceWriter( ...
    'SampleRate',fileReader.SampleRate);

setup(deviceWriter,ones(frameLength,2))
```

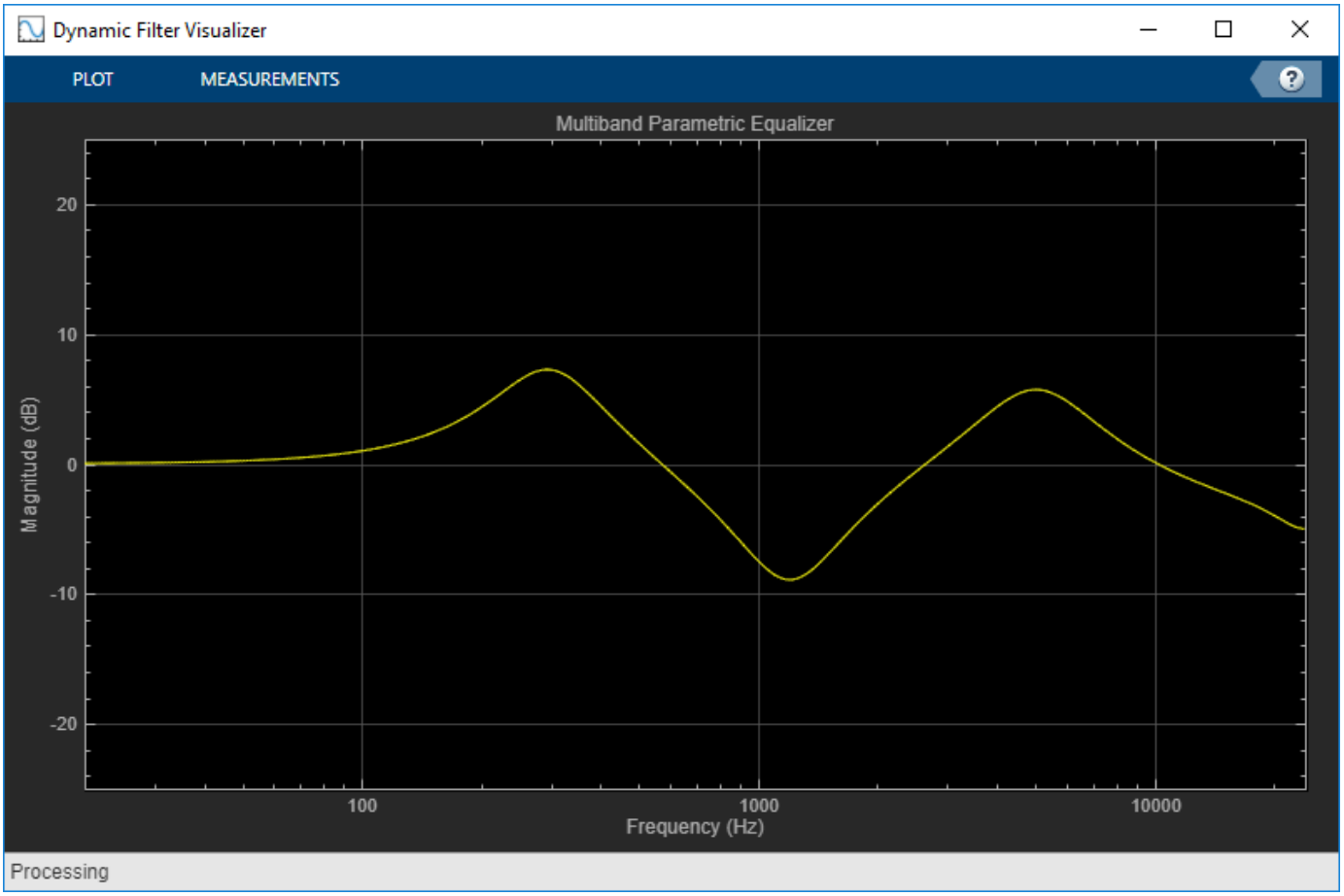
Construct a three-band parametric equalizer with a high-shelf filter.

```
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.

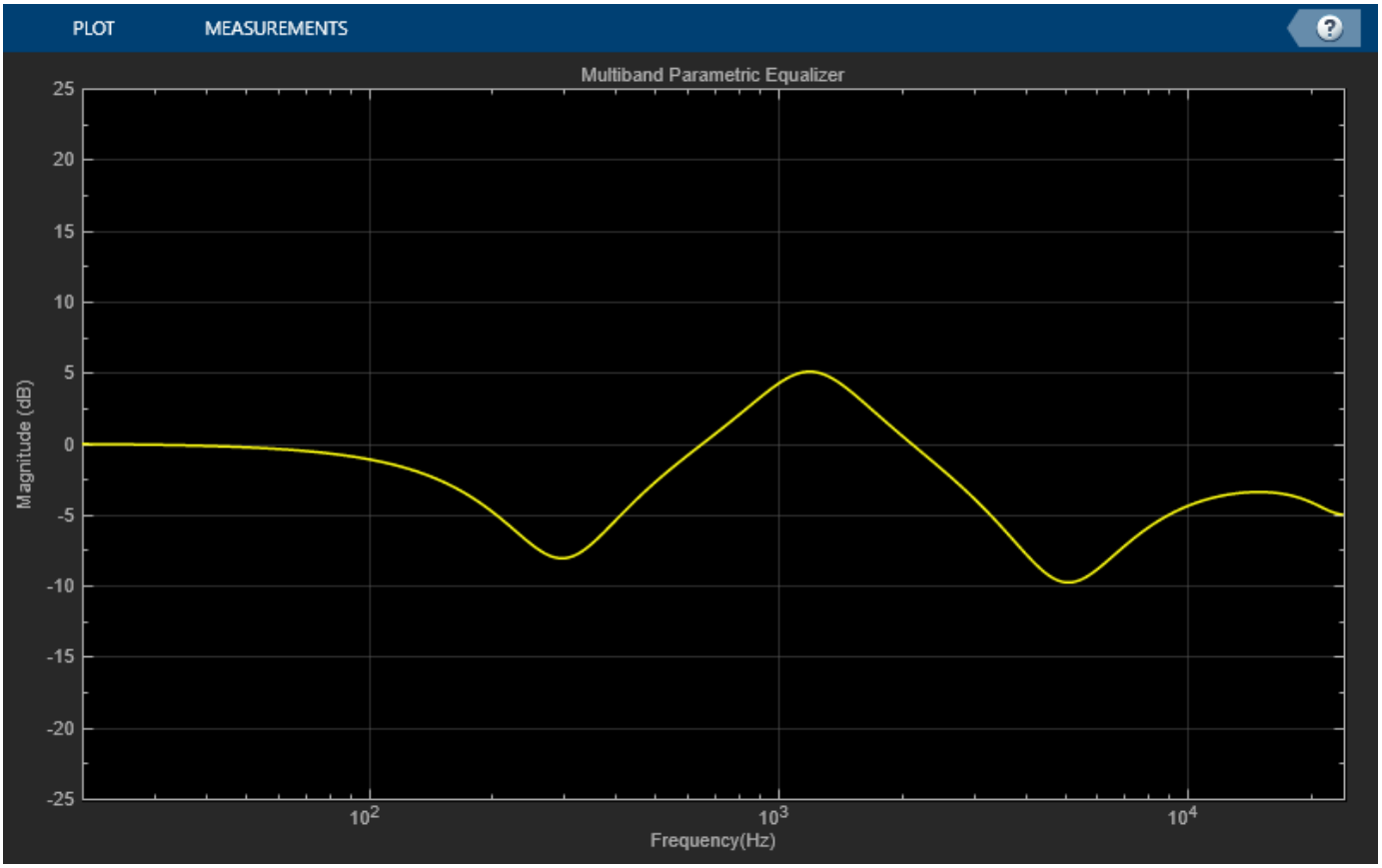
```
visualize(mPEQ)
```

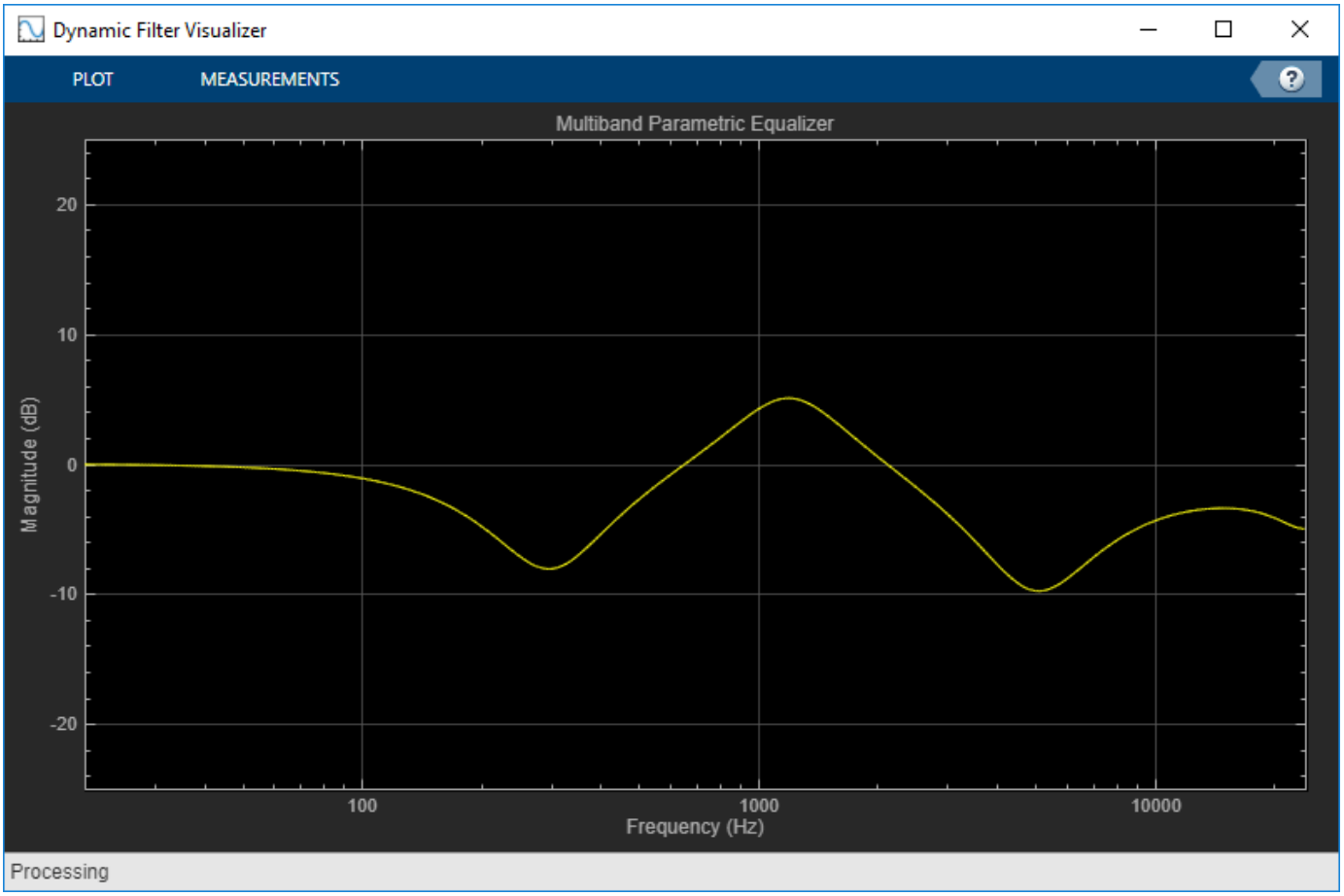

PLOT
MEASUREMENTS
Multiband Parametric Equalizer
25
20
15
10
5
0
-5
-10
-15
-20
-25
Magnitude (dB)
10^2
10^3
10^4
Frequency(Hz)



Play the equalized audio signal. Update the peak gains of your equalizer band to hear the effect of the equalizer and visualize the changing magnitude response.

```
count = 0;
while ~isDone(fileReader)
    originalSignal = fileReader();
    equalizedSignal = mPEQ(originalSignal);
    deviceWriter(equalizedSignal);
    if mod(count,100) == 0
        mPEQ.PeakGains(1) = mPEQ.PeakGains(1) - 1.5;
        mPEQ.PeakGains(2) = mPEQ.PeakGains(2) + 1.5;
        mPEQ.PeakGains(3) = mPEQ.PeakGains(3) - 1.5;
    end
    count = count + 1;
end
```


PLOT
MEASUREMENTS
Multiband Parametric Equalizer
25
20
15
10
5
0
-5
-10
-15
-20
-25
Magnitude (dB)
10^2
10^3
10^4
Frequency(Hz)



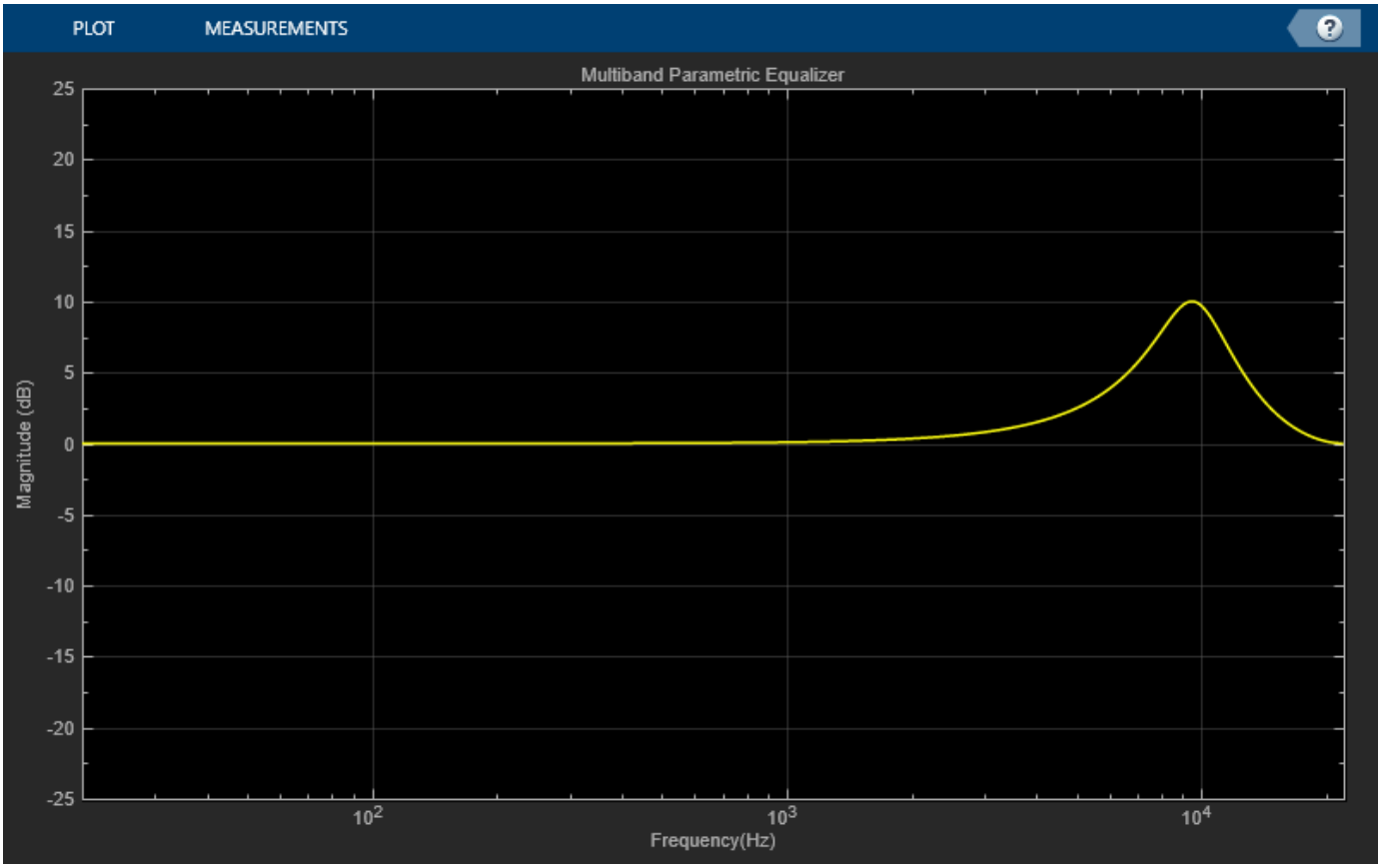

```
release(fileReader)
release(mPEQ)
release(deviceWriter)
```

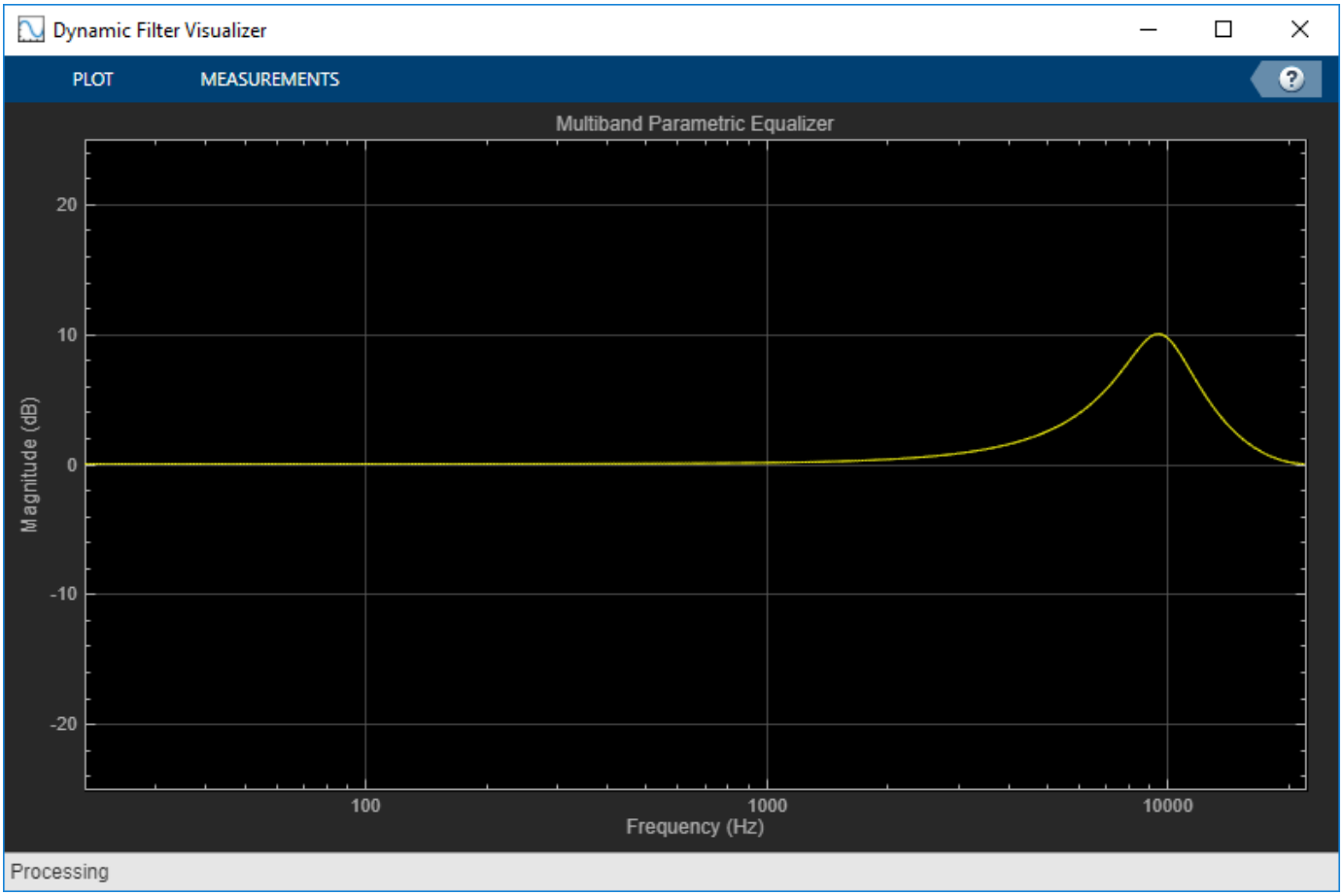
Oversample Audio Signal

Reduce warping by specifying your `multibandParametricEQ` object to perform oversampling before equalization.

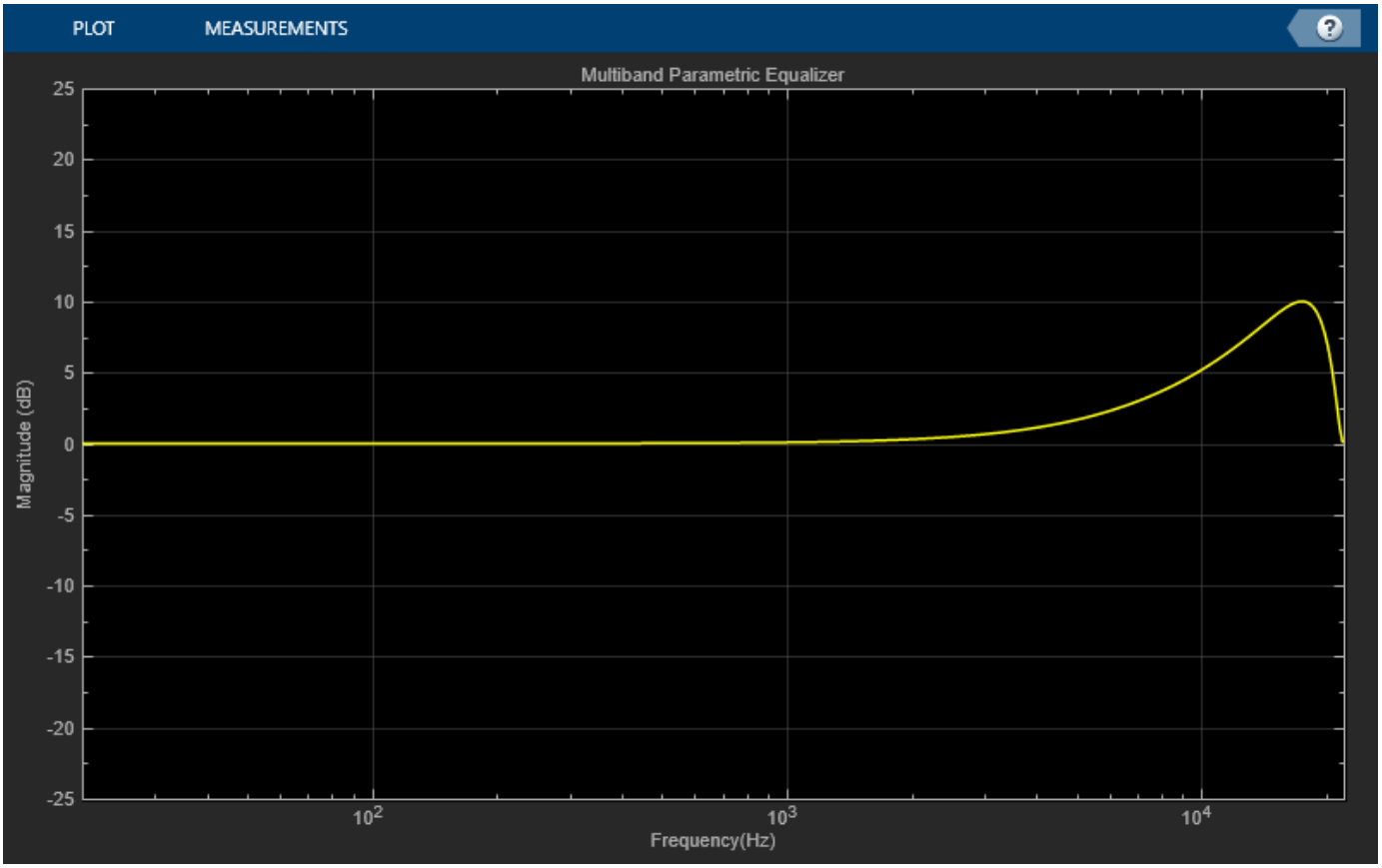
Create a one-band equalizer. Visualize the equalizer band as its center frequency approaches the Nyquist rate.

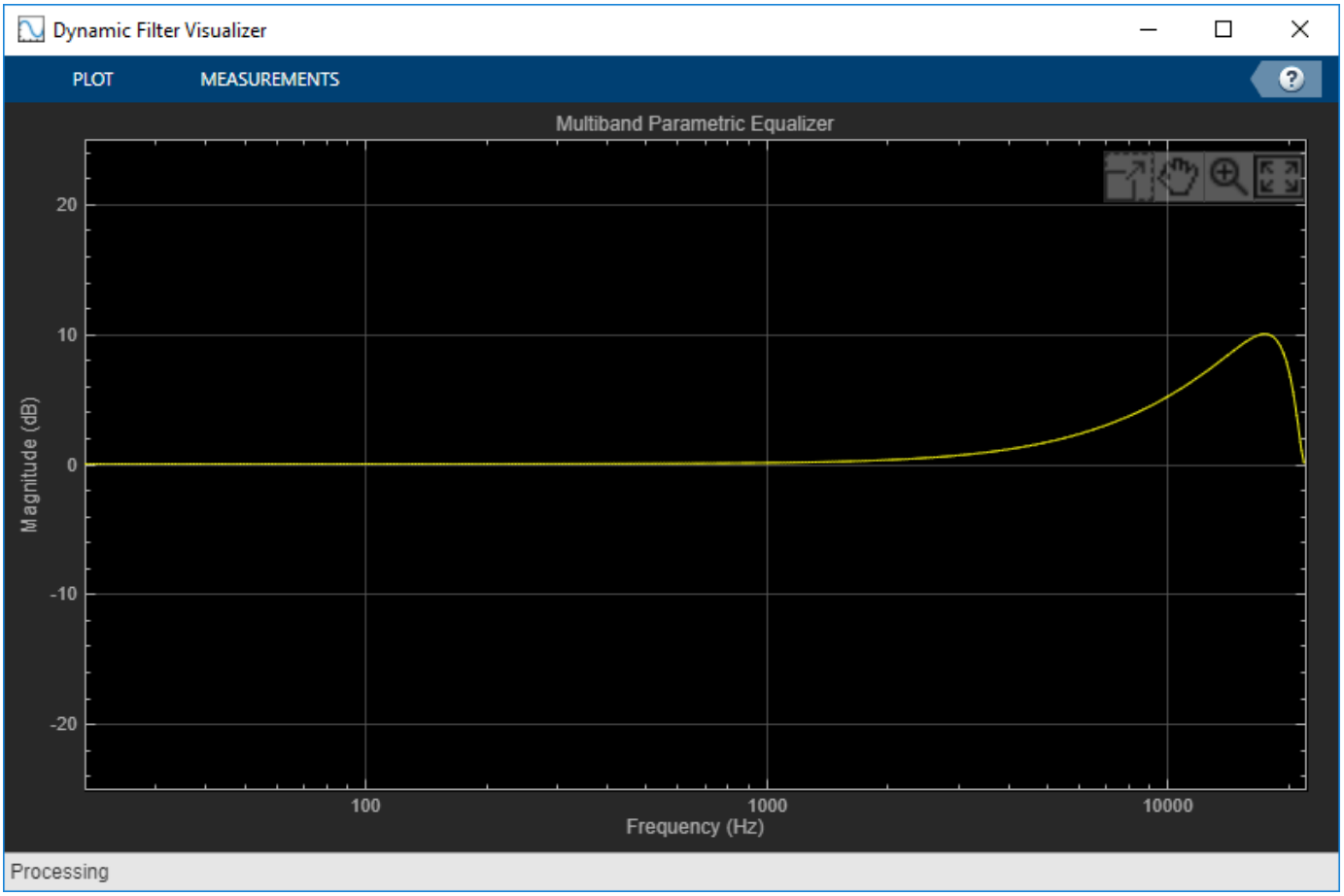
```
mPEQ = multibandParametricEQ( ...
    'NumEQBands',1, ...
    'Frequencies',9.5e3, ...
    'PeakGains',10);
visualize(mPEQ)
```


PLOT
MEASUREMENTS
Multiband Parametric Equalizer
Magnitude (dB)
25
20
15
10
5
0
-5
-10
-15
-20
-25
10^2
10^3
10^4
Frequency(Hz)




```
for i = 1:1000
    mPEQ.Frequencies = mPEQ.Frequencies + 8;
end
```

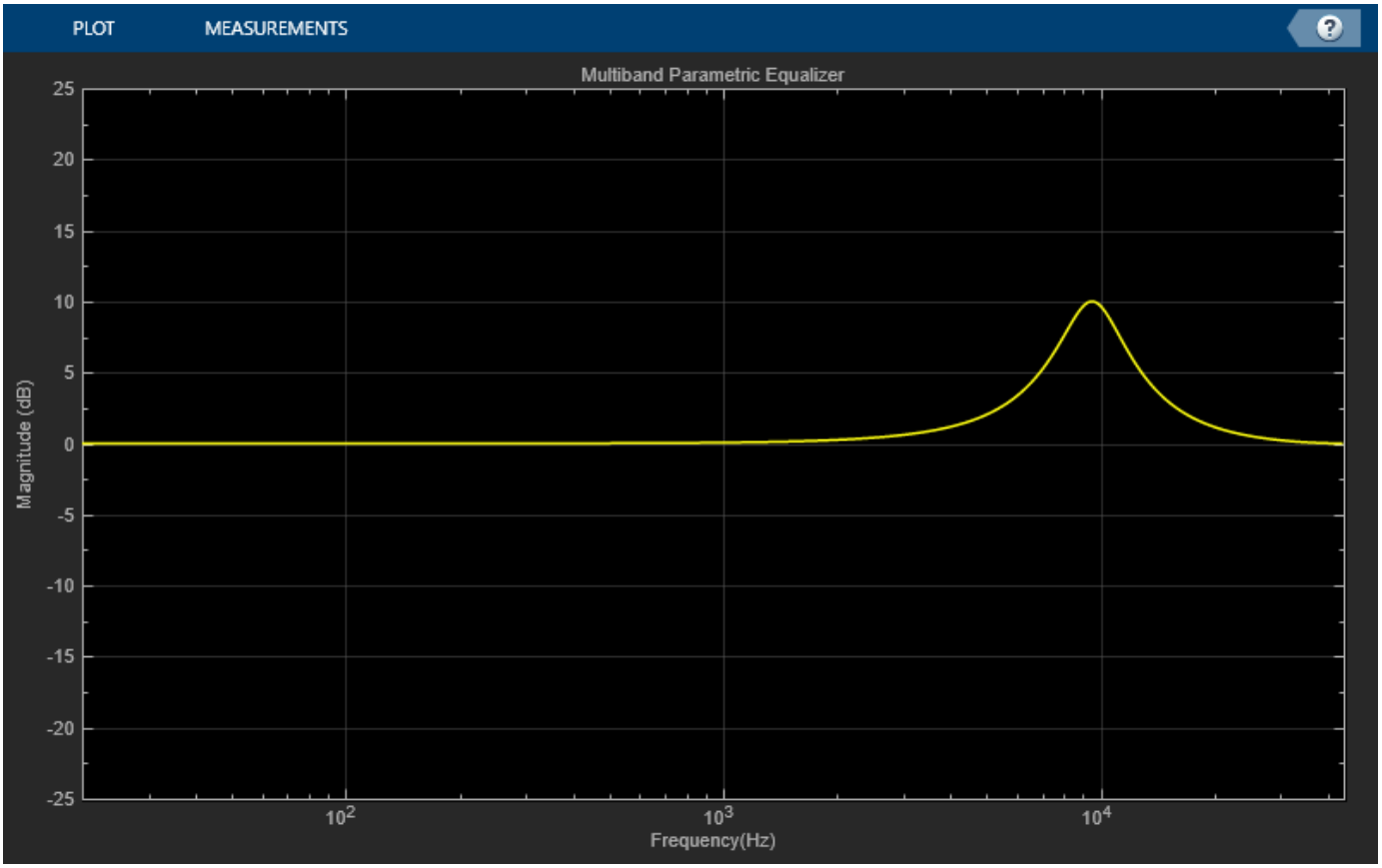

PLOT
MEASUREMENTS
Multiband Parametric Equalizer
Magnitude (dB)
25
20
15
10
5
0
-5
-10
-15
-20
-25
10^2
10^3
10^4
Frequency(Hz)

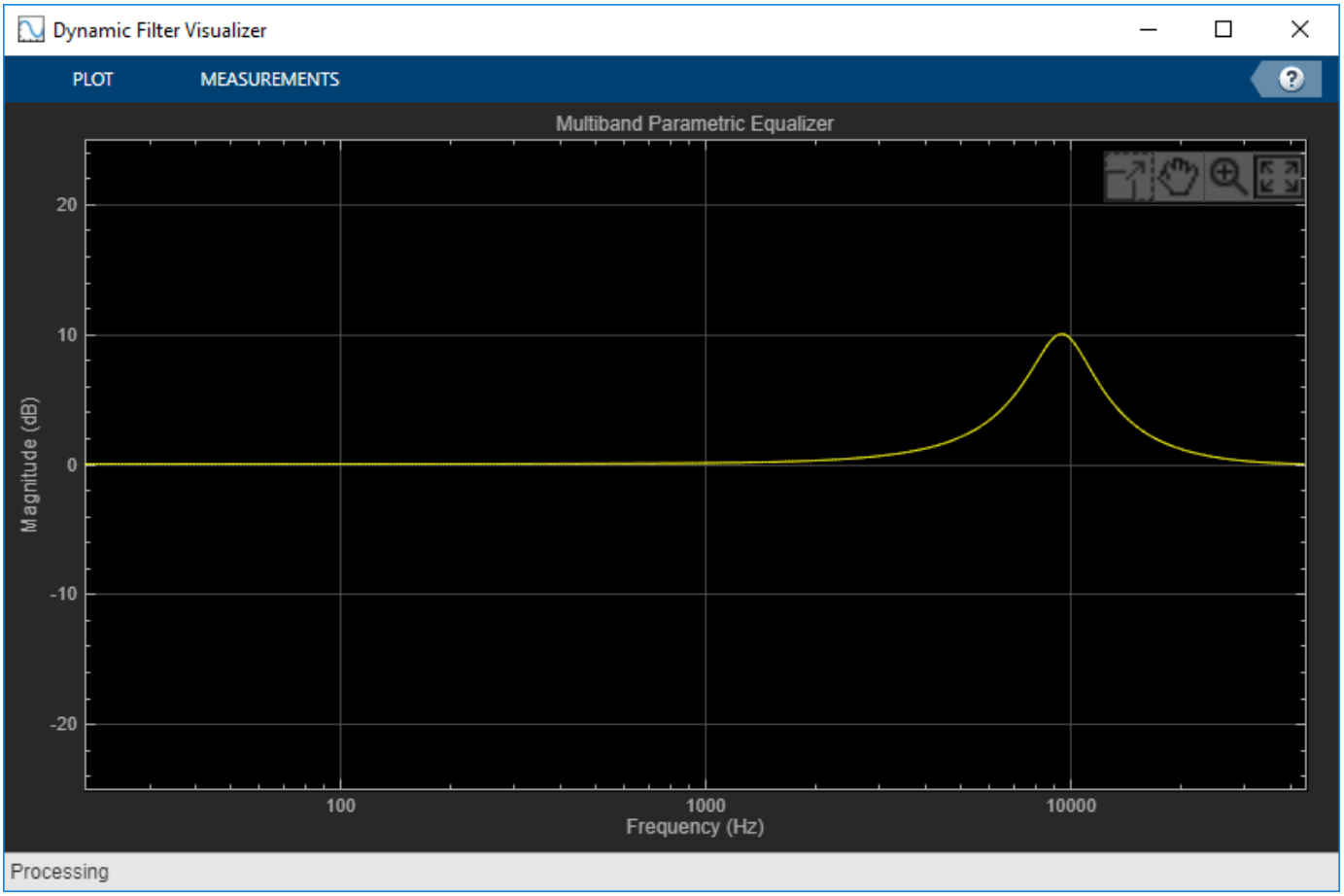


The equalizer band is warped.

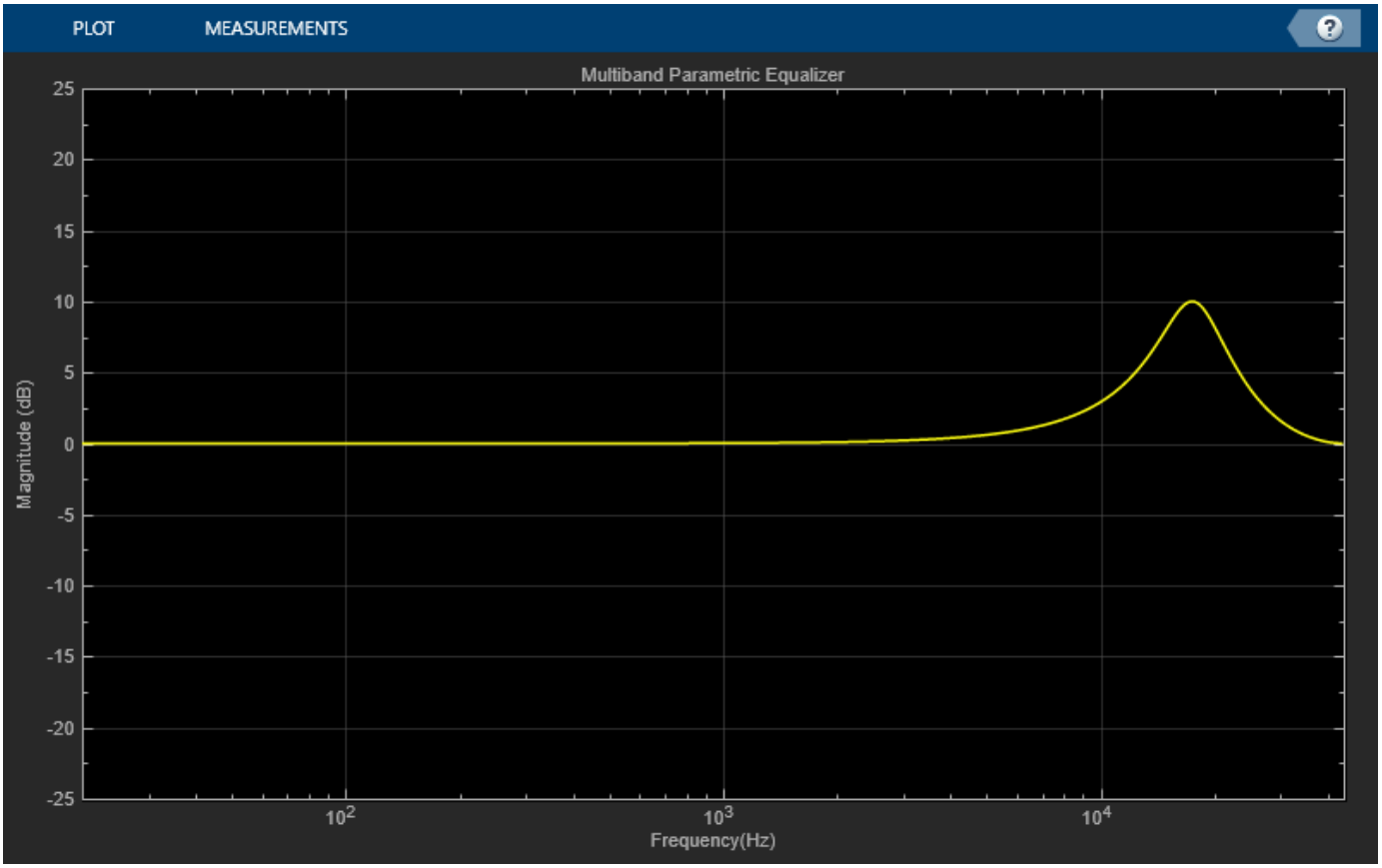
Create a one-band equalizer with `Oversample` set to `true`. Visualize the equalizer band as its center frequency approaches the Nyquist rate.

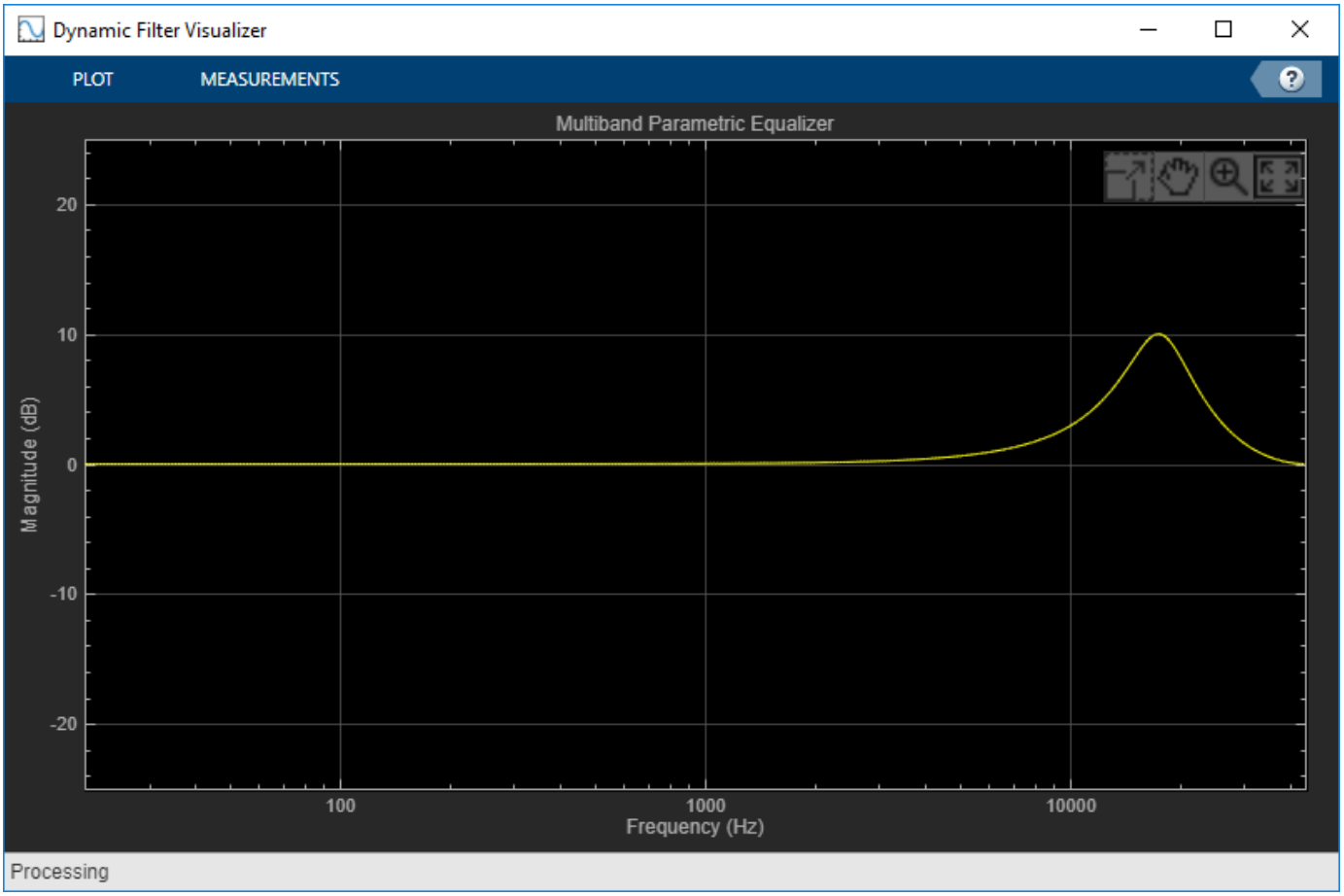
```
mPEQOversampled = multibandParametricEQ( ...
    'NumEQBands',1, ...
    'Frequencies',9.5e3, ...
    'PeakGains',10, ...
    'Oversample',true);
visualize(mPEQOversampled)
```


PLOT
MEASUREMENTS
Multiband Parametric Equalizer
Magnitude (dB)
25
20
15
10
5
0
-5
-10
-15
-20
-25
10^2
10^3
10^4
Frequency(Hz)




```
for i = 1:1000
    mPEQOversampled.Frequencies = mPEQOversampled.Frequencies + 8;
end
```


PLOT
MEASUREMENTS
Multiband Parametric Equalizer
25
20
15
10
5
0
-5
-10
-15
-20
-25
Magnitude (dB)
10^2
10^3
10^4
Frequency(Hz)

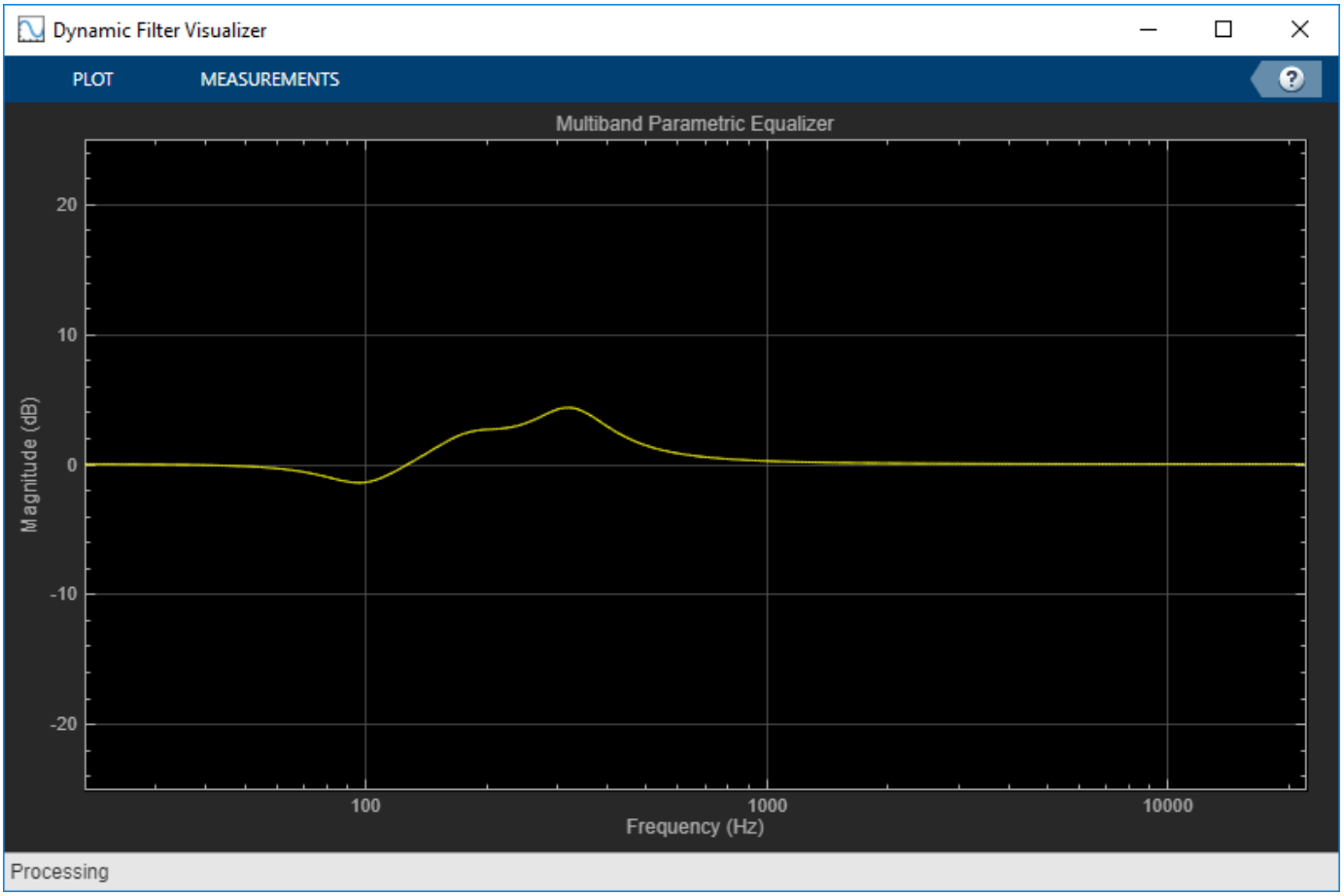


Warping is reduced.

Tune Multiband Parametric EQ Parameters

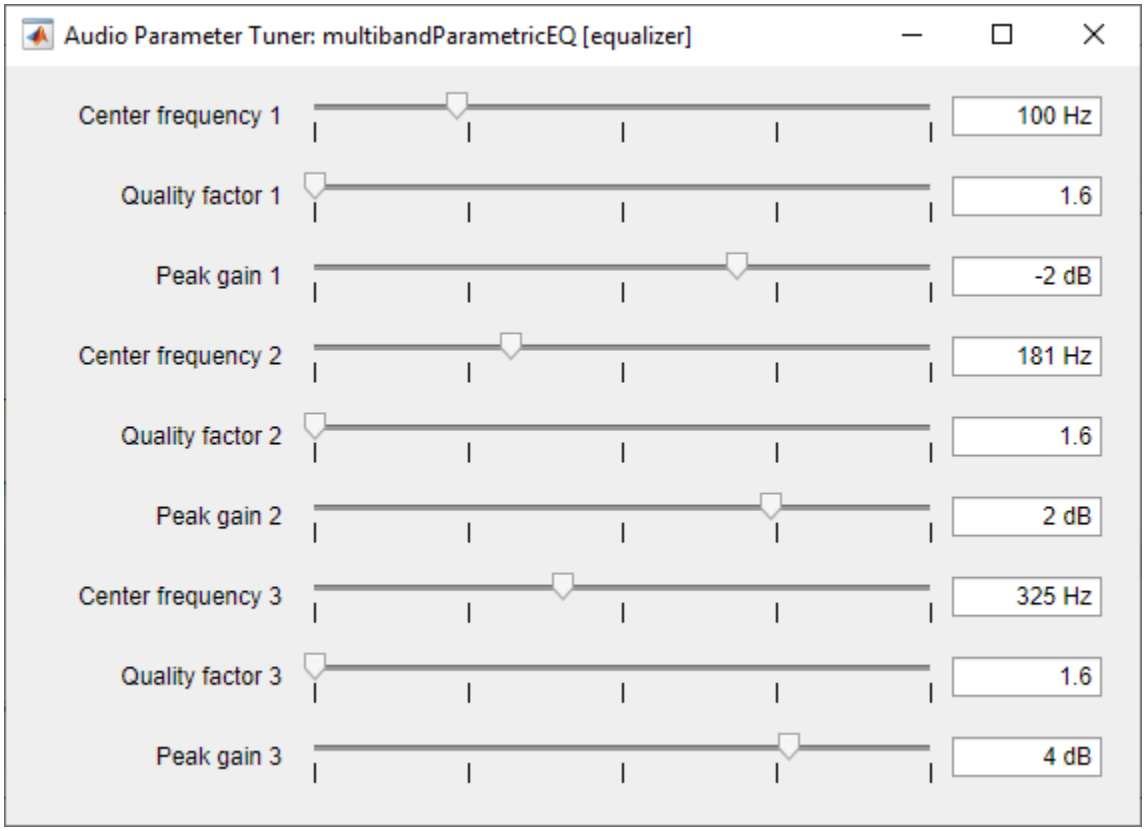
Create a `dsp.AudioFileReader` to read in audio frame-by-frame. Create a `audioDeviceWriter` to write audio to your sound card. Create a `multibandParametricEQ` to process the audio data. Call `visualize` to plot the frequency response of the equalizer.

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

equalizer = multibandParametricEQ('SampleRate',fileReader.SampleRate, 'PeakGains',[-2,2,4]);
visualize(equalizer)
```



Call `parameterTuner` to open a UI to tune parameters of the equalizer while streaming.

```
parameterTuner(equalizer)
```



In an audio stream loop:

1. Read in a frame of audio from the file.
2. Apply equalization.
3. Write the frame of audio to your audio device for listening.

While streaming, tune parameters of the equalizer and listen to the effect.

```
while ~isDone(fileReader)
    audioIn = fileReader();
    audioOut = equalizer(audioIn);
    deviceWriter(audioOut);
    drawnow limitrate % required to update parameter
end
```

As a best practice, release your objects once done.

```
release(deviceWriter)
release(fileReader)
release(equalizer)
```

Version History

Introduced in R2016a

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

`designShelvingEQ` | Single-Band Parametric EQ | Multiband Parametric EQ | `designVarSlopeFilter` | `designParamEQ`

Topics

“Parametric Equalizer Design”
“Equalization”

visualize

Visualize magnitude response of multiband parametric equalizer

Syntax

```
visualize(mPEQ)
visualize(mPEQ,NFFT)
hvsz = visualize( ___ )
```

Description

`visualize(mPEQ)` plots the magnitude response of the `multibandParametricEQ` object `mPEQ`. The plot is updated automatically when properties of the object change.

`visualize(mPEQ,NFFT)` specifies an N-point FFT used to calculate the magnitude response.

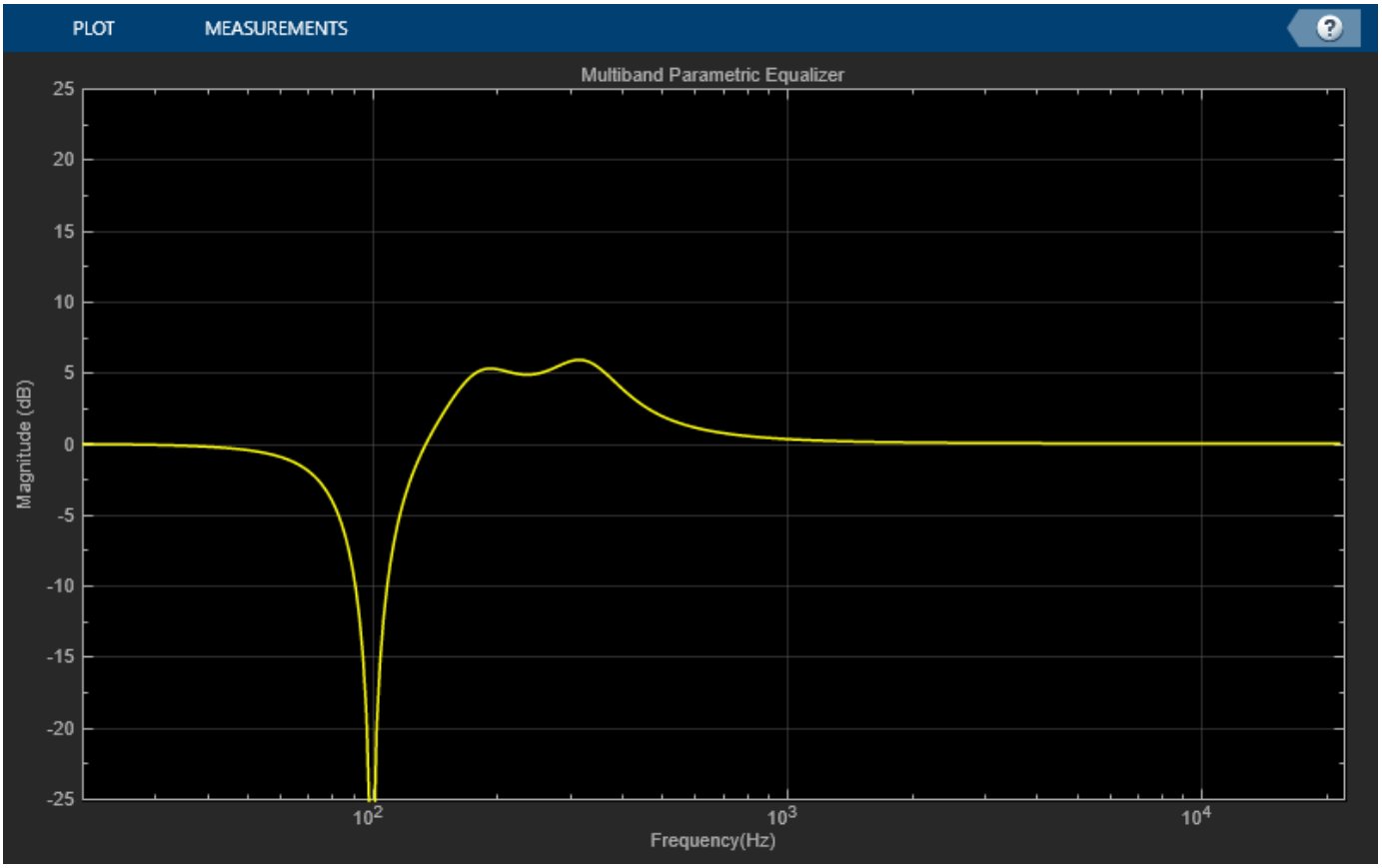
`hvsz = visualize( ___ )` returns a handle to the visualizer as a `dsp.DynamicFilterVisualizer` object when called with any of the previous syntaxes.

Examples

Specify a Nondefault Number of FFT Points

Create a `multibandParametricEQ` System object™, and then call `visualize` to plot the magnitude response using a 5096-point FFT.

```
mPEQ = multibandParametricEQ('PeakGains',[-inf,5,5]);
visualize(mPEQ,5096)
```



Input Arguments

`mPEQ` — Multiband parametric equalizer to visualize

object of `multibandParametricEQ` System object

Multiband parametric equalizer whose magnitude response you want to plot.

`NFFT` — N-point FFT

2048 (default) | positive scalar

Number of bins used to calculate the DFT, specified as a positive scalar.

Data Types: `single` | `double`

Version History

Introduced in R2016a

See Also

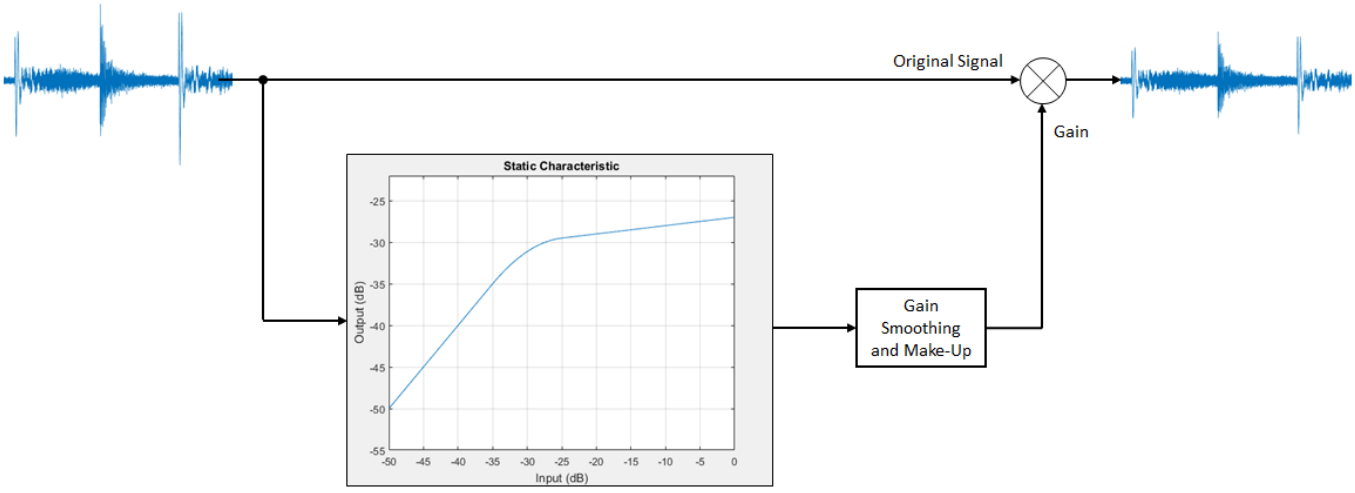
`multibandParametricEQ`

compressor

Dynamic range compressor

Description

The `compressor` System object performs dynamic range compression independently across each input channel. Dynamic range compression attenuates the volume of loud sounds that cross a given threshold. It uses specified attack and release times to achieve a smooth applied gain curve. Properties of the `compressor` System object specify the type of dynamic range compression.



To perform dynamic range compression:

1. Create the `compressor` object and set its properties.
2. Call the object with arguments, as if it were a function.

To learn more about how System objects work, see What Are System Objects?

Creation

Syntax

```
dRC = compressor
dRC = compressor(thresholdValue)
dRC = compressor(thresholdValue,ratioValue)
dRC = compressor( ___ ,Name,Value)
```

Description

`dRC = compressor` creates a System object, `dRC`, that performs dynamic range compression independently across each input channel over time.

`dRC = compressor(thresholdValue)` sets the Threshold property to `thresholdValue`.

`dRC = compressor(thresholdValue,ratioValue)` sets the Ratio property to `ratioValue`.

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

Example: `dRC = compressor('AttackTime',0.01,'SampleRate',16000)` creates a System object, `dRC`, with a 10 ms attack time operating at a 16 kHz sample rate.

Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see System Design in MATLAB Using System Objects.

`Threshold` — Operation threshold (dB)

`−10` (default) | real scalar

Operation threshold in dB, specified as a real scalar.

Operation threshold is the level above which gain is applied to the input signal.

Tunable: Yes

Data Types: `single` | `double`

`Ratio` — Compression ratio

`5` (default) | real scalar

Compression ratio, specified as a real scalar greater than or equal to 1.

Compression ratio is the input/output ratio for signals that overshoot the operation threshold.

Assuming a hard knee characteristic and a steady-state input such that $x[n]$ dB > `Threshold`, the compression ratio is defined as $R = \frac{(x[n] - T)}{(y[n] - T)}$.

- R is the compression ratio.
- $x[n]$ is the input signal in dB.
- $y[n]$ is the output signal in dB.
- T is the threshold in dB.

Tunable: Yes

Data Types: `single` | `double`

`KneeWidth` — Knee width (dB)

`0` (default) | real scalar

Knee width in dB, specified as a real scalar greater than or equal to 0.

Knee width is the transition area in the compression characteristic.

For soft knee characteristics, the transition area is defined by the relation

$$y = x + \frac{\left(\frac{1}{R} - 1\right) \times \left(x - T + \frac{W}{2}\right)^2}{(2 \times W)}$$

for the range $(2 \times |x - T|) \le W$.

- 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

Data Types: `single` | `double`

`AttackTime` — Attack time (s)
`0.05` (default) | real scalar

Attack time in seconds, specified as a real scalar greater than or equal to 0.

Attack time is the time it takes the compressor gain to rise from 10% to 90% of its final value when the input goes above the threshold.

Tunable: Yes

Data Types: `single` | `double`

`ReleaseTime` — Release time (s)
`0.2` (default) | real scalar

Release time in seconds, specified as a real scalar greater than or equal to 0.

Release time is the time it takes the compressor gain to drop from 90% to 10% of its final value when the input goes below the threshold.

Tunable: Yes

Data Types: `single` | `double`

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

Make-up gain mode, specified as `'Auto'` or `'Property'`.

- `'Auto'` –– Make-up gain is applied at the output of the dynamic range compressor such that a steady-state 0 dB input has a 0 dB output.
- `'Property'` –– Make-up gain is set to the value specified in the MakeUpGain property.

Tunable: No

Data Types: `char` | `string`

`MakeUpGain` — Make-up gain (dB)

`0` (default) | real scalar

Make-up gain in dB, specified as a real scalar.

Make-up gain compensates for gain lost during compression. It is applied at the output of the dynamic range compressor.

Tunable: Yes

Dependencies

To enable this property, set MakeUpGainMode to `'Property'`.

Data Types: `single` | `double`

`SampleRate` — Input sample rate (Hz)

`44100` (default) | positive scalar

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

Tunable: Yes

Data Types: `single` | `double`

`EnableSidechain` — Enable sidechain input

`false` (default) | `true`

Enable sidechain input, specified as `true` or `false`. This property determines the number of available inputs on the `compressor` object.

- `false` –– Sidechain input is disabled and the `compressor` object accepts one input: the `audioIn` data to be compressed.
- `true` –– Sidechain input is enabled and the `compressor` object accepts two inputs: the `audioIn` data to be compressed and the sidechain input used to compute the compression `gain`.

The sidechain datatype and (frame) length must be the same as `audioIn`.

The number of channels of the sidechain input must be equal to the number of channels of `audioIn` or be equal to one. When the number of sidechain channels is one, the `gain` computed based on this channel is applied to all channels of `audioIn`. When the number of sidechain channels is equal to the number of channels in `audioIn`, the `gain` computed for each sidechain channel is applied to the corresponding channel of `audioIn`.

Tunable: No

Usage

Syntax

```
audioOut = dRC(audioIn)
[audioOut,gain] = dRC(audioIn)
```

Description

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

`[audioOut,gain] = dRC(audioIn)` also returns the applied gain, in dB, at each input sample.

Input Arguments

`audioIn` — Audio input to compressor
matrix

Audio input to the compressor, specified as a matrix. The columns of the matrix are treated as independent audio channels.

Data Types: `single` | `double`

Output Arguments

`audioOut` — Audio output from compressor
matrix

Audio output from the compressor, returned as a matrix the same size as `audioIn`.

Data Types: `single` | `double`

`gain` — Gain applied by compressor (dB)
matrix

Gain applied by compressor, returned as a matrix the same size as `audioIn`.

Data Types: `single` | `double`

Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

Specific to compressor

visualize	Visualize static characteristic of dynamic range controller
staticCharacteristic	Return static characteristic of dynamic range controller
createAudioPluginClass	Create audio plugin class that implements functionality of System object
parameterTuner	Tune object parameters while streaming

MIDI

configureMIDI	Configure MIDI connections between audio object and MIDI controller
disconnectMIDI	Disconnect MIDI controls from audio object
getMIDIConnections	Get MIDI connections of audio object

Common to All System Objects

clone	Create duplicate System object
isLocked	Determine if System object is in use
release	Release resources and allow changes to System object property values and input characteristics
reset	Reset internal states of System object
step	Run System object algorithm

The `createAudioPluginClass` and `configureMIDI` functions map tunable properties of the `compressor` System object to user-facing parameters:

Property	Range	Mapping	Unit
`Threshold`	[–50, 0]	linear	dB
`Ratio`	[1, 50]	linear	none
`KneeWidth`	[0, 20]	linear	dB
`AttackTime`	[0, 4]	linear	seconds
`ReleaseTime`	[0, 4]	linear	seconds
`MakeUpGain` (available when you set `MakeUpGainMode` to `'Property'`)	[–10, 24]	linear	dB

Examples

Compress Audio Signal

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

Set up the `dsp.AudioFileReader` and `audioDeviceWriter` System objects™.

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

Set up the compressor to have a threshold of -15 dB, a ratio of 7, and a knee width of 5 dB. Use the sample rate of your audio file reader.

```
dRC = compressor(-15,7, ...
    'KneeWidth',5, ...
    'SampleRate',fileReader.SampleRate);
```

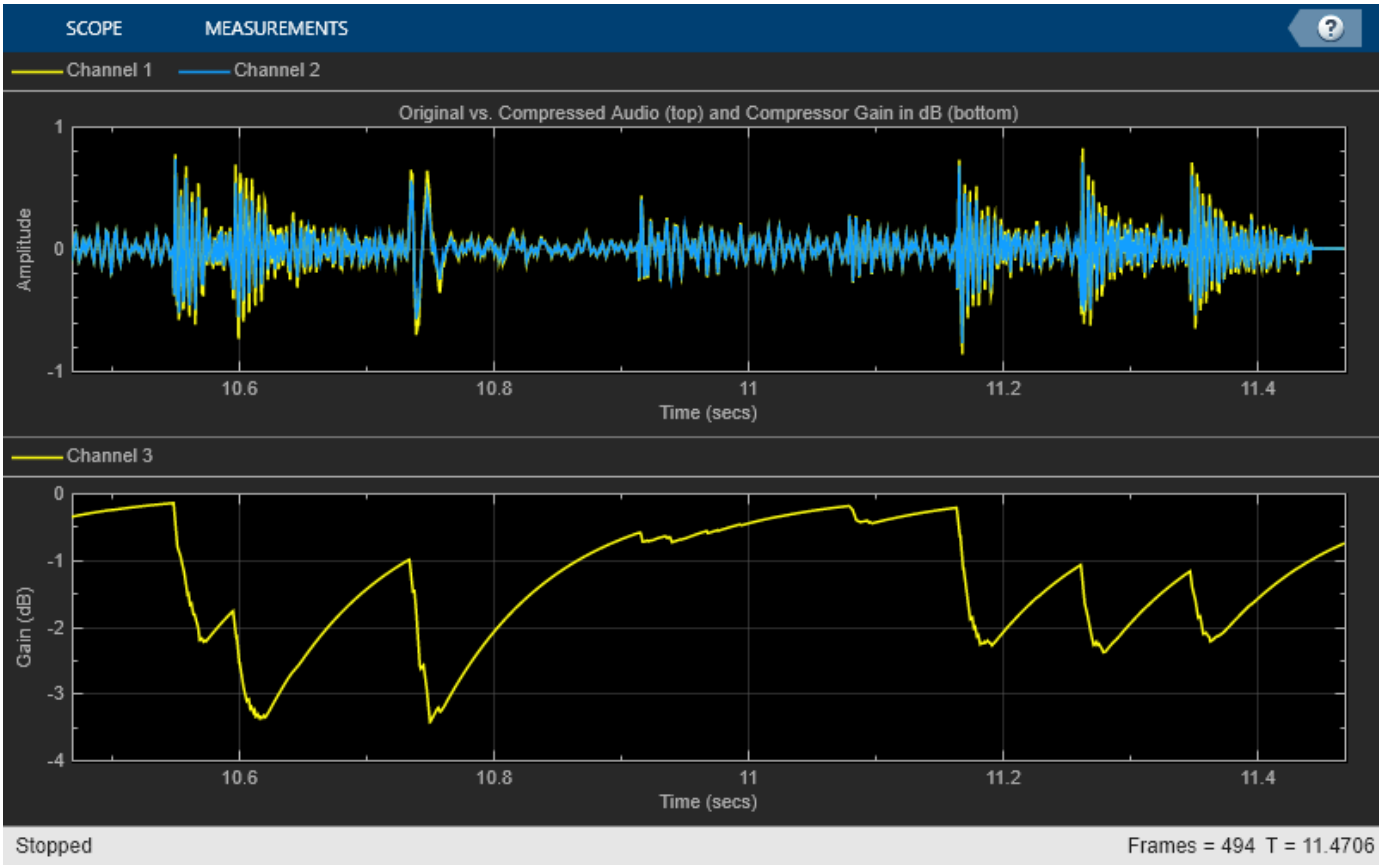
Set up the scope to visualize the original audio signal, the compressed audio signal, and the applied compressor gain.

```
scope = timescope( ...
    'SampleRate',fileReader.SampleRate, ...
    'TimeSpanSource','Property','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`.

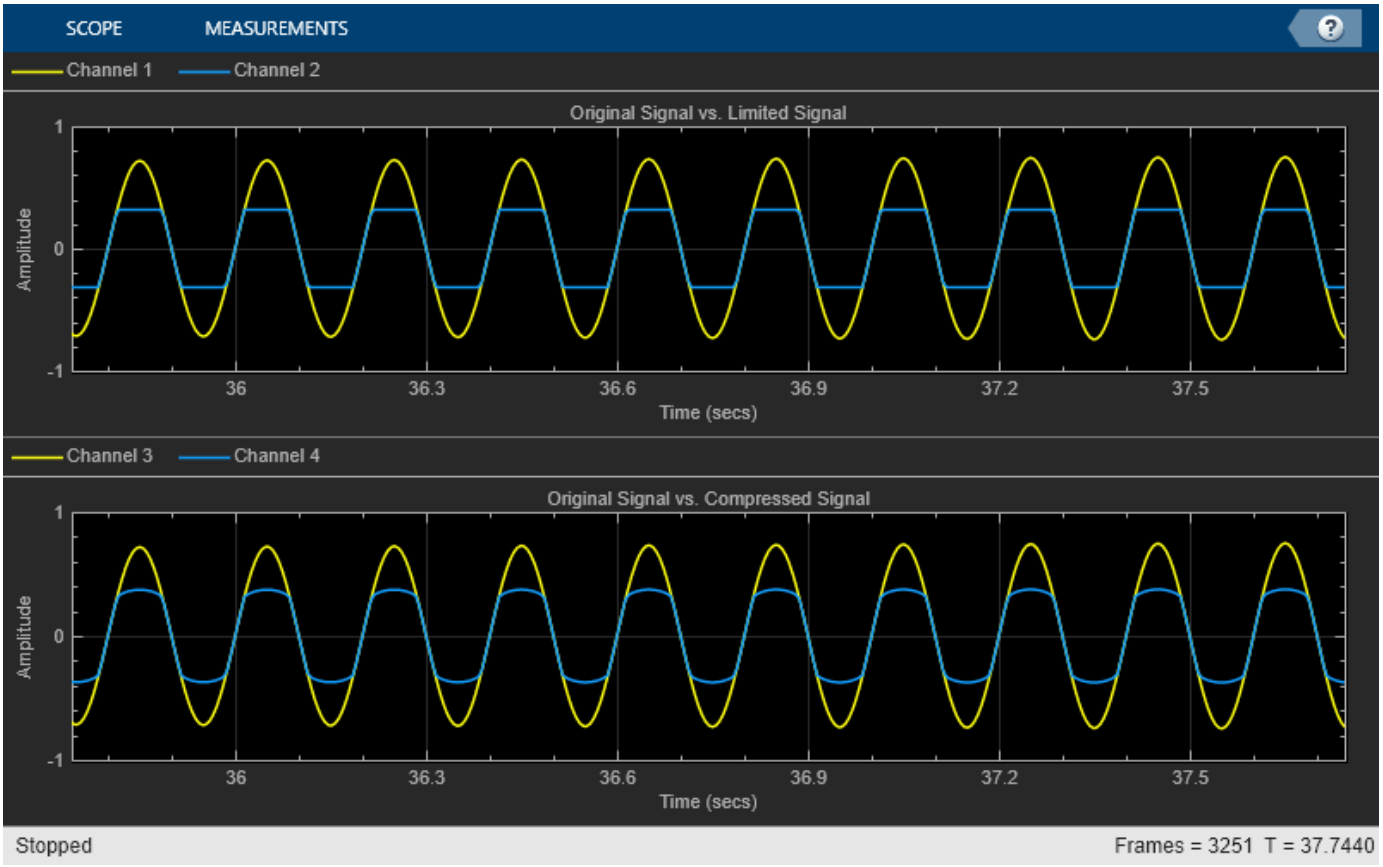
```
dRL = limiter('AttackTime',0,'ReleaseTime',0);
dRC = compressor('AttackTime',0,'ReleaseTime',0);

osc = audioOscillator('Frequency',5,'Amplitude',0.1);
```

Create a time scope to visualize the generated sinusoid and the processed sinusoid.

```
scope = timescope( ...
    'SampleRate',osc.SampleRate, ...
    'TimeSpanSource','Property','TimeSpan',2, ...
    'BufferLength',osc.SampleRate*4, ...
    'TimeSpanOverrunAction','Scroll', ...
    'ShowGrid',true, ...
    'LayoutDimensions',[2 1], ...
    'NumInputPorts',2);
scope.ActiveDisplay = 1;
scope.Title = 'Original Signal vs. Limited Signal';
scope.YLimits = [-1,1];
scope.ActiveDisplay = 2;
scope.Title = 'Original Signal vs. Compressed Signal';
scope.YLimits = [-1,1];
```

In an audio stream loop, visualize the original sinusoid and the sinusoid processed by a limiter and a compressor. Increment the amplitude of the original sinusoid to illustrate the effect.

```
while osc.Amplitude < 0.75
    x = osc();

    xLimited    = dRL(x);
    xCompressed = dRC(x);

    scope([x xLimited],[x xCompressed]);

    osc.Amplitude = osc.Amplitude + 0.0002;
end
release(scope)
```




```
release(dRL)
release(dRC)
release(osc)
```

Compare Limiter and Compressor Applied to Audio Signal

Compare the effect of dynamic range limiters and compressors on a drum track. Create a `dsp.AudioFileReader` System object and a `audioDeviceWriter` System object to read audio from a file and write to your audio output device. To emphasize the effect of dynamic range control, set the operational threshold of the limiter and compressor to -20 dB.

```
dRL.Threshold = -20;
dRC.Threshold = -20;

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

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

```
numFrames = 300;

fprintf('Now playing original signal...\n')
```

```
Now playing original signal...
```

```
for i = 1:numFrames
    x = fileReader();
```

```
    deviceWriter(x);
end
reset(fileReader);

fprintf('Now playing limited signal...\n')
```

```
Now playing limited signal...
```

```
for i = 1:numFrames
    x = fileReader();
    xLimited = dRL(x);
    deviceWriter(xLimited);
end
reset(fileReader);

fprintf('Now playing compressed signal...\n')
```

```
Now playing compressed signal...
```

```
for i = 1:numFrames
    x = fileReader();
    xCompressed = dRC(x);
    deviceWriter(xCompressed);
end

release(fileReader)
release(deviceWriter)
release(dRC)
release(dRL)
```

Diminish Plosives from Speech Signal

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

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

```
fileReader = dsp.AudioFileReader('audioPlosives.wav');
deviceWriter = audioDeviceWriter('SampleRate',fileReader.SampleRate);

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` 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/(fileReader.SampleRate/2),"hi");
biquadFilter = dsp.BiquadFilter( ...
    "SOSMatrixSource","Input port", ...
    "ScaleValuesInputPort",false);

crossFilt = crossoverFilter( ...
    "SampleRate",fileReader.SampleRate, ...
    "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 = timescope( ...
    "SampleRate",fileReader.SampleRate, ...
    "TimeSpanSource","property","TimeSpan",3, ...
    "BufferLength",fileReader.SampleRate*3*2, ...
    "YLimits",[-1 1], ...
    "ShowGrid",true, ...
    "ShowLegend",true, ...
    "ChannelNames",{'Original','Processed'});
```

In an audio stream loop:

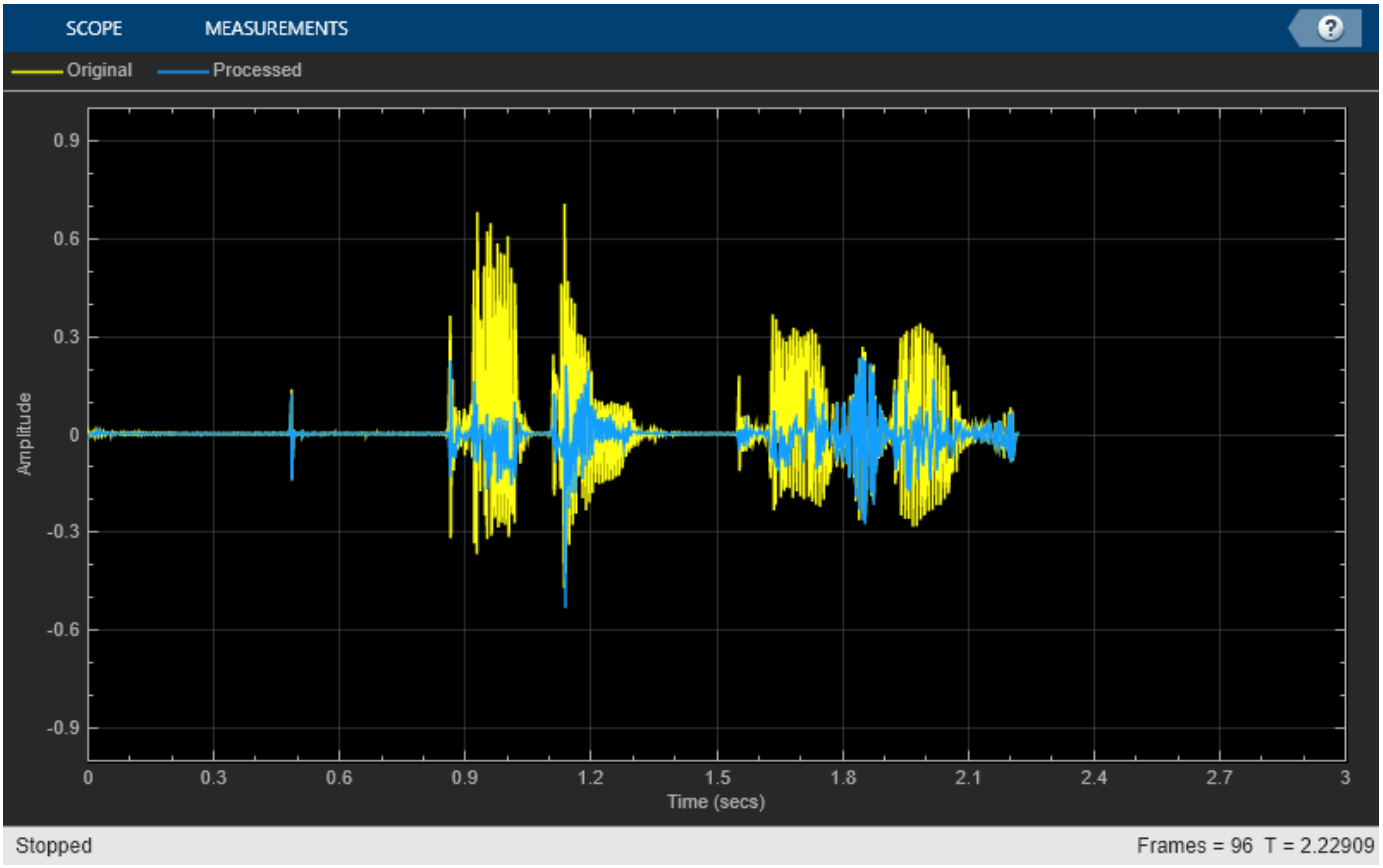
1. Read in a frame of the audio file.
2. Apply highpass filtering using your biquad filter.
3. Split the audio signal into two bands.
4. Apply dynamic range compression to the lower band.
5. Remix the channels.
6. Write the processed audio signal to your audio device for listening.
7. Visualize the processed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

As a best practice, release your objects once done.

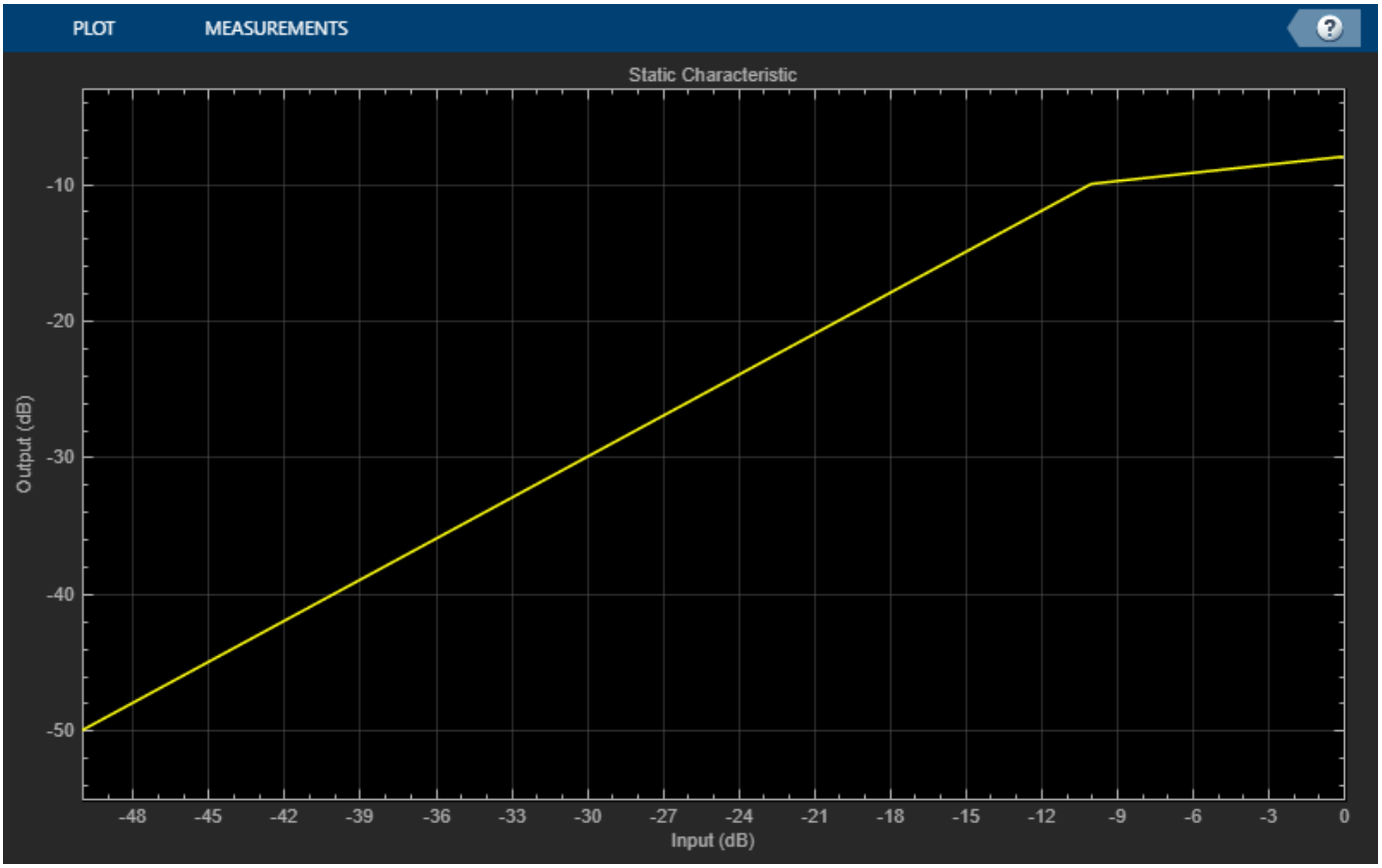
```
release(deviceWriter)
release(fileReader)
release(crossFilt)
release(dRCompressor)
release(scope)
```



Tune Compressor Parameters

Create a `dsp.AudioFileReader` to read in audio frame-by-frame. Create an `audioDeviceWriter` to write audio to your sound card. Create a `compressor` to process the audio data. Call `visualize` to plot the static characteristic of the `compressor`.

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

dRC = compressor('SampleRate',fileReader.SampleRate);
visualize(dRC)
```

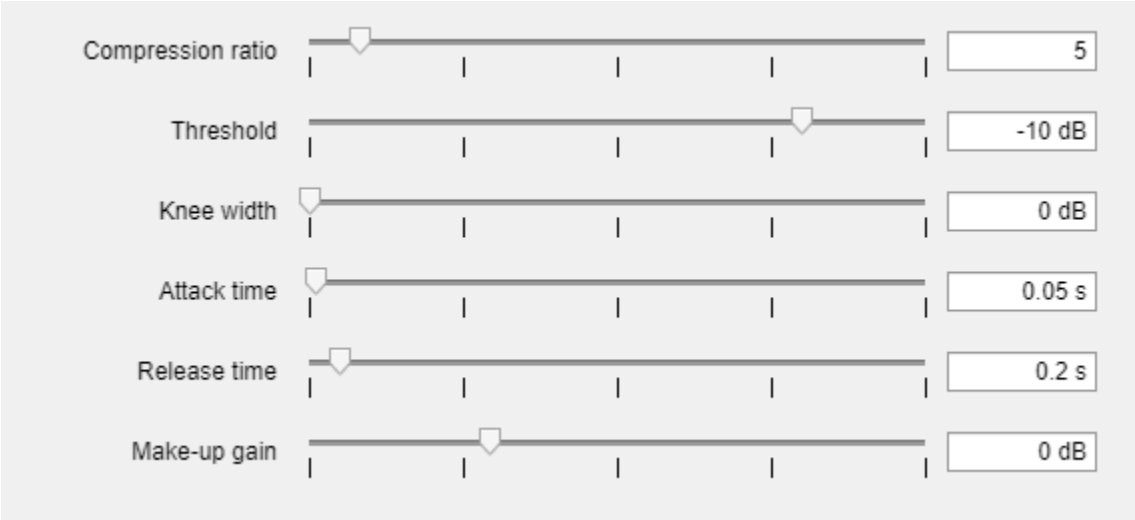


Create a `timescope` to visualize the original and processed audio.

```
scope = timescope( ...
    'SampleRate',fileReader.SampleRate, ...
    'TimeSpanSource','property',...
    'TimeSpan',1, ...
    'BufferLength',fileReader.SampleRate*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)';
```

Call `parameterTuner` to open a UI to tune parameters of the compressor while streaming.

```
parameterTuner(dRC)
```



In an audio stream loop:

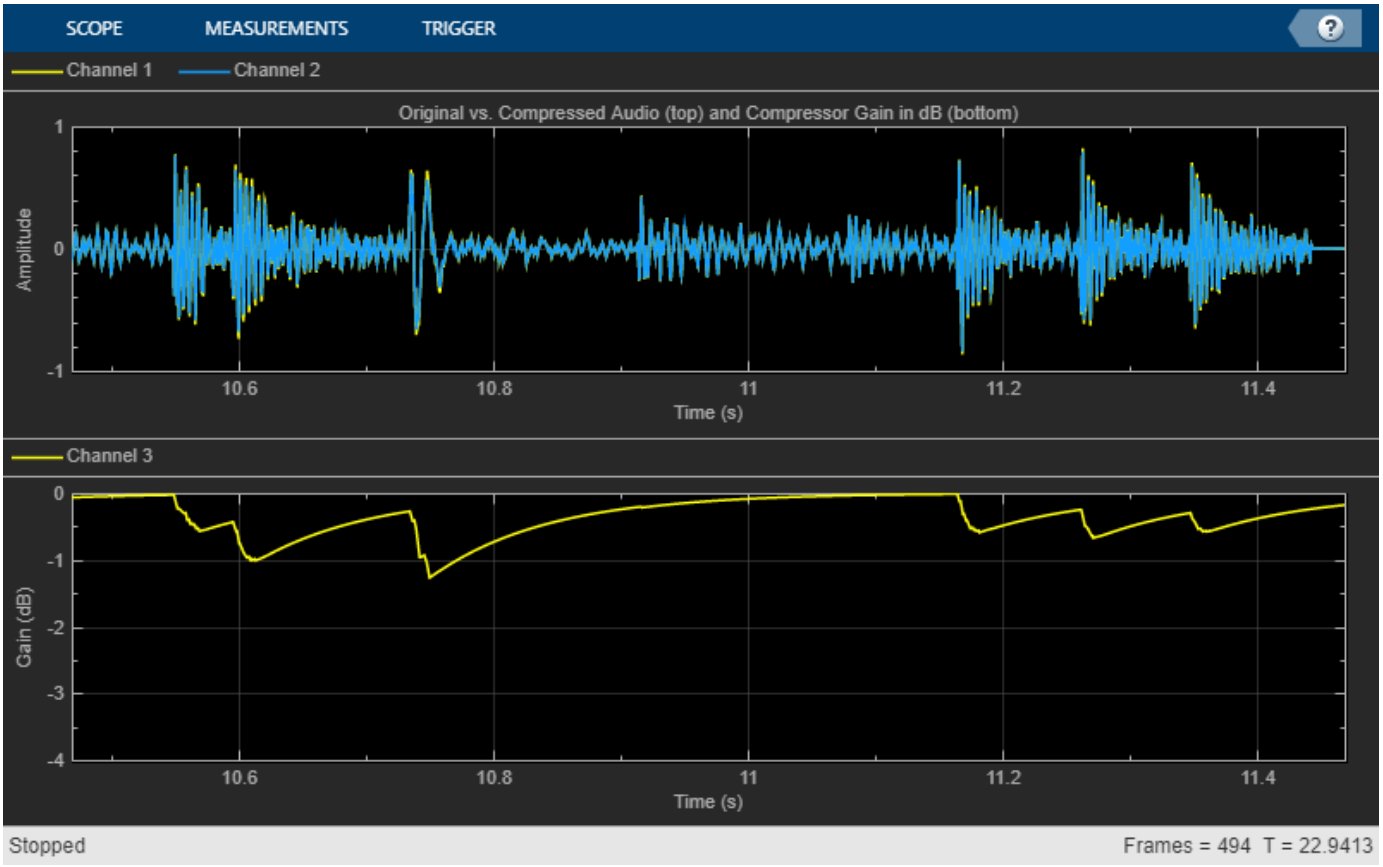
1. Read in a frame of audio from the file.
2. Apply dynamic range compression.
3. Write the frame of audio to your audio device for listening.
4. Visualize the original audio, the processed audio, and the gain applied.

While streaming, tune parameters of the dynamic range compressor and listen to the effect.

```
while ~isDone(fileReader)
    audioIn = fileReader();
    [audioOut,g] = dRC(audioIn);
    deviceWriter(audioOut);
    scope([audioIn(:,1),audioOut(:,1)],g(:,1));
    drawnow limitrate % required to update parameter
end
```

As a best practice, release your objects once done.

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



Sidechain Ducking with Compressor

Use the "EnableSidechain" on page 3-0 input of a `compressor` object to reduce the amplitude level of a separate audio signal. The sidechain signal controls the compression on the input audio signal. When the sidechain signal exceeds the compressor "Threshold" on page 3-0 , the compressor activates and decreases the amplitude of the input signal. When the sidechain signal level falls below the threshold, the audio input returns to its uncompressed amplitude.

Prepare Audio Files

In this section, you resample and zero-pad a speech file to use as input to the `EnableSidechain` property of your `compressor` object.

Read in an audio signal. Resample it to match the sample rate of the input audio signal (44.1 kHz).

```
targetFs = 44100;
[originalSpeech,originalFs] = audioread('Rainbow-16-8-mono-114secs.wav');
resampledSpeech = resample(originalSpeech,targetFs,originalFs);
```

Pad the beginning of the resampled signal with 10 seconds worth of zeros. This allows the input audio signal to be clearly heard before any compression is applied.

```
resampledSpeech = [zeros(10*targetFs,1);resampledSpeech];
```

Normalize the amplitude to avoid potential clipping.

```
resampledSpeech = resampledSpeech ./ max(resampledSpeech);
```

Write the resampled, zero-padded, and normalized sidechain signal to a file.

```
audiowrite('resampledSpeech.wav',resampledSpeech,targetFs);
```

Construct Audio Objects

Construct a `dsp.AudioFileReader` object for the input and sidechain signals. Using the "ReadRange" property of the `AudioFileReader`, select the second verse of the input signal and the first 26.5 seconds of the sidechain signal for playback. To allow the script to run indefinitely, change the `playbackCount` variable from `1` to `Inf`.

```
inputAudio = 'SoftGuitar-44p1_mono-10mins.ogg';
sidechainAudio  = 'resampledSpeech.wav';
playbackCount = 1;
inputAudioAFR = dsp.AudioFileReader(inputAudio,'PlayCount',playbackCount,'ReadRange',...
    [115*targetFs round(145.4*targetFs)]);
sidechainAudioAFR = dsp.AudioFileReader(sidechainAudio,'PlayCount',playbackCount,...
    'ReadRange',[1 round(26.5*targetFs)]);
```

Construct a `compressor` object. Use a high "Ratio" on page 3-0 , a fast "AttackTime" on page 3-0 , and a moderately slow "ReleaseTime" on page 3-0 . These settings are ideal for voice-over work. The fast attack time ensures that the input audio is compressed almost immediately after the sidechain signal surpasses the compressor threshold. The slow release time ensures the compression on the input audio lasts through any potential short silent regions in the sidechain signal.

```
iAmYourCompressor = compressor('EnableSidechain',true,...
                'SampleRate',targetFs,...
                'Threshold',-40,...
                'Ratio',8,...
                'AttackTime',0.01,...
                'ReleaseTime',1.5);
```

Construct an `audioDeviceWriter` object to play the sidechain and input signals.

```
afw = audioDeviceWriter;
```

Construct a `timescope` object to view the uncompressed input signal, the sidechain signal, as well as the compressed input signal.

```
scope = timescope('NumInputPorts',3,...
                      'SampleRate',targetFs,...
                      'TimeSpanSource','property',...
                      'TimeSpan',5,...
                       'TimeDisplayOffset',0,...
                      'LayoutDimensions',[3 1],...
                      'BufferLength',targetFs*15,...
                      'TimeSpanOverrunAction','Scroll',...
                      'YLimits',[-1 1],...
                      'ShowGrid',true,...
                      'Title','Uncompressed Input Audio - Guitar');
scope.ActiveDisplay = 2;
scope.YLimits = [-1 1];
scope.Title = 'Sidechain Audio - Speech';
scope.ShowGrid = true;
```

```
scope.ActiveDisplay = 3;
scope.YLimits = [-1 1];
scope.ShowGrid = true;
scope.Title = 'Compressed Input Audio - Guitar';
```

Create Audio Streaming Loop

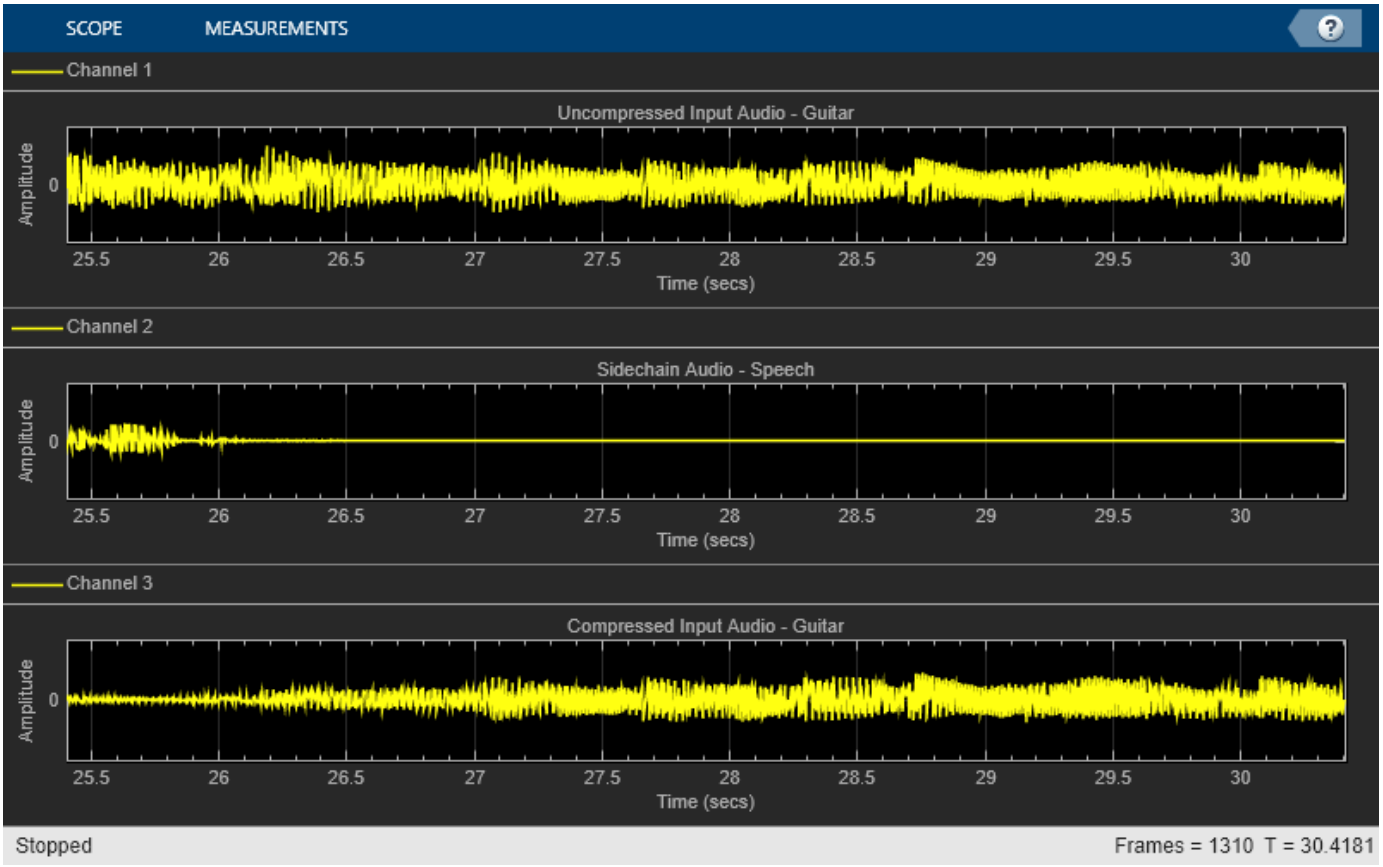
Read in a frame of audio from your input and sidechain signals. Process your input and sidechain signals with your `compressor` object. Playback your processed audio signals and display the audio data using a `timescope` object.

The top panel of your `timescope` displays the uncompressed input audio signal and the middle panel displays the sidechain audio signal. The bottom panel displays the compressed input audio signal. Notice the amplitudes of the signals in the top and bottom panels are identical until the sidechain signal begins. Once the sidechain signal activates, the amplitude of the signal in the bottom panel is compressed. Once the sidechain signal ends, the amplitude of the signal in the bottom panel begins to return to its uncompressed level.

```
while ~isDone(inputAudioAFR)
   inputAudioFrame = inputAudioAFR();
   sideChainAudioFrame = sidechainAudioAFR();
   compressorOutput  = iAmYourCompressor(inputAudioFrame,sideChainAudioFrame);
   afw(sideChainAudioFrame+compressorOutput);
   scope(inputAudioFrame,sideChainAudioFrame,compressorOutput);
end
```

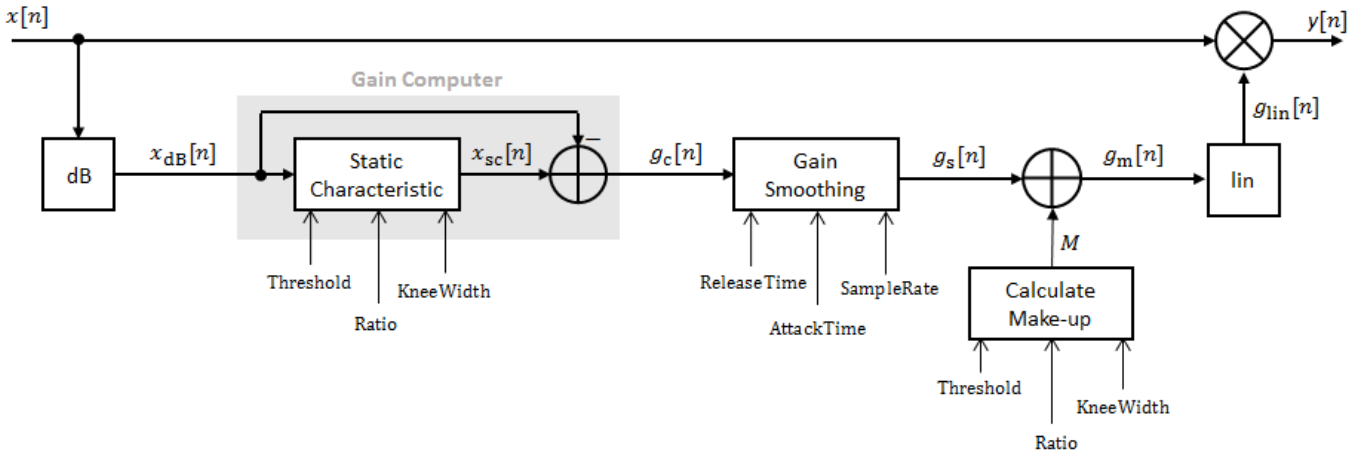
Release your objects.

```
release(inputAudioAFR)
release(sidechainAudioAFR)
release(iAmYourCompressor)
release(afw)
release(scope)
```



Algorithms

The `compressor` System object processes a signal frame by frame and element by element.



Convert Input Signal to dB

The N-point signal, $x[n]$, is converted to decibels:

$$x_{dB}[n] = 20 \times \log_{10}|x[n]|$$

Gain Computer

$x_{\mathrm{dB}}[n]$ passes through the gain computer. The gain computer uses the static characteristic properties of the dynamic range compressor to attenuate gain that is above the threshold.

If you specified a soft knee, the gain computer has the following static characteristic:

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

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:

$$x_{\mathrm{sc}}(x_{\mathrm{dB}}) = \begin{cases} x_{\mathrm{dB}} & x_{\mathrm{dB}} < T \\ T + \frac{(x_{\mathrm{dB}} - T)}{R} & x_{\mathrm{dB}} \ge T \end{cases}$$

The computed gain, $g_{\mathrm{c}}[n]$, is calculated as

$$g_{\mathrm{c}}[n] = x_{\mathrm{sc}}[n] - x_{\mathrm{dB}}[n]\,.$$

Gain Smoothing

$g_{\mathrm{c}}[n]$ is smoothed using specified attack and release time properties:

$$g_{\mathrm{s}}[n] = \begin{cases} \alpha_{\mathrm{A}} g_{\mathrm{s}}[n-1] + (1 - \alpha_{\mathrm{A}}) g_{\mathrm{c}}[n], & g_{\mathrm{c}}[n] \le g_{\mathrm{s}}[n-1] \\ \alpha_{\mathrm{R}} g_{\mathrm{s}}[n-1] + (1 - \alpha_{\mathrm{R}}) g_{\mathrm{c}}[n], & g_{\mathrm{c}}[n] > g_{\mathrm{s}}[n-1] \end{cases}$$

The attack time coefficient, α_{A} , is calculated as

$$\alpha_{\mathrm{A}} = \exp\left(\frac{-\log(9)}{Fs \times T_{\mathrm{A}}}\right).$$

The release time coefficient, α_{R} , is calculated as

$$\alpha_{\mathrm{R}} = \exp\left(\frac{-\log(9)}{Fs \times T_{\mathrm{R}}}\right).$$

T_A is the attack time period, specified by the `AttackTime` property. T_{R} is the release time period, specified by the `ReleaseTime` property. Fs is the input sampling rate, specified by the `SampleRate` property.

Calculate and Apply Make-up Gain

If `MakeUpGainMode` is set to the default `'Auto'`, the make-up gain is calculated as the negative of the computed gain for a 0 dB input,

$$M = -x_{\mathrm{sc}}\big|_{x_{\mathrm{dB}} = 0}\,.$$

Given a steady-state input of 0 dB, this configuration achieves a steady-state output of 0 dB. The make-up gain is determined by the `Threshold`, `Ratio`, and `KneeWidth` properties. It does not depend on the input signal.

The make-up gain, M, is added to the smoothed gain, $g_s[n]$:

$$g_m[n] = g_s[n] + M$$

Calculate and Apply Linear Gain

The calculated gain in dB, $g_m[n]$, is translated to a linear domain:

$$g_{lin}[n] = 10^{\left(\frac{g_m[n]}{20}\right)}$$

The output of the dynamic range compressor is given as

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

Version History

Introduced in R2016a

References

[1] Giannoulis, Dimitrios, Michael Massberg, and Joshua D. Reiss. "Digital Dynamic Range Compressor Design –– A Tutorial and Analysis." *Journal of Audio Engineering Society*. Vol. 60, Issue 6, 2012, pp. 399–408.

Extended Capabilities

C/C++ Code Generation

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

Usage notes and limitations:

“System Objects in MATLAB Code Generation” (MATLAB Coder)

See Also

`noiseGate` | `expander` | `limiter` | Compressor

Topics

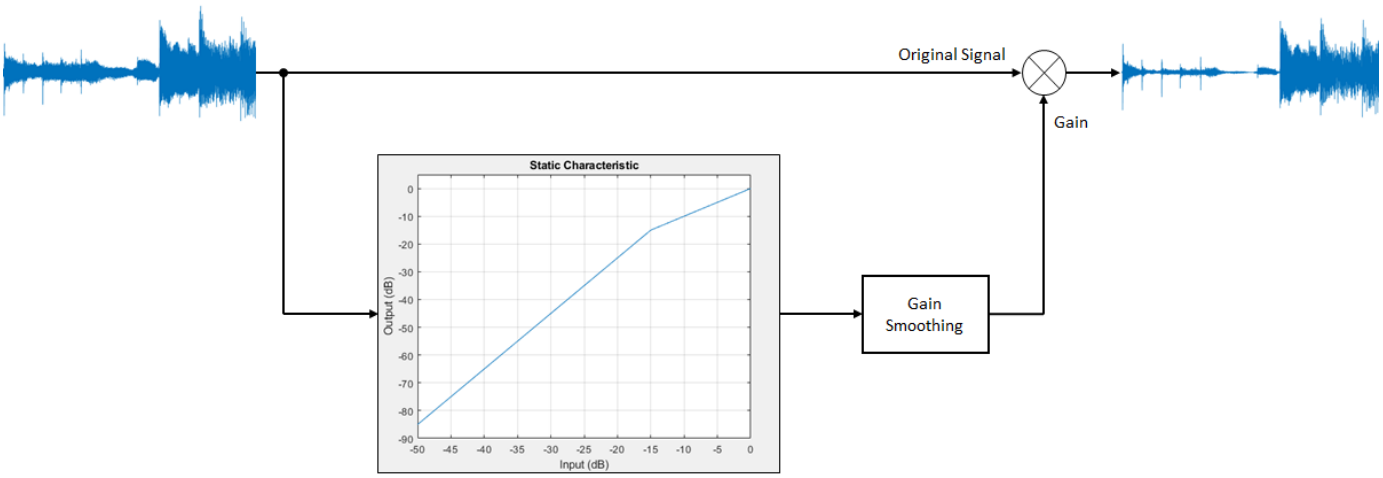
“Dynamic Range Control”

expander

Dynamic range expander

Description

The `expander` System object performs dynamic range expansion independently across each input channel. Dynamic range expansion attenuates the volume of quiet sounds below a given threshold. It uses specified attack, release, and hold times to achieve a smooth applied gain curve. Properties of the `expander` System object specify the type of dynamic range expansion.



To perform dynamic range expansion:

1. Create the `expander` object and set its properties.
2. Call the object with arguments, as if it were a function.

To learn more about how System objects work, see What Are System Objects?

Creation

Syntax

```
dRE = expander
dRE = expander(thresholdValue)
dRE = expander(thresholdValue,ratioValue)
dRE = expander(___,Name,Value)
```

Description

`dRE = expander` creates a System object, `dRE`, that performs dynamic range expansion independently across each input channel.

`dRE = expander(thresholdValue)` sets the Threshold property to `thresholdValue`.

`dRE = expander(thresholdValue,ratioValue)` sets the Ratio property to `ratioValue`.

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

Example: `dRE = expander('AttackTime',0.01,'SampleRate',16000)` creates a System object, `dRE`, with a 0.01 second attack time and a 16 kHz sample rate.

Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see System Design in MATLAB Using System Objects.

`Threshold` — Operation threshold (dB)

`−10` (default) | real scalar

Operation threshold in dB, specified as a real scalar.

Operation threshold is the level below which gain is applied to the input signal.

Tunable: Yes

Data Types: `single` | `double`

`Ratio` — Expansion ratio

`5` (default) | real scalar

Expansion ratio, specified as a real scalar greater than or equal to 1.

Expansion ratio is the input/output ratio for signals that undershoot the operation threshold.

Assuming a hard knee characteristic and a steady-state input such that $x[n]$ dB < `thresholdValue`, the expansion ratio is defined as $R = \frac{(y[n] - T)}{(x[n] - T)}$.

- R is the expansion ratio.
- $y[n]$ is the output signal in dB.
- $x[n]$ is the input signal in dB.
- T is the threshold in dB.

Tunable: Yes

Data Types: `single` | `double`

`KneeWidth` — Knee width (dB)

`0` (default) | real scalar

Knee width in dB, specified as a real scalar greater than or equal to 0.

Knee width is the transition area in the expansion characteristic.

For soft knee characteristics, the transition area is defined by the relation

$$y = x + \frac{(1 - R) \times \left(x - T - \frac{W}{2}\right)^2}{(2 \times W)}$$

for the range $(2 \times |x - T|) \le W$.

- 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

Data Types: `single` | `double`

`AttackTime` — Attack time (s)
`0.05` (default) | real scalar

Attack time in seconds, specified as a real scalar greater than or equal to 0.

Attack time is the time it takes the expander gain to rise from 10% to 90% of its final value when the input goes below the threshold.

Tunable: Yes

Data Types: `single` | `double`

`ReleaseTime` — Release time (s)
`0.2` (default) | real scalar

Release time in seconds, specified as a real scalar greater than or equal to 0.

Release time is the time it takes the expander gain to drop from 90% to 10% of its final value when the input goes above the threshold.

Tunable: Yes

Data Types: `single` | `double`

`HoldTime` — Hold time (s)
`0.05` (default) | real scalar

Hold time in seconds, specified as a real scalar greater than or equal to 0.

Hold time is the period for which the (negative) gain is held before starting to decrease towards its steady state value when the input level drops below the threshold.

Tunable: Yes

Data Types: `single` | `double`

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

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

Tunable: Yes

Data Types: `single` | `double`

`EnableSidechain` — Enable sidechain input
`false` (default) | `true`

Enable sidechain input, specified as `true` or `false`. This property determines the number of available inputs on the `expander` object.

- `false` –– Sidechain input is disabled and the `expander` object accepts one input: the `audioIn` data to be expanded.
- `true` –– Sidechain input is enabled and the `expander` object accepts two inputs: the `audioIn` data to be expanded and the sidechain input used to compute the expander `gain`.

The sidechain datatype and (frame) length must be the same as `audioIn`.

The number of channels of the sidechain must be equal to the number of channels of `audioIn` or be equal to one. When the number of sidechain channels is one, the `gain` computed based on this channel is applied to all channels of `audioIn`. When the number of sidechain channels is equal to the number of channels in `audioIn`, the `gain` computed for each sidechain channel is applied to the corresponding channel of `audioIn`.

Tunable: No

Usage

Syntax

```
audioOut = dRE(audioIn)
[audioOut,gain] = dRE(audioIn)
```

Description

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

`[audioOut,gain] = dRE(audioIn)` also returns the applied gain, in dB, at each input sample.

Input Arguments

`audioIn` — Audio input to expander
matrix

Audio input to the expander, specified as a matrix. The columns of the matrix are treated as independent audio channels.

Data Types: `single` | `double`

Output Arguments

`audioOut` — Audio output from expander

matrix

Audio output from the expander, returned as a matrix the same size as `audioIn`.

Data Types: `single` | `double`

`gain` — Gain applied by expander (dB)

matrix

Gain applied by expander, returned as a matrix the same size as `audioIn`.

Data Types: `single` | `double`

Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

Specific to expander

visualize	Visualize static characteristic of dynamic range controller
staticCharacteristic	Return static characteristic of dynamic range controller
createAudioPluginClass	Create audio plugin class that implements functionality of System object
parameterTuner	Tune object parameters while streaming

MIDI

configureMIDI	Configure MIDI connections between audio object and MIDI controller
disconnectMIDI	Disconnect MIDI controls from audio object
getMIDIConnections	Get MIDI connections of audio object

Common to All System Objects

clone	Create duplicate System object
isLocked	Determine if System object is in use
release	Release resources and allow changes to System object property values and input characteristics
reset	Reset internal states of System object
step	Run System object algorithm

The `createAudioPluginClass` and `configureMIDI` functions map tunable properties of the `expander` System object to user-facing parameters:

Property	Range	Mapping	Unit
`Threshold`	[–140, 0]	linear	dB
`Ratio`	[1, 50]	linear	none
`KneeWidth`	[0, 20]	linear	dB
`AttackTime`	[0, 4]	linear	seconds

Property	Range	Mapping	Unit
`ReleaseTime`	[0, 4]	linear	seconds
`HoldTime`	[0, 4]	linear	seconds

Examples

Expand Audio Signal

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

Set up the `dsp.AudioFileReader` and `audioDeviceWriter` System objects.

```
frameLength = 1024;
fileReader = dsp.AudioFileReader( ...
    'Filename','Counting-16-44p1-mono-15secs.wav', ...
    'SamplesPerFrame',frameLength);
deviceWriter = audioDeviceWriter( ...
    'SampleRate',fileReader.SampleRate);
```

Corrupt the audio signal with Gaussian noise. Play the audio.

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

release(fileReader)
```

Set up the expander with a threshold of -40 dB, a ratio of 10, an attack time of 0.01 seconds, a release time of 0.02 seconds, and a hold time of 0 seconds. Use the sample rate of your audio file reader.

```
dRE = expander(-40,10, ...
    'AttackTime',0.01, ...
    'ReleaseTime',0.02, ...
    'HoldTime',0, ...
    'SampleRate',fileReader.SampleRate);
```

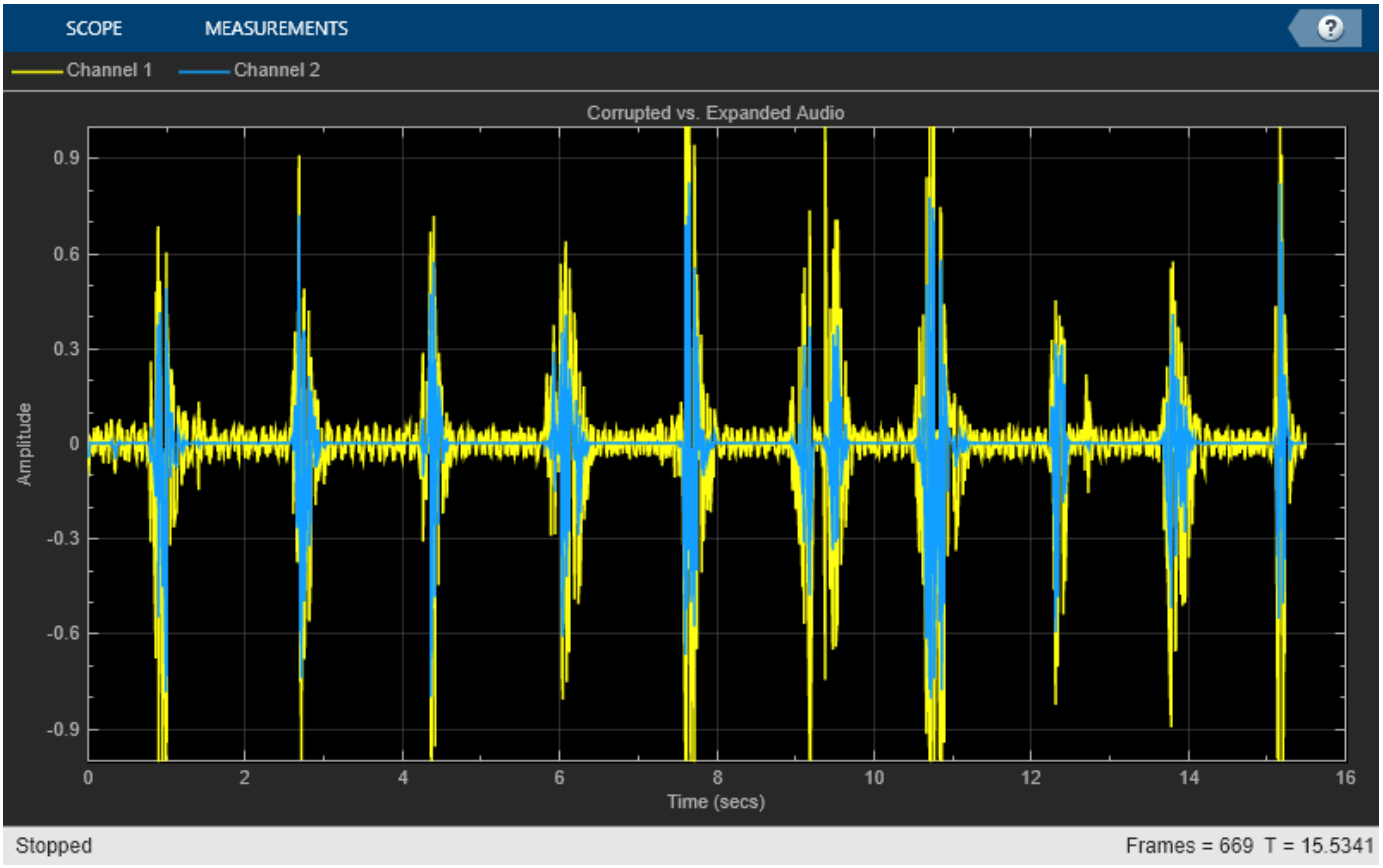
Set up the scope to visualize the signal before and after dynamic range expansion.

```
scope = timescope( ...
    'SampleRate',fileReader.SampleRate, ...
    'TimeSpanOverrunAction','Scroll', ...
    'TimeSpanSource','property','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

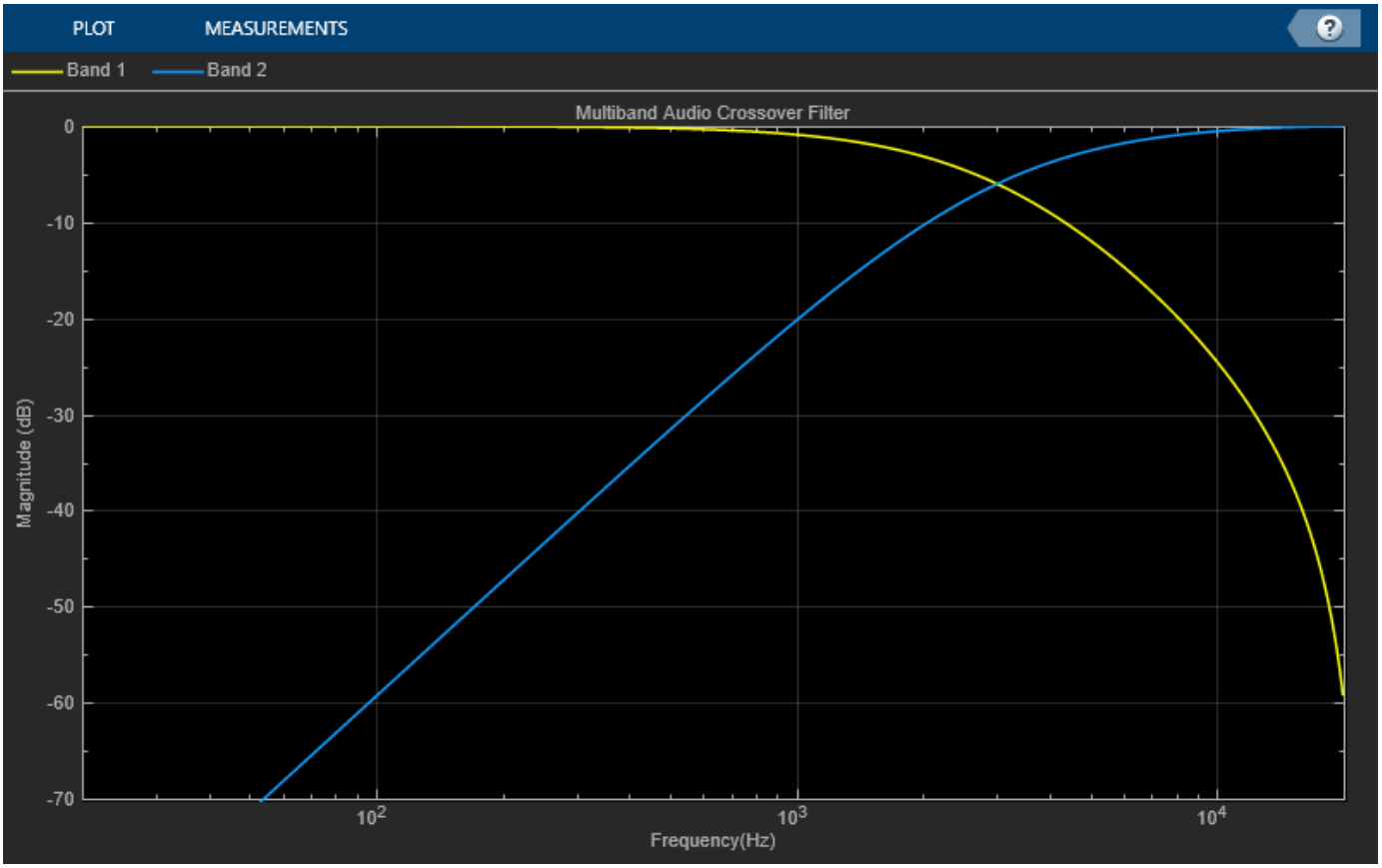
De-essing is the process of diminishing sibilant sounds in an audio signal. Sibilance refers to the *s*, *z*, and *sh* sounds in speech, which can be disproportionately emphasized during recording. *es* sounds fall under the category of unvoiced speech with all consonants and have a higher frequency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

```
scope = timescope( ...
    'SampleRate',fileReader.SampleRate, ...
    'TimeSpanOverrunAction','Scroll', ...
    'TimeSpanSource','Property','TimeSpan',4, ...
    'BufferLength',fileReader.SampleRate*8, ...
    'YLimits',[-1 1], ...
    'ShowGrid',true, ...
    'ShowLegend',true, ...
    'ChannelNames',{'Original','Processed'});
```

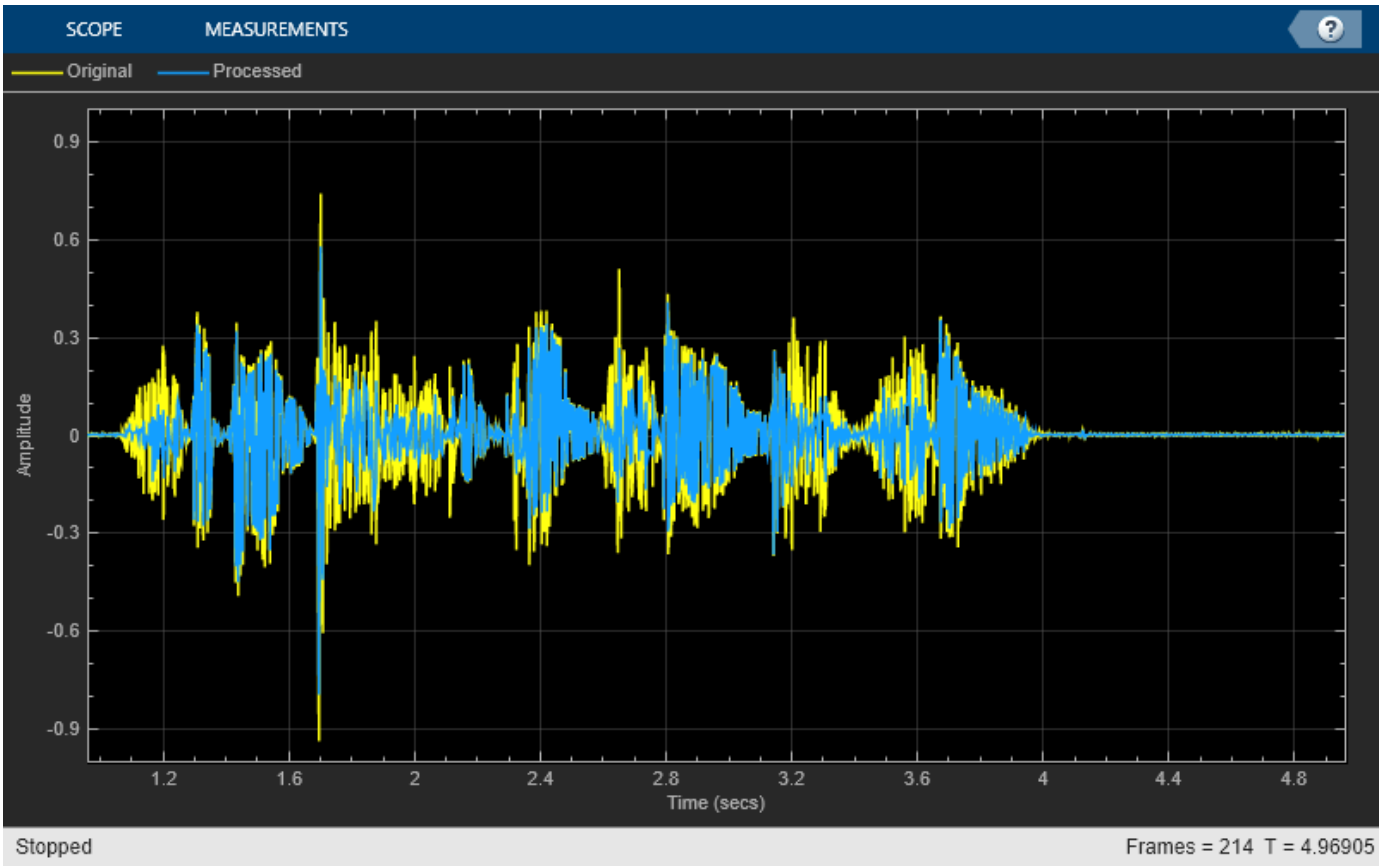
In an audio stream loop:

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)
```

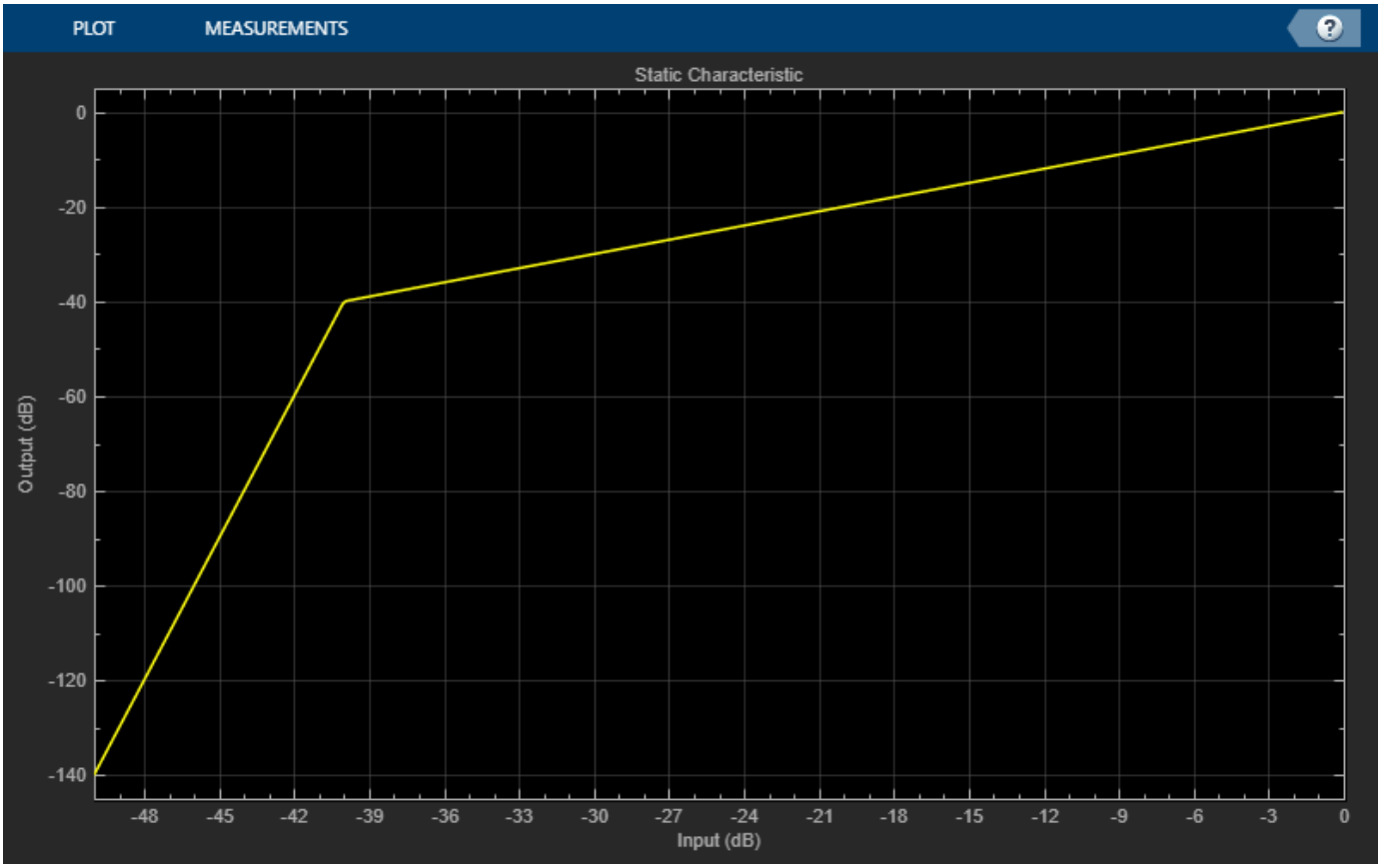



```
release(crossFilt)
release(dRExpander)
```

Tune Expander Parameters

Create a `dsp.AudioFileReader` to read in audio frame-by-frame. Create a `audioDeviceWriter` to write audio to your sound card. Create a `expander` to process the audio data. Call `visualize` to plot the static characteristic of the `expander`.

```
frameLength = 1024;
fileReader = dsp.AudioFileReader('Counting-16-44p1-mono-15secs.wav', ...
    'SamplesPerFrame',frameLength);
deviceWriter = audioDeviceWriter('SampleRate',fileReader.SampleRate);

dRE = expander(-40,10, ...
    'AttackTime',0.01, ...
    'ReleaseTime',0.02, ...
    'HoldTime',0, ...
    'SampleRate',fileReader.SampleRate);
visualize(dRE)
```

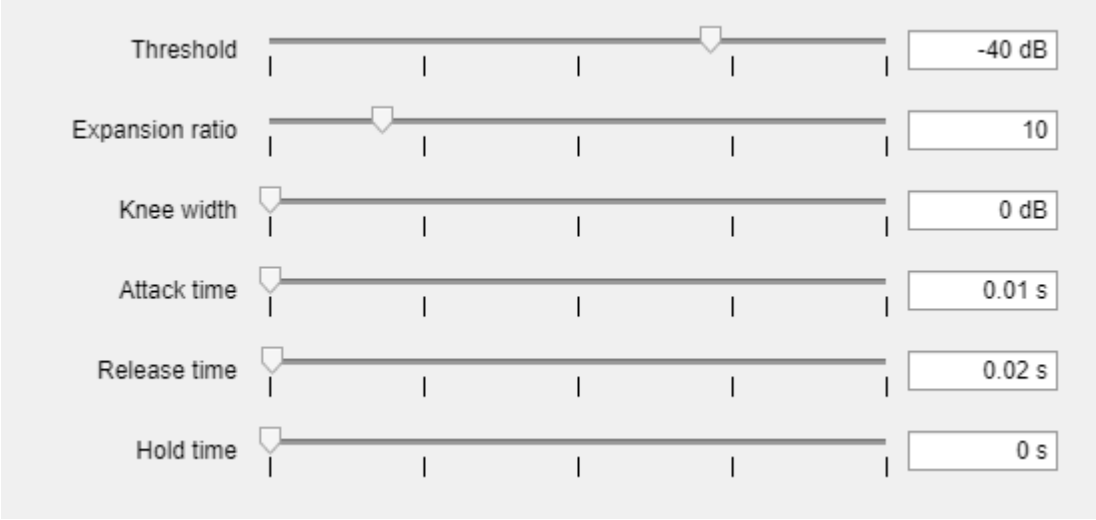


Create a `timescope` to visualize the original and processed audio.

```
scope = timescope( ...
    'SampleRate',fileReader.SampleRate, ...
    'TimeSpanSource','property','TimeSpan',1, ...
    'BufferLength',fileReader.SampleRate*4, ...
    'YLimits',[-1,1], ...
    'TimeSpanOverrunAction','Scroll', ...
    'ShowGrid',true, ...
    'LayoutDimensions',[2,1], ...
    'NumInputPorts',2, ...
    'Title','Original vs. Processed Audio (top) and Applied Gain in dB (bottom)');
scope.ActiveDisplay = 2;
scope.YLimits = [-300,0];
scope.YLabel = 'Gain (dB)';
```

Call `parameterTuner` to open a UI to tune parameters of the expander while streaming.

```
parameterTuner(dRE)
```



In an audio stream loop:

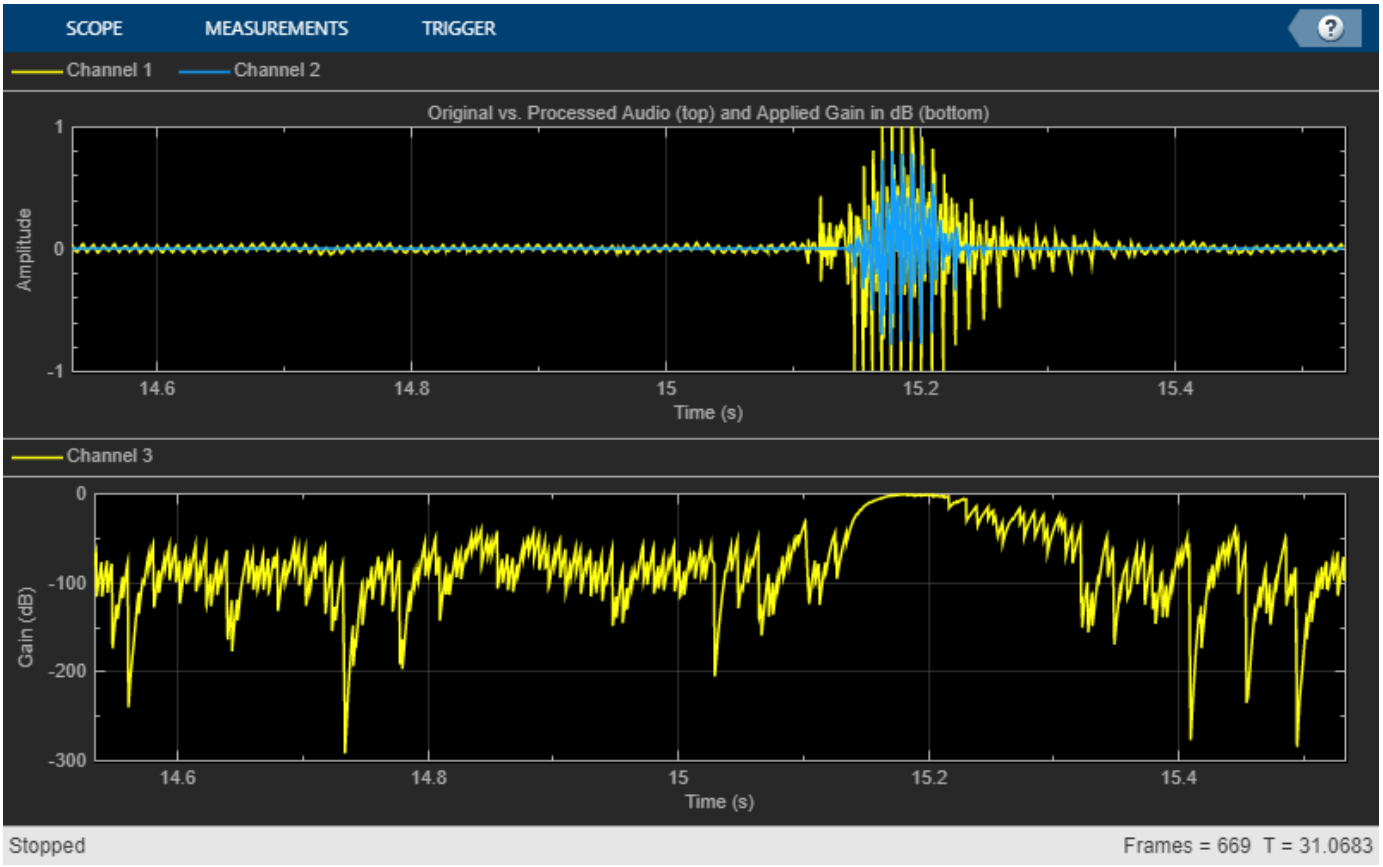
1. Read in a frame of audio from the file.
2. Apply dynamic range expansion.
3. Write the frame of audio to your audio device for listening.
4. Visualize the original and processed audio, and the gain applied.

While streaming, tune parameters of the dynamic range expander and listen to the effect.

```
while ~isDone(fileReader)
    audioIn = fileReader();
    [audioOut,g] = dRE(audioIn);
    deviceWriter(audioOut);
    scope([audioIn(:,1),audioOut(:,1)],g(:,1));
    drawnow limitrate % required to update parameter
end
```

As a best practice, release your objects when done.

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



Sidechain Dynamic Range Expansion

Use the "EnableSidechain" on page 3-0 input of an `expander` object to emulate an electronic drum controller, also known as a *multipad*. This technique is common in recording studio production and creates interesting changes to the timbre of an instrument. The sidechain signal controls the expansion on the input signal. Sidechain expansion decreases the amplitude of the input signal when the sidechain signal falls below the "Threshold" on page 3-0 of the `expander`.

Prepare Audio Files

Convert the sidechain signal from stereo to mono.

```
[expanderSideChainStereo,Fs] = audioread('FunkyDrums-44p1-stereo-25secs.mp3');
expanderSideChainMono = (expanderSideChainStereo(:,1) + expanderSideChainStereo(:,2)) / 2;
```

Write the converted sidechain signal to a file.

```
audiowrite('convertedSidechainSig.wav',expanderSideChainMono,Fs);
```

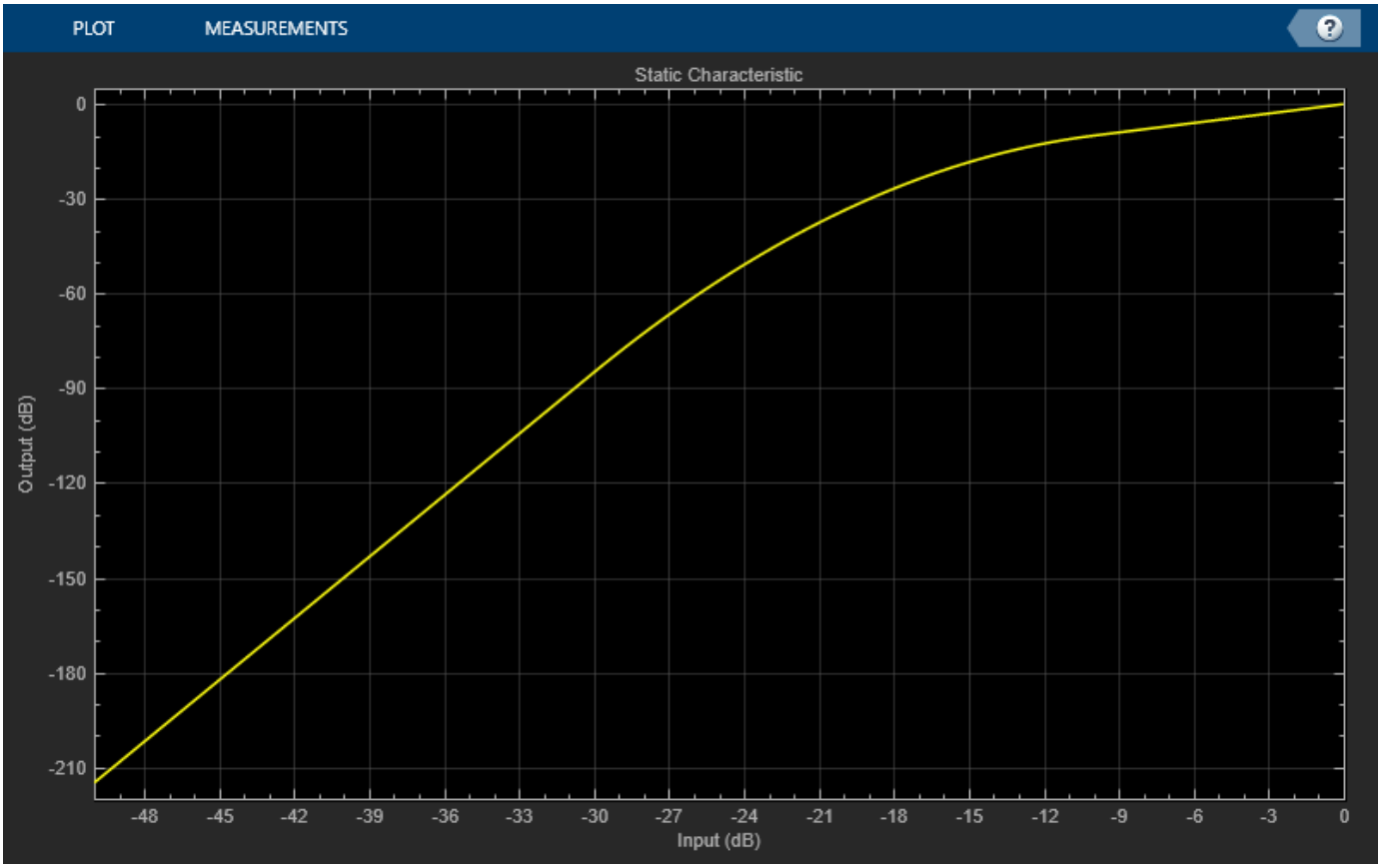
Construct Audio Objects

Construct a `dsp.AudioFileReader` object for the input and sidechain signals. To allow the script to run indefinitely, change the `playbackCount` variable from `1` to `Inf`.

```
inputAudio = 'SoftGuitar-44p1_mono-10mins.ogg';
sidechainAudio  = 'convertedSidechainSig.wav';
playbackCount = 1;
inputAudioAFR = dsp.AudioFileReader(inputAudio,'PlayCount',playbackCount);
sidechainAudioAFR = dsp.AudioFileReader(sidechainAudio,'PlayCount',playbackCount);
```

Construct and visualize an `expander` object. Use a high “Ratio” on page 3-0 , a soft “KneeWidth” on page 3-0 , a fast “AttackTime” on page 3-0 and “ReleaseTime” on page 3-0 , and a short “HoldTime” on page 3-0 .

```
dRE = expander('EnableSidechain',true,'Threshold',-20,'Ratio',6.5,...
    'KneeWidth',20,'AttackTime',0.84,'ReleaseTime',0.001,'HoldTime',0.0001);
visualize(dRE)
```



Construct an `audioDeviceWriter` object to play the sidechain and input signals.

```
afw = audioDeviceWriter;
```

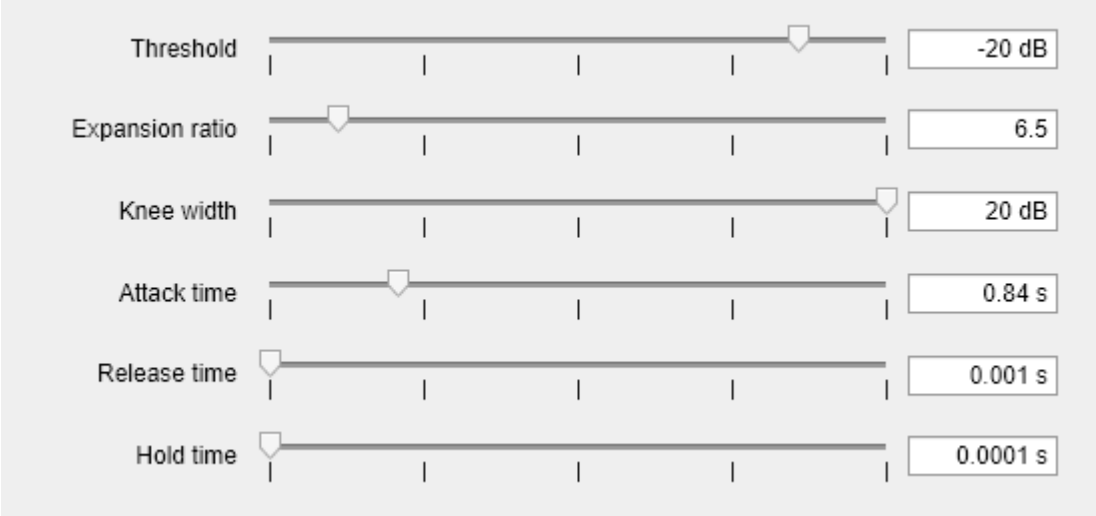
Construct a `timescope` object to view the input signal, the sidechain signal, as well as the expanded input signal.

```
scope = timescope('NumInputPorts',3,...
                  'SampleRate',Fs,...
                  'TimeSpanSource','property',...
                  'TimeSpan',5,...
                   'TimeDisplayOffset',0,...
                  'LayoutDimensions',[3 1],...
```

```
                        'BufferLength',Fs*15,...
                        'TimeSpanOverrunAction','Scroll',...
                        'YLimits',[-1 1],...
                        'ShowGrid',true,...
                        'Title','Input Audio - Classical Guitar');
scope.ActiveDisplay = 2;
scope.YLimits = [-1 1];
scope.Title = 'Sidechain Audio - Drums';
scope.ShowGrid = true;
scope.ActiveDisplay = 3;
scope.YLimits = [-1 1];
scope.ShowGrid = true;
scope.Title = 'Expanded Input Audio - Classical Guitar';
```

Call `parameterTuner` to open a UI to tune parameters of the expander while streaming. Adjust the property values and listen to the effect in real time.

```
parameterTuner(dRE)
```



Create Audio Streaming Loop

Read in a frame of audio from your input and sidechain signals. Process your input and sidechain signals with your `expander` object. Playback your processed audio signals and display the audio data using a `timescope` object.

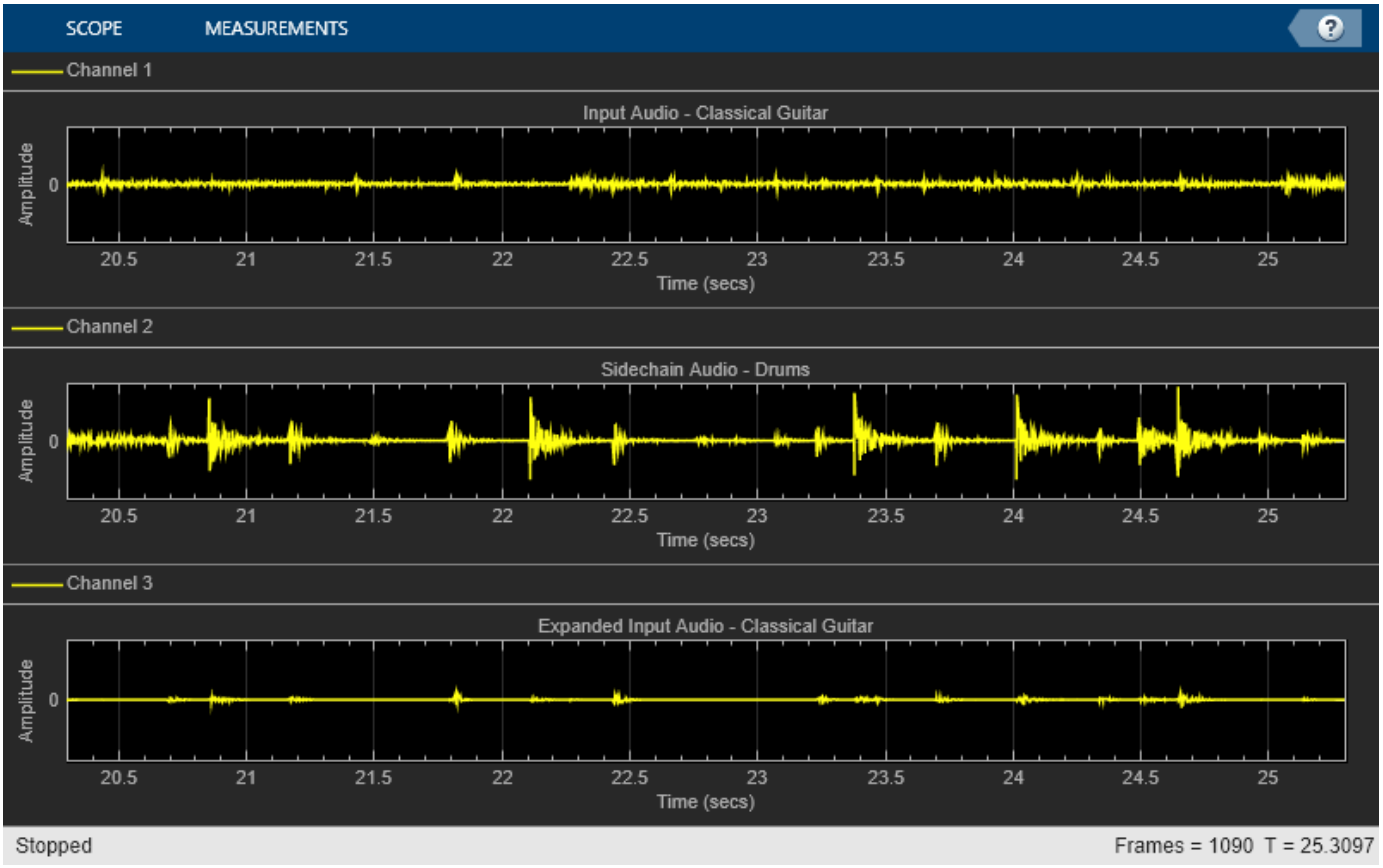
The top panel of your `timescope` displays the input audio signal and the middle panel displays the sidechain audio signal. The bottom panel displays the expanded input audio signal.

Substitute different audio files for your `inputAudio` variable to create different textures and timbres in your drum mix.

```
while ~isDone(sidechainAudioAFR)
   inputAudioFrame = inputAudioAFR();
   sideChainAudioFrame = sidechainAudioAFR();
   expanderOutput  = dRE(inputAudioFrame,sideChainAudioFrame);
   afw(sideChainAudioFrame+expanderOutput);
   scope(inputAudioFrame,sideChainAudioFrame,expanderOutput);
   drawnow limitrate;   % required to update parameter settings from UI
end
```

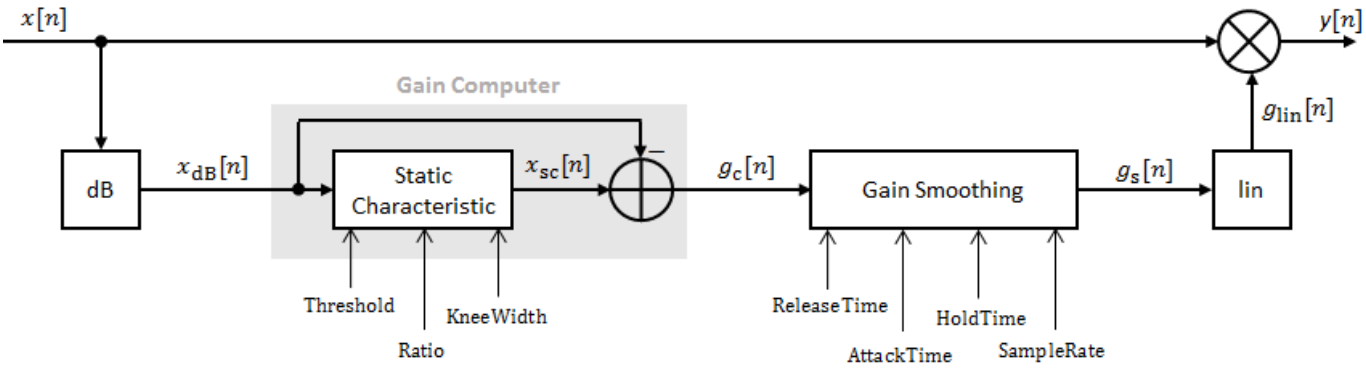
Release your objects.

```
release(inputAudioAFR)
release(sidechainAudioAFR)
release(dRE)
release(afw)
release(scope)
```



Algorithms

The `expander` System object processes a signal frame by frame and element by element.



Convert Input Signal to dB

The N-point signal, $x[n]$, is converted to decibels:

$$x_{\mathrm{dB}}[n] = 20 \times \log_{10}|x[n]|$$

Gain Computer

$x_{\mathrm{dB}}[n]$ passes through the gain computer. The gain computer uses the static characteristic properties of the dynamic range expander to attenuate gain that is below the threshold.

If you specified a soft knee, the gain computer has the following static characteristic:

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

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:

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

The computed gain, $g_{\mathrm{c}}[n]$, is calculated as

$$g_{\mathrm{c}}[n] = x_{\mathrm{sc}}[n] - x_{\mathrm{dB}}[n]\,.$$

Gain Smoothing

$g_{\mathrm{c}}[n]$ is smoothed using specified attack, release, and hold time properties:

$$g_{\mathrm{s}}[n] = \begin{cases} \alpha_{\mathrm{A}} g_{\mathrm{s}}[n-1] + (1-\alpha_{\mathrm{A}}) g_{\mathrm{c}}[n] & (C_{\mathrm{A}} > T_{\mathrm{H}}) \ \&\ (g_{\mathrm{c}}[n] \le g_{\mathrm{s}}[n-1]) \\ g_{\mathrm{s}}[n-1] & C_{\mathrm{A}} \le T_{\mathrm{H}} \\ \alpha_{\mathrm{R}} g_{\mathrm{s}}[n-1] + (1-\alpha_{\mathrm{R}}) g_{\mathrm{c}}[n] & g_{\mathrm{c}}[n] > g_{\mathrm{s}}[n-1] \end{cases}$$

The attack time coefficient, α_{A} , is calculated as

$$\alpha_{\mathrm{A}} = \exp\left(\frac{-\log(9)}{Fs \times T_{\mathrm{A}}}\right).$$

The release time coefficient, α_{R} , is calculated as

$$\alpha_{\mathrm{R}} = \exp\left(\frac{-\log(9)}{Fs \times T_{\mathrm{R}}}\right).$$

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} is the hold counter for attack. The limit, T_{H} , is determined by the `HoldTime` property.

Calculate and Apply Linear Gain

The smoothed gain in dB, $g_s[n]$, is translated to a linear domain:

$$g_{\text{lin}}[n] = 10^{\left(\frac{g_S[n]}{20}\right)}$$

The output of the dynamic range expander is given as

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

Version History

Introduced in R2016a

References

[1] Giannoulis, Dimitrios, Michael Massberg, and Joshua D. Reiss. "Digital Dynamic Range Compressor Design –– A Tutorial and Analysis." *Journal of Audio Engineering Society*. Vol. 60, Issue 6, 2012, pp. 399–408.

Extended Capabilities

C/C++ Code Generation

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

Usage notes and limitations:

“System Objects in MATLAB Code Generation” (MATLAB Coder)

See Also

`noiseGate` | `compressor` | `limiter` | Expander

Topics

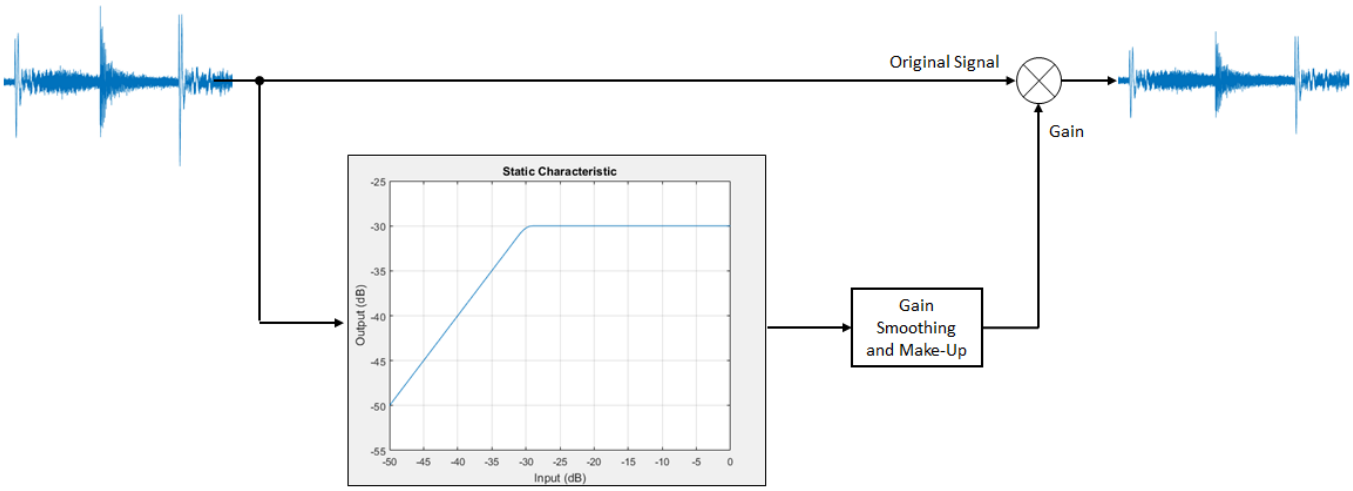
“Dynamic Range Control”

limiter

Dynamic range limiter

Description

The `limiter` System object performs brick-wall dynamic range limiting independently across each input channel. Dynamic range limiting suppresses the volume of loud sounds that cross a given threshold. It uses specified attack and release times to achieve a smooth applied gain curve. Properties of the `limiter` System object specify the type of dynamic range limiting.



To perform dynamic range limiting:

1. Create the `limiter` object and set its properties.
2. Call the object with arguments, as if it were a function.

To learn more about how System objects work, see What Are System Objects?

Creation

Syntax

```
dRL = limiter
dRL = limiter(thresholdValue)
dRL = limiter( ___ ,Name,Value)
```

Description

`dRL = limiter` creates a System object, `dRL`, that performs brick-wall dynamic range limiting independently across each input channel.

`dRL = limiter(thresholdValue)` sets the Threshold property to `thresholdValue`.

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

Example: `dRL = limiter('AttackTime',0.01,'SampleRate',16000)` creates a System object, `dRL`, with a 10 ms attack time and a sample rate of 16 kHz.

Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see System Design in MATLAB Using System Objects.

`Threshold` — Operation threshold (dB)

`−10` (default) | real scalar

Operation threshold in dB, specified as a real scalar.

Operation threshold is the level above which gain is applied to the input signal.

Tunable: Yes

Data Types: `single` | `double`

`KneeWidth` — Knee width (dB)

`0` (default) | real scalar

Knee width in dB, specified as a real scalar greater than or equal to 0.

Knee width is the transition area in the limiter characteristic.

For soft knee characteristics, the transition area is defined by the relation

$$y = x - \frac{\left(x - T + \frac{W}{2}\right)^2}{(2 \times W)}$$

for the range $(2 \times |x - T|) \le W$.

- y is the output level in dB.
- x is the input level in dB.
- T is the threshold in dB.
- W is the knee width in dB.

Tunable: Yes

Data Types: `single` | `double`

`AttackTime` — Attack time (s)

`0` (default) | real scalar

Attack time in seconds, specified as a real scalar greater than or equal to 0.

Attack time is the time it takes the limiter gain to rise from 10% to 90% of its final value when the input goes above the threshold.

Tunable: Yes

Data Types: `single` | `double`

`ReleaseTime` — Release time (s)

`0.2` (default) | real scalar

Release time in seconds, specified as a real scalar greater than or equal to 0.

Release time is the time it takes the limiter gain to drop from 90% to 10% of its final value when the input goes below the threshold.

Tunable: Yes

Data Types: `single` | `double`

`MakeUpGainMode` — Make-up gain mode

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

Make-up gain mode, specified as `'Auto'` or `'Property'`.

- `'Auto'` –– Make-up gain is applied at the output of the dynamic range limiter such that a steady-state 0 dB input has a 0 dB output.
- `'Property'` –– Make-up gain is set to the value specified in the MakeUpGain property.

Tunable: No

Data Types: `char` | `string`

`MakeUpGain` — Make-up gain (dB)

`0` (default) | real scalar

Make-up gain in dB, specified as a real scalar.

Make-up gain compensates for gain lost during limiting. It is applied at the output of the dynamic range limiter.

Tunable: Yes

Dependencies

To enable this property, set MakeUpGainMode to `'Property'`.

Data Types: `single` | `double`

`SampleRate` — Input sample rate (Hz)

`44100` (default) | positive scalar

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

Tunable: Yes

Data Types: `single` | `double`

`EnableSidechain` — Enable sidechain input

`false` (default) | `true`

Enable sidechain input, specified as `true` or `false`. This property determines the number of available inputs on the `limiter` object.

- `false` –- Sidechain input is disabled and the `limiter` object accepts one input: the `audioIn` data to be limited.
- `true` –- Sidechain input is enabled and the `limiter` object accepts two inputs: the `audioIn` data to be limited and the sidechain input used to compute the limiter `gain`.

The sidechain datatype and (frame) length must be the same as `audioIn`.

The number of channels of the sidechain input must be equal to the number of channels of `audioIn` or be equal to one. When the number of sidechain channels is one, the `gain` computed based on this channel is applied to all channels of `audioIn`. When the number of sidechain channels is equal to the number of channels in `audioIn`, the `gain` computed for each sidechain channel is applied to the corresponding channel of `audioIn`.

Tunable: No

Usage

Syntax

```
audioOut = dRL(audioIn)
[audioOut,gain] = dRL(audioIn)
```

Description

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

`[audioOut,gain] = dRL(audioIn)` also returns the applied gain, in dB, at each input sample.

Input Arguments

`audioIn` — Audio input to limiter

matrix

Audio input to the limiter, specified as a matrix. The columns of the matrix are treated as independent audio channels.

Data Types: `single` | `double`

Output Arguments

`audioOut` — Audio output from limiter

matrix

Audio output from the limiter, returned as a matrix the same size as `audioIn`.

Data Types: `single` | `double`

`gain` — Gain applied by limiter (dB)
matrix

Gain applied by the limiter, returned as a matrix the same size as `audioIn`.

Data Types: `single` | `double`

Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

Specific to `limiter`

visualize	Visualize static characteristic of dynamic range controller
staticCharacteristic	Return static characteristic of dynamic range controller
createAudioPluginClass	Create audio plugin class that implements functionality of System object
parameterTuner	Tune object parameters while streaming

MIDI

configureMIDI	Configure MIDI connections between audio object and MIDI controller
disconnectMIDI	Disconnect MIDI controls from audio object
getMIDIConnections	Get MIDI connections of audio object

Common to All System Objects

clone	Create duplicate System object
isLocked	Determine if System object is in use
release	Release resources and allow changes to System object property values and input characteristics
reset	Reset internal states of System object
step	Run System object algorithm

The `createAudioPluginClass` and `configureMIDI` functions map tunable properties of the `limiter` System object to user-facing parameters:

Property	Range	Mapping	Unit
`Threshold`	[–50, 0]	linear	dB
`KneeWidth`	[0, 20]	linear	dB
`AttackTime`	[0, 4]	linear	seconds
`ReleaseTime`	[0, 4]	linear	seconds
`MakeUpGain` (available when you set `MakeUpGainMode` to `'Property'`)	[–10, 24]	linear	dB

Examples

Limit Audio Signal

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

Set up the `dsp.AudioFileReader` and `audioDeviceWriter` System objects™.

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

Set up the `limiter` to have a threshold of -15 dB, an attack time of 0.005 seconds, and a release time of 0.1 seconds. Set make-up gain to 0 dB (default). To specify this value, set the make-up gain mode to `'Property'` but do not specify the `MakeUpGain` property. Use the sample rate of your audio file reader.

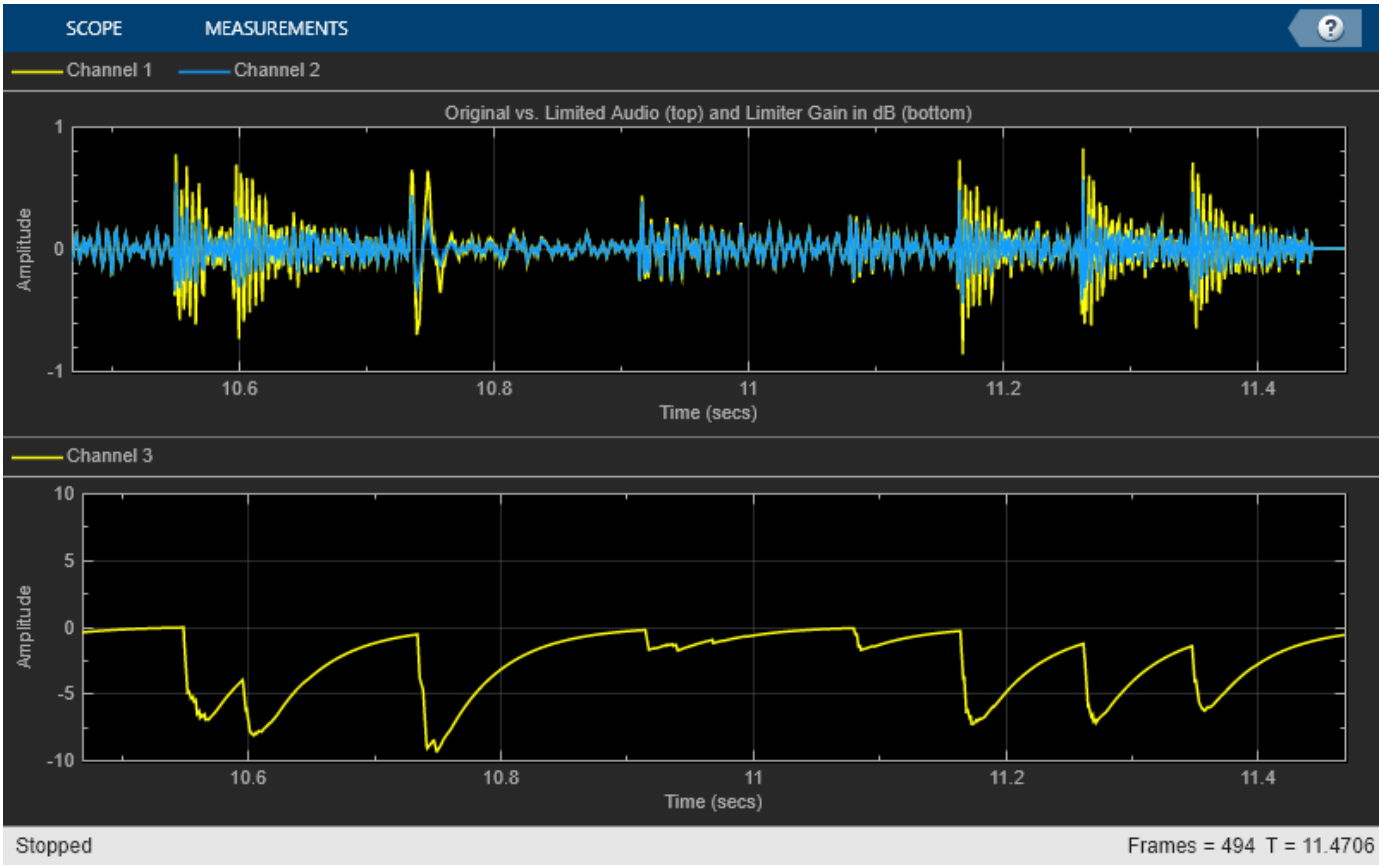
```
dRL = limiter(-15, ...
    'AttackTime',0.005, ...
    'ReleaseTime',0.1, ...
    'MakeUpGainMode','Property', ...
    'SampleRate',fileReader.SampleRate);
```

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

```
scope = timescope( ...
    'SampleRate',fileReader.SampleRate, ...
    'TimeSpanOverrunAction','Scroll', ...
    'TimeSpanSource','property',...
    '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);
```

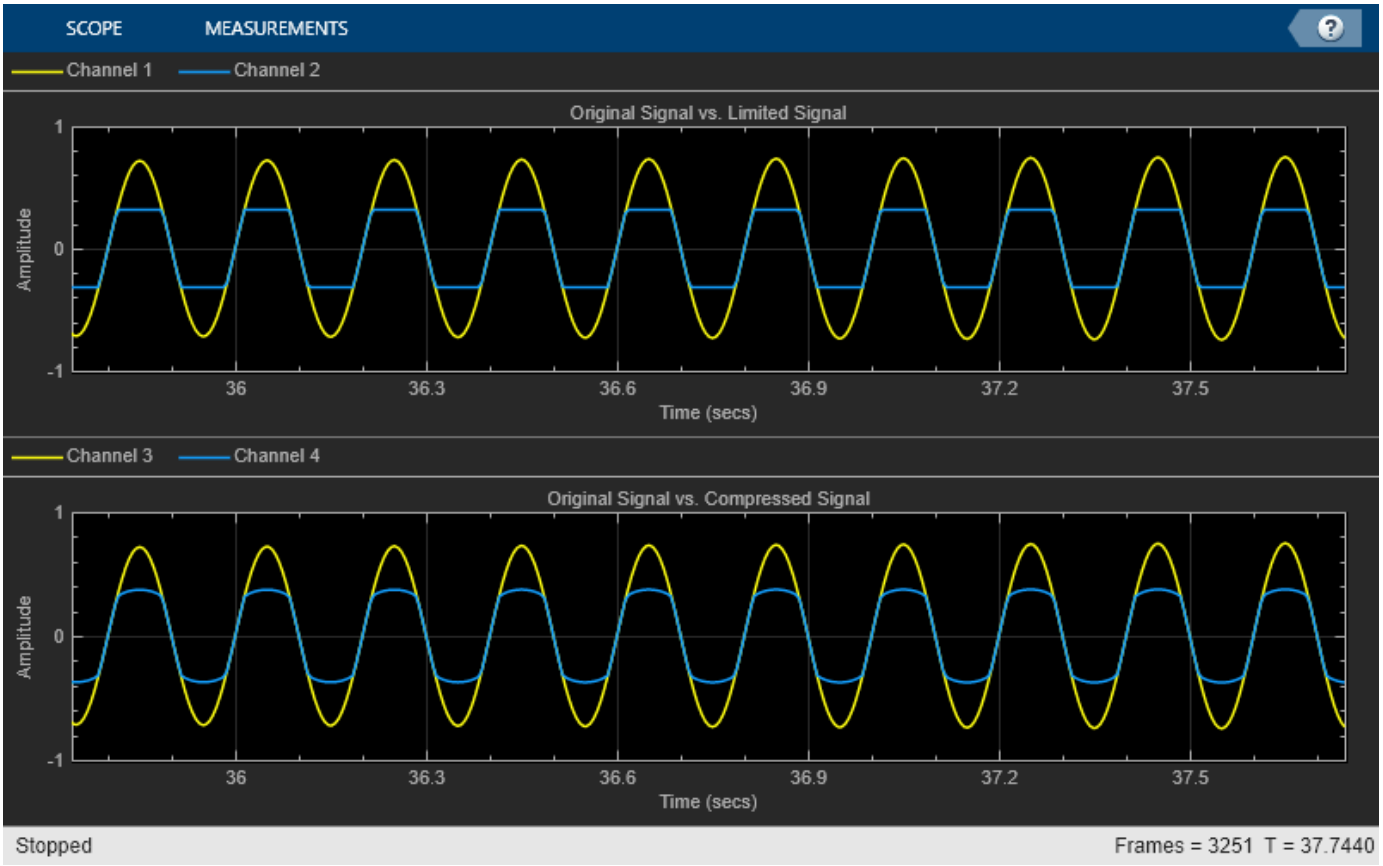
Create a time scope to visualize the generated sinusoid and the processed sinusoid.

```
scope = timescope( ...
    'SampleRate',osc.SampleRate, ...
    'TimeSpanSource','Property','TimeSpan',2, ...
```

```
    'BufferLength',osc.SampleRate*4, ...
    'TimeSpanOverrunAction','Scroll', ...
    'ShowGrid',true, ...
    'LayoutDimensions',[2 1], ...
    'NumInputPorts',2);
scope.ActiveDisplay = 1;
scope.Title = 'Original Signal vs. Limited Signal';
scope.YLimits = [-1,1];
scope.ActiveDisplay = 2;
scope.Title = 'Original Signal vs. Compressed Signal';
scope.YLimits = [-1,1];
```

In an audio stream loop, visualize the original sinusoid and the sinusoid processed by a limiter and a compressor. Increment the amplitude of the original sinusoid to illustrate the effect.

```
while osc.Amplitude < 0.75
    x = osc();

    xLimited    = dRL(x);
    xCompressed = dRC(x);

    scope([x xLimited],[x xCompressed]);

    osc.Amplitude = osc.Amplitude + 0.0002;
end
release(scope)
```



```
release(dRL)
release(dRC)
release(osc)
```

Compare Limiter and Compressor Applied to Audio Signal

Compare the effect of dynamic range limiters and compressors on a drum track. Create a `dsp.AudioFileReader` System object and a `audioDeviceWriter` System object to read audio from a file and write to your audio output device. To emphasize the effect of dynamic range control, set the operational threshold of the limiter and compressor to -20 dB.

```
dRL.Threshold = -20;
dRC.Threshold = -20;

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

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

```
numFrames = 300;

fprintf('Now playing original signal...\n')
```

```
Now playing original signal...
```

```
for i = 1:numFrames
    x = fileReader();
    deviceWriter(x);
end
reset(fileReader);

fprintf('Now playing limited signal...\n')
```

```
Now playing limited signal...
```

```
for i = 1:numFrames
    x = fileReader();
    xLimited = dRL(x);
    deviceWriter(xLimited);
end
reset(fileReader);

fprintf('Now playing compressed signal...\n')
```

```
Now playing compressed signal...
```

```
for i = 1:numFrames
    x = fileReader();
    xCompressed = dRC(x);
    deviceWriter(xCompressed);
end

release(fileReader)
release(deviceWriter)
release(dRC)
release(dRL)
```

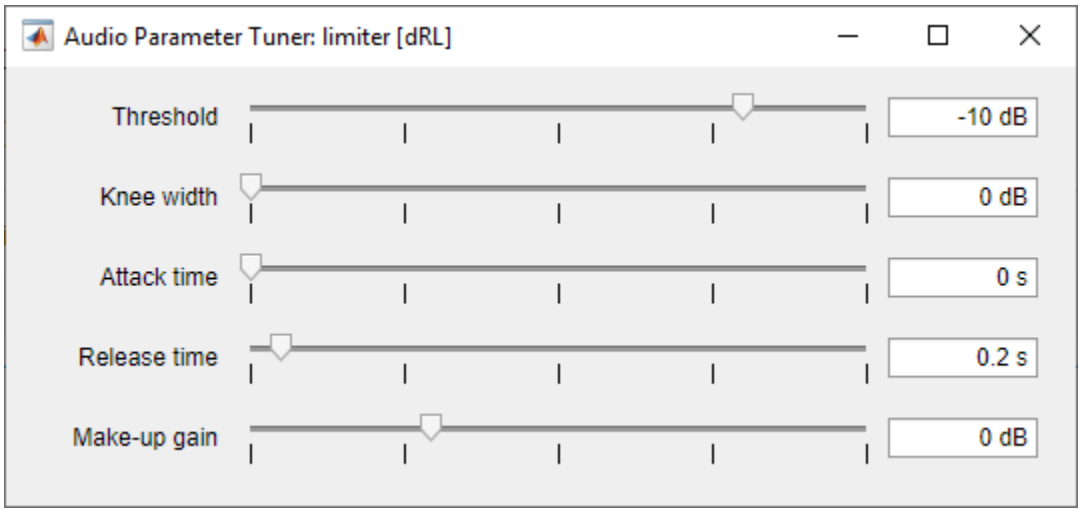
Tune Limiter Parameters

Create a `dsp.AudioFileReader` to read in audio frame-by-frame. Create a `audioDeviceWriter` to write audio to your sound card. Create a `limiter` to process the audio data.

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

dRL = limiter('SampleRate',fileReader.SampleRate);
```

Call `parameterTuner` to open a UI to tune parameters of the limiter while streaming.

```
parameterTuner(dRL)
```



In an audio stream loop:

1. Read in a frame of audio from the file.
2. Apply dynamic range limiting.
3. Write the frame of audio to your audio device for listening.

While streaming, tune parameters of the dynamic range limiter and listen to the effect.

```
while ~isDone(fileReader)
    audioIn = fileReader();
    audioOut = dRL(audioIn);
    deviceWriter(audioOut);
    drawnow limitrate % required to update parameter
end
```

As a best practice, release your objects once done.

```
release(deviceWriter)
release(fileReader)
release(dRL)
```

Sidechain Ducking with Limiter

Use the "EnableSidechain" on page 3-0 input of a `limiter` object to limit the amplitude level of a separate audio signal. The sidechain signal controls the dynamic range limiting of the input audio signal. When the sidechain signal exceeds the limiter "Threshold" on page 3-0 , the limiter activates and limits the amplitude of the input signal. When the sidechain signal level falls below the threshold, the audio input returns to its original amplitude. For a detailed comparison of compression and dynamic range limiting, see "Compare Dynamic Range Limiter and Compressor" on page 3-256.

Prepare Audio Files

In this section, you resample and zero-pad a speech file to use as input to the `EnableSidechain` property of your `limiter` object.

Read in an audio signal. Resample it to match the sample rate of the input audio signal (44.1 kHz).

```
targetFs = 44100;
[originalSpeech,originalFs] = audioread('Rainbow-16-8-mono-114secs.wav');
resampledSpeech = resample(originalSpeech,targetFs,originalFs);
```

Pad the beginning of the resampled signal with 10 seconds worth of zeros. This allows the input audio signal to be clearly heard before any limiting is applied.

```
resampledSpeech = [zeros(10*targetFs,1);resampledSpeech];
```

Normalize the amplitude to avoid potential clipping.

```
resampledSpeech = resampledSpeech ./ max(resampledSpeech);
```

Write the resampled, zero-padded, and normalized sidechain signal to a file.

```
audiowrite('resampledSpeech.wav',resampledSpeech,targetFs);
```

Construct Audio Objects

Construct a `dsp.AudioFileReader` object for the input and sidechain signals. Using the "ReadRange" property of the `AudioFileReader`, select the second verse of the input signal and the first 26.5 seconds of the sidechain signal for playback. To allow the script to run indefinitely, change the `playbackCount` variable from `1` to `Inf`.

```
inputAudio = 'SoftGuitar-44p1_mono-10mins.ogg';
sidechainAudio  = 'resampledSpeech.wav';
playbackCount = 1;
inputAudioAFR = dsp.AudioFileReader(inputAudio,'PlayCount',playbackCount,'ReadRange',...
    [115*targetFs round(145.4*targetFs)]);
sidechainAudioAFR = dsp.AudioFileReader(sidechainAudio,'PlayCount',playbackCount,...
    'ReadRange',[1 round(26.5*targetFs)]);
```

Construct a `limiter` object. Use a fast "AttackTime" on page 3-0 , and a moderately slow "ReleaseTime" on page 3-0 . These settings are ideal for voice-over work. The fast attack time ensures that the input audio is limited almost immediately after the sidechain signal surpasses the limiter threshold. The slow release time ensures the limiting on the input audio lasts through any potential short silent regions in the sidechain signal.

```
iAmYourLimiter = limiter('EnableSidechain',true,...
                'SampleRate',targetFs,...
                'Threshold',-48,...
```

```
                'AttackTime',0.01,...
                'ReleaseTime',1.75);
```

Construct an `audioDeviceWriter` object to play the sidechain and input signals.

```
afw = audioDeviceWriter;
```

Construct a `timescope` object to view the uncompressed input signal, the sidechain signal, as well as the compressed input signal.

```
scope = timescope('NumInputPorts',3,...
                        'SampleRate',targetFs,...
                        'TimeSpanSource','property',...
                        'TimeSpan',5,...
                         'TimeDisplayOffset',0,...
                        'LayoutDimensions',[3 1],...
                        'BufferLength',targetFs*15,...
                        'TimeSpanOverrunAction','Scroll',...
                        'YLimits',[-1 1],...
                        'ShowGrid',true,...
                        'Title','Original Input Audio - Guitar');
scope.ActiveDisplay = 2;
scope.YLimits = [-1 1];
scope.Title = 'Sidechain Audio - Speech';
scope.ShowGrid = true;
scope.ActiveDisplay = 3;
scope.YLimits = [-1 1];
scope.ShowGrid = true;
scope.Title = 'Dynamic Range Limited Input Audio - Guitar';
```

Create Audio Streaming Loop

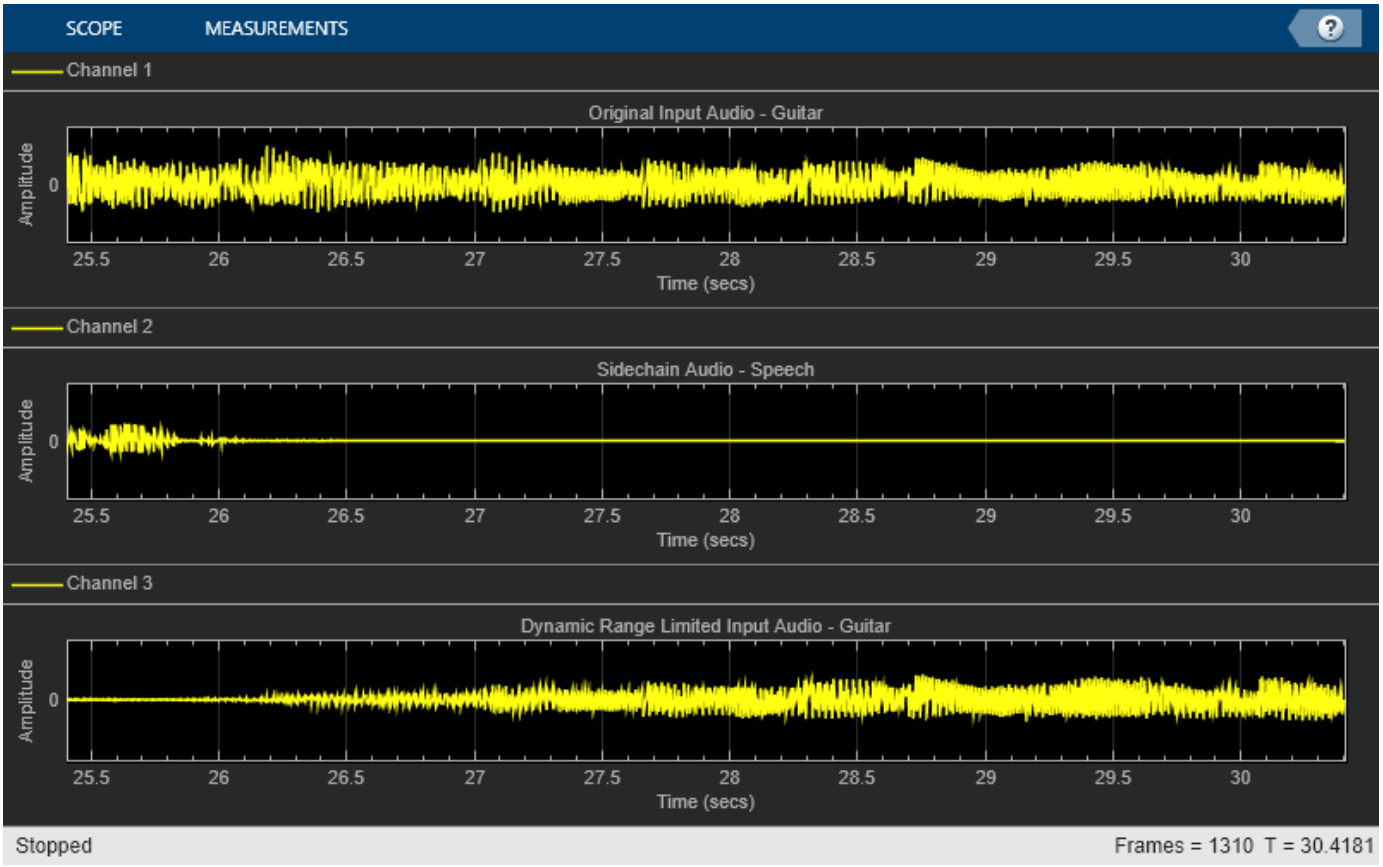
Read in a frame of audio from your input and sidechain signals. Process your input and sidechain signals with your `limiter` object. Playback your processed audio signals and display the audio data using a `timescope` object.

The top panel of your `timescope` displays the unprocessed input audio signal and the middle panel displays the sidechain audio signal. The bottom panel displays the limited input audio signal. Notice the amplitudes of the signals in the top and bottom panels are identical until the sidechain signal begins. Once the sidechain signal activates, the amplitude in the bottom panel decreases. Once the sidechain signal ends, the amplitude of the bottom panel returns to its original level.

```
while ~isDone(inputAudioAFR)
   inputAudioFrame = inputAudioAFR();
   sideChainAudioFrame = sidechainAudioAFR();
   limiterOutput  = iAmYourLimiter(inputAudioFrame,sideChainAudioFrame);
   afw(sideChainAudioFrame+limiterOutput);
   scope(inputAudioFrame,sideChainAudioFrame,limiterOutput);
end
```

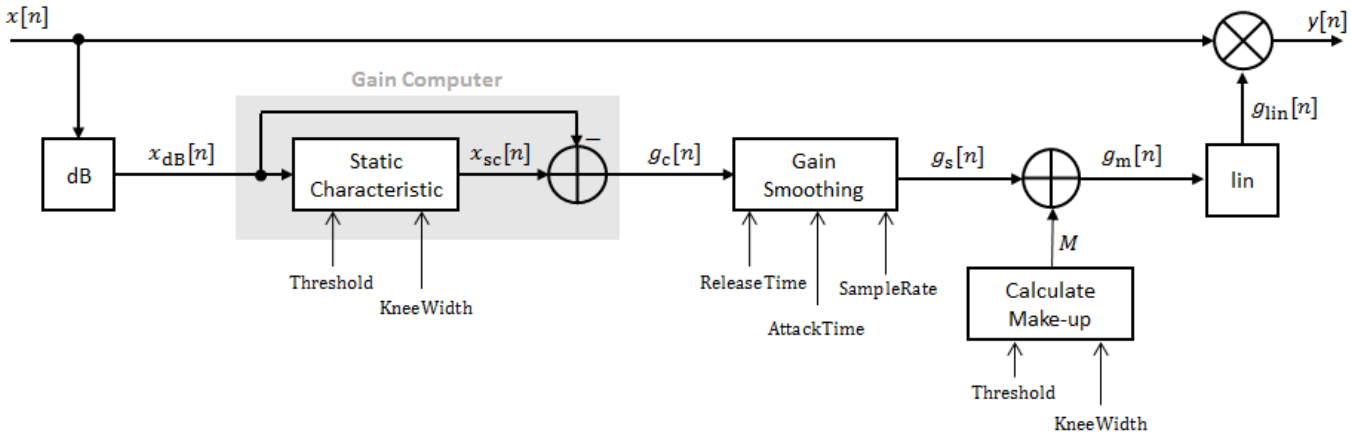
Release your objects.

```
release(inputAudioAFR)
release(sidechainAudioAFR)
release(iAmYourLimiter)
release(afw)
release(scope)
```



Algorithms

The `limiter` System object processes a signal frame by frame and element by element.



Convert Input Signal to dB

The N-point signal, $x[n]$, is converted to decibels:

$$x_{dB}[n] = 20 \times \log_{10}|x[n]|$$

Gain Computer

$x_{\mathrm{dB}}[n]$ passes through the gain computer. The gain computer uses the static characteristic properties of the dynamic range limiter to brick-wall gain that is above the threshold.

If you specified a soft knee, the gain computer has the following static characteristic:

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

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:

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

The computed gain, $g_{\mathrm{c}}[n]$, is calculated as

$$g_{\mathrm{c}}[n] = x_{\mathrm{sc}}[n] - x_{\mathrm{dB}}[n]\,.$$

Gain Smoothing

$g_{\mathrm{c}}[n]$ is smoothed using specified attack and release time:

$$g_{\mathrm{s}}[n] = \begin{cases} \alpha_{\mathrm{A}} g_{\mathrm{s}}[n-1] + (1-\alpha_{\mathrm{A}}) g_{\mathrm{c}}[n], & g_{\mathrm{c}}[n] \le g_{\mathrm{s}}[n-1] \\ \alpha_{\mathrm{R}} g_{\mathrm{s}}[n-1] + (1-\alpha_{\mathrm{R}}) g_{\mathrm{c}}[n], & g_{\mathrm{c}}[n] > g_{\mathrm{s}}[n-1] \end{cases}$$

The attack time coefficient, α_{A} , is calculated as

$$\alpha_{\mathrm{A}} = \exp\left(\frac{-\log(9)}{Fs \times T_{\mathrm{A}}}\right).$$

The release time coefficient, α_{R} , is calculated as

$$\alpha_{\mathrm{R}} = \exp\left(\frac{-\log(9)}{Fs \times T_{\mathrm{R}}}\right).$$

T_{A} is the attack time period, specified by the `AttackTime` property. T_{R} is the release time period, specified by the `ReleaseTime` property. *Fs* is the input sampling rate, specified by the `SampleRate` property.

Calculate and Apply Make-up Gain

If `MakeUpGainMode` is set to the default `'Auto'`, the make-up gain is calculated as the negative of the computed gain for a 0 dB input:

$$M = -x_{\mathrm{sc}}\big|_{x_{\mathrm{dB}} = 0}$$

Given a steady-state input of 0 dB, this configuration achieves a steady-state output of 0 dB. The make-up gain is determined by the `Threshold` and `KneeWidth` properties. It does not depend on the input signal.

The make-up gain, M, is added to the smoothed gain, $g_s[n]$:

$$g_m[n] = g_s[n] + M$$

Calculate and Apply Linear Gain

The calculated gain in dB, $g_m[n]$, is translated to a linear domain:

$$g_{lin}[n] = 10^{\left(\frac{g_m[n]}{20}\right)}.$$

The output of the dynamic range limiter is given as

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

Version History

Introduced in R2016a

References

[1] Giannoulis, Dimitrios, Michael Massberg, and Joshua D. Reiss. "Digital Dynamic Range Compressor Design –– A Tutorial and Analysis." *Journal of Audio Engineering Society*. Vol. 60, Issue 6, 2012, pp. 399–408.

Extended Capabilities

C/C++ Code Generation

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

Usage notes and limitations:

“System Objects in MATLAB Code Generation” (MATLAB Coder)

See Also

Limiter | `noiseGate` | `compressor` | `expander`

Topics

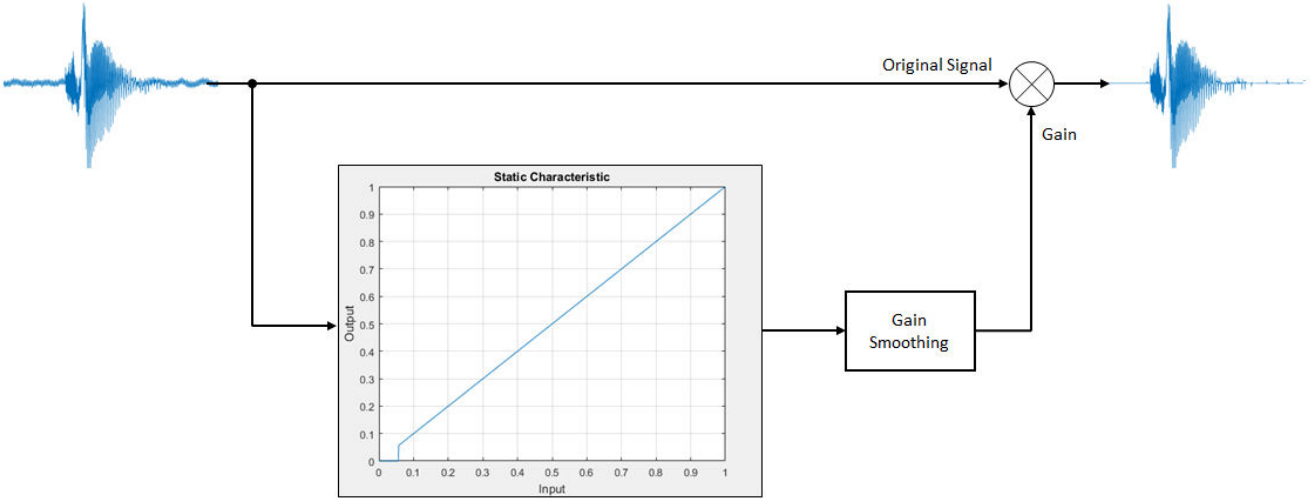
“Dynamic Range Control”

noiseGate

Dynamic range gate

Description

The `noiseGate` System object performs dynamic range gating independently across each input channel. Dynamic range gating suppresses signals below a given threshold. It uses specified attack, release, and hold times to achieve a smooth applied gain curve. Properties of the `noiseGate` System object specify the type of dynamic range gating.



To perform dynamic range gating:

1. Create the `noiseGate` object and set its properties.
2. Call the object with arguments, as if it were a function.

To learn more about how System objects work, see What Are System Objects?

Creation

Syntax

```
dRG = noiseGate
dRG = noiseGate(thresholdValue)
dRG = noiseGate( ___ ,Name,Value)
```

Description

`dRG = noiseGate` creates a System object, `dRG`, that performs dynamic range gating independently across each input channel.

`dRG = noiseGate(thresholdValue)` sets the Threshold property to `thresholdValue`.

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

Example: `dRG = noiseGate('AttackTime',0.01,'SampleRate',16000)` creates a System object, `dRG`, with a 10 ms attack time and a 16 kHz sample rate.

Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see System Design in MATLAB Using System Objects.

`Threshold` — Operation threshold (dB)

`–10` (default) | real scalar

Operation threshold in dB, specified as a real scalar.

Operation threshold is the level below which gain is applied to the input signal.

Tunable: Yes

Data Types: `single` | `double`

`AttackTime` — Attack time (s)

`0.05` (default) | real scalar

Attack time in seconds, specified as a real scalar greater than or equal to 0.

Attack time is the time it takes the applied gain to rise from 10% to 90% of its final value when the input goes below the threshold.

Tunable: Yes

Data Types: `single` | `double`

`ReleaseTime` — Release time (s)

`0.02` (default) | real scalar

Release time in seconds, specified as a real scalar greater than or equal to 0.

Release time is the time it takes the applied gain to drop from 90% to 10% of its final value when the input goes above the threshold.

Tunable: Yes

Data Types: `single` | `double`

`HoldTime` — Hold time (s)

`0.05` (default) | real finite scalar

Hold time in seconds, specified as a real scalar greater than or equal to 0.

Hold time is the period for which the (negative) gain is held before starting to decrease towards its steady state value when the input level drops below the threshold.

Tunable: Yes

Data Types: `single` | `double`

`SampleRate` — Input sample rate (Hz)

`44100` (default) | positive scalar

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

Tunable: Yes

Data Types: `single` | `double`

`EnableSidechain` — Enable sidechain input

`false` (default) | `true`

Enable sidechain input, specified as `true` or `false`. This property determines the number of available inputs on the `noiseGate` object.

- `false` –– Sidechain input is disabled and the `noiseGate` object accepts one input: the `audioIn` data to be gated.
- `true` –– Sidechain input is enabled and the `noiseGate` object accepts two inputs: the `audioIn` data to be gated and the sidechain input used to compute the `gain` applied by `noiseGate`.

The sidechain datatype and (frame) length must be the same as `audioIn`.

The number of channels of the sidechain must be equal to the number of channels of `audioIn` or be equal to one. When the number of sidechain channels is one, the `gain` computed based on this channel is applied to all channels of `audioIn`. When the number of sidechain channels is equal to the number of channels in `audioIn`, the `gain` computed for each sidechain channel is applied to the corresponding channel of `audioIn`.

Tunable: No

Usage

Syntax

```
audioOut = dRG(audioIn)
[audioOut,gain] = dRG(audioIn)
```

Description

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

`[audioOut,gain] = dRG(audioIn)` also returns the applied gain, in dB, at each input sample.

Input Arguments

`audioIn` — Audio input to noise gate
matrix

Audio input to the noise gate, specified as a matrix. The columns of the matrix are treated as independent audio channels.

Data Types: `single` | `double`

Output Arguments

`audioOut` — Audio output from noise gate
matrix

Audio output from the noise gate, returned as a matrix the same size as `audioIn`.

Data Types: `single` | `double`

`gain` — Gain applied by noise gate (dB)
matrix

Gain applied by noise gate, returned as a matrix the same size as `audioIn`.

Data Types: `single` | `double`

Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

Specific to noiseGate

visualize	Visualize static characteristic of dynamic range controller
createAudioPluginClass	Create audio plugin class that implements functionality of System object
parameterTuner	Tune object parameters while streaming

MIDI

configureMIDI	Configure MIDI connections between audio object and MIDI controller
disconnectMIDI	Disconnect MIDI controls from audio object
getMIDIConnections	Get MIDI connections of audio object

Common to All System Objects

clone	Create duplicate System object
isLocked	Determine if System object is in use
release	Release resources and allow changes to System object property values and input characteristics
reset	Reset internal states of System object
step	Run System object algorithm

The `createAudioPluginClass` and `configureMIDI` functions map tunable properties of the `noiseGate` System object to user-facing parameters:

Property	Range	Mapping	Unit
`Threshold`	[–140, 0]	linear	dB
`AttackTime`	[0, 4]	linear	seconds
`ReleaseTime`	[0, 4]	linear	seconds
`HoldTime`	[0, 4]	linear	seconds

Examples

Gate Audio Signal

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

Set up the `dsp.AudioFileReader` and `audioDeviceWriter` System objects™.

```
frameLength = 1024;
fileReader = dsp.AudioFileReader( ...
    'Filename','Counting-16-44p1-mono-15secs.wav', ...
    'SamplesPerFrame',frameLength);
deviceWriter = audioDeviceWriter( ...
    'SampleRate',fileReader.SampleRate);
```

Corrupt the audio signal with Gaussian noise. Play the audio.

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

release(fileReader)
```

Set up a dynamic range gate with a threshold of -25 dB, an attack time of 0.01 seconds, a release time of 0.02 seconds, and a hold time of 0 seconds. Use the sample rate of your audio file reader.

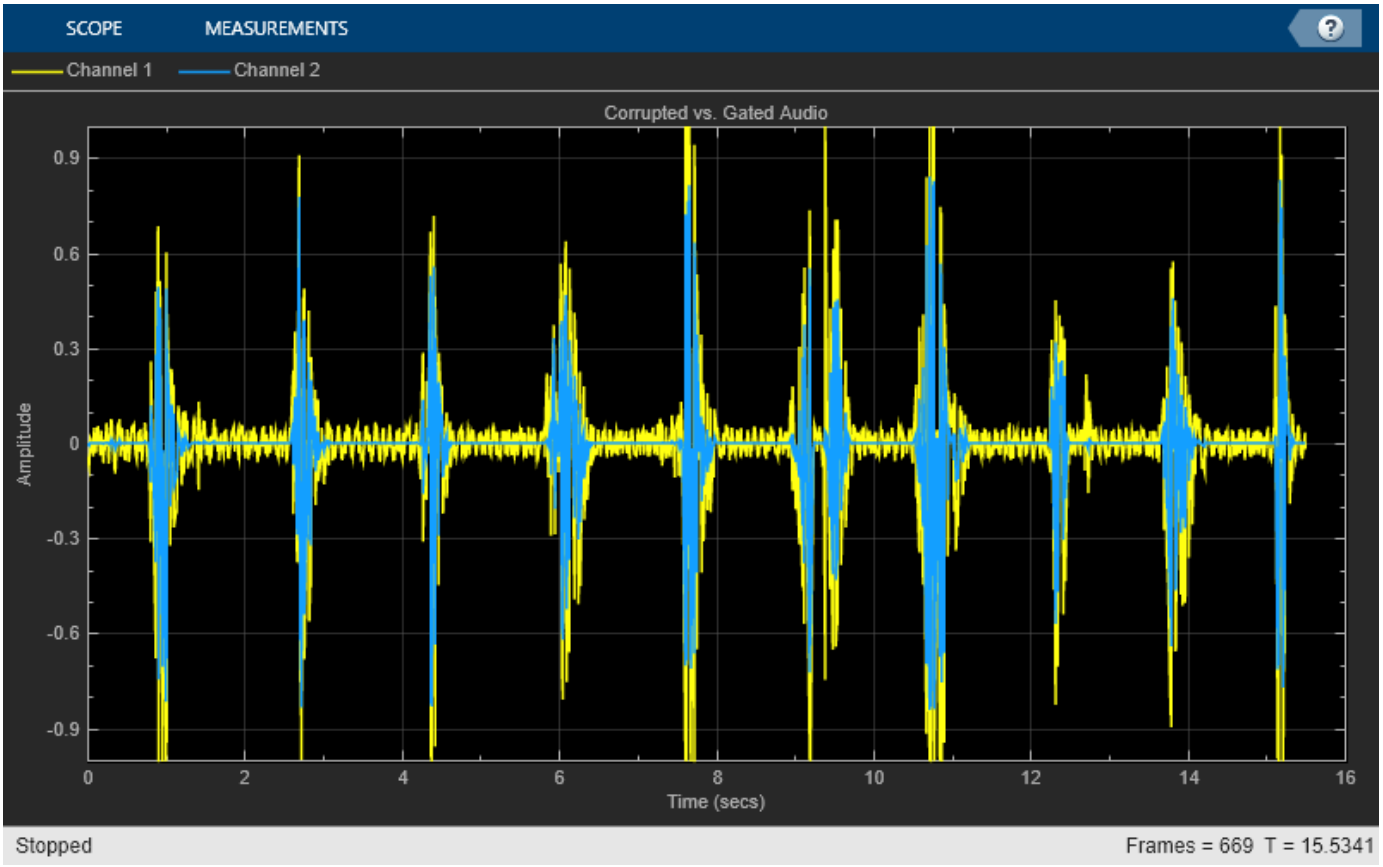
```
gate = noiseGate(-25, ...
    'AttackTime',0.01, ...
    'ReleaseTime',0.02, ...
    'HoldTime',0, ...
    'SampleRate',fileReader.SampleRate);
```

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

```
scope = timescope( ...
    'SampleRate',fileReader.SampleRate, ...
    'TimeSpanOverrunAction','Scroll', ...
    'TimeSpanSource','property',...
    '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)
```



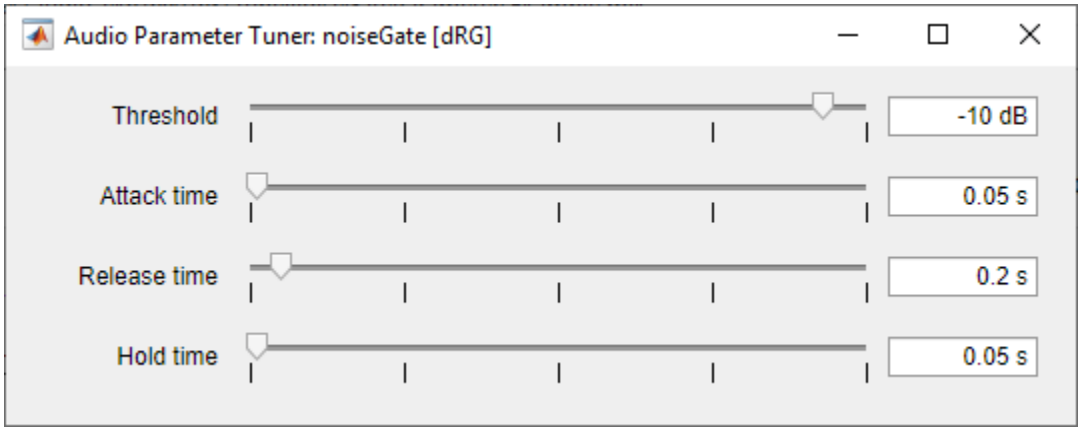
Tune Noise Gate Parameters

Create a `dsp.AudioFileReader` to read in audio frame-by-frame. Create an `audioDeviceWriter` to write audio to your sound card. Create a `noiseGate` to process the audio data.

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

dRG = noiseGate('SampleRate',fileReader.SampleRate);
```

Call `parameterTuner` to open a UI to tune parameters of the `noiseGate` while streaming.

```
parameterTuner(dRG)
```



In an audio stream loop:

1. Read in a frame of audio from the file.
2. Apply dynamic range gating.
3. Write the frame of audio to your audio device for listening.

While streaming, tune parameters of the dynamic range gate and listen to the effect.

```
while ~isDone(fileReader)
    audioIn = fileReader();
    audioOut = dRG(audioIn);
    deviceWriter(audioOut);
    drawnow limitrate % required to update parameter
end
```

As a best practice, release your objects once done.

```
release(deviceWriter)
release(fileReader)
release(dRG)
```

Sidechain Dynamic Range Gating

Use the “EnableSidechain” on page 3-0 input of a `noiseGate` object to emulate an electronic drum controller, also known as a *multipad*. This technique is common in recording studio production and creates interesting changes to the timbre of an instrument. The sidechain signal controls the gating on the input signal. Sidechain gating decreases the amplitude of the input signal when the sidechain signal falls below the “Threshold” on page 3-0 of the `noiseGate`. A noise gate is essentially an `expander` with an infinite “Ratio” on page 3-0 .

Prepare Audio Files

Convert the sidechain signal from stereo to mono.

```
[expanderSideChainStereo,Fs] = audioread('FunkyDrums-44p1-stereo-25secs.mp3');
expanderSideChainMono = (expanderSideChainStereo(:,1) + expanderSideChainStereo(:,2)) / 2;
```

Write the converted sidechain signal to a file.

```
audiowrite('convertedSidechainSig.wav',expanderSideChainMono,Fs);
```

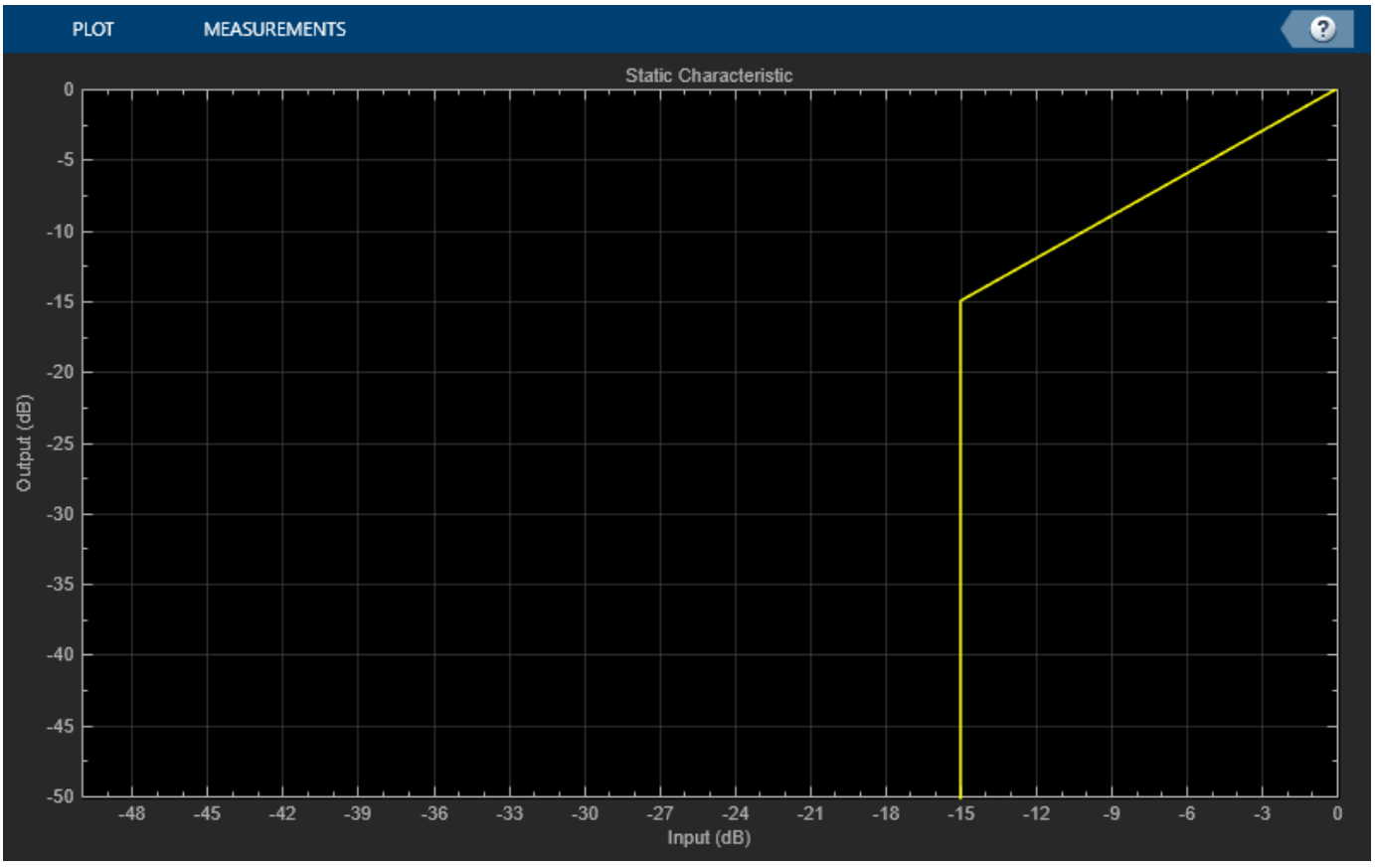
Construct Audio Objects

Construct a `dsp.AudioFileReader` object for the input and sidechain signals. To allow the script to run indefinitely, change the `playbackCount` variable from `1` to `Inf`.

```
inputAudio = 'SoftGuitar-44p1_mono-10mins.ogg';
sidechainAudio  = 'convertedSidechainSig.wav';
playbackCount = 1;
inputAudioAFR = dsp.AudioFileReader(inputAudio,'PlayCount',playbackCount);
sidechainAudioAFR = dsp.AudioFileReader(sidechainAudio,'PlayCount',playbackCount);
```

Construct and visualize a `noiseGate` object. Use fast “AttackTime” on page 3-0 and “ReleaseTime” on page 3-0 , and a short “HoldTime” on page 3-0 .

```
dRG = noiseGate('EnableSidechain',true,'Threshold',-15,'AttackTime',...
    0.08,'ReleaseTime',0.0001,'HoldTime',0.00001);
visualize(dRG)
```



Construct an `audioDeviceWriter` object to play the sidechain and input signals.

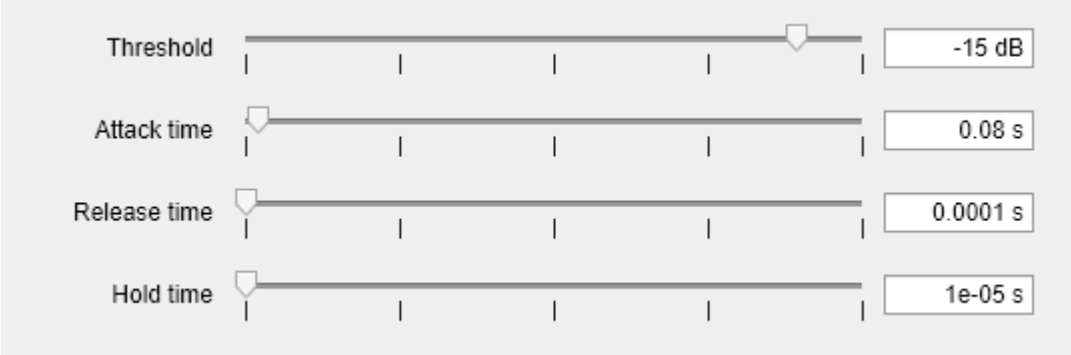
```
afw = audioDeviceWriter;
```

Construct a `timescope` object to view the input signal, the sidechain signal, as well as the gated input signal.

```
scope = timescope('NumInputPorts',3,...
                      'SampleRate',Fs,...
                      'TimeSpanSource','property',...
                      'TimeSpan',5,...
                       'TimeDisplayOffset',0,...
                      'LayoutDimensions',[3 1],...
                      'BufferLength',Fs*15,...
                      'TimeSpanOverrunAction','Scroll',...
                      'YLimits',[-1 1],...
                      'ShowGrid',true,...
                      'Title','Input Audio - Classical Guitar');
scope.ActiveDisplay = 2;
scope.YLimits = [-1 1];
scope.Title = 'Sidechain Audio - Drums';
scope.ShowGrid = true;
scope.ActiveDisplay = 3;
scope.YLimits = [-1 1];
scope.ShowGrid = true;
scope.Title = 'Gated Input Audio - Classical Guitar';
```

Call `parameterTuner` to open a UI to tune parameters of the gate while streaming. Adjust the property values and listen to the effect in real time.

```
parameterTuner(dRG)
```



Create Audio Streaming Loop

Read in a frame of audio from your input and sidechain signals. Process your input and sidechain signals with your `noiseGate` object. Playback your processed audio signals and display the audio data using a `timescope` object.

The top panel of your `timescope` displays the input audio signal and the middle panel displays the sidechain audio signal. The bottom panel displays the gated input audio signal.

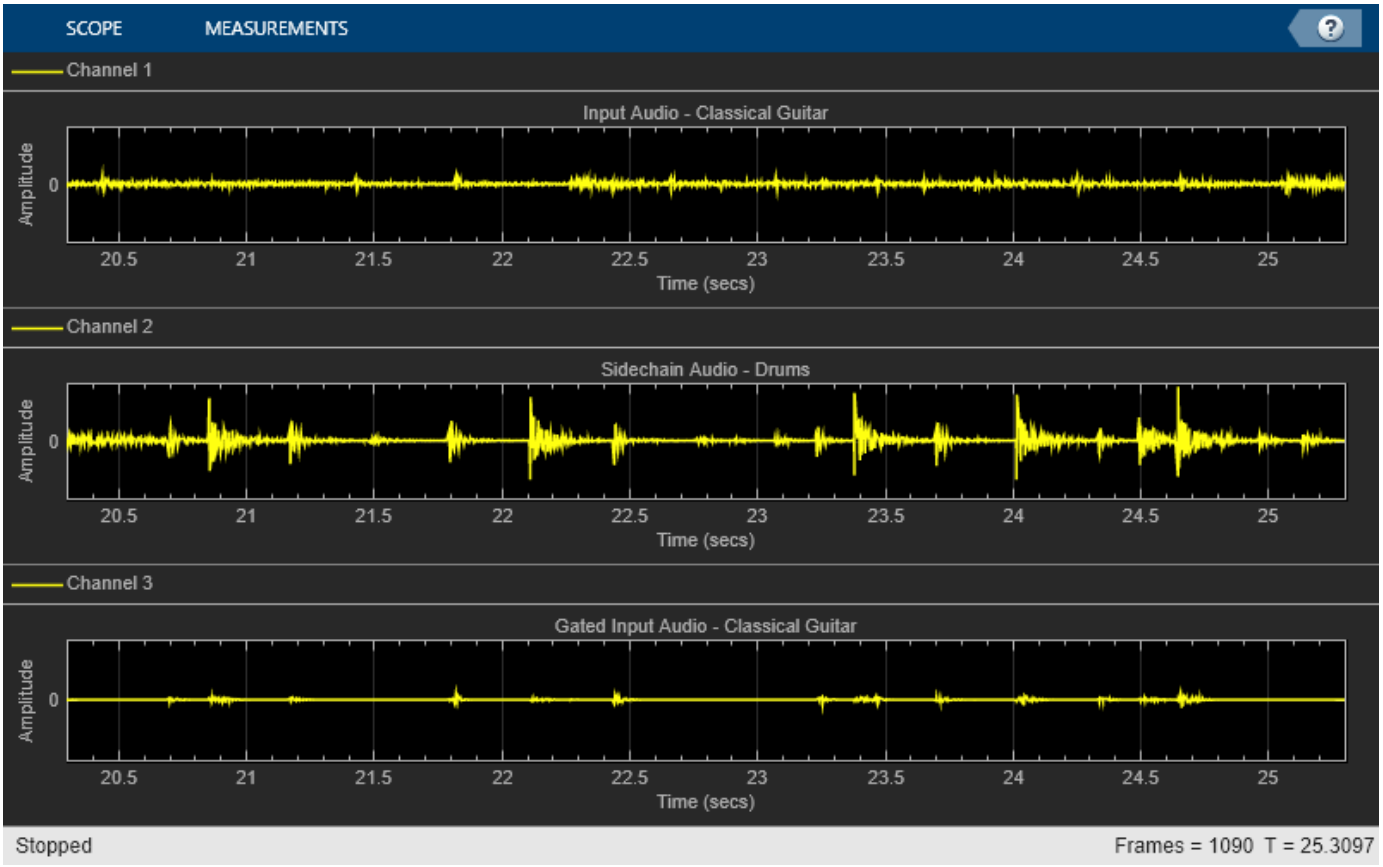
Substitute different audio files for your `inputAudio` variable to create different textures and timbres in your drum mix.

```
while ~isDone(sidechainAudioAFR)
   inputAudioFrame = inputAudioAFR();
   sideChainAudioFrame = sidechainAudioAFR();
   noiseGateOutput  = dRG(inputAudioFrame,sideChainAudioFrame);
```

```
    afw(sideChainAudioFrame+noiseGateOutput);
    scope(inputAudioFrame,sideChainAudioFrame,noiseGateOutput);
    drawnow limitrate;   % required to update parameter settings from UI
end
```

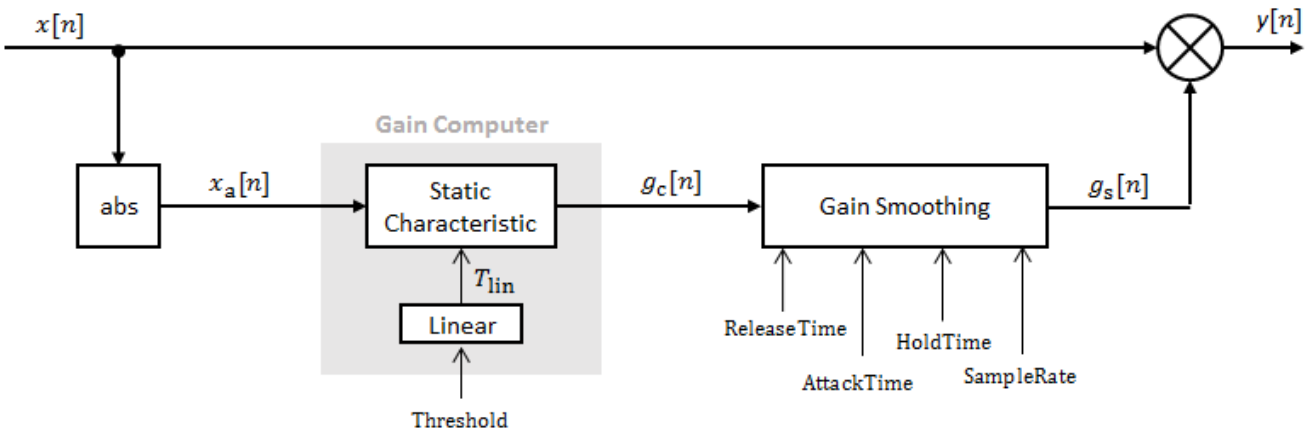
Release your objects.

```
release(inputAudioAFR)
release(sidechainAudioAFR)
release(dRG)
release(afw)
release(scope)
```



Algorithms

The `noiseGate` System object processes a signal frame by frame and element by element.



Convert Input Signal to Magnitude

The N-point signal, $x[n]$, is converted to magnitude:

$$x_a[n] = |x[n]|.$$

Gain Computer

$x_a[n]$ passes through the gain computer. The gain computer uses the static characteristic properties of the dynamic range gate to determine a brick-wall gain for signal below the threshold:

$$g_c(x_a) = \begin{cases} 0 & x_a < T_{lin} \\ 1 & x_a \geq T_{lin} \end{cases}.$$

T_{lin} is the threshold property converted to a linear domain:

$$T_{lin} = 10^{\left(T_{dB}/20\right)}.$$

Gain Smoothing

The computed gain, $g_c[n]$, is smoothed using specified attack, release, and hold time properties:

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

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.

C_A is the hold counter for attack. The limit, T_H , is determined by the `HoldTime` property.

Apply Gain

The output of the dynamic range gate is given as

$$y[n] = x[n] \times g_s[n] .$$

Version History

Introduced in R2016a

References

[1] Giannoulis, Dimitrios, Michael Massberg, and Joshua D. Reiss. "Digital Dynamic Range Compressor Design –– A Tutorial and Analysis." *Journal of Audio Engineering Society*. Vol. 60, Issue 6, 2012, pp. 399–408.

Extended Capabilities

C/C++ Code Generation

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

Usage notes and limitations:

“System Objects in MATLAB Code Generation” (MATLAB Coder)

See Also

Noise Gate | `expander` | `compressor` | `limiter`

Topics

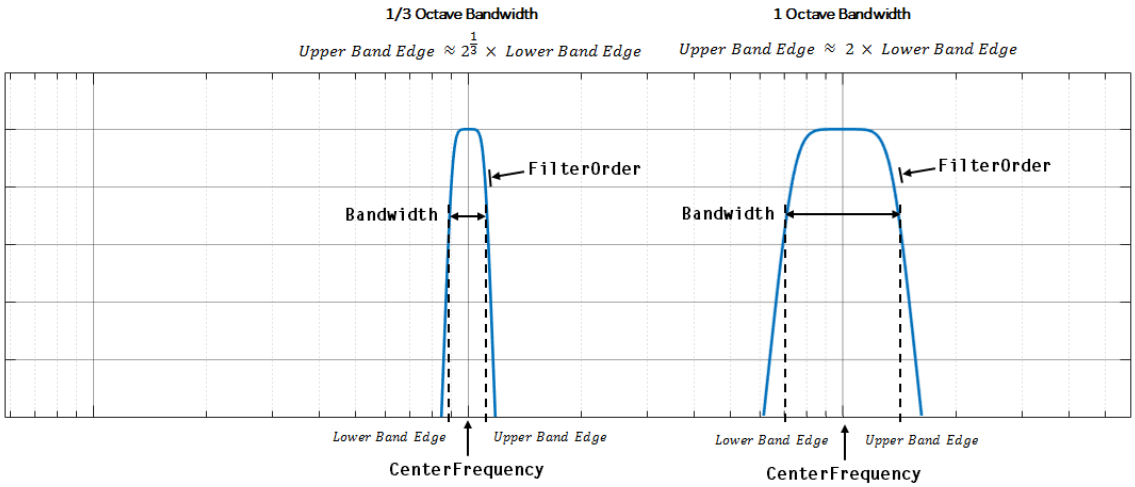
“Dynamic Range Control”

octaveFilter

Octave-band and fractional octave-band filter

Description

The `octaveFilter` System object performs octave-band or fractional octave-band filtering independently across each input channel. An octave-band is a frequency band where the highest frequency is twice the lowest frequency. Octave-band and fractional octave-band filters are commonly used to mimic how humans perceive loudness. Octave filters are best understood when viewed on a logarithmic scale, which models how the human ear weights the spectrum.



To perform octave-band or fractional octave-band filtering on your input:

1. Create the `octaveFilter` object and set its properties.
2. Call the object with arguments, as if it were a function.

To learn more about how System objects work, see What Are System Objects?

Creation

Syntax

```
octFilt = octaveFilter
octFilt = octaveFilter(centerFreq)
octFilt = octaveFilter(centerFreq,bw)
octFilt = octaveFilter( ___ ,Name,Value)
```

Description

`octFilt = octaveFilter` creates a System object, `octFilt`, that performs octave-band filtering independently across each input channel.

`octFilt = octaveFilter(centerFreq)` sets the CenterFrequency property to `centerFreq`.

`octFilt = octaveFilter(centerFreq,bw)` sets the Bandwidth property to `bw`.

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

Example: `octFilt = octaveFilter(1000,'1/3 octave','SampleRate',96000)` creates a System object, `octFilt`, with a center frequency of 1000 Hz, a 1/3 octave filter bandwidth, and a sample rate of 96,000 Hz.

Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see System Design in MATLAB Using System Objects.

`FilterOrder` — Order of octave filter

`6` (default) | even integer

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

Tunable: No

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

`CenterFrequency` — Center frequency of octave filter (Hz)

`1000` (default) | positive scalar

Center frequency of the octave filter in Hz, specified as a positive scalar.

When using the `parameterTuner`, the center frequency must be in the range [0.2, SampleRate/2] Hz.

Tunable: Yes

Data Types: `single` | `double`

`Bandwidth` — Filter bandwidth (octaves)

`'1 octave'` (default) | `'2/3 octave'` | `'1/2 octave'` | `'1/3 octave'` | `'1/6 octave'` | `'1/12 octave'` | `'1/24 octave'` | `'1/48 octave'`

Filter bandwidth in octaves, specified as `'1 octave'`, `'2/3 octave'`, `'1/2 octave'`, `'1/3 octave'`, `'1/6 octave'`, `'1/12 octave'`, `'1/24 octave'`, or `'1/48 octave'`.

Tunable: Yes

Data Types: `char` | `string`

`Oversample` — Oversample toggle

`false` (default) | `true`

Oversample toggle, specified as `false` or `true`.

- `false` –– The octave filter runs at the input sample rate.
- `true` –– The octave filter runs at two times the input sample rate. Oversampling minimizes the frequency warping effects introduced by the bilinear transformation. An FIR halfband interpolator implements oversampling before octave filtering. A halfband decimator reduces the sample rate back to the input sampling rate after octave filtering.

Tunable: No

Data Types: `logical`

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

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

Tunable: Yes

Data Types: `single` | `double`

Usage

Syntax

```
audioOut = octFilt(audioIn)
```

Description

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

Input Arguments

`audioIn` — Audio input to octave filter
matrix

Audio input to the octave filter, specified as a matrix. The columns of the matrix are treated as independent audio channels.

Data Types: `single` | `double`

Output Arguments

`audioOut` — Audio output from octave filter
matrix

Audio output from the octave filter, returned as a matrix the same size as `audioIn`.

Data Types: `single` | `double`

Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

Specific to octaveFilter

createAudioPluginClass	Create audio plugin class that implements functionality of System object
visualize	Visualize and validate filter response
isStandardCompliant	Verify octave filter design is ANSI S1.11-2004 compliant
getFilter	Return biquad filter object with design parameters set
getANSICenterFrequencies	Get the list of valid ANSI S1.11-2004 center frequencies
parameterTuner	Tune object parameters while streaming

MIDI

configureMIDI	Configure MIDI connections between audio object and MIDI controller
disconnectMIDI	Disconnect MIDI controls from audio object
getMIDIConnections	Get MIDI connections of audio object

Common to All System Objects

clone	Create duplicate System object
isLocked	Determine if System object is in use
release	Release resources and allow changes to System object property values and input characteristics
reset	Reset internal states of System object
step	Run System object algorithm

Note `octaveFilter` supports additional filter analysis functions. See Analyze Octave Filter Design on page 3-320 for details.

Examples

Perform Fractional Octave-Band Filtering

Use `octaveFilter` to design a 1/3 octave-band filter centered at 1000 Hz. Process an audio signal using your octave filter design.

Create a `dsp.AudioFileReader` object.

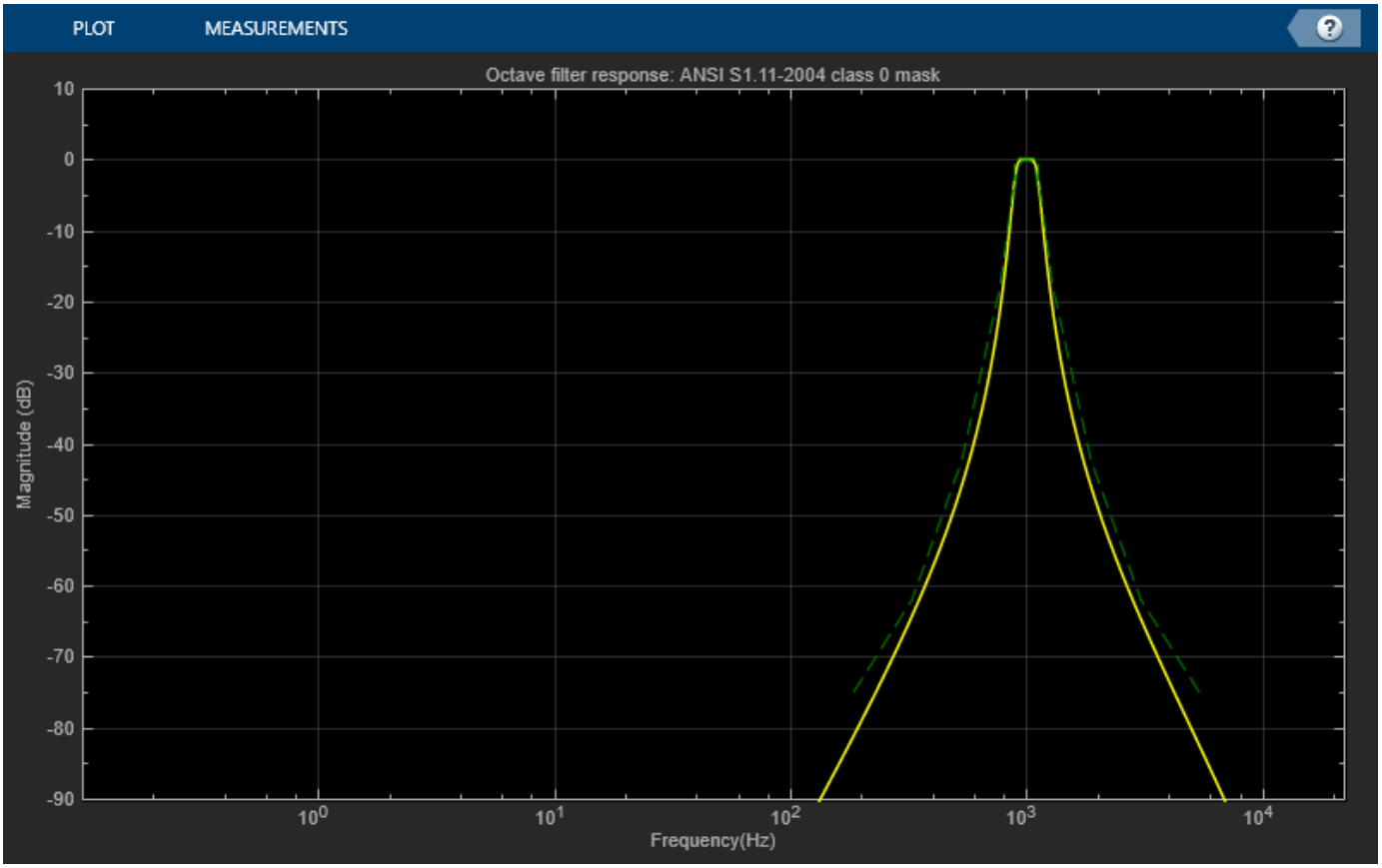
```
samplesPerFrame = 1024;
reader = dsp.AudioFileReader("RockGuitar-16-44p1-stereo-72secs.wav",SamplesPerFrame=samplesPerFra
```

Create an `octaveFilter` 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")
```

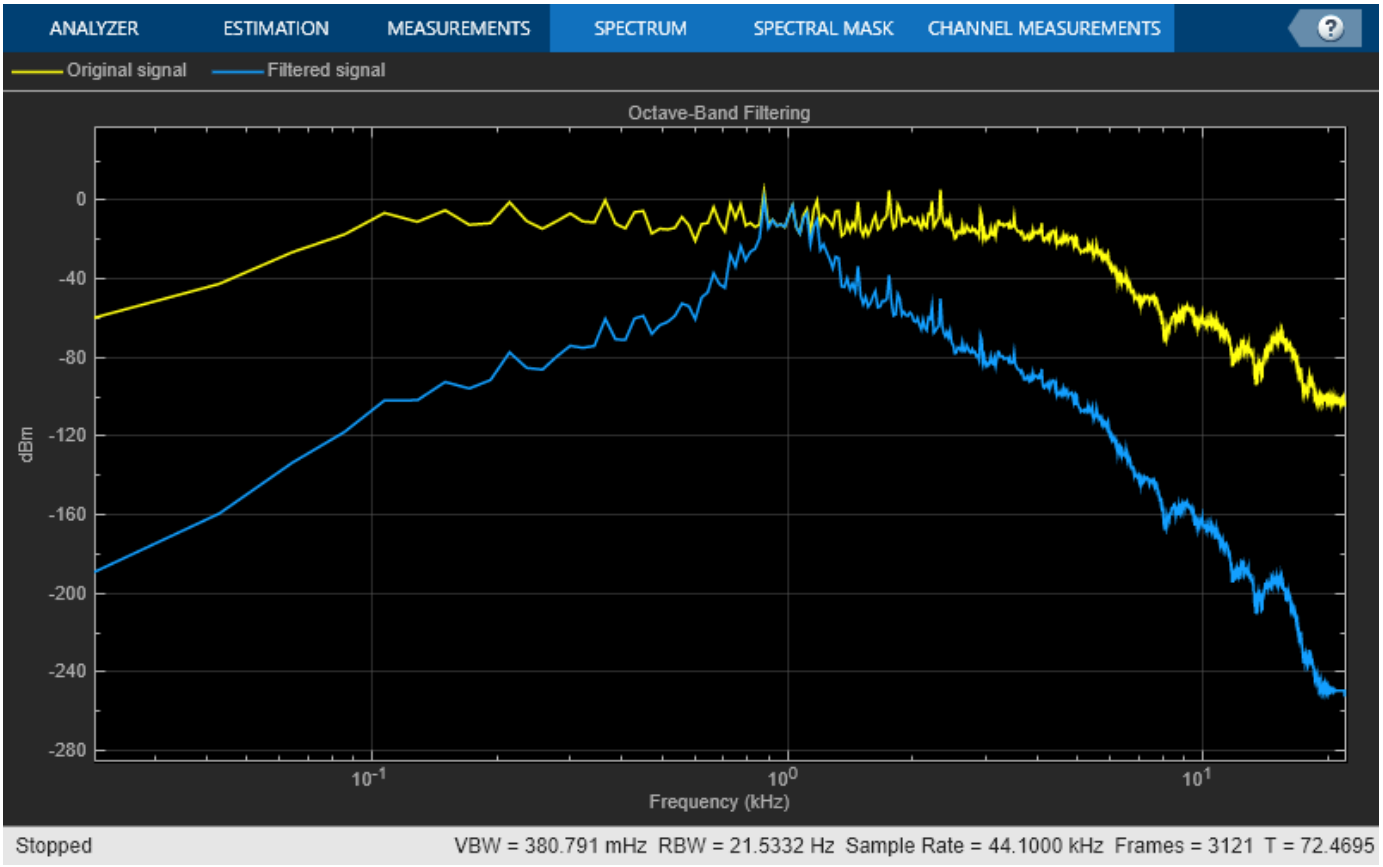


Create a spectrum analyzer to visualize the original audio signal and the audio signal after octave-band filtering.

```
scope = spectrumAnalyzer( ...
    SampleRate=Fs, ...
    PlotAsTwoSidedSpectrum=false, ...
    FrequencyScale="log", ...
    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.

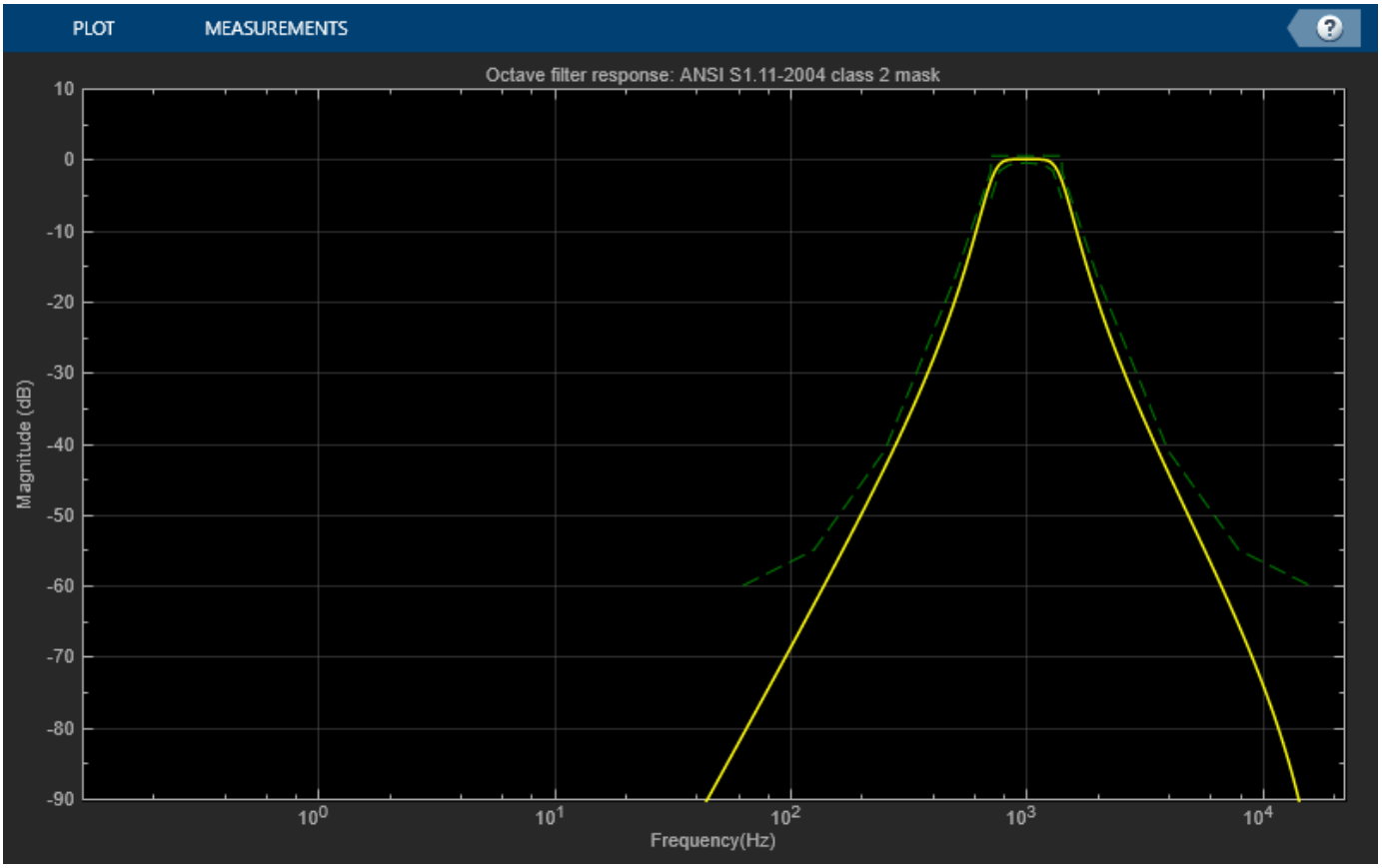
```
while ~isDone(reader)
    x = reader();
    y = octFilt(x);
    scope([x(:,1),y(:,1)])
end

release(octFilt)
release(reader)
release(scope)
```

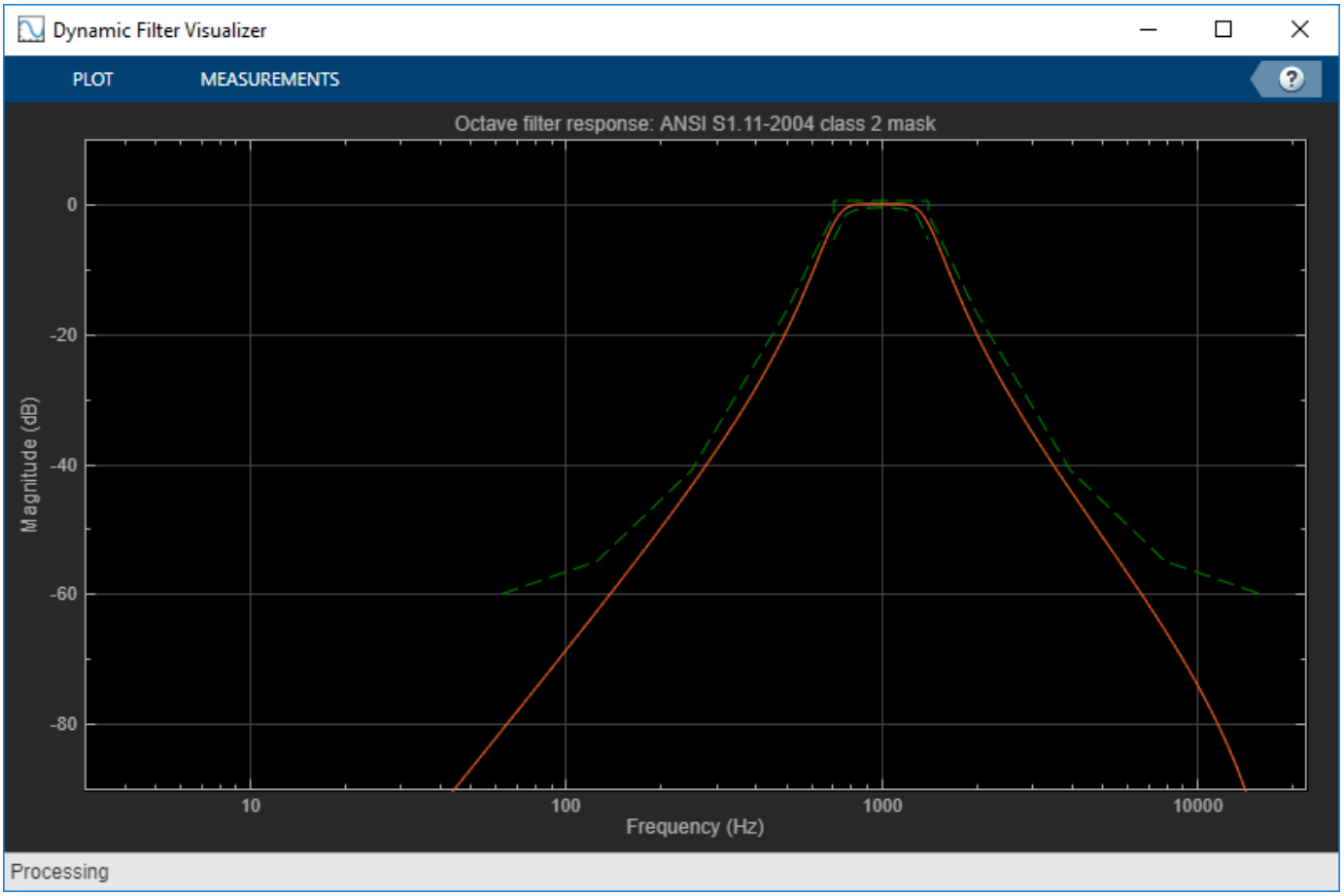


Analyze Octave Filter Design

Create an octave filter. Visualize the filter response and validate that it is class 2 compliant.

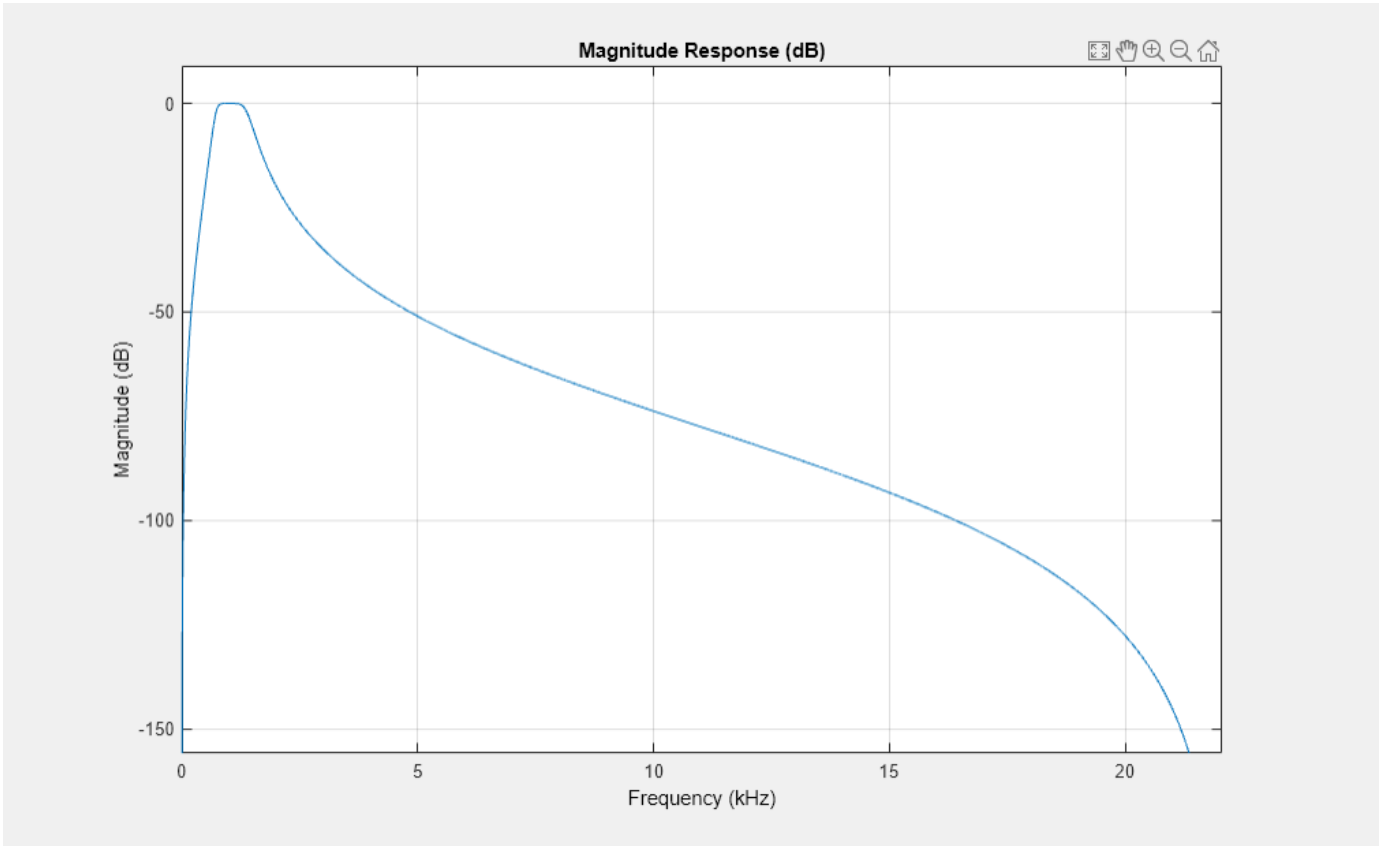
```
octFilt = octaveFilter('CenterFrequency',1000);
visualize(octFilt,'class 2')
```


PLOT
MEASUREMENTS
Octave filter response: ANSI S1.11-2004 class 2 mask
Magnitude (dB)
10
0
-10
-20
-30
-40
-50
-60
-70
-80
-90
10^0
10^1
10^2
10^3
10^4
Frequency(Hz)



Analyze the filter using `fvtool`.

```
fvtool(octFilt,'Fs',octFilt.SampleRate)
```



The `octaveFilter` object supports several filter analysis methods. For more information, use `help` at the command line:

```
help octaveFilter.helpFilterAnalysis
```

```
    The following analysis methods are available for discrete-time filter System objects:

    fvtool       -  Filter visualization tool
    info         -  Filter information
    freqz        -  Frequency response
    phasez       -  Phase response
    zerophase    -  Zero-phase response
    grpdelay     -  Group delay response
    phasedelay   -  Phase delay response
    impz         -  Impulse response
    impzlength   -  Length of impulse response
    stepz        -  Step response
    zplane       -  Pole/zero plot
    cost         -  Cost estimate for implementation of the filter System object
    measure      -  Measure characteristics of the frequency response

    outputDelay  -  Output delay value
    order        -  Filter order
    coeffs       -  Filter coefficients in a structure
    firtype      -  Determine the type (1-4) of a linear phase FIR filter System object
    tf           -  Convert to transfer function
    zpk          -  Convert to zero-pole-gain
    ss           -  Convert to state space representation
```

```
    isallpass    -  Verify if filter System object is allpass
    isfir        -  Verify if filter System object is FIR
    islinphase   -  Verify if filter System object is linear phase
    ismaxphase   -  Verify if filter System object is maximum phase
    isminphase   -  Verify if filter System object is minimum phase
    isreal       -  Verify if filter System object is minimum real
    issos        -  Verify if filter System object is in second-order sections form
    isstable     -  Verify if filter System object is stable

    realizemdl   -  Filter realization (Simulink diagram)

    specifyall   -  Fully specify fixed-point filter System object settings

    cascade      -  Create a FilterCascade System object

    Second-order sections:

    scale        -  Scale second-order sections of BiquadFilter System object
    scalecheck   -  Check scaling of BiquadFilter System object
    reorder      -  Reorder second-order sections of BiquadFilter System object
    cumsec       -  Cumulative second-order section of BiquadFilter System object
    scaleopts    -  Create an options object for second-order section scaling
    sos          -  Convert to second-order-sections (for IIRFilter System objects only)

    Fixed-Point (Fixed-Point Designer Required):

    freqrespest  -  Frequency response estimate via filtering
    freqrespopts -  Create an options object for frequency response estimate
    noisepsd     -  Power spectral density of filter output due to roundoff noise
    noisepsdopts -  Create an options object for output noise PSD computation

    Multirate Analysis:

    polyphase                - Polyphase decomposition of multirate filter System object
    gain (CIC decimator)     - Gain of CIC decimator filter System object
    gain (CIC interpolator)  - Gain of CIC interpolator filter System object

    For decimator, interpolator, or rate change filter System objects
    the analysis tools perform computations relative to the rate at
    which the filter is running. If a sampling frequency is specified,
    it is assumed that the filter is running at that rate.

Help for octaveFilter.helpFilterAnalysis is inherited from superclass dsp.internal.FilterAnalysis
```

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.

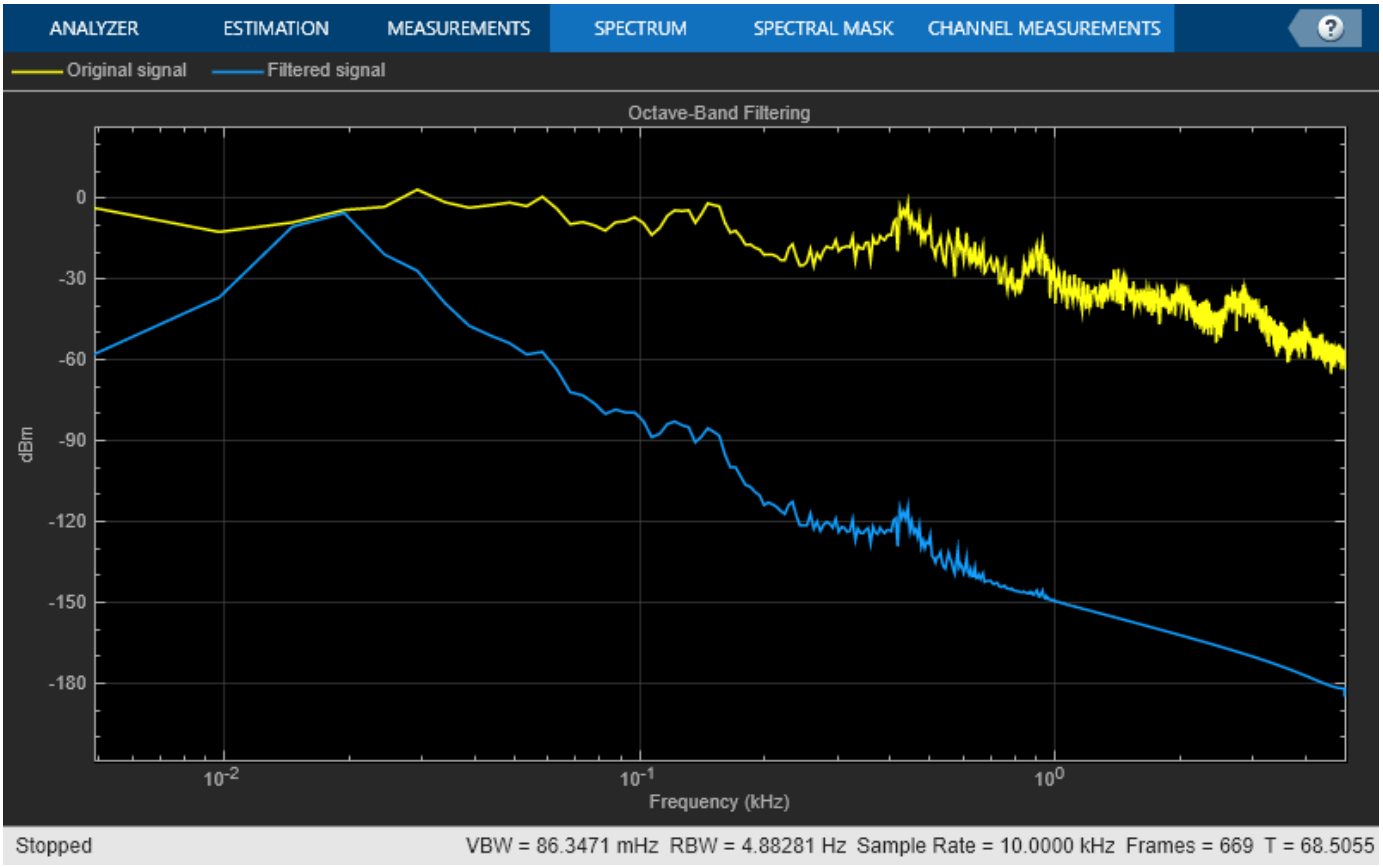
```
fileReader = dsp.AudioFileReader("Counting-16-44p1-mono-15secs.wav");
deviceWriter = audioDeviceWriter(SampleRate=fileReader.SampleRate);
```

Create a scope to visualize the filtered and unfiltered signals.

```
scope = 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`.

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.

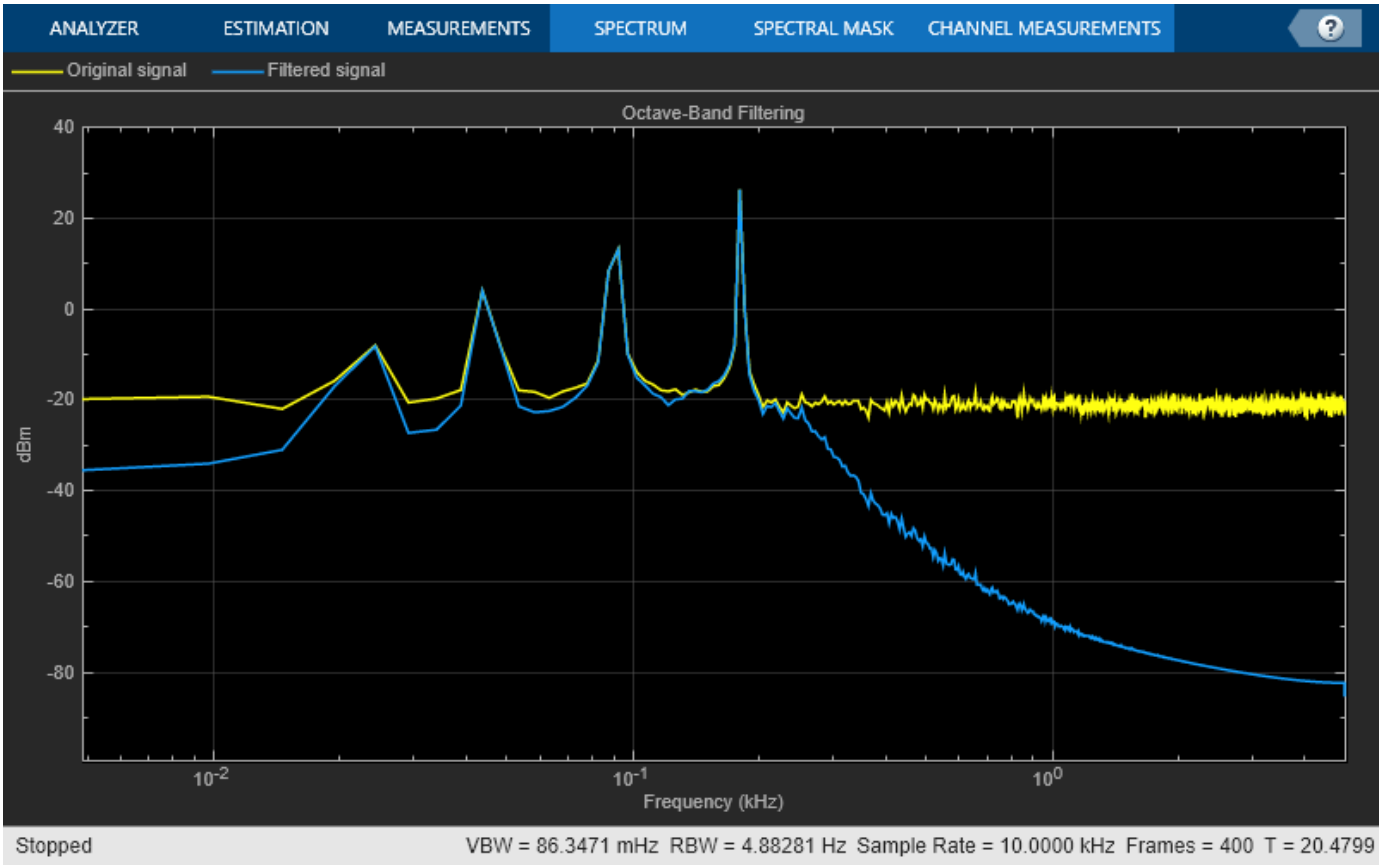
```
scope = spectrumAnalyzer( ...
    PlotAsTwoSidedSpectrum=false, ...
    FrequencyScale="log", ...
    Title="Octave-Band Filtering", ...
    ShowLegend=true, ...
    ChannelNames=["Original signal","Filtered signal"]);
```

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)
release(scope)
```



Design Compliant High-Frequency Filters

Design a sixth-order 1/3 octave filter with a sample rate of 96 kHz.

```
octFilt = octaveFilter('FilterOrder',6, ...
    'Bandwidth','1/3 octave', ...
    'SampleRate',96e3);
```

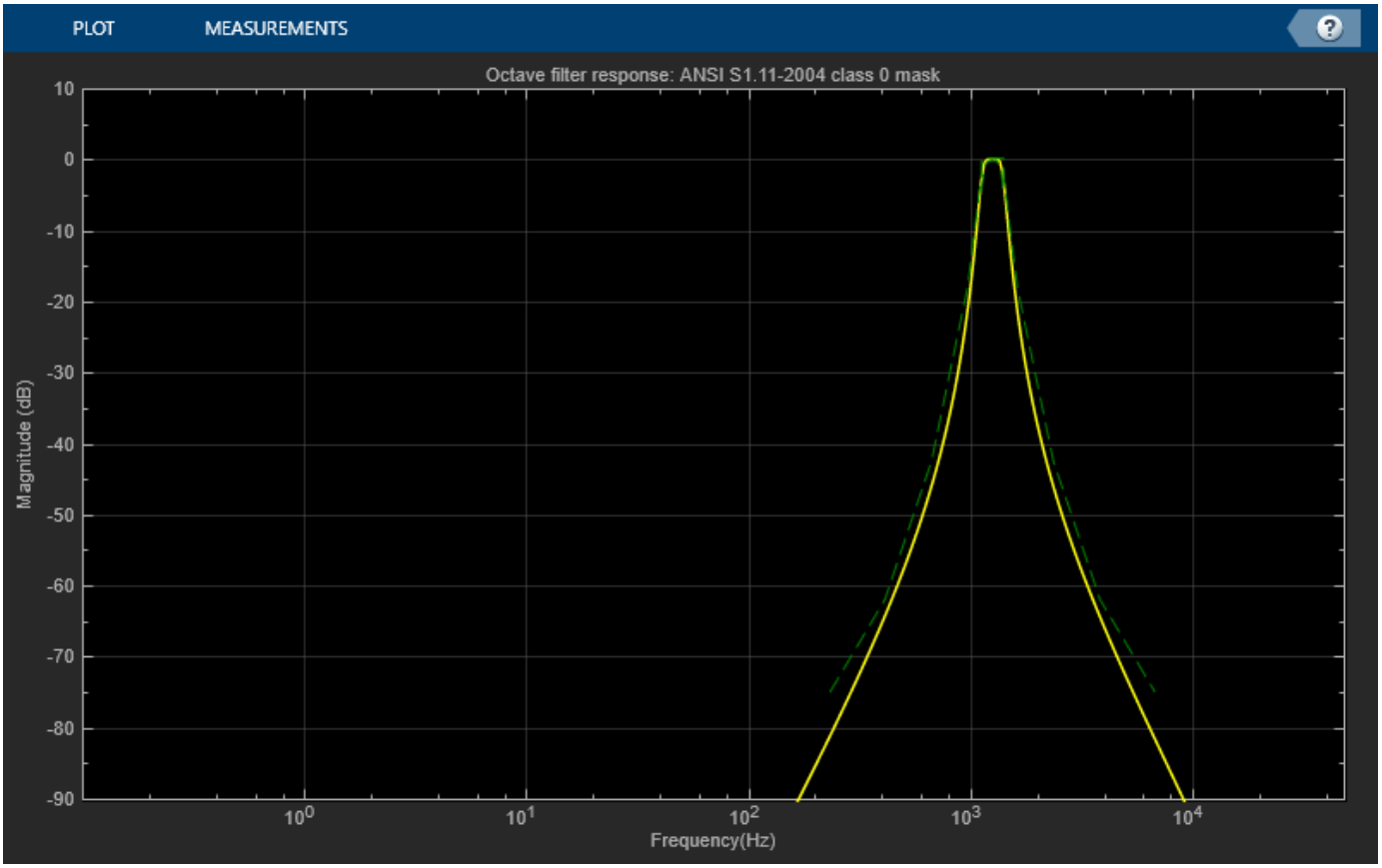
Get the center frequencies defined by the ANSI S1.11-2004 standard. The center frequencies defined by the standard depend on the `Bandwidth` and `SampleRate` properties.

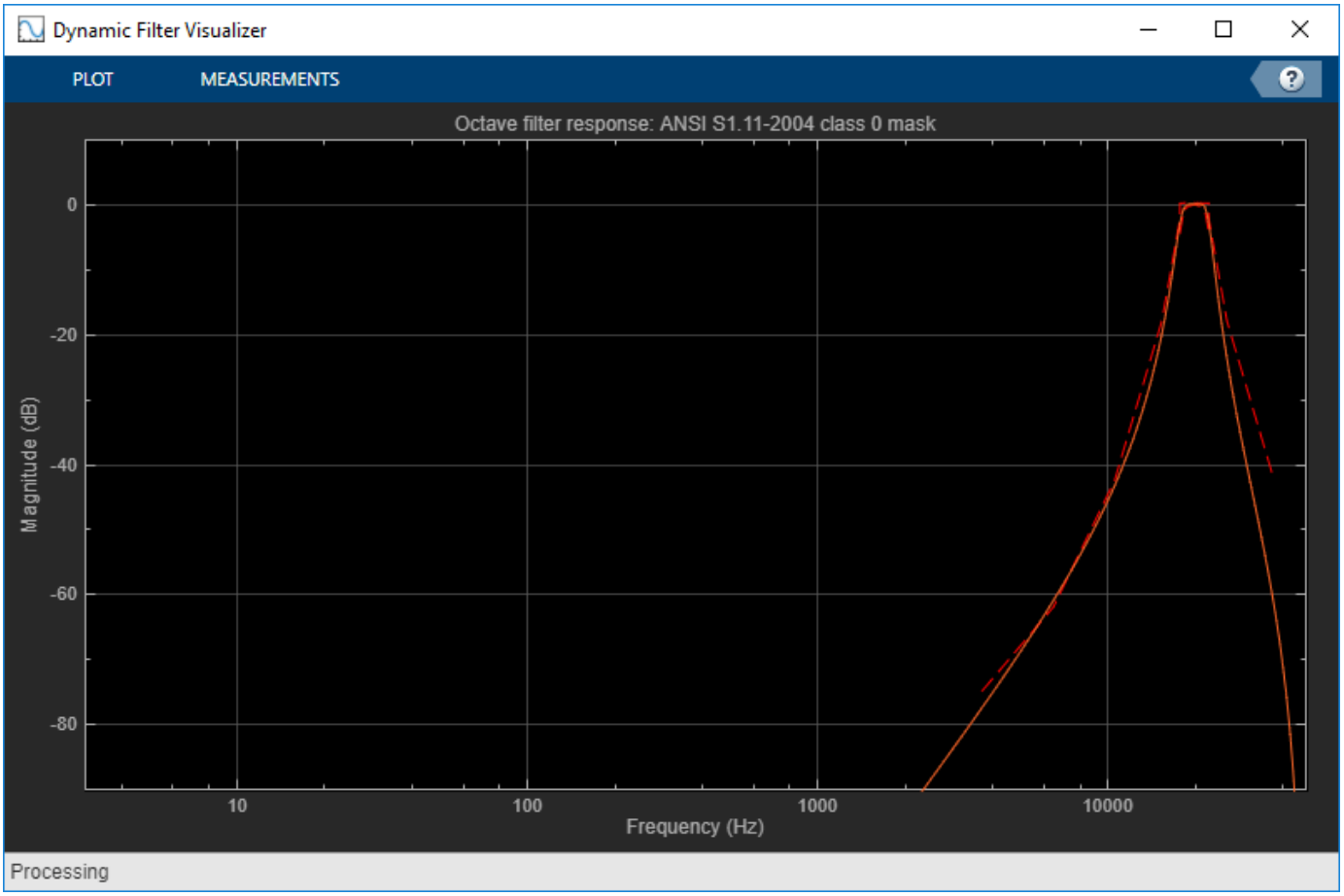
```
centerFrequencies = getANSICenterFrequencies(octFilt)
```

```
centerFrequencies = 1×53
10^4 ×

    0.0000    0.0000    0.0000    0.0001    0.0001    0.0001    0.0001    0.0001    0.0002    0.0
```

Set the center frequency of the octave filter to 19.953 kHz and visualize the response with a 'class 0' compliance mask.

```
octFilt.CenterFrequency = centerFrequencies(38);
visualize(octFilt,'class 0')
```

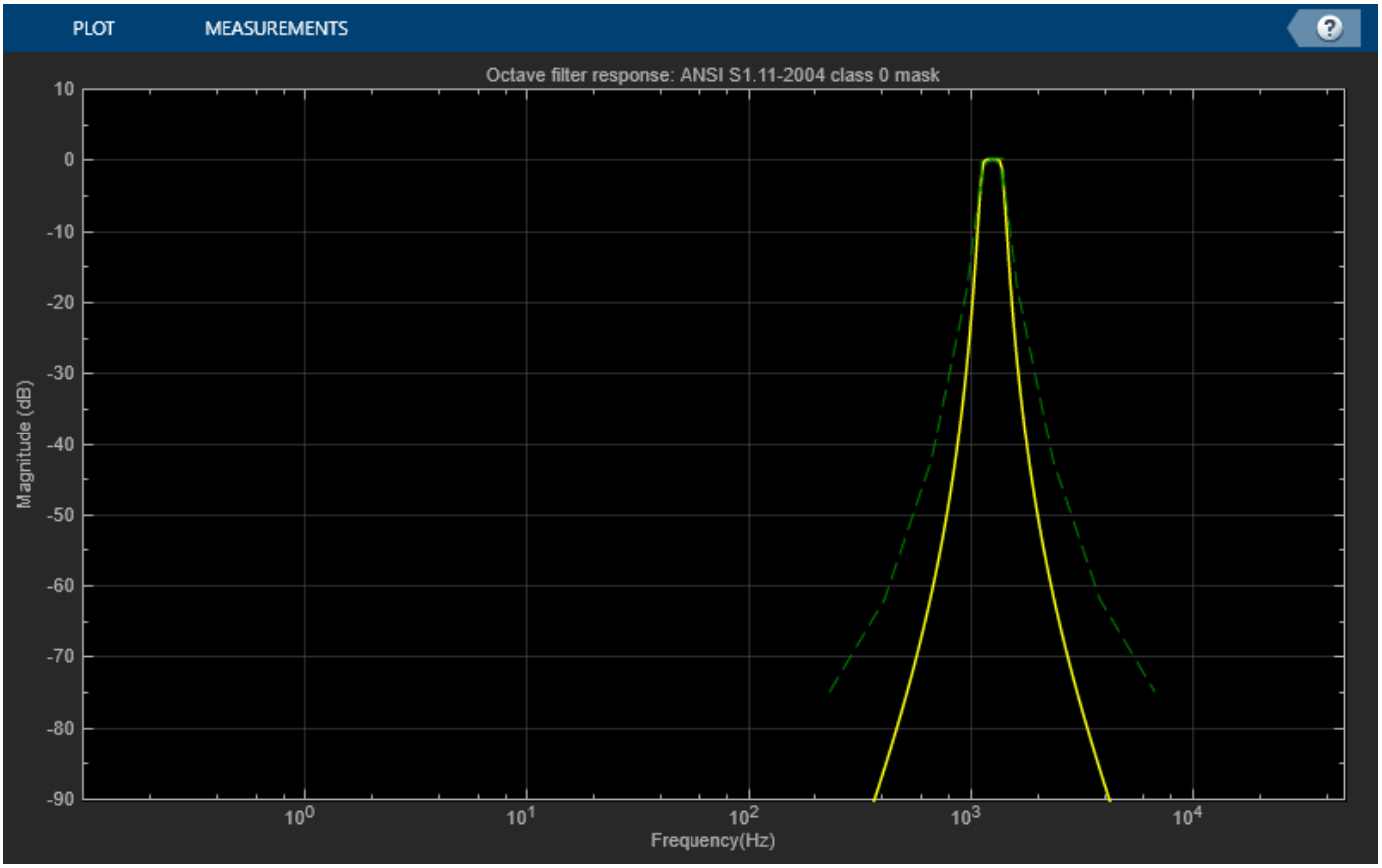

PLOT
MEASUREMENTS
Octave filter response: ANSI S1.11-2004 class 0 mask
Magnitude (dB)
10
0
-10
-20
-30
-40
-50
-60
-70
-80
-90
10^0
10^1
10^2
10^3
10^4
Frequency(Hz)

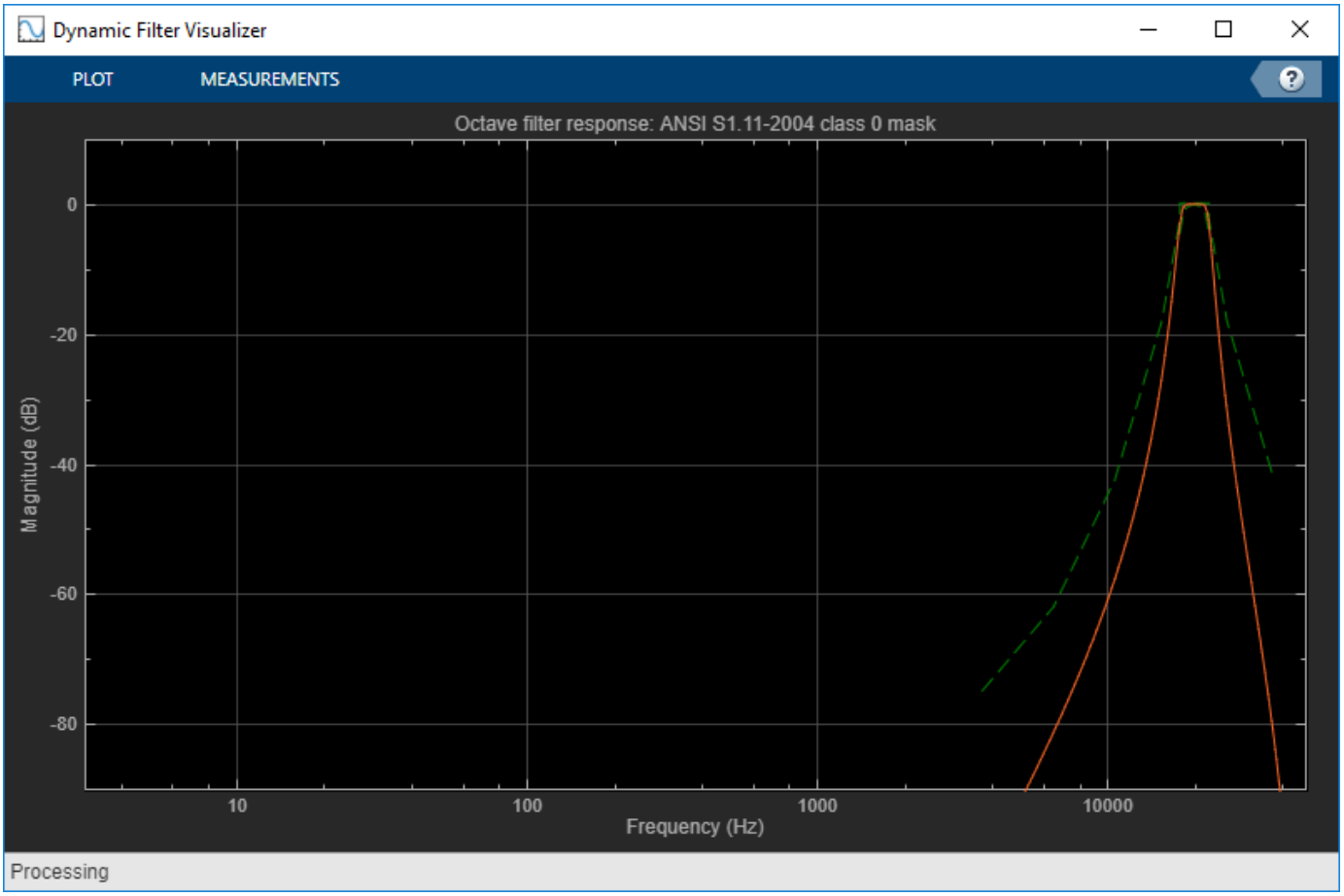


The red mask on the plot defines the bounds for the magnitude response of the filter. The magnitude response of this filter goes above the upper bound of the compliance mask around 6.6 kHz. One way to counter this is to increase the filter order so that the filter's rolloff is steeper.

To bring the octave filter design into compliance, set the octave filter order to 8.

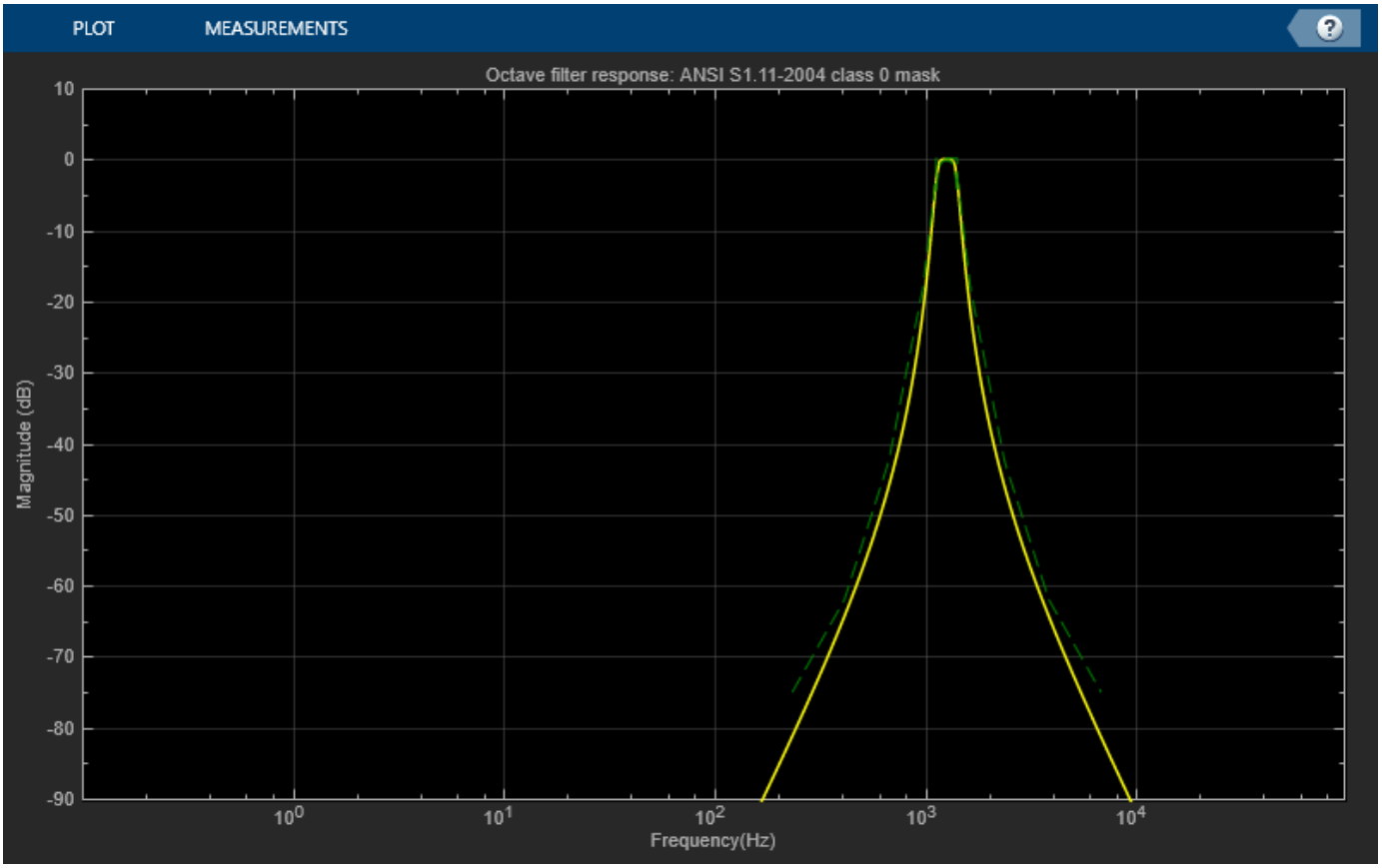
```
octFilt.FilterOrder = 8;
```

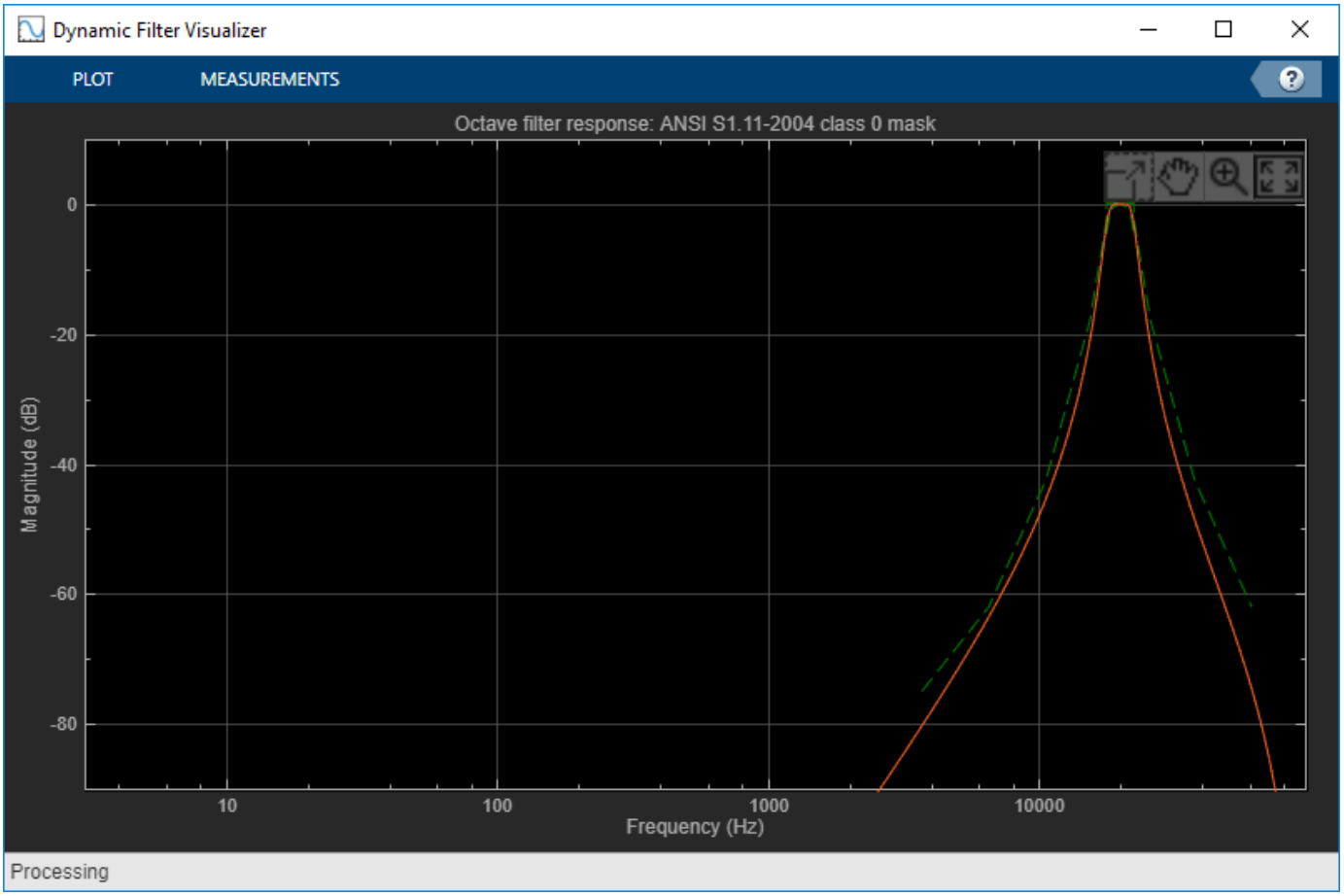

PLOT
MEASUREMENTS
Octave filter response: ANSI S1.11-2004 class 0 mask
Magnitude (dB)
10
0
-10
-20
-30
-40
-50
-60
-70
-80
-90
10^0
10^1
10^2
10^3
10^4
Frequency(Hz)



Another option to bring the octave filter design into compliance is to set the `Oversample` property to `true`. This designs and runs the filter at twice the specified `SampleRate` to reduce the effects of the bilinear transformation during the design stage.

```
octFilt.FilterOrder = 6;
octFilt.Oversample = true;
```


PLOT
MEASUREMENTS
Octave filter response: ANSI S1.11-2004 class 0 mask
Magnitude (dB)
10
0
-10
-20
-30
-40
-50
-60
-70
-80
-90
10^0
10^1
10^2
10^3
10^4
Frequency(Hz)



Design Compliant Low-Frequency Filters

Design a sixth-order 2/3 octave filter with a 96 kHz sample rate.

```
octFilt = octaveFilter('FilterOrder',6, ...
    'Bandwidth','2/3 octave', ...
    'SampleRate',96e3);
```

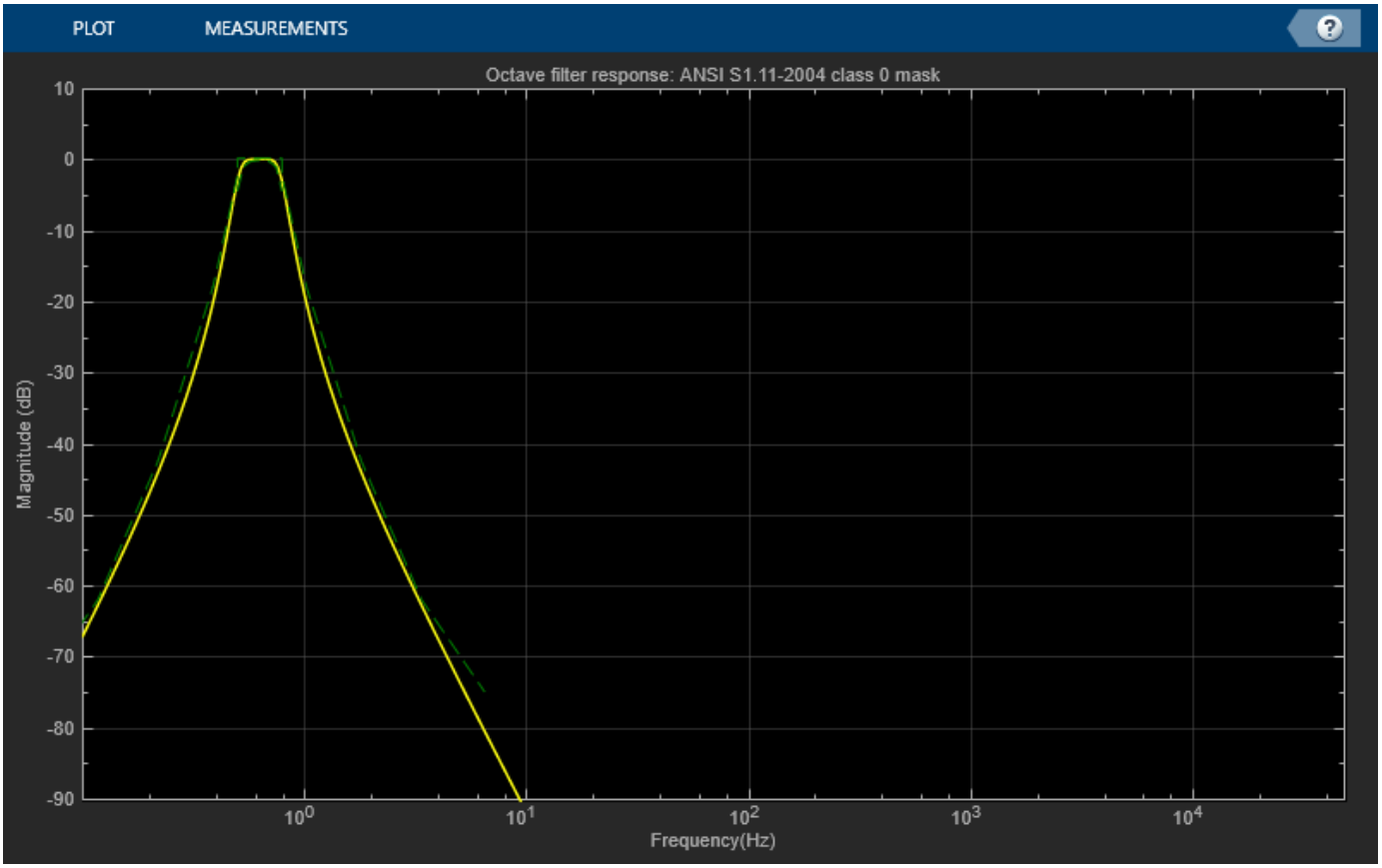
Get the center frequencies defined by the ANSI S1.11-2004 standard. The center frequencies defined by the standard depend on the `Bandwidth` and `SampleRate` properties.

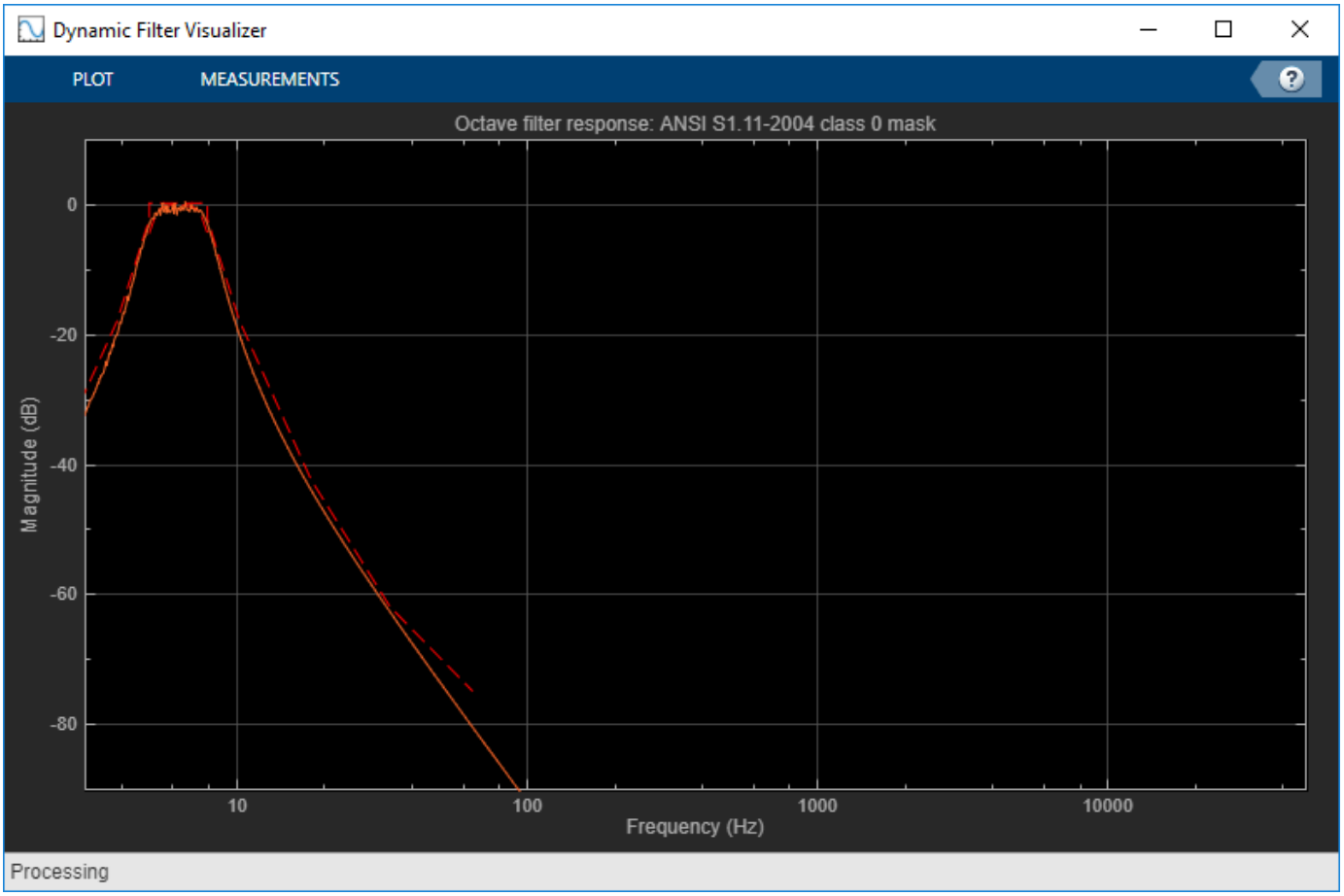
```
centerFrequencies = getANSICenterFrequencies(octFilt)
```

```
centerFrequencies = 1×25
10^4 ×

    0.0000    0.0001    0.0001    0.0002    0.0003    0.0004    0.0006    0.0010    0.0016    0.0
```

Set the center frequency of the octave filter to ~6 Hz and visualize the response with a 'class 0' compliance mask.

```
octFilt.CenterFrequency = centerFrequencies(2);
visualize(octFilt,'class 0')
```



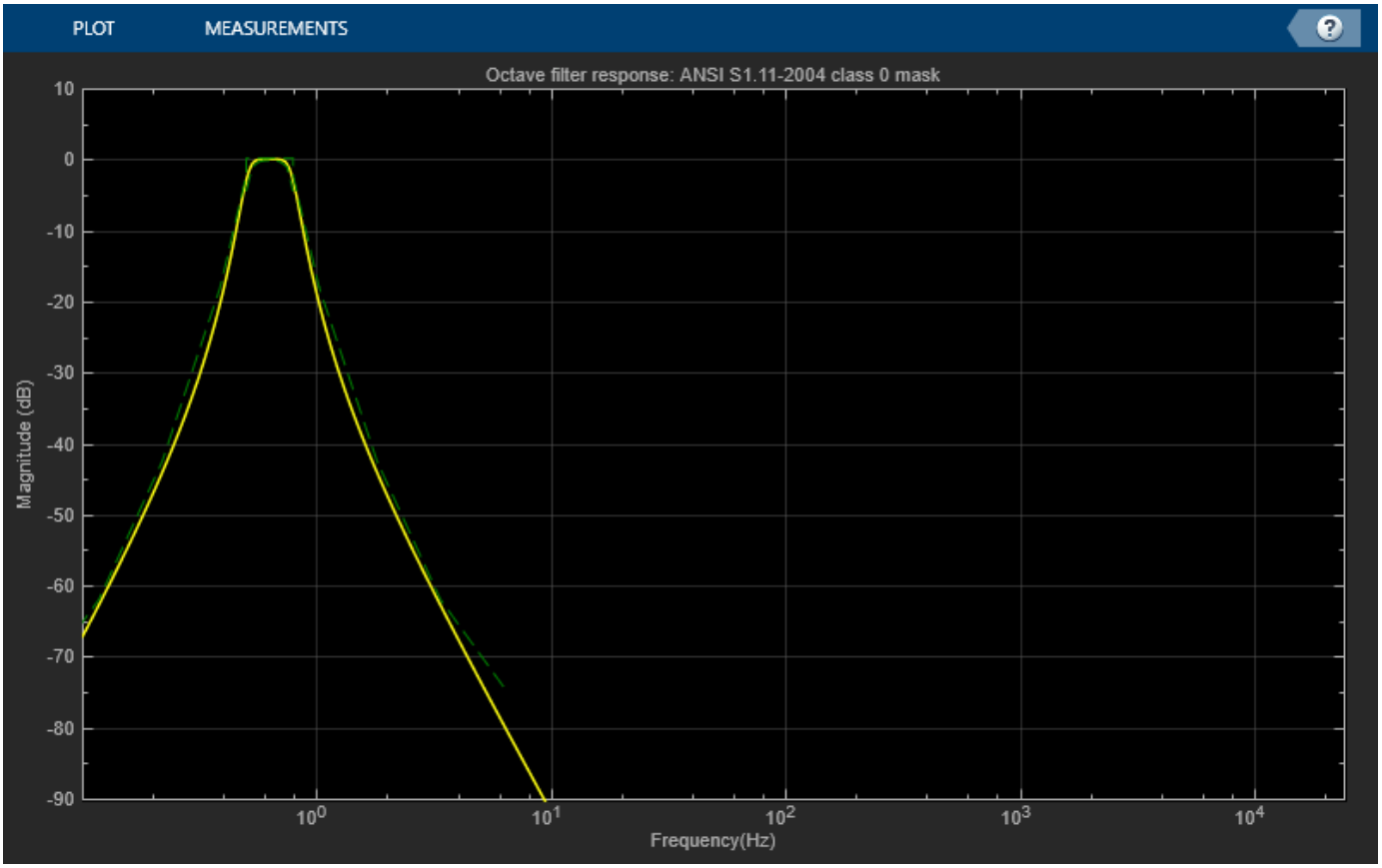


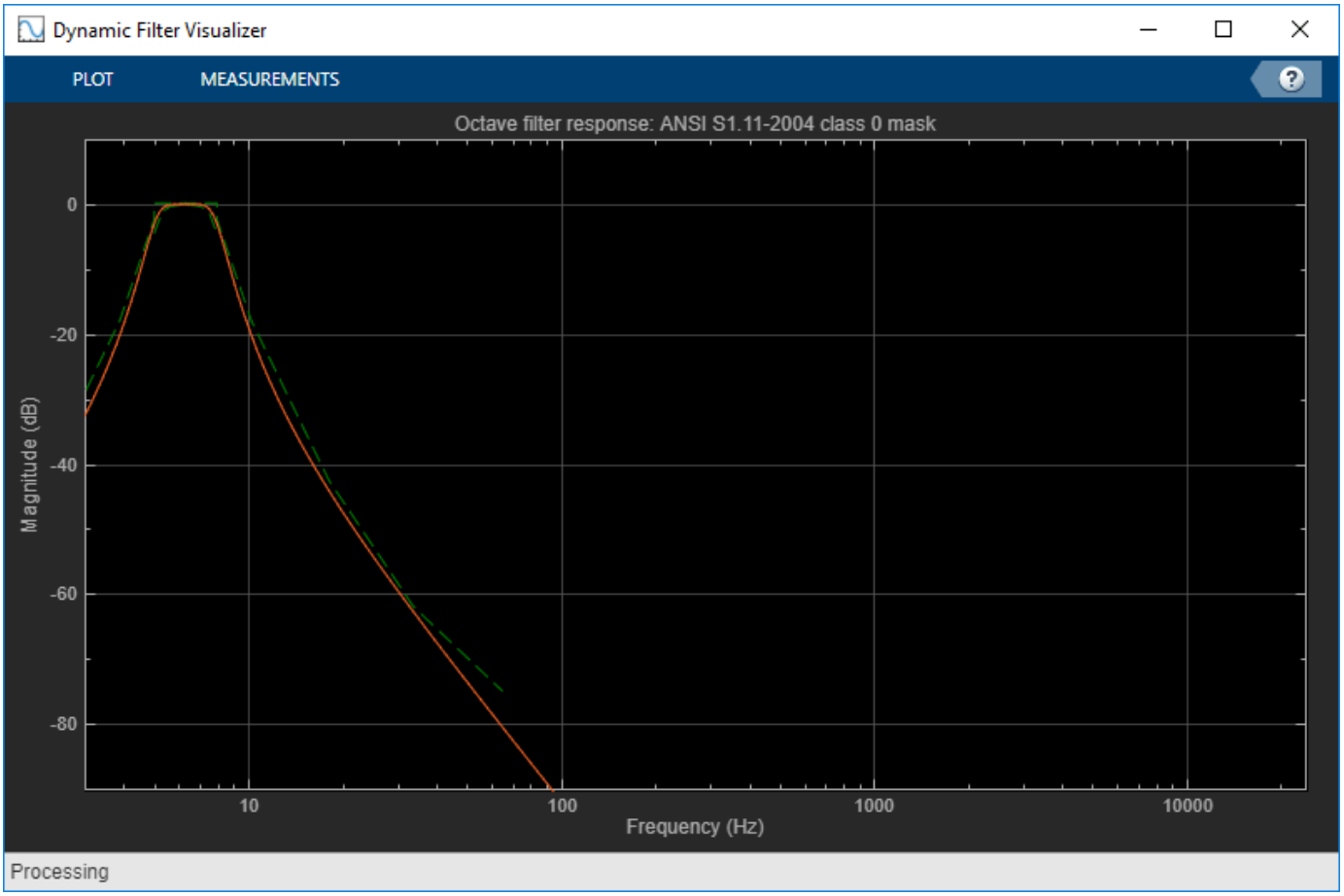
The red mask on the plot defines the bounds for the magnitude response of the filter. The magnitude response of this filter goes below the lower bound of the compliance mask between 5.5 and 7.5 Hz.

Low-frequency filters in an octave filter bank have very low normalized center frequencies, and the filters designed for them have poles that are almost on the unit circle. To make this filter ANSI compliant, it has to be designed and operated at a lower sample rate.

To bring the octave filter design into compliance, set the sample rate to 48 kHz.

```
octFilt.SampleRate = 48e3;
```

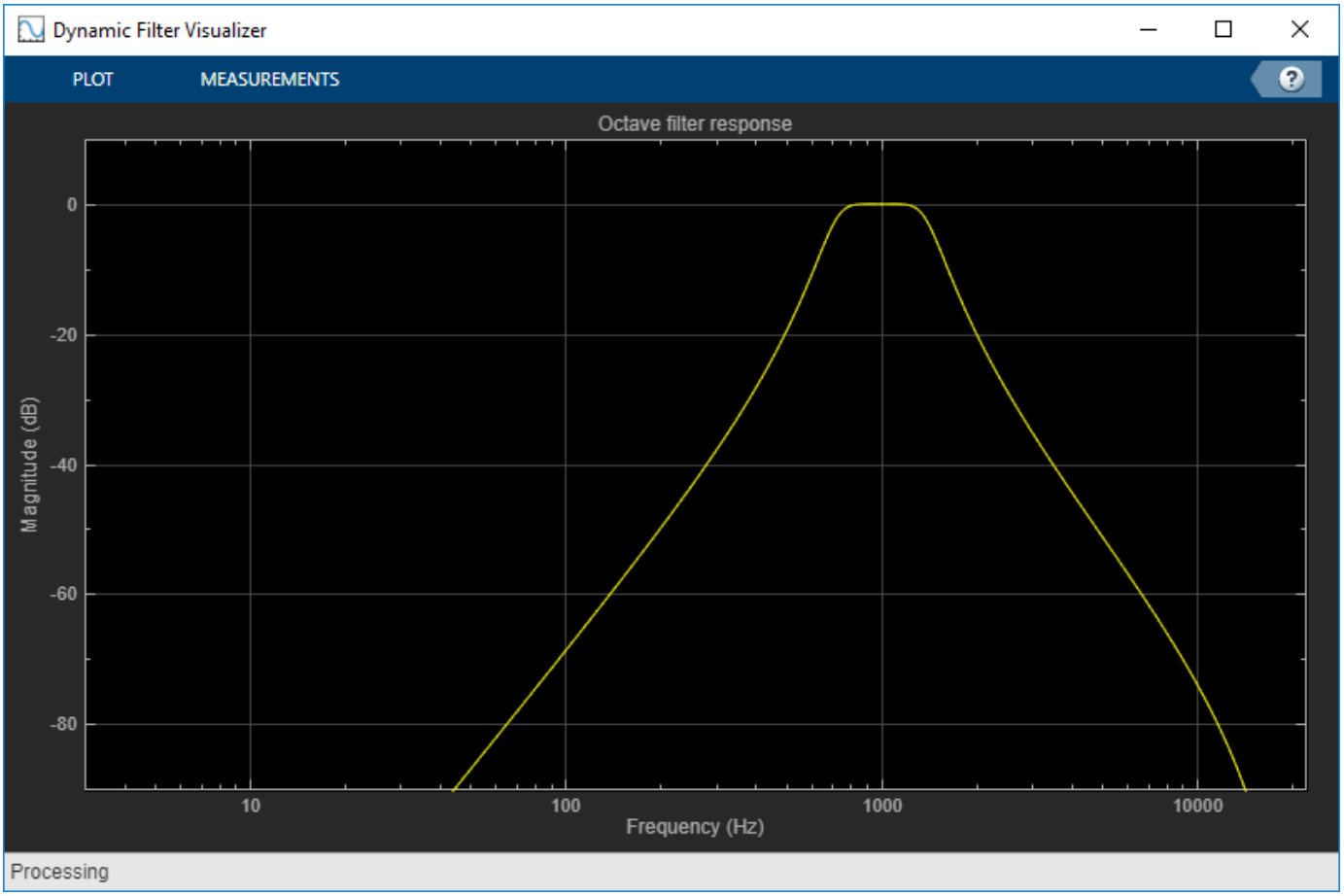

PLOT
MEASUREMENTS
Octave filter response: ANSI S1.11-2004 class 0 mask
Magnitude (dB)
10
0
-10
-20
-30
-40
-50
-60
-70
-80
-90
10^0
10^1
10^2
10^3
10^4
Frequency(Hz)



Tune Octave Filter Parameters

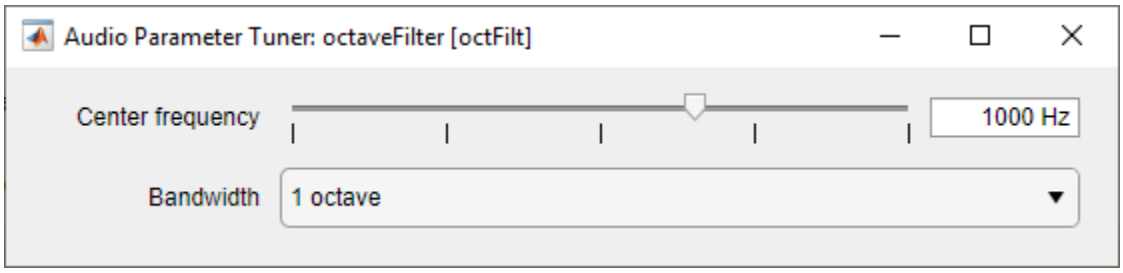
Create a `dsp.AudioFileReader` to read in audio frame-by-frame. Create a `audioDeviceWriter` to write audio to your sound card. Create an `octaveFilter` to process the audio data. Call `visualize` to plot the frequency response of the octave filter.

```
frameLength = 1024;
fileReader = dsp.AudioFileReader('RockDrums-44p1-stereo-11secs.mp3', ...
    'SamplesPerFrame',frameLength);
deviceWriter = audioDeviceWriter('SampleRate',fileReader.SampleRate);

octFilt = octaveFilter('SampleRate',fileReader.SampleRate);
visualize(octFilt)
```



Call `parameterTuner` to open a UI to tune parameters of the `octaveFilter` while streaming.

```
parameterTuner(octFilt)
```



In an audio stream loop:

1. Read in a frame of audio from the file.
2. Apply octave filtering.
3. Write the frame of audio to your audio device for listening.

While streaming, tune parameters of the octave filter and listen to the effect.

```
while ~isDone(fileReader)
    audioIn = fileReader();
```

```
    audioOut = octFilt(audioIn);
    deviceWriter(audioOut);
    drawnow limitrate % required to update parameter
end
```

As a best practice, release your objects once done.

```
release(deviceWriter)
release(fileReader)
release(octFilt)
```

More About

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.

Tips

The `createAudioPluginClass` and `configureMIDI` functions map tunable properties of the `octaveFilter` to user-facing parameters:

Property	Range	Mapping	Units
`CenterFrequency`	[3, 22000]	log	Hz
`Bandwidth`	`'1 octave'`, `'2/3 octave'`, `'1/2 octave'`, `'1/3 octave'`, `'1/6 octave'`, `'1/12 octave'`, `'1/24 octave'`, or `'1/48 octave'`	Your MIDI controller range is discretized into seven levels, corresponding to the seven `Bandwidth` choices.	--

Algorithms

Octave Bandwidth to Band Edge Conversion

The `octaveFilter` System object uses the specified center frequency and filter bandwidth in octaves to determine the normalized band edges [2].

The object computes the upper and lower band edge frequencies:

$$f_{pa} = f_c \times G^{-1/2b}$$

$$f_{pb} = f_c \times G^{1/2b}$$

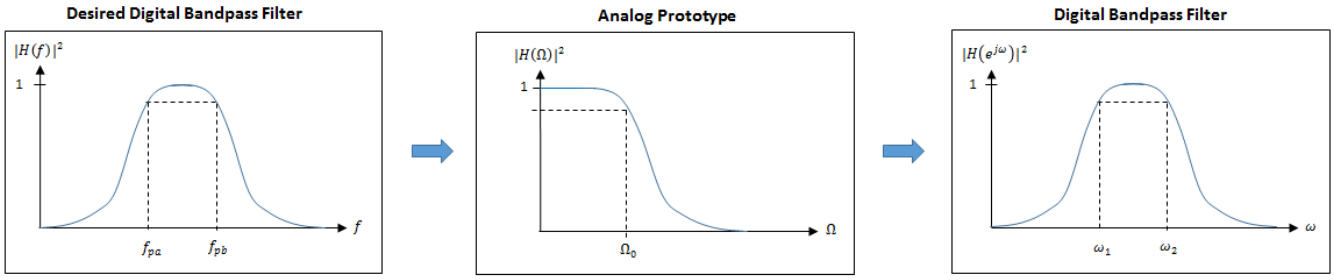
- f_c is the normalized center frequency specified by the `CenterFrequency` property.
- b is the octave bandwidth specified by the `Bandwidth` property. For example, if `Bandwidth` is specified as `'1/3 octave'`, the value of b is 3.
- G is a conversion constant:

$$G = 10^{3/10}.$$

Digital Filter Design

The `octaveFilter` System object implements a higher-order digital bandpass filter design method specified in [1].

In this design method, a desired digital bandpass filter maps to a Butterworth lowpass analog prototype, which is then mapped back to a digital bandpass filter:



1. The analog Butterworth filter is expressed as a cascade of second-order sections:

$$H(s) = H_1(s)H_2(s)\cdots H_{2N}(s),$$

where:

$$H_i(s) = \frac{1}{1 - 2\frac{s}{\Omega_0}\cos\theta_i + \frac{s^2}{\Omega_0^2}}, \quad i = 1, 2, \ldots, 2N$$

$$\theta_i = \frac{\pi}{2N}(N - 1 + 2i), \quad i = 1, 2, .., 2N$$

N is the filter order specified by the `FilterOrder` property.

2. The analog Butterworth filter is mapped to a digital filter using a bandpass version of the bilinear transformation:

$$s = \frac{1 - cz^{-1} + z^{-2}}{1 - z^{-2}},$$

where

$$c = \frac{\sin(\omega_{\text{pa}} + \omega_{\text{pb}})}{\sin\omega_{\text{pa}} + \sin\omega_{\text{pb}}}.$$

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.

Version History

Introduced in R2016b

References

[1] Orfanidis, Sophocles J. *Introduction to Signal Processing*. Englewood Cliffs, NJ: Prentice Hall, 2010.

[2] Acoustical Society of America. *American National Standard Specification for Octave-Band and Fractional-Octave-Band Analog and Digital Filters*. ANSI S1.11-2004. Melville, NY: Acoustical Society of America, 2009.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

“System Objects in MATLAB Code Generation” (MATLAB Coder)

See Also

Octave Filter | `multibandParametricEQ` | `weightingFilter` | `dsp.BiquadFilter` | `octaveFilterBank`

Topics

“Octave-Band and Fractional Octave-Band Filters”

getANSICenterFrequencies

Get the list of valid ANSI S1.11-2004 center frequencies

Syntax

```
centerFrequencies = getANSICenterFrequencies(octFilt)
```

Description

`centerFrequencies = getANSICenterFrequencies(octFilt)` returns a vector of valid center frequencies as specified by the ANSI S1.11-2004 standard.

Examples

Get ANSI Center Frequencies

Create an object of the `octaveFilter` System object™. Call `getANSICenterFrequencies` to get a list of valid center frequencies.

```
octFilt = octaveFilter;
centerFrequencies = getANSICenterFrequencies(octFilt)
```

```
centerFrequencies = 1×15
10^3 ×

    0.0005    0.0010    0.0020    0.0040    0.0079    0.0158    0.0316    0.0631    0.1259    0.2
```

Input Arguments

`octFilt` — Object of `octaveFilter`

object

Object of the `octaveFilter` System object.

Output Arguments

`centerFrequencies` — Center frequencies

vector

Center frequencies specified by the ANSI S1.11-2004 standard, returned as a vector.

The range for computing valid center frequencies is 3 Hz to (Fs/2) Hz, where the `SampleRate` property of your octave filter defines Fs.

Data Types: `single` | `double`

Version History

Introduced in R2016b

See Also

Blocks
Octave Filter

Topics
“Octave-Band and Fractional Octave-Band Filters”

isStandardCompliant

Verify octave filter design is ANSI S1.11-2004 compliant

Syntax

```
complianceStatus = isStandardCompliant(octFilt,classType)
[complianceStatus,centerFreq] = isStandardCompliant(octFilt,classType)
```

Description

`complianceStatus = isStandardCompliant(octFilt,classType)` returns a logical scalar, `complianceStatus`, indicating whether the `complianceStatus` filter design is compliant with the ANSI S1.11-2004 standard for `classType`.

The mask used to determine compliance is centered on the nearest ANSI-compliant center frequency that ensures the center frequency of the object falls between the upper and lower band edges of the mask.

`[complianceStatus,centerFreq] = isStandardCompliant(octFilt,classType)` also returns the ANSI-compliant center frequency used to create the mask.

Examples

Verify Standard Compliance

Create an object of the `octaveFilter` System object™. Call `isStandardCompliant`, specifying the compliance class type to check as the second argument.

```
octFilt = octaveFilter;
complianceStatus = isStandardCompliant(octFilt,'class 2')
```

```
complianceStatus = logical
   1
```

Get ANSI-Compliant Center Frequency

Create an object of the octaveFilter System object. Check the compliance to class 0 status of your object, and get the center frequency used to create the compliance mask.

```
octFilt = octaveFilter('CenterFrequency',1266);
[compliant, centerFreq] = isStandardCompliant(octFilt,'class 0')
```

```
compliant = logical
   0


centerFreq = 1000
```

Input Arguments

`octFilt` — Object of `octaveFilter`

object

Object of the `octaveFilter` System object.

`classType` — Compliance class type

`'class 0'` | `'class 1'` | `'class 2'`

Compliance class type to verify, specified as `'class 0'`, `'class 1'`, or `'class 2'`.

Data Types: `char`

Output Arguments

`complianceStatus` — Compliance status of filter design

scalar

Compliance status of filter design, returned as a logical scalar. The compliance status indicates whether the `octFilt` filter design is compliant with the ANSI S1.11-2004 standard for `classType`.

If your octave filter is noncompliant, try any of the following:

- Set the center frequency to one of the values returned by `getANSICenterFrequencies`
- Increase filter order
- Increase sample rate

Data Types: `logical`

`centerFreq` — Center frequency of mask

scalar

Center frequency used to create the compliance mask, returned as a scalar.

Data Types: `single` | `double`

Version History

Introduced in R2016b

See Also

Octave Filter | `multibandParametricEQ` | `weightingFilter` | `dsp.BiquadFilter`

Topics

“Octave-Band and Fractional Octave-Band Filters”

visualize

Visualize and validate filter response

Syntax

```
visualize(octFilt)
visualize(octFilt,N)
visualize( ___ ,mType)
hvsz = visualize( ___ )
```

Description

`visualize(octFilt)` plots the magnitude response of the octave-band filter `octFilt`. The plot is updated automatically when properties of the object change.

`visualize(octFilt,N)` uses an `N`-point FFT to calculate the magnitude response.

`visualize( ___ ,mType)` creates a mask based on the class of filter specified by `mType`, using either of the previous syntaxes. Specify `mType` as `'class 0'`, `'class 1'`, or `'class 2'`. The mask attenuation limits are defined in the ANSI S1.11-2004 standard. The mask center frequency is the ANSI standard center frequency, with band edge frequencies on either side of the `CenterFrequency` set in `octFilt`.

- If the mask is green, the design is compliant with the ANSI S1.11-2004 standard.
- If the mask is red, the design breaks compliance.

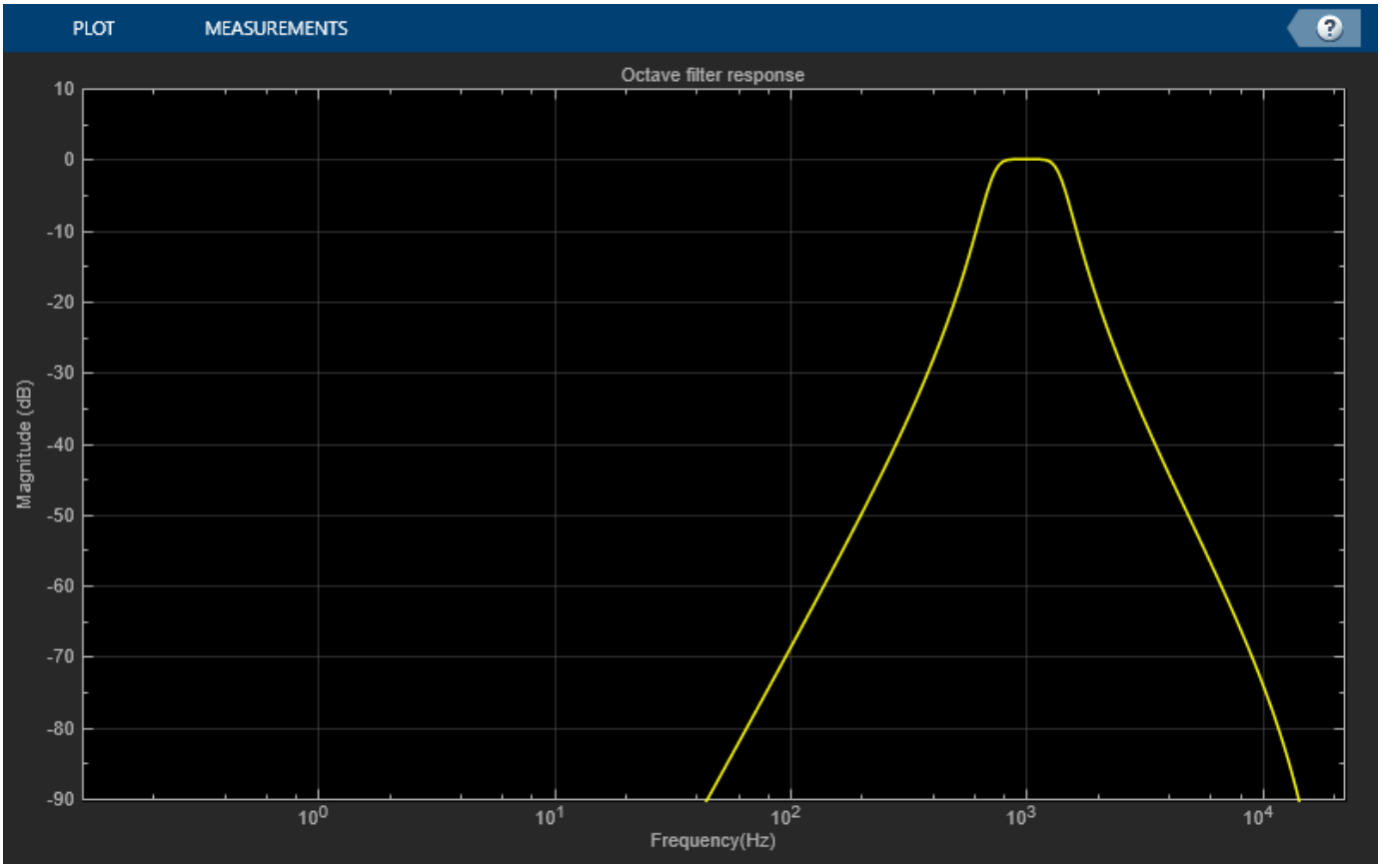
`hvsz = visualize( ___ )` returns a handle to the visualizer as a `dsp.DynamicFilterVisualizer` object when called with any of the previous syntaxes.

Examples

Plot Octave Filter Magnitude Response

Create an `octaveFilter` System object™ and then plot the magnitude response of the filter.

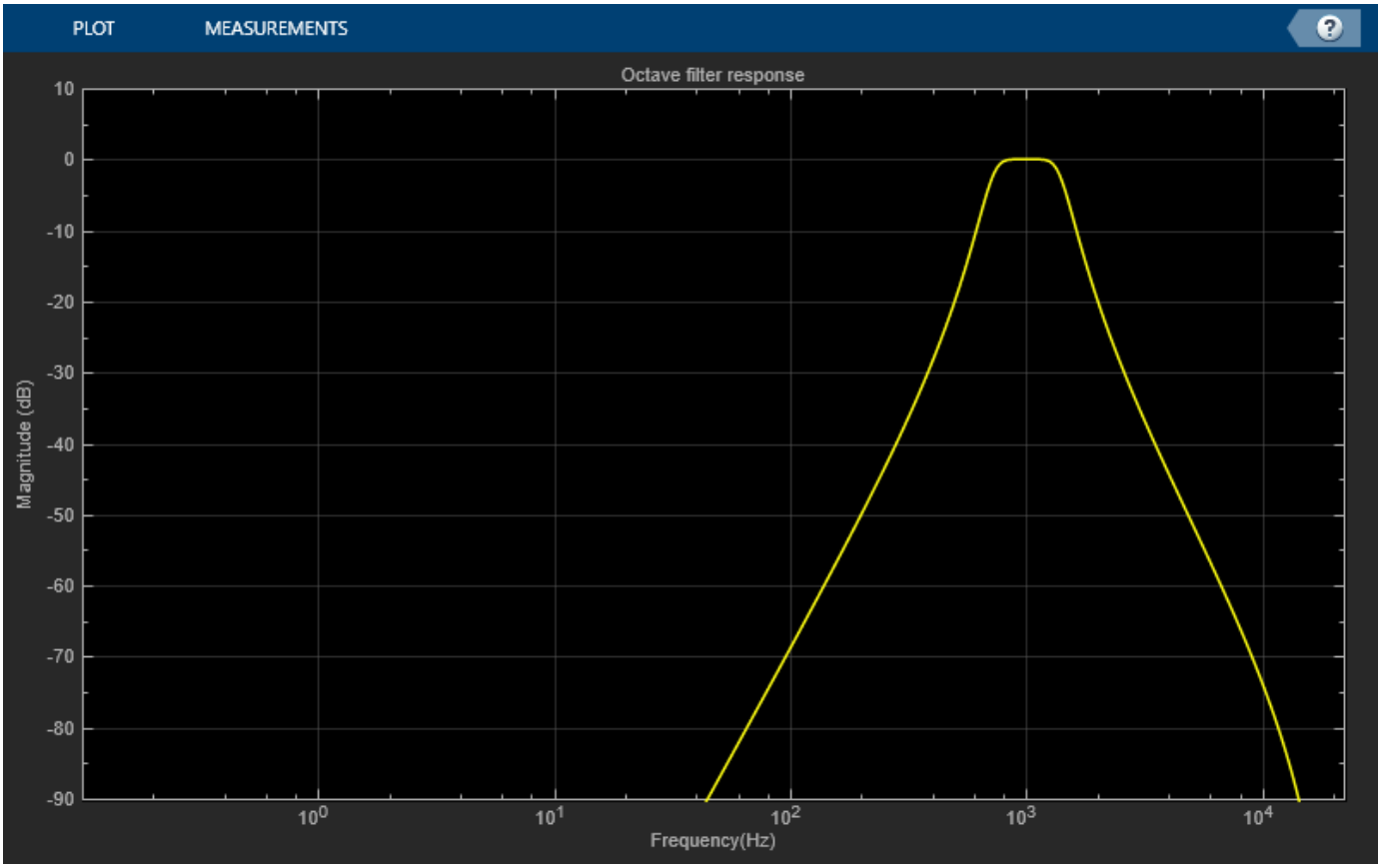
```
octFilt = octaveFilter;
visualize(octFilt)
```



Specify Number of Frequency Bins

Create an `octaveFilter` System object™. Plot a 5096-point frequency representation.

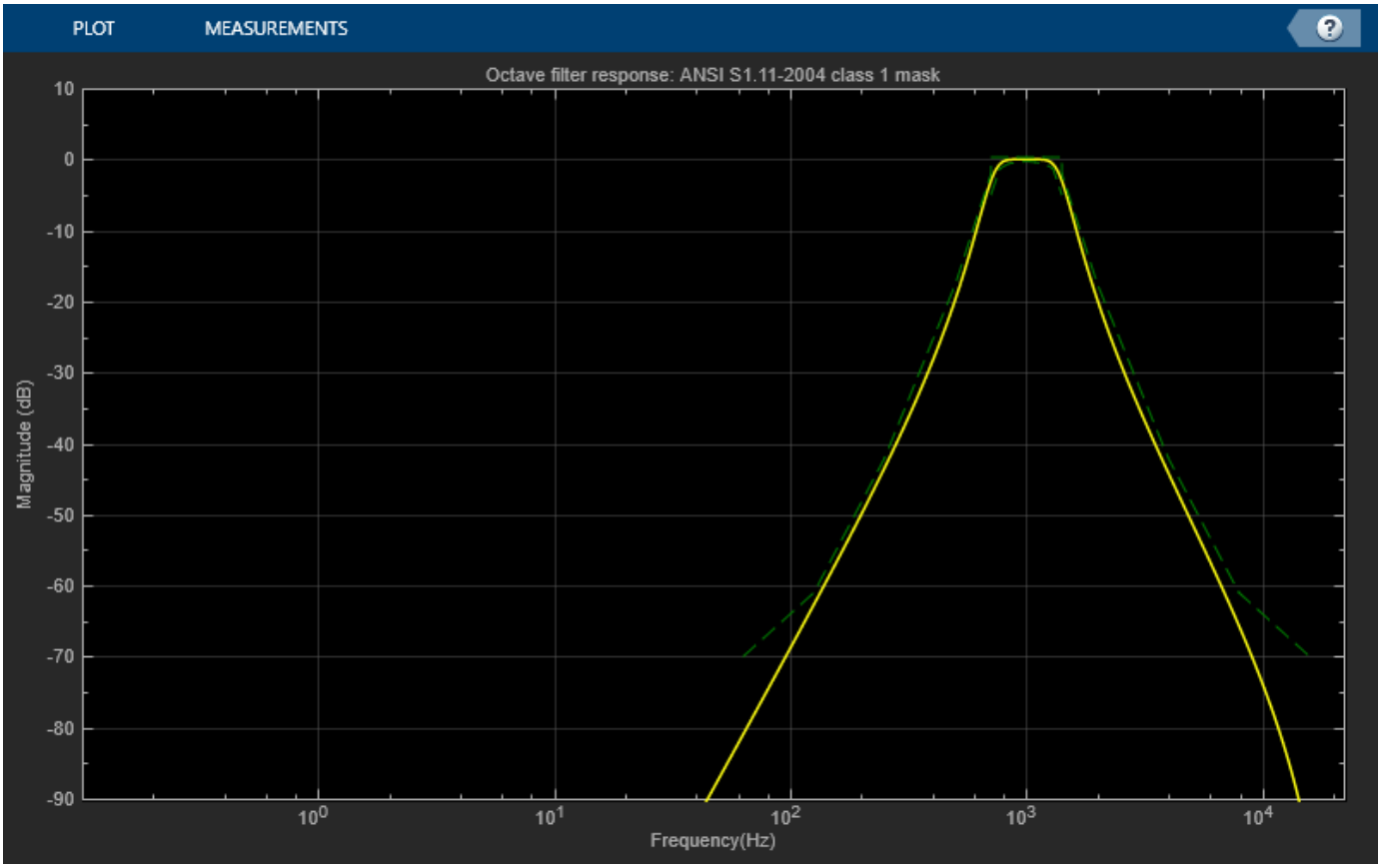
```
octFilt = octaveFilter;
visualize(octFilt,5096)
```



Visualize Standard-Compliance Mask

Create an `octaveFilter` System object™. Visualize the class 1 compliance of the filter design.

```
octFilt = octaveFilter;
visualize(octFilt,'class 1')
```



Input Arguments

`octFilt` — Object of `octaveFilter`

object

Object of the `octaveFilter` System object.

`N` — Number of DFT bins

`2048` | positive scalar

Number of DFT bins in frequency-domain representation, specified as a positive scalar. The default is `2048`.

Data Types: `single` | `double`

`mType` — Type of mask

`'class 0'` | `'class 1'` | `'class 2'`

Type of mask, specified as `'class 0'`, `'class 1`, or `'class 2'`.

The mask attenuation limits are defined in the ANSI S1.11-2004 standard. The mask center frequency is the ANSI standard center frequency, with band edge frequencies on either side of the `CenterFrequency` set in `octFilt`.

- If the mask is green, the design is compliant with the ANSI S1.11-2004 standard.
- If the mask is red, the design breaks compliance.

Data Types: `char`

Version History

Introduced in R2016b

See Also

Topics

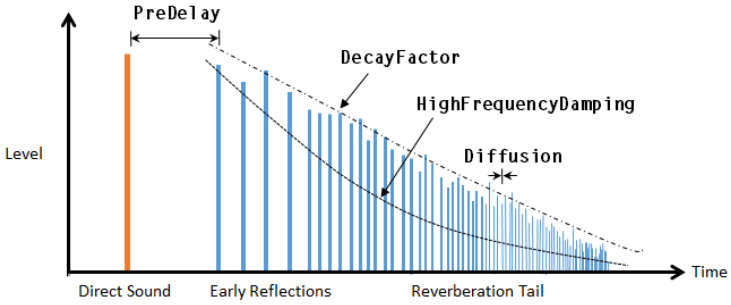
“Octave-Band and Fractional Octave-Band Filters”

reverberator

Add reverberation to audio signal

Description

The `reverberator` System object adds reverberation to mono or stereo audio signals.



To add reverberation to your input:

1. Create the `reverberator` object and set its properties.
2. Call the object with arguments, as if it were a function.

To learn more about how System objects work, see What Are System Objects?

Creation

Syntax

```
reverb = reverberator
reverb = reverberator(Name,Value)
```

Description

`reverb = reverberator` creates a System object, `reverb`, that adds artificial reverberation to an audio signal.

`reverb = reverberator(Name,Value)` sets each property `Name` to the specified `Value`. Unspecified properties have default values.

Example: `reverb = reverberator('PreDelay',0.5,'WetDryMix',1)` creates a System object, `reverb`, with a 0.5 second pre-delay and a wet-to-dry mix ratio of one.

Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see System Design in MATLAB Using System Objects.

`PreDelay` — Pre-delay for reverberation (s)

`0` (default) | real positive scalar

Pre-delay for reverberation in seconds, specified as a real scalar in the range [0, 1].

Pre-delay for reverberation is the time between hearing direct sound and the first early reflection. The value of `PreDelay` is proportional to the size of the room being modeled.

Tunable: Yes

Data Types: `single` | `double`

`HighCutFrequency` — Lowpass filter cutoff (Hz)

`20000` (default) | real positive scalar

Lowpass filter cutoff in Hz, specified as a real positive scalar in the range 0 to $\left(\frac{SampleRate}{2}\right)$.

Lowpass filter cutoff is the –3 dB cutoff frequency for the single-pole lowpass filter at the front of the reverberator structure. It prevents the application of reverberation to high-frequency components of the input.

Tunable: Yes

Data Types: `single` | `double`

`Diffusion` — Density of reverb tail

`0.5` (default) | real scalar

Density of reverb tail, specified as a real positive scalar in the range [0, 1].

`Diffusion` is proportional to the rate at which the reverb tail builds in density. Increasing `Diffusion` pushes the reflections closer together, thickening the sound. Reducing `Diffusion` creates more discrete echoes.

Tunable: Yes

Data Types: `single` | `double`

`DecayFactor` — Decay factor of reverb tail

`0.5` (default) | real scalar

Decay factor of reverb tail, specified as a real positive scalar in the range [0, 1].

`DecayFactor` is inversely proportional to the time it takes for reflections to run out of energy. To model a large room, use a long reverb tail (low decay factor). To model a small room, use a short reverb tail (high decay factor).

Tunable: Yes

Data Types: `single` | `double`

`HighFrequencyDamping` — High-frequency damping
`0.0005` (default) | real scalar

High-frequency damping, specified as a real positive scalar in the range [0, 1].

`HighFrequencyDamping` is proportional to the attenuation of high frequencies in the reverberation output. Setting `HighFrequencyDamping` to a large value makes high-frequency reflections decay faster than low-frequency reflections.

Tunable: Yes

Data Types: `single` | `double`

`WetDryMix` — Wet-dry mix
`0.3` (default) | real scalar

Wet-dry mix, specified as a real positive scalar in the range [0, 1].

Wet-dry mix is the ratio of wet (reverberated) to dry (original) signal that your `reverberator` System object outputs.

Tunable: Yes

Data Types: `single` | `double`

`SampleRate` — Input sample rate (Hz)
`44100` (default) | positive scalar

Input sample rate in Hz, specified as a positive scalar.

Tunable: Yes

Data Types: `single` | `double`

Usage

Syntax

```
audioOut = reverb(audioIn)
```

Description

`audioOut = reverb(audioIn)` adds reverberation to the input signal, `audioIn`, and returns the mixed signal, `audioOut`. The type of reverberation is specified by the algorithm and properties of the reverberator System object, `reverb`.

Input Arguments

`audioIn` — Audio input to reverberator
column vector | *N*-by-2 matrix

Audio input to the reverberator, specified as a column vector or two-column matrix. The columns of the matrix are treated as independent audio channels.

Data Types: `single` | `double`

Output Arguments

`audioOut` — Audio output from reverberator

N-by-2 matrix

Audio output from the reverberator, returned as a two-column matrix.

Data Types: `single` | `double`

Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

Specific to reverberator

createAudioPluginClass	Create audio plugin class that implements functionality of System object
parameterTuner	Tune object parameters while streaming

MIDI

configureMIDI	Configure MIDI connections between audio object and MIDI controller
disconnectMIDI	Disconnect MIDI controls from audio object
getMIDIConnections	Get MIDI connections of audio object

Common to All System Objects

clone	Create duplicate System object
isLocked	Determine if System object is in use
release	Release resources and allow changes to System object property values and input characteristics
reset	Reset internal states of System object
step	Run System object algorithm

Examples

Add Reverberation to Audio Signal

Use the `reverberator` System object™ to add artificial reverberation to an audio signal read from a file.

Create the `dsp.AudioFileReader` and `audioDeviceWriter` System objects. Use the sample rate of the reader as the sample rate of the writer.

```
fileReader = dsp.AudioFileReader('FunkyDrums-44p1-stereo-25secs.mp3','SamplesPerFrame',1024);
deviceWriter = audioDeviceWriter('SampleRate',fileReader.SampleRate);
```

Play 10 seconds of the audio signal through your device.

```
tic
while toc < 10
    audio = fileReader();
    deviceWriter(audio);
end
release(fileReader)
```

Construct a `reverberator` System object with default settings.

```
reverb = reverberator
```

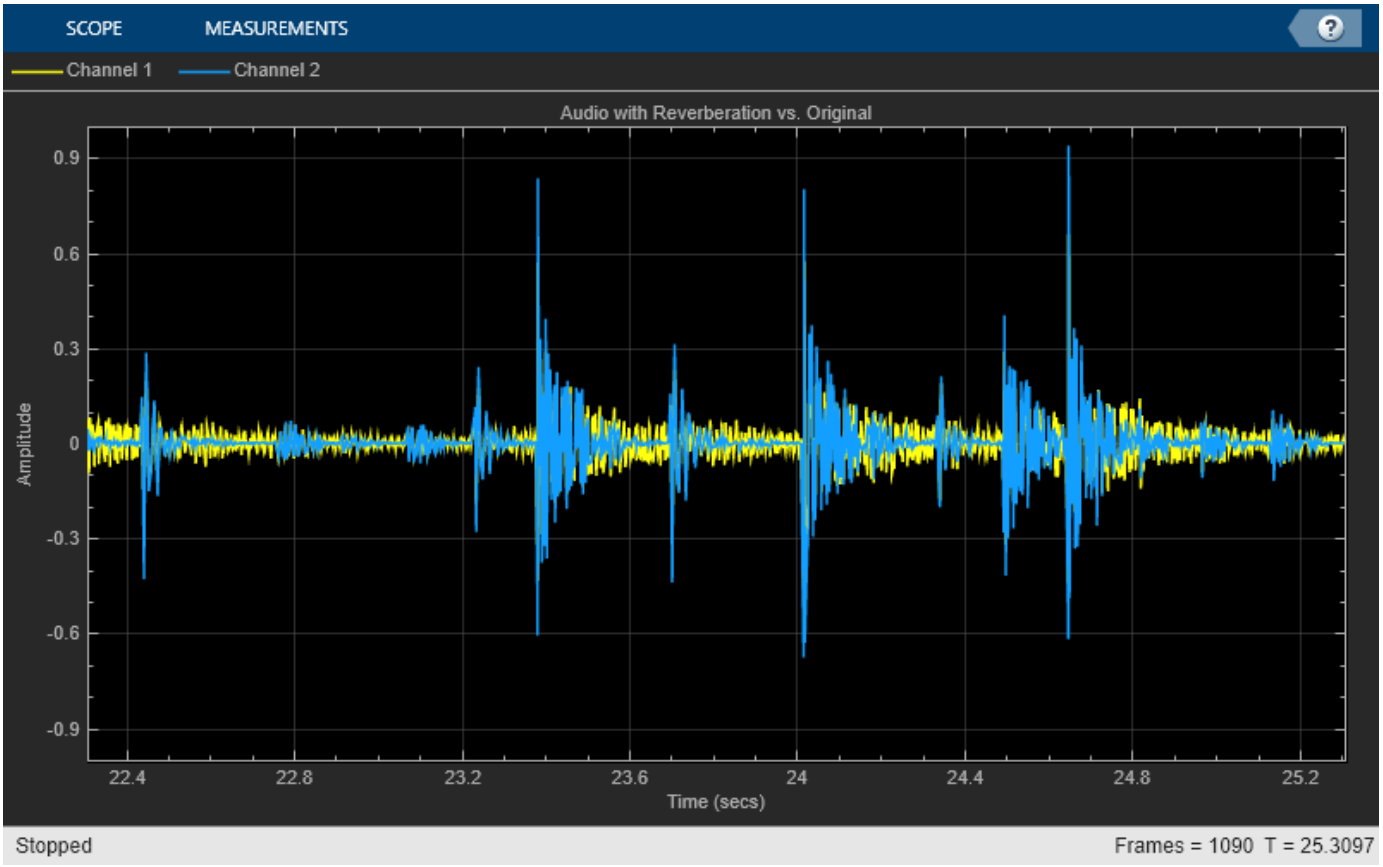
```
reverb =
  reverberator with properties:

                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 = timescope( ...
    'SampleRate',fileReader.SampleRate,...
    'TimeSpanOverrunAction','Scroll',...
    'TimeSpanSource','property',...
    'TimeSpan',3,...
    'BufferLength',3*fileReader.SampleRate*2, ...
    '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)
release(scope)
```



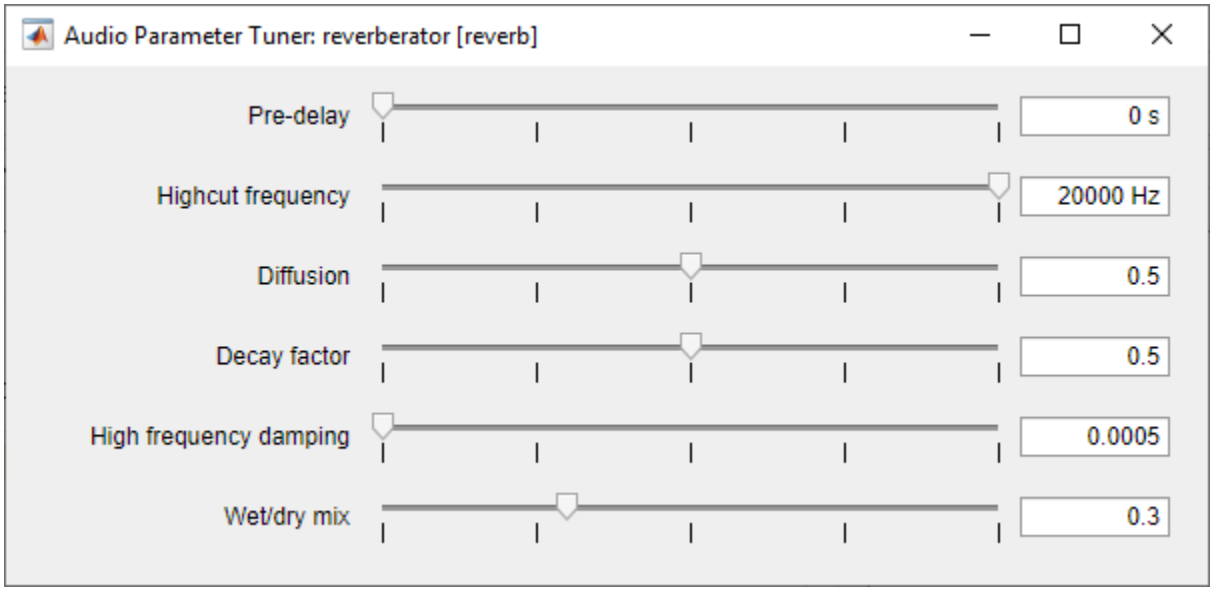
Tune Reverberator Parameters

Create a `dsp.AudioFileReader` to read in audio frame-by-frame. Create an `audioDeviceWriter` to write audio to your sound card. Create a `reverberator` to process the audio data.

```
frameLength = 1024;
fileReader = dsp.AudioFileReader('RockDrums-44p1-stereo-11secs.mp3', ...
    'SamplesPerFrame',frameLength,'PlayCount',2);
deviceWriter = audioDeviceWriter('SampleRate',fileReader.SampleRate);
reverb = reverberator('SampleRate',fileReader.SampleRate);
```

Call `parameterTuner` to open a UI to tune parameters of the `octaveFilter` while streaming.

```
parameterTuner(reverb)
```



In an audio stream loop:

1. Read in a frame of audio from the file.
2. Apply reverberation.
3. Write the frame of audio to your audio device for listening.

While streaming, tune parameters of the reverberator and listen to the effect.

```
while ~isDone(fileReader)
    audioIn = fileReader();
    audioOut = reverb(audioIn);
    deviceWriter(audioOut);
    drawnow limitrate % required to update parameter
end
```

As a best practice, release your objects once done.

```
release(deviceWriter)
release(fileReader)
release(reverb)
```

Tips

The `createAudioPluginClass` and `configureMIDI` functions map tunable properties of the `compressor` to user-facing parameters:

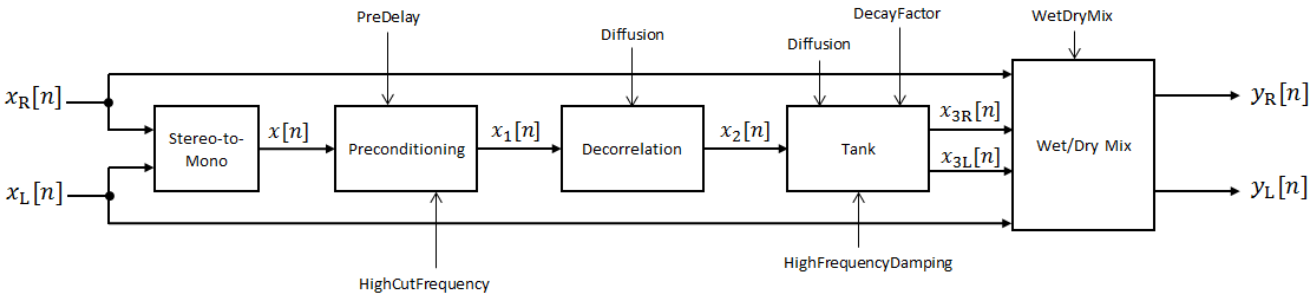
Property	Range	Mapping	Unit
`PreDelay`	[0, 1]	linear	s
`HighCutFrequency`	[20, 20000]	log	Hz
`Diffusion`	[0, 1]	linear	none
`DecayFactor`	[0, 1]	linear	none

Property	Range	Mapping	Unit
`HighFrequencyDamping`	[0, 1]	linear	none
`WetDryMix`	[0, 1]	linear	none

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.



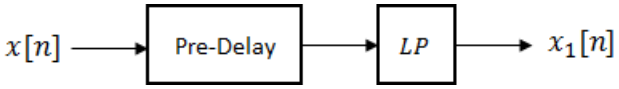
The description for the algorithm that follows is for a stereo input. A mono input is a simplified case.

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

  $$LP(z) = \frac{1 - \alpha}{1 - \alpha z^{-1}},$$

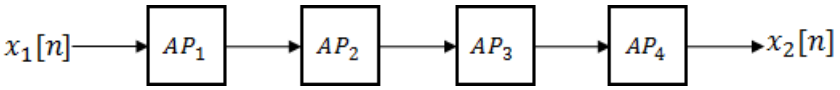
  where

  $$\alpha = \exp\left(-2\pi \times \frac{f_c}{f_s}\right).$$

  - 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

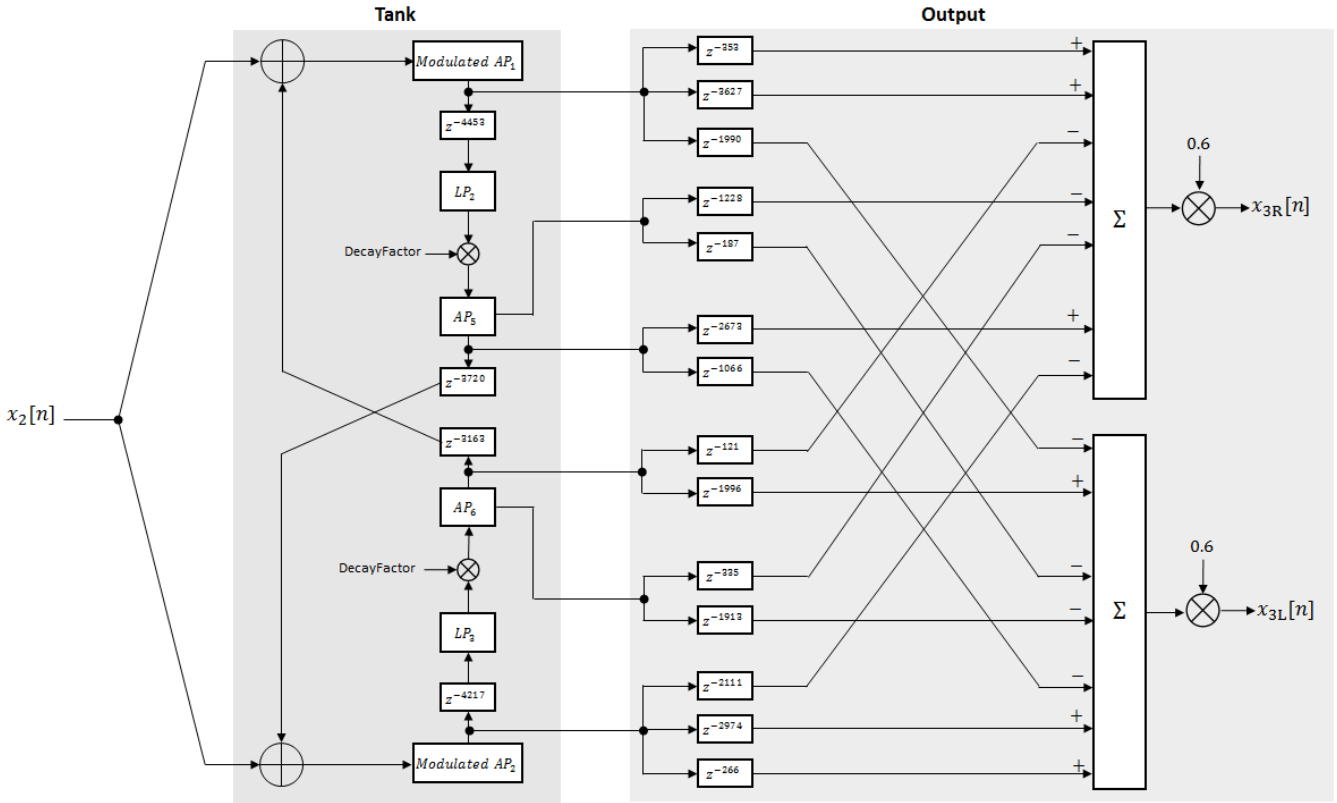
$$AP(z) = \frac{\beta + z^{-k}}{1 + \beta z^{-k}},$$

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.

Version History

Introduced in R2016a

References

[1] Dattorro, Jon. "Effect Design, Part 1: Reverberator and Other Filters." *Journal of the Audio Engineering Society*. Vol. 45, Issue 9, 1997, pp. 660–684.

[2] Dattorro, Jon. "Effect Design, Part 2: Delay-Line Modulation and Chorus." *Journal of the Audio Engineering Society*. Vol. 45, Issue 10, 1997, pp. 764–788.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

“System Objects in MATLAB Code Generation” (MATLAB Coder)

See Also

Reverberator

shelvingFilter

Second-order IIR shelving filter

Description

The `shelvingFilter` System object implements a shelving filter, which boosts or cuts the frequency spectrum of the input signal above or below a given cutoff frequency.

To use a shelving filter:

1. Create the `shelvingFilter` object and set its properties.
2. Call the object with arguments, as if it were a function.

To learn more about how System objects work, see What Are System Objects?

Creation

Syntax

```
shelvFilt = shelvingFilter
shelvFilt = shelvingFilter(gain)
shelvFilt = shelvingFilter(gain,slope)
shelvFilt = shelvingFilter(gain,slope,cutoffFreq)
shelvFilt = shelvingFilter(gain,slope,cutoffFreq,type)
shelvFilt = shelvingFilter(___,Name=Value)
```

Description

`shelvFilt = shelvingFilter` creates a shelving filter with default values.

`shelvFilt = shelvingFilter(gain)` sets the `Gain` property to `gain`.

`shelvFilt = shelvingFilter(gain,slope)` sets the `Slope` property to `slope`.

`shelvFilt = shelvingFilter(gain,slope,cutoffFreq)` sets the `CutoffFrequency` property to `cutoffFreq`.

`shelvFilt = shelvingFilter(gain,slope,cutoffFreq,type)` sets the `FilterType` property to `type`.

`shelvFilt = shelvingFilter(___,Name=Value)` sets "Properties" on page 3-364 using one or more name-value arguments in addition to the input arguments in previous syntaxes. For example, `shelvFilt = shelvingFilter(SampleRate=96000)` creates a shelving filter with a sample rate of 96,000 Hz.

Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see System Design in MATLAB Using System Objects.

`Gain` — Peak gain (dB)

`0` (default) | real scalar

Peak gain of the filter in dB, specified as a real scalar. The gain specifies how much the filter will boost (if the gain is positive) or cut (if the gain is negative) the frequency spectrum of the input signal.

Tunable: Yes

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`Slope` — Filter slope

`1.5` (default) | positive scalar

Slope of the filter, specified as a positive scalar. The slope controls the width of the transition band in the filter response.

Tunable: Yes

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`CutoffFrequency` — Cutoff frequency of filter

`200` (default) | nonnegative scalar

Cutoff frequency of the filter in Hz, specified as a nonnegative scalar in the range [0,`SampleRate`/2]. The cutoff frequency specifies the frequency at half of the peak gain of the filter, `Gain`/2 dB.

Tunable: Yes

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`FilterType` — Type of filter

`"lowpass"` (default) | `"highpass"`

Type of shelving filter, specified as `"lowpass"` or `"highpass"`.

- `"lowpass"` –– Boost or cut the frequency spectrum below the cutoff frequency.
- `"highpass"` –– Boost or cut the frequency spectrum above the cutoff frequency.

Tunable: Yes

Data Types: `string` | `char`

`SampleRate` — Sample rate of the input (Hz)

`44100` (default) | positive scalar

Sample rate of the input in Hz, specified as a positive scalar.

Tunable: No

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Usage

Syntax

```
audioOut = shelvFilt(audioIn)
```

Description

`audioOut = shelvFilt(audioIn)` applies the shelving filter to the input signal, `audioIn` and returns the output signal `audioOut`. The type of filtering applied by the function depends on the algorithm and properties of the `shelvingFilter` System object.

Input Arguments

`audioIn` — Audio input to shelving filter

column vector | matrix

Audio input to the shelving filter, specified as a column vector or a matrix. If the input is a matrix, the columns are treated as independent channels.

Data Types: `single` | `double`

Output Arguments

`audioOut` — Audio output of shelving filter

column vector | matrix

Audio output of the shelving filter, returned as a column vector or matrix with the same size and data type as the `audioIn`.

Data Types: `single` | `double`

Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

Specific to `shelvingFilter`

visualize	Visualize magnitude response of shelving filter
createAudioPluginClass	Create audio plugin class that implements functionality of System object
coeffs	Get filter coefficients
parameterTuner	Tune object parameters while streaming

MIDI

configureMIDI	Configure MIDI connections between audio object and MIDI controller
disconnectMIDI	Disconnect MIDI controls from audio object

getMIDIConnections Get MIDI connections of audio object

Common to All System Objects

step Run System object algorithm
release Release resources and allow changes to System object property values and input characteristics
reset Reset internal states of System object

Examples

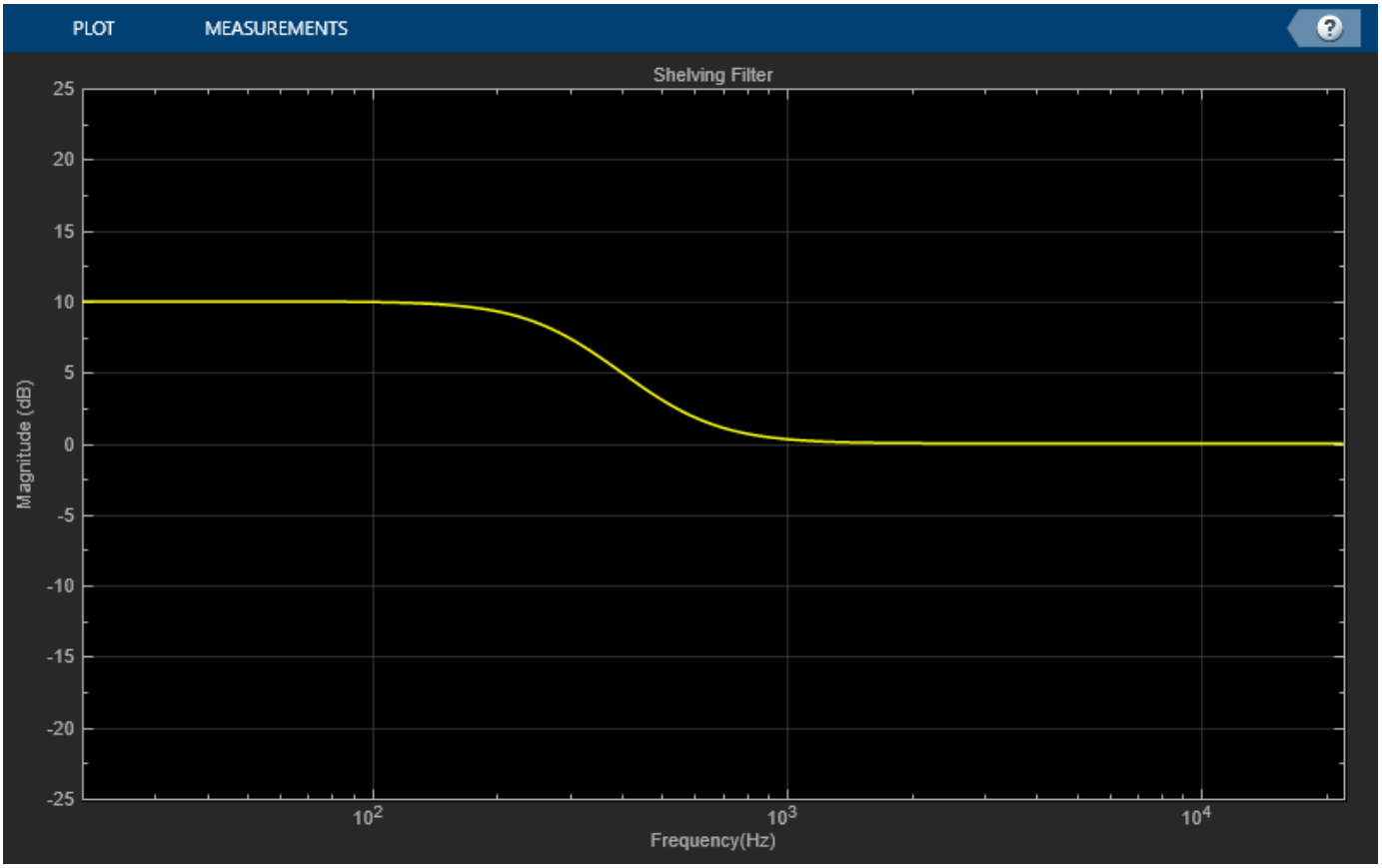
Filter Audio Using Shelving Filter

Create a lowpass `shelvingFilter` object with a gain of 10 dB, a slope of 1, and a cutoff frequency of 400 Hz.

```
shelvFilt = shelvingFilter(10,1,400,"lowpass");
```

Visualize the magnitude response of the filter.

```
visualize(shelvFilt)
```



Create a `dsp.AudioFileReader` object to read in the audio signal for filtering.

```
samplesPerFrame = 1024;
reader = dsp.AudioFileReader( ...
```

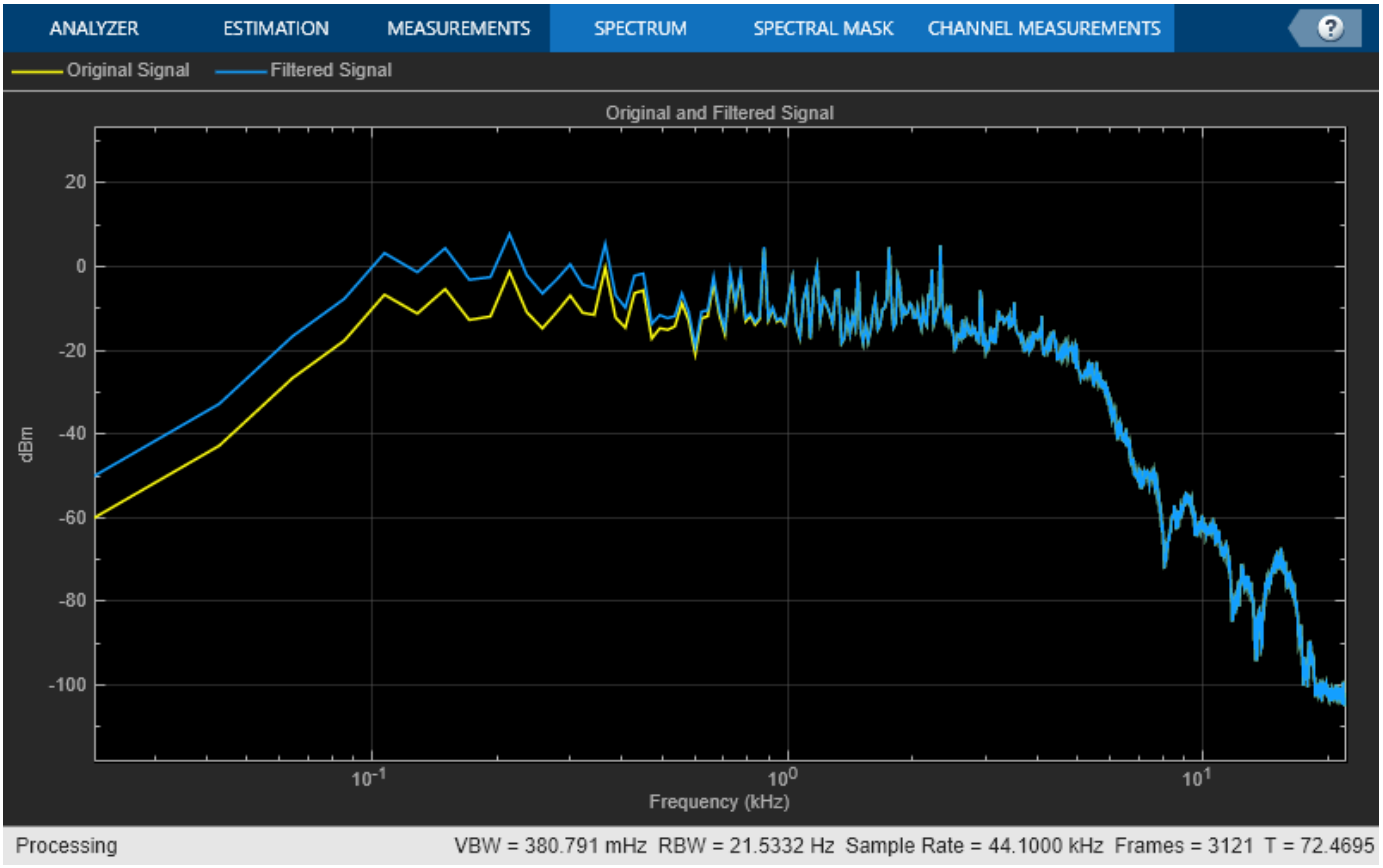
```
    Filename="RockGuitar-16-44p1-stereo-72secs.wav", ...
    SamplesPerFrame=samplesPerFrame);
```

Create a `spectrumAnalyzer` object to visualize the spectrum of the original audio signal and the spectrum of the filtered signal.

```
scope = spectrumAnalyzer( ...
    SampleRate=reader.SampleRate, ...
    PlotAsTwoSidedSpectrum=false, ...
    FrequencyScale="log", ...
    Title="Original and Filtered Signal", ...
    ShowLegend=true, ...
    ChannelNames=["Original Signal","Filtered Signal"]);
```

Filter the audio signal and visualize the results.

```
while ~isDone(reader)
    audioIn = reader();
    filteredSignal = shelvFilt(audioIn);
    scope([audioIn(:,1),filteredSignal(:,1)]);
end
```



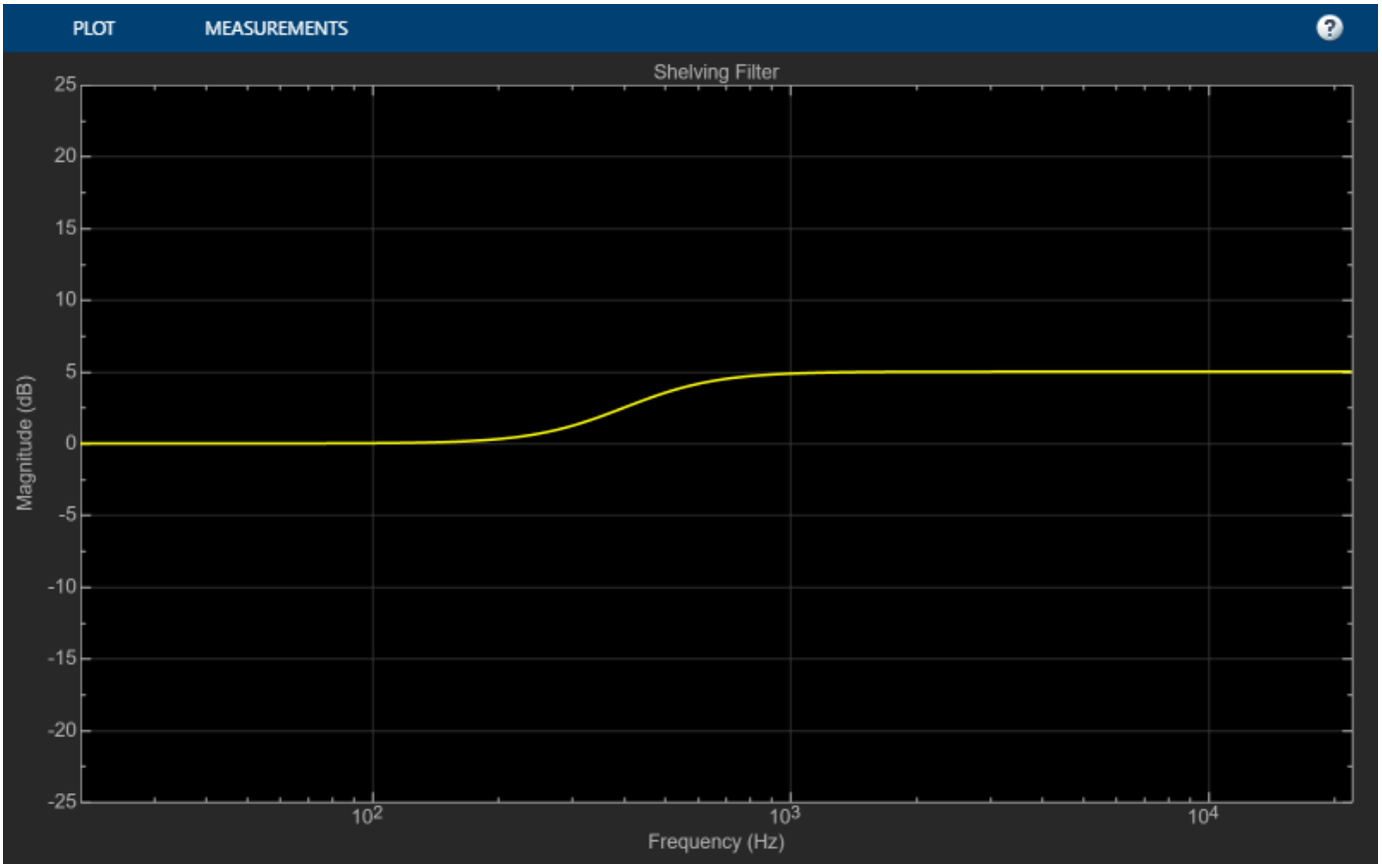
Tune Shelving Filter Parameters

Create a highpass `shelvingFilter` object with a gain of 5 dB, a slope of 1, and a cutoff frequency of 400 Hz.

```
shelvFilt = shelvingFilter(5,1,400,"highpass");
```

Visualize the magnitude response of the filter.

```
visualize(shelvFilt)
```

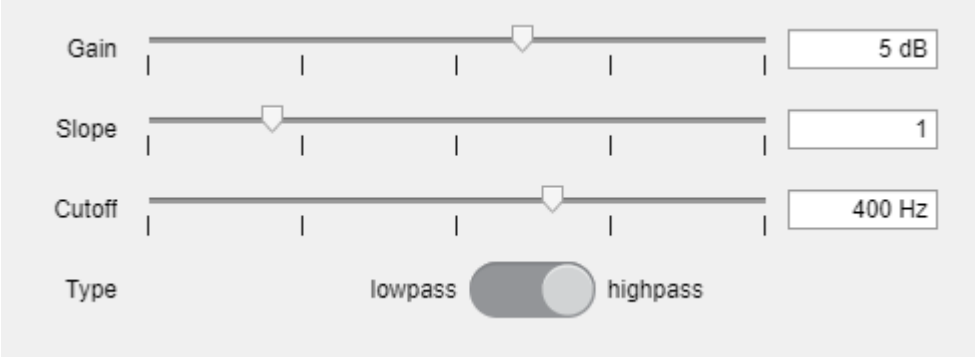


Create `dsp.AudioFileReader` and `audioDeviceWriter` objects to read in the audio signal and write the filtered signal to your audio device to listen to it.

```
samplesPerFrame = 1024;
reader = dsp.AudioFileReader( ...
    Filename="RockGuitar-16-44p1-stereo-72secs.wav", ...
    SamplesPerFrame=samplesPerFrame);
deviceWriter = audioDeviceWriter(SampleRate=reader.SampleRate);
```

Call `parameterTuner` to open a UI to tune the parameters of the shelving filter while streaming.

```
parameterTuner(shelvFilt)
```



In an audio stream loop:

1. Read in a frame of audio from the file.
2. Apply shelving filter.
3. Write the filtered frame of audio to your audio device for listening.

While streaming, tune the parameters of the shelving filter and listen to the effect.

```
while ~isDone(reader)
    audioIn = reader();
    audioOut = shelvFilt(audioIn);
    deviceWriter(audioOut);
    drawnow limitrate % required to update parameter
end
```

As a best practice, release your objects once done.

```
release(reader)
release(shelvFilt)
release(deviceWriter)
```

Version History

Introduced in R2022a

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

Blocks

Shelving Filter | Graphic EQ | Multiband Parametric EQ

Functions

`designParamEQ` | `designShelvingEQ` | `designVarSlopeFilter`

Objects
`dsp.SOSFilter` | `graphicEQ` | `multibandParametricEQ`

visualize

Visualize magnitude response of shelving filter

Syntax

```
visualize(shelvFilt)
visualize(shelvFilt,NFFT)
hvsz = visualize( ___ )
```

Description

`visualize(shelvFilt)` plots the magnitude response of the shelving filter. The plot is updated automatically when you change the object properties.

`visualize(shelvFilt,NFFT)` specifies an *N*-point FFT to calculate the magnitude response.

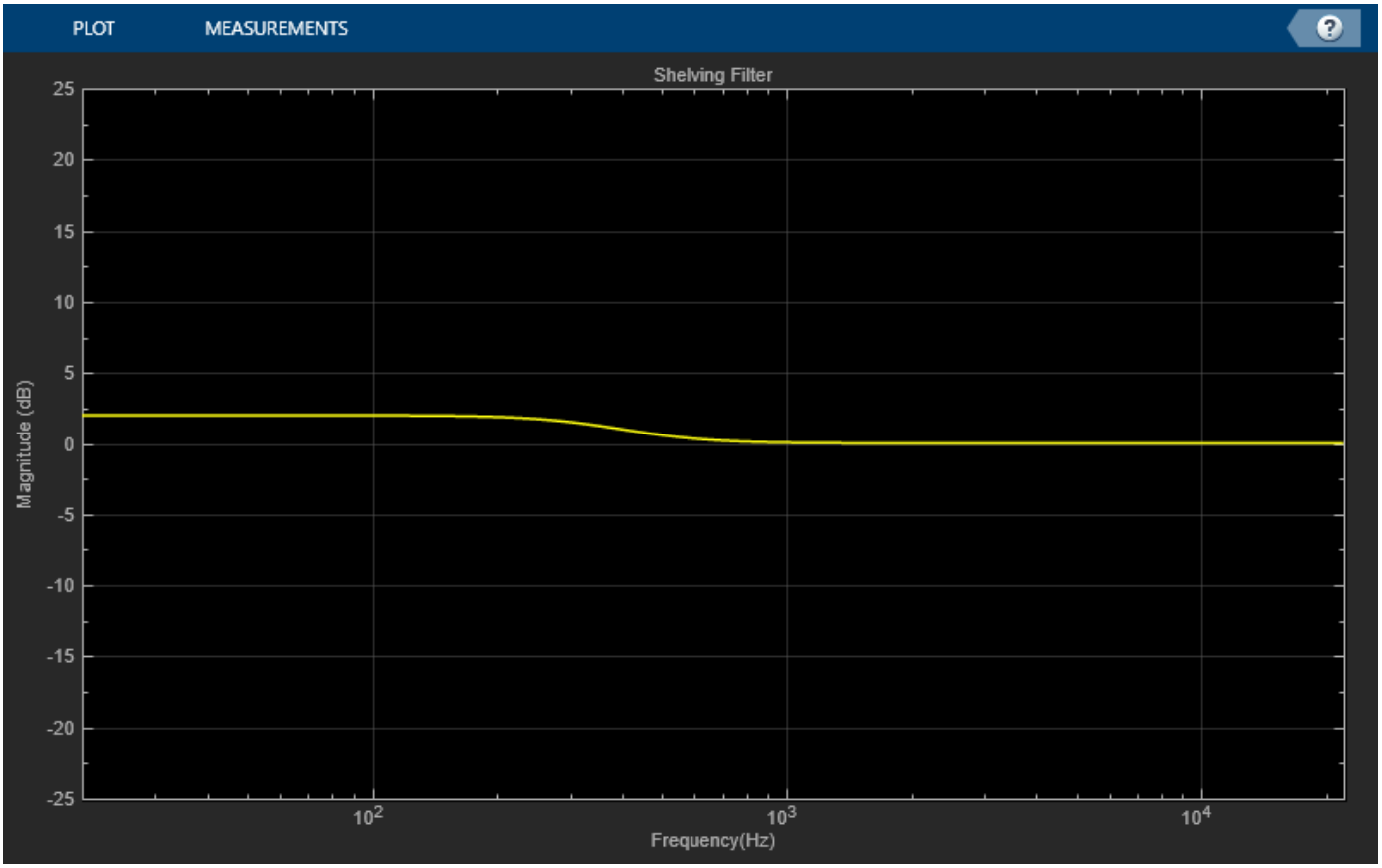
`hvsz = visualize( ___ )` returns a handle to the visualizer as a `dsp.DynamicFilterVisualizer` object when you call this syntax with any of the previous input arguments.

Examples

Visualize Magnitude Response of Shelving Filter

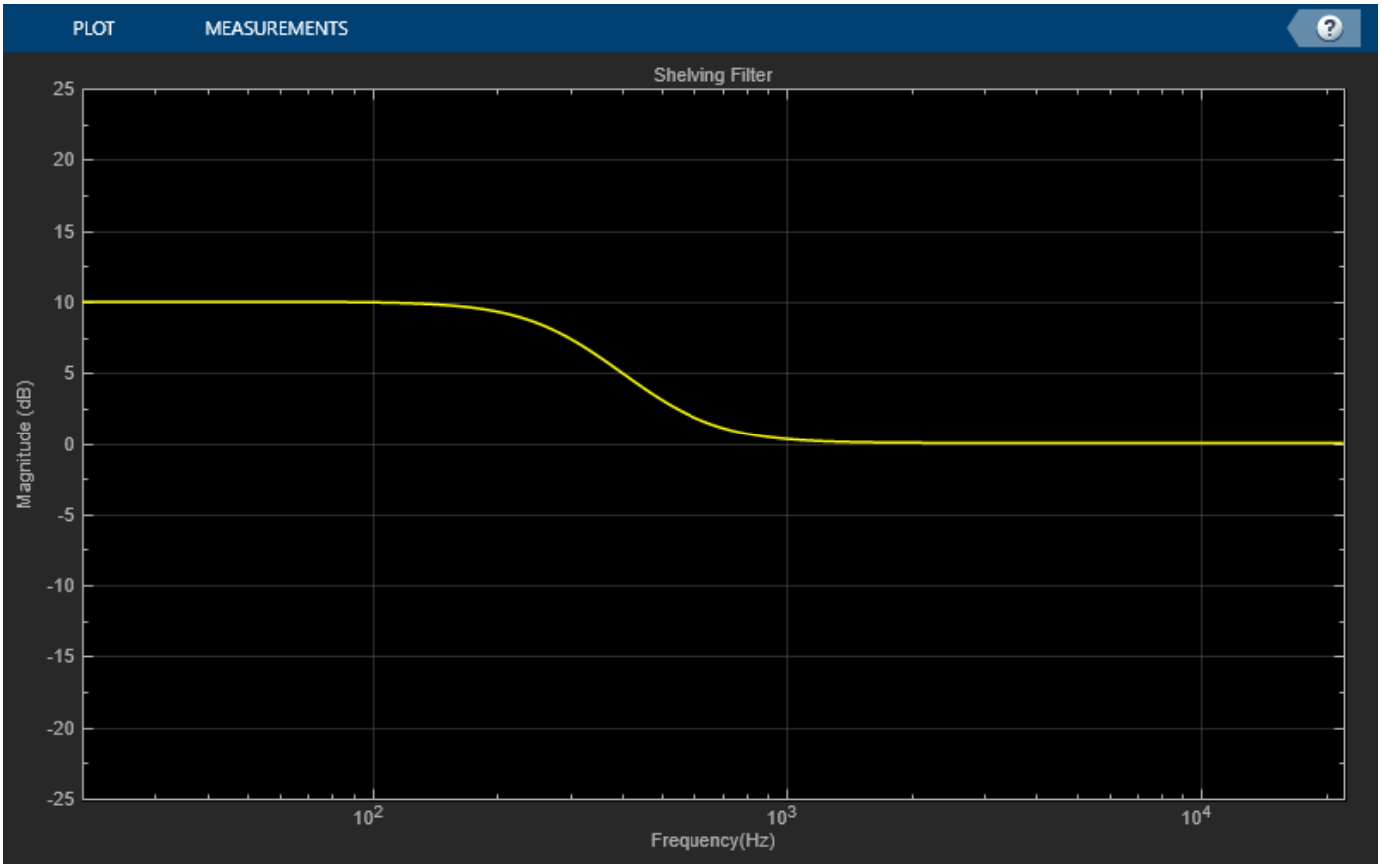
Create a `shelvingFilter` object, and then call `visualize` to plot the magnitude response of the filter.

```
shelvFilt = shelvingFilter(2,1,400,"lowpass");
visualize(shelvFilt)
```



Modify the gain and observe that the plot is updated automatically.

```
shelvFilt.Gain = 10;
```



Input Arguments

`shelvFilt` — Shelving filter

object

Shelving filter whose magnitude response you want to plot, specified as a `shelvingFilter` System object.

`NFFT` — N-point FFT

`2048` (default) | positive scalar

Number of bins used to calculate the DFT, specified as a positive scalar.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Version History

Introduced in R2022a

See Also

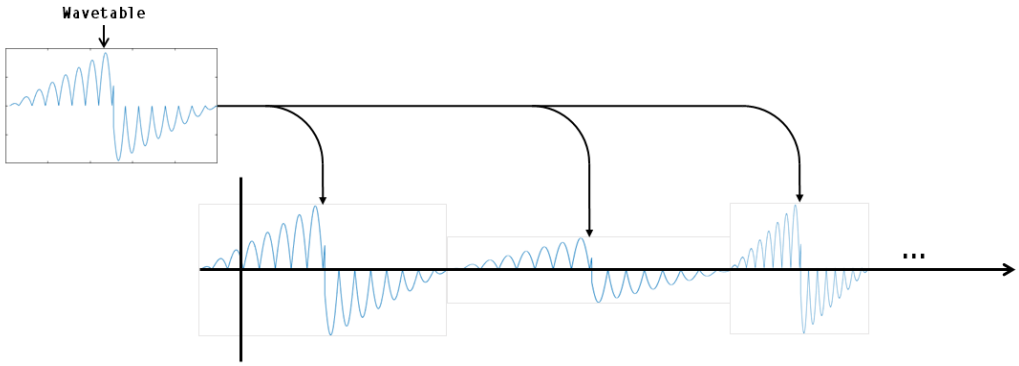
`shelvingFilter`

wavetableSynthesizer

Generate periodic signal from single-cycle waveforms

Description

The `wavetableSynthesizer` System object generates a periodic signal with tunable properties. The periodic signal is defined by a single-cycle waveform cached as the `Wavetable` property of your `wavetableSynthesizer` object.



To generate a periodic signal:

1. Create the `wavetableSynthesizer` object and set its properties.
2. Call the object with arguments, as if it were a function.

To learn more about how System objects work, see What Are System Objects?

Creation

Syntax

```
waveSynth = wavetableSynthesizer
waveSynth = wavetableSynthesizer(wavetableValue)
waveSynth = wavetableSynthesizer(wavetableValue,frequencyValue)
waveSynth = wavetableSynthesizer( ___ ,Name,Value)
```

Description

`waveSynth = wavetableSynthesizer` creates a wavetable synthesizer System object, `waveSynth`, with default property values.

`waveSynth = wavetableSynthesizer(wavetableValue)` sets the Wavetable property to `wavetableValue`.

`waveSynth = wavetableSynthesizer(wavetableValue,frequencyValue)` sets the Frequency property to `frequencyValue`.

`waveSynth = wavetableSynthesizer( ___ ,Name,Value)` sets each property `Name` to the specified `Value`. Unspecified properties have default values.

Example: `waveSynth = wavetableSynthesizer('Amplitude',2,'DCOffset',2.5)` creates a System object, `waveSynth`, that generates the default sine waveform with an amplitude of 2 and a DC offset of 2.5.

Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see System Design in MATLAB Using System Objects.

`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: Yes

Data Types: `single` | `double`

`Frequency` — Frequency of generated signal (Hz)

`100` (default) | real scalar

Frequency of generated signal in Hz, specified as a real scalar greater than or equal to 0.

Tunable: Yes

Data Types: `single` | `double`

`Amplitude` — Amplitude of generated signal

`1` (default) | real scalar

Amplitude of generated signal, specified as a real scalar greater than or equal to 0.

The generated signal is multiplied by the value specified by `Amplitude` at the output, before DCOffset is applied.

Tunable: Yes

Data Types: `single` | `double`

`PhaseOffset` — Normalized phase offset of generated signal

`0` (default) | real scalar

Normalized phase offset of generated signal, specified as a real scalar with values in the range [0, 1]. The range is a normalized 2π radians interval.

Tunable: No

Data Types: `single` | `double`

`DCOffset` — Value added to each element of generated signal
`0` (default) | real scalar

Value added to each element of the generated signal, specified as a real scalar.

Tunable: Yes

Data Types: `single` | `double`

`SamplesPerFrame` — Number of samples per frame
`512` (default) | positive integer

Number of samples per frame, specified as a positive integer in the range [1, `MaxSamplesPerFrame`].

This property determines the vector length that your `wavetableSynthesizer` object outputs.

Tunable: Yes

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`MaxSamplesPerFrame` — Maximum number of samples per frame
`192000` (default) | positive integer

Maximum number of samples per frame, specified as a positive integer. Setting this property to a lower value can save memory when using code generation.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`SampleRate` — Sample rate of generated signal (Hz)
`44100` (default) | real positive scalar

Sample rate of generated signal in Hz, specified as a real positive scalar.

Tunable: Yes

`OutputDataType` — Data type of generated signal
`'double'` (default) | `'single'`

Data type of generated signal, specified as `'double'` or `'single'`.

Tunable: No

Data Types: `char` | `string`

Usage

Syntax

```
waveform = waveSynth()
```

Description

`waveform = waveSynth()` generates a periodic signal, `waveform`. The type of signal is specified by the algorithm and properties of the `wavetableSynthesizer` System object, `waveSynth`.

Output Arguments

`waveform` — Waveform output from wavetable synthesizer
column vector (default)

Waveform output from the wavetable synthesizer, returned as a column vector with length specified by the SamplesPerFrame property and data type specified by the OutputDataType property.

Data Types: `single` | `double`

Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

Specific to wavetableSynthesizer

createAudioPluginClass	Create audio plugin class that implements functionality of System object
parameterTuner	Tune object parameters while streaming

MIDI

configureMIDI	Configure MIDI connections between audio object and MIDI controller
disconnectMIDI	Disconnect MIDI controls from audio object
getMIDIConnections	Get MIDI connections of audio object

Common to All System Objects

clone	Create duplicate System object
isLocked	Determine if System object is in use
release	Release resources and allow changes to System object property values and input characteristics
reset	Reset internal states of System object
step	Run System object algorithm

The `createAudioPluginClass` and `configureMIDI` functions map tunable properties of the `wavetableSynthesizer` System object to user-facing parameters:

Property	Range	Mapping	Unit
`Frequency`	[0.1, 20000]	log	Hz
`Amplitude`	[0, 10]	linear	none
`DCOffset`	[–10, 10]	linear	none

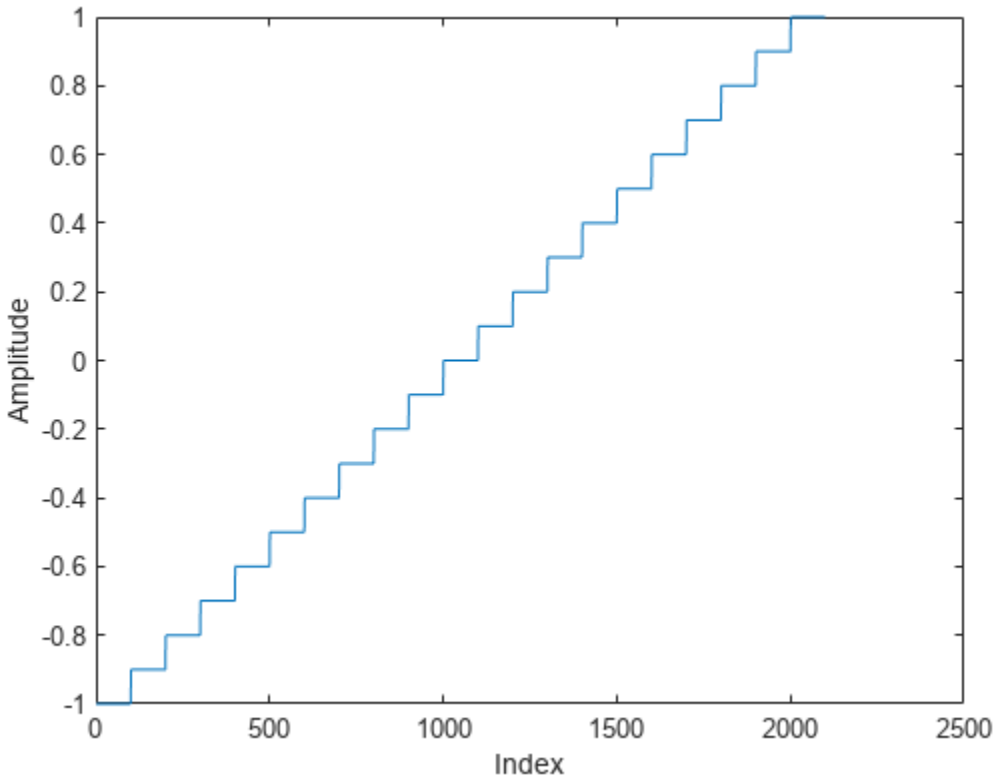
Examples

Generate Variable-Frequency Staircase Wave

Define and plot a single-cycle waveform.

```
values = -1:0.1:1;
singleCycleWave = ones(100,1) * values;
```

```
singleCycleWave = reshape(singleCycleWave,numel(singleCycleWave),1);

plot(singleCycleWave)
xlabel('Index')
ylabel('Amplitude')
```



Create a wavetable synthesizer, `waveSynth`, to generate a staircase wave using the single-cycle waveform. Specify a frequency of 10 Hz.

```
waveSynth = wavetableSynthesizer(singleCycleWave,10);
```

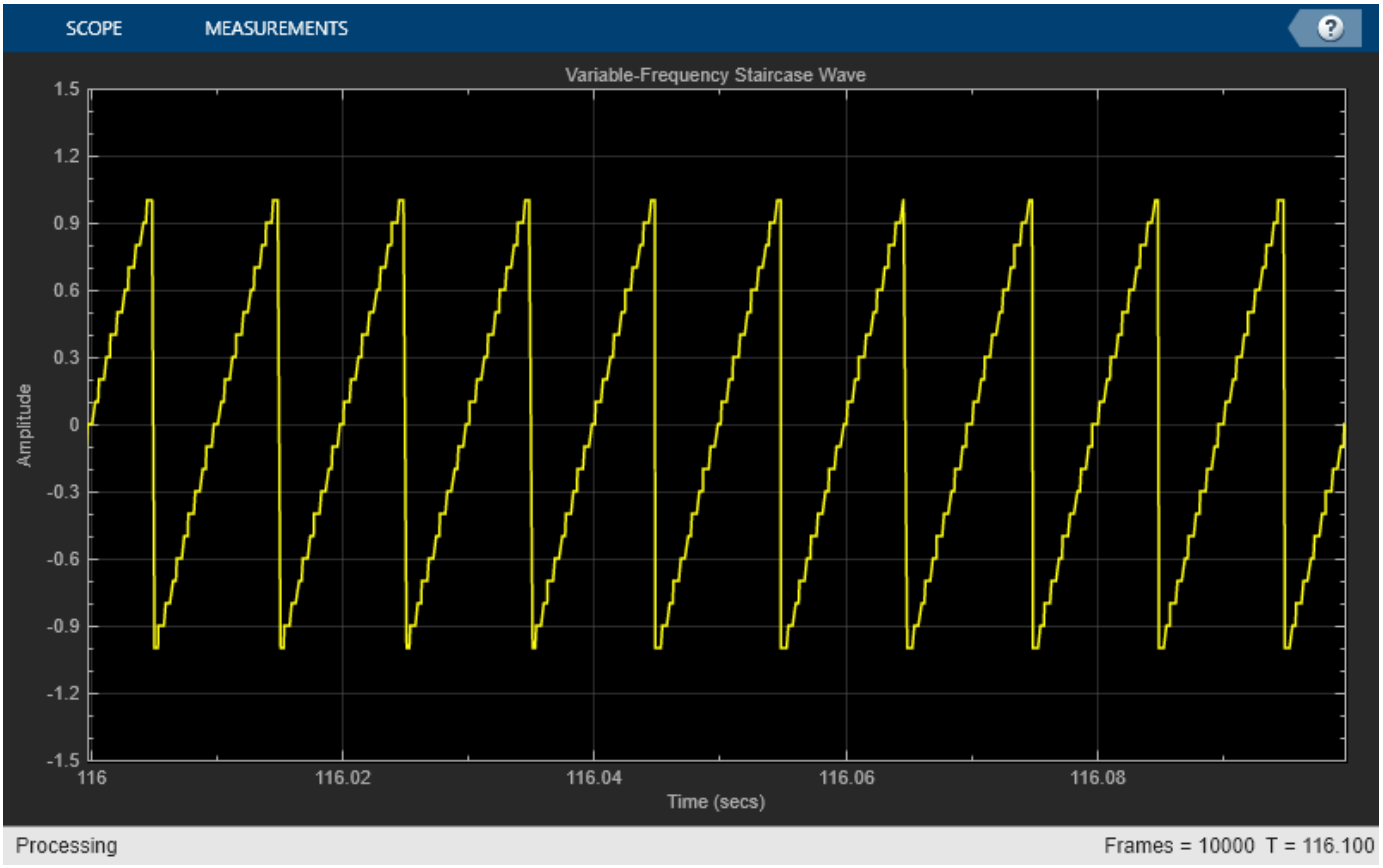
Create a time scope to visualize the staircase wave generated by `waveSynth`.

```
scope = timescope( ...
    'SampleRate',waveSynth.SampleRate, ...
    'TimeSpanSource','Property','TimeSpan',0.1, ...
    'YLimits',[-1.5,1.5], ...
    'TimeSpanOverrunAction','Scroll', ...
    'ShowGrid',true, ...
    'Title','Variable-Frequency Staircase Wave');
```

Place the wavetable synthesizer in an audio stream loop. Increase the frequency of your staircase wave in 10 Hz increments.

```
counter = 0;
while (counter < 1e4)
    counter = counter + 1;
    staircaseWave = waveSynth();
```

```
    scope(staircaseWave)
    if mod(counter,1000)==0
        waveSynth.Frequency = waveSynth.Frequency + 10;
    end
end
```



Manipulate Audio Samples Using Wavetable Synthesizer

Sample an audio file and save it to the `Wavetable` property of a `wavetableSynthesizer` System object™. Use the wavetable synthesizer to manipulate your audio sample.

Read in an entire audio file. Clip out an interesting sound from the audio and then play it.

```
[audio,fs] = audioread('MainStreetOne-16-16-mono-12secs.wav');

aSound = audio(2.5e4:5e4);
sound(aSound,fs)
```

Create a wavetable synthesizer using your audio clip. The duration of the `engine` audio clip is `numel(aSound)/fs` seconds. In the `wavetableSynthesizer`, set the `Frequency` property to 1/(clip duration). The generated signal now plays back at the same rate it was recorded at.

```
duration = numel(aSound)/fs;
waveSynth = wavetableSynthesizer('Wavetable',aSound,'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` object while stream processing. Visualize the wavetable and play the resulting audio.

Create a single-cycle waveform for the `wavetableSynthesizer` to index into. Create a wavetable synthesizer object.

```
t = 0:0.001:1;
exponent = 5;
waveTable = [t.^exponent,fliplr(t.^exponent)] - 0.5;

waveSynth = wavetableSynthesizer('Wavetable',waveTable);
```

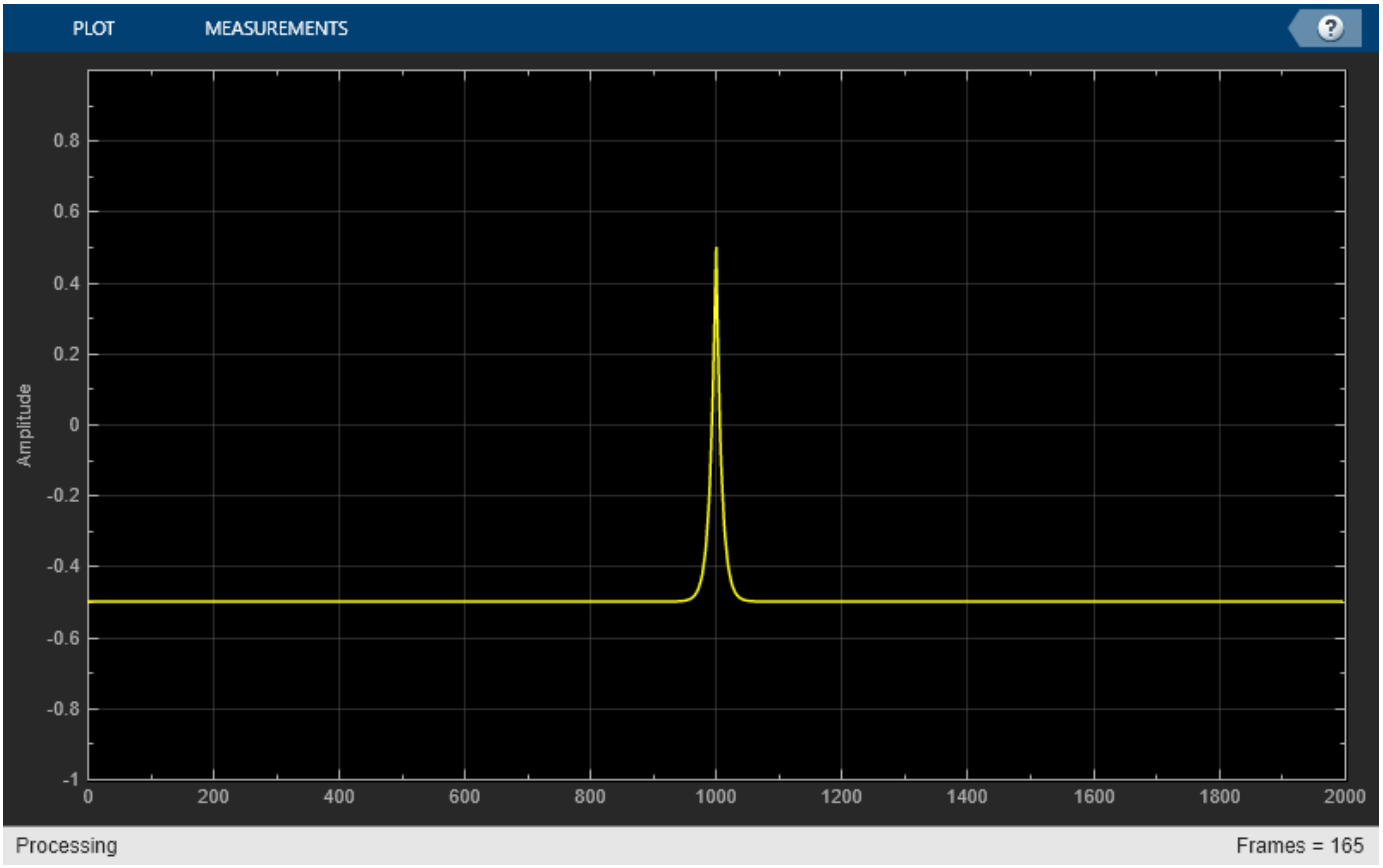
Create a `dsp.ArrayPlot` object to plot the wavetable as it is modified over time. Create an `audioDeviceWriter` object to listen to the signal output by your wavetable synthesizer.

```
arrayPlotter = dsp.ArrayPlot('YLimits',[-1,1],'PlotType','Line');
deviceWriter = audioDeviceWriter;
```

In an audio stream loop, incrementally modify the `Wavetable` property of the wavetable synthesizer and plot it. Call the synthesizer to output a waveform and play the waveform to your audio device.

```
tic
while toc < 10
    exponent = exponent - 0.01;
    waveSynth.Wavetable = [t.^abs(exponent),fliplr(t.^abs(exponent))] - 0.5;

    arrayPlotter(waveSynth.Wavetable')
```

```
    deviceWriter(waveSynth());
end
```




```
release(deviceWriter)
```

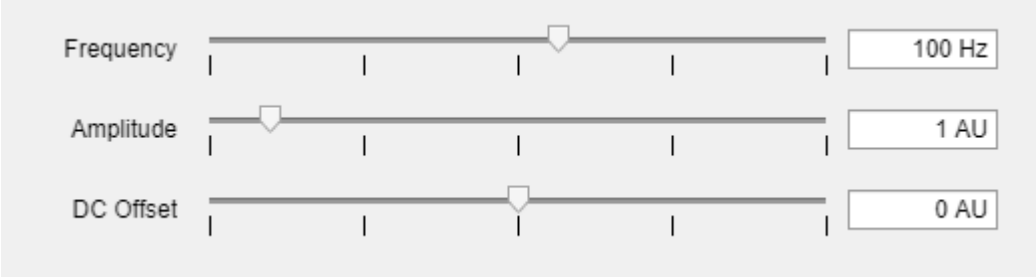
Tune Wavetable Synthesizer Parameters

Create a `wavetableSynthesizer` to generate a waveform. Create a `timescope` to visualize the waveform. Create an `audioDeviceWriter` to write audio to your sound card.

```
fs = 44.1e3;
wvSynth = wavetableSynthesizer('SampleRate',fs);

scope = timescope( ...
    'SampleRate',wvSynth.SampleRate, ...
    'TimeSpanSource','Property','TimeSpan',1, ...
    'YLimits',[-2,2], ...
    'TimeSpanOverrunAction','Scroll', ...
    'ShowGrid',true);

deviceWriter = audioDeviceWriter('SampleRate',wvSynth.SampleRate);
```

Call `parameterTuner` to open a UI to tune parameters of the wavetable synthesizer while streaming.

```
parameterTuner(wvSynth)
```



In an audio stream loop:

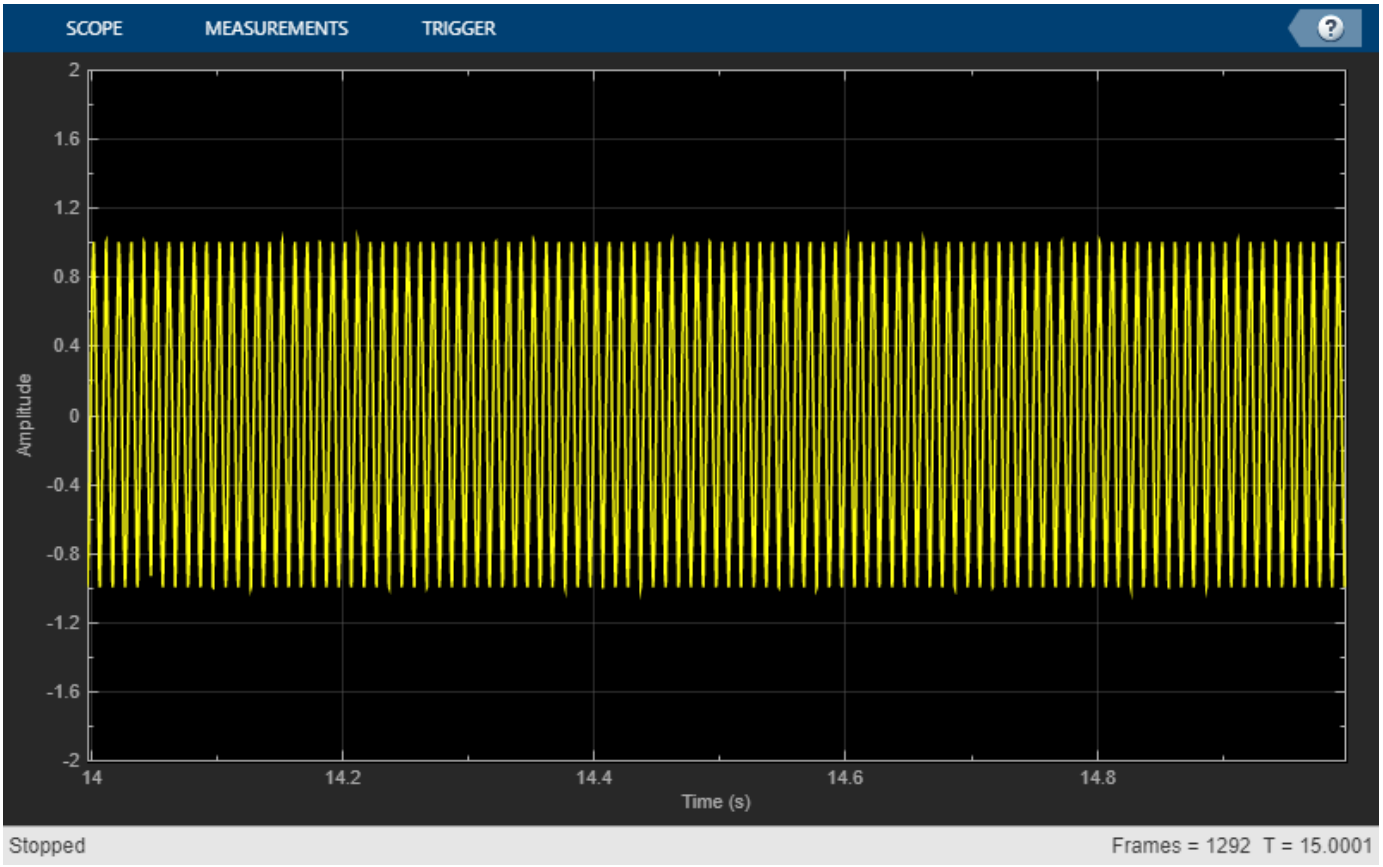
1. Call the wavetable synthesizer without arguments to output one frame of data.
2. Visualize the data using the time scope.
3. Write the frame of audio to your audio device for listening.

While streaming, tune parameters of the wavetable synthesizer and listen to the effect.

```
duration = 15;
numIterations = round(wvSynth.SampleRate*duration/wvSynth.SamplesPerFrame);
for i = 1:numIterations
    audioOut = wvSynth();
    scope(audioOut)
    deviceWriter(audioOut);
    drawnow limitrate % required to update parameter
end
```

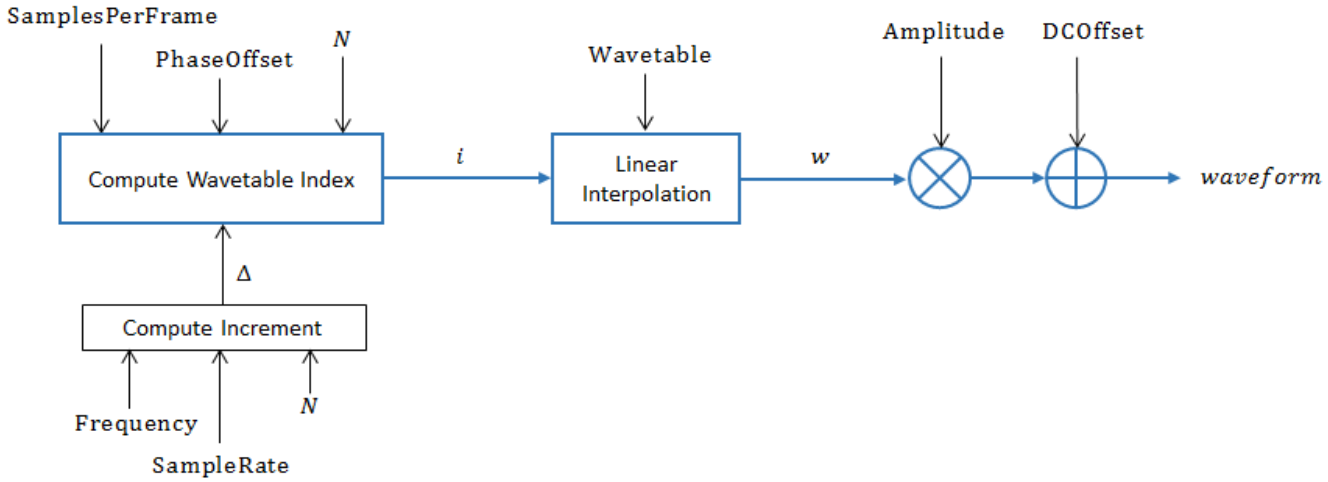
As a best practice, release your objects when done.

```
release(deviceWriter)
release(wvSynth)
release(scope)
```



Algorithms

The `wavetableSynthesizer` System object synthesizes periodic signals using a cached single-cycle waveform, specified waveform properties, and phase memory.



Compute Increment

Compute the increment step size:

$$\Delta = \frac{\text{Frequency}}{\text{SampleRate}} \times N,$$

where N is the number of elements in your wavetable.

Compute Wavetable Index

Compute `Wavetable` index,

$$i[n] = \begin{cases} i[n-1] + \Delta & i[n-1] < N \\ i[n-1] + \Delta - N & i[n-1] \geq N \end{cases},$$

for $2 \leq n \leq$ SamplesPerFrame. The `PhaseOffset` property determines $i[n{=}1]$.

Linear Interpolation

Index into the `Wavetable` and perform linear interpolation:

$$w = \begin{cases} (\text{Wavetable}[1] - \text{Wavetable}[i_L]) \times \varepsilon + \text{Wavetable}[i_L] & i_H > N \\ (\text{Wavetable}[i_H] - \text{Wavetable}[i_L]) \times \varepsilon + \text{Wavetable}[i_L] & i_H \leq N \end{cases}.$$

- $i_L = \text{floor}(i[n] + 1)$
- $i_H = i_L + 1$
- $\varepsilon = i - \text{floor}(i)$

Apply Amplitude and DC Offset

Multiply by `Amplitude` and add `DCOffset`.

$$waveform = w \times \text{Amplitude} + \text{DCOffset}$$

Version History

Introduced in R2016a

R2022b: New `MaxSamplesPerFrame` property

Use the `MaxSamplesPerFrame` property to specify the maximum number of samples per frame. Setting the property to a lower value can save memory when using code generation.

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

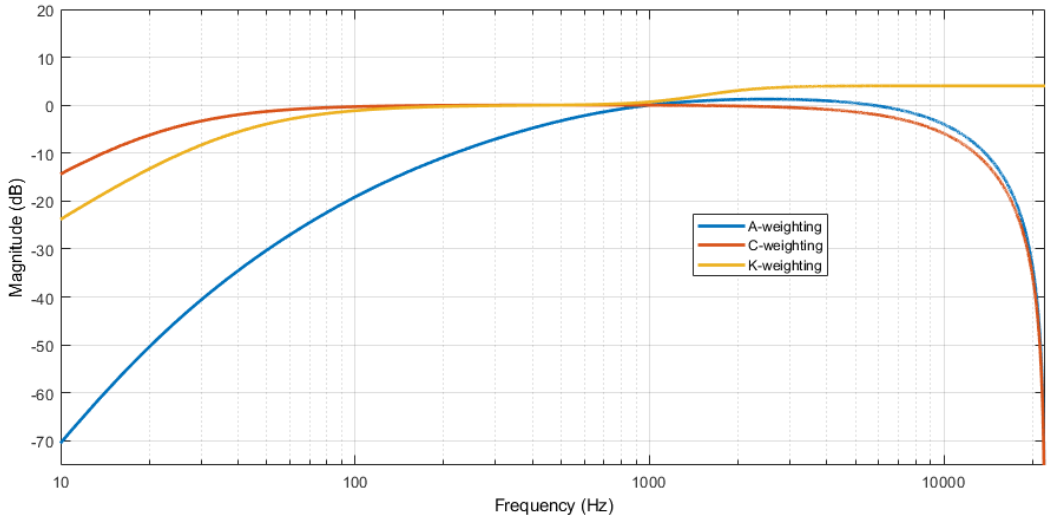
`audioOscillator` | Wavetable Synthesizer

weightingFilter

Frequency-weighted filter

Description

The `weightingFilter` System object performs frequency-weighted filtering independently across each input channel.



To perform frequency-weighted filtering:

1. Create the `weightingFilter` object and set its properties.
2. Call the object with arguments, as if it were a function.

To learn more about how System objects work, see What Are System Objects?

Creation

Syntax

```
weightFilt = weightingFilter
weightFilt = weightingFilter(weightType)
weightFilt = weightingFilter(weightType,Fs)
weightFilt = weightingFilter(___,Name,Value)
```

Description

`weightFilt = weightingFilter` creates a System object, `weightFilt`, that performs frequency-weighted filtering independently across each input channel.

`weightFilt = weightingFilter(weightType)` sets the Method property to `weightType`.

`weightFilt = weightingFilter(weightType,Fs)` sets the SampleRate property to `Fs`.

`weightFilt = weightingFilter( ___ ,Name,Value)` sets each property `Name` to the specified `Value`. Unspecified properties have default values.

Example: `weightFilt = weightingFilter('C-weighting','SampleRate',96000)` creates a C-weighting filter with a sample rate of 96,000 Hz.

Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see System Design in MATLAB Using System Objects.

`Method` — Type of weighting
`'A-weighting'` (default) | `'C-weighting'` | `'K-weighting'`

Type of weighting, specified as `'A-weighting'`, `'C-weighting'`, or `'K-weighting'`. See “Algorithms” on page 3-398 for more information.

Tunable: No

Data Types: `char` | `string`

`SampleRate` — Input sample rate (Hz)
`44100` (default) | positive scalar

Input sample rate in Hz, specified as a positive scalar.

Tunable: Yes

Data Types: `single` | `double`

Usage

Syntax

```
audioOut = weightFilt(audioIn)
```

Description

`audioOut = weightFilt(audioIn)` applies frequency-weighted filtering to the input signal, `audioIn`, and returns the filtered signal, `audioOut`. The type of filtering is specified by the algorithm and properties of the `weightingFilter` System object, `weightFilt`.

Input Arguments

`audioIn` — Audio input to weighting filter
matrix

Audio input to the weighting filter, specified as a matrix. The columns of the matrix are treated as independent audio channels.

Data Types: `single` | `double`

Output Arguments

`audioOut` — Audio output from weighting filter

matrix

Audio output from the weighting filter, returned as a matrix the same size as `audioIn`.

Data Types: `single` | `double`

Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

Specific to weightingFilter

visualize	Visualize and validate filter response
getFilter	Return biquad filter object with design parameters set
createAudioPluginClass	Create audio plugin class that implements functionality of System object
isStandardCompliant	Verify filter design is IEC 61672-1:2002 compliant

Common to All System Objects

clone	Create duplicate System object
isLocked	Determine if System object is in use
release	Release resources and allow changes to System object property values and input characteristics
reset	Reset internal states of System object
step	Run System object algorithm

Note `weightingFilter` supports additional filter analysis functions. See “Compare and Analyze Weighting Types” on page 3-393 for details.

Examples

Validate Filter Compliance

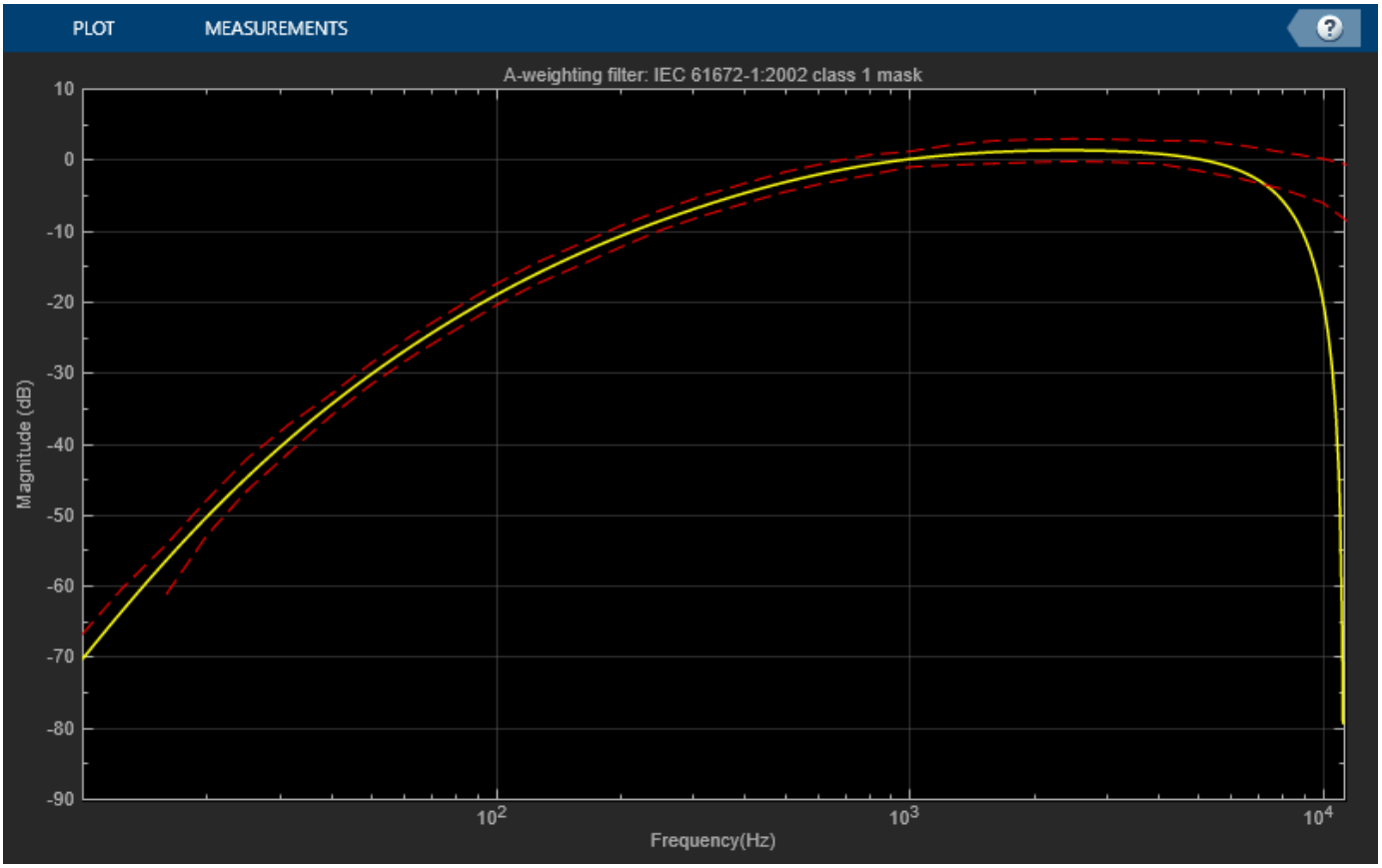
Check the compliance status of filter designs and visualize them.

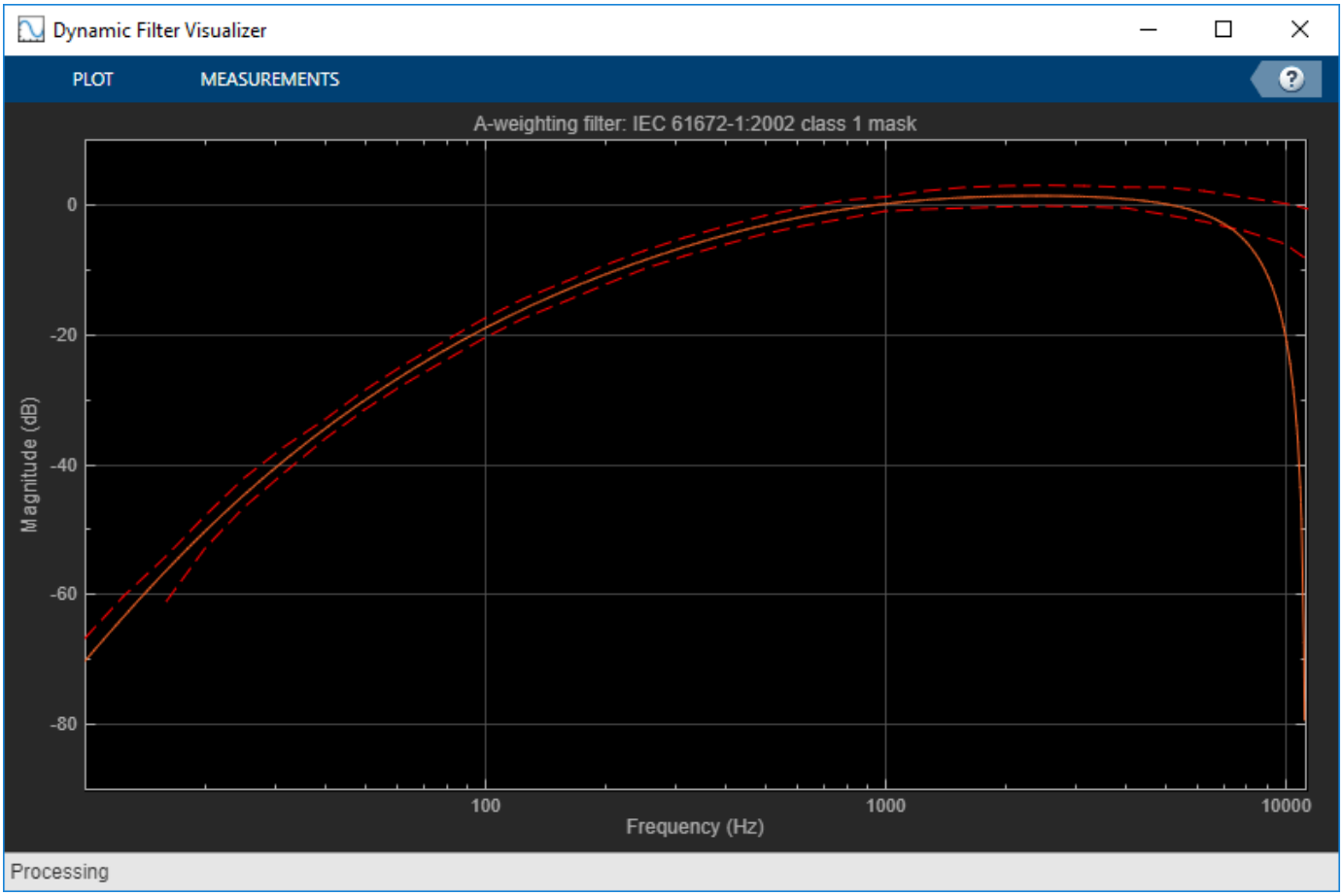
Create an A-weighting filter with a 22.5 kHz sample rate. Verify that the filter is standard compliant and visualize the filter design.

```
aWeight = weightingFilter('A-weighting','SampleRate',22500);
complianceStatus = isStandardCompliant(aWeight,'class 1')
```

```
complianceStatus = logical
   0
```

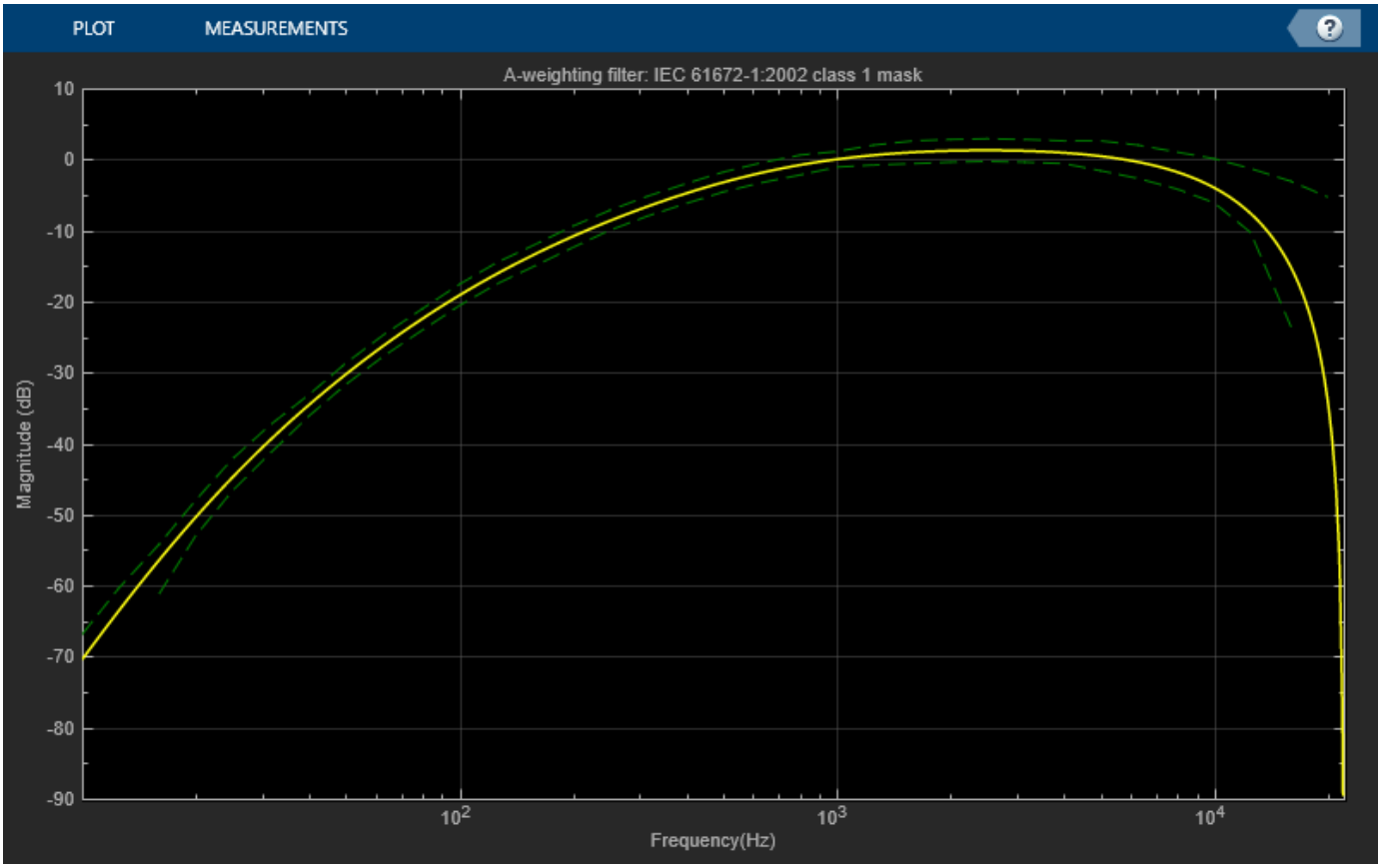
```
visualize(aWeight,'class 1')
```





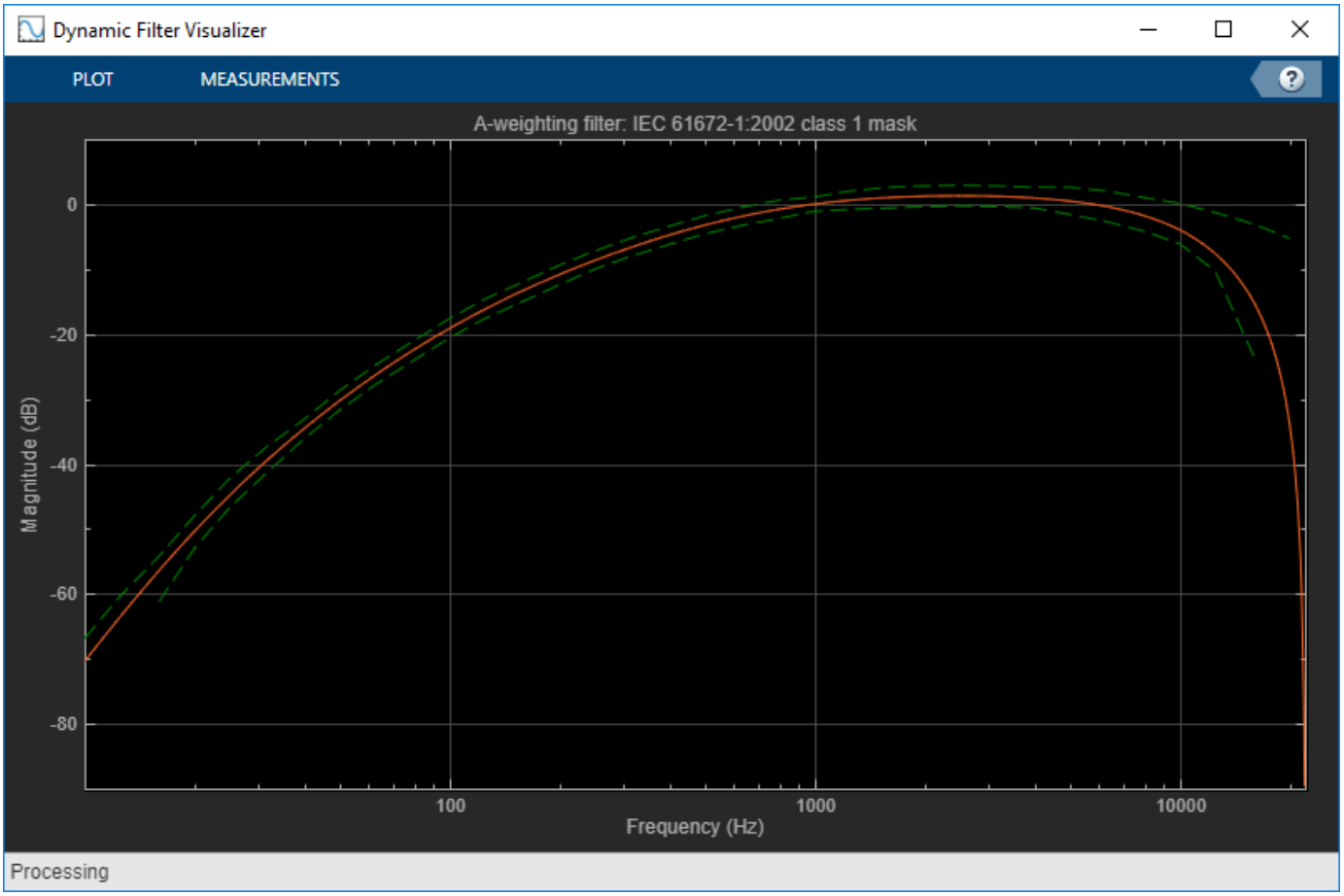
Change your A-weighting filter sample rate to 44.1 kHz. Verify that the filter is standard compliant and visualize the filter design.

```
aWeight.SampleRate = 44100;
```




```
complianceStatus = isStandardCompliant(aWeight,'class 1')
```

```
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 a `dsp.AudioFileReader` System object.

```
samplesPerFrame = 1024;
reader = dsp.AudioFileReader(Filename="RockGuitar-16-44p1-stereo-72secs.wav", ...
    SamplesPerFrame=samplesPerFrame, ...
    PlayCount=Inf);
```

Create a `weightingFilter` System object. Use the sample rate of the reader as the sample rate of the weighting filter.

```
Fs = reader.SampleRate;
weightFilt = weightingFilter("A-weighting",Fs);
```

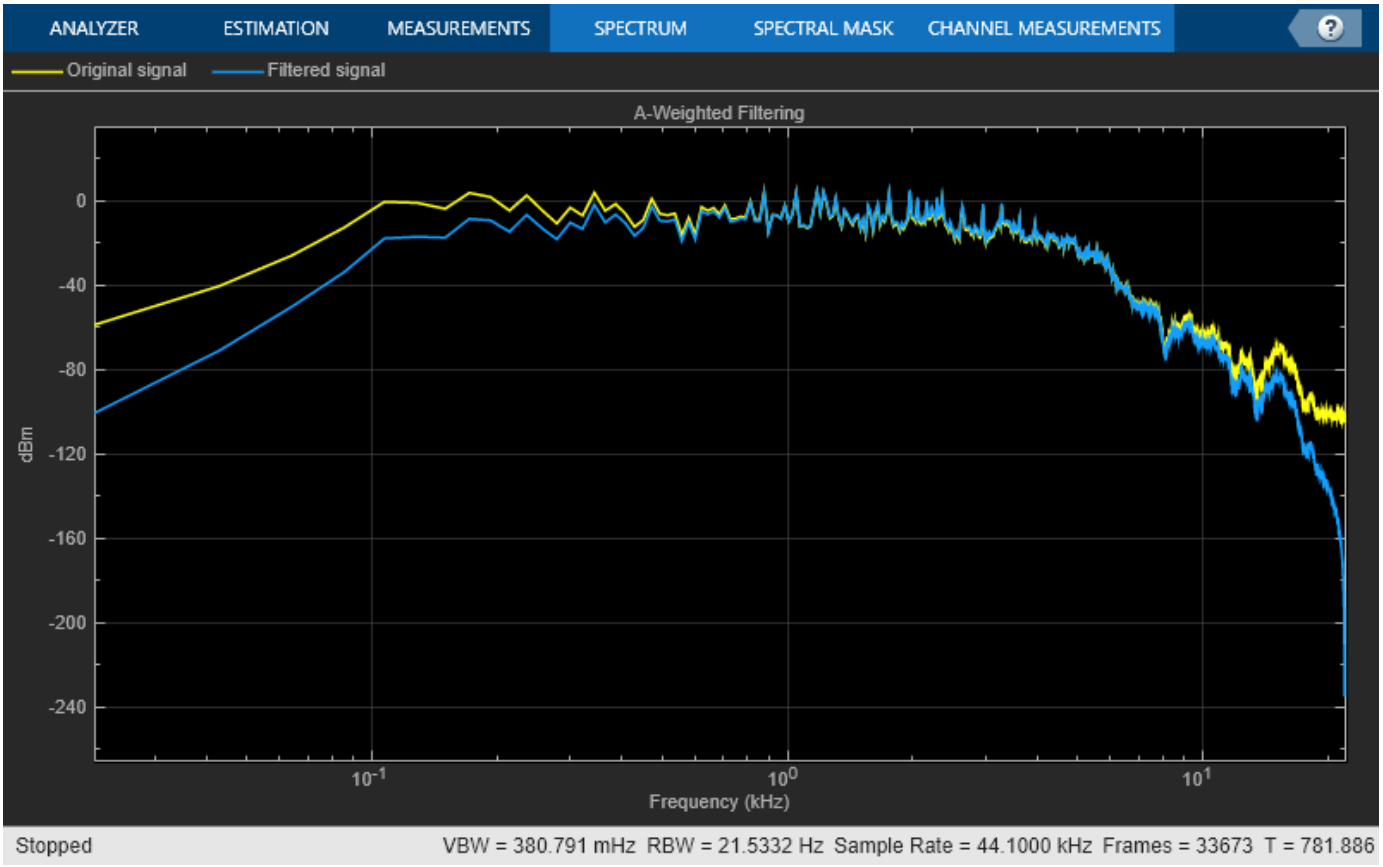
Create a spectrum analyzer to visualize the original audio signal and the audio signal after frequency-weighted filtering.

```
scope = spectrumAnalyzer( ...
    SampleRate=Fs, ...
```

```
    PlotAsTwoSidedSpectrum=false, ...
    FrequencyScale="log", ...
    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(reader)
release(scope)
```

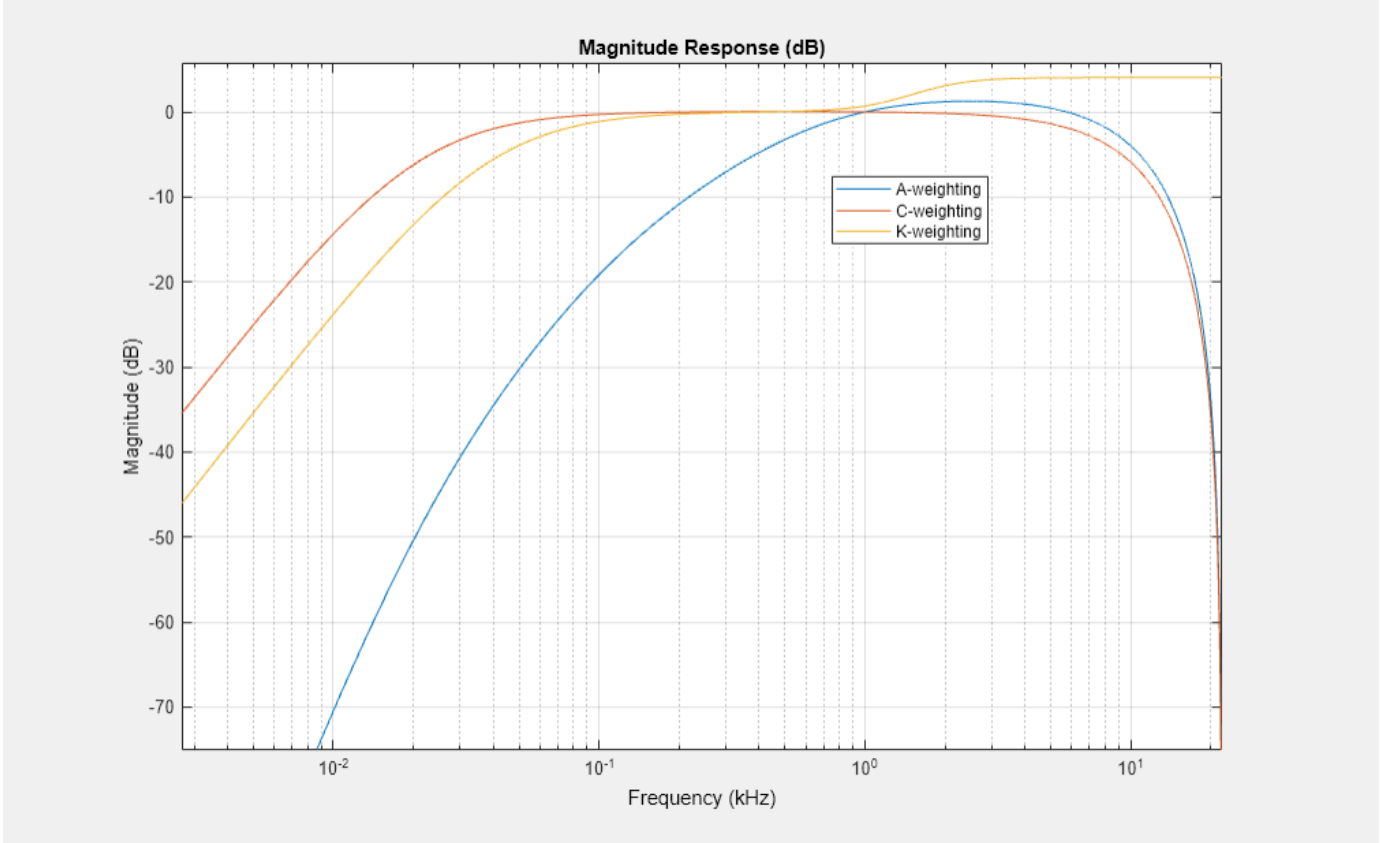


Compare and Analyze Weighting Types

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. Compare and analyze the filters using `FVTool`.

```
wF{1} = weightingFilter;
wF{2} = weightingFilter('C-weighting');
wF{3} = weightingFilter('K-weighting');

fvt = fvtool(wF{1},wF{2},wF{3},'FrequencyScale','Log', ...
    'Fs',wF{1}.SampleRate);
legend(fvt,'A-weighting','C-weighting','K-weighting', ...
    Location="best")
```



The `weightingFilter` object supports several filter analysis methods. For more information, use `help` at the command line:

```
help weightingFilter.helpFilterAnalysis
```

```
    The following analysis methods are available for discrete-time filter System objects:

    fvtool       -  Filter visualization tool
    info         -  Filter information
    freqz        -  Frequency response
    phasez       -  Phase response
    zerophase    -  Zero-phase response
    grpdelay     -  Group delay response
    phasedelay   -  Phase delay response
    impz         -  Impulse response
    impzlength   -  Length of impulse response
```

```
    stepz        -  Step response
    zplane       -  Pole/zero plot
    cost         -  Cost estimate for implementation of the filter System object
    measure      -  Measure characteristics of the frequency response

    outputDelay  -  Output delay value
    order        -  Filter order
    coeffs       -  Filter coefficients in a structure
    firtype      -  Determine the type (1-4) of a linear phase FIR filter System object
    tf           -  Convert to transfer function
    zpk          -  Convert to zero-pole-gain
    ss           -  Convert to state space representation

    isallpass    -  Verify if filter System object is allpass
    isfir        -  Verify if filter System object is FIR
    islinphase   -  Verify if filter System object is linear phase
    ismaxphase   -  Verify if filter System object is maximum phase
    isminphase   -  Verify if filter System object is minimum phase
    isreal       -  Verify if filter System object is minimum real
    issos        -  Verify if filter System object is in second-order sections form
    isstable     -  Verify if filter System object is stable

    realizemdl   -  Filter realization (Simulink diagram)

    specifyall   -  Fully specify fixed-point filter System object settings

    cascade      -  Create a FilterCascade System object

    Second-order sections:

    scale        -  Scale second-order sections of BiquadFilter System object
    scalecheck   -  Check scaling of BiquadFilter System object
    reorder      -  Reorder second-order sections of BiquadFilter System object
    cumsec       -  Cumulative second-order section of BiquadFilter System object
    scaleopts    -  Create an options object for second-order section scaling
    sos          -  Convert to second-order-sections (for IIRFilter System objects only)

    Fixed-Point (Fixed-Point Designer Required):

    freqrespest  -  Frequency response estimate via filtering
    freqrespopts -  Create an options object for frequency response estimate
    noisepsd     -  Power spectral density of filter output due to roundoff noise
    noisepsdopts -  Create an options object for output noise PSD computation

    Multirate Analysis:

    polyphase                - Polyphase decomposition of multirate filter System object
    gain (CIC decimator)     - Gain of CIC decimator filter System object
    gain (CIC interpolator)  - Gain of CIC interpolator filter System object

    For decimator, interpolator, or rate change filter System objects
    the analysis tools perform computations relative to the rate at
    which the filter is running. If a sampling frequency is specified,
    it is assumed that the filter is running at that rate.

Help for weightingFilter.helpFilterAnalysis is inherited from superclass dsp.internal.FilterAnaly
```

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...')
```

```
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
```

```
A-weighted filtering...

C-weighted filtering...

K-weighted filtering...
```

As a best practice, release your objects once done.

```
release(deviceWriter)
release(fileReader)
```

Use Weighting Filter Design with Biquad Filter

The `weightingFilter` object uses second-order sections (SOS) for filtering. To extract the weighting filter design, use `getFilter` to return a `dsp.BiquadFilter` object with the `SOSMatrix` and `ScaleValues` properties set.

Use `weightingFilter` to create C-weighted and A-weighted filter objects. Use `getFilter` to return corresponding `dsp.BiquadFilter` objects.

```
cFilt = weightingFilter('C-weighting');
aFilt = weightingFilter('A-weighting');
cSOSFilter = getFilter(cFilt);
aSOSFilter = getFilter(aFilt);
```

Create an audio file reader and audio device writer for audio input/output. Use the sample rate of your reader as the sample rate of your writer.

```
fileReader = dsp.AudioFileReader('JetAirplane-16-11p025-mono-16secs.wav');
deviceWriter = audioDeviceWriter('SampleRate',fileReader.SampleRate);
```

In an audio stream loop, play the unfiltered signal. Release your file reader so that the next time you call it, it reads from the beginning of the file.

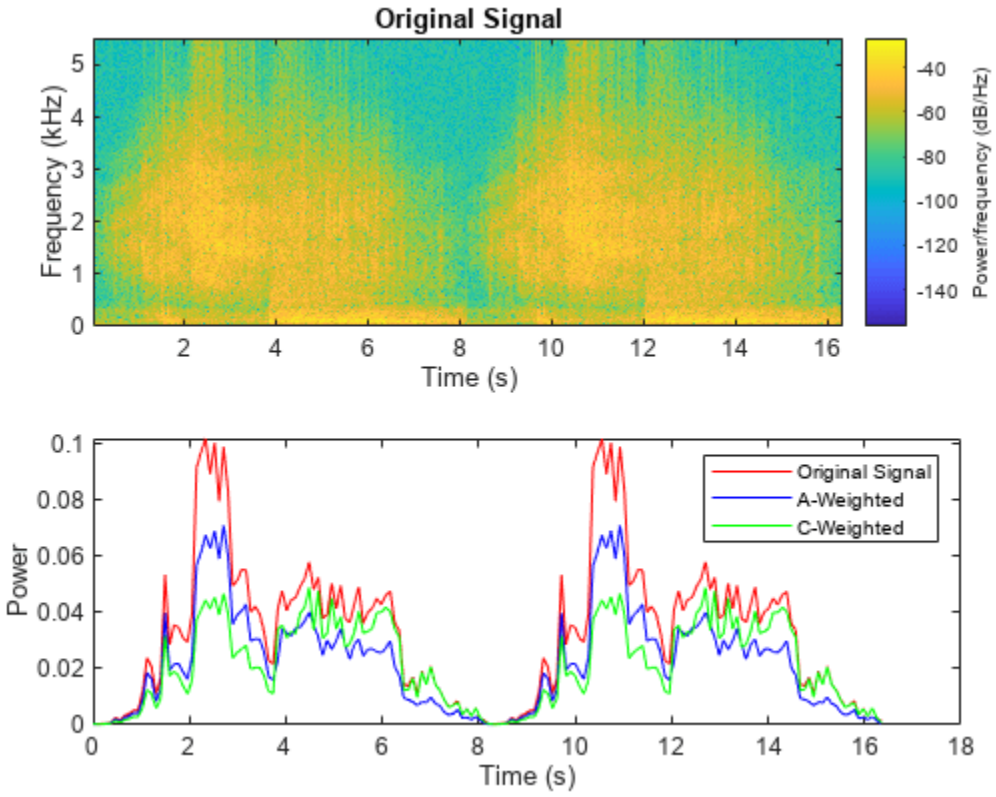
```
tic
while toc<8
    x = fileReader();
    deviceWriter(x);
end
release(fileReader)
```

Play the signal processed by the A-weighted filter. Then play the signal processed by the C-weighted filter. Cache the power in each frame of the original and filtered signals for analysis. As a best practice, release your file reader and device writer once complete.

```
y = [];
count = 1;
tic
while ~isDone(fileReader)
    x = fileReader();
    aFiltered = aSOSFilter(x);
    cFiltered = cSOSFilter(x);
    if toc>8
        deviceWriter(cFiltered);
    else
        deviceWriter(aFiltered);
    end
    xPower(count) = var(x);
    aPower(count) = var(aFiltered);
    cPower(count) = var(cFiltered);
    y = [y;x];
    count = count+1;
end

release(fileReader)
release(deviceWriter)
```

Plot the power of the original signal, the A-weighted signal, and the C-weighted signal over time.

```
subplot(2,1,1)
spectrogram(y,512,256,4096,fileReader.SampleRate,'yaxis')
title('Original Signal')

subplot(2,1,2)
t = linspace(0,16.3468,count-1);
plot(t,xPower,'r',t,aPower,'b',t,cPower,'g')
legend('Original Signal','A-Weighted','C-Weighted')
xlabel('Time (s)')
ylabel('Power')
```



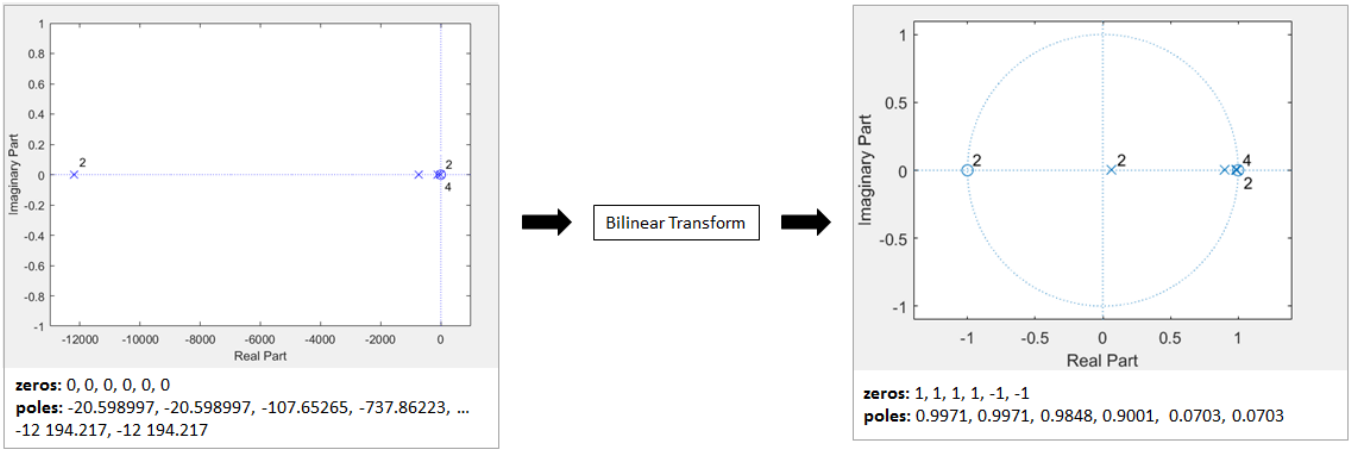
Algorithms

A-Weighting

The A-curve is a wide bandpass filter centered at 2.5 kHz, with approximately 20 dB attenuation at 100 Hz. 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 Toolbox converts the specified poles and zeros to the digital domain using a bilinear transform:

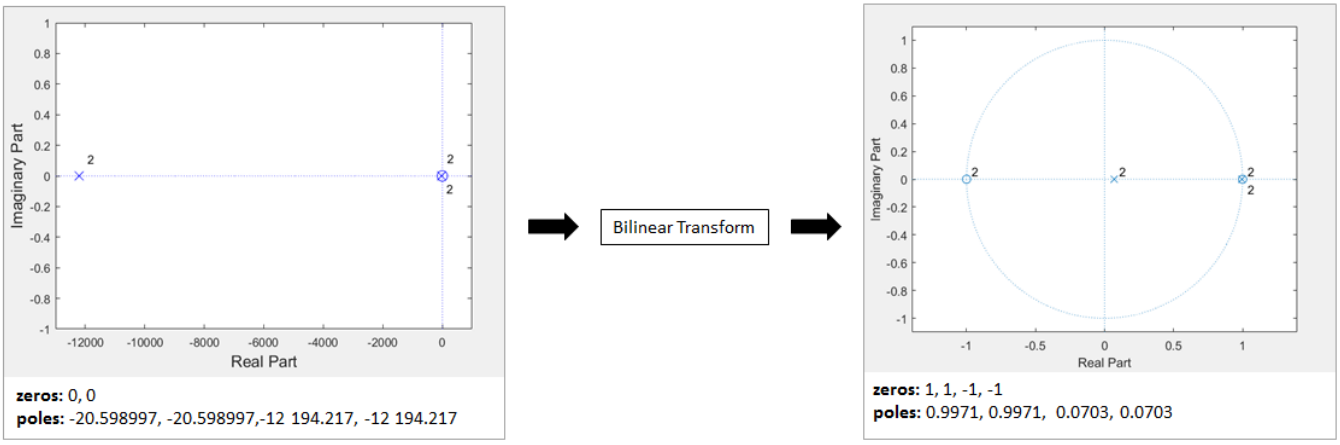


C-Weighting

The C-curve is "flat," but with limited bandwidth: It has –3 dB corners at 31.5 Hz and 8 kHz. C-curves are used in sound level meters for sounds that are louder than those intended for A-weighting filters.

The ANSI S1.42-2001 [1] defines the C-weighting curve. The IEC 61672-1:2002 [2] standard defines the minimum and maximum attenuation limits for C-weighting filters.

ANSI S1.42.2001 defines the weighting curve by specifying analog poles and zeros. Audio Toolbox converts the specified poles and zeros to the digital domain using a bilinear transform:

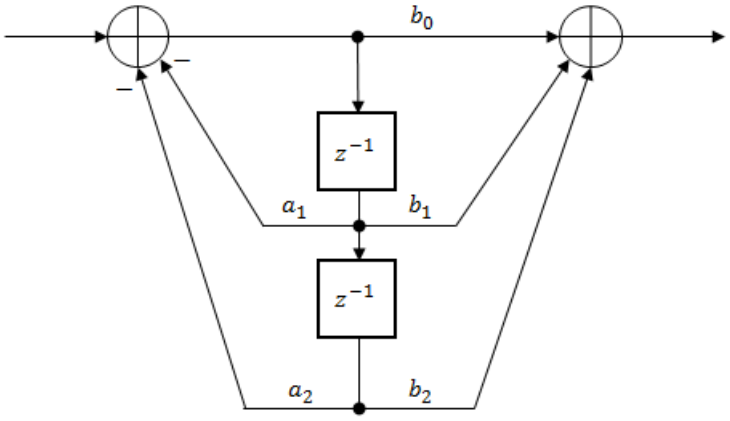


K-Weighting

The K-weighting filter is used for loudness normalization in broadcast. It is composed of two stages of filtering: a first stage shelving filter and a second stage highpass filter.

The ITU-R BS.1770-4 [3] standard defines this curve.

Assume a second-order filter.



The table shows the coefficients for the filters.

First Stage Shelving Coefficients	Second Stage Highpass Coefficients
$a_1 = -1.69065929318241$	$a_1 = -1.99004745483398$
$a_2 = 0.73248077421585$	$a_2 = 0.99007225036621$
$b_0 = 1.53512485958697$	$b_0 = 1.0$
$b_1 = -2.6916918940638$	$b_1 = -2.0$
$b_2 = 1.19839281085285$	$b_2 = 1.0$

The coefficients presented by ITU-R BS.1770-4 are defined for 48 kHz. These coefficients are recomputed for nonstandard sample rates using the algorithm described in [4].

Version History

Introduced in R2016b

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

Weighting Filter | `multibandParametricEQ` | `octaveFilter` | `dsp.BiquadFilter`

Topics

“Audio Weighting Filters”
“Sound Pressure Measurement of Octave Frequency Bands”

isStandardCompliant

Verify filter design is IEC 61672-1:2002 compliant

Syntax

```
complianceStatus = isStandardCompliant(weightFilt,classType)
complianceStatus = isStandardCompliant( ___ ,freqRange)
```

Description

`complianceStatus = isStandardCompliant(weightFilt,classType)` returns a logical scalar, `complianceStatus`, indicating whether the `weightFilt` filter design is compliant with the minimum and maximum attenuation specifications for the `classType` design specified in IEC 61672-1:2002. You can check compliance for A-weighting and C-weighting filters only.

`complianceStatus = isStandardCompliant( ___ ,freqRange)` specifies the range of frequencies checked for compliance.

Examples

Verify Class 1 Standard Compliance

Create an object of the `weightingFilter` System object™. Call `isStandardCompliant`, specifying the compliance class type to check as the second argument.

```
weightFilt = weightingFilter;
complianceStatus = isStandardCompliant(weightFilt,'class 1')
```

```
complianceStatus = logical
   1
```

Specify Frequency Range Checked for Compliance

Create an object of the `weightingFilter` System object™. Check the 'class 2' compliance status of the filter design over a specified frequency range.

```
weightFilt = weightingFilter;
isStandardCompliant(weightFilt,'class 2',[120,2000])
```

```
ans = logical
   1
```

Input Arguments

`weightFilt` — Object of `weightingFilter`
object

Object of the `weightingFilter` System object.

`classType` — Compliance class type
`'class 1'` | `'class 2'`

Compliance class type to verify, specified as `'class 1'` or `'class 2'`.

Data Types: `char`

`freqRange` — Frequency range checked for compliance (Hz)
`[minFreq,maxFreq]` | two-element vector of increasing values

Specify the frequency range, in Hz, checked for compliance as a two-element vector of increasing values: `[minFreq,maxFreq]`.

Data Types: `single` | `double`

Output Arguments

`complianceStatus` — Compliance status of filter design
scalar

Compliance status of filter design, returned as a logical scalar. The compliance status indicates whether the `weightFilt` filter design is compliant with the minimum and maximum attenuation specifications for the class type design specified by IEC 61672-1:2002 standard. Compliance can only be checked for A-weighting and C-weighting filters.

Data Types: `logical`

Note The pole-zero values defined in the ANSI S1.42-2001 standard are used for designing the A-weighted and C-weighted filters. The pole-zero values are based on analog filters, so the design can break compliance for lower sample rates.

Version History

Introduced in R2016b

See Also

Topics
“Audio Weighting Filters”
“Sound Pressure Measurement of Octave Frequency Bands”

visualize

Visualize and validate filter response

Syntax

```
visualize(weightFilt)
visualize(weightFilt,N)
visualize( ___ ,mType)
hvsz = visualize( ___ )
```

Description

`visualize(weightFilt)` plots the magnitude response of the frequency-weighted filter `weightFilt`. The plot is updated automatically when properties of the object change.

`visualize(weightFilt,N)` uses an `N`-point FFT to calculate the magnitude response.

`visualize( ___ ,mType)` creates a mask based on the class of filter specified by `mType`, using either of the previous syntaxes.

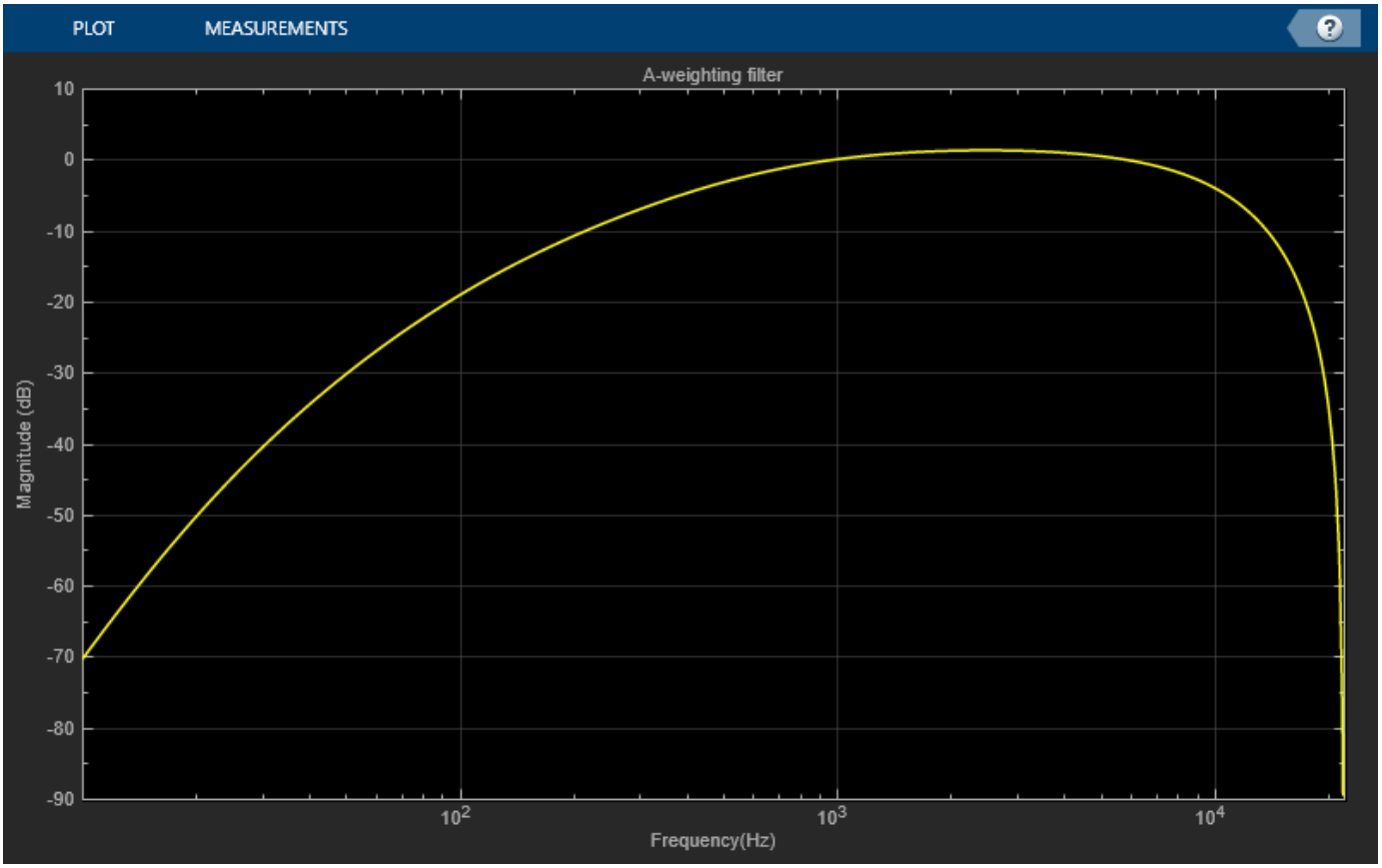
`hvsz = visualize( ___ )` returns a handle to the visualizer as a `dsp.DynamicFilterVisualizer` object when called with any of the previous syntaxes.

Examples

Plot Weighting Filter Magnitude Response

Create a `weightingFilter` System object™ and then plot the magnitude response of the filter.

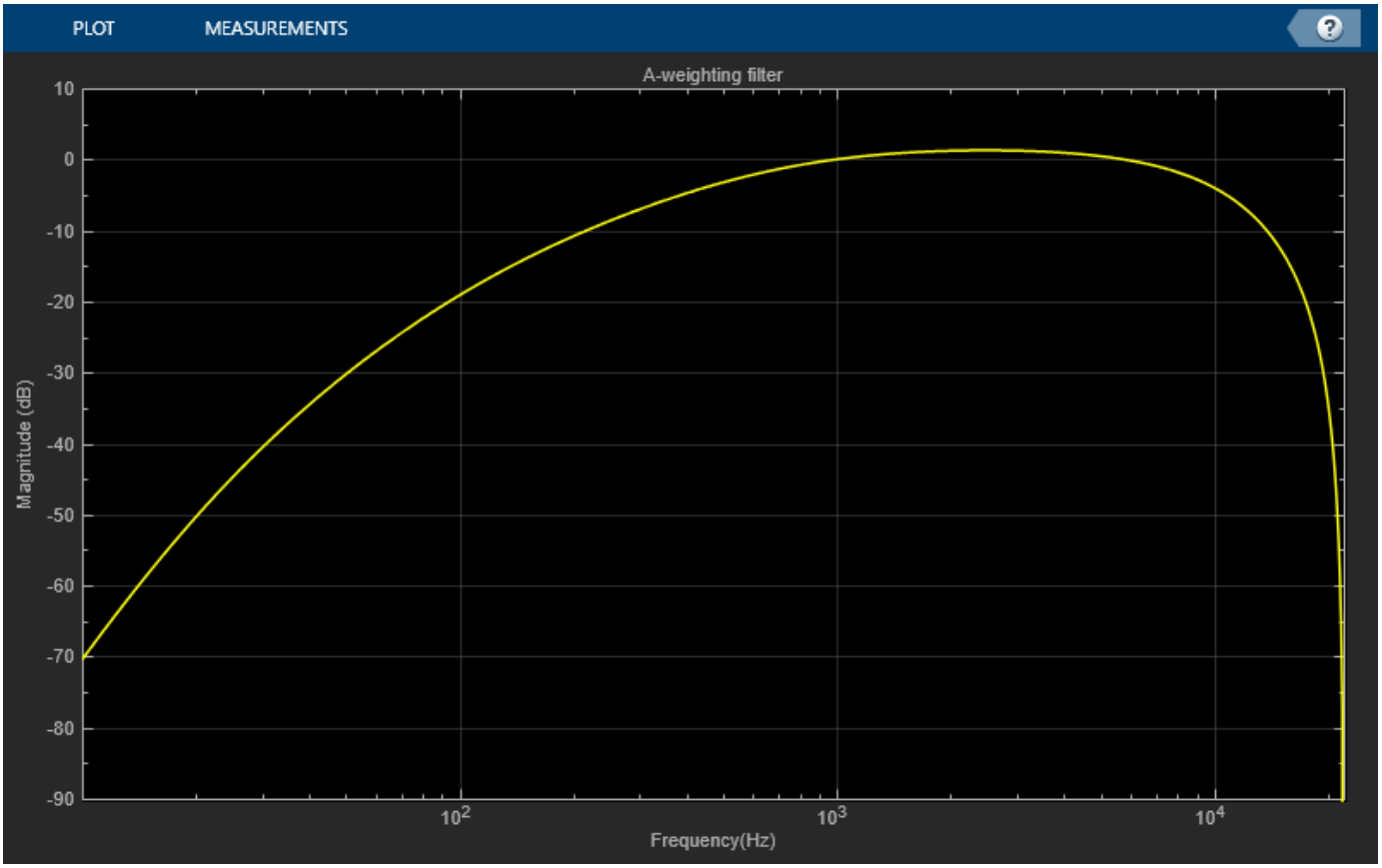
```
weightFilt = weightingFilter;
visualize(weightFilt)
```



Specify Number of Frequency Bins in FFT Calculation

Create a `weightingFilter` System object™. Plot a 1024-point frequency representation.

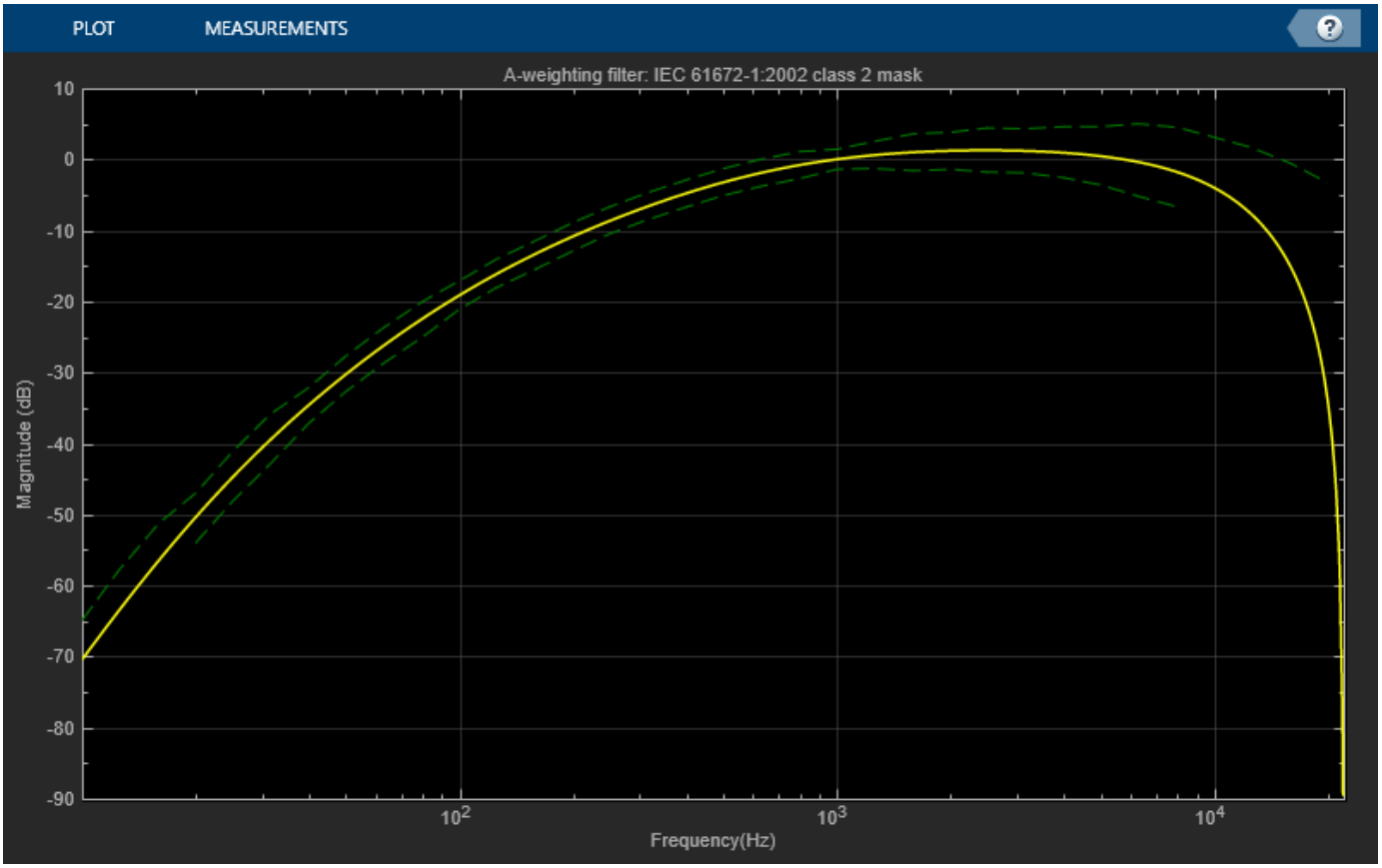
```
weightFilt = weightingFilter;
visualize(weightFilt,1024)
```



Visualize Class 2 Standard-Compliance Mask

Create a `weightingFilter` System object™. Visualize the class 2 compliance of the filter design.

```
weightFilt = weightingFilter;
visualize(weightFilt,'class 2')
```



Input Arguments

`weightFilt` — Object of `weightingFilter`

object

Object of the `weightingFilter` System object.

`N` — Number of DFT bins

`2048` | positive scalar

Number of DFT bins in frequency-domain representation, specified as a positive scalar. The default is `2048`.

Data Types: `single` | `double`

`mType` — Type of mask

`'class 1'` (default) | `'class 2'`

Type of mask, specified as `'class 1'` or `'class 2'`.

The mask attenuation limits are defined in the IEC 61672-1:2002 standard. The mask is defined for A-weighting and C-weighting filters only.

- If the mask is green, the design is compliant with the IEC 61672-1:2002 standard.

- If the mask is red, the design breaks compliance.

Note The pole-zero values defined in the ANSI S1.42-2001 standard are used for designing the A-weighted and C-weighted filters. The pole-zero values are based on analog filters, so the design can break compliance for lower sample rates.

Data Types: `char`

Version History

Introduced in R2016b

See Also

Topics

“Audio Weighting Filters”
“Sound Pressure Measurement of Octave Frequency Bands”

audioLevelMeter

Measure digital audio peak level

Description

The `audioLevelMeter` System object measures the digital peak level of an audio signal. You can use `audioLevelMeter` to identify and prevent audio clipping during recording and playback.

To measure the peak level:

1. Create the `audioLevelMeter` object and set its properties.
2. Call the object with arguments, as if it were a function.

To learn more about how System objects work, see What Are System Objects?

Creation

Syntax

```
lvlMeter = audioLevelMeter
lvlMeter = audioLevelMeter(Name=Value)
```

Description

`lvlMeter = audioLevelMeter` creates an audio level meter System object with default property values.

`lvlMeter = audioLevelMeter(Name=Value)` sets "Properties" on page 3-409 using one or more name-value arguments.

Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see System Design in MATLAB Using System Objects.

`Method` — Measurement method

`"sample-peak"` (default) | `"true-peak"`

Measurement method to compute the peak levels, specified as `"sample-peak"` or `"true-peak"`. The true-peak measurement method follows the ITU-R BS.1770-4 standards [2]. For more information about these methods, see "Algorithms" on page 3-414.

Data Types: `char` | `string`

`WindowLength` — Window length
1024 (default) | positive integer

Window length, specified as a positive integer. The window length defines the size of the nonoverlapping frames in the input signal over which the object computes each peak level.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`DecayRate` — Decay rate of peak values (dB/s)
11.76 (default) | nonnegative scalar

Decay rate of the peak values in dB/s, specified as a nonnegative scalar. For more information about the decay, see “Decay” on page 3-415.

A decay rate of `Inf` means the object does not apply the decay.

Tunable: Yes

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`Units` — Output units
`"dBFS"` | `"dBTP"`

This property is read-only.

Output units of the measured peak values, returned as `"dBFS"` or `"dBTP"`. The units are `"dBFS"` if you set Method to `"sample-peak"`, and `"dBTP"` if you set `Method` to `"true-peak"`.

Data Types: `string`

`SampleRate` — Sample rate of input signal (Hz)
44100 (default) | finite positive scalar

Sample rate of input signal in Hz, specified as a finite positive scalar.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Usage

Syntax

```
peakLevel = lvlMeter(audioIn)
```

Description

`peakLevel = lvlMeter(audioIn)` computes the peak levels over nonoverlapping frames in the input signal.

Input Arguments

`audioIn` — Audio input signal
column vector | matrix

Audio input signal, specified as a column vector or matrix. If the input is a matrix, the object treats the columns as independent channels

Output Arguments

peakLevel — Peak level

matrix

Peak level of each frame in the input signal, returned as an *L*-by-*C* matrix, where *C* is the number of channels and *L* is determined by the WindowLength property and the length of `audioIn`.

Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

Specific to audioLevelMeter

clone	Create duplicate System object
isLocked	Determine if System object is in use

Common to All System Objects

step	Run System object algorithm
release	Release resources and allow changes to System object property values and input characteristics
reset	Reset internal states of System object

Examples

Measure Sample-Peak Level of Audio Signal

Create a `dsp.AudioFileReader` to stream an audio file for processing. Create an `audioDeviceWriter` to play the audio as you stream it.

```
reader = dsp.AudioFileReader("FunkyDrums-44p1-stereo-25secs.mp3");
player = audioDeviceWriter(SampleRate=reader.SampleRate);
```

Create an `audioLevelMeter` object. Specify the `SampleRate` and `WindowLength` to match the file reader object.

```
lvlMeter = audioLevelMeter(SampleRate=reader.SampleRate, ...
    WindowLength=reader.SamplesPerFrame);
```

Create a `timescope` object to plot both the audio signal and the measured peak level.

```
scope = timescope(SampleRate=reader.SampleRate,LayoutDimensions=[2,1], ...
    TimeSpanSource="property",TimeSpan=5);
scope.Title = "Audio Waveform";
scope.YLabel = "Amplitude";
scope.YLimits = [-1.25, 1.25];

scope.ActiveDisplay = 2;
scope.Title = "Peak Level";
scope.YLabel = lvlMeter.Units;
```

```
scope.YLimits = [-40, 6];
scope.ChannelNames = ["Channel 1", "Channel 2", "Channel 1", "Channel 2"];
```

In a streaming loop:

1. Read in a frame of audio data.
2. Compute the sample-peak level of the frame.
3. Plot the signal and the peak level.
4. Play the audio with the device writer.

```
while ~isDone(reader)
    audioIn = reader();
    peakLevel = lvlMeter(audioIn);
    peakLevelHold = peakLevel.*ones(size(audioIn));
    scope(audioIn,peakLevelHold);
    player(audioIn);
end
```

Measure True-Peak Level of Audio Signal

Create a `dsp.AudioFileReader` to stream an audio file for processing. Create an `audioDeviceWriter` to play the audio as you stream it.

```
reader = dsp.AudioFileReader("FunkyDrums-44p1-stereo-25secs.mp3");
player = audioDeviceWriter(SampleRate=reader.SampleRate);
```

Create an `audioLevelMeter` object. Set the `Method` property to `"true-peak"` to use the true-peak measurement method according to the ITU-R BS.1770-4 standards. Specify the `SampleRate` and `WindowLength` to match the file reader object.

```
lvlMeter = audioLevelMeter(Method="true-peak",SampleRate=reader.SampleRate, ...
    WindowLength=reader.SamplesPerFrame);
```

Create a `timescope` object to plot both the audio signal and the measured peak level.

```
scope = timescope(SampleRate=reader.SampleRate,LayoutDimensions=[2,1], ...
    TimeSpanSource="property",TimeSpan=5);
scope.Title = "Audio Waveform";
scope.YLabel = "Amplitude";
scope.YLimits = [-1.25, 1.25];

scope.ActiveDisplay = 2;
scope.Title = "Peak Level";
scope.YLabel = lvlMeter.Units;
scope.YLimits = [-40, 6];
scope.ChannelNames = ["Channel 1", "Channel 2", "Channel 1", "Channel 2"];
```

In a streaming loop:

1. Read in a frame of audio data.
2. Compute the true-peak level of the frame.
3. Plot the signal and the peak level.
4. Play the audio with the device writer.

```
while ~isDone(reader)
    audioIn = reader();
    peakLevel = lvlMeter(audioIn);
    peakLevelHold = peakLevel.*ones(size(audioIn));
    scope(audioIn,peakLevelHold);
    player(audioIn);
end
```

Algorithms

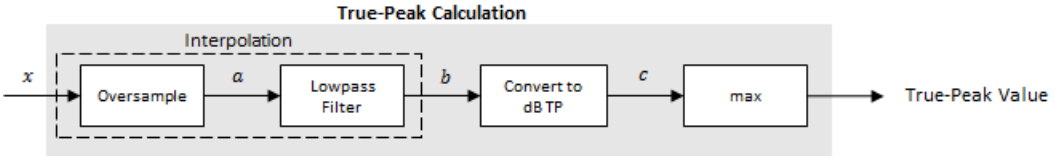
Sample-Peak Measurement

The sample-peak level of an audio frame is simply the maximum of the sample values converted to decibels.

```
max(20*log10(abs(audioFrame)))
```

True-Peak Measurement

Use the true-peak level to accurately measure peaks that can occur between samples in the reconstructed analog version of the digital audio signal.



1. The signal is oversampled to at least 192 kHz. To optimize processing, the input sample rate determines the exact oversampling. The algorithm does not consider an input sample rate below 750 Hz.

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

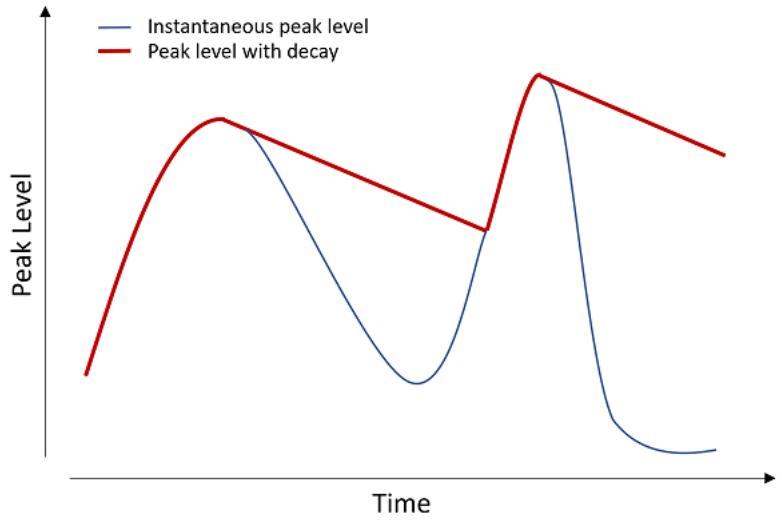
2. The oversampled signal a passes through a lowpass filter with a half-polyphase length of 12 and stopband attenuation of 80 dB. The filter design uses `designMultirateFIR`.
3. The filtered signal b is rectified and converted to the dB TP scale:

   $$c = 20 \times \log_{10}(|b|)$$

4. The true-peak is determined as the maximum of the converted signal c.

Decay

For both measurement methods, the `audioLevelMeter` object applies a linear decay to the peak level in the dB scale to prevent a steep dropoff. It applies the delay before it takes the maximum peak level from each frame.



The default decay rate of 11.76 dB/s satisfies the IEC TR 60268-18:1995 standard [1], which specifies 1.7 (+/- 0.3) seconds for the return time of a 20-dB fall in the peak value.

Version History

Introduced in R2023a

References

[1] IEC TR 60268-18:1995. "Sound system equipment - Part 18: Peak programme level meters - Digital audio peak level meter." *International Electrotechnical Commission*.

[2] ITU-R BS.1770-4. "Algorithms to measure audio programme loudness and true-peak audio level." *International Telecommunication Union, Radiocommunication Sector*.

See Also

Objects
`loudnessMeter` | `splMeter`

Functions
`integratedLoudness` | `acousticLoudness`

Classes

plotFeatures

Plot extracted audio features

Syntax

```
plotFeatures(afe,audioIn)
plotFeatures( ___ ,Name=Value)
figureHandle = plotFeatures( ___ )
```

Description

`plotFeatures(afe,audioIn)` extracts the enabled features of the `audioFeatureExtractor` object `afe` from the audio input and plots them.

`plotFeatures( ___ ,Name=Value)` specifies properties of the plot using one or more name-value arguments. For example, to plot the audio signal along with the features, set `PlotInput` to `true`.

`figureHandle = plotFeatures( ___ )` returns a handle to the figure containing the plot.

Examples

Visualize Extracted Audio Features

Use `plotFeatures` to visualize audio features extracted with an `audioFeatureExtractor` object.

Read in an audio signal from a file.

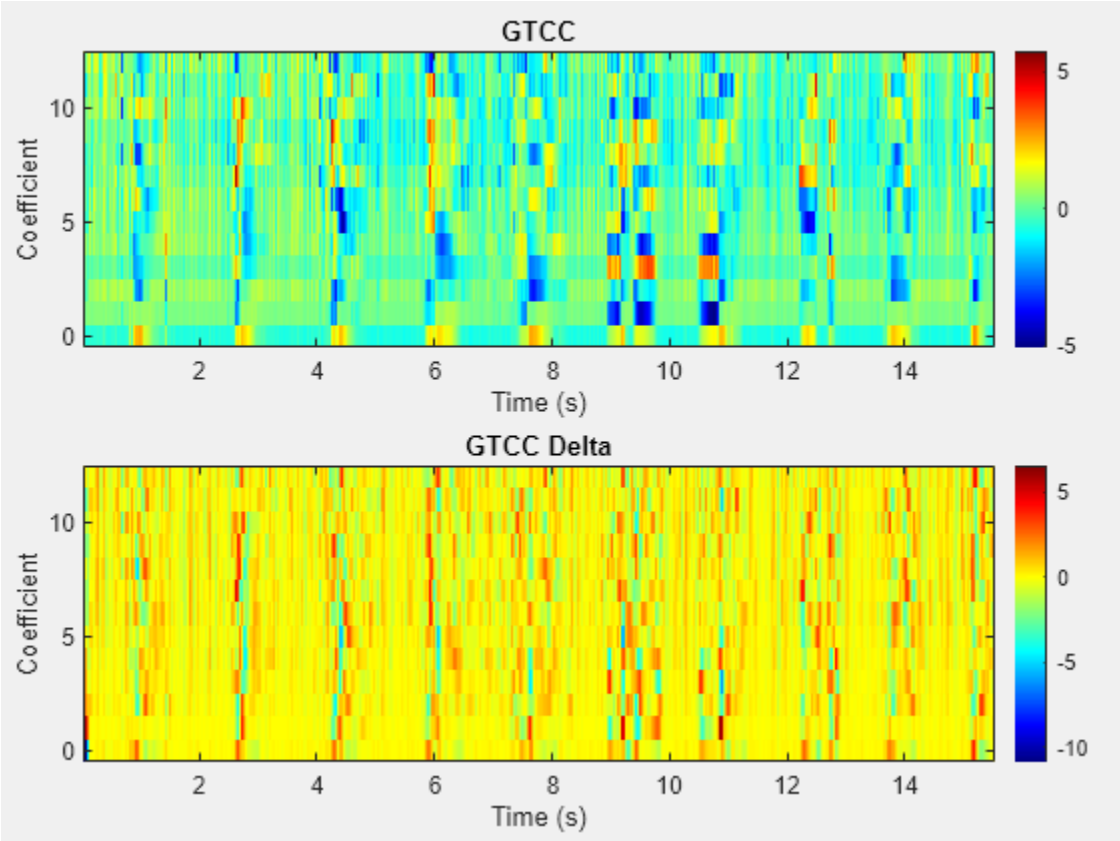
```
[audioIn,fs] = audioread("Counting-16-44p1-mono-15secs.wav");
```

Create an `audioFeatureExtractor` object that extracts the gammatone cepstral coefficients (GTCCs) and the delta of the GTCCs. Set the `SampleRate` property to the sample rate of the audio signal, and use the default values for the other properties.

```
afe = audioFeatureExtractor(SampleRate=fs,gtcc=true,gtccDelta=true);
```

Plot the features extracted from the audio signal.

```
plotFeatures(afe,audioIn)
```



Plot Audio Signal with Extracted Features

Read in an audio signal from a file.

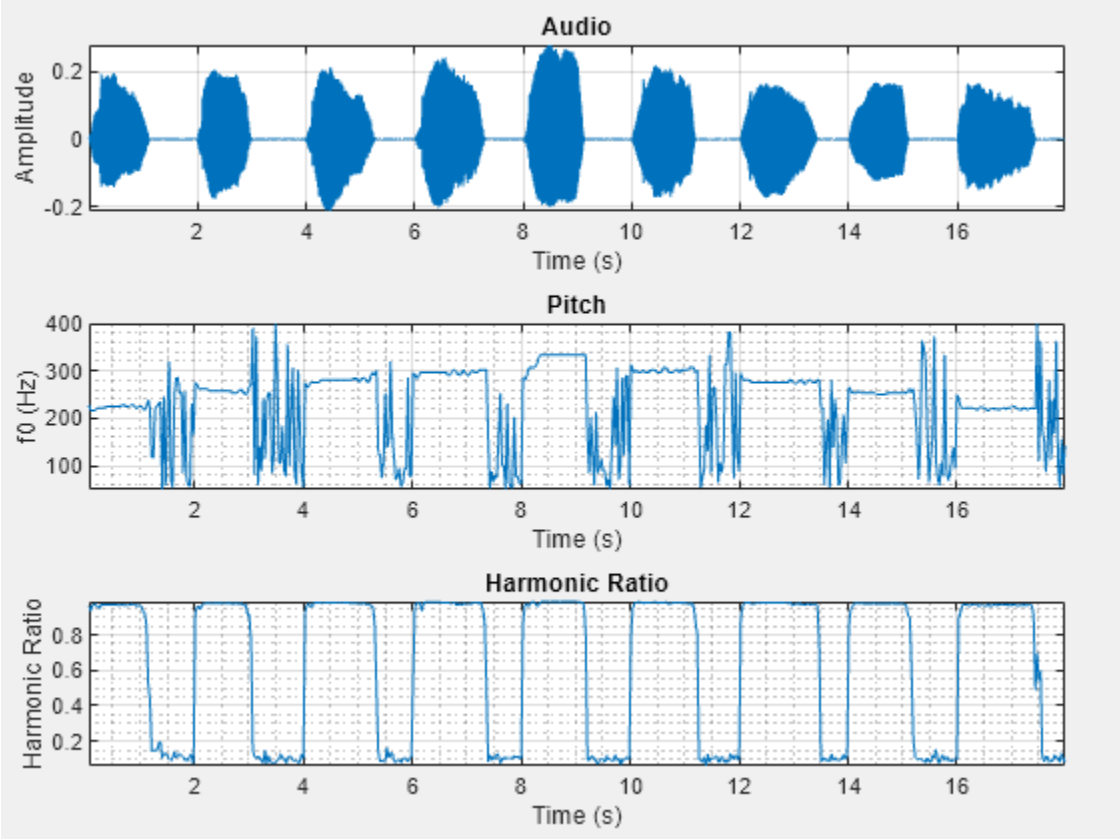
```
[audioIn,fs] = audioread("SingingAMajor-16-mono-18secs.ogg");
```

Create an `audioFeatureExtractor` object that extracts the pitch and harmonic ratio. Set the `SampleRate` property to the sample rate of the audio signal, and use the default values for the other properties.

```
afe = audioFeatureExtractor(SampleRate=fs,pitch=true,harmonicRatio=true);
```

Plot the features with `PlotInput` set to `true` to include the audio signal in the plot.

```
plotFeatures(afe,audioIn,PlotInput=true)
```



Input Arguments

`afe` — Input object

`audioFeatureExtractor` object

Input object, specified as an `audioFeatureExtractor` object. The `audioFeatureExtractor` object must have at least one feature enabled.

`audioIn` — Input audio

column vector | matrix

Input audio, specified as a column vector or matrix of independent channels (columns). If the input has multiple channels, then the function plots only the features of the first channel. The length of the audio signal must be greater than or equal to the length of the `Window` property of the `audioFeatureExtractor` object.

Data Types: `single` | `double`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `plotFeatures(afe,audioIn,PlotInput=true)`

`PlotInput` — Plot input audio signal
`false` (default) | `true`

Plot the input audio signal in addition to the extracted features, specified as `true` or `false`. If `audioIn` has multiple channels (columns), then the function plots only the first channel.

Data Types: `logical`

`Parent` — Parent container
container object

Parent container for the plot, specified as an object that is a valid `Parent` of a `TiledChartLayout` object.

Output Arguments

`figureHandle` — Figure handle
`Figure` object

Handle to the figure containing the plot, returned as a `Figure` object.

If the `Parent` argument is specified, `plotFeatures` returns the specified parent.

Version History

Introduced in R2022b

See Also

`audioFeatureExtractor` | `extract`

generateMATLABFunction

Create MATLAB function compatible with C/C++ code generation

Syntax

```
generateMATLABFunction(afe)
generateMATLABFunction(afe,fileName)

generateMATLABFunction( ___ ,'IsStreaming',TF)
```

Description

Generate Equivalent MATLAB Function

`generateMATLABFunction(afe)` generates code and opens an untitled file containing a function named `extractAudioFeatures`. The generated MATLAB function has the signature:

```
featureVector = extractAudioFeatures(audioIn)
```

The signature is equivalent to:

```
featureVector = extract(afe,audioIn)
```

`generateMATLABFunction(afe,fileName)` generates code and saves the resulting function to the file specified by `fileName`. The generated MATLAB function has the signature:

```
featureVector = functionName(audioIn)
```

The signature is equivalent to:

```
featureVector = extract(afe,audioIn)
```

Generate MATLAB Function for Stream Processing

`generateMATLABFunction( ___ ,'IsStreaming',TF)` specifies whether the function is intended for stream (single-frame) processing. If `TF` is specified as `true`, the resulting function requires single-frame input of length `numel(afe.Window)`. If individual feature extractors have state, the resulting function maintains the state between calls. If unspecified, `TF` defaults to `false`. The streaming function has the signature:

```
featureVector = functionName(audioIn,varargin)
```

The size of `featureVector` depends on the value of `IsStreaming`.

- If `IsStreaming` was set to `true`, then `featureVector` is returned as an *M*-by-*N* matrix, where *M* is the number of features extracted and *N* is the number of channels.
- If `IsStreaming` was set to `false`, then `featureVector` is returned as an *L*-by-*M*-by-*N* array, where *L* is the number of hops, *M* is the number of feature vectors, and *N* is the number of channels.

The possible values of `varargin` depends on the configuration of your `audioFeatureExtractor` object, `afe`.

- If the features your `audioFeatureExtractor` object extracts do not require state, then `varargin` must be empty.
- If the features your `audioFeatureExtractor` object extracts require state, then `varargin` can be the optional name-value pair `'Reset'` and either `true` or `false`. If you call the function with `'Reset'` set to `true`, then the function clears any state before calculating and returning the feature vector.

Examples

Generate Equivalent MATLAB® Function

You can use the `audioFeatureExtractor` object while developing a feature extraction pipeline in MATLAB. Once you are ready to deploy your system to a device or integrate it into a larger system, use `generateMATLABFunction` to create a MATLAB function suitable for C/C++ code generation. Then use MATLAB Coder™ to generate equivalent C/C++ code.

Read in an audio file. You will use this audio file to verify the equivalency of the `audioFeatureExtractor` object and the generated MATLAB function.

```
[audioIn,fs] = audioread('Counting-16-44p1-mono-15secs.wav');
```

Create an `audioFeatureExtractor` object to extract the Bark spectrum, the delta gammatone cepstral coefficients (GTCC), and the harmonic ratio.

```
afe = audioFeatureExtractor("Window",hann(512,"periodic"), ...
    "OverlapLength",256, ...
    "SampleRate",fs, ...
    "FFTLength",1024, ...
    'barkSpectrum',true, ...
    "gtccDelta",true, ...
    "harmonicRatio",true)
```

```
afe =
  audioFeatureExtractor with properties:

   Properties
                     Window: [512x1 double]
              OverlapLength: 256
                 SampleRate: 44100
                  FFTLength: 1024
     SpectralDescriptorInput: 'linearSpectrum'
         FeatureVectorLength: 46

   Enabled Features
     barkSpectrum, gtccDelta, harmonicRatio

   Disabled Features
     linearSpectrum, melSpectrum, erbSpectrum, mfcc, mfccDelta, mfccDeltaDelta
     gtcc, gtccDeltaDelta, spectralCentroid, spectralCrest, spectralDecrease, spectralEntropy
     spectralFlatness, spectralFlux, spectralKurtosis, spectralRolloffPoint, spectralSkewness, sp
     spectralSpread, pitch, zerocrossrate, shortTimeEnergy


   To extract a feature, set the corresponding property to true.
```

```
   For example, obj.mfcc = true, adds mfcc to the list of enabled features.
```

Call `generateMATLABFunction` on the object and specify a name for the generated MATLAB function.

```
functionName = 'extractAudioFeatures';
generateMATLABFunction(afe,functionName)
```

The generated function is saved to your current folder.

```
type extractAudioFeatures
```

```
function featureVector = extractAudioFeatures(x)
%extractAudioFeatures Extract multiple features from batch audio
%   featureVector = extractAudioFeatures(audioIn) returns audio features
%   extracted from audioIn.
%
%   Parameters of the audioFeatureExtractor used to generate this
%   function must be honored when calling this function.
%    - Sample rate of the input should be 44100 Hz.
%    - Input frame length should be greater than or equal to 512 samples.
%
%
%      % EXAMPLE 1: Extract features
%        source = dsp.ColoredNoise("SamplesPerFrame",44100);
%        for ii = 1:10
%            audioIn = source();
%            featureArray = extractAudioFeatures(audioIn);
%            % ... do something with featureArray ...
%        end
%
%
%      % EXAMPLE 2: Generate code
%        targetDataType = "single";
%        codegen extractAudioFeatures -args {ones(44100,1,targetDataType)}
%        source = dsp.ColoredNoise("SamplesPerFrame",44100, ...
%                                  "OutputDataType",targetDataType);
%        for ii = 1:10
%            audioIn = source();
%            featureArray = extractAudioFeatures_mex(audioIn);
%            % ... do something with featureArray ...
%        end
%
%   See also audioFeatureExtractor, dsp.AsyncBuffer, codegen.

%   Generated by audioFeatureExtractor on 03-Mar-2023 22:45:26 UTC-05:00
%#codegen

dataType = underlyingType(x);
[numSamples,numChannels] = size(x);

props = coder.const(getProps(dataType));

persistent config outputIndex
if isempty(outputIndex)
    [config, outputIndex] = coder.const(@getConfig,dataType,props);
end
```

```
% Preallocate feature vector
numHops = floor((numSamples-numel(props.Window))/(numel(props.Window) - props.OverlapLength)) + 1
featureVector = coder.nullcopy(zeros(numHops,props.NumFeatures,numChannels,dataType));

% Short-time Fourier transform
Y = stft(x,"Window",props.Window,"OverlapLength",props.OverlapLength,"FFTLength",props.FFTLength
Z = reshape(Y,[],numHops*numChannels);
Zpower = real(Z.*conj(Z));

% Bark spectrum
y = applyFilterBank(config.barkSpectrum.FilterBank, Zpower, dataType);
barkSpectrum = reshape(y,[],numHops,numChannels);
featureVector(:,outputIndex.barkSpectrum,:) = permute(barkSpectrum,[2,1,3]);

% ERB spectrum
y = applyFilterBank(config.erbSpectrum.FilterBank, Zpower, dataType);
erbSpectrum = reshape(y,[],numHops,numChannels);

% Gammatone-frequency cepstral coefficients (GTCC)
gammacc = cepstralCoefficients(erbSpectrum,"NumCoeffs",13,"Rectification","log");
featureVector(:,outputIndex.gtccDelta,:) = audioDelta(gammacc,9);

% Periodicity features
featureVector(:,outputIndex.harmonicRatio,:) = harmonicRatio(x,props.SampleRate,"Window",props.Wi
end

function props = getProps(dataType)
props.Window = cast([0;3.764908042774850471801073581446e-05;0.000150590651897875016373973267036649
props.OverlapLength = cast(256,dataType);
props.SampleRate = cast(44100,dataType);
props.FFTLength = uint16(1024);
props.NumFeatures = uint8(46);
end

function [config, outputIndex] = getConfig(dataType, props)
powerNormalizationFactor = 1/(sum(props.Window)^2);

barkFilterbank = designAuditoryFilterBank(props.SampleRate, ...
    "FrequencyScale","bark", ...
    "FFTLength",props.FFTLength, ...
    "OneSided",true, ...
    "FrequencyRange",[0 22050], ...
    "NumBands",32, ...
    "Normalization","bandwidth", ...
    "FilterBankDesignDomain","linear");
barkFilterbank = barkFilterbank*powerNormalizationFactor;
config.barkSpectrum.FilterBank = cast(barkFilterbank,dataType);

erbFilterbank = coder.const(@feval,'designAuditoryFilterBank',props.SampleRate, ...
    "FrequencyScale","erb", ...
    "FFTLength",props.FFTLength, ...
    "OneSided",true, ...
    "FrequencyRange",[0 22050], ...
    "NumBands",43, ...
    "Normalization","bandwidth");
erbFilterbank = erbFilterbank*powerNormalizationFactor;
config.erbSpectrum.FilterBank = cast(erbFilterbank,dataType);
```

```
outputIndex.barkSpectrum = uint8(1:32);
outputIndex.gtccDelta = uint8(33:45);
outputIndex.harmonicRatio = uint8(46);
end

function y = applyFilterBank(filterBank, Z, dataType, varargin)
if isempty(coder.target)
    y = filterBank * Z;
else
    % Generate optimized C/C++ code for filter bank operation
    [numBands, filterLength] = size(filterBank);
    numChan = size(Z, 2);

    if nargin == 3
        y = zeros(numBands, numChan, dataType);
    else
        y = varargin{1};
    end

    for channel = 1:numChan
        temp = zeros(numBands, 1, dataType);
        for j = 1:filterLength
            temp = temp + Z(j, channel)*filterBank(:, j);
        end
        y(:, channel) = temp;
    end
end
end
```

Calling the generated function is equivalent to calling `extract` on the `audioFeatureExtrator` object. You can replace calls to `extract` with calls to the generated function in your code. Verify the equivalency between the object and the function.

```
a = extract(afe,audioIn);
b = extractAudioFeatures(audioIn);
isequal(a,b)
```

```
ans = logical
   1
```

The generated function contains help text that indicates any requirements on the input. In this example, the only requirement is that the input sample rate should be 44.1 kHz. The help text also contains custom examples. These examples show how to use the function directly in MATLAB and how to generate C/C++ code.

```
help extractAudioFeatures
```

```
 extractAudioFeatures Extract multiple features from batch audio
    featureVector = extractAudioFeatures(audioIn) returns audio features
    extracted from audioIn.

    Parameters of the audioFeatureExtractor used to generate this
    function must be honored when calling this function.
     - Sample rate of the input should be 44100 Hz.
     - Input frame length should be greater than or equal to 512 samples.
```

```
        % EXAMPLE 1: Extract features
          source = dsp.ColoredNoise("SamplesPerFrame",44100);
          for ii = 1:10
              audioIn = source();
              featureArray = extractAudioFeatures(audioIn);
              % ... do something with featureArray ...
          end


        % EXAMPLE 2: Generate code
          targetDataType = "single";
          codegen extractAudioFeatures -args {ones(44100,1,targetDataType)}
          source = dsp.ColoredNoise("SamplesPerFrame",44100, ...
                                    "OutputDataType",targetDataType);
          for ii = 1:10
              audioIn = source();
              featureArray = extractAudioFeatures_mex(audioIn);
              % ... do something with featureArray ...
          end

    See also audioFeatureExtractor, dsp.AsyncBuffer, codegen.
```

Run the first example to see how to use the function to extract features in MATLAB.

```
source = dsp.ColoredNoise("SamplesPerFrame",44100);
for ii = 1:10
    audioIn = source();
    featureArray = extractAudioFeatures(audioIn);
    % ... do something with featureArray ...
end
```

Run the second example to see how to generate a MATLAB executable from the function. Then use the MEX file to extract features while working in MATLAB. MATLAB Coder™ is required to run the following code.

```
targetDataType = "single";
codegen extractAudioFeatures -args {ones(44100,1,targetDataType)}

Code generation successful.

source = dsp.ColoredNoise("SamplesPerFrame",44100, ...
                          "OutputDataType",targetDataType);
for ii = 1:10
    audioIn = source();
    featureArray = extractAudioFeatures_mex(audioIn);
    % ... do something with featureArray ...
end
```

Generate MATLAB® Function for Stream Processing

You can use the `audioFeatureExtractor` object to develop a feature extraction pipeline in MATLAB. The `audioFeatureExtractor` is optimized to extract features from audio signals that contain several windows of data. Typically, audio features are extracted on time scales from 5 ms to 100 ms, depending on the application. When you are ready to deploy your system to a device or to

integrate it into a larger system, you can use `generateMATLABFunction` to create a MATLAB function suitable for C/C++ code generation. Deployed systems are often concerned with minimizing latency. You can set the `IsStreaming` parameter to `true` when calling `generateMATLABFunction` to generate a MATLAB function that is optimized for stream processing. The generated MATLAB function assumes that the input has already been buffered and requires a fixed input frame size. The generated MATLAB function also maintains any required state for you between calls.

Read in an audio file. You will use this audio file to verify the approximate equivalency of the `audioFeatureExtractor` object and the generated MATLAB function.

```
[audioToVerify,fs] = audioread('Counting-16-44p1-mono-15secs.wav');
```

Create an `audioFeatureExtractor` object to extract the mel frequency cepstral coefficients (MFCC), the delta and delta-delta MFCC, the spectral centroid, and the pitch. Extract features from 30 ms windows with 20 ms overlap between windows.

```
afe = audioFeatureExtractor("Window",hann(round(0.03*fs),'periodic'), ...
    "OverlapLength",round(0.02*fs), ...
    "SampleRate",fs, ...
    "mfcc",true, ...
    "mfccDelta",true, ...
    "mfccDeltaDelta",true, ...
    "spectralCentroid",true, ...
    "pitch",true)
```

```
afe =
  audioFeatureExtractor with properties:

   Properties
                     Window: [1323x1 double]
              OverlapLength: 882
                 SampleRate: 44100
                  FFTLength: []
    SpectralDescriptorInput: 'linearSpectrum'
        FeatureVectorLength: 41

   Enabled Features
     mfcc, mfccDelta, mfccDeltaDelta, spectralCentroid, pitch

   Disabled Features
     linearSpectrum, melSpectrum, barkSpectrum, erbSpectrum, gtcc, gtccDelta
     gtccDeltaDelta, spectralCrest, spectralDecrease, spectralEntropy, spectralFlatness, spectra
     spectralKurtosis, spectralRolloffPoint, spectralSkewness, spectralSlope, spectralSpread, ha
     zerocrossrate, shortTimeEnergy


   To extract a feature, set the corresponding property to true.
   For example, obj.mfcc = true, adds mfcc to the list of enabled features.
```

Call `generateMATLABFunction` on the object and specify a name and the full path for the generated MATLAB function. Set `IsStreaming` to `true` to generate a MATLAB function optimized for stream processing.

```
filename = fullfile(tempdir,"extractAudioFeatures");
generateMATLABFunction(afe,filename,'IsStreaming',true);
```

The generated function is saved to the `tempdir` folder. Because the `mfccDelta` and `mfccDeltaDelta` features require state, the generated function includes the ability to reset states using the optional name-value pair `"Reset"` and either `true` or `false`. If you generate a function that does not require state, the `"Reset"` parameter is not included in the generated function.

```
cd(tempdir)
type extractAudioFeatures
```

```
function featureVector = extractAudioFeatures(x, varargin)
%extractAudioFeatures Extract multiple features from streaming audio
%   featureVector = extractAudioFeatures(audioIn) returns audio features
%   extracted from audioIn.
%
%   featureVector = extractAudioFeatures(audioIn,"Reset",TF) returns feature extractors
%   to their initial conditions before extracting features.
%
%   Parameters of the audioFeatureExtractor used to generate this
%   function must be honored when calling this function.
%    - Sample rate of the input should be 44100 Hz.
%    - Frame length of the input should be 1323 samples.
%    - Successive frames of the input should be overlapped by
%      882 samples before calling extractAudioFeatures.
%
%
%      % EXAMPLE 1: Extract features
%        source = dsp.ColoredNoise();
%        inputBuffer = dsp.AsyncBuffer;
%        for ii = 1:10
%            audioIn = source();
%            write(inputBuffer,audioIn);
%            while inputBuffer.NumUnreadSamples > 441
%                x = read(inputBuffer,1323,882);
%                featureVector = extractAudioFeatures(x);
%                % ... do something with featureVector ...
%            end
%         end
%
%
%      % EXAMPLE 2: Extract features from speech regions only
%        [audioIn,fs] = audioread("Counting-16-44p1-mono-15secs.wav");
%        audioIn = resample(audioIn,44100,fs);
%        source = dsp.AsyncBuffer(size(audioIn,1));
%        write(source,audioIn);
%        TF = false;
%        while source.NumUnreadSamples > 441
%            x = read(source,1323,882);
%            isSilence = var(x) < 0.01;
%            if ~isSilence
%                featureVector = extractAudioFeatures(x,"Reset",TF);
%                TF = false;
%            else
%                TF = true;
%            end
%            % ... do something with featureVector ...
%        end
%
%
%      % EXAMPLE 3: Generate code that does not use reset
```

```
%        targetDataType = "single";
%        codegen extractAudioFeatures -args {ones(1323,1,targetDataType)}
%        source = dsp.ColoredNoise('OutputDataType',targetDataType);
%        inputBuffer = dsp.AsyncBuffer;
%        for ii = 1:10
%            audioIn = source();
%            write(inputBuffer,audioIn);
%            while inputBuffer.NumUnreadSamples > 441
%                x = read(inputBuffer,1323,882);
%                featureVector = extractAudioFeatures_mex(x);
%                % ... do something with featureVector ...
%            end
%         end
%
%
%      % EXAMPLE 4: Generate code that uses reset
%        targetDataType = "single";
%        codegen extractAudioFeatures -args {ones(1323,1,targetDataType),'Reset',true}
%        [audioIn,fs] = audioread("Counting-16-44p1-mono-15secs.wav");
%        audioIn = resample(audioIn,44100,fs);
%        source = dsp.AsyncBuffer(size(audioIn,1));
%        write(source,cast(audioIn,targetDataType));
%        TF = false;
%        while source.NumUnreadSamples > 441
%            x = read(source,1323,882);
%            isSilence = var(x) < 0.01;
%            if ~isSilence
%                featureVector = extractAudioFeatures_mex(x,'Reset',TF);
%                TF = false;
%            else
%                TF = true;
%            end
%            % ... do something with featureVector ...
%        end
%
%   See also audioFeatureExtractor, dsp.AsyncBuffer, codegen.

%   Generated by audioFeatureExtractor on 03-Mar-2023 22:46:43 UTC-05:00
%#codegen

dataType = underlyingType(x);
numChannels = size(x,2);

props = coder.const(getProps(dataType));

persistent config outputIndex state
if isempty(outputIndex)
    [config, outputIndex] = coder.const(@getConfig,dataType,props);
    state = getState(dataType,numChannels);
else
    assert(state.NumChannels == numChannels)
end
if nargin==3
    if strcmpi(varargin{1},"Reset") && varargin{2}
        state = reset(state);
    end
end
```

```
% Preallocate feature vector
featureVector = coder.nullcopy(zeros(props.NumFeatures,numChannels,dataType));

% Fourier transform
Y = fft(bsxfun(@times,x,props.Window),props.FFTLength);
Z = Y(config.OneSidedSpectrumBins,:);
Zpower = real(Z.*conj(Z));

% Linear spectrum
linearSpectrum = Zpower(config.linearSpectrum.FrequencyBins,:)*config.linearSpectrum.Normalizatio
linearSpectrum(1,:) = 0.5*linearSpectrum(1,:);
linearSpectrum = reshape(linearSpectrum,[],1,numChannels);

% Mel spectrum
y = applyFilterBank(config.melSpectrum.FilterBank, Zpower, dataType);
melSpectrum = reshape(y,[],1,numChannels);

% Mel-frequency cepstral coefficients (MFCC)
melcc = cepstralCoefficients(melSpectrum,"NumCoeffs",13,"Rectification","log");
featureVector(outputIndex.mfcc,:) = melcc;
[melccDelta,state.mfccDelta] = audioDelta(melcc,9,state.mfccDelta);
featureVector(outputIndex.mfccDelta,:) = melccDelta;
[featureVector(outputIndex.mfccDeltaDelta,:),state.mfccDeltaDelta] = audioDelta(melccDelta,9,sta

% Spectral descriptors
featureVector(outputIndex.spectralCentroid,:) = spectralCentroid(linearSpectrum,config.SpectralDe

% Periodicity features
featureVector(outputIndex.pitch,:) = pitch(x,props.SampleRate,"WindowLength",numel(props.Window)
end

function props = getProps(dataType)
props.Window = cast([0;5.638703266919176115834488882683 2e-06;2.255468588774345306546820211 2243e-0
props.OverlapLength = cast(882,dataType);
props.SampleRate = cast(44100,dataType);
props.FFTLength = uint16(1323);
props.NumFeatures = uint8(41);
end

function [config, outputIndex] = getConfig(dataType, props)
powerNormalizationFactor = 1/(sum(props.Window)^2);

config.OneSidedSpectrumBins = uint16(1:662);

linearSpectrumFrequencyBins = 1:662;
config.linearSpectrum.FrequencyBins = uint16(linearSpectrumFrequencyBins);
config.linearSpectrum.NormalizationFactor = cast(2*powerNormalizationFactor,dataType);

melFilterbank = designAuditoryFilterBank(props.SampleRate, ...
    "FrequencyScale","mel", ...
    "FFTLength",props.FFTLength, ...
    "OneSided",true, ...
    "FrequencyRange",[0 22050], ...
    "NumBands",32, ...
    "Normalization","bandwidth", ...
    "FilterBankDesignDomain","linear");
melFilterbank = melFilterbank*powerNormalizationFactor;
config.melSpectrum.FilterBank = cast(melFilterbank,dataType);
```

```
FFTLength = cast(props.FFTLength,'like',props.SampleRate);
w = (props.SampleRate/FFTLength)*(linearSpectrumFrequencyBins-1);
w(end) = props.SampleRate*(FFTLength-1)/(2*FFTLength);
config.SpectralDescriptorInput.FrequencyVector = cast(w(:),dataType);

outputIndex.mfcc = uint8(1:13);
outputIndex.mfccDelta = uint8(14:26);
outputIndex.mfccDeltaDelta = uint8(27:39);
outputIndex.spectralCentroid = uint8(40);
outputIndex.pitch = uint8(41);
end

function state = getState(dataType, numChannels)
state.NumChannels = numChannels;
state.mfccDelta = zeros(8,13,numChannels,dataType);
state.mfccDeltaDelta = zeros(8,13,numChannels,dataType);
end

function state = reset(state)
state.mfccDelta(:,:,:) = 0;
state.mfccDeltaDelta(:,:,:) = 0;
end

function y = applyFilterBank(filterBank, Z, dataType, varargin)
if isempty(coder.target)
    y = filterBank * Z;
else
    % Generate optimized C/C++ code for filter bank operation
    [numBands, filterLength] = size(filterBank);
    numChan = size(Z, 2);

    if nargin == 3
        y = zeros(numBands, numChan, dataType);
    else
        y = varargin{1};
    end

    for channel = 1:numChan
        temp = zeros(numBands, 1, dataType);
        for j = 1:filterLength
            temp = temp + Z(j, channel)*filterBank(:, j);
        end
        y(:, channel) = temp;
    end
end
end
```

The generated function contains help text that indicates any requirements on the input. In this example, the sample rate of the input should be 44.1 kHz, the frame input to the function should be 1323 samples, and successive frames should be overlapped by 882 samples before calling the function. The differences between the `audioFeatureExtractor` object and the function are described in more detail in Approximate Equivalency Between audioFeatureExtractor and Generated Function on page 4-19.

```
help extractAudioFeatures
```

```
extractAudioFeatures Extract multiple features from streaming audio
   featureVector = extractAudioFeatures(audioIn) returns audio features
   extracted from audioIn.

   featureVector = extractAudioFeatures(audioIn,"Reset",TF) returns feature extractors
   to their initial conditions before extracting features.

   Parameters of the audioFeatureExtractor used to generate this
   function must be honored when calling this function.
    - Sample rate of the input should be 44100 Hz.
    - Frame length of the input should be 1323 samples.
    - Successive frames of the input should be overlapped by
      882 samples before calling extractAudioFeatures.


      % EXAMPLE 1: Extract features
        source = dsp.ColoredNoise();
        inputBuffer = dsp.AsyncBuffer;
        for ii = 1:10
            audioIn = source();
            write(inputBuffer,audioIn);
            while inputBuffer.NumUnreadSamples > 441
                x = read(inputBuffer,1323,882);
                featureVector = extractAudioFeatures(x);
                % ... do something with featureVector ...
            end
         end


      % EXAMPLE 2: Extract features from speech regions only
        [audioIn,fs] = audioread("Counting-16-44p1-mono-15secs.wav");
        audioIn = resample(audioIn,44100,fs);
        source = dsp.AsyncBuffer(size(audioIn,1));
        write(source,audioIn);
        TF = false;
        while source.NumUnreadSamples > 441
            x = read(source,1323,882);
            isSilence = var(x) < 0.01;
            if ~isSilence
                featureVector = extractAudioFeatures(x,"Reset",TF);
                TF = false;
            else
                TF = true;
            end
            % ... do something with featureVector ...
        end


      % EXAMPLE 3: Generate code that does not use reset
        targetDataType = "single";
        codegen extractAudioFeatures -args {ones(1323,1,targetDataType)}
        source = dsp.ColoredNoise('OutputDataType',targetDataType);
        inputBuffer = dsp.AsyncBuffer;
        for ii = 1:10
            audioIn = source();
            write(inputBuffer,audioIn);
            while inputBuffer.NumUnreadSamples > 441
                x = read(inputBuffer,1323,882);
```

```
                featureVector = extractAudioFeatures_mex(x);
                % ... do something with featureVector ...
            end
        end


    % EXAMPLE 4: Generate code that uses reset
      targetDataType = "single";
      codegen extractAudioFeatures -args {ones(1323,1,targetDataType),'Reset',true}
      [audioIn,fs] = audioread("Counting-16-44p1-mono-15secs.wav");
      audioIn = resample(audioIn,44100,fs);
      source = dsp.AsyncBuffer(size(audioIn,1));
      write(source,cast(audioIn,targetDataType));
      TF = false;
      while source.NumUnreadSamples > 441
          x = read(source,1323,882);
          isSilence = var(x) < 0.01;
          if ~isSilence
              featureVector = extractAudioFeatures_mex(x,'Reset',TF);
              TF = false;
          else
              TF = true;
          end
          % ... do something with featureVector ...
      end

See also audioFeatureExtractor, dsp.AsyncBuffer, codegen.
```

The examples in the help show how to use the function directly in MATLAB and how to generate C/C++ code. Run the first example to see how to use the function to extract features in MATLAB.

```
source = dsp.ColoredNoise();
inputBuffer = dsp.AsyncBuffer;
for ii = 1:10
    audioIn = source();
    write(inputBuffer,audioIn);
    while inputBuffer.NumUnreadSamples > 441
        x = read(inputBuffer,1323,882);
        featureVector = extractAudioFeatures(x);
        % ... do something with featureVector ...
    end
end
```

Run the second example to see how to extract features in MATLAB while using the optional `"Reset"` name-value pair. The `Reset` name-value pair enables you to reset states on the function. For example, if you are only concerned with extracting features from regions of voiced speech and want to avoid the overhead of extracting features constantly, you can use the `"Reset"` parameter to avoid bleeding feature information between regions.

```
[audioIn,fs] = audioread("Counting-16-44p1-mono-15secs.wav");
source = dsp.AsyncBuffer(size(audioIn,1));
write(source,audioIn);
TF = false;
while source.NumUnreadSamples > 441
    x = read(source,1323,882);
    isSilence = var(x) < 0.01;
    if ~isSilence
        featureVector = extractAudioFeatures(x,"Reset",TF);
```

```
        TF = false;
    else
        TF = true;
    end
    % ... do something with featureVector ...
end
```

Run the third example to see how to generate code that does not include the ability to reset state. When generating code that does not use the `'Reset'` parameter, only specify a prototype for the audio input argument. The following code requires MATLAB Coder™.

```
targetDataType = "single";
codegen extractAudioFeatures -args {ones(1323,1,targetDataType)}
```

```
Code generation successful.
```

```
source = dsp.ColoredNoise('OutputDataType',targetDataType);
inputBuffer = dsp.AsyncBuffer;
for ii = 1:10
    audioIn = source();
    write(inputBuffer,audioIn);
    while inputBuffer.NumUnreadSamples > 441
        x = read(inputBuffer,1323,882);
        featureVector = extractAudioFeatures_mex(x);
        % ... do something with featureVector ...
    end
end
```

Run the fourth example to see how to generate code that can reset state. When generating code that uses the `'Reset'` parameter, specify prototype input arguments for the full function signature. The following code requires MATLAB Coder™.

```
targetDataType = "single";
codegen extractAudioFeatures -args {ones(1323,1,targetDataType),'Reset',true}
```

```
Code generation successful.
```

```
[audioIn,fs] = audioread("Counting-16-44p1-mono-15secs.wav");
source = dsp.AsyncBuffer(size(audioIn,1));
write(source,cast(audioIn,targetDataType));
TF = false;
while source.NumUnreadSamples > 441
    x = read(source,1323,882);
    isSilence = var(x) < 0.01;
    if ~isSilence
        featureVector = extractAudioFeatures_mex(x,'Reset',TF);
        TF = false;
    else
        TF = true;
    end
    % ... do something with featureVector ...
end
```

Approximate Equivalency Between `audioFeatureExtractor` and Generated Function

When you call `extract` using `audioFeatureExtractor`, the input is buffered internally prior to feature extraction. The output from `extract` is an *L*-by-*M*-by-*N* array, where *L* is the number of feature vectors and is equal to the number of analysis windows. *M* is the number of features extracted per analysis window. *N* is the number of channels.

```
featuresA = extract(afe,audioToVerify);
[L,M,N] = size(featuresA)
```

```
L = 1551
```

```
M = 41
```

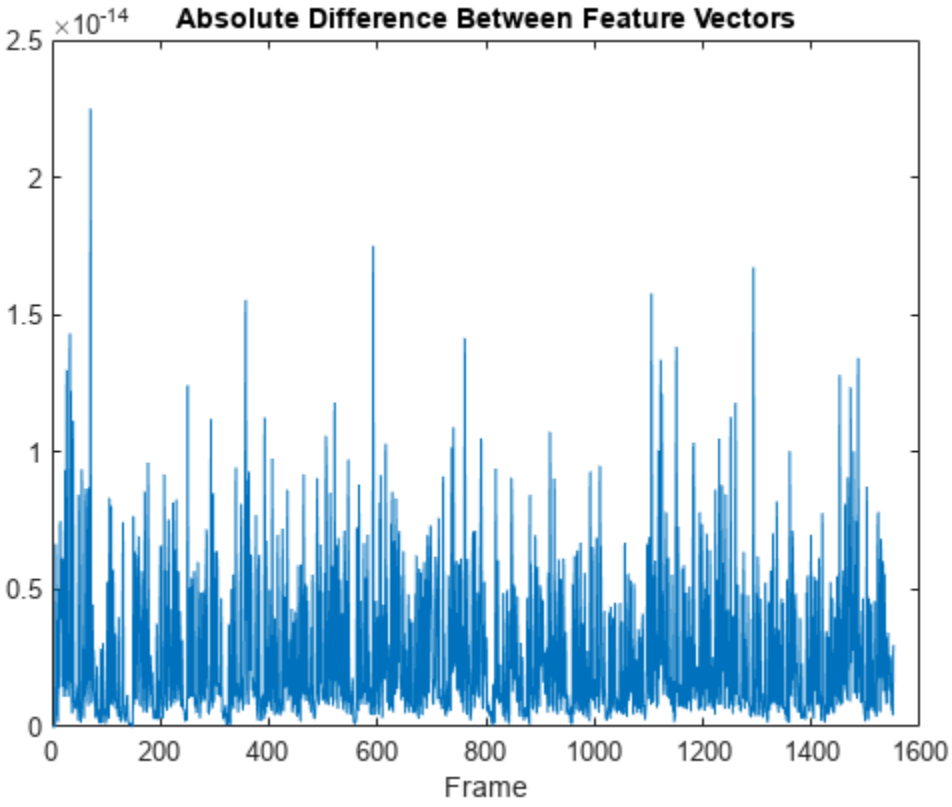
```
N = 1
```

When you call the generated function, extractAudioFeatures, the input should represent a single frame of audio data. The output from the generated function is an *M*-by-*N* matrix, where *M* is the number of features extracted and *N* is the number of channels. Use the generated function to extract features from the audio signal `audioIn`. Use the `dsp.AsyncBuffer` object to buffer the input into the required frame length and overlap length prior to calling extractAudioFeatures. Reshape the extracted feature vectors to match the orientation output from `audioFeatureExtractor`.

```
frameLength = 1323;
overlapLength = 882;
hopLength = frameLength - overlapLength;

featuresB = zeros(L,M,N);

buff = dsp.AsyncBuffer('Capacity',numel(audioToVerify));
write(buff,audioToVerify);

hop = 1;
while buff.NumUnreadSamples > hopLength
    if hop==1
        x = read(buff,frameLength);
        features = extractAudioFeatures(x,'Reset',true);
    else
        x = read(buff,frameLength,overlapLength);
        features = extractAudioFeatures(x);
    end
    featuresB(hop,:,:) = reshape(features,[1,M,N]);
    hop = hop + 1;
end
```

Visualize the difference between the output from `audioFeatureExtractor` and the generated function. The differences between frames are less than `1.8e-14` and are due to the different code paths being optimized for batch versus stream processing.

```
differenceBetweenFrames = sum(abs(featuresA-featuresB),2);
plot(differenceBetweenFrames)
xlabel('Frame')
title('Absolute Difference Between Feature Vectors')
```



Input Arguments

`afe` — Input object
`audioFeatureExtractor` object

Input object, specified as an `audioFeatureExtractor` object.

`fileName` — File name
character vector | string scalar

File name where the generated function is saved, specified as a character vector or string scalar.

Data Types: `char` | `string`

`TF` — Flag to specify if function is for streaming
`false` (default) | `true`

Flag to specify if generated function is intended for stream processing, specified as `true` or `false`.

Data Types: `logical`

Version History

Introduced in R2020b

See Also

`codegen` | `dsp.AsyncBuffer` | `audioFeatureExtractor` | `vggishEmbeddings`

setExtractorParameters

Set nondefault parameter values for individual feature extractors

Syntax

```
setExtractorParameters(aFE,featureName,params)
setExtractorParameters(aFE,featureName)
```

Description

`setExtractorParameters(aFE,featureName,params)` specifies parameters used to extract `featureName`.

`setExtractorParameters(aFE,featureName)` returns the parameters used to extract `featureName` to default values.

Examples

Extract Pitch Using the LHS Method

Read in an audio signal.

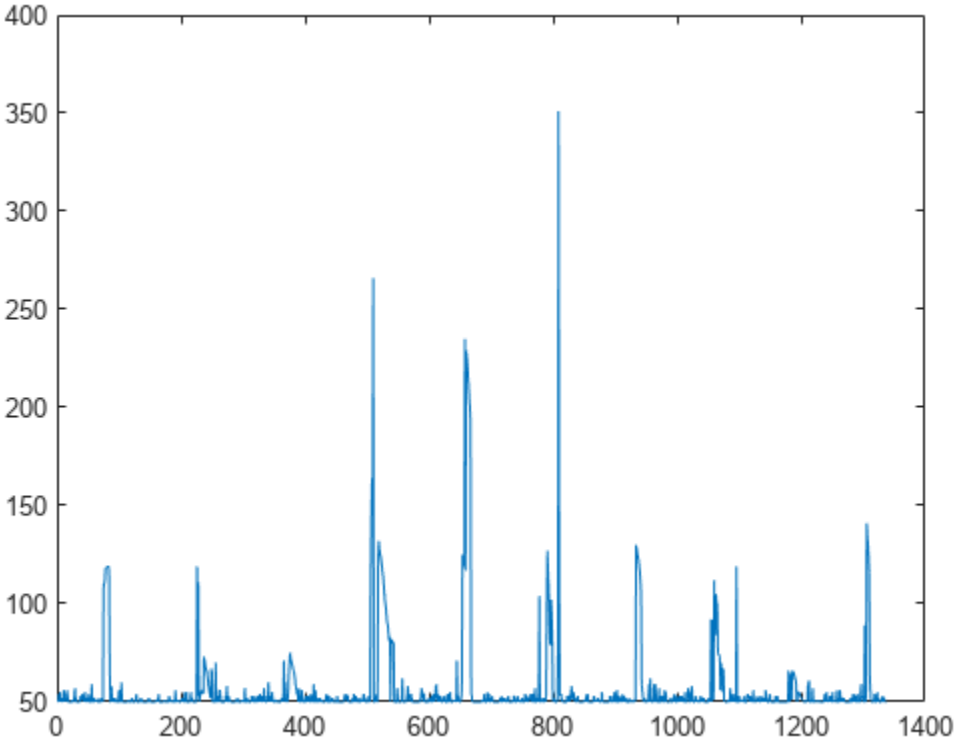
```
[audioIn,fs] = audioread("Counting-16-44p1-mono-15secs.wav");
```

Create an `audioFeatureExtractor` object to extract pitch. Set the method of pitch extraction to `"LHS"`.

```
aFE = audioFeatureExtractor(SampleRate=fs,pitch=true);
setExtractorParameters(aFE,"pitch",Method="LHS")
```

Call `extract` and plot the results.

```
f0 = extract(aFE,audioIn);
plot(f0)
```



Modify Spectral Rolloff Threshold and Mel Spectrum Parameters

Read in an audio signal.

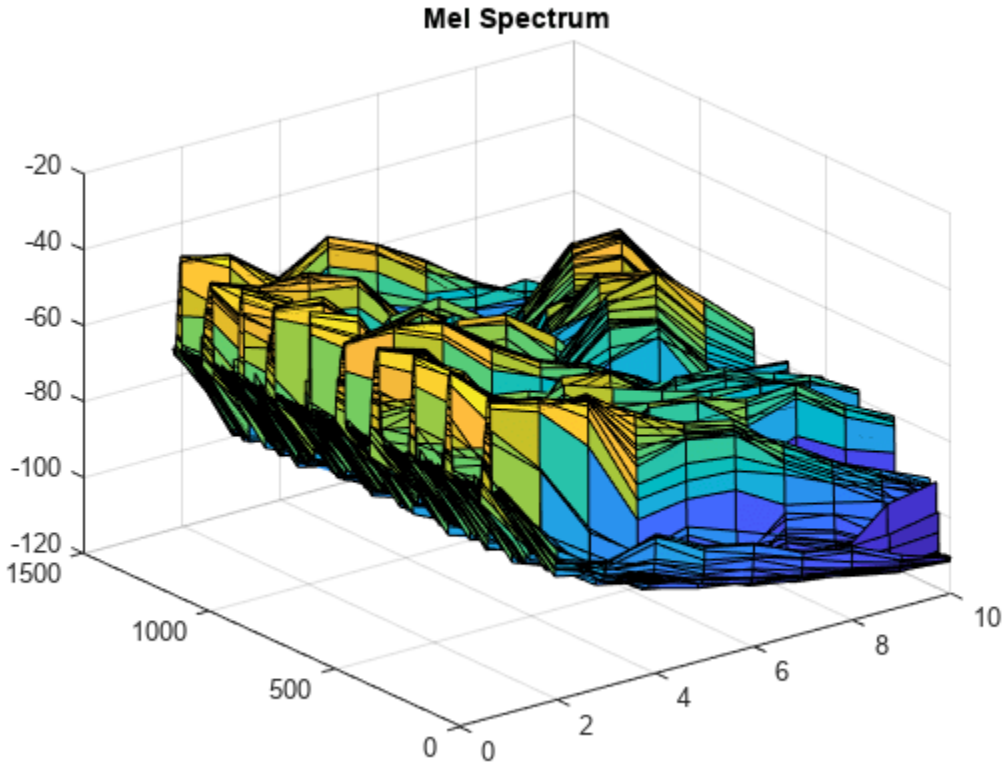
```
[audioIn,fs] = audioread("Counting-16-44p1-mono-15secs.wav");
```

Create an `audioFeatureExtractor` object to extract the `melSpectrum` and `spectralRolloffPoint`. Specify ten bands for the mel spectrum and set the threshold for the rolloff point to 50% of the total energy.

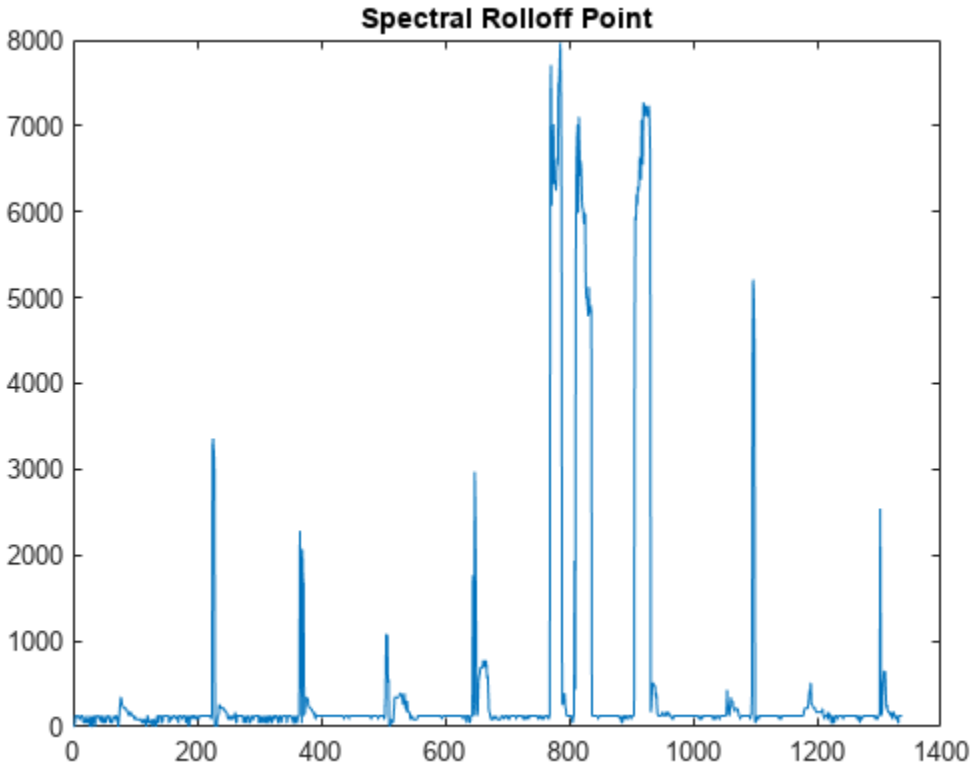
```
aFE = audioFeatureExtractor(SampleRate=fs,melSpectrum=true,spectralRolloffPoint=true);
setExtractorParameters(aFE,"melSpectrum",NumBands=10)
setExtractorParameters(aFE,"spectralRolloffPoint",Threshold=0.5)
```

Call `extract` and plot the results.

```
features = extract(aFE,audioIn);
idx = info(aFE);

surf(10*log10(features(:,idx.melSpectrum)))
title("Mel Spectrum")
```




```
plot(features(:,idx.spectralRolloffPoint))
title("Spectral Rolloff Point")
```

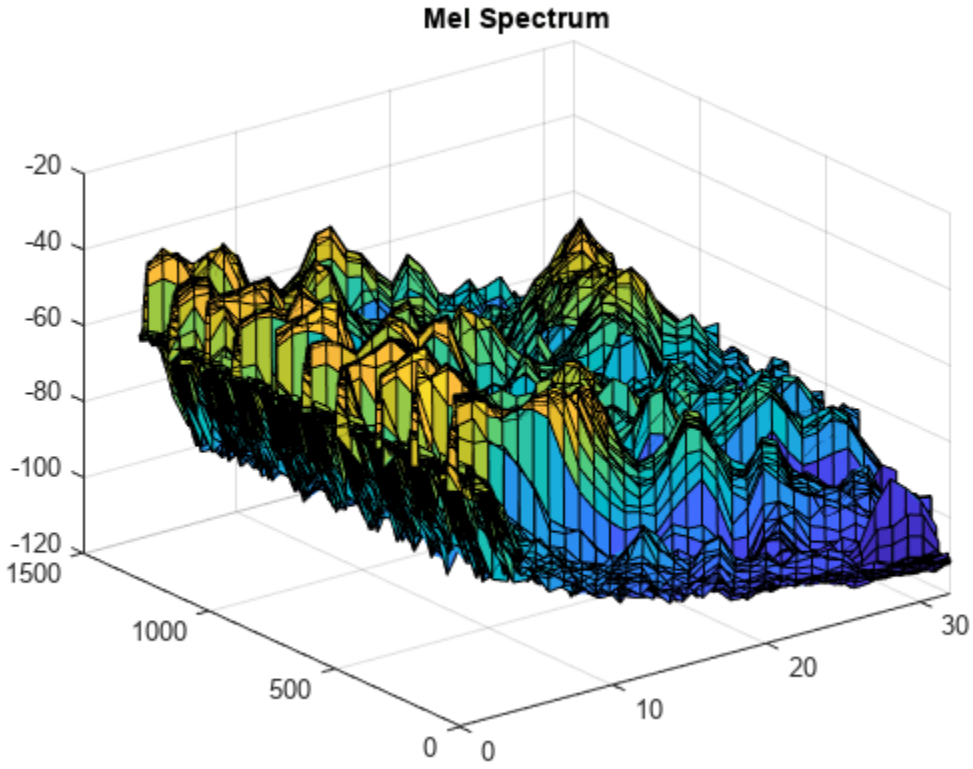


To return individual audio feature extractors to their default values, call `setExtractorParameters` without specifying any parameters to set.

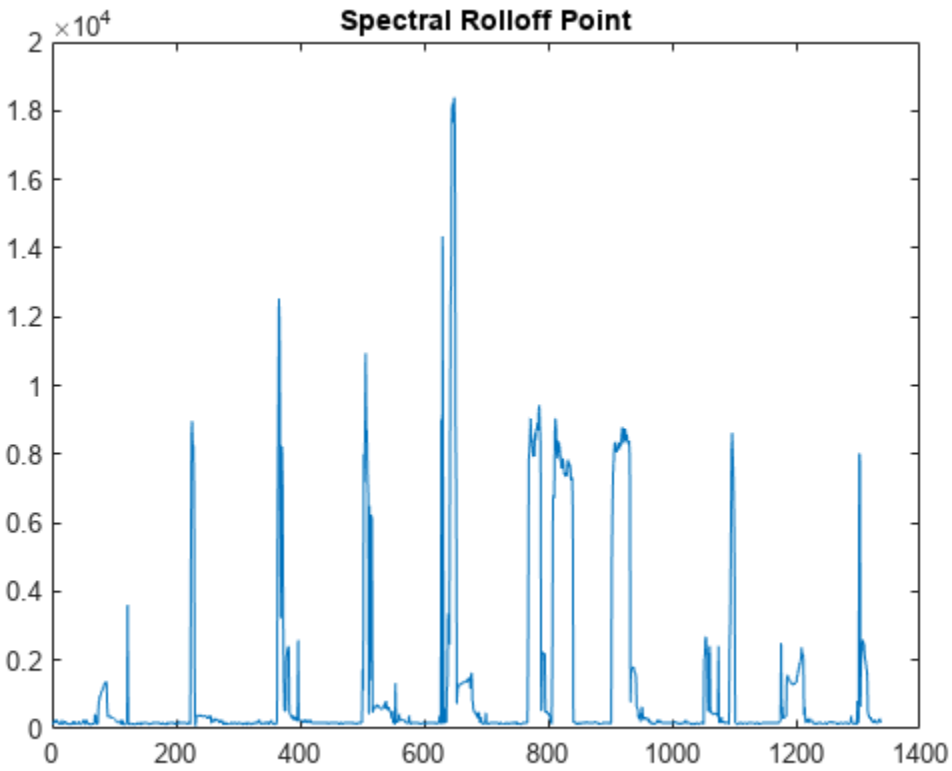
```
setExtractorParameters(aFE,"melSpectrum")
setExtractorParameters(aFE,"spectralRolloffPoint")
```

Call `extract` and plot the results.

```
features = extract(aFE,audioIn);
idx = info(aFE);

surf(10*log10(features(:,idx.melSpectrum)))
title("Mel Spectrum")
```




```
plot(features(:,idx.spectralRolloffPoint))
title("Spectral Rolloff Point")
```



Input Arguments

aFE — Input object
`audioFeatureExtractor` object

Input object, specified as an `audioFeatureExtractor` object.

`featureName` — Name of feature extractor
character array | string

Name of feature extractor, specified as a character array or string.

Data Types: `char` | `string`

`params` — Parameters to set
name-value arguments | struct

Parameters to set, specified as name-value arguments or a struct.

Version History

Introduced in R2022a

See Also

Objects
`audioFeatureExtractor`

setExtractorParams

(To be removed) Set nondefault parameter values for individual feature extractors

Note The `setExtractorParams` function will be removed in a future release. Use `setExtractorParameters` instead. For more information, see “Compatibility Considerations”.

Syntax

```
setExtractorParams(aFE,featureName,params)
setExtractorParams(aFE,featureName)
```

Description

`setExtractorParams(aFE,featureName,params)` specifies parameters used to extract `featureName`.

`setExtractorParams(aFE,featureName)` returns the parameters used to extract `featureName` to default values.

Examples

Extract Pitch Using the LHS Method

Read in an audio signal.

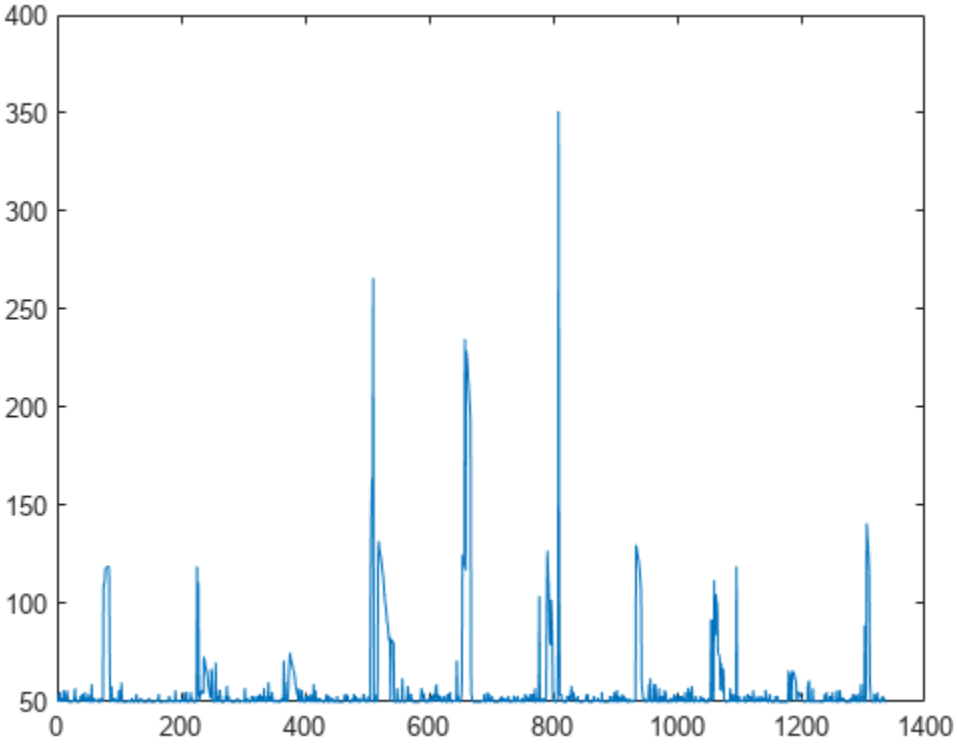
```
[audioIn,fs] = audioread("Counting-16-44p1-mono-15secs.wav");
```

Create an `audioFeatureExtractor` object to extract pitch. Set the method of pitch extraction to `"LHS"`.

```
aFE = audioFeatureExtractor(SampleRate=fs,pitch=true);
setExtractorParameters(aFE,"pitch",Method="LHS")
```

Call `extract` and plot the results.

```
f0 = extract(aFE,audioIn);
plot(f0)
```



Modify Spectral Rolloff Threshold and Mel Spectrum Parameters

Read in an audio signal.

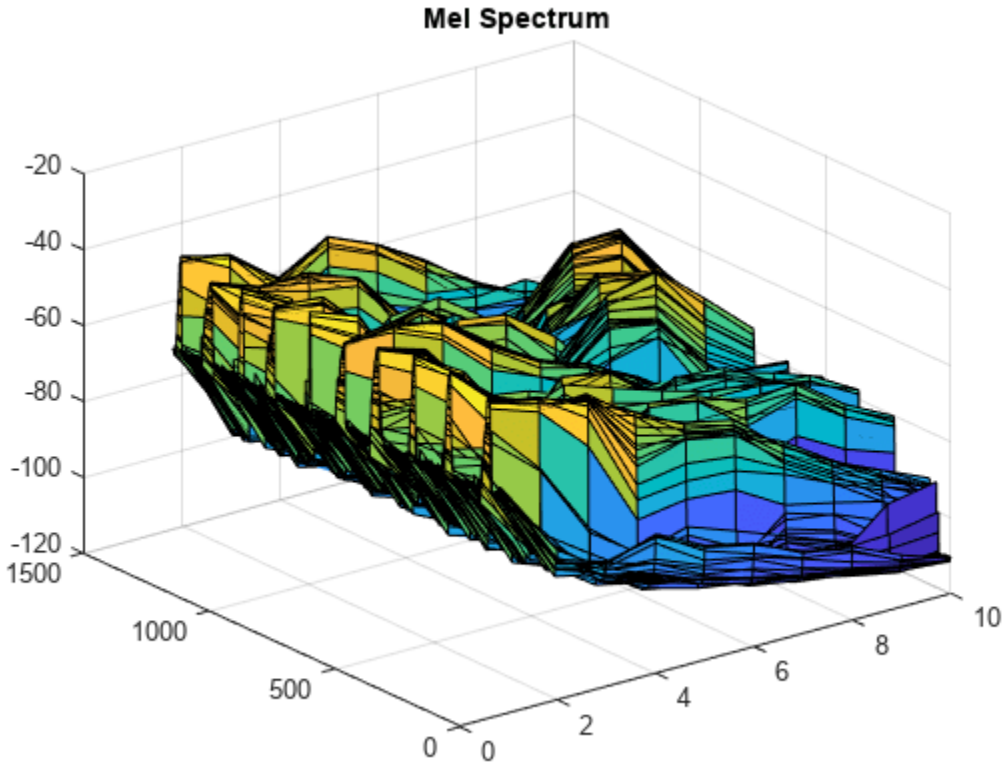
```
[audioIn,fs] = audioread("Counting-16-44p1-mono-15secs.wav");
```

Create an `audioFeatureExtractor` object to extract the `melSpectrum` and `spectralRolloffPoint`. Specify ten bands for the mel spectrum and set the threshold for the rolloff point to 50% of the total energy.

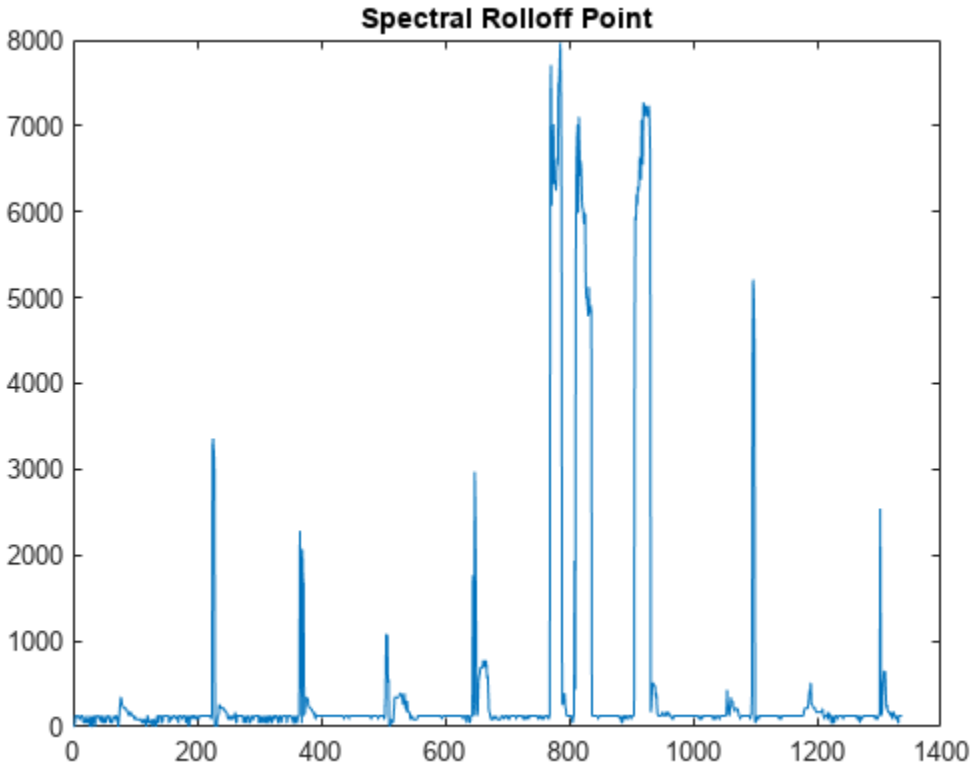
```
aFE = audioFeatureExtractor(SampleRate=fs,melSpectrum=true,spectralRolloffPoint=true);
setExtractorParameters(aFE,"melSpectrum",NumBands=10)
setExtractorParameters(aFE,"spectralRolloffPoint",Threshold=0.5)
```

Call `extract` and plot the results.

```
features = extract(aFE,audioIn);
idx = info(aFE);

surf(10*log10(features(:,idx.melSpectrum)))
title("Mel Spectrum")
```




```
plot(features(:,idx.spectralRolloffPoint))
title("Spectral Rolloff Point")
```

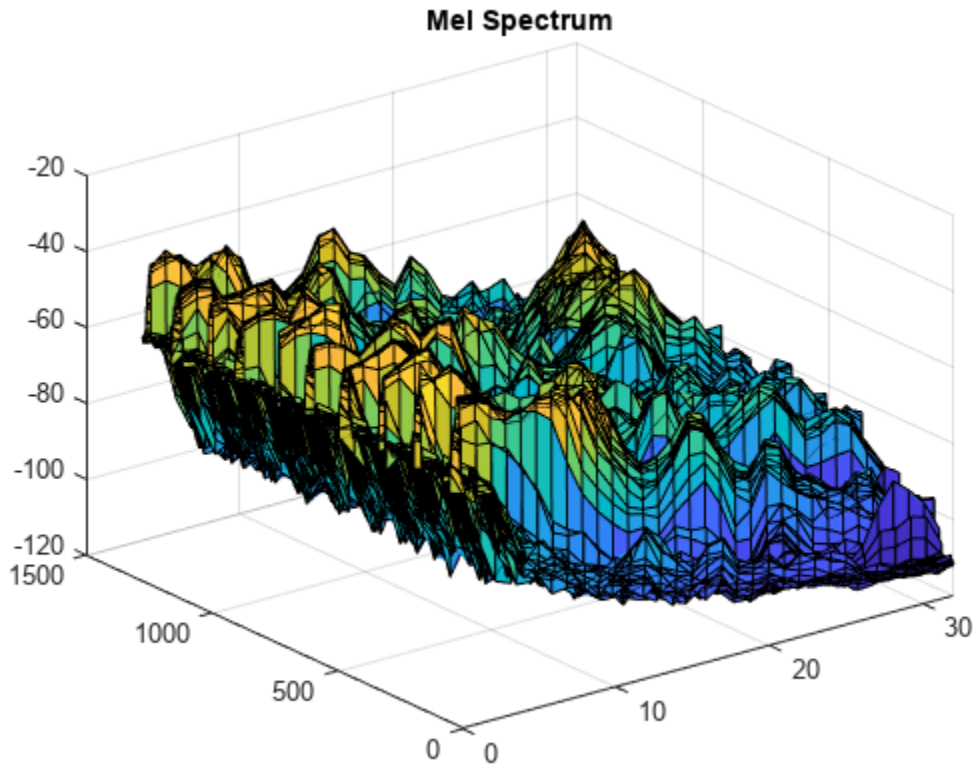


To return individual audio feature extractors to their default values, call `setExtractorParameters` without specifying any parameters to set.

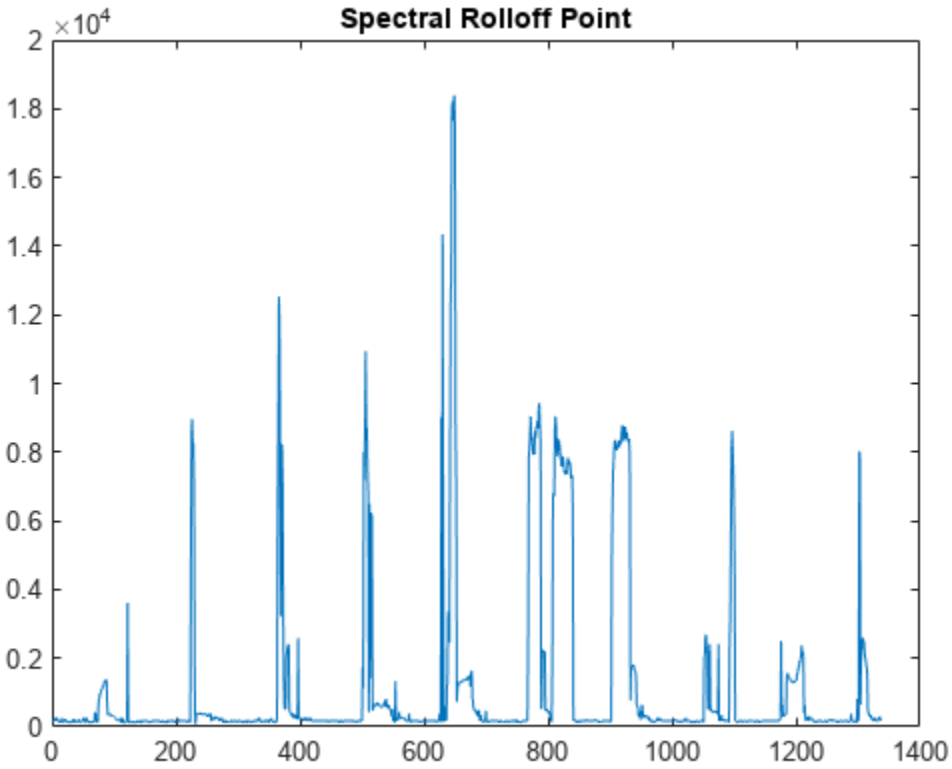
```
setExtractorParameters(aFE,"melSpectrum")
setExtractorParameters(aFE,"spectralRolloffPoint")
```

Call `extract` and plot the results.

```
features = extract(aFE,audioIn);
idx = info(aFE);

surf(10*log10(features(:,idx.melSpectrum)))
title("Mel Spectrum")
```




```
plot(features(:,idx.spectralRolloffPoint))
title("Spectral Rolloff Point")
```



Input Arguments

aFE — Input object
`audioFeatureExtractor` object

`audioFeatureExtractor` object.

`featureName` — Name of feature extractor
character array | string

Name of feature extractor, specified as a character array or string.

Data Types: `char` | `string`

`params` — Parameters to set
comma-separated name-value pairs | struct

Parameters to set, specified as comma-separated name-value pairs or as a struct.

Version History

Introduced in R2019b

R2022a: `setExtractorParams` will be removed
Not recommended starting in R2022a

The `setExtractorParams` function will be removed in a future release. Use `setExtractorParameters` instead. Existing calls to `setExtractorParams` continue to run.

See Also

Objects
`audioFeatureExtractor`

Functions
`setExtractorParameters`

info

Output mapping and individual feature extractor parameters

Syntax

```
idx = info(aFE)
idx = info(aFE,"all")
[idx,params] = info( ___ )
```

Description

`idx = info(aFE)` returns a struct with field names corresponding to enabled feature extractors. The field values correspond to the column indices that the extracted features occupy in the output from `extract`.

`idx = info(aFE,"all")` returns a struct with field names corresponding to all available feature extractors. If the feature extractor is disabled, the field value is empty.

`[idx,params] = info( ___ )` returns a second struct, `params`. The field names of `params` correspond to the feature extractors with settable parameters. If the `"all"` flag is specified, `params` contains all feature extractors with settable parameters. If the `"all"` flag is not specified, `params` contains only the enabled feature extractors with settable parameters. You can set parameters using `setExtractorParameters`.

Examples

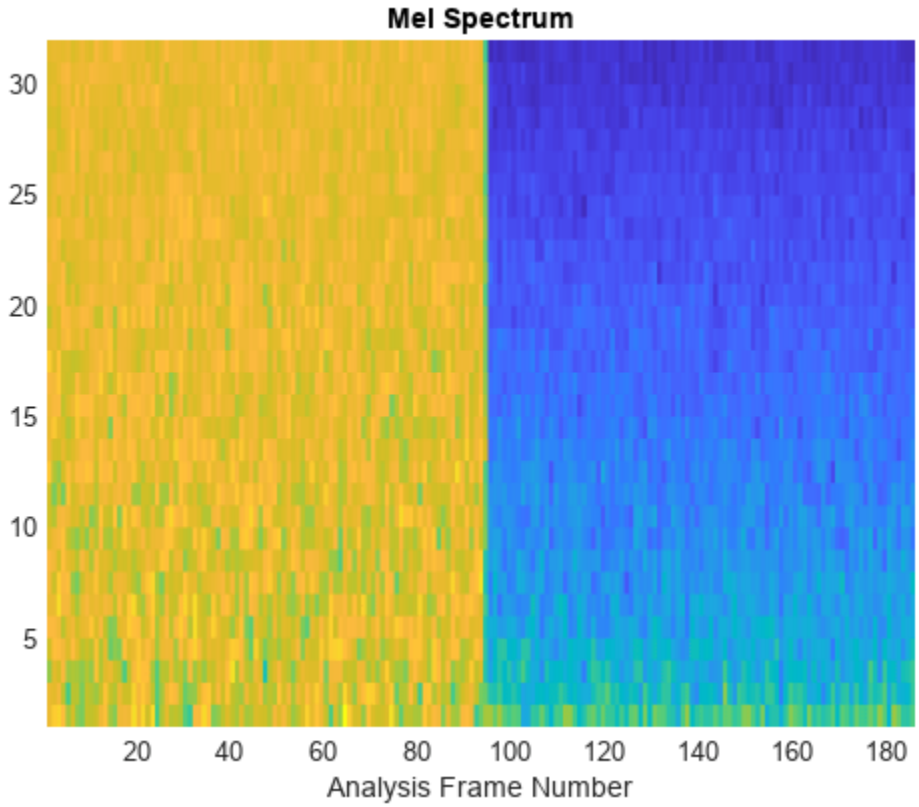
Interpret Output from `extract`

Extract the mel spectrum, mel spectral centroid, and mel spectral skewness from concatenated white and pink noise.

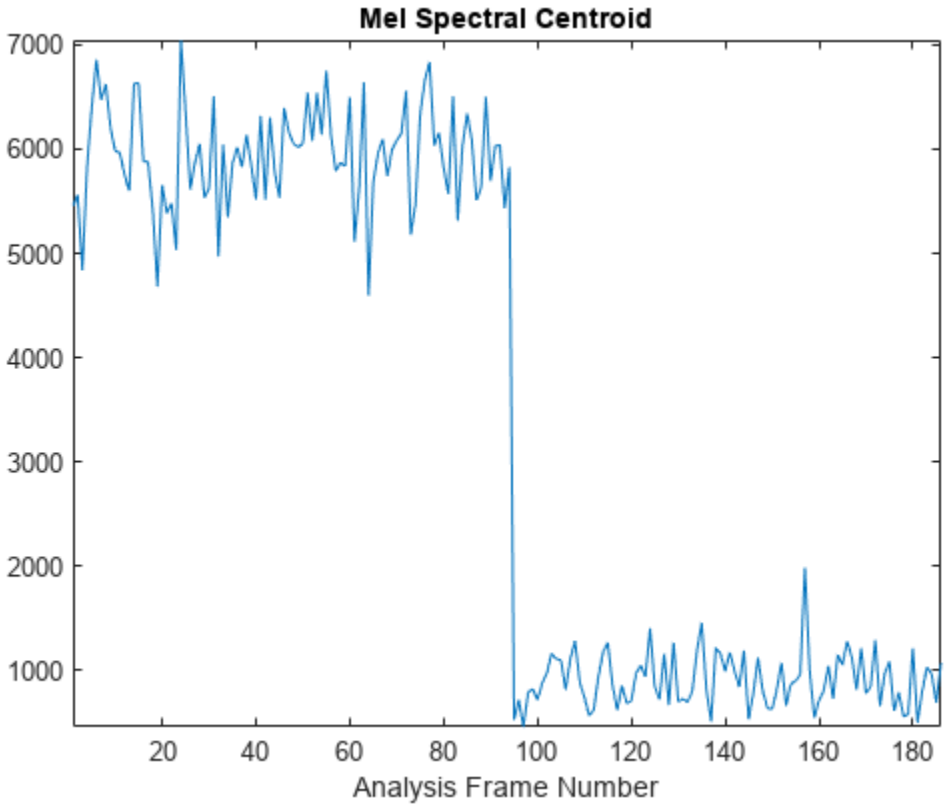
```
fs = 48e3;
aFE = audioFeatureExtractor("SampleRate",fs, ...
    "melSpectrum",true, ...
    "SpectralDescriptorInput","melSpectrum", ...
    "spectralCentroid",true, ...
    "spectralSkewness",true);

features = extract(aFE,[2*rand(fs,1)-1;pinknoise(fs,1)]);
```

Use `info` to determine which columns of the output correspond to which feature. Plot the features separately.

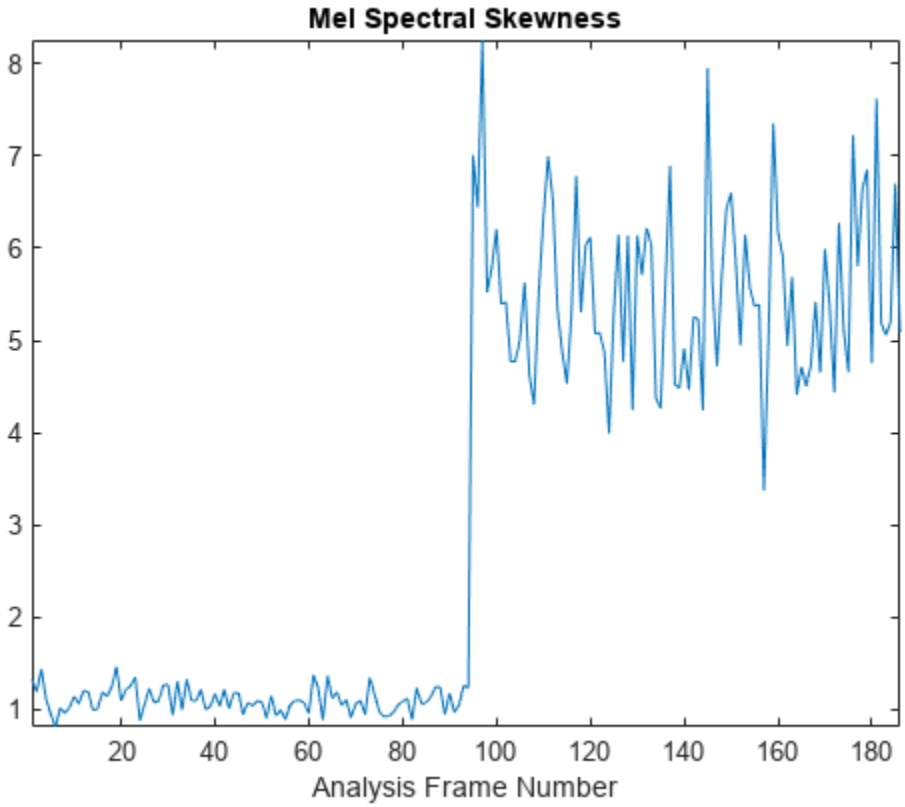
```
idx = info(aFE);

surf(log10(features(:,idx.melSpectrum)),"EdgeColor","none");
view([90,-90])
axis tight
title("Mel Spectrum")
ylabel("Analysis Frame Number")
```




```
plot(features(:,idx.spectralCentroid))
axis tight
title("Mel Spectral Centroid")
xlabel("Analysis Frame Number")
```




```
plot(features(:,idx.spectralSkewness))
axis tight
title("Mel Spectral Skewness")
xlabel("Analysis Frame Number")
```



Get List of All Features `audioFeatureExtractor` Provides

Create a default `audioFeatureExtractor` object. By default, all feature extractors are disabled.

```
aFE = audioFeatureExtractor
```

```
aFE =
  audioFeatureExtractor with properties:

   Properties
                    Window: [1024x1 double]
             OverlapLength: 512
                SampleRate: 44100
                 FFTLength: []
    SpectralDescriptorInput: 'linearSpectrum'
        FeatureVectorLength: 0

   Enabled Features
     none

   Disabled Features
     linearSpectrum, melSpectrum, barkSpectrum, erbSpectrum, mfcc, mfccDelta
     mfccDeltaDelta, gtcc, gtccDelta, gtccDeltaDelta, spectralCentroid, spectralCrest
     spectralDecrease, spectralEntropy, spectralFlatness, spectralFlux, spectralKurtosis, spectra
     spectralSkewness, spectralSlope, spectralSpread, pitch, harmonicRatio, zerocrossrate
```

```
     shortTimeEnergy


   To extract a feature, set the corresponding property to true.
   For example, obj.mfcc = true, adds mfcc to the list of enabled features.
```

The `info` function returns information about enabled feature extractors. To view information about all feature extractors, call `info` using the `"all"` flag.

```
[idx,params] = info(aFE,"all")
```

```
idx = struct with fields:
          linearSpectrum: [1x0 double]
             melSpectrum: [1x0 double]
            barkSpectrum: [1x0 double]
             erbSpectrum: [1x0 double]
                    mfcc: [1x0 double]
               mfccDelta: [1x0 double]
          mfccDeltaDelta: [1x0 double]
                    gtcc: [1x0 double]
               gtccDelta: [1x0 double]
          gtccDeltaDelta: [1x0 double]
        spectralCentroid: [1x0 double]
           spectralCrest: [1x0 double]
        spectralDecrease: [1x0 double]
         spectralEntropy: [1x0 double]
        spectralFlatness: [1x0 double]
            spectralFlux: [1x0 double]
        spectralKurtosis: [1x0 double]
    spectralRolloffPoint: [1x0 double]
        spectralSkewness: [1x0 double]
           spectralSlope: [1x0 double]
          spectralSpread: [1x0 double]
                   pitch: [1x0 double]
           harmonicRatio: [1x0 double]
           zerocrossrate: [1x0 double]
         shortTimeEnergy: [1x0 double]


params = struct with fields:
          linearSpectrum: [1x1 struct]
             melSpectrum: [1x1 struct]
            barkSpectrum: [1x1 struct]
             erbSpectrum: [1x1 struct]
                    mfcc: [1x1 struct]
                    gtcc: [1x1 struct]
            spectralFlux: [1x1 struct]
    spectralRolloffPoint: [1x1 struct]
                   pitch: [1x1 struct]
           zerocrossrate: [1x1 struct]
```

Use the `idx` struct to enable all feature extractors on the `audioFeatureExtractor` object.

```
features = fieldnames(idx);
for i = 1:numel(features)
    aFE.(features{i}) = true;
```

```
end
aFE
```

```
aFE =
  audioFeatureExtractor with properties:

   Properties
                     Window: [1024x1 double]
              OverlapLength: 512
                 SampleRate: 44100
                  FFTLength: []
    SpectralDescriptorInput: 'linearSpectrum'
        FeatureVectorLength: 713

   Enabled Features
     linearSpectrum, melSpectrum, barkSpectrum, erbSpectrum, mfcc, mfccDelta
     mfccDeltaDelta, gtcc, gtccDelta, gtccDeltaDelta, spectralCentroid, spectralCrest
     spectralDecrease, spectralEntropy, spectralFlatness, spectralFlux, spectralKurtosis, spectra
     spectralSkewness, spectralSlope, spectralSpread, pitch, harmonicRatio, zerocrossrate
     shortTimeEnergy

   Disabled Features
     none


   To extract a feature, set the corresponding property to true.
   For example, obj.mfcc = true, adds mfcc to the list of enabled features.
```

Determine Settable Parameters of Individual Feature Extractors

Create an `audioFeatureExtractor` to extract the ERB spectrum.

```
aFE = audioFeatureExtractor(erbSpectrum=true)
```

```
aFE =
  audioFeatureExtractor with properties:

   Properties
                     Window: [1024x1 double]
              OverlapLength: 512
                 SampleRate: 44100
                  FFTLength: []
    SpectralDescriptorInput: 'linearSpectrum'
        FeatureVectorLength: 43

   Enabled Features
     erbSpectrum

   Disabled Features
     linearSpectrum, melSpectrum, barkSpectrum, mfcc, mfccDelta, mfccDeltaDelta
     gtcc, gtccDelta, gtccDeltaDelta, spectralCentroid, spectralCrest, spectralDecrease
     spectralEntropy, spectralFlatness, spectralFlux, spectralKurtosis, spectralRolloffPoint, spe
     spectralSlope, spectralSpread, pitch, harmonicRatio, zerocrossrate, shortTimeEnergy
```

```
   To extract a feature, set the corresponding property to true.
   For example, obj.mfcc = true, adds mfcc to the list of enabled features.
```

The second output argument from `info` is a `struct` that contains the settable parameters of the individual feature extractors and their current value.

```
[~,params] = info(aFE)
```

```
params = struct with fields:
    erbSpectrum: [1x1 struct]
```

```
params.erbSpectrum
```

```
ans = struct with fields:
                    NumBands: 43
               FrequencyRange: [0 22050]
    FilterBankNormalization: "bandwidth"
        WindowNormalization: 1
               SpectrumType: "power"
```

If you are using the default parameter values, then the parameters are dynamic and updated when properties they depend on are updated. For example, the default frequency range of the ERB filter bank and the default number of bandpass filters in the ERB filter bank depends on the sample rate. Decrease the sample rate of the `audioFeatureExtractor` object and then call `info` again.

```
aFE.SampleRate = 16e3;
[~,params] = info(aFE);
params.erbSpectrum
```

```
ans = struct with fields:
                    NumBands: 34
               FrequencyRange: [0 8000]
    FilterBankNormalization: "bandwidth"
        WindowNormalization: 1
               SpectrumType: "power"
```

You can modify the individual feature extractor parameters using `setExtractorParameters`. Set the number of bands to `40` and the spectrum type to `"magnitude"`. Call `info` to confirm that the parameters are updated.

```
params.erbSpectrum.NumBands = 40;
params.erbSpectrum.SpectrumType = "magnitude";
setExtractorParameters(aFE,erbSpectrum=params.erbSpectrum)
[~,params] = info(aFE);
params.erbSpectrum
```

```
ans = struct with fields:
                    NumBands: 40
               FrequencyRange: [0 8000]
    FilterBankNormalization: "bandwidth"
        WindowNormalization: 1
               SpectrumType: "magnitude"
```

When you set individual feature extractor parameters, they remain at the set value until you set them to another value or return them to defaults. Return the sample rate of the `audioFeatureExtractor` object to its initial value and then call `info`. The parameters remain at their set value.

```
aFE.SampleRate = 44.1e3;

[~,params] = info(aFE);
params.erbSpectrum
```

```
ans = struct with fields:
                    NumBands: 40
               FrequencyRange: [0 8000]
    FilterBankNormalization: "bandwidth"
        WindowNormalization: 1
               SpectrumType: "magnitude"
```

To return parameters to their default values, call `setExtractorParameters` and specify no parameters.

```
setExtractorParameters(aFE,"erbSpectrum")
[~,params] = info(aFE);
params.erbSpectrum
```

```
ans = struct with fields:
                    NumBands: 43
               FrequencyRange: [0 22050]
    FilterBankNormalization: "bandwidth"
        WindowNormalization: 1
               SpectrumType: "power"
```

Input Arguments

aFE — Input object

`audioFeatureExtractor` object

`audioFeatureExtractor` object.

Output Arguments

idx — Mapping of requested features with output from extract

struct

Mapping of requested features with output from `extract`, returned as a struct with field names corresponding to individual feature extractors and field values corresponding to column indices.

params — Settable parameters of individual feature extractors

struct

Settable parameters of individual feature extractors, returned as a struct with field names corresponding to individual feature extractors and field values containing parameter specification structs. The parameter specification structs have field names corresponding to settable parameter names and field values corresponding to the current parameter setting.

Version History

Introduced in R2019b

See Also

`audioFeatureExtractor`

extract

Extract audio features

Syntax

```
features = extract(aFE,audioIn)
```

Description

`features = extract(aFE,audioIn)` returns an array containing features of the audio input.

Examples

Extract and Normalize Audio Features

Read in an audio signal.

```
[audioIn,fs] = audioread("Counting-16-44p1-mono-15secs.wav");
```

Create an `audioFeatureExtractor` to extract the centroid of the Bark spectrum, the kurtosis of the Bark spectrum, and the `pitch` of an audio signal.

```
aFE = audioFeatureExtractor("SampleRate",fs, ...
    "SpectralDescriptorInput","barkSpectrum", ...
    "spectralCentroid",true, ...
    "spectralKurtosis",true, ...
    "pitch",true)
```

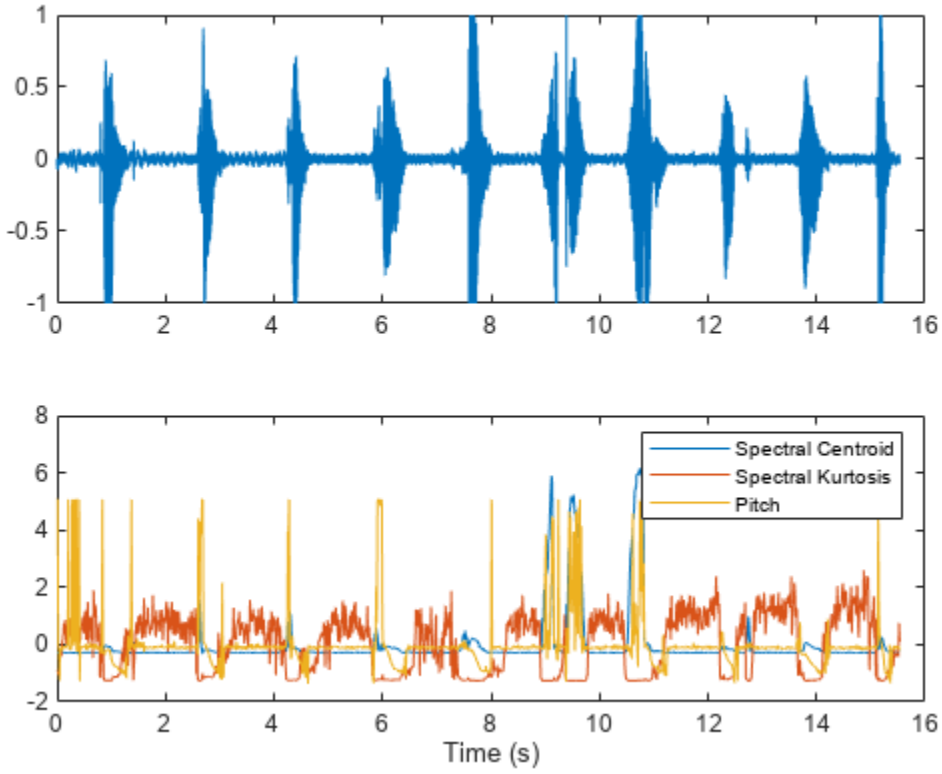
```
aFE =
  audioFeatureExtractor with properties:

   Properties
                     Window: [1024x1 double]
              OverlapLength: 512
                 SampleRate: 44100
                  FFTLength: []
    SpectralDescriptorInput: 'barkSpectrum'
        FeatureVectorLength: 3

   Enabled Features
     spectralCentroid, spectralKurtosis, pitch

   Disabled Features
     linearSpectrum, melSpectrum, barkSpectrum, erbSpectrum, mfcc, mfccDelta
     mfccDeltaDelta, gtcc, gtccDelta, gtccDeltaDelta, spectralCrest, spectralDecrease
     spectralEntropy, spectralFlatness, spectralFlux, spectralRolloffPoint, spectralSkewness, spe
     spectralSpread, harmonicRatio, zerocrossrate, shortTimeEnergy


   To extract a feature, set the corresponding property to true.
   For example, obj.mfcc = true, adds mfcc to the list of enabled features.
```

Call `extract` to extract the features from the audio signal. Normalize the features by their mean and standard deviation.

```
features = extract(aFE,audioIn);
features = (features - mean(features,1))./std(features,[],1);
```

Plot the normalized features over time.

```
idx = info(aFE);
duration = size(audioIn,1)/fs;

subplot(2,1,1)
t = linspace(0,duration,size(audioIn,1));
plot(t,audioIn)

subplot(2,1,2)
t = linspace(0,duration,size(features,1));
plot(t,features(:,idx.spectralCentroid), ...
     t,features(:,idx.spectralKurtosis), ...
     t,features(:,idx.pitch));
legend("Spectral Centroid","Spectral Kurtosis", "Pitch")
xlabel("Time (s)")
```



Input Arguments

aFE — Input object
`audioFeatureExtractor` object

`audioFeatureExtractor` object.

`audioIn` — Input audio
column vector | matrix

Input audio, specified as a column vector or matrix of independent channels (columns).

Data Types: `single` | `double`

Output Arguments

`features` — Extracted audio features
vector | matrix | 3-D array

Extracted audio features, returned as an *L*-by-*M*-by-*N* array, where:

- *L* –– Number of feature vectors (hops)
- *M* –– Number of features extracted per analysis window
- *N* –– Number of channels

Data Types: `single` | `double`

Version History

Introduced in R2019b

See Also

`audioFeatureExtractor` | **Extract Audio Features**

audioFeatureExtractor

Streamline audio feature extraction

Description

`audioFeatureExtractor` encapsulates multiple audio feature extractors into a streamlined and modular implementation.

Creation

Syntax

```
aFE = audioFeatureExtractor()
aFE = audioFeatureExtractor(Name=Value)
```

Description

`aFE = audioFeatureExtractor()` creates an audio feature extractor with default property values.

`aFE = audioFeatureExtractor(Name=Value)` specifies nondefault properties for `aFE` using one or more name-value arguments.

Properties

Main Properties

`Window` — Analysis window
`hamming(1024,"periodic")` (default) | real vector

Analysis window, specified as a real vector.

Data Types: `single` | `double`

`OverlapLength` — Overlap length of adjacent analysis windows
`512` (default) | integer in the range [0, `numel(Window)`)

Overlap length of adjacent analysis windows, specified as an integer in the range [0, `numel(Window)`).

Data Types: `single` | `double`

`FFTLength` — FFT length
`[]` (default) | positive integer

FFT length, specified as an integer. The default value of `[]` means that the FFT length is equal to the window length `numel(Window)`.

Data Types: `single` | `double`

`SampleRate` — Input sample rate (Hz)
`44100` (default) | positive scalar

Input sample rate in Hz, specified as a positive scalar.

Data Types: `single` | `double`

`SpectralDescriptorInput` — Input to spectral descriptors
`"linearSpectrum"` (default) | `"melSpectrum"` | `"barkSpectrum"` | `"erbSpectrum"`

Input to spectral descriptors, specified as `"linearSpectrum"`, `"melSpectrum"`, `"barkSpectrum"`, or `"erbSpectrum"`.

Spectral descriptors affected by this property are:

- spectralCentroid
- spectralCrest
- spectralDecrease
- spectralEntropy
- spectralFlatness
- spectralFlux
- spectralKurtosis
- spectralRolloffPoint
- spectralSkewness
- spectralSlope
- spectralSpread

The spectrum input to the spectral descriptors is the same as output from the corresponding feature:

- linearSpectrum
- melSpectrum
- barkSpectrum
- erbSpectrum

For example, if you set `SpectralDescriptorInput` to `"barkSpectrum"`, and `spectralCentroid` to `true`, then `aFE` returns the centroid of the default Bark spectrum.

```
[audioIn,fs] = audioread("Counting-16-44p1-mono-15secs.wav");
aFE = audioFeatureExtractor(SampleRate=fs, ...
                            SpectralDescriptorInput="barkSpectrum", ...
                            spectralCentroid=true);
barkSpectralCentroid = extract(aFE,audioIn);
```

If you specify a nondefault `barkSpectrum` using `setExtractorParameters`, then the nondefault Bark spectrum is the input to the spectral descriptors. For example, if you call `setExtractorParameters(aFE,"barkSpectrum",NumBands=40)`, then `aFE` returns the centroid of a 40-band Bark spectrum.

```
setExtractorParameters(aFE,"barkSpectrum",NumBands=40)
bark40SpectralCentroid = extract(aFE,audioIn);
```

Data Types: `char` | `string`

`FeatureVectorLength` — Number of features output from extract

positive integer

This property is read-only.

Total number of features output from `extract` for the current object configuration, specified as a positive integer. `FeatureVectorLength` is equal to the second dimension of the output from the `extract` function.

Data Types: `single` | `double`

Features to Extract

`linearSpectrum` — Extract linear spectrum

`false` (default) | `true`

Extract the one-sided linear spectrum, specified as `true` or `false`.

To set parameters of the linear spectrum extraction, use `setExtractorParameters`:

```
setExtractorParameters(aFE,"linearSpectrum",Name=Value)
```

Settable parameters for the linear spectrum extraction are:

- `FrequencyRange` –– Frequency range of the extracted spectrum in Hz, specified as a two-element vector of increasing numbers in the range [0, SampleRate/2]. If unspecified, `FrequencyRange` defaults to `[0, SampleRate/2]`.
- `SpectrumType` –– Spectrum type, specified as `"power"` or `"magnitude"`. If unspecified, `SpectrumType` defaults to `"power"`.
- `WindowNormalization` –– Apply window normalization, specified as `true` or `false`. If unspecified, `WindowNormalization` defaults to `true`.

Data Types: `logical`

`melSpectrum` — Extract mel spectrum

`false` (default) | `true`

Extract the one-sided mel spectrum, specified as `true` or `false`.

To set parameters of the mel spectrum extraction, use `setExtractorParameters`:

```
setExtractorParameters(aFE,"melSpectrum",Name=Value)
```

Settable parameters for the mel spectrum extraction are:

- `FrequencyRange` –– Frequency range of the extracted spectrum in Hz, specified as a two-element vector of increasing numbers in the range [0, SampleRate/2]. If unspecified, `FrequencyRange` defaults to `[0, SampleRate/2]`.
- `SpectrumType` –– Spectrum type, specified as `"power"` or `"magnitude"`. If unspecified, `SpectrumType` defaults to `"power"`.
- `NumBands` –– Number of mel bands, specified as an integer. If unspecified, `NumBands` defaults to `32`.
- `FilterBankNormalization` –– Normalization applied to bandpass filters, specified as `"bandwidth"`, `"area"`, or `"none"`. If unspecified, `FilterBankNormalization` defaults to `"bandwidth"`.

- `WindowNormalization` -- Apply window normalization, specified as `true` or `false`. If unspecified, `WindowNormalization` defaults to `true`.
- `FilterBankDesignDomain` -- Domain in which the filter bank is designed, specified as either `"linear"` or `"warped"`. If unspecified, `FilterBankDesignDomain` defaults to `"linear"`.

Data Types: `logical`

`barkSpectrum` — Extract Bark spectrum
`false` (default) | `true`

Extract the one-sided Bark spectrum, specified as `true` or `false`.

To set parameters of the Bark spectrum extraction, use `setExtractorParameters`:

```
setExtractorParameters(aFE,"barkSpectrum",Name=Value)
```

Settable parameters for the Bark spectrum extraction are:

- `FrequencyRange` -- Frequency range of the extracted spectrum in Hz, specified as a two-element vector of increasing numbers in the range [0, SampleRate/2]. If unspecified, `FrequencyRange` defaults to `[0, SampleRate/2]`.
- `SpectrumType` -- Spectrum type, specified as `"power"` or `"magnitude"`. If unspecified, `SpectrumType` defaults to `"power"`.
- `NumBands` -- Number of Bark bands, specified as an integer. If unspecified, `NumBands` defaults to `32`.
- `FilterBankNormalization` -- Normalization applied to bandpass filters, specified as `"bandwidth"`, `"area"`, or `"none"`. If unspecified, `FilterBankNormalization` defaults to `"bandwidth"`.
- `WindowNormalization` -- Apply window normalization, specified as `true` or `false`. If unspecified, `WindowNormalization` defaults to `true`.
- `FilterBankDesignDomain` -- Domain in which the filter bank is designed, specified as either `"linear"` or `"warped"`. If unspecified, `FilterBankDesignDomain` defaults to `"linear"`.

Data Types: `logical`

`erbSpectrum` — Extract ERB spectrum
`false` (default) | `true`

Extract the one-sided ERB spectrum, specified as `true` or `false`.

To set parameters of the ERB spectrum extraction, use `setExtractorParameters`:

```
setExtractorParameters(aFE,"erbSpectrum",Name=Value)
```

Settable parameters for the ERB spectrum extraction are:

- `FrequencyRange` -- Frequency range of the extracted spectrum in Hz, specified as a two-element vector of increasing numbers in the range [0, SampleRate/2]. If unspecified, `FrequencyRange` defaults to `[0, SampleRate/2]`.
- `SpectrumType` -- Spectrum type, specified as `"power"` or `"magnitude"`. If unspecified, `SpectrumType` defaults to `"power"`.
- `NumBands` -- Number of ERB bands, specified as an integer. If unspecified, `NumBands` defaults to `ceil(hz2erb(FrequencyRange(2))-hz2erb(FrequencyRange(1)))`.

- `FilterBankNormalization` -- Normalization applied to bandpass filters, specified as `"bandwidth"`, `"area"`, or `"none"`. If unspecified, `FilterBankNormalization` defaults to `"bandwidth"`.
- `WindowNormalization` -- Apply window normalization, specified as `true` or `false`. If unspecified, `WindowNormalization` defaults to `true`.

Data Types: `logical`

`mfcc` — Extract mel-frequency cepstral coefficients (MFCC)
`false` (default) | `true`

Extract mel-frequency cepstral coefficients (MFCC), specified as `true` or `false`.

To set parameters of the MFCC extraction, use `setExtractorParameters`:

```
setExtractorParameters(aFE,"mfcc",Name=Value)
```

Settable parameters for the MFCC extraction are:

- `NumCoeffs` -- Number of coefficients returned for each window, specified as a positive integer. If unspecified, `NumCoeffs` defaults to 13.
- `DeltaWindowLength` -- Delta window length, specified as an odd integer greater than 2. If unspecified, `DeltaWindowLength` defaults to 9. This parameter affects the `mfccDelta` and `mfccDeltaDelta` features.
- `Rectification` -- Type of nonlinear rectification, specified as `"log"` or `"cubic-root"`.

The mel-frequency cepstral coefficients are calculated using the melSpectrum.

Data Types: `logical`

`mfccDelta` — Extract delta of MFCC
`false` (default) | `true`

Extract delta of MFCC, specified as `true` or `false`.

The delta MFCC is calculated based on the extracted MFCC. Parameters set on `mfcc` affect `mfccDelta`.

Data Types: `logical`

`mfccDeltaDelta` — Extract delta-delta of MFCC
`false` (default) | `true`

Extract delta-delta of MFCC, specified as `true` or `false`.

The delta-delta MFCC is calculated based on the extracted MFCC. Parameters set on `mfcc` affect `mfccDeltaDelta`.

Data Types: `logical`

`gtcc` — Extract gammatone cepstral coefficients (GTCC)
`false` (default) | `true`

Extract gammatone cepstral coefficients (GTCC), specified as `true` or `false`.

To set parameters of the GTCC extraction, use `setExtractorParameters`:

```
setExtractorParameters(aFE,"gtcc",Name=Value)
```

Settable parameters for the GTCC extraction are:

- `NumCoeffs` –– Number of coefficients returned for each window, specified as a positive integer. If unspecified, `NumCoeffs` defaults to `13`.
- `DeltaWindowLength` –– Delta window length, specified as an odd integer greater than 2. If unspecified, `DeltaWindowLength` defaults to `9`. This parameter affects the `gtccDelta` and `gtccDeltaDelta` features.
- `Rectification` –– Type of nonlinear rectification, specified as `"log"` or `"cubic-root"`.

The gammatone cepstral coefficients are calculated using the erbSpectrum.

Data Types: `logical`

`gtccDelta` — Extract delta of GTCC
`false` (default) | `true`

Extract delta of GTCC, specified as `true` or `false`.

The delta GTCC is calculated based on the extracted GTCC. Parameters set on `gtcc` affect `gtccDelta`.

Data Types: `logical`

`gtccDeltaDelta` — Extract delta-delta of GTCC
`false` (default) | `true`

Extract delta-delta of GTCC, specified as `true` or `false`.

The delta-delta GTCC is calculated based on the extracted GTCC. Parameters set on `gtcc` affect `gtccDeltaDelta`.

Data Types: `logical`

`spectralCentroid` — Extract spectral centroid
`false` (default) | `true`

Extract spectral centroid, specified as `true` or `false`.

The spectral centroid is calculated on one of the following spectral representations, as specified by the SpectralDescriptorInput property:

- linearSpectrum
- melSpectrum
- barkSpectrum
- erbSpectrum

Data Types: `logical`

`spectralCrest` — Extract spectral crest
`false` (default) | `true`

Extract spectral crest, specified as `true` or `false`.

The spectral crest is calculated on one of the following spectral representations, as specified by the SpectralDescriptorInput property:

- linearSpectrum
- melSpectrum
- barkSpectrum
- erbSpectrum

Data Types: `logical`

`spectralDecrease` — Extract spectral decrease
`false` (default) | `true`

Extract spectral decrease, specified as `true` or `false`.

The spectral decrease is calculated on one of the following spectral representations, as specified by the SpectralDescriptorInput property:

- linearSpectrum
- melSpectrum
- barkSpectrum
- erbSpectrum

Data Types: `logical`

`spectralEntropy` — Extract spectral entropy
`false` (default) | `true`

Extract spectral entropy, specified as `true` or `false`.

The spectral entropy is calculated on one of the following spectral representations, as specified by the SpectralDescriptorInput property:

- linearSpectrum
- melSpectrum
- barkSpectrum
- erbSpectrum

Data Types: `logical`

`spectralFlatness` — Extract spectral flatness
`false` (default) | `true`

Extract spectral flatness, specified as `true` or `false`.

The spectral flatness is calculated on one of the following spectral representations, as specified by the SpectralDescriptorInput property:

- linearSpectrum
- melSpectrum
- barkSpectrum

- erbSpectrum

Data Types: `logical`

`spectralFlux` — Extract spectral flux
`false` (default) | `true`

Extract spectral flux, specified as `true` or `false`.

The spectral flux is calculated on one of the following spectral representations, as specified by the SpectralDescriptorInput property:

- linearSpectrum
- melSpectrum
- barkSpectrum
- erbSpectrum

To set parameters of the spectral flux extraction, use `setExtractorParameters`:

```
setExtractorParameters(aFE,"spectralFlux",Name=Value)
```

Settable parameters for the spectral flux extraction are:

- `NormType` –– Norm type used to calculate the spectral flux, specified as `1` or `2`. If unspecified, `NormType` defaults to `2`.

Data Types: `logical`

`spectralKurtosis` — Extract spectral kurtosis
`false` (default) | `true`

Extract spectral kurtosis, specified as `true` or `false`.

The spectral kurtosis is calculated on one of the following spectral representations, as specified by the SpectralDescriptorInput property:

- linearSpectrum
- melSpectrum
- barkSpectrum
- erbSpectrum

Data Types: `logical`

`spectralRolloffPoint` — Extract spectral rolloff point
`false` (default) | `true`

Extract spectral rolloff point, specified as `true` or `false`.

The spectral rolloff point is calculated on one of the following spectral representations, as specified by the SpectralDescriptorInput property:

- linearSpectrum
- melSpectrum

- barkSpectrum
- erbSpectrum

To set parameters of the spectral rolloff point extraction, use `setExtractorParameters`:

```
setExtractorParameters(aFE,"spectralRolloffPoint",Name=Value)
```

Settable parameters for the spectral flux extraction are:

- `Threshold` –– Threshold of the rolloff point, specified as a scalar in the range (0, 1). If unspecified, `Threshold` defaults to `0.95`.

Data Types: `logical`

`spectralSkewness` — Extract spectral skewness
`false` (default) | `true`

Extract spectral skewness, specified as `true` or `false`.

The spectral skewness is calculated on one of the following spectral representations, as specified by the SpectralDescriptorInput property:

- linearSpectrum
- melSpectrum
- barkSpectrum
- erbSpectrum

Data Types: `logical`

`spectralSlope` — Extract spectral slope
`false` (default) | `true`

Extract spectral slope, specified as `true` or `false`.

The spectral slope is calculated on one of the following spectral representations, as specified by the SpectralDescriptorInput property:

- linearSpectrum
- melSpectrum
- barkSpectrum
- erbSpectrum

Data Types: `logical`

`spectralSpread` — Extract spectral spread
`false` (default) | `true`

Extract spectral spread, specified as `true` or `false`.

The spectral spread is calculated on one of the following spectral representations, as specified by the SpectralDescriptorInput property:

- linearSpectrum

- melSpectrum
- barkSpectrum
- erbSpectrum

Data Types: `logical`

`pitch` — Extract pitch
`false` (default) | `true`

Extract pitch, specified as `true` or `false`.

To set parameters of the pitch extraction, use `setExtractorParameters`:

```
setExtractorParameters(aFE,"pitch",Name=Value)
```

Settable parameters for the pitch extraction are:

- `Method` -- Method used to calculate the pitch, specified as `"PEF"`, `"NCF"`, `"CEP"`, `"LHS"`, or `"SRH"`. If unspecified, `Method` defaults to `"NCF"`. For a description of available pitch extraction methods, see `pitch`.
- `Range` -- Range within to search for the pitch in Hz, specified as a two-element row vector of increasing values. If unspecified, `Range` defaults to `[50,400]`.
- `MedianFilterLength` -- Median filter length used to smooth pitch estimates over time, specified as a positive integer. If unspecified, `MedianFilterLength` defaults to `1` (no median filtering).

Data Types: `logical`

`harmonicRatio` — Extract harmonic ratio
`false` (default) | `true`

Extract harmonic ratio, specified as `true` or `false`.

Data Types: `logical`

`zerocrossrate` — Extract zero-crossing rate
`false` (default) | `true`

Extract zero-crossing rate, specified as `true` or `false`.

To set parameters of the zero-crossing rate extraction, use `setExtractorParameters`:

```
setExtractorParameters(aFE,"zerocrossrate",Name=Value)
```

Settable parameters for the zero-crossing rate extraction are:

- `Method` -- Method for computing the zero-crossing rate, specified as `"difference"` or `"comparison"`. If unspecified, `Method`, defaults to `"difference"`. For more information, see `zerocrossrate`.
- `Level` -- Signal level for which the crossing rate is computed, specified as a real scalar. `audioFeatureExtractor` subtracts the `Level` value from the signal and then finds the zero crossings. If unspecified, `Level` defaults to `0`.
- `Threshold` -- Threshold above and below the `Level` value over which the crossing rate is computed, specified as a real scalar. `audioFeatureExtractor` sets all the values of the input in the range `[-Threshold, Threshold]` to `0` and then finds the zero crossings. If unspecified, `Threshold` defaults to `0`.

- `TransitionEdge` — Transitions to include when counting zero crossings, specified as `"falling"`, `"rising"`, or `"both"`. If you specify `"falling"`, only negative-going transitions are counted. If you specify `"rising"`, only positive-going transitions are counted. If unspecified, `TransitionEdge` defaults to `"both"`.
- `ZeroPositive` — Sign convention, specified as a logical scalar. If you specify `ZeroPositive` as `true`, then `0` is considered positive. If you specify `ZeroPositive` as `false`, then `audioFeatureExtractor` considers `0`, `−1`, and `+1` to have distinct signs following the convention of the `sign` function. If unspecified, `ZeroPositive` defaults to `false`.

Data Types: `logical`

`shortTimeEnergy` — Extract short-time energy
`false` (default) | `true`

Extract short-time energy, specified as `true` or `false`. The short-time energy is computed using
```
sTE = sum(xbw.^2,1),
```
where `xbw` is the buffered and windowed signal.

Example: Chirp Function

Generate a chirp sampled at 1 kHz for 3 seconds. The instantaneous frequency is 100 Hz at $t = 0$ and crosses 200 Hz at $t = 1$ second. Divide the signal into 103-sample segments with 43 samples of overlap between adjoining segments. Window each segment with a periodic Hamming window.

```
fs = 1e3;
x = chirp(0:1/fs:3,100,1,200)';

win = hamming(103,"periodic");
nover = 43;

[xb,~] = buffer(x,length(win),nover,"nodelay");
xbw = xb.*win;
```

Compute the short-time energy using the definition.

```
Edef = sum(xbw.^2,1)';
```

Use `audioFeatureExtractor` to compute the short-time energy.

```
EaFE = extract(audioFeatureExtractor(shortTimeEnergy=true, ...
    SampleRate=fs,Window=win,OverlapLength=nover),x);
```

Verify that both procedures give the same short-time energy.

```
dff = max(abs(EaFE-Edef))
```

```
dff = 0
```

Data Types: `logical`

Object Functions

extract	Extract audio features
setExtractorParameters	Set nondefault parameter values for individual feature extractors
info	Output mapping and individual feature extractor parameters
generateMATLABFunction	Create MATLAB function compatible with C/C++ code generation

plotFeatures	Plot extracted audio features

Examples

Extract Multiple Audio Features

Read in an audio signal.

```
[audioIn,fs] = audioread("Counting-16-44p1-mono-15secs.wav");
```

Create an `audioFeatureExtractor` object that extracts the MFCC, delta MFCC, delta-delta MFCC, pitch, spectral centroid, zero-crossing rate, and short-time energy of the signal. Use a 30 ms analysis window with 20 ms overlap.

```
aFE = audioFeatureExtractor( ...
    SampleRate=fs, ...
    Window=hamming(round(0.03*fs),"periodic"), ...
    OverlapLength=round(0.02*fs), ...
    mfcc=true, ...
    mfccDelta=true, ...
    mfccDeltaDelta=true, ...
    pitch=true, ...
    spectralCentroid=true, ...
    zerocrossrate=true, ...
    shortTimeEnergy=true);
```

Call `extract` to extract the audio features from the audio signal.

```
features = extract(aFE,audioIn);
```

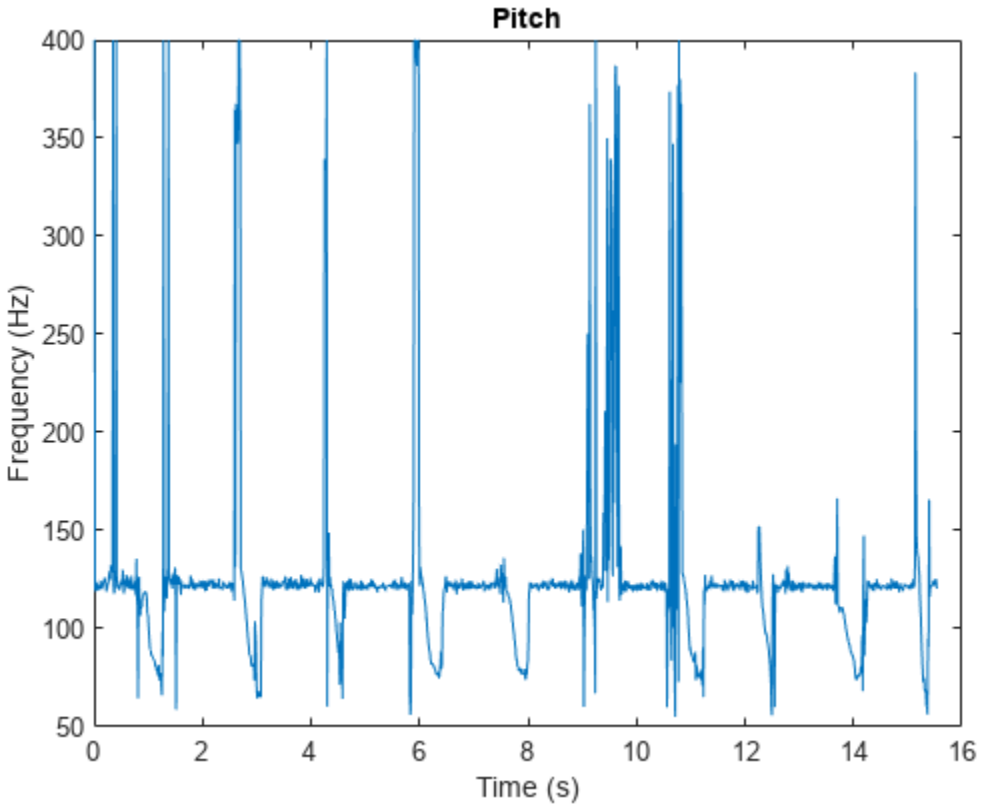
Use `info` to determine which column of the feature extraction matrix corresponds to the requested pitch extraction.

```
idx = info(aFE)
```

```
idx = struct with fields:
                mfcc: [1 2 3 4 5 6 7 8 9 10 11 12 13]
           mfccDelta: [14 15 16 17 18 19 20 21 22 23 24 25 26]
      mfccDeltaDelta: [27 28 29 30 31 32 33 34 35 36 37 38 39]
    spectralCentroid: 40
               pitch: 41
        zerocrossrate: 42
     shortTimeEnergy: 43
```

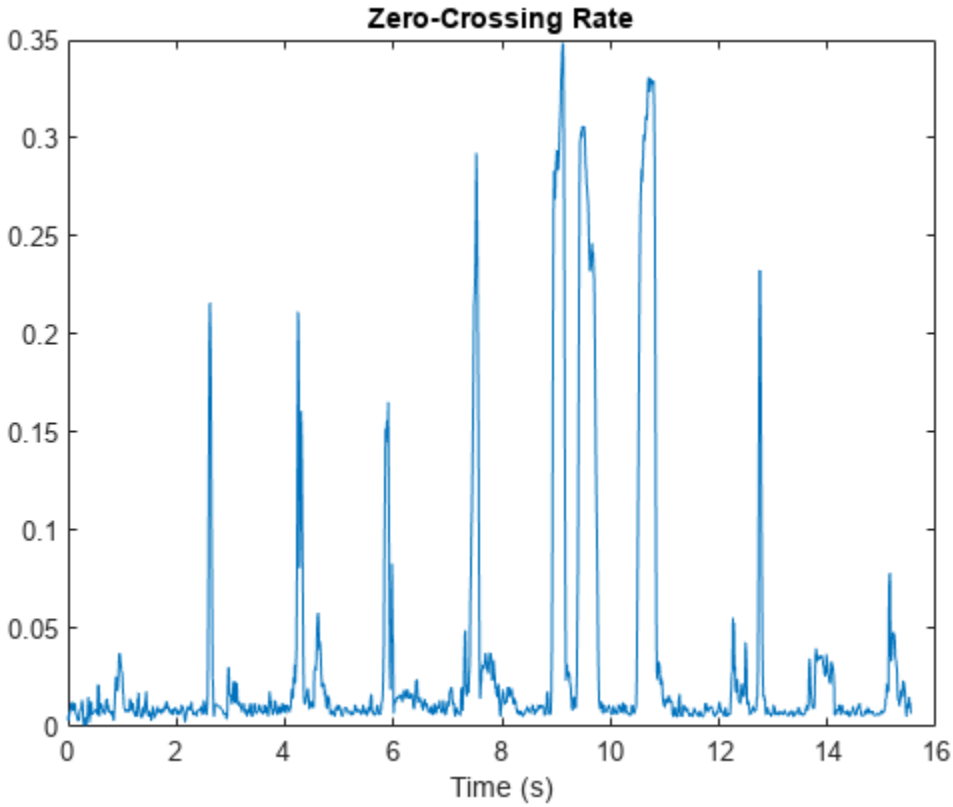
Plot the detected pitch over time.

```
t = linspace(0,size(audioIn,1)/fs,size(features,1));
plot(t,features(:,idx.pitch))
title("Pitch")
xlabel("Time (s)")
ylabel("Frequency (Hz)")
```



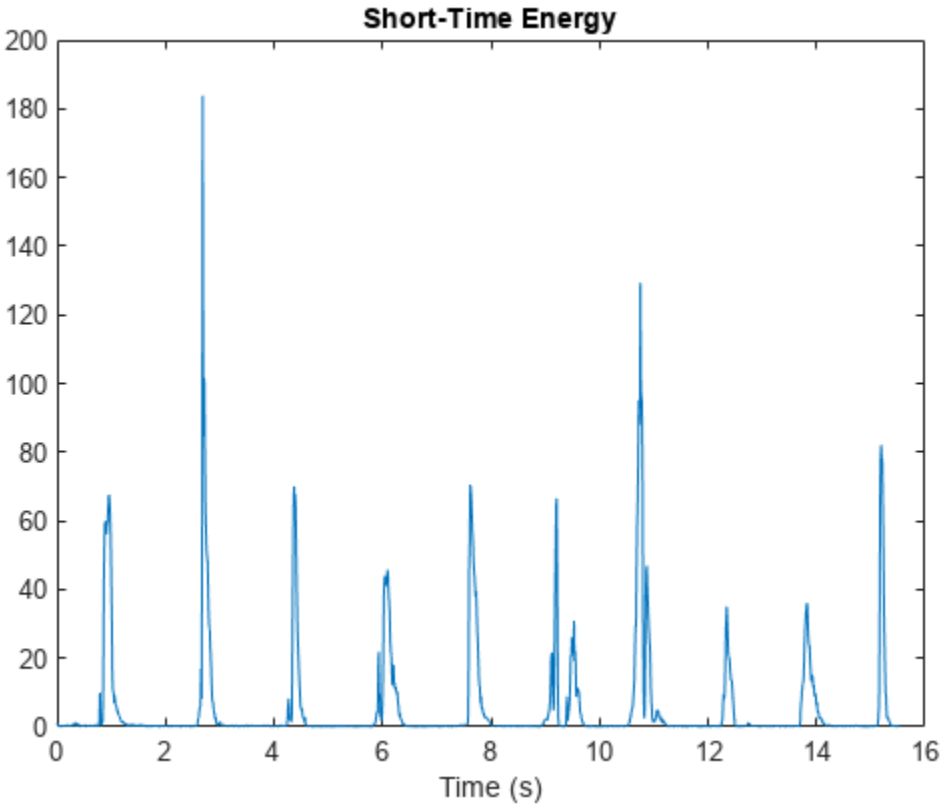
Plot the zero-crossing rate over time.

```
plot(t,features(:,idx.zerocrossrate))
title("Zero-Crossing Rate")
xlabel("Time (s)")
```



Plot the short-time energy over time.

```
plot(t,features(:,idx.shortTimeEnergy))
title("Short-Time Energy")
xlabel("Time (s)")
```



Extract Features from Dataset

Create an audio datastore that points to audio samples included with Audio Toolbox®.

```
folder = fullfile(matlabroot,"toolbox","audio","samples");
ads = audioDatastore(folder);
```

Find all files that correspond to a sample rate of 44.1 kHz and then `subset` the datastore.

```
keepFile = cellfun(@(x)contains(x,"44p1"),ads.Files);
ads = subset(ads,keepFile);
```

Convert the data to a `tall` array. `tall` arrays are evaluated only when you request them explicitly using `gather`. MATLAB® automatically optimizes the queued calculations by minimizing the number of passes through the data. If you have Parallel Computing Toolbox™, you can spread the calculations across multiple workers. The audio data is represented as an *M*-by-1 tall cell array, where *M* is the number of files in the audio datastore.

```
adsTall = tall(ads)
```

```
Starting parallel pool (parpool) using the 'local' profile ...
Connected to the parallel pool (number of workers: 6).

adsTall =
```

```
  M×1 tall cell array

    { 539648×1 double}
    { 227497×1 double}
    {   8000×1 double}
    { 685056×1 double}
    { 882688×2 double}
    {1115760×2 double}
    { 505200×2 double}
    {3195904×2 double}
        :         :
        :         :
```

Create an `audioFeatureExtractor` object to extract the mel spectrum, Bark spectrum, ERB spectrum, and linear spectrum from each audio file. Use the default analysis window and overlap length for the spectrum extraction.

```
aFE = audioFeatureExtractor(SampleRate=44.1e3, ...
    melSpectrum=true, ...
    barkSpectrum=true, ...
    erbSpectrum=true, ...
    linearSpectrum=true);
```

Define a `cellfun` function so that audio features are extracted from each cell of the tall array. Call `gather` to evaluate the tall array.

```
specsTall = cellfun(@(x)extract(aFE,x),adsTall,UniformOutput=false);
specs = gather(specsTall);
```

```
Evaluating tall expression using the Parallel Pool 'local':
- Pass 1 of 1: Completed in 14 sec
Evaluation completed in 14 sec
```

The `specs` variable returned from gather is a *numFiles*-by-1 cell array, where *numFiles* is the number of files in the datastore. Each element of the cell array is a *numHops*-by-*numFeatures*-by-*numChannels* array, where the number of hops and number of channels depends on the length and number of channels of the audio file, and the number of features is the requested number of features from the audio data.

```
numFiles = numel(specs)
```

```
numFiles = 12
```

```
[numHops1,numFeaturesFile1,numChanelsFile1] = size(specs{1})
```

```
numHops1 = 1053
```

```
numFeaturesFile1 = 620
```

```
numChanelsFile1 = 1
```

```
[numHops2,numFeaturesFile2,numChanelsFile2] = size(specs{2})
```

```
numHops2 = 443
```

```
numFeaturesFile2 = 620
```

```
numChanelsFile2 = 1
```

Visualize Extracted Audio Features

Use `plotFeatures` to visualize audio features extracted with an `audioFeatureExtractor` object.

Read in an audio signal from a file.

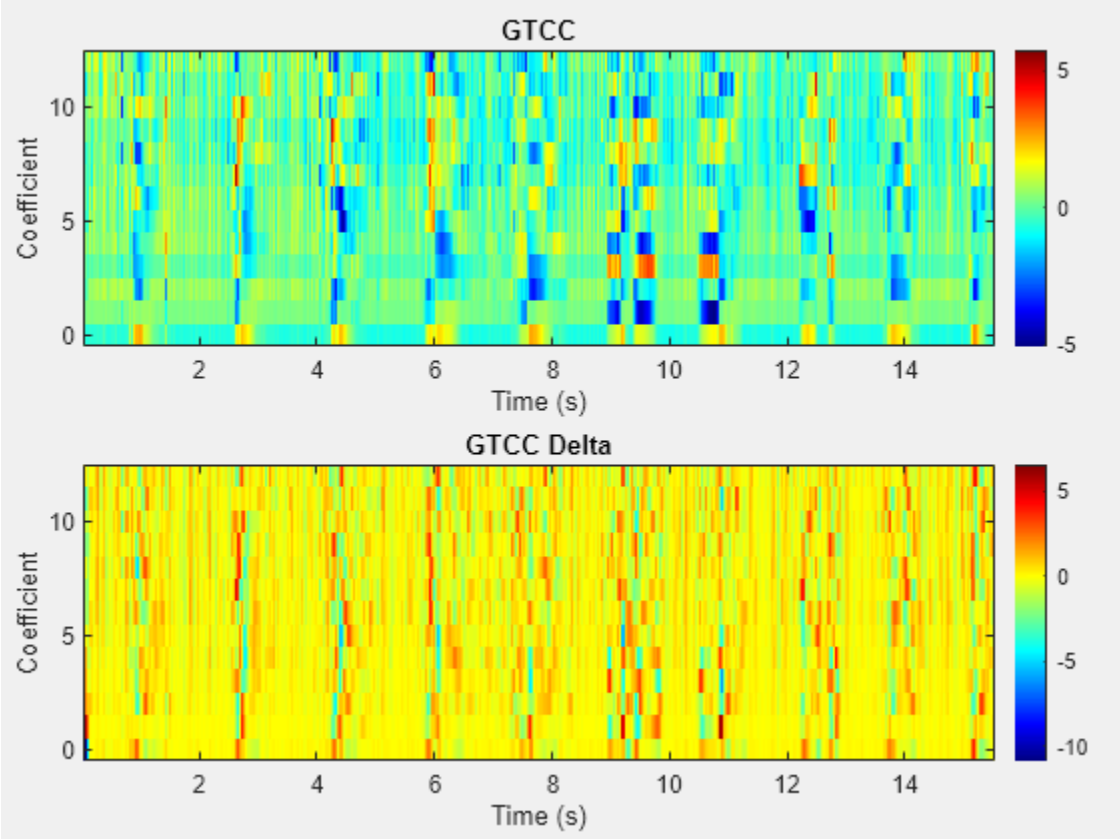
```
[audioIn,fs] = audioread("Counting-16-44p1-mono-15secs.wav");
```

Create an `audioFeatureExtractor` object that extracts the gammatone cepstral coefficients (GTCCs) and the delta of the GTCCs. Set the `SampleRate` property to the sample rate of the audio signal, and use the default values for the other properties.

```
afe = audioFeatureExtractor(SampleRate=fs,gtcc=true,gtccDelta=true);
```

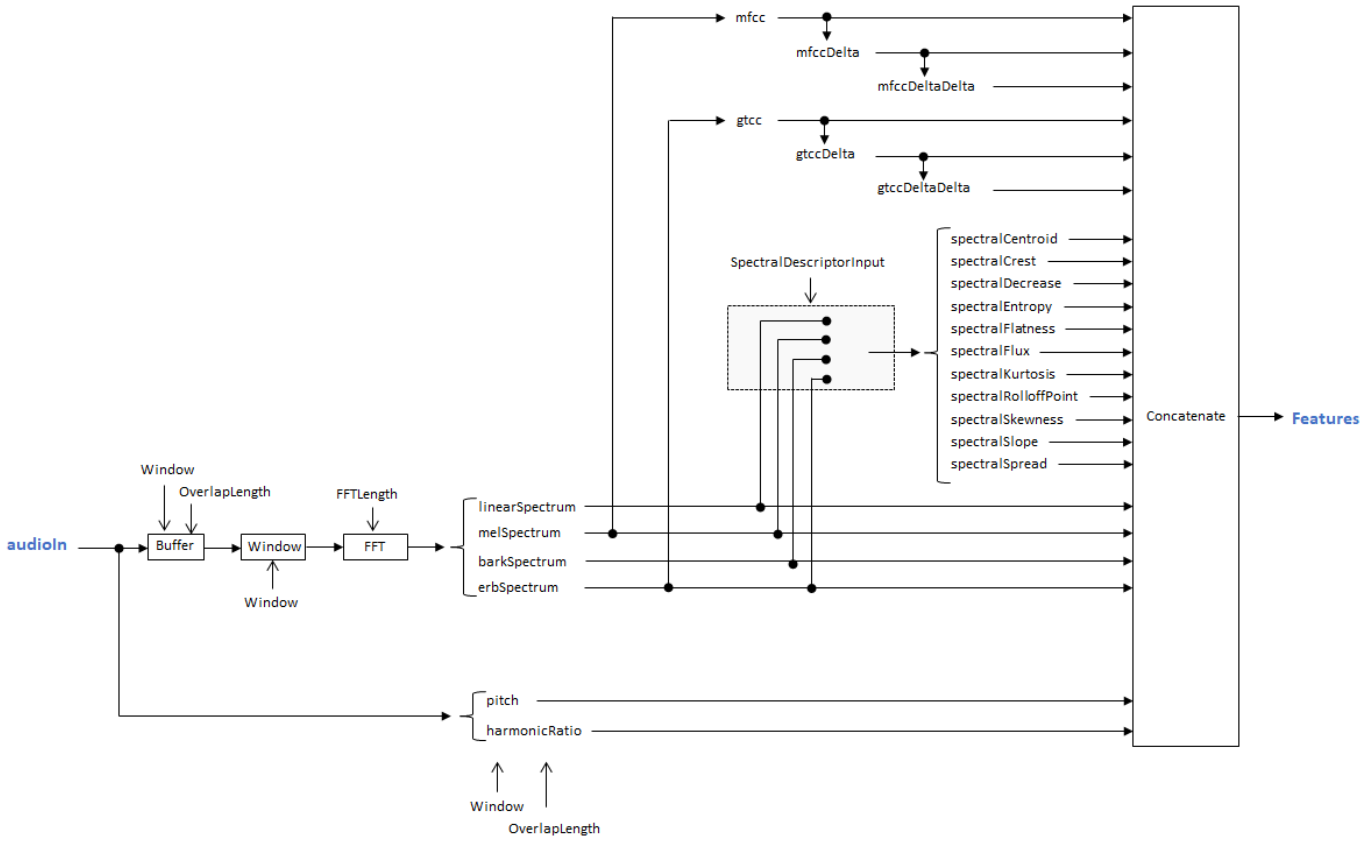
Plot the features extracted from the audio signal.

```
plotFeatures(afe,audioIn)
```



Algorithms

The `audioFeatureExtractor` creates a feature extraction pipeline based on your selected features. To reduce computations, `audioFeatureExtractor` reuses intermediary representations and outputs some intermediate representations as features.

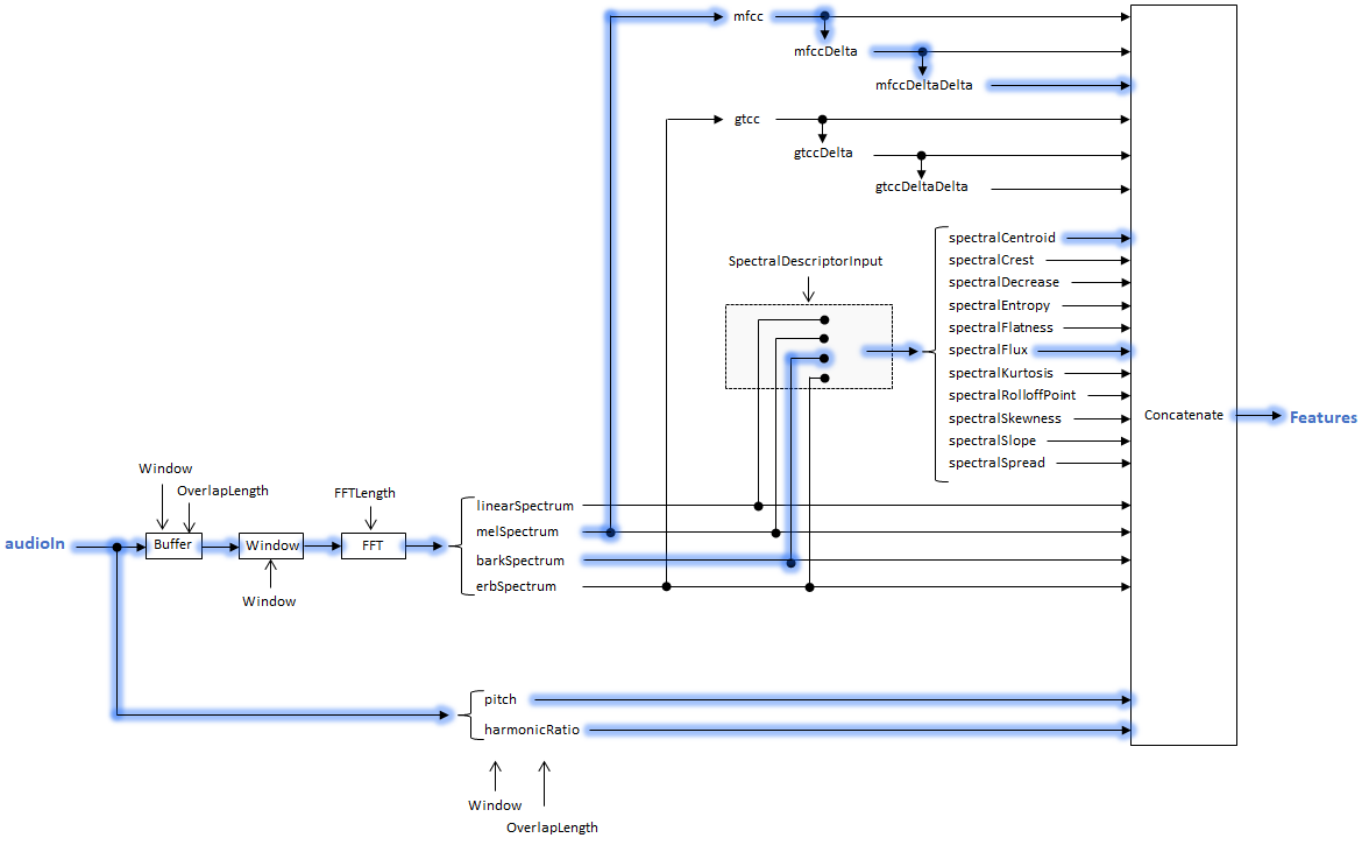


For example, to create an object that extracts the centroid of the Bark spectrum, the flux of the Bark spectrum, the pitch, the harmonic ratio, and the delta-delta of the MFCC, specify the `audioFeatureExtractor` as follows.

```
aFE = audioFeatureExtractor( ...
     SpectralDescriptorInput="barkSpectrum", ...
     spectralCentroid=true, ...
     spectralFlux=true, ...
     pitch=true, ...
     harmonicRatio=true, ...
     mfccDeltaDelta=true)
```

```
aFE =

  audioFeatureExtractor with properties:

   Properties
                  Window: [1024×1 double]
           OverlapLength: 512
              SampleRate: 44100
               FFTLength: []
    SpectralDescriptorInput: 'barkSpectrum'

   Enabled Features
     mfccDeltaDelta, spectralCentroid, spectralFlux, pitch, harmonicRatio

   Disabled Features
     linearSpectrum, melSpectrum, barkSpectrum, erbSpectrum, mfcc, mfccDelta
     gtcc, gtccDelta, gtccDeltaDelta, spectralCrest, spectralDecrease, spectralEntropy
     spectralFlatness, spectralKurtosis, spectralRolloffPoint, spectralSkewness, spectralSlope, spectralSpread


   To extract a feature, set the corresponding property to true.
   For example, obj.mfcc = true, adds mfcc to the list of enabled features.
```

This configuration corresponds to the highlighted feature extraction pipeline.



Note Because `audioFeatureExtractor` reuses intermediary representations, the features output from `audioFeatureExtractor` might not correspond with the default configuration of features output by corresponding individual feature extractors.

Version History

Introduced in R2019b

R2023a: Generate optimized C/C++ code for computing auditory spectrum

Functions returned by `generateMATLABFunction` that compute an auditory spectrum (mel, Bark, ERB) support optimized C/C++ code generation using single instruction, multiple data (SIMD) instructions.

R2022b: Visualize extracted features

Use the `plotFeatures` object function to visualize extracted audio features.

R2020b: Computation of deltas and delta-deltas

Behavior changed in R2020b

The `audioDelta` function is now used to compute `mfccDelta`, `mfccDeltaDelta`, `gtccDelta`, and `gtccDeltaDelta`. The `audioDelta` algorithm has a different startup behavior than the previous algorithm. The default window length used to compute the deltas has changed from 2 to 9. A delta window length of 2 is no longer supported.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- You cannot generate code directly from `audioFeatureExtractor`. You can generate C/C++ code from the function returned by `generateMATLABFunction`.
- Functions returned by `generateMATLABFunction` that compute an auditory spectrum (mel, Bark, ERB) support optimized code generation using single instruction, multiple data (SIMD) instructions. For more information about SIMD code generation, see “Generate SIMD Code for MATLAB Functions” (MATLAB Coder).
- `zerocrossrate` code generation does not support disabling dynamic memory allocation when the input is multichannel.

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This function fully supports GPU arrays. For more information, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

See Also

Extract Audio Features | `audioDatastore` | `audioDataAugmenter` | **Signal Labeler** | `vggishEmbeddings`

removeAugmentationMethod

Remove custom augmentation method

Syntax

```
removeAugmentationMethod(aug,algorithmName)
```

Description

`removeAugmentationMethod(aug,algorithmName)` removes the custom augmentation algorithm from an `audioDataAugmenter` object.

Examples

Remove Augmentation Method

Create a default `audioDataAugmenter` object.

```
aug = audioDataAugmenter
```

```
aug =
  audioDataAugmenter with properties:

               AugmentationMode: 'sequential'
    AugmentationParameterSource: 'random'
               NumAugmentations: 1
         TimeStretchProbability: 0.5000
             SpeedupFactorRange: [0.8000 1.2000]
          PitchShiftProbability: 0.5000
             SemitoneShiftRange: [-2 2]
       VolumeControlProbability: 0.5000
                VolumeGainRange: [-3 3]
            AddNoiseProbability: 0.5000
                        SNRRange: [0 10]
           TimeShiftProbability: 0.5000
                 TimeShiftRange: [-0.0050 0.0050]
```

Add a custom augmentation method that applies a random DC offset.

```
algorithmName = 'DCOffset';
algorithmHandle = @(x)x+rand(1,'like',x);
addAugmentationMethod(aug,algorithmName,algorithmHandle)
aug
```

```
aug =
  audioDataAugmenter with properties:

               AugmentationMode: 'sequential'
    AugmentationParameterSource: 'random'
               NumAugmentations: 1
```

```
          TimeStretchProbability: 0.5000
              SpeedupFactorRange: [0.8000 1.2000]
           PitchShiftProbability: 0.5000
              SemitoneShiftRange: [-2 2]
         VolumeControlProbability: 0.5000
                   VolumeGainRange: [-3 3]
            AddNoiseProbability: 0.5000
                        SNRRange: [0 10]
           TimeShiftProbability: 0.5000
                 TimeShiftRange: [-0.0050 0.0050]
            DCOffsetProbability: 0.5000
```

Remove the custom augmentation method.

```
removeAugmentationMethod(aug,algorithmName)
aug
```

```
aug =
  audioDataAugmenter with properties:

                AugmentationMode: 'sequential'
    AugmentationParameterSource: 'random'
                NumAugmentations: 1
          TimeStretchProbability: 0.5000
              SpeedupFactorRange: [0.8000 1.2000]
           PitchShiftProbability: 0.5000
              SemitoneShiftRange: [-2 2]
         VolumeControlProbability: 0.5000
                   VolumeGainRange: [-3 3]
            AddNoiseProbability: 0.5000
                        SNRRange: [0 10]
           TimeShiftProbability: 0.5000
                 TimeShiftRange: [-0.0050 0.0050]
```

Input Arguments

aug — Audio data augmenter
`audioDataAugmenter` object

`audioDataAugmenter` object.

`algorithmName` — Algorithm name
character vector | string

Algorithm name, specified as a character vector or string. `algorithmName` must match the algorithm name you used to add the algorithm using `addAugmentationMethod`.

Data Types: `char` | `string`

Version History

Introduced in R2019b

See Also

`addAugmentationMethod` | `audioDataAugmenter`

augment

Augment audio data

Syntax

```
data = augment(aug,audioIn)
data = augment(aug,audioIn,fs)
```

Description

`data = augment(aug,audioIn)` returns a table containing augmented audio data and information about the augmentation applied.

`data = augment(aug,audioIn,fs)` specifies the sample rate of the audio input.

Examples

Apply Random Sequential Augmentations

Read in an audio signal and listen to it.

```
[audioIn,fs] = audioread("Counting-16-44p1-mono-15secs.wav");
sound(audioIn,fs)
```

Create an `audioDataAugmenter` object that applies time stretching, volume control, and time shifting in cascade. Apply each of the augmentations with 80% probability. Set `NumAugmentations` to `5` to output five independently augmented signals. To skip pitch shifting and noise addition for each augmentation, set the respective probabilities to `0`. Define parameter ranges for each relevant augmentation algorithm.

```
augmenter = audioDataAugmenter( ...
    "AugmentationMode","sequential", ...
    "NumAugmentations",5, ...
    ...
    "TimeStretchProbability",0.8, ...
    "SpeedupFactorRange", [1.3,1.4], ...
    ...
    "PitchShiftProbability",0, ...
    ...
    "VolumeControlProbability",0.8, ...
    "VolumeGainRange",[-5,5], ...
    ...
    "AddNoiseProbability",0, ...
    ...
    "TimeShiftProbability",0.8, ...
    "TimeShiftRange", [-500e-3,500e-3])
```

```
augmenter =
  audioDataAugmenter with properties:

               AugmentationMode: "sequential"
```

```
    AugmentationParameterSource: 'random'
               NumAugmentations: 5
         TimeStretchProbability: 0.8000
             SpeedupFactorRange: [1.3000 1.4000]
          PitchShiftProbability: 0
       VolumeControlProbability: 0.8000
                VolumeGainRange: [-5 5]
           AddNoiseProbability: 0
          TimeShiftProbability: 0.8000
                TimeShiftRange: [-0.5000 0.5000]
```

Call `augment` on the audio to create 5 augmentations. The augmented audio is returned in a table with variables `Audio` and `AugmentationInfo`. The number of rows in the table is defined by `NumAugmentations`.

```
data = augment(augmenter,audioIn,fs)
```

```
data=5×2 table
          Audio           AugmentationInfo
    ____________________    ________________

    {685056x1 double}       1x1 struct
    {685056x1 double}       1x1 struct
    {505183x1 double}       1x1 struct
    {685056x1 double}       1x1 struct
    {490728x1 double}       1x1 struct
```

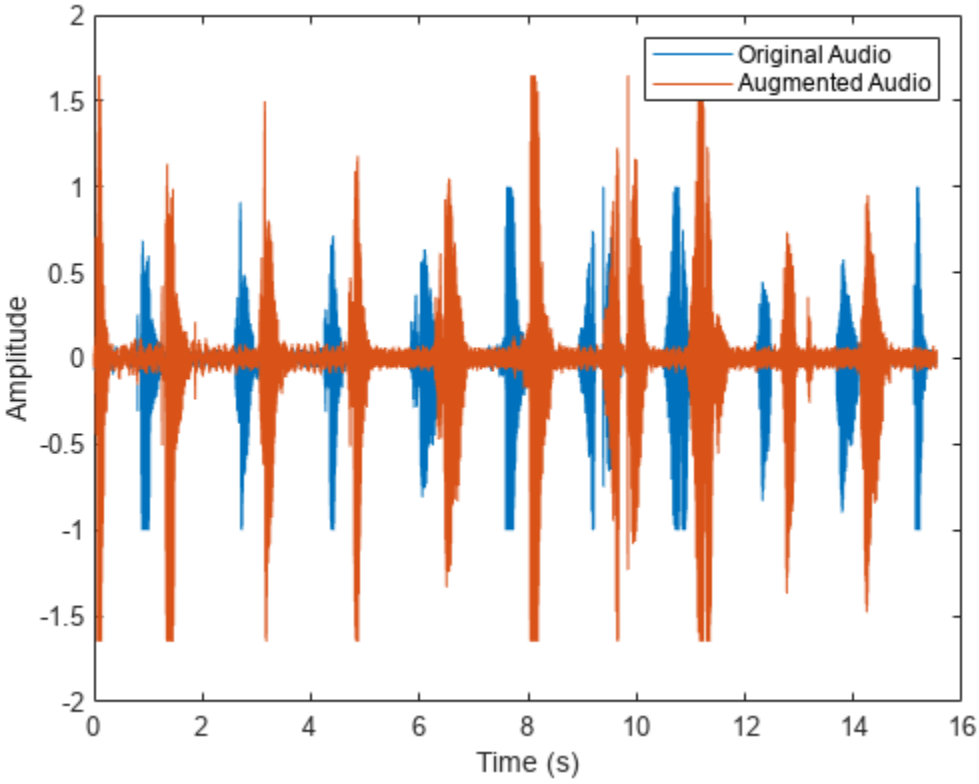
In the current augmentation pipeline, augmentation parameters are assigned randomly from within the specified ranges. To determine the exact parameters used for an augmentation, inspect `AugmentationInfo`.

```
augmentationToInspect = 4;
data.AugmentationInfo(augmentationToInspect)
```

```
ans = struct with fields:
    SpeedupFactor: 1
       VolumeGain: 4.3399
        TimeShift: 0.4502
```

Listen to the augmentation you are inspecting. Plot time representation of the original and augmented signals.

```
augmentation = data.Audio{augmentationToInspect};
sound(augmentation,fs)

t = (0:(numel(audioIn)-1))/fs;
taug = (0:(numel(augmentation)-1))/fs;
plot(t,audioIn,taug,augmentation)
legend("Original Audio","Augmented Audio")
ylabel("Amplitude")
xlabel("Time (s)")
```



Apply Specified Sequential Augmentations

Read in an audio signal and listen to it.

```
[audioIn,fs] = audioread("Counting-16-44p1-mono-15secs.wav");
sound(audioIn,fs)
```

Create an `audioDataAugmenter` object that applies time stretching, pitch shifting, and noise corruption in cascade. Specify the time stretch speedup factors as `0.9`, `1.1`, and `1.2`. Specify the pitch shifting in semitones as `-2`, `-1`, `1`, and `2`. Specify the noise corruption SNR as `10` dB and `15` dB.

```
augmenter = audioDataAugmenter( ...
    "AugmentationMode","sequential", ...
    "AugmentationParameterSource","specify", ...
    "SpeedupFactor",[0.9,1.1,1.2], ...
    "ApplyTimeStretch",true, ...
    "ApplyPitchShift",true, ...
    "SemitoneShift",[-2,-1,1,2], ...
    "SNR",[10,15], ...
    "ApplyVolumeControl",false, ...
    "ApplyTimeShift",false)
```

```
augmenter =
  audioDataAugmenter with properties:
```

```
            AugmentationMode: "sequential"
    AugmentationParameterSource: "specify"
            ApplyTimeStretch: 1
               SpeedupFactor: [0.9000 1.1000 1.2000]
             ApplyPitchShift: 1
               SemitoneShift: [-2 -1 1 2]
          ApplyVolumeControl: 0
               ApplyAddNoise: 1
                         SNR: [10 15]
              ApplyTimeShift: 0
```

Call `augment` on the audio to create 24 augmentations. The augmentations represent every combination of the specified augmentation parameters ($3 \times 4 \times 2 = 24$).

```
data = augment(augmenter,audioIn,fs)
```

```
data=24×2 table
          Audio           AugmentationInfo
    ____________________    ________________

    {761243x1 double}       1x1 struct
    {622888x1 double}       1x1 struct
    {571263x1 double}       1x1 struct
    {761243x1 double}       1x1 struct
    {622888x1 double}       1x1 struct
    {571263x1 double}       1x1 struct
    {761243x1 double}       1x1 struct
    {622888x1 double}       1x1 struct
    {571263x1 double}       1x1 struct
    {761243x1 double}       1x1 struct
    {622888x1 double}       1x1 struct
    {571263x1 double}       1x1 struct
    {761243x1 double}       1x1 struct
    {622888x1 double}       1x1 struct
    {571263x1 double}       1x1 struct
    {761243x1 double}       1x1 struct
      ⋮
```

You can check the parameter configuration of each augmentation using the `AugmentationInfo` table variable.

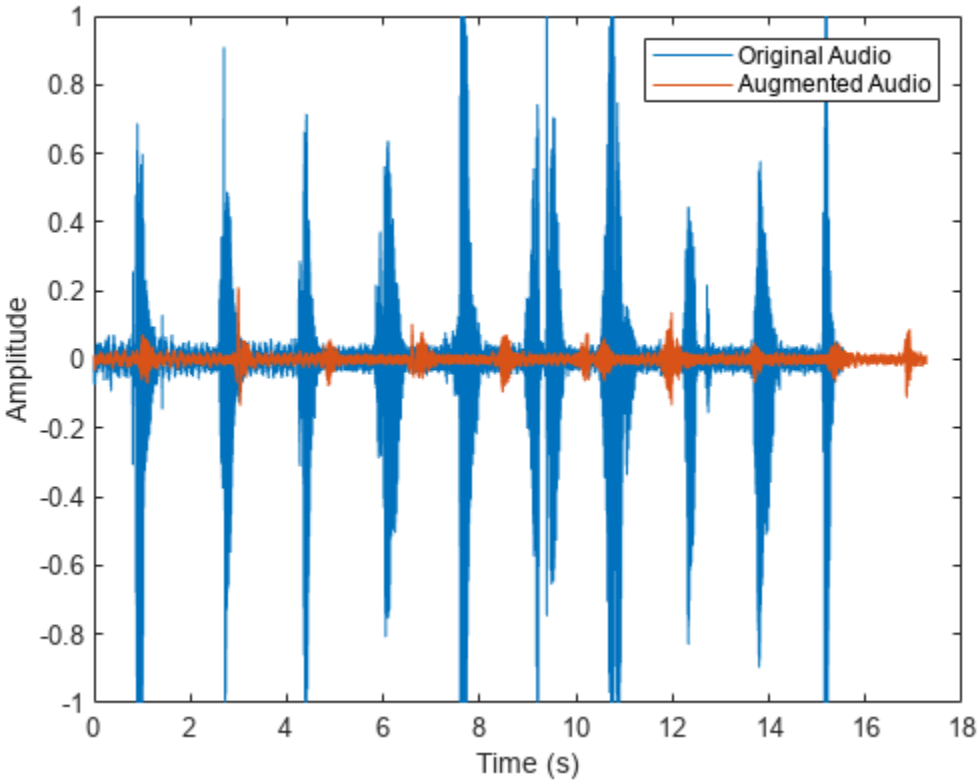
```
augmentationToInspect = 1;
data.AugmentationInfo(augmentationToInspect)
```

```
ans = struct with fields:
    SpeedupFactor: 0.9000
    SemitoneShift: -2
              SNR: 10
```

Listen to the augmentation you are inspecting. Plot the time-domain representation of the original and augmented signals.

```
augmentation = data.Audio{augmentationToInspect};
sound(augmentation,fs)
```

```
t = (0:(numel(audioIn)-1))/fs;
taug = (0:(numel(augmentation)-1))/fs;
plot(t,audioIn,taug,augmentation)
legend("Original Audio","Augmented Audio")
ylabel("Amplitude")
xlabel("Time (s)")
```



Apply Random Independent Augmentations

Read in an audio signal and listen to it.

```
[audioIn,fs] = audioread("Counting-16-44p1-mono-15secs.wav");
```

Create an `audioDataAugmenter` object that applies noise corruption, and time shifting in parallel branches. For the noise corruption branch, randomly apply noise with an SNR in the range `0` dB to `20` dB. For the time shifting branch, randomly apply time shifting in the range -`300` ms to `300` ms. Apply augmentation 2 times for each branch, for 4 total augmentations.

```
augmenter = audioDataAugmenter( ...
    "AugmentationMode","independent", ...
    "AugmentationParameterSource","random", ...
    "NumAugmentations",2, ...
    "ApplyTimeStretch",false, ...
    "ApplyPitchShift",false, ...
```

```
    "ApplyVolumeControl",false, ...
    "SNRRange",[0,20], ...
    "TimeShiftRange",[-300e-3,300e-3])

augmenter =
  audioDataAugmenter with properties:

              AugmentationMode: "independent"
    AugmentationParameterSource: "random"
              NumAugmentations: 2
              ApplyTimeStretch: 0
               ApplyPitchShift: 0
             ApplyVolumeControl: 0
                  ApplyAddNoise: 1
                       SNRRange: [0 20]
                 ApplyTimeShift: 1
                 TimeShiftRange: [-0.3000 0.3000]
```

Call `augment` on the audio to create 3 augmentations.

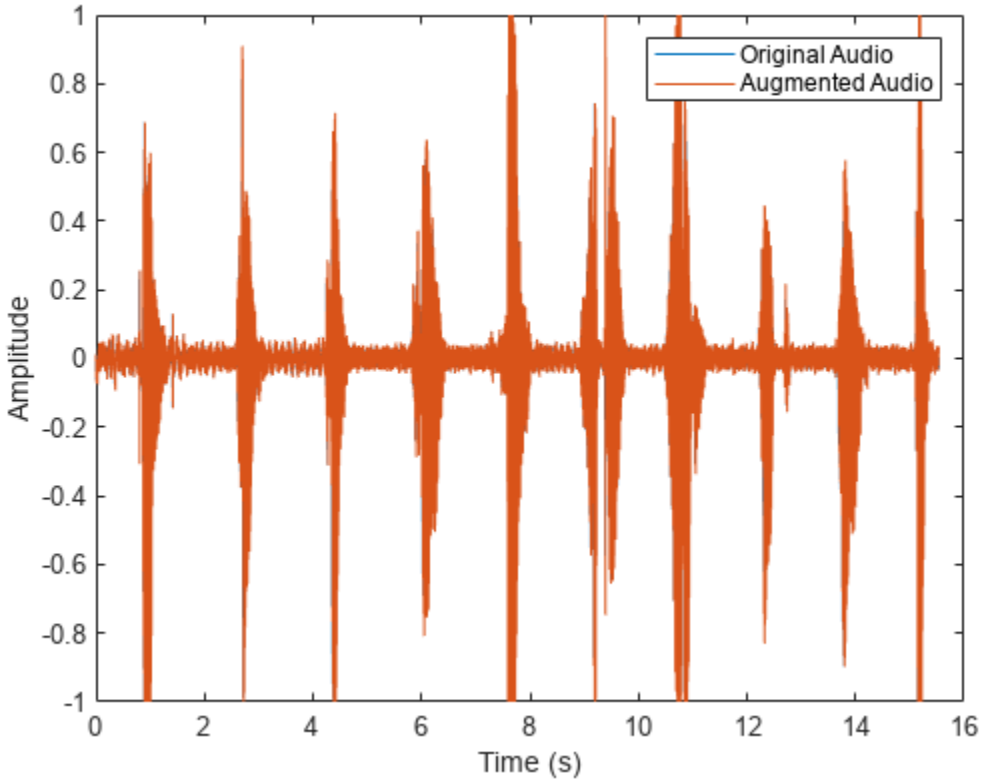
```
data = augment(augmenter,audioIn,fs);
```

You can check the parameter configuration of each augmentation using the `AugmentatioInfo` table variable.

```
augmentationToInspect = 4;
data.AugmentationInfo{augmentationToInspect}

ans = struct with fields:
    TimeShift: 0.0016
```

Listen to the audio you are inspecting. Plot the time-domain representation of the original and augmented signals.

```
augmentation = data.Audio{augmentationToInspect};
sound(augmentation,fs)

t = (0:(numel(audioIn)-1))/fs;
taug = (0:(numel(augmentation)-1))/fs;
plot(t,audioIn,taug,augmentation)
legend("Original Audio","Augmented Audio")
ylabel("Amplitude")
xlabel("Time (s)")
```



Apply Specified Independent Augmentations

Read in an audio signal and listen to it.

```
[audioIn,fs] = audioread("Counting-16-44p1-mono-15secs.wav");
```

Create an `audioDataAugmenter` object that applies volume control, noise corruption, and time shifting in parallel branches.

```
augmenter = audioDataAugmenter( ...
    "AugmentationMode","independent", ...
    "AugmentationParameterSource","specify", ...
    "ApplyTimeStretch",false, ...
    "ApplyPitchShift",false, ...
    "VolumeGain",2, ...
    "SNR",0, ...
    "TimeShift",2)
```

```
augmenter =
  audioDataAugmenter with properties:

                AugmentationMode: "independent"
    AugmentationParameterSource: "specify"
                ApplyTimeStretch: 0
                 ApplyPitchShift: 0
```

```
    ApplyVolumeControl: 1
           VolumeGain: 2
        ApplyAddNoise: 1
                  SNR: 0
       ApplyTimeShift: 1
            TimeShift: 2
```

Call `augment` on the audio to create 3 augmentations.

```
data = augment(augmenter,audioIn,fs)
```

```
data=3×2 table
          Audio          AugmentationInfo
    ____________________    ________________

    {685056x1 double}      {1x1 struct}
    {685056x1 double}      {1x1 struct}
    {685056x1 double}      {1x1 struct}
```

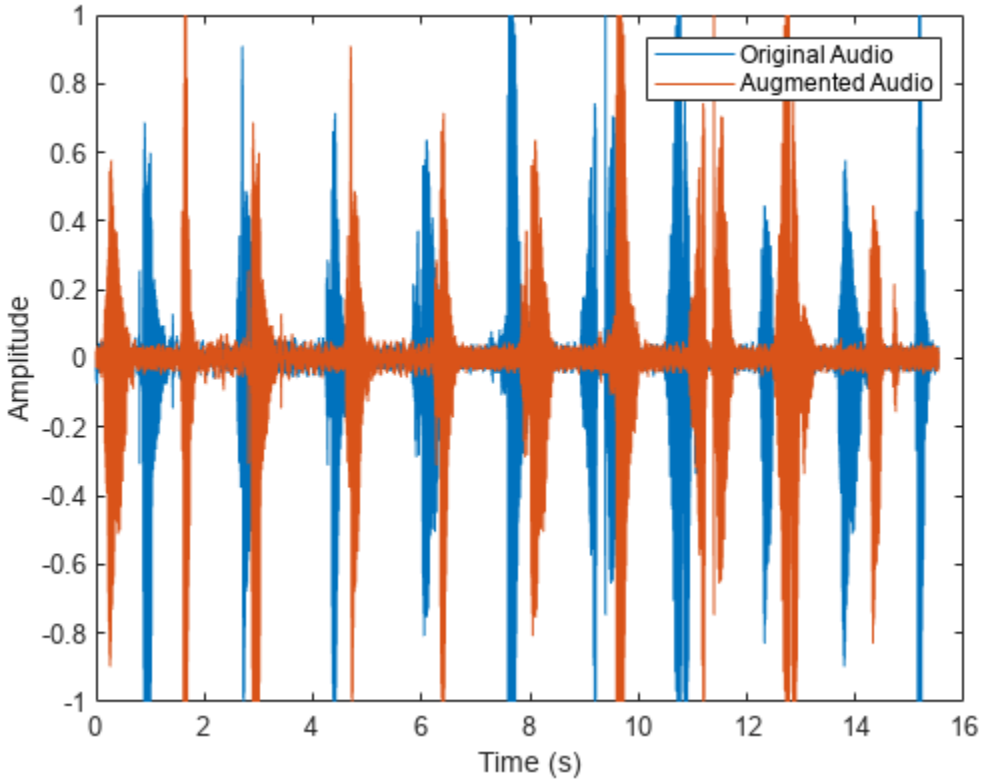
You can check the parameter configuration of each augmentation using the `AugmentatioInfo` table variable.

```
augmentationToInspect = 3;
data.AugmentationInfo{augmentationToInspect}
```

```
ans = struct with fields:
    TimeShift: 2
```

Listen to the audio you are inspecting. Plot the time-domain representations of the original and augmented signals.

```
augmentation = data.Audio{augmentationToInspect};
sound(augmentation,fs)

t = (0:(numel(audioIn)-1))/fs;
taug = (0:(numel(augmentation)-1))/fs;
plot(t,audioIn,taug,augmentation)
legend("Original Audio","Augmented Audio")
ylabel("Amplitude")
xlabel("Time (s)")
```



Augment Audio Dataset

The `audioDataAugmenter` supports multiple workflows for augmenting your datastore, including:

- Offline augmentation
- Augmentation using tall arrays
- Augmentation using transform datastores

In each workflow, begin by creating an audio datastore to point to your audio data. In this example, you create an audio datastore that points to audio samples included with Audio Toolbox™. Count the number of files in the dataset.

```
folder = fullfile(matlabroot,"toolbox","audio","samples");
ADS = audioDatastore(folder)
```

```
ADS =
  audioDatastore with properties:

                      Files: {
                             ' ...\matlab\toolbox\audio\samples\Ambiance-16-44p1-mono-12secs.wav
                             ' ...\matlab\toolbox\audio\samples\AudioArray-16-16-4channels-20sec
                             ' ...\toolbox\audio\samples\ChurchImpulseResponse-16-44p1-mono-5sec
                              ... and 26 more
                             }
```

```
    AlternateFileSystemRoots: {}
              OutputDataType: 'double'
                      Labels: {}
```

```
numFilesInDataset = numel(ADS.Files)
```

```
numFilesInDataset = 29
```

Create an `audioDataAugmenter` that applies random sequential augmentations. Set `NumAugmentations` to 2.

```
aug = audioDataAugmenter('NumAugmentations',2)
```

```
aug =
  audioDataAugmenter with properties:

               AugmentationMode: 'sequential'
    AugmentationParameterSource: 'random'
               NumAugmentations: 2
         TimeStretchProbability: 0.5000
             SpeedupFactorRange: [0.8000 1.2000]
          PitchShiftProbability: 0.5000
             SemitoneShiftRange: [-2 2]
       VolumeControlProbability: 0.5000
                VolumeGainRange: [-3 3]
            AddNoiseProbability: 0.5000
                        SNRRange: [0 10]
           TimeShiftProbability: 0.5000
                 TimeShiftRange: [-0.0050 0.0050]
```

Offline Augmentation

To augment the audio dataset, create two augmentations of each file and then write the augmentations as WAV files.

```
while hasdata(ADS)
    [audioIn,info] = read(ADS);

    data = augment(aug,audioIn,info.SampleRate);

    [~,fn] = fileparts(info.FileName);
    for i = 1:size(data,1)
        augmentedAudio = data.Audio{i};

        % If augmentation caused an audio signal to have values outside of -1 and 1,
        % normalize the audio signal to avoid clipping when writing.
        if max(abs(augmentedAudio),[],'all')>1
            augmentedAudio = augmentedAudio/max(abs(augmentedAudio),[],'all');
        end

        audiowrite(sprintf('%s_aug%d.wav',fn,i),augmentedAudio,info.SampleRate)
    end
end
```

Create an `audioDatastore` that points to the augmented dataset and confirm that the number of files in the dataset is double the original number of files.

```
augmentedADS = audioDatastore(pwd)

augmentedADS =
  audioDatastore with properties:

                      Files: {
                             ' ...\Examples\audio-ex28074079\Ambiance-16-44p1-mono-12secs_aug1.w
                             ' ...\Examples\audio-ex28074079\Ambiance-16-44p1-mono-12secs_aug2.w
                             ' ...\Examples\audio-ex28074079\AudioArray-16-16-4channels-20secs_a
                              ... and 55 more
                             }
    AlternateFileSystemRoots: {}
              OutputDataType: 'double'
                      Labels: {}


numFilesInAugmentedDataset = numel(augmentedADS.Files)

numFilesInAugmentedDataset = 58
```

Augment Using Tall Arrays

When augmenting a dataset using tall arrays, the input data to the augmenter should be sampled at a consistent rate. Subset the original audio dataset to only include files with a sample rate of 44.1 kHz. Most datasets are already cleaned to have a consistent sample rate.

```
keepFile = cellfun(@(x)contains(x,'44p1'),ADS.Files);
ads44p1 = subset(ADS,keepFile);
fs = 44.1e3;
```

Convert the audio datastore to a tall array. `tall` arrays are evaluated only when you request them explicitly using `gather`. MATLAB® automatically optimizes the queued calculations by minimizing the number of passes through the data. If you have the Parallel Computing Toolbox™, you can spread the calculations across multiple machines. The audio data is represented as an *M*-by-1 tall cell array, where *M* is the number of files in the audio datastore.

```
adsTall = tall(ads44p1)

Starting parallel pool (parpool) using the 'local' profile ...
Connected to the parallel pool (number of workers: 6).

adsTall =

  M×1 tall cell array

    { 539648×1 double}
    { 227497×1 double}
    {   8000×1 double}
    { 685056×1 double}
    { 882688×2 double}
    {1115760×2 double}
    { 505200×2 double}
    {3195904×2 double}
        :         :
        :         :
```

Define a `cellfun` function so that augmentation is applied to each cell of the tall array. Call `gather` to evaluate the tall array.

```
augTall = cellfun(@(x)augment(aug,x,fs),adsTall,"UniformOutput",false);
augmentedDataset = gather(augTall)
```

```
Evaluating tall expression using the Parallel Pool 'local':
- Pass 1 of 1: Completed in 1 min 34 sec
Evaluation completed in 1 min 34 sec

augmentedDataset=12×1 cell array
    {2×2 table}
    {2×2 table}
    {2×2 table}
    {2×2 table}
    {2×2 table}
    {2×2 table}
    {2×2 table}
    {2×2 table}
    {2×2 table}
    {2×2 table}
    {2×2 table}
    {2×2 table}
```

The augmented dataset is returned as a *numFiles*-by-1 cell array, where *numFiles* is the number of files in the datastore. Each element of the cell array is a *numAugmentationsPerFile*-by-2 table, where *numAugmentationsPerFile* is the number of augmentations returned per file.

```
numFiles = numel(augmentedDataset)
```

```
numFiles = 12
```

```
numAugmentationsPerFile = size(augmentedDataset{1},1)
```

```
numAugmentationsPerFile = 2
```

Augment Using Transform Datastore

You can perform online data augmentation while you train your machine learning application using a transform datastore. Call `transform` to create a new datastore that applies data augmentation while reading.

```
transformADS = transform(ADS,@(x,info)augment(aug,x,info),'IncludeInfo',true)
```

```
transformADS =
  TransformedDatastore with properties:

    UnderlyingDatastore: [1×1 audioDatastore]
             Transforms: {@(x,info)augment(aug,x,info)}
            IncludeInfo: 1
```

Call `read` to return the augmented first file from the transform datastore.

```
augmentedRead = read(transformADS)
```

```
augmentedRead=2×2 table
          Audio          AugmentationInfo
    _________________    ________________

    {539648×1 double}      [1×1 struct]
```

```
    {586683×1 double}      [1×1 struct]
```

Input Arguments

aug — Audio data augmenter

`audioDataAugmenter` object

`audioDataAugmenter` object.

audioIn — Audio input

vector | matrix

Audio input, specified as a column vector or matrix of independent channels (columns).

Data Types: `single` | `double`

fs — Sample rate (Hz)

`44100` (default) | positive scalar

Sample rate in Hz, specified as a positive scalar. The allowable range of `fs` depends on the properties of the `audioDataAugmenter` object.

Data Types: `single` | `double`

Output Arguments

data — Augmented audio and augmentation information

table

Augmented audio and augmentation information, returned as a two-column `table`. The first column holds the augmented audio signal. The second column holds information about the applied augmentation methods. The number of rows in `data` corresponds to the number of output augmented signals. The number of output augmented signals depends on the property values of the object.

Version History

Introduced in R2019b

See Also

`audioDataAugmenter` | `addAugmentationMethod` | `removeAugmentationMethod` | `table`

addAugmentationMethod

Add custom augmentation method

Syntax

```
addAugmentationMethod(aug,algorithmName,algorithmHandle)
addAugmentationMethod(aug,algorithmName,algorithmHandle,Name,Value)
```

Description

`addAugmentationMethod(aug,algorithmName,algorithmHandle)` adds a custom augmentation algorithm to an `audioDataAugmenter` object.

`addAugmentationMethod(aug,algorithmName,algorithmHandle,Name,Value)` specifies options using one or more `Name,Value` pair arguments.

Examples

Add Custom Augmentation Method

You can expand the capabilities of `audioDataAugmenter` by adding custom augmentation methods.

Read in an audio signal and listen to it.

```
[audioIn,fs] = audioread('Counting-16-44p1-mono-15secs.wav');
sound(audioIn,fs)
```

Create an `audioDataAugmenter` object. Set the probability of applying white noise to `0`.

```
augmenter = audioDataAugmenter('AddNoiseProbability',0)
```

```
augmenter =
  audioDataAugmenter with properties:

               AugmentationMode: 'sequential'
    AugmentationParameterSource: 'random'
               NumAugmentations: 1
         TimeStretchProbability: 0.5000
             SpeedupFactorRange: [0.8000 1.2000]
          PitchShiftProbability: 0.5000
             SemitoneShiftRange: [-2 2]
       VolumeControlProbability: 0.5000
                VolumeGainRange: [-3 3]
            AddNoiseProbability: 0
          TimeShiftProbability: 0.5000
                TimeShiftRange: [-0.0050 0.0050]
```

Specify a custom augmentation algorithm that applies pink noise. The `AddPinkNoise` algorithm is added to the `augmenter` properties.

```
algorithmName = 'AddPinkNoise';
algorithmHandle = @(x)x+pinknoise(size(x),'like',x);
addAugmentationMethod(augmenter,algorithmName,algorithmHandle)

augmenter
```

```
augmenter =
  audioDataAugmenter with properties:

               AugmentationMode: 'sequential'
    AugmentationParameterSource: 'random'
               NumAugmentations: 1
         TimeStretchProbability: 0.5000
             SpeedupFactorRange: [0.8000 1.2000]
          PitchShiftProbability: 0.5000
             SemitoneShiftRange: [-2 2]
       VolumeControlProbability: 0.5000
                VolumeGainRange: [-3 3]
            AddNoiseProbability: 0
           TimeShiftProbability: 0.5000
                 TimeShiftRange: [-0.0050 0.0050]
        AddPinkNoiseProbability: 0.5000
```

Set the probability of adding pink noise to 1.

```
augmenter.AddPinkNoiseProbability = 1
```

```
augmenter =
  audioDataAugmenter with properties:

               AugmentationMode: 'sequential'
    AugmentationParameterSource: 'random'
               NumAugmentations: 1
         TimeStretchProbability: 0.5000
             SpeedupFactorRange: [0.8000 1.2000]
          PitchShiftProbability: 0.5000
             SemitoneShiftRange: [-2 2]
       VolumeControlProbability: 0.5000
                VolumeGainRange: [-3 3]
            AddNoiseProbability: 0
           TimeShiftProbability: 0.5000
                 TimeShiftRange: [-0.0050 0.0050]
        AddPinkNoiseProbability: 1
```

Augment the original signal and listen to the result. Inspect parameters of the augmentation algorithms applied.

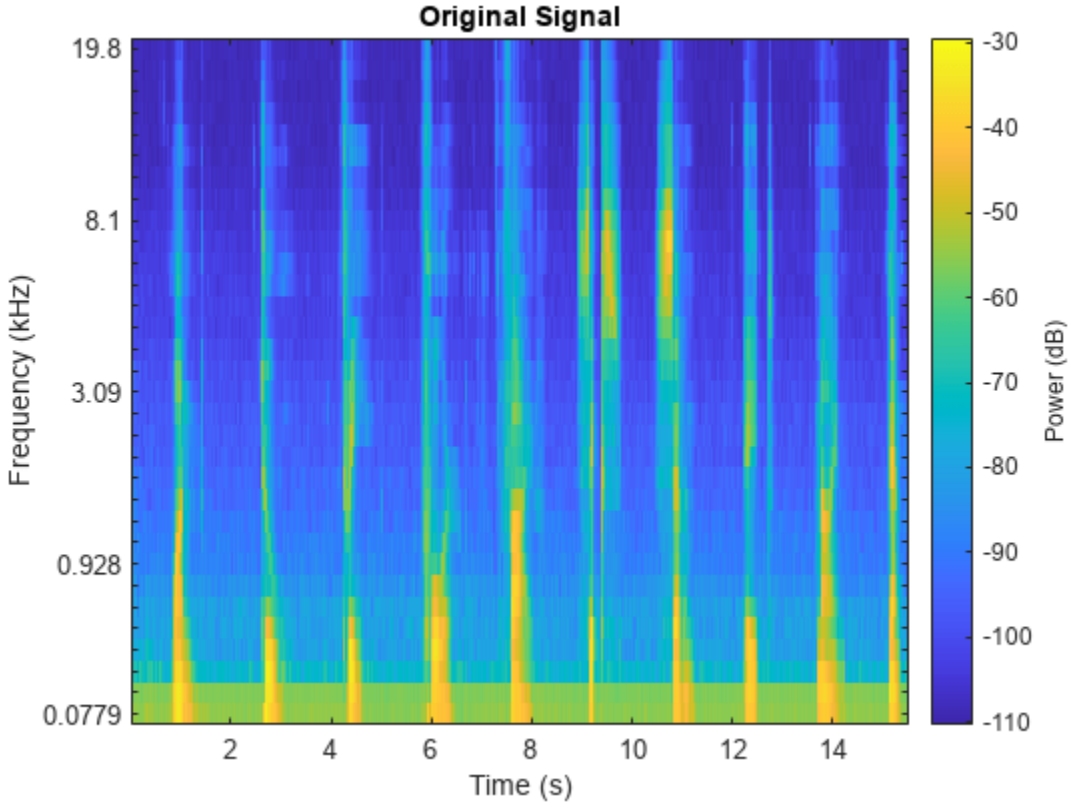
```
data = augment(augmenter,audioIn,fs);
sound(data.Audio{1},fs)

data.AugmentationInfo(1)
```

```
ans = struct with fields:
    SpeedupFactor: 1
    SemitoneShift: 0
       VolumeGain: 2.4803
        TimeShift: -0.0022
```

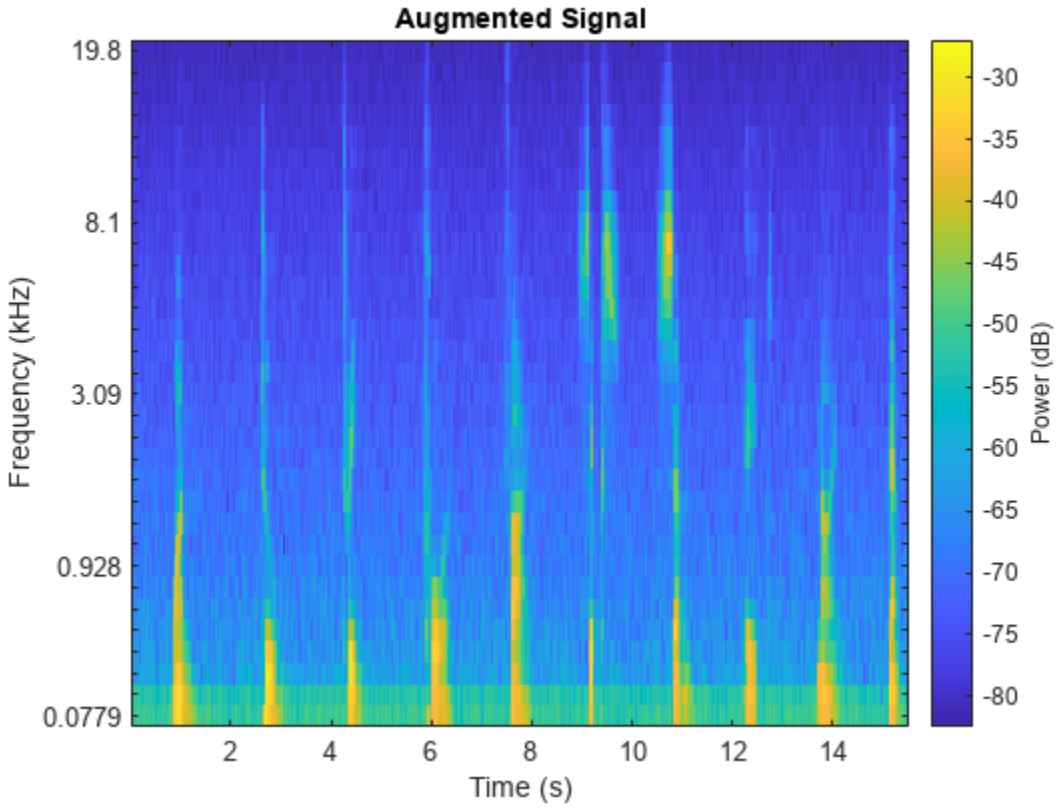
```
    AddPinkNoise: 'Applied'
```

Plot the mel spectrograms of the original and augmented signals.

```
melSpectrogram(audioIn,fs)
title('Original Signal')
```




```
melSpectrogram(data.Audio{1},fs)
title('Augmented Signal')
```



Specify Parameters of Custom Augmentation Method

In this example, you add a custom augmentation method that applies median filtering to your audio.

Read in an audio signal and listen to it.

```
[audioIn,fs] = audioread("Counting-16-44p1-mono-15secs.wav");
sound(audioIn,fs)
```

Create a random sequential augmenter that adds noise with an SNR range of 5 dB to 10 dB. Set the probability of applying volume control, time stretching, pitch shifting, and time shifting to `0`. Set `NumAugmentations` to 4 to create 4 separate augmentations.

```
aug = audioDataAugmenter('NumAugmentations',4, ...
    "AddNoiseProbability",1, ...
    "SNRRange",[5,10], ...
    "VolumeControlProbability",0, ...
    "TimeStretchProbability",0, ...
    "TimeShiftProbability",0, ...
    "PitchShiftProbability",0)
```

```
aug =
  audioDataAugmenter with properties:

               AugmentationMode: 'sequential'
```

```
    AugmentationParameterSource: 'random'
                NumAugmentations: 4
          TimeStretchProbability: 0
           PitchShiftProbability: 0
       VolumeControlProbability: 0
            AddNoiseProbability: 1
                        SNRRange: [5 10]
           TimeShiftProbability: 0
```

Call `addAugmentationMethod` with an algorithm name and function handle. Specify the algorithm name as `MedianFilter` and the function handle as `movmedian` with a 3-element window length. The augmentation is added to the properties of your `audioDataAugmenter` object.

```
algorithmName = 'MedianFilter';
algorithmHandle = @(x)(movmedian(x,100));
addAugmentationMethod(aug,algorithmName,algorithmHandle)

aug
```

```
aug =
  audioDataAugmenter with properties:

                AugmentationMode: 'sequential'
    AugmentationParameterSource: 'random'
                NumAugmentations: 4
          TimeStretchProbability: 0
           PitchShiftProbability: 0
       VolumeControlProbability: 0
            AddNoiseProbability: 1
                        SNRRange: [5 10]
           TimeShiftProbability: 0
        MedianFilterProbability: 0.5000
```

Set the probability of applying median filtering to 80%.

```
aug.MedianFilterProbability = 0.8
```

```
aug =
  audioDataAugmenter with properties:

                AugmentationMode: 'sequential'
    AugmentationParameterSource: 'random'
                NumAugmentations: 4
          TimeStretchProbability: 0
           PitchShiftProbability: 0
       VolumeControlProbability: 0
            AddNoiseProbability: 1
                        SNRRange: [5 10]
           TimeShiftProbability: 0
        MedianFilterProbability: 0.8000
```

Call `augment` on the audio to create 4 augmentations.

```
data = augment(aug,audioIn,fs);
```

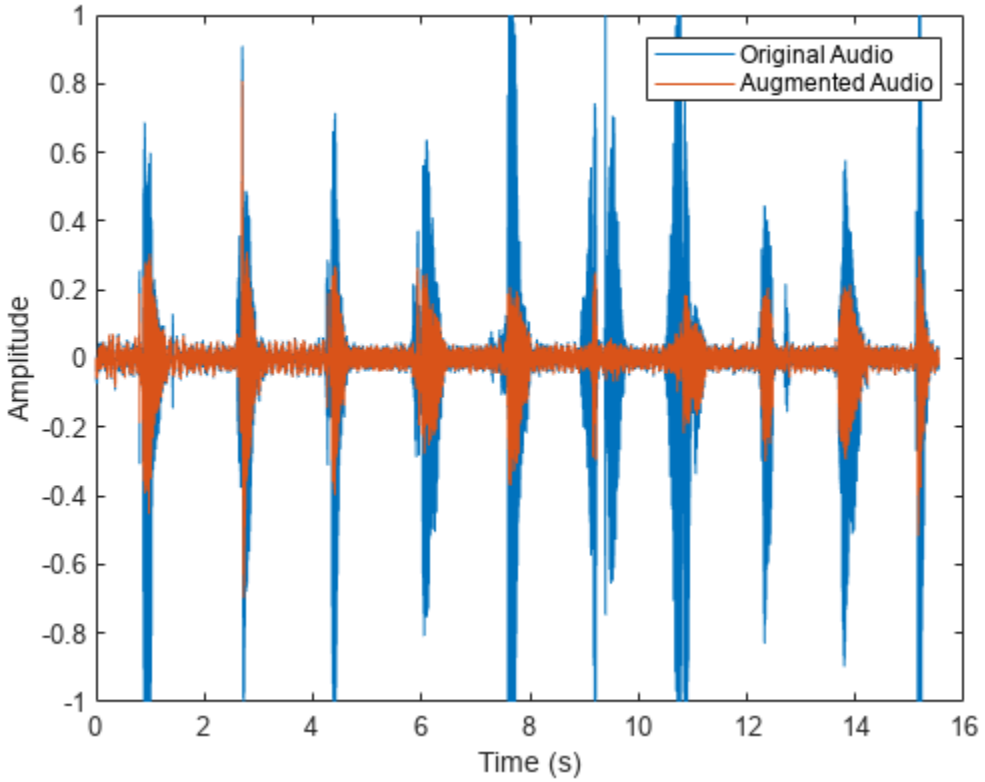
You can check the parameter configuration of each augmentation using the `AugmentationInfo` table variable. If median filtering was applied for an augmentation, then `AugmentationInfo` lists the parameter as `'Applied'`. If median filtering was not applied for an augmentation, then `AugmentationInfo` lists the parameter as `'Bypassed'`.

```
augmentationToInspect = 2;
data.AugmentationInfo(augmentationToInspect)
```

```
ans = struct with fields:
              SNR: 9.5787
    MedianFilter: 'Applied'
```

Listen to the audio you are inspecting. Plot the time-domain representation of the original and augmented signals.

```
augmentation = data.Audio{augmentationToInspect};
sound(augmentation,fs)
t = (0:(numel(audioIn)-1))/fs;
taug = (0:(numel(augmentation)-1))/fs;
plot(t,audioIn,taug,augmentation)
legend("Original Audio","Augmented Audio")
ylabel("Amplitude")
xlabel("Time (s)")
```



You can specify additional parameters and corresponding parameter ranges (for use when `AugmentationParameterSource` is set to `'random'`) and parameter values (for use when

`AugmentationParameterSource` is set to `'specify'`). You must specify additional parameters, parameter ranges, and parameter values during your call to `addAugmentationMethod`.

Call `removeAugmentationMethod` to remove the `MedianFilter` augmentation method. Call `addAugmentationMethod` again, this time specifying an augmentation parameter, parameter range, and parameter value. The augmentation and parameter range is added to the properties of your `audioDataAugmenter` object.

```
removeAugmentationMethod(aug,'MedianFilter')

algorithmName = 'MedianFilter';
augmentationParameter = 'MedianFilterWindowLength';
parameterRange = [1,200];
parameterValue = 100;

algorithmHandle = @(x,k)(movmedian(x,k));
addAugmentationMethod(aug,algorithmName,algorithmHandle, ...
    'AugmentationParameter',augmentationParameter, ...
    'ParameterRange',parameterRange, ...
    'ParameterValue',parameterValue)

aug
```

```
aug =
  audioDataAugmenter with properties:

                AugmentationMode: 'sequential'
      AugmentationParameterSource: 'random'
                NumAugmentations: 4
           TimeStretchProbability: 0
            PitchShiftProbability: 0
         VolumeControlProbability: 0
              AddNoiseProbability: 1
                         SNRRange: [5 10]
             TimeShiftProbability: 0
          MedianFilterProbability: 0.5000
    MedianFilterWindowLengthRange: [1 200]
```

In the current augmentation pipeline configuration, the parameter value is not applicable. `ParameterValue` is applicable when `AugmetnationParameterSource` is set to `'specify'`. Set `AugmentationParameterSource` to `'specify'` to enable the current parameter value.

```
aug.AugmentationParameterSource = 'specify'
```

```
aug =
  audioDataAugmenter with properties:

               AugmentationMode: 'sequential'
    AugmentationParameterSource: 'specify'
               ApplyTimeStretch: 1
                  SpeedupFactor: 0.8000
                ApplyPitchShift: 1
                  SemitoneShift: -3
             ApplyVolumeControl: 1
                     VolumeGain: -3
                 ApplyAddNoise: 1
                           SNR: 5
```

```
                ApplyTimeShift: 1
                     TimeShift: 0.0050
            ApplyMedianFilter: 1
    MedianFilterWindowLength: 100
```

Set `AugmentationParameterSource` to `random` and then call `augment`.

```
aug.AugmentationParameterSource = "random";
data = augment(aug,audioIn,fs);
```

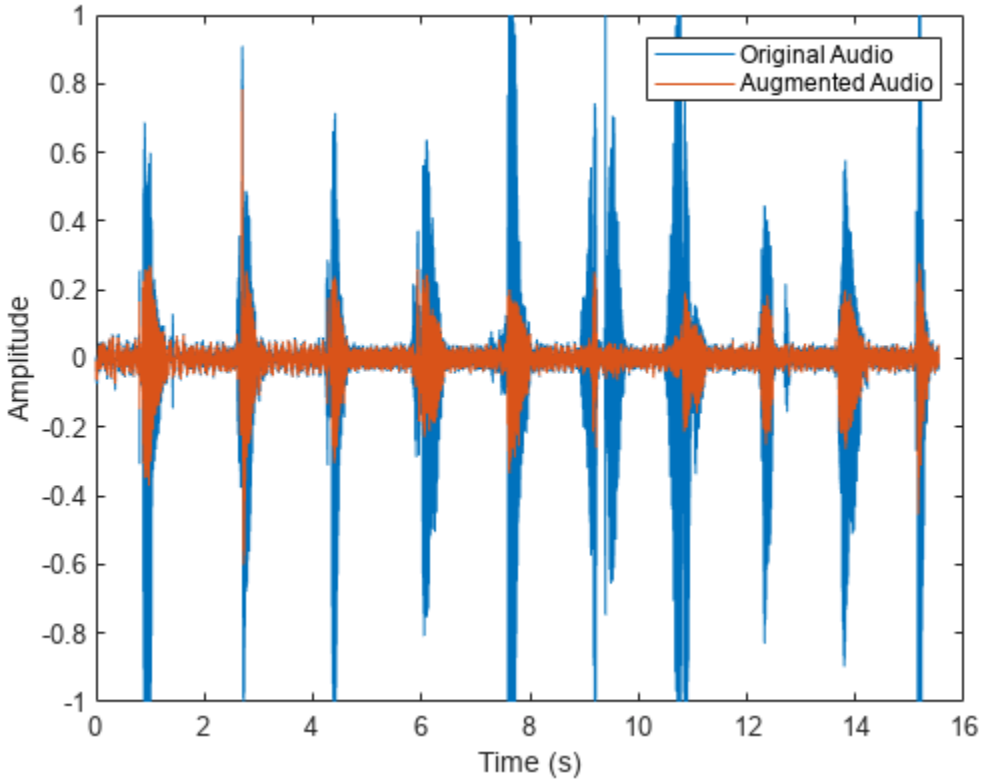
If median filtering was applied for an augmentation, then `AugmentationInfo` lists the value applied.

```
augmentationToInspect = 3;
data.AugmentationInfo(augmentationToInspect)
```

```
ans = struct with fields:
          SNR: 8.7701
    MedianFilter: 117.9847
```

Listen to the audio you are inspecting. Plot the time-domain representation of the original and augmented signals.

```
augmentation = data.Audio{augmentationToInspect};
sound(augmentation,fs)
t = (0:(numel(audioIn)-1))/fs;
taug = (0:(numel(augmentation)-1))/fs;
plot(t,audioIn,taug,augmentation)
legend("Original Audio","Augmented Audio")
ylabel("Amplitude")
xlabel("Time (s)")
```



Specify Multiple Parameters of Custom Augmentation Method

Read in an audio signal and listen to it.

```
[audioIn,fs] = audioread('RockDrums-44p1-stereo-11secs.mp3');
sound(audioIn,fs)
```

Create and `audioDataAugmenter` object that outputs 5 augmentations. Set the `AddNoiseProbability` to `0`.

```
aug = audioDataAugmenter('NumAugmentations',5,'AddNoiseProbability',0);
```

Add reverberation as a custom augmentation algorithm. The `applyReverb` function creates a `reverberator` object, updates the sample rate, pre-delay, and wet/dry mix as indicated, and then applies reverberation. To minimize computational overhead, the reverberator object is persistent. The object is reset on every call to avoid mixing the reverberation tail between audio files.

```
type applyReverb.m
```

```
function audioOut = applyReverb(audio,preDelay,wetDryMix,sampleRate)
    persistent reverbObject
    if isempty(reverbObject)
        reverbObject = reverberator;
    end
    reverbObject.SampleRate = sampleRate;
```

```
    reverbObject.PreDelay = preDelay;
    reverbObject.WetDryMix = wetDryMix;

    audioOut = reverbObject(audio);
    reset(reverbObject)
end
```

Add `applyReverb` as a custom augmentation method. To specify multiple parameters for a custom method, specify the parameters, parameter ranges, and parameter values as cell arrays with the same number of cells. Set the probability of applying reverberation to `1`.

```
algorithmName = 'Reverb';
algorithmHandle = @(x,preDelay,weDryMix)applyReverb(x,preDelay,weDryMix,fs);
parameters = {'PreDelay','WetDryMix'};
parameterRanges = {[0,1],[0,1]};
parameterValues = {0,0.3};

addAugmentationMethod(aug,algorithmName,algorithmHandle, ...
    'AugmentationParameter',parameters, ...
    'ParameterRange',parameterRanges, ...
    'ParameterValue',parameterValues)

aug.ReverbProbability = 1
```

```
aug =
  audioDataAugmenter with properties:

              AugmentationMode: 'sequential'
    AugmentationParameterSource: 'random'
              NumAugmentations: 5
         TimeStretchProbability: 0.5000
             SpeedupFactorRange: [0.8000 1.2000]
          PitchShiftProbability: 0.5000
             SemitoneShiftRange: [-2 2]
       VolumeControlProbability: 0.5000
                VolumeGainRange: [-3 3]
           AddNoiseProbability: 0
          TimeShiftProbability: 0.5000
                TimeShiftRange: [-0.0050 0.0050]
             ReverbProbability: 1
                 PreDelayRange: [0 1]
                WetDryMixRange: [0 1]
```

Call `augment` to create 5 augmentations.

```
data = augment(aug,audioIn,fs);
```

Check the configuration of each augmentation using `AugmentationInfo`.

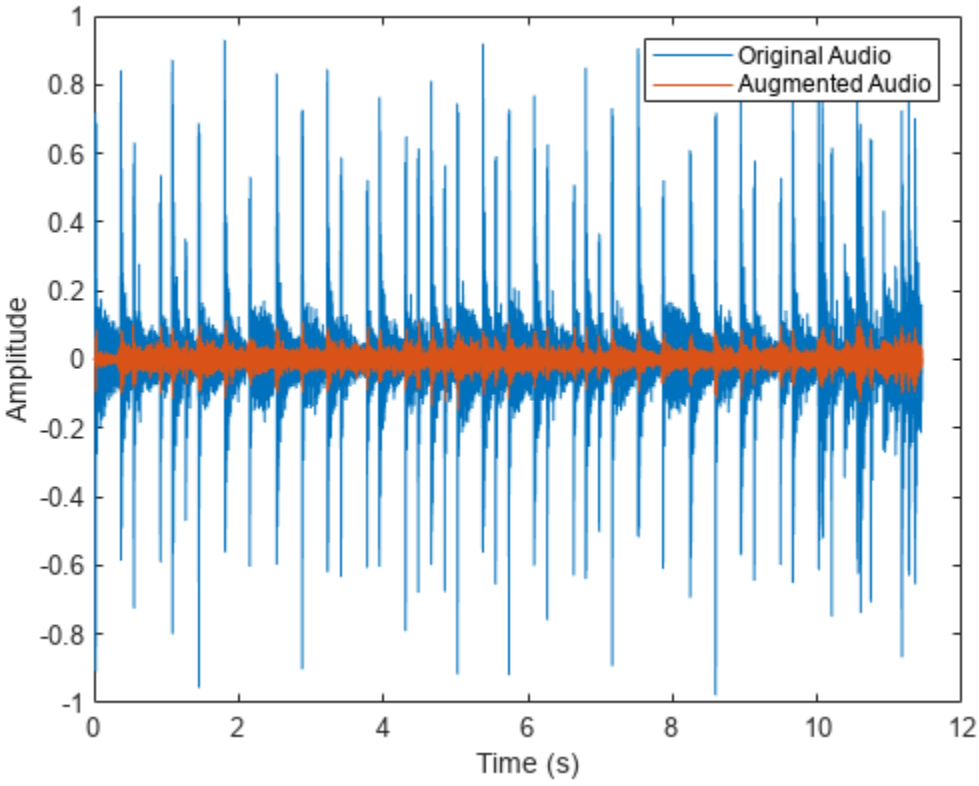
```
augmentationToInspect = 1;
data.AugmentationInfo(augmentationToInspect)
```

```
ans = struct with fields:
    SpeedupFactor: 1
    SemitoneShift: -1.4325
       VolumeGain: 0
        TimeShift: 0
```

```
    Reverb: [0.2760 0.4984]
```

Listen to the audio you are inspecting. Plot the time-domain representation of the original and augmented signals.

```
augmentation = data.Audio{augmentationToInspect};
sound(augmentation,fs)
t = (0:(size(audioIn,1)-1))/fs;
taug = (0:(size(augmentation,1)-1))/fs;
plot(t,audioIn(:,1),taug,augmentation(:,1))
legend("Original Audio","Augmented Audio")
ylabel("Amplitude")
xlabel("Time (s)")
```



Input Arguments

aug — Audio data augmenter
`audioDataAugmenter` object

`audioDataAugmenter` object.

`algorithmName` — Algorithm name
character vector | string

Algorithm name, specified as a character vector or string. `algorithmName` must be a unique property name on the `audioDataAugmenter`, `aug`.

Data Types: `char` | `string`

`algorithmHandle` — Handle to function that implements custom augmentation algorithm
`function_handle`

Handle to function that implements custom augmentation algorithm, specified as a `function_handle`.

Data Types: `function_handle`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `'AugmentationParameter,'PreDelay'`

`AugmentationParameter` — Augmentation parameter
character vector | string | cell array of character vectors | cell array of strings

Augmentation parameter, specified as a character vector, string, cell array of character vectors, or cell array of strings.

Use cell arrays to create multiple augmentation parameters. If you create multiple augmentation parameters, you must also specify `ParameterRange` and `ParameterValue` as cell arrays containing information for each augmentation parameter.

Example: `'AugmentationParameter','PreDelay'`

Example: `'AugmentationParameter',{'PreDelay','HighCutFrequency'}`

Data Types: `char` | `string`

`ParameterRange` — Parameter range
`[0,1]` (default) | two-element vector of nondecreasing values | cell array of two-element vectors of nondecreasing values

Parameter range, specified as a two-element vector of nondecreasing values (for a single parameter) or a cell array of two-element vectors of nondecreasing values (for multiple parameters).

Example: `'ParameterRange',[0,1]`

Example: `'ParameterRange',{[0,1],[20,20000]}`

Dependencies

To enable this property, set the `AugmentationParameterSource` property of your `audioDataAugmenter` object to `'random'`.

Data Types: `single` | `double` | `cell`

`ParameterValue` — Parameter value
`0` (default) | scalar | vector | cell array of scalars or vectors

Parameter value, specified as a scalar, vector, or cell array of scalars or vectors.

Example: `'ParameterValue',0`

Example: `'ParameterValue',[0,0.5,1]`

Example: `'ParameterValue',{0,20000}`

Example: `'ParameterValue',{[0,0.5,1],20000}`

Dependencies

To enable this property, set the `AugmentationParameterSource` property of your `audioDataAugmenter` to `'specify'`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`
Complex Number Support: Yes

Version History

Introduced in R2019b

See Also

`removeAugmentationMethod` | `audioDataAugmenter` | `reverberator`

setAugmenterParams

Set parameters of augmentation algorithm

Syntax

```
setAugmenterParams(aug,algorithmName,params)
setAugmenterParams(aug,algorithmName)
```

Description

`setAugmenterParams(aug,algorithmName,params)` sets parameters of the augmentation algorithm associated with the `audioDataAugmenter` object.

`setAugmenterParams(aug,algorithmName)` without the `params` argument restores the `algorithmName` parameters to their default values.

Examples

Set Augmenter Parameters

Modify the default parameters of the `shiftPitch` and `stretchAudio` augmentation algorithms.

Read in an audio signal and listen to it.

```
[audioIn,fs] = audioread('FemaleSpeech-16-8-mono-3secs.wav');
soundsc(audioIn,fs)
```

Create an `audioDataAugmenter` object that applies a pitch shift of 3 semitones and a time stretch with a `SpeedupFactor` of `1.5`.

```
aug = audioDataAugmenter('AugmentationParameterSource','specify', ...
                         'ApplyPitchShift',true, ...
                         'SemitoneShift',3, ...
                         'ApplyTimeStretch',true, ...
                         'SpeedupFactor',1.5, ...
                         'ApplyVolumeControl',false, ...
                         'ApplyAddNoise',false, ...
                         'ApplyTimeShift',false)
```

```
aug =
  audioDataAugmenter with properties:

              AugmentationMode: 'sequential'
    AugmentationParameterSource: 'specify'
              ApplyTimeStretch: 1
                 SpeedupFactor: 1.5000
               ApplyPitchShift: 1
                 SemitoneShift: 3
            ApplyVolumeControl: 0
                 ApplyAddNoise: 0
```

```
                ApplyTimeShift: 0
```

Call `setAugmenterParams` to set the `LockPhase` and `PreserveFormants` parameters of the `shiftPitch` augmentation algorithm to `false`. Set the `LockPhase` parameter of the `stretchAudio` augmentation algorithm to `false`. Set the `CepstralOrder` parameter of the `shiftPitch` algorithm to `30`.

Augment the original signal and listen to the result. The resulting file has an audible distortion that sounds unnatural. View the parameters of the augmentation algorithms.

```
setAugmenterParams(aug,'shiftPitch','LockPhase',false,'PreserveFormants',false,'CepstralOrder',30
setAugmenterParams(aug,'stretchAudio','LockPhase',false);
data = augment(aug,audioIn,fs);

pause(3)

augmentationPre = data.Audio{1};
soundsc(augmentationPre,fs)

data.AugmentationInfo(1)
```

```
ans = struct with fields:
    SpeedupFactor: 1.5000
    SemitoneShift: 3
```

```
augmenterParamsPre = getAugmenterParams(aug);
augmenterParamsPre.stretchAudio
```

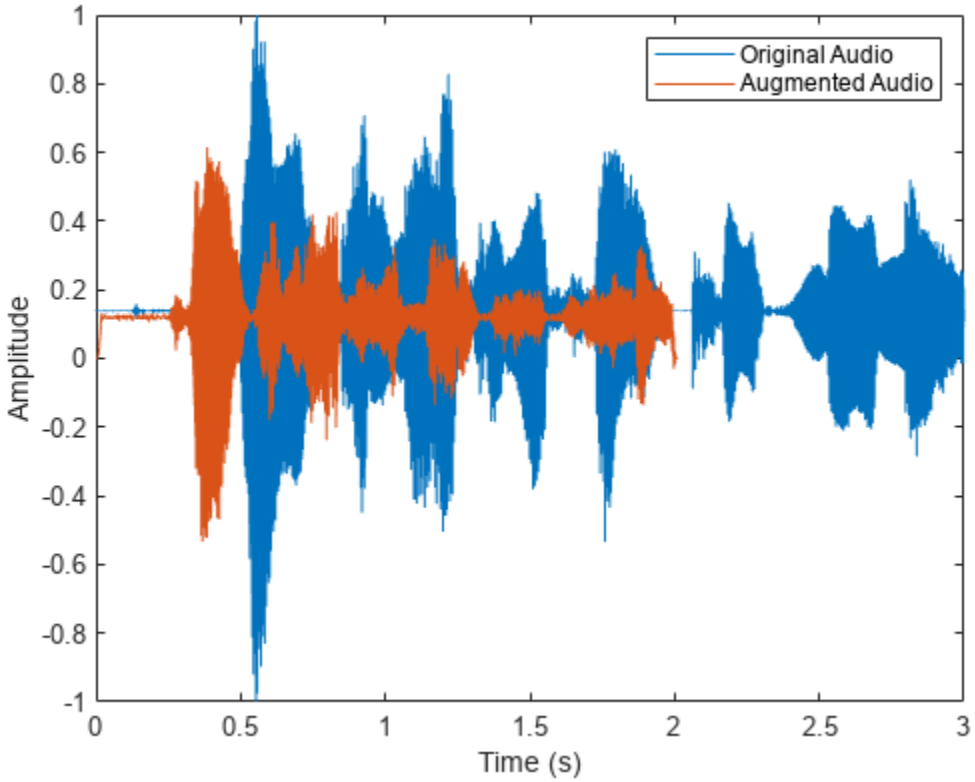
```
ans = struct with fields:
    LockPhase: 0
```

```
augmenterParamsPre.shiftPitch
```

```
ans = struct with fields:
           LockPhase: 0
    PreserveFormants: 0
       CepstralOrder: 30
```

Plot the time-domain representation of the original and the augmented signals.

```
t = (0:(numel(audioIn)-1))/fs;
taug = (0:(numel(augmentationPre)-1))/fs;
plot(t,audioIn,taug,augmentationPre)
legend("Original Audio","Augmented Audio")
ylabel("Amplitude")
xlabel("Time (s)")
```



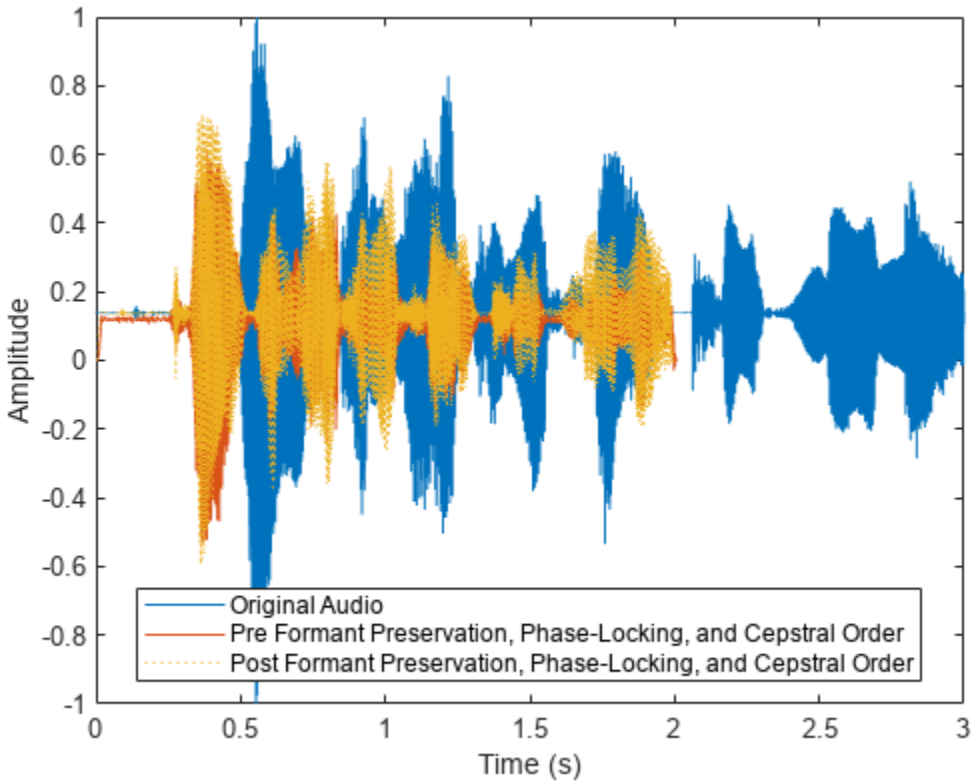
To partially compensate for the audible distortion and increase the fidelity of the augmentation algorithms, apply formant preservation to the `shiftPitch` algorithm, apply phase-locking to both algorithms, and change the cepstral order of the `shiftPitch` algorithm to `25`. Listen to the processed audio.

```
setAugmenterParams(aug,'shiftPitch','LockPhase',true,'PreserveFormants',true,'CepstralOrder',25)
setAugmenterParams(aug,'stretchAudio','LockPhase',true);
data = augment(aug,audioIn,fs);

augmentationPost = data.Audio{1};
soundsc(augmentationPost,fs)

data.AugmentationInfo(1)

ans = struct with fields:
    SpeedupFactor: 1.5000
    SemitoneShift: 3


augmenterParamsPost = getAugmenterParams(aug);
augmenterParamsPost.stretchAudio

ans = struct with fields:
    LockPhase: 1


augmenterParamsPost.shiftPitch
```

```
ans = struct with fields:
           LockPhase: 1
    PreserveFormants: 1
       CepstralOrder: 25
```

Plot the original audio as well as the augmented data before and after formant preservation, phase-locking, and cepstral order modification.

```
taug = (0:(numel(augmentationPost)-1))/fs;
plot(t,audioIn,taug,augmentationPre)
hold on
plot(taug,augmentationPost,'LineStyle',':')
legend("Original Audio","Pre Formant Preservation," + ...
    " Phase-Locking, and Cepstral Order", ...
    "Post Formant Preservation, Phase-Locking, and Cepstral Order")
ylabel("Amplitude")
xlabel("Time (s)")
legend('Location','best')
```



Return the augmentation algorithm parameters to their default values. Call `getAugmenterParams` to display the current parameter values for the `audioAugmenter` object.

```
setAugmenterParams(aug,'shiftPitch')
setAugmenterParams(aug,'stretchAudio')
augmenterParamsDefault = getAugmenterParams(aug);
augmenterParamsDefault.stretchAudio
```

```
ans = struct with fields:
    LockPhase: 0
```

```
augmenterParamsDefault.shiftPitch
```

```
ans = struct with fields:
           LockPhase: 0
    PreserveFormants: 0
       CepstralOrder: 30
```

Input Arguments

aug — Audio data augmenter

`audioDataAugmenter` object

Audio data augmenter, specified as an `audioDataAugmenter` object.

algorithmName — Algorithm name

`'stretchAudio'` | `'shiftPitch'`

Algorithm name, specified as `'stretchAudio'` or `'shiftPitch`.

Note Augmentation algorithms must be modified independently using separate calls to `setAugmenterParams` for each algorithm.

Data Types: `char` | `string`

params — Parameter used with augmentation algorithm

character vector | string | structure array

Parameter name, specified as a character vector, string, or structure array. Parameter values depend on `algorithmName`. Specify `params` as one of these:

- When you set `algorithmName` to `'stretchAudio'`, specify `params` as `'LockPhase'` and `true` or `false`.
- When you set `algorithmName` to `'shiftPitch'`, specify `params` as one or all of these:
  - `'LockPhase'` and `true` or `false`
  - `'PreserveFormants'` and `true` or `false`
  - `'CepstralOrder'` and a positive integer

Example: `setAugmenterParams(aug,'shiftPitch','LockPhase',true,'PreserveFormants',false,'CepstralOrder',15)` enables the `LockPhase` parameter, disables the `PreserveFormants` parameter, and sets a cepstral order of 15 for the `shiftPitch` augmentation algorithm.

Data Types: `char` | `string` | `struct`

Version History

Introduced in R2021a

See Also

`removeAugmentationMethod` | `augment` | `getAugmenterParams` | `addAugmentationMethod` | `audioDataAugmenter`

getAugmenterParams

Get parameters of augmentation algorithm

Syntax

```
augmenterParams = getAugmenterParams(aug,algorithmName)
augmenterParams = getAugmenterParams(aug)
```

Description

`augmenterParams = getAugmenterParams(aug,algorithmName)` returns parameters of the augmentation algorithm associated with the `audioDataAugmenter` object.

`augmenterParams = getAugmenterParams(aug)` returns the parameters of all augmentation algorithms associated with the `audioDataAugmenter` object.

Examples

Set Augmenter Parameters

Modify the default parameters of the `shiftPitch` and `stretchAudio` augmentation algorithms.

Read in an audio signal and listen to it.

```
[audioIn,fs] = audioread('FemaleSpeech-16-8-mono-3secs.wav');
soundsc(audioIn,fs)
```

Create an `audioDataAugmenter` object that applies a pitch shift of 3 semitones and a time stretch with a `SpeedupFactor` of `1.5`.

```
aug = audioDataAugmenter('AugmentationParameterSource','specify', ...
                         'ApplyPitchShift',true, ...
                         'SemitoneShift',3, ...
                         'ApplyTimeStretch',true, ...
                         'SpeedupFactor',1.5, ...
                         'ApplyVolumeControl',false, ...
                         'ApplyAddNoise',false, ...
                         'ApplyTimeShift',false)
```

```
aug =
  audioDataAugmenter with properties:

              AugmentationMode: 'sequential'
    AugmentationParameterSource: 'specify'
              ApplyTimeStretch: 1
                 SpeedupFactor: 1.5000
               ApplyPitchShift: 1
                 SemitoneShift: 3
            ApplyVolumeControl: 0
                 ApplyAddNoise: 0
```

```
                ApplyTimeShift: 0
```

Call `setAugmenterParams` to set the `LockPhase` and `PreserveFormants` parameters of the `shiftPitch` augmentation algorithm to `false`. Set the `LockPhase` parameter of the `stretchAudio` augmentation algorithm to `false`. Set the `CepstralOrder` parameter of the `shiftPitch` algorithm to `30`.

Augment the original signal and listen to the result. The resulting file has an audible distortion that sounds unnatural. View the parameters of the augmentation algorithms.

```
setAugmenterParams(aug,'shiftPitch','LockPhase',false,'PreserveFormants',false,'CepstralOrder',30
setAugmenterParams(aug,'stretchAudio','LockPhase',false);
data = augment(aug,audioIn,fs);

pause(3)

augmentationPre = data.Audio{1};
soundsc(augmentationPre,fs)

data.AugmentationInfo(1)
```

```
ans = struct with fields:
    SpeedupFactor: 1.5000
    SemitoneShift: 3
```

```
augmenterParamsPre = getAugmenterParams(aug);
augmenterParamsPre.stretchAudio
```

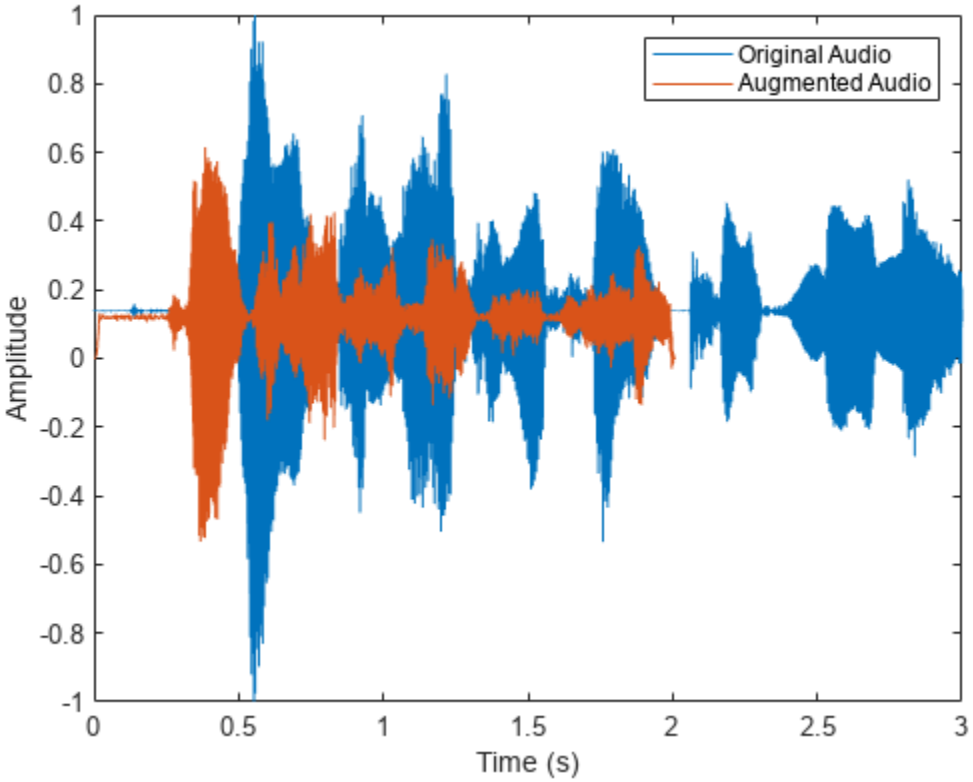
```
ans = struct with fields:
    LockPhase: 0
```

```
augmenterParamsPre.shiftPitch
```

```
ans = struct with fields:
           LockPhase: 0
    PreserveFormants: 0
       CepstralOrder: 30
```

Plot the time-domain representation of the original and the augmented signals.

```
t = (0:(numel(audioIn)-1))/fs;
taug = (0:(numel(augmentationPre)-1))/fs;
plot(t,audioIn,taug,augmentationPre)
legend("Original Audio","Augmented Audio")
ylabel("Amplitude")
xlabel("Time (s)")
```



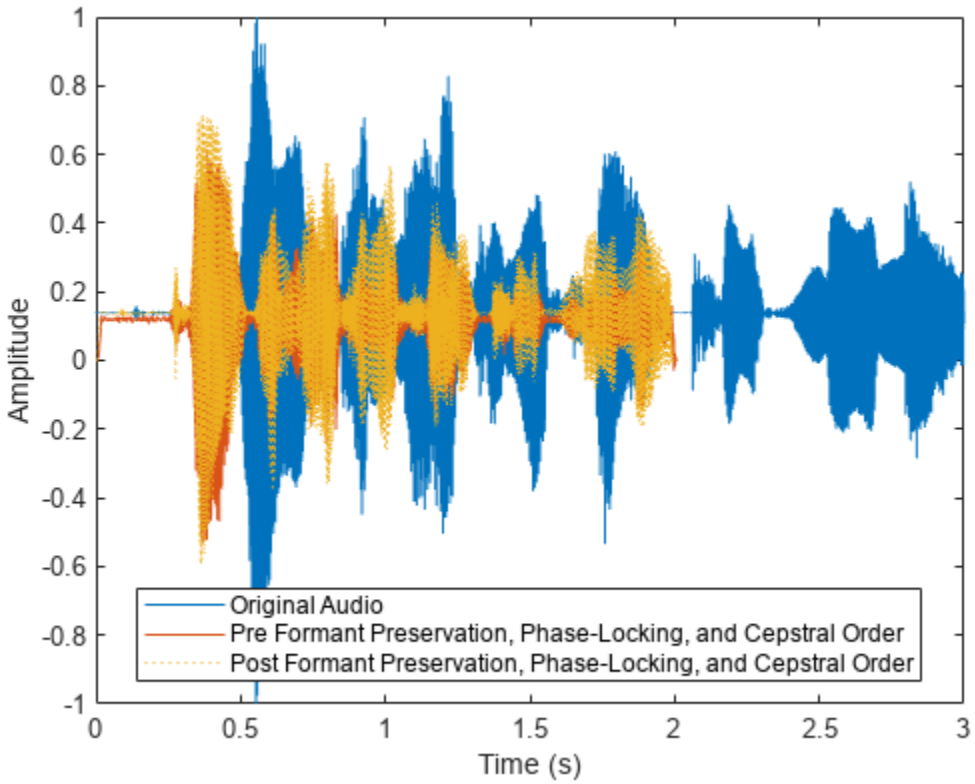
To partially compensate for the audible distortion and increase the fidelity of the augmentation algorithms, apply formant preservation to the `shiftPitch` algorithm, apply phase-locking to both algorithms, and change the cepstral order of the `shiftPitch` algorithm to `25`. Listen to the processed audio.

```
setAugmenterParams(aug,'shiftPitch','LockPhase',true,'PreserveFormants',true,'CepstralOrder',25)
setAugmenterParams(aug,'stretchAudio','LockPhase',true);
data = augment(aug,audioIn,fs);

augmentationPost = data.Audio{1};
soundsc(augmentationPost,fs)

data.AugmentationInfo(1)

ans = struct with fields:
    SpeedupFactor: 1.5000
    SemitoneShift: 3


augmenterParamsPost = getAugmenterParams(aug);
augmenterParamsPost.stretchAudio

ans = struct with fields:
    LockPhase: 1


augmenterParamsPost.shiftPitch
```

```
ans = struct with fields:
           LockPhase: 1
    PreserveFormants: 1
       CepstralOrder: 25
```

Plot the original audio as well as the augmented data before and after formant preservation, phase-locking, and cepstral order modification.

```
taug = (0:(numel(augmentationPost)-1))/fs;
plot(t,audioIn,taug,augmentationPre)
hold on
plot(taug,augmentationPost,'LineStyle',':')
legend("Original Audio","Pre Formant Preservation," + ...
    " Phase-Locking, and Cepstral Order", ...
    "Post Formant Preservation, Phase-Locking, and Cepstral Order")
ylabel("Amplitude")
xlabel("Time (s)")
legend('Location','best')
```



Return the augmentation algorithm parameters to their default values. Call `getAugmenterParams` to display the current parameter values for the `audioAugmenter` object.

```
setAugmenterParams(aug,'shiftPitch')
setAugmenterParams(aug,'stretchAudio')
augmenterParamsDefault = getAugmenterParams(aug);
augmenterParamsDefault.stretchAudio
```

```
ans = struct with fields:
    LockPhase: 0
```

```
augmenterParamsDefault.shiftPitch
```

```
ans = struct with fields:
           LockPhase: 0
    PreserveFormants: 0
       CepstralOrder: 30
```

Input Arguments

aug — Audio data augmenter

`audioDataAugmenter` object

Audio data augmenter, specified as an `audioDataAugmenter` object.

algorithmName — Algorithm name

`'stretchAudio'` | `'shiftPitch'`

Algorithm name, specified as `'stretchAudio'` or `'shiftPitch`.

Data Types: `char` | `string`

Output Arguments

augmenterParams — Audio augmenter parameters

structure array

Audio augmenter parameters, returned as a structure array.

Data Types: `struct`

Version History

Introduced in R2021a

See Also

`removeAugmentationMethod` | `augment` | `setAugmenterParams` | `addAugmentationMethod` | `audioDataAugmenter`

audioDataAugmenter

Augment audio data

Description

Enlarge your audio dataset using audio-specific augmentation techniques like pitch shifting, time-scale modification, time shifting, noise addition, and volume control. You can create cascaded or parallel augmentation pipelines to apply multiple algorithms deterministically or probabilistically.

Creation

Syntax

```
aug = audioDataAugmenter()
aug = audioDataAugmenter(Name,Value)
```

Description

`aug = audioDataAugmenter()` creates an audio data augmenter object with default property values.

`aug = audioDataAugmenter(Name,Value)` specifies nondefault properties for `aug` using one or more name-value arguments.

Properties

Augmentation Pipeline

`AugmentationMode` — Augmentation mode
`'sequential'` (default) | `'independent'`

Augmentation mode, specified as `'sequential'` or `'independent'`.

- `'sequential'` –– Augmentation algorithms are applied sequentially (in series).
- `'independent'` –– Augmentation algorithms are applied independently (in parallel).

Data Types: `char` | `string`

`AugmentationParameterSource` — Source of augmentation parameters
`'random'` (default) | `'specify'`

Source of augmentation parameters, specified as `'random'` or `'specify'`.

- `'random'` –– Augmentation algorithms are applied probabilistically using a probability parameter and a range parameter.

  For example, to create an `audioDataAugmenter` that applies time-stretching using a speedup factor between `0.5` and `1.5` with a 60% probability, enter the following in the Command Window:

```
aug = audioDataAugmenter('AugmentationParameterSource','random', ...
                         'TimeStretchProbability',0.6, ...
                         'SpeedupFactorRange',[0.5,1.5]);
```

When time-stretching is applied, the speedup factor is drawn from a uniform distribution centered at 1 (the mean of the range) with a minimum of `0.5` and a maximum of `1.5`.

- `'specify'` –– Augmentation algorithms are applied deterministically using a logical parameter and a specified parameter value. For example, to create an `audioDataAugmenter` that applies time-stretching using a `1.5` speedup factor with a 100% probability, enter the following in the Command Window:

```
aug = audioDataAugmenter('AugmentationParameterSource','specify', ...
                         'ApplyTimeStretch',true, ...
                         'SpeedupFactor',1.5);
```

Data Types: `char` | `string`

`NumAugmentations` — Number of augmented signals to output
`1` (default) | positive integer

Number of augmented signals to output, specified as a positive integer.

Dependencies

To enable this property, set AugmentationParameterSource to `'random'`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Stretch Time

`TimeStretchProbability` — Probability of applying time stretch
`0.5` (default) | scalar in the range [0, 1]

Probability of applying time stretch, specified as a scalar in the range [0, 1]. Set the probability to `1` to apply time stretching every time you call `augment`. Set the probability to `0` to skip time stretching every time you call `augment`.

Dependencies

To enable this property, set AugmentationParameterSource to `'random'` and AugmentationMode to `'sequential'`.

Data Types: `single` | `double`

`SpeedupFactorRange` — Range of time stretch speedup factor
`[0.8 1.2]` (default) | two-element row vector of positive nondecreasing values

Range of time stretch speedup factor, specified as a two-element row vector of positive nondecreasing values.

Dependencies

To enable this property, set AugmentationParameterSource to `'random'`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`ApplyTimeStretch` — Apply time stretch
`true` (default) | `false`

Apply time stretch, specified as `true` or `false`.

Dependencies

To enable this property, set AugmentationParameterSource to `'specify'`.

Data Types: `logical`

`SpeedupFactor` — Time stretch speedup factor

`0.8` (default) | real positive scalar | real positive vector

Time stretch speedup factor, specified as a scalar or vector of real positive values.

Dependencies

To enable this property, set AugmentationParameterSource to `'specify'`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Shift Pitch

`PitchShiftProbability` — Probability of applying pitch shift

`0.5` (default) | scalar in the range [0, 1]

Probability of applying pitch shift, specified as a scalar in the range [0, 1]. Set the probability to `1` to apply pitch shifting every time you call `augment`. Set the probability to `0` to skip pitch shifting every time you call `augment`.

Dependencies

To enable this property, set AugmentationParameterSource to `'random'` and AugmentationMode to `'sequential'`.

Data Types: `single` | `double`

`SemitoneShiftRange` — Range of pitch shift (semitones)

`[-2,2]` (default) | two-element row vector of nondecreasing values

Range of pitch shift in semitones, specified as a two-element row vector of nondecreasing values.

Dependencies

To enable this property, set AugmentationParameterSource to `'random'`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`ApplyPitchShift` — Apply pitch shift

`true` (default) | `false`

Apply pitch shift, specified as `true` or `false`.

Dependencies

To enable this property, set AugmentationParameterSource to `'specify'`.

Data Types: `logical`

`SemitoneShift` — Pitch shift (semitones)

`-3` (default) | real scalar | real vector

Pitch shift in semitones, specified as a real scalar or vector.

Dependencies

To enable this property, set AugmentationParameterSource to `'specify'`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Control Volume

`VolumeControlProbability` — Probability of applying volume control

`0.5` (default) | scalar in the range [0, 1]

Probability of applying volume control, specified as a scalar in the range [0, 1]. Set the probability to `1` to apply volume control every time you call `augment`. Set the probability to `0` to skip volume control every time you call `augment`.

Dependencies

To enable this property, set AugmentationParameterSource to `'random'` and AugmentationMode to `'sequential'`.

Data Types: `single` | `double`

`VolumeGainRange` — Range of volume gain (dB)

`[-3,3]` (default) | two-element row vector of nondecreasing values

Range of volume gain in dB, specified as a two-element row vector of nondecreasing values.

Dependencies

To enable this property, set AugmentationParameterSource to `'random'`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`ApplyVolumeControl` — Apply volume gain

`true` (default) | `false`

Apply volume gain, specified as `true` or `false`.

Dependencies

To enable this property, set AugmentationParameterSource to `'specify'`.

Data Types: `logical`

`VolumeGain` — Volume gain (dB)

`-3` (default) | scalar | vector

Volume gain in dB, specified as a scalar or vector.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Add Noise

`AddNoiseProbability` — Probability of applying noise addition

`0.5` (default) | scalar in the range [0, 1]

Probability of applying Gaussian white noise addition, specified as a scalar in the range [0, 1]. Set the probability to `1` to add noise every time you call `augment`. Set the probability to `0` to skip adding noise every time you call `augment`.

Dependencies

To enable this property, set AugmentationParameterSource to `'random'` and AugmentationMode to `'sequential'`.

Data Types: `single` | `double`

SNRRange — Range of noise addition SNR (dB)

`[0,10]` (default) | two-element row vector of nondecreasing values

Range of noise addition SNR in dB, specified as a two-element row vector of nondecreasing values.

Dependencies

To enable this property, set AugmentationParameterSource to `'range'`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

ApplyAddNoise — Apply noise addition

`true` (default) | `false`

Apply Gaussian white noise addition, specified as `true` or `false`.

Dependencies

To enable this property, set AugmentationParameterSource to `'specify'`.

Data Types: `logical`

SNR — Noise addition SNR (dB)

`5` (default) | scalar | vector

Noise addition SNR in dB, specified as a scalar or vector.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Shift Time

TimeShiftProbability — Probability of applying time shift

`0.5` (default) | scalar in the range [0, 1]

Probability of applying time shift, specified as a scalar in the range [0, 1]. Set the probability to `1` to apply time shifting every time you call `augment`. Set the property to `0` to skip time shifting every time you call `augment`.

Time-shifting applies a circular shift on the time-domain audio data.

Dependencies

To enable this property, set AugmentationParameterSource to `'random'` and AugmentationMode to `'sequential'`.

Data Types: `single` | `double`

TimeShiftRange — Range of time shift (s)

`[-5e-3,5e3]` (default) | two-element row vector of nondecreasing values.

Range of time shift in seconds, specified as a two-element row vector of nondecreasing values.

Dependencies

To enable this property, set AugmentationParameterSource to `'random'`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`ApplyTimeShift` — Apply time shift
`true` (default) | `false`

Apply time shift, specified as `true` or `false`.

Dependencies

To enable this property, set AugmentationParameterSource to `'specify'`.

Time-shifting applies a circular shift on the time-domain audio data.

Data Types: `logical`

`TimeShift` — Time shift (s)
`5e-3` (default) | scalar | vector

Time shift in seconds, specified as a scalar or vector.

Dependencies

To enable this property, set AugmentationParameterSource to `'specify'`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Object Functions

addAugmentationMethod	Add custom augmentation method
removeAugmentationMethod	Remove custom augmentation method
augment	Augment audio data
setAugmenterParams	Set parameters of augmentation algorithm
getAugmenterParams	Get parameters of augmentation algorithm

Examples

Apply Random Sequential Augmentations

Read in an audio signal and listen to it.

```
[audioIn,fs] = audioread("Counting-16-44p1-mono-15secs.wav");
sound(audioIn,fs)
```

Create an `audioDataAugmenter` object that applies time stretching, volume control, and time shifting in cascade. Apply each of the augmentations with 80% probability. Set `NumAugmentations` to `5` to output five independently augmented signals. To skip pitch shifting and noise addition for each augmentation, set the respective probabilities to `0`. Define parameter ranges for each relevant augmentation algorithm.

```
augmenter = audioDataAugmenter( ...
    "AugmentationMode","sequential", ...
```

```
    "NumAugmentations",5, ...
    ...
    "TimeStretchProbability",0.8, ...
    "SpeedupFactorRange", [1.3,1.4], ...
    ...
    "PitchShiftProbability",0, ...
    ...
    "VolumeControlProbability",0.8, ...
    "VolumeGainRange",[-5,5], ...
    ...
    "AddNoiseProbability",0, ...
    ...
    "TimeShiftProbability",0.8, ...
    "TimeShiftRange", [-500e-3,500e-3])

augmenter =
  audioDataAugmenter with properties:

              AugmentationMode: "sequential"
    AugmentationParameterSource: 'random'
              NumAugmentations: 5
         TimeStretchProbability: 0.8000
             SpeedupFactorRange: [1.3000 1.4000]
          PitchShiftProbability: 0
       VolumeControlProbability: 0.8000
                VolumeGainRange: [-5 5]
            AddNoiseProbability: 0
           TimeShiftProbability: 0.8000
                 TimeShiftRange: [-0.5000 0.5000]
```

Call `augment` on the audio to create 5 augmentations. The augmented audio is returned in a table with variables `Audio` and `AugmentationInfo`. The number of rows in the table is defined by `NumAugmentations`.

```
data = augment(augmenter,audioIn,fs)

data=5×2 table
          Audio           AugmentationInfo
    ____________________    ________________

    {685056x1 double}       1x1 struct
    {685056x1 double}       1x1 struct
    {505183x1 double}       1x1 struct
    {685056x1 double}       1x1 struct
    {490728x1 double}       1x1 struct
```

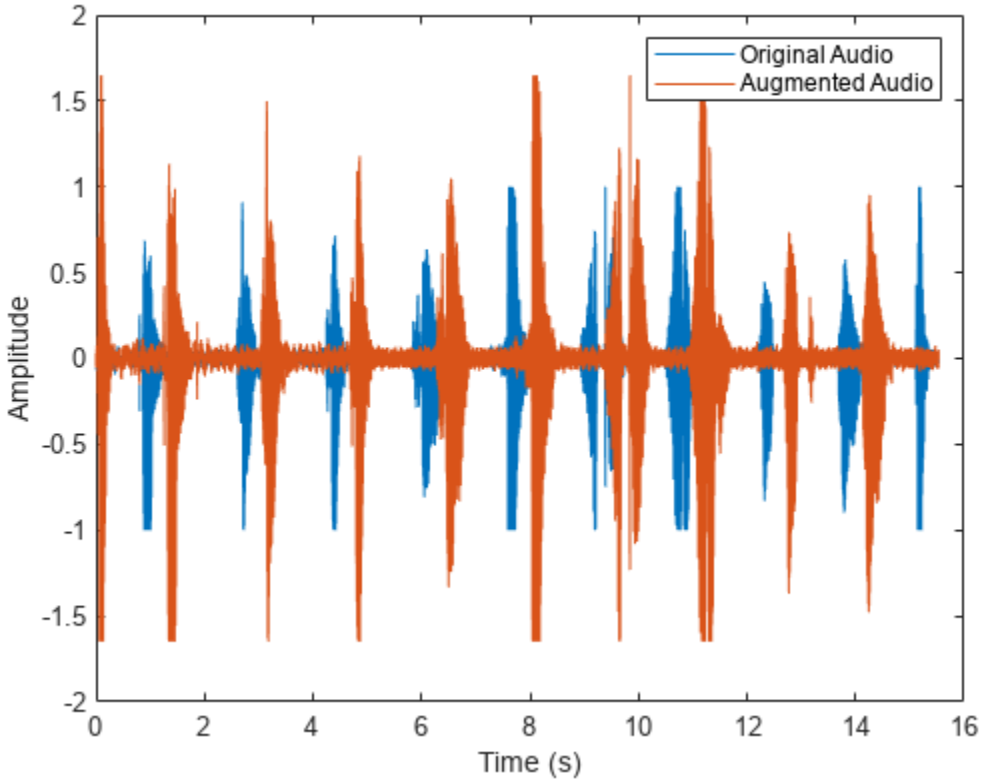
In the current augmentation pipeline, augmentation parameters are assigned randomly from within the specified ranges. To determine the exact parameters used for an augmentation, inspect `AugmentationInfo`.

```
augmentationToInspect = 4;
data.AugmentationInfo(augmentationToInspect)

ans = struct with fields:
    SpeedupFactor: 1
```

```
    VolumeGain: 4.3399
     TimeShift: 0.4502
```

Listen to the augmentation you are inspecting. Plot time representation of the original and augmented signals.

```
augmentation = data.Audio{augmentationToInspect};
sound(augmentation,fs)

t = (0:(numel(audioIn)-1))/fs;
taug = (0:(numel(augmentation)-1))/fs;
plot(t,audioIn,taug,augmentation)
legend("Original Audio","Augmented Audio")
ylabel("Amplitude")
xlabel("Time (s)")
```



Apply Specified Sequential Augmentations

Read in an audio signal and listen to it.

```
[audioIn,fs] = audioread("Counting-16-44p1-mono-15secs.wav");
sound(audioIn,fs)
```

Create an `audioDataAugmenter` object that applies time stretching, pitch shifting, and noise corruption in cascade. Specify the time stretch speedup factors as `0.9`, `1.1`, and `1.2`. Specify the pitch shifting in semitones as `-2`, `-1`, `1`, and `2`. Specify the noise corruption SNR as `10` dB and `15` dB.

```
augmenter = audioDataAugmenter( ...
    "AugmentationMode","sequential", ...
    "AugmentationParameterSource","specify", ...
    "SpeedupFactor",[0.9,1.1,1.2], ...
    "ApplyTimeStretch",true, ...
    "ApplyPitchShift",true, ...
    "SemitoneShift",[-2,-1,1,2], ...
    "SNR",[10,15], ...
    "ApplyVolumeControl",false, ...
    "ApplyTimeShift",false)
```

```
augmenter =
  audioDataAugmenter with properties:

               AugmentationMode: "sequential"
    AugmentationParameterSource: "specify"
               ApplyTimeStretch: 1
                  SpeedupFactor: [0.9000 1.1000 1.2000]
                ApplyPitchShift: 1
                  SemitoneShift: [-2 -1 1 2]
              ApplyVolumeControl: 0
                   ApplyAddNoise: 1
                             SNR: [10 15]
                 ApplyTimeShift: 0
```

Call `augment` on the audio to create 24 augmentations. The augmentations represent every combination of the specified augmentation parameters ($3 \times 4 \times 2 = 24$).

```
data = augment(augmenter,audioIn,fs)
```

```
data=24×2 table
          Audio          AugmentationInfo
    ___________________    ________________

    {761243x1 double}       1x1 struct
    {622888x1 double}       1x1 struct
    {571263x1 double}       1x1 struct
    {761243x1 double}       1x1 struct
    {622888x1 double}       1x1 struct
    {571263x1 double}       1x1 struct
    {761243x1 double}       1x1 struct
    {622888x1 double}       1x1 struct
    {571263x1 double}       1x1 struct
    {761243x1 double}       1x1 struct
    {622888x1 double}       1x1 struct
    {571263x1 double}       1x1 struct
    {761243x1 double}       1x1 struct
    {622888x1 double}       1x1 struct
    {571263x1 double}       1x1 struct
    {761243x1 double}       1x1 struct
      ⋮
```

You can check the parameter configuration of each augmentation using the `AugmentationInfo` table variable.

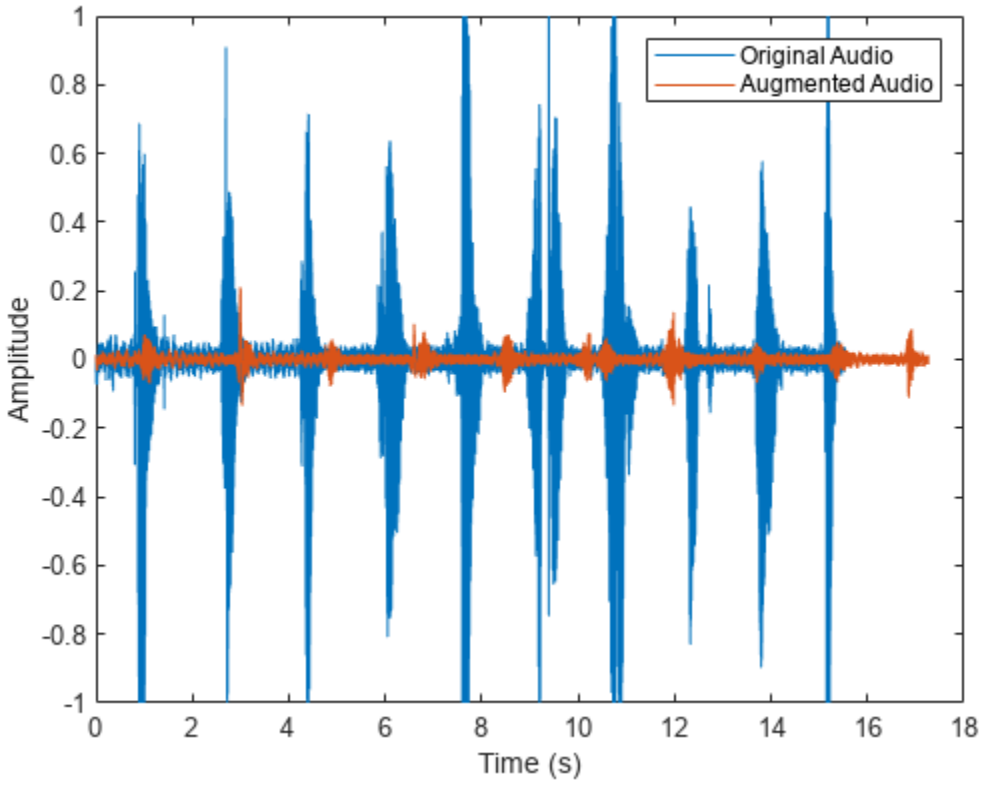
```
augmentationToInspect = 1;
data.AugmentationInfo(augmentationToInspect)
```

```
ans = struct with fields:
    SpeedupFactor: 0.9000
    SemitoneShift: -2
              SNR: 10
```

Listen to the augmentation you are inspecting. Plot the time-domain representation of the original and augmented signals.

```
augmentation = data.Audio{augmentationToInspect};
sound(augmentation,fs)
```

```
t = (0:(numel(audioIn)-1))/fs;
taug = (0:(numel(augmentation)-1))/fs;
plot(t,audioIn,taug,augmentation)
legend("Original Audio","Augmented Audio")
ylabel("Amplitude")
xlabel("Time (s)")
```



Apply Random Independent Augmentations

Read in an audio signal and listen to it.

```
[audioIn,fs] = audioread("Counting-16-44p1-mono-15secs.wav");
```

Create an `audioDataAugmenter` object that applies noise corruption, and time shifting in parallel branches. For the noise corruption branch, randomly apply noise with an SNR in the range `0` dB to `20` dB. For the time shifting branch, randomly apply time shifting in the range -`300` ms to `300` ms. Apply augmentation 2 times for each branch, for 4 total augmentations.

```
augmenter = audioDataAugmenter( ...
    "AugmentationMode","independent", ...
    "AugmentationParameterSource","random", ...
    "NumAugmentations",2, ...
    "ApplyTimeStretch",false, ...
    "ApplyPitchShift",false, ...
    "ApplyVolumeControl",false, ...
    "SNRRange",[0,20], ...
    "TimeShiftRange",[-300e-3,300e-3])
```

```
augmenter =
  audioDataAugmenter with properties:

              AugmentationMode: "independent"
    AugmentationParameterSource: "random"
              NumAugmentations: 2
              ApplyTimeStretch: 0
               ApplyPitchShift: 0
            ApplyVolumeControl: 0
                 ApplyAddNoise: 1
                      SNRRange: [0 20]
                ApplyTimeShift: 1
                TimeShiftRange: [-0.3000 0.3000]
```

Call `augment` on the audio to create 3 augmentations.

```
data = augment(augmenter,audioIn,fs);
```

You can check the parameter configuration of each augmentation using the `AugmentatioInfo` table variable.

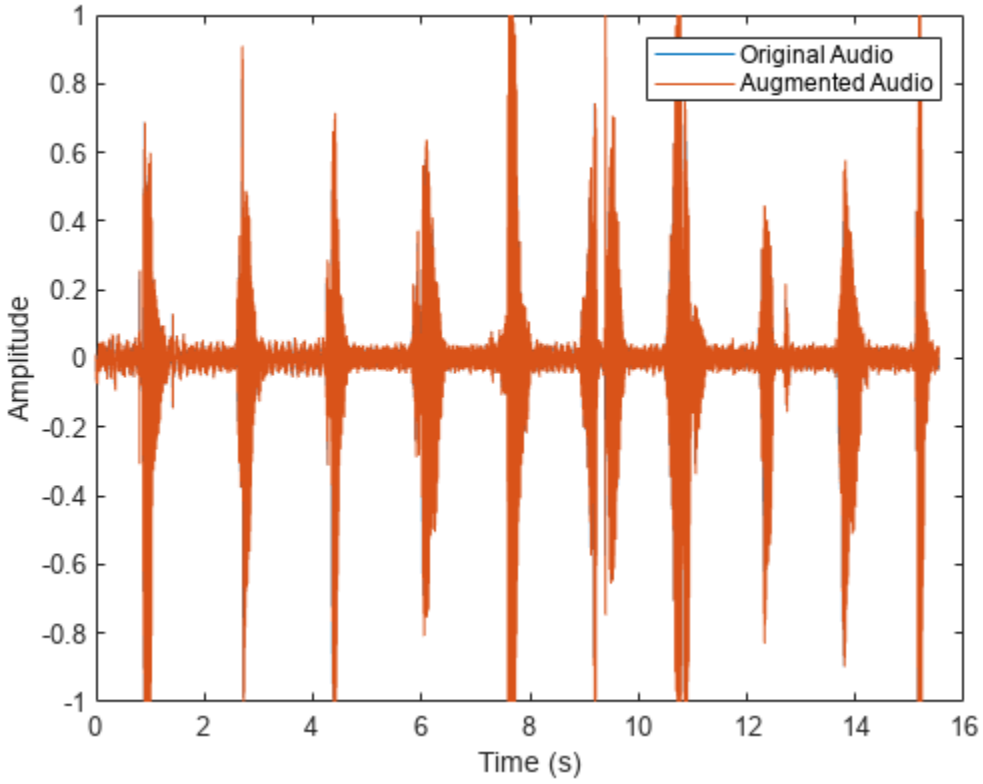
```
augmentationToInspect = 4;
data.AugmentationInfo{augmentationToInspect}
```

```
ans = struct with fields:
    TimeShift: 0.0016
```

Listen to the audio you are inspecting. Plot the time-domain representation of the original and augmented signals.

```
augmentation = data.Audio{augmentationToInspect};
sound(augmentation,fs)
```

```
t = (0:(numel(audioIn)-1))/fs;
taug = (0:(numel(augmentation)-1))/fs;
plot(t,audioIn,taug,augmentation)
legend("Original Audio","Augmented Audio")
ylabel("Amplitude")
xlabel("Time (s)")
```



Apply Specified Independent Augmentations

Read in an audio signal and listen to it.

```
[audioIn,fs] = audioread("Counting-16-44p1-mono-15secs.wav");
```

Create an `audioDataAugmenter` object that applies volume control, noise corruption, and time shifting in parallel branches.

```
augmenter = audioDataAugmenter( ...
    "AugmentationMode","independent", ...
    "AugmentationParameterSource","specify", ...
    "ApplyTimeStretch",false, ...
    "ApplyPitchShift",false, ...
    "VolumeGain",2, ...
    "SNR",0, ...
    "TimeShift",2)
```

```
augmenter =
  audioDataAugmenter with properties:

               AugmentationMode: "independent"
    AugmentationParameterSource: "specify"
               ApplyTimeStretch: 0
                ApplyPitchShift: 0
              ApplyVolumeControl: 1
                      VolumeGain: 2
                  ApplyAddNoise: 1
                            SNR: 0
                 ApplyTimeShift: 1
                      TimeShift: 2
```

Call `augment` on the audio to create 3 augmentations.

```
data = augment(augmenter,audioIn,fs)
```

```
data=3×2 table
          Audio          AugmentationInfo
    ___________________    ________________

    {685056x1 double}      {1x1 struct}
    {685056x1 double}      {1x1 struct}
    {685056x1 double}      {1x1 struct}
```

You can check the parameter configuration of each augmentation using the `AugmentatioInfo` table variable.

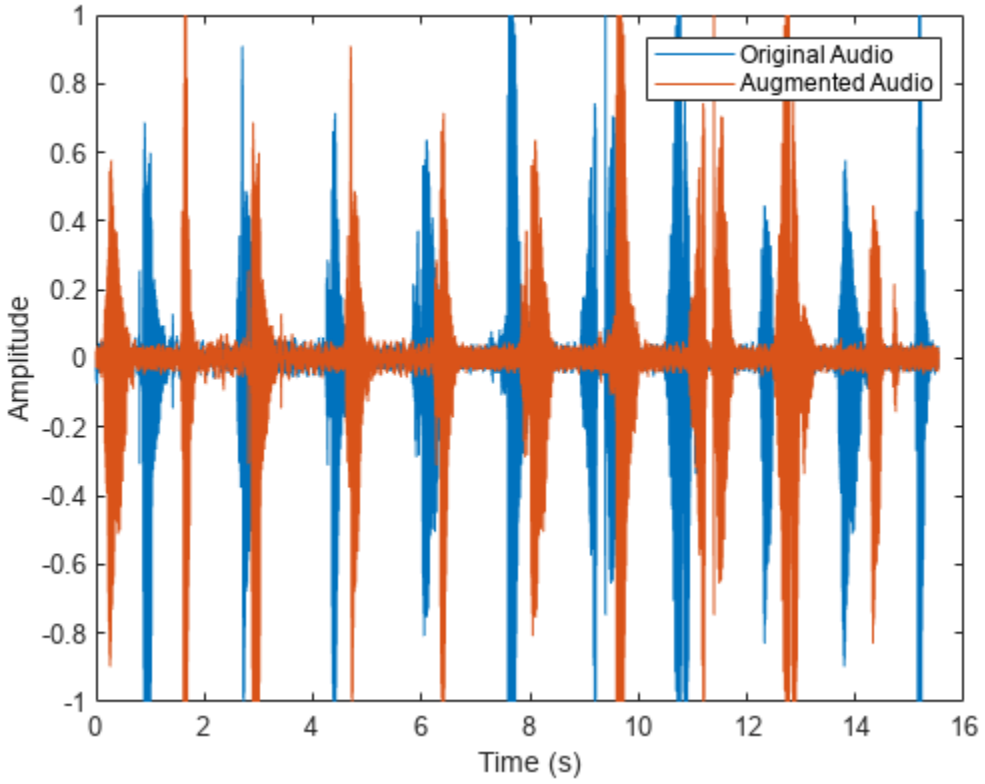
```
augmentationToInspect = 3;
data.AugmentationInfo{augmentationToInspect}
```

```
ans = struct with fields:
    TimeShift: 2
```

Listen to the audio you are inspecting. Plot the time-domain representations of the original and augmented signals.

```
augmentation = data.Audio{augmentationToInspect};
sound(augmentation,fs)
```

```
t = (0:(numel(audioIn)-1))/fs;
taug = (0:(numel(augmentation)-1))/fs;
plot(t,audioIn,taug,augmentation)
legend("Original Audio","Augmented Audio")
ylabel("Amplitude")
xlabel("Time (s)")
```



Augment Audio Dataset

The `audioDataAugmenter` supports multiple workflows for augmenting your datastore, including:

- Offline augmentation
- Augmentation using tall arrays
- Augmentation using transform datastores

In each workflow, begin by creating an audio datastore to point to your audio data. In this example, you create an audio datastore that points to audio samples included with Audio Toolbox™. Count the number of files in the dataset.

```
folder = fullfile(matlabroot,"toolbox","audio","samples");
ADS = audioDatastore(folder)
```

```
ADS =
  audioDatastore with properties:

                      Files: {
                             ' ...\matlab\toolbox\audio\samples\Ambiance-16-44p1-mono-12secs.wav
                             ' ...\matlab\toolbox\audio\samples\AudioArray-16-16-4channels-20sec
                             ' ...\toolbox\audio\samples\ChurchImpulseResponse-16-44p1-mono-5sec
                              ... and 26 more
                             }
```

```
    AlternateFileSystemRoots: {}
              OutputDataType: 'double'
                      Labels: {}
```

```
numFilesInDataset = numel(ADS.Files)
```

```
numFilesInDataset = 29
```

Create an `audioDataAugmenter` that applies random sequential augmentations. Set `NumAugmentations` to 2.

```
aug = audioDataAugmenter('NumAugmentations',2)
```

```
aug =
  audioDataAugmenter with properties:

                AugmentationMode: 'sequential'
    AugmentationParameterSource: 'random'
               NumAugmentations: 2
         TimeStretchProbability: 0.5000
             SpeedupFactorRange: [0.8000 1.2000]
          PitchShiftProbability: 0.5000
             SemitoneShiftRange: [-2 2]
       VolumeControlProbability: 0.5000
                VolumeGainRange: [-3 3]
            AddNoiseProbability: 0.5000
                        SNRRange: [0 10]
           TimeShiftProbability: 0.5000
                 TimeShiftRange: [-0.0050 0.0050]
```

Offline Augmentation

To augment the audio dataset, create two augmentations of each file and then write the augmentations as WAV files.

```
while hasdata(ADS)
    [audioIn,info] = read(ADS);

    data = augment(aug,audioIn,info.SampleRate);

    [~,fn] = fileparts(info.FileName);
    for i = 1:size(data,1)
        augmentedAudio = data.Audio{i};

        % If augmentation caused an audio signal to have values outside of -1 and 1,
        % normalize the audio signal to avoid clipping when writing.
        if max(abs(augmentedAudio),[],'all')>1
            augmentedAudio = augmentedAudio/max(abs(augmentedAudio),[],'all');
        end

        audiowrite(sprintf('%s_aug%d.wav',fn,i),augmentedAudio,info.SampleRate)
    end
end
```

Create an `audioDatastore` that points to the augmented dataset and confirm that the number of files in the dataset is double the original number of files.

```
augmentedADS = audioDatastore(pwd)

augmentedADS =
  audioDatastore with properties:

                      Files: {
                             ' ...\Examples\audio-ex28074079\Ambiance-16-44p1-mono-12secs_aug1.w
                             ' ...\Examples\audio-ex28074079\Ambiance-16-44p1-mono-12secs_aug2.w
                             ' ...\Examples\audio-ex28074079\AudioArray-16-16-4channels-20secs_a
                              ... and 55 more
                             }
    AlternateFileSystemRoots: {}
              OutputDataType: 'double'
                      Labels: {}


numFilesInAugmentedDataset = numel(augmentedADS.Files)

numFilesInAugmentedDataset = 58
```

Augment Using Tall Arrays

When augmenting a dataset using tall arrays, the input data to the augmenter should be sampled at a consistent rate. Subset the original audio dataset to only include files with a sample rate of 44.1 kHz. Most datasets are already cleaned to have a consistent sample rate.

```
keepFile = cellfun(@(x)contains(x,'44p1'),ADS.Files);
ads44p1 = subset(ADS,keepFile);
fs = 44.1e3;
```

Convert the audio datastore to a tall array. `tall` arrays are evaluated only when you request them explicitly using `gather`. MATLAB® automatically optimizes the queued calculations by minimizing the number of passes through the data. If you have the Parallel Computing Toolbox™, you can spread the calculations across multiple machines. The audio data is represented as an *M*-by-1 tall cell array, where *M* is the number of files in the audio datastore.

```
adsTall = tall(ads44p1)

Starting parallel pool (parpool) using the 'local' profile ...
Connected to the parallel pool (number of workers: 6).

adsTall =

  M×1 tall cell array

    { 539648×1 double}
    { 227497×1 double}
    {   8000×1 double}
    { 685056×1 double}
    { 882688×2 double}
    {1115760×2 double}
    { 505200×2 double}
    {3195904×2 double}
        :         :
        :         :
```

Define a `cellfun` function so that augmentation is applied to each cell of the tall array. Call `gather` to evaluate the tall array.

```
augTall = cellfun(@(x)augment(aug,x,fs),adsTall,"UniformOutput",false);
augmentedDataset = gather(augTall)
```

```
Evaluating tall expression using the Parallel Pool 'local':
- Pass 1 of 1: Completed in 1 min 34 sec
Evaluation completed in 1 min 34 sec

augmentedDataset=12×1 cell array
    {2×2 table}
    {2×2 table}
    {2×2 table}
    {2×2 table}
    {2×2 table}
    {2×2 table}
    {2×2 table}
    {2×2 table}
    {2×2 table}
    {2×2 table}
    {2×2 table}
    {2×2 table}
```

The augmented dataset is returned as a *numFiles*-by-1 cell array, where *numFiles* is the number of files in the datastore. Each element of the cell array is a *numAugmentationsPerFile*-by-2 table, where *numAugmentationsPerFile* is the number of augmentations returned per file.

```
numFiles = numel(augmentedDataset)
```

```
numFiles = 12
```

```
numAugmentationsPerFile = size(augmentedDataset{1},1)
```

```
numAugmentationsPerFile = 2
```

Augment Using Transform Datastore

You can perform online data augmentation while you train your machine learning application using a transform datastore. Call `transform` to create a new datastore that applies data augmentation while reading.

```
transformADS = transform(ADS,@(x,info)augment(aug,x,info),'IncludeInfo',true)
```

```
transformADS =
  TransformedDatastore with properties:

    UnderlyingDatastore: [1×1 audioDatastore]
             Transforms: {@(x,info)augment(aug,x,info)}
            IncludeInfo: 1
```

Call `read` to return the augmented first file from the transform datastore.

```
augmentedRead = read(transformADS)
```

```
augmentedRead=2×2 table
          Audio          AugmentationInfo
    _________________    ________________

    {539648×1 double}      [1×1 struct]
```

```
    {586683×1 double}      [1×1 struct]
```

Add Custom Augmentation Method

You can expand the capabilities of `audioDataAugmenter` by adding custom augmentation methods.

Read in an audio signal and listen to it.

```
[audioIn,fs] = audioread('Counting-16-44p1-mono-15secs.wav');
sound(audioIn,fs)
```

Create an `audioDataAugmenter` object. Set the probability of applying white noise to `0`.

```
augmenter = audioDataAugmenter('AddNoiseProbability',0)
```

```
augmenter =
  audioDataAugmenter with properties:

              AugmentationMode: 'sequential'
    AugmentationParameterSource: 'random'
              NumAugmentations: 1
         TimeStretchProbability: 0.5000
             SpeedupFactorRange: [0.8000 1.2000]
          PitchShiftProbability: 0.5000
             SemitoneShiftRange: [-2 2]
       VolumeControlProbability: 0.5000
                VolumeGainRange: [-3 3]
            AddNoiseProbability: 0
           TimeShiftProbability: 0.5000
                 TimeShiftRange: [-0.0050 0.0050]
```

Specify a custom augmentation algorithm that applies pink noise. The `AddPinkNoise` algorithm is added to the `augmenter` properties.

```
algorithmName = 'AddPinkNoise';
algorithmHandle = @(x)x+pinknoise(size(x),'like',x);
addAugmentationMethod(augmenter,algorithmName,algorithmHandle)

augmenter
```

```
augmenter =
  audioDataAugmenter with properties:

              AugmentationMode: 'sequential'
    AugmentationParameterSource: 'random'
              NumAugmentations: 1
         TimeStretchProbability: 0.5000
             SpeedupFactorRange: [0.8000 1.2000]
          PitchShiftProbability: 0.5000
             SemitoneShiftRange: [-2 2]
       VolumeControlProbability: 0.5000
                VolumeGainRange: [-3 3]
            AddNoiseProbability: 0
           TimeShiftProbability: 0.5000
                 TimeShiftRange: [-0.0050 0.0050]
```

```
        AddPinkNoiseProbability: 0.5000
```

Set the probability of adding pink noise to 1.

```
augmenter.AddPinkNoiseProbability = 1
```

```
augmenter =
  audioDataAugmenter with properties:

               AugmentationMode: 'sequential'
    AugmentationParameterSource: 'random'
               NumAugmentations: 1
         TimeStretchProbability: 0.5000
             SpeedupFactorRange: [0.8000 1.2000]
          PitchShiftProbability: 0.5000
             SemitoneShiftRange: [-2 2]
       VolumeControlProbability: 0.5000
                VolumeGainRange: [-3 3]
            AddNoiseProbability: 0
           TimeShiftProbability: 0.5000
                 TimeShiftRange: [-0.0050 0.0050]
        AddPinkNoiseProbability: 1
```

Augment the original signal and listen to the result. Inspect parameters of the augmentation algorithms applied.

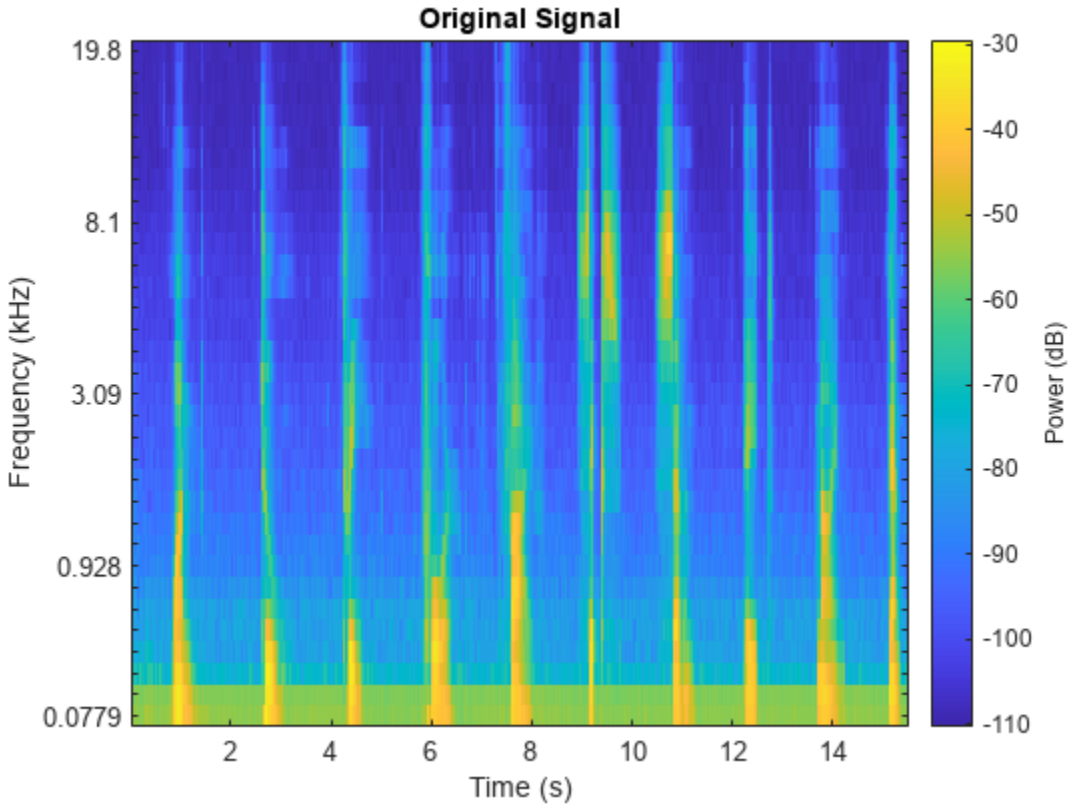
```
data = augment(augmenter,audioIn,fs);
sound(data.Audio{1},fs)
```

```
data.AugmentationInfo(1)
```

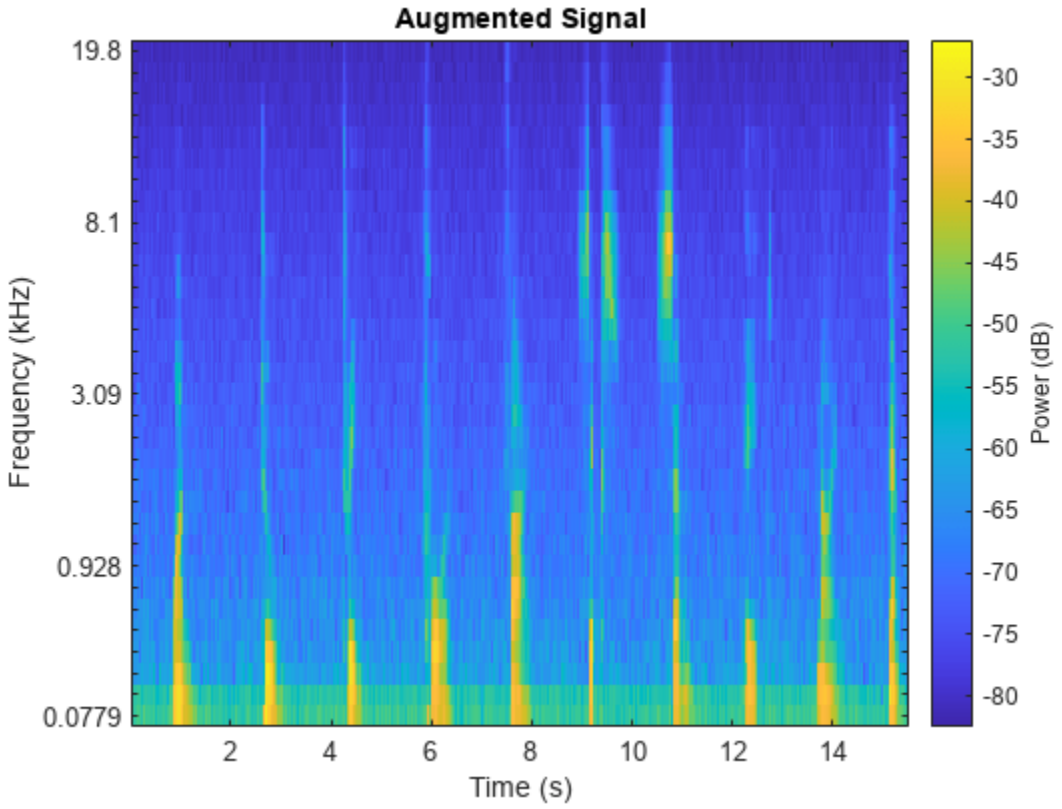
```
ans = struct with fields:
    SpeedupFactor: 1
    SemitoneShift: 0
       VolumeGain: 2.4803
        TimeShift: -0.0022
     AddPinkNoise: 'Applied'
```

Plot the mel spectrograms of the original and augmented signals.

```
melSpectrogram(audioIn,fs)
title('Original Signal')
```




```
melSpectrogram(data.Audio{1},fs)
title('Augmented Signal')
```



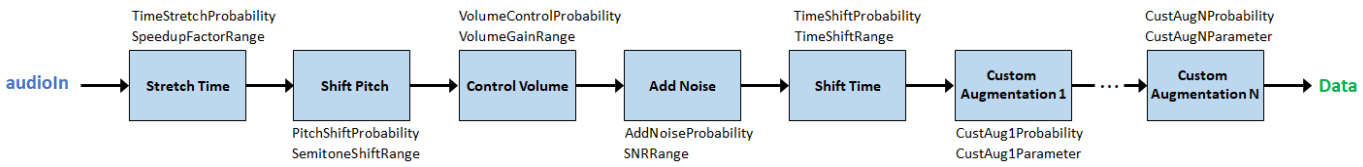
Algorithms

The `audioDataAugmenter` object enables you to configure your augmentation pipeline as deterministic or probabilistic using the AugmentationParameterSource property. You can also choose to apply the augmentations in series or in parallel using the AugmentationMode property. The following sections describe the pipelines you can create and the applicable properties for each architecture.

Random Sequential Augmentations

To define your augmentation as a sequence of probabilistically applied augmentations, set AugmentationParameterSource to `'random'` and AugmentationMode to `'sequential'`.

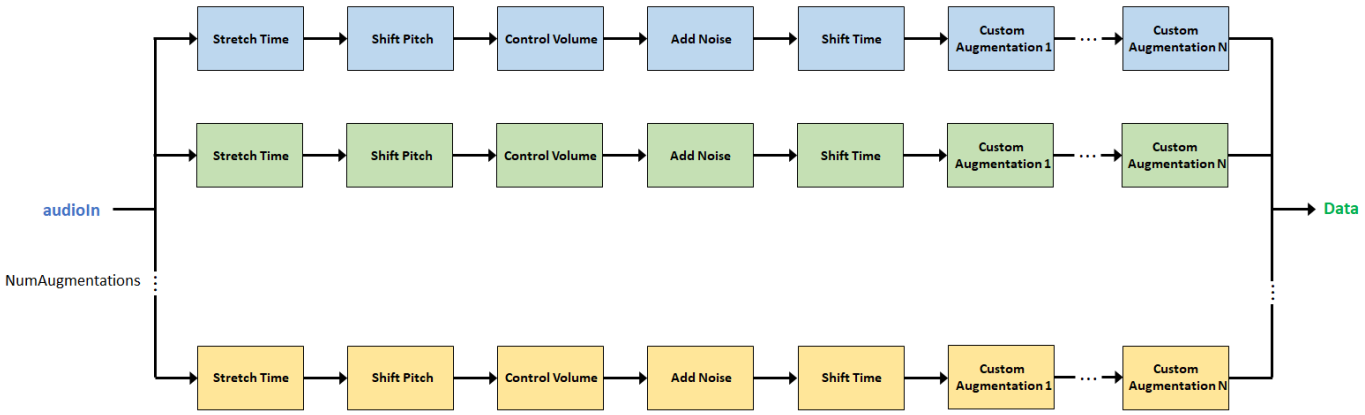
The order that augmentations are applied is always the same. If you specify custom algorithms, they are applied at the end of the sequence, in the order you specified them.



In this pipeline configuration, these parameters apply:

Augmentation Method	Parameters
Stretch Time	TimeStretchProbability SpeedupFactorRange
Shift Pitch	PitchShiftProbability SemitoneShiftRange
Control Volume	VolumeControlProbability VolumeGainRange
Add Noise	AddNoiseProbability SNRRange
Shift Time	TimeShiftProbability TimeShiftRange

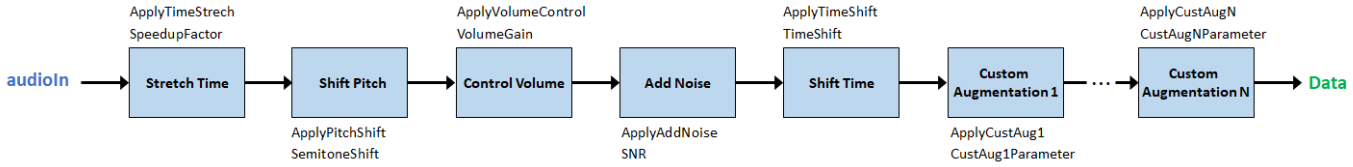
If you specify NumAugmentations as greater than 1, then the object applies `NumAugmentations` parallel random sequential augmentations. The probability of applying an augmentation, and the value of any parameters that are probabilistically determined, are independent.



Specified Sequential Augmentations

To define your augmentation as a sequence of deterministically applied augmentations, set AugmentationParameterSource to `'specify'` and AugmentationMode to `'sequential'`.

The order that augmentations are applied is always the same. If you specify custom algorithms, they are applied at the end of the sequence, in the order you specified them.

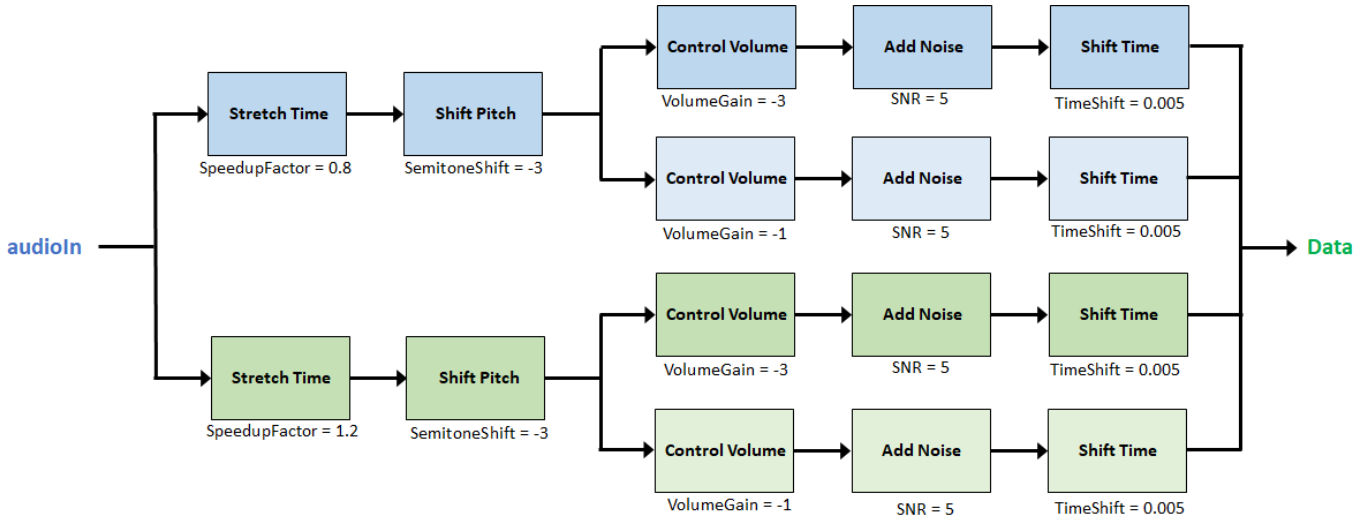


In this pipeline configuration, these parameters apply:

Augmentation Method	Parameters
Stretch Time	ApplyTimeStretch SpeedupFactor
Shift Pitch	ApplyPitchShift SemitoneShift
Control Volume	ApplyVolumeControl VolumeGain
Add Noise	ApplyAddNoise SNR
Shift Time	ApplyTimeShift TimeShift

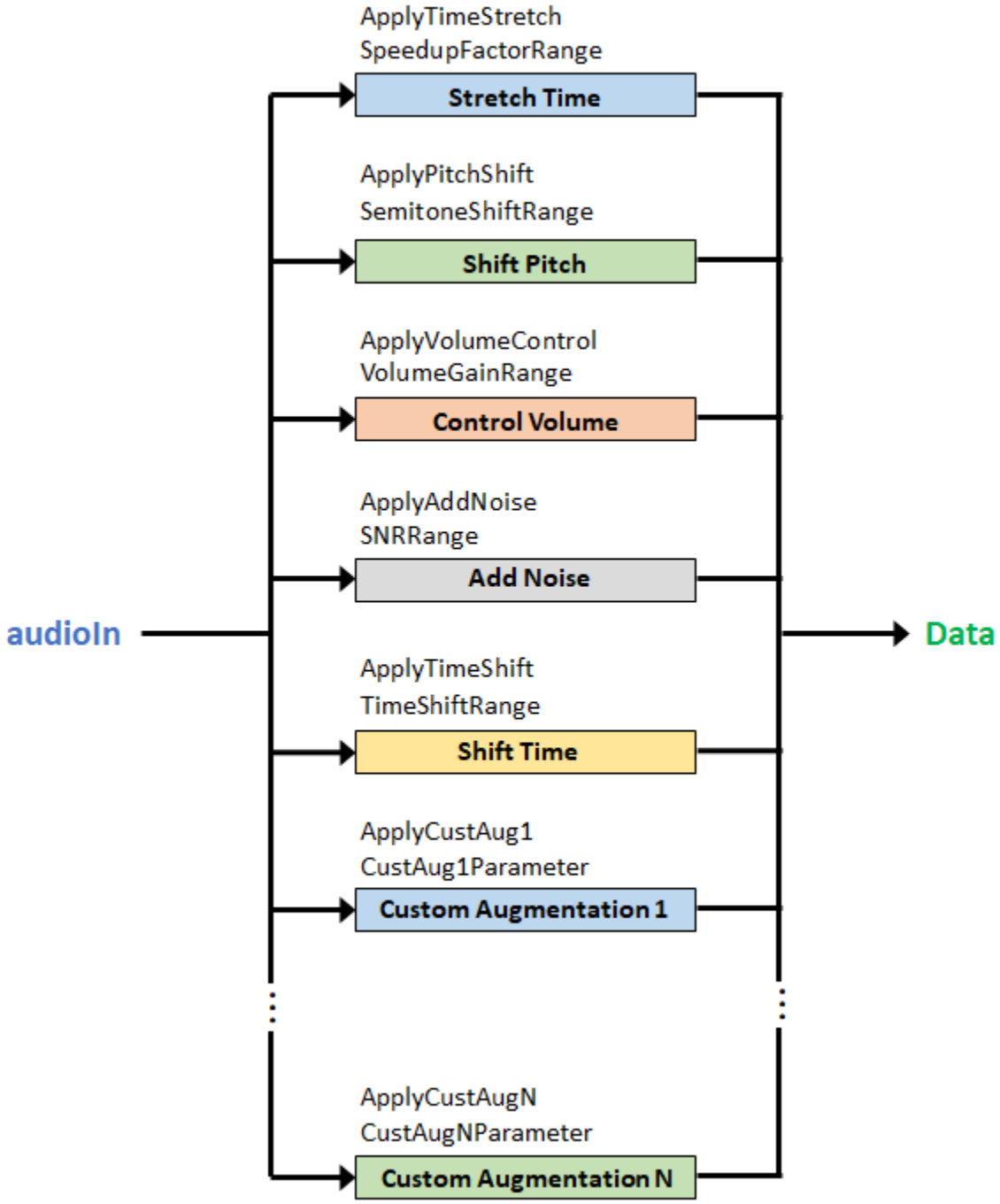
If you specify an augmentation method as a vector, then each element of the vector creates a separate branch in the augmentation pipeline. For example, the following object creates an augmentation pipeline that results in four separate augmentations:

```
aug = audioDataAugmenter("AugmentationMode","sequential", ...
    "AugmentationParameterSource","specify", ...
    "SpeedupFactor",[0.8,1.2], ...
    "VolumeGain",[-3,-1])

aug =

  audioDataAugmenter with properties:

             AugmentationMode: "sequential"
    AugmentationParameterSource: "specify"
              ApplyTimeStretch: 1
                 SpeedupFactor: [0.8000 1.2000]
               ApplyPitchShift: 1
                 SemitoneShift: -3
            ApplyVolumeControl: 1
                    VolumeGain: [-3 -1]
                ApplyAddNoise: 1
                           SNR: 5
               ApplyTimeShift: 1
                    TimeShift: 0.0050
```



Random Independent Augmentations

To define your augmentation as independently applied augmentations with randomly determined parameters, set AugmentationParameterSource to `'random'` and AugmentationMode to `'independent'`.

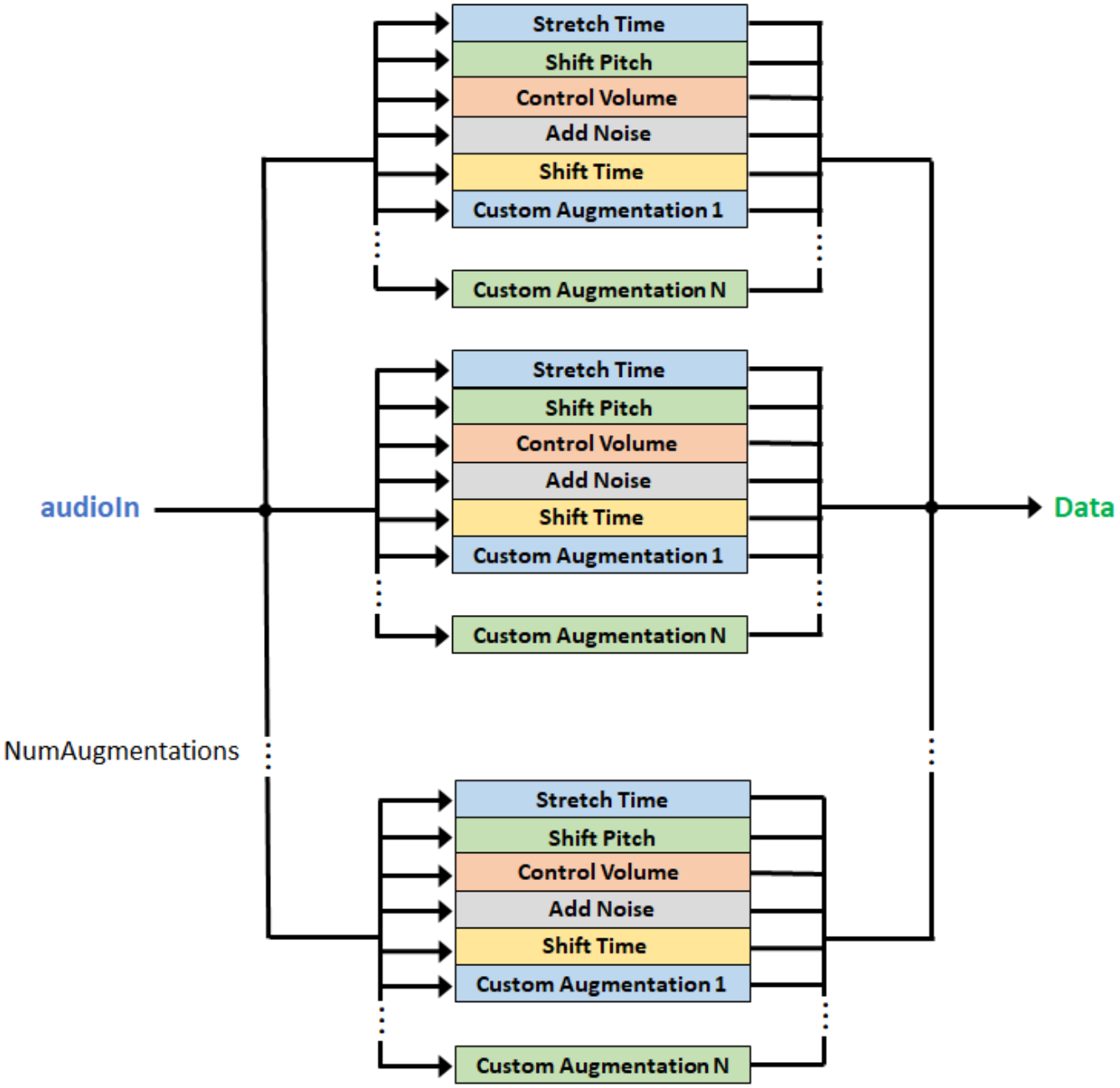


In this pipeline configuration, these parameters apply:

Augmentation Method	Parameters
Stretch Time	ApplyTimeStretch SpeedupFactorRange

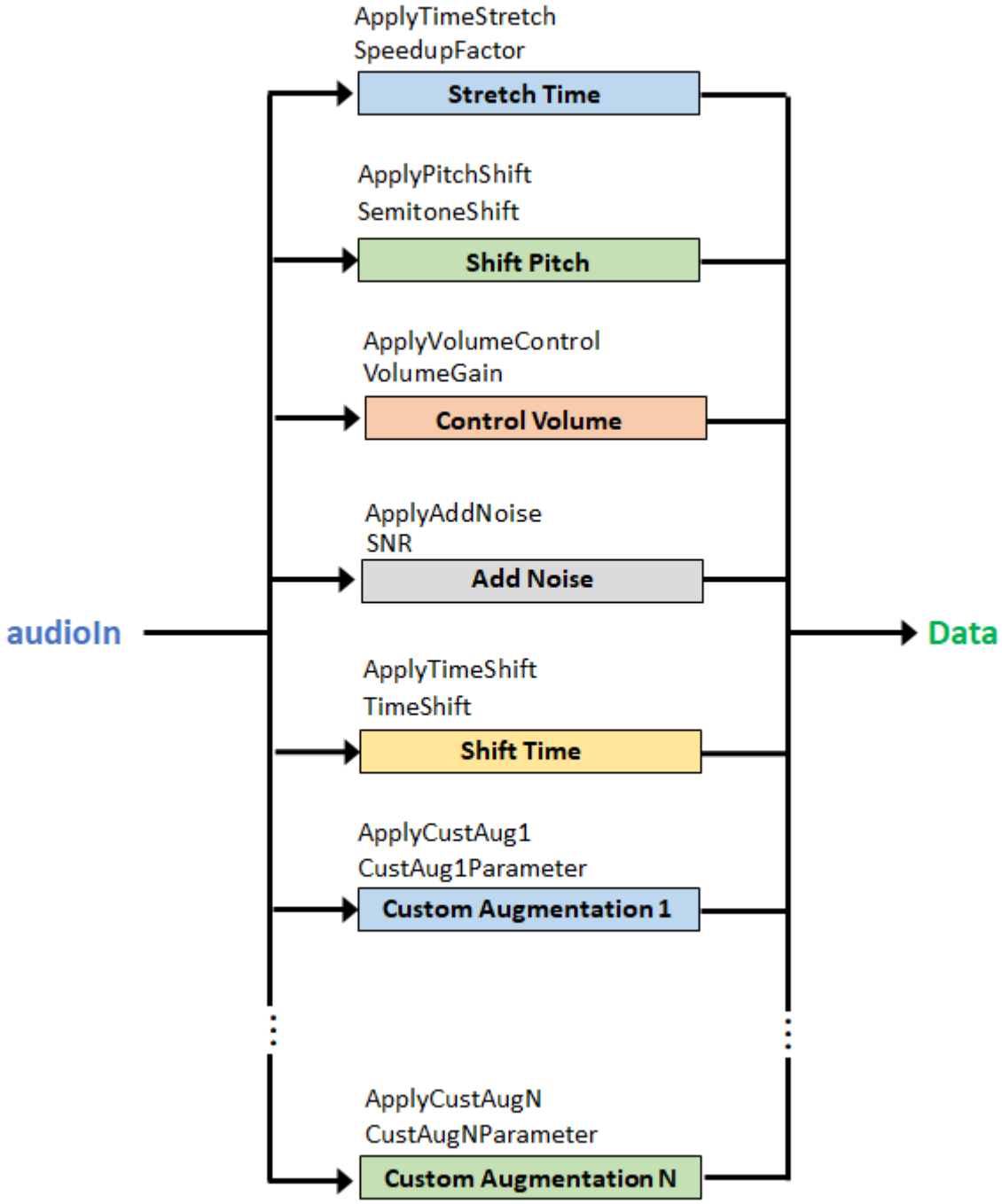
Augmentation Method	Parameters
Shift Pitch	ApplyPitchShift SemitoneShiftRange
Control Volume	ApplyVolumeControl VolumeGainRange
Add Noise	ApplyAddNoise SNRRange
Shift Time	ApplyTimeShift TimeShiftRange

If you specify NumAugmentations as greater than 1, then the object applies `NumAugmentations` parallel random independent augmentations. The value of any parameters that are probabilistically determined are independent.



Specified Independent Augmentations

To define your augmentation as deterministically applied independent augmentations with deterministic parameters, set AugmentationParameterSource to `'specify'` and AugmentationMode to `'independent'`.



In this pipeline configuration, these parameters apply:

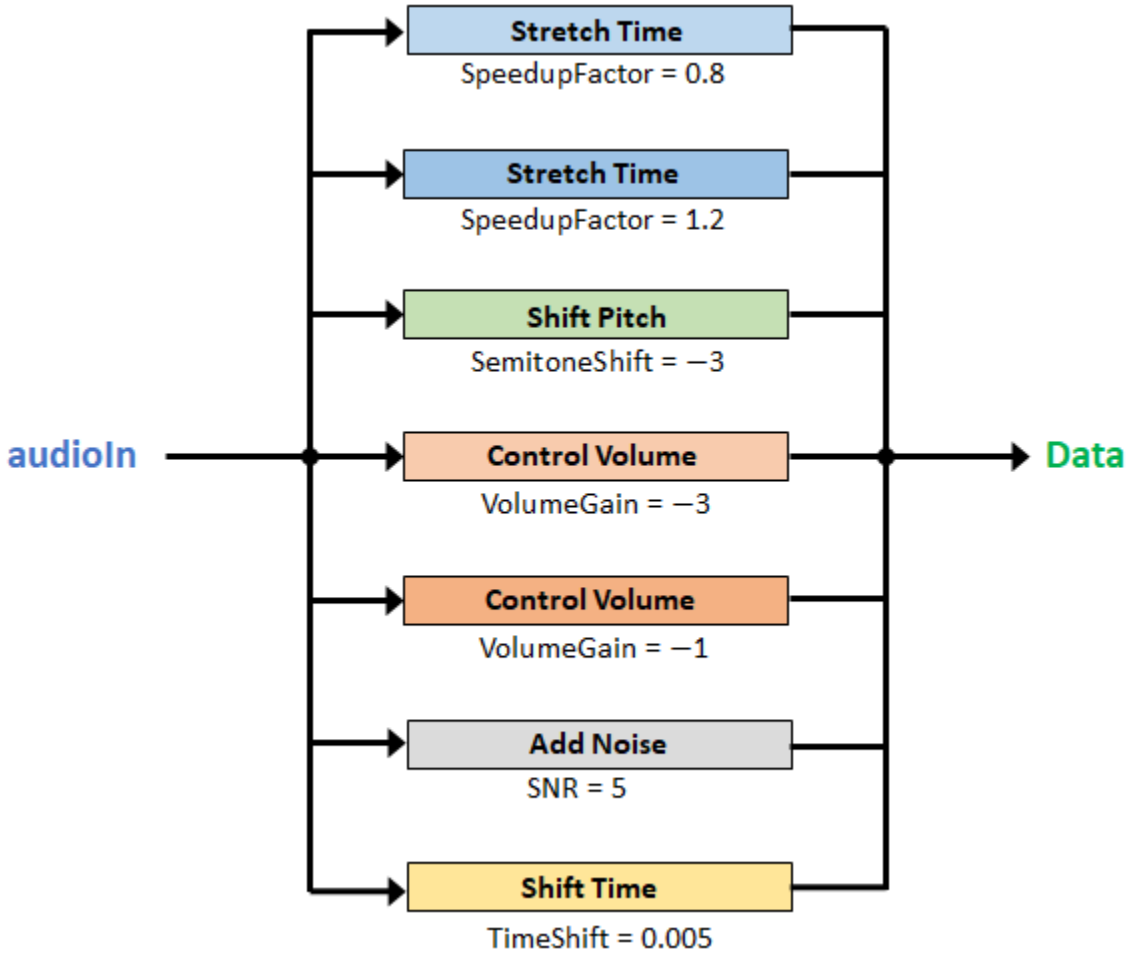
Augmentation Method	Parameters
Stretch Time	ApplyTimeStretch SpeedupFactor

Augmentation Method	Parameters
Shift Pitch	ApplyPitchShift SemitoneShift
Control Volume	ApplyVolumeControl VolumeGain
Add Noise	ApplyAddNoise SNR
Shift Time	ApplyTimeShift TimeShift

If you specify an augmentation method as a vector, then each element of the vector creates a separate branch in the augmentation pipeline. For example, the following object creates an augmentation pipeline that results in seven separate augmentations:

```
aug = audioDataAugmenter("AugmentationMode","independent", ...
    "AugmentationParameterSource","specify", ...
    "SpeedupFactor",[0.8,1.2], ...
    "VolumeGain",[-3,-1])
```

```
aug =

  audioDataAugmenter with properties:

              AugmentationMode: "independent"
    AugmentationParameterSource: "specify"
              ApplyTimeStretch: 1
                 SpeedupFactor: [0.8000 1.2000]
               ApplyPitchShift: 1
                 SemitoneShift: -3
            ApplyVolumeControl: 1
                    VolumeGain: [-3 -1]
                ApplyAddNoise: 1
                           SNR: 5
               ApplyTimeShift: 1
                    TimeShift: 0.0050
```



Version History

Introduced in R2019b

References

[1] Salamon, Justin, and Juan Pablo Bello. "Deep Convolutional Neural Networks and Data Augmentation for Environmental Sound Classification." *IEEE Signal Processing Letters*. Vol. 24, Issue 3, 2017.

Extended Capabilities

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Usage notes and limitations:

- `LockPhase` must be set to `false` for the time stretching and pitch shifting augmentations. For more information, see `setAugmenterParams`.

- Using `gpuArray` (Parallel Computing Toolbox) input with `audioDataAugmenter` is only recommended for a GPU with compute capability 7.0 ("Volta") or above. Other hardware might not offer any performance advantage. To check your GPU compute capability, see `ComputeCompability` in the output from the `gpuDevice` (Parallel Computing Toolbox) function. For more information, see “GPU Computing Requirements” (Parallel Computing Toolbox).

For an overview of GPU usage in MATLAB, see “Run MATLAB Functions on a GPU” (Parallel Computing Toolbox).

See Also

`shiftPitch` | `stretchAudio` | `audioTimeScaler` | `audioFeatureExtractor`

writeall

Write datastore to files

Syntax

```
writeall(ADS,outputLocation)
writeall(ADS,outputLocation,Name,Value)
```

Description

`writeall(ADS,outputLocation)` writes the data from the audio datastore `ADS` to files located at `outputLocation`.

`writeall(ADS,outputLocation,Name,Value)` writes the data with additional options specified by one or more name-value pair arguments.

Example: `writeall(ADS,outputLocation,OutputFormat="flac")` writes the data to FLAC files.

Examples

Write Audio Data Set to New Location

Create an `audioDatastore` object that points to the WAV audio samples included with Audio Toolbox™. The `audioDatastore` object includes read-only properties indicating the supported file formats, and the default output format (WAV).

```
folder = fullfile(matlabroot,'toolbox','audio','samples');
ads = audioDatastore(folder,'FileExtensions','.wav')
```

```
ads =
  audioDatastore with properties:

                      Files: {
                             'B:\matlab\toolbox\audio\samples\Ambiance-16-44p1-mono-12secs.wav'
                             'B:\matlab\toolbox\audio\samples\AudioArray-16-16-4channels-20secs
                             ' ...\toolbox\audio\samples\ChurchImpulseResponse-16-44p1-mono-5se
                              ... and 17 more
                             }
                    Folders: {
                             'B:\matlab\toolbox\audio\samples'
                             }
    AlternateFileSystemRoots: {}
              OutputDataType: 'double'
                      Labels: {}
      SupportedOutputFormats: ["wav"    "flac"    "ogg"    "opus"    "mp4"    "m4a"]
         DefaultOutputFormat: "wav"
```

Write the audio data set to your current folder. Save all files in the default (WAV) format.

```
outputLocation = pwd;
writeall(ads,outputLocation)
```

The folder, `samples`, and the audio files that the folder contains have been written to your current folder.

```
dir samples
```

```
.                                                   ..                                                  Ambia
```

Pre-Extract Features from Audio Data Set

You can use pre-extracted features to reduce iteration time when developing a machine learning or deep learning system. It is also a common practice to use pre-extracted features for unsupervised learning tasks such as similarity clustering, and for content-based indexing tasks such as music information retrieval (MIR).

Create an `audioDatastore` object that points to the audio samples included with Audio Toolbox™.

```
folder = fullfile(matlabroot,"toolbox","audio","samples");
ads = audioDatastore(folder)
```

```
ads =
  audioDatastore with properties:

                      Files: {
                             'B:\matlab\toolbox\audio\samples\Ambiance-16-44p1-mono-12secs.wav'
                             'B:\matlab\toolbox\audio\samples\AudioArray-16-16-4channels-20secs
                             ' ...\toolbox\audio\samples\ChurchImpulseResponse-16-44p1-mono-5se
                              ... and 33 more
                             }
                    Folders: {
                             'B:\matlab\toolbox\audio\samples'
                             }
    AlternateFileSystemRoots: {}
              OutputDataType: 'double'
                      Labels: {}
      SupportedOutputFormats: ["wav"    "flac"    "ogg"    "opus"    "mp4"    "m4a"]
         DefaultOutputFormat: "wav"
```

Create a custom write function that extracts mel frequency cepstral coefficients (`mfcc`) from the audio and writes the them to a MAT file. The function definition is located at the end of this example.

```
function myWriter(audioIn,info,~)
    fs = info.ReadInfo.SampleRate;

    % Extract MFCC
    [coeffs,delta,deltaDelta] = mfcc(audioIn,fs);

    % Replace the file extension of the suggested output name with MAT.
    filename = strrep(info.SuggestedOutputName,".wav",".mat");

    % Save the MFCC coefficients to the MAT file.
    save(filename,"coeffs","delta","deltaDelta")
end
```

Define the output location for the extracted features.

```
outputLocation = pwd;
```

Call the `writeall` function with the `audioDatastore` object, output location, and custom write function. Also specify the suffix `_MFCC` to the file names.

```
tic
writeall(ads,outputLocation,"WriteFcn",@myWriter,"FilenameSuffix","_MFCC")
fprintf("Data set creation completed (%0.0f seconds)\n",toc)
```

```
Data set creation completed (14 seconds)
```

You have now created a data set consisting of MFCCs for each audio file.

```
fds = fileDatastore(pwd,"ReadFcn",@load,"FileExtensions",".mat","IncludeSubfolders",true)
```

```
fds =
  FileDatastore with properties:

                      Files: {
                             ' ...\audio-ex80013303\samples\Ambiance-16-44p1-mono-12secs_MFCC.ma
                             ' ...\audio-ex80013303\samples\AudioArray-16-16-4channels-20secs_MF
                             ' ...\samples\ChurchImpulseResponse-16-44p1-mono-5secs_MFCC.mat'
                              ... and 33 more
                             }
                    Folders: {
                             'C:\TEMP\Bdoc23a_2213998_3568\ib570499\18\tpb47fd747\audio-ex800133
                             }
                UniformRead: 0
                   ReadMode: 'file'
                  BlockSize: Inf
                 PreviewFcn: @load
      SupportedOutputFormats: ["txt"    "csv"    "xlsx"    "xls"    "parquet"    "parq"    "png"
                     ReadFcn: @load
    AlternateFileSystemRoots: {}
```

Helper Function

```
function myWriter(audioIn,info,~)
    fs = info.ReadInfo.SampleRate;
    [coeffs,delta,deltaDelta] = mfcc(audioIn,fs);
    filename = strrep(info.SuggestedOutputName,".wav",".mat");
    save(filename,"coeffs","delta","deltaDelta")
end
```

Augment Audio Data Set

Create an `audioDatastore` object that points to the audio samples included with Audio Toolbox™.

```
folder = fullfile(matlabroot,"toolbox","audio","samples");
ads = audioDatastore(folder);
```

Create an `audioDataAugmenter` object that outputs two augmentations for every input signal.

```
augmenter = audioDataAugmenter(NumAugmentations=2);
```

Define a custom write function, `myWriter` on page 4-143, that applies the `audioDataAugmenter` object to an audio file and writes the resulting new signals to separate files.

Call the `writeall` function to create a new augmented data set. To speed up processing, set `UseParallel` to `true`.

```
outputLocation = pwd;
writeall(ads,outputLocation,FilenameSuffix="_Aug", ...
    UseParallel=true,WriteFcn=@(x,y,z,a)myWriter(x,y,z,augmenter))
```

Create a new datastore that points to the augmented audio data set.

```
adsAug = audioDatastore(outputLocation,IncludeSubfolders=true);
```

`myWriter` Helper Function

```
function myWriter(audioIn,info,fileExtension,varargin)
    % Create convenient variables for the augmenter and sample rate
    augmenter = varargin{1};
    fs = info.ReadInfo.SampleRate;
    % Perform augmentation
    augmentations = augment(augmenter,audioIn,fs);
    for ii = 1:augmenter.NumAugmentations
        x = augmentations.Audio{ii};
        % Protect against clipping
        if any(x<-1|x>1)
            x = x./max(abs(x));
        end
        % Update the audio file name to include the augmentation number
        filename = insertBefore(info.SuggestedOutputName,("."+fileExtension),string(ii));
        % Write the audio file
        audiowrite(filename,x,fs)
    end
end
```

Segment Audio Data

Use the `detectSpeech` and `writeall` functions to create a new audio data set that contains isolated speech segments.

Create an `audioDatastore` object that points to the audio samples included with this example.

```
ads = audioDatastore(pwd)
```

```
ads =
  audioDatastore with properties:

                      Files: {
                             ' ...\18\tpb47fd747\audio-ex78151030\KeywordSpeech-16-16-mono-34sec
                             ' ...\ib570499\18\tpb47fd747\audio-ex78151030\Plosives.wav';
                             ' ...\ib570499\18\tpb47fd747\audio-ex78151030\Sibilance.wav'
                             }
                    Folders: {
                             'C:\TEMP\Bdoc23a_2213998_3568\ib570499\18\tpb47fd747\audio-ex78151
                             }
    AlternateFileSystemRoots: {}
```

```
               OutputDataType: 'double'
                       Labels: {}
      SupportedOutputFormats: ["wav"    "flac"    "ogg"    "opus"    "mp4"    "m4a"]
         DefaultOutputFormat: "wav"
```

Define a custom write function that first determines the regions of speech in the audio signals read from the datastore, then writes the individual regions of speech to separate files. Append the region number to the file names. The function definition is located at the end of this example.

```
function myWriter(audioIn,info,fileExtension)
    fs = info.ReadInfo.SampleRate;

    % Get indices corresponding to regions of speech
    idx = detectSpeech(audioIn,fs);

    % For each region of speech
    for ii = 1:size(idx,1)
        x = audioIn(idx(ii,1):idx(ii,2),:);

        % Create a unique file name
        filename = insertBefore(info.SuggestedOutputName,("."+fileExtension),string(ii));

        % Write the detected region of speech
        audiowrite(filename,x,fs)
    end
end
```

Call the `writeall` function using the custom write function to create a new data set that consists of the isolated speech segments. Create a folder named `segmented` in your temporary directory and then write the data to that folder.

```
outputLocation = fullfile(tempdir,"segmented");
writeall(ads,outputLocation,'WriteFcn',@myWriter)
```

Create a new `audioDatastore` object that points to the segmented data set.

```
adsSegmented = audioDatastore(outputLocation,"IncludeSubfolders",true)
```

```
adsSegmented =
  audioDatastore with properties:

                      Files: {
                             ' ...\18\segmented\audio-ex78151030\KeywordSpeech-16-16-mono-34secs
                             ' ...\18\segmented\audio-ex78151030\KeywordSpeech-16-16-mono-34secs
                             ' ...\18\segmented\audio-ex78151030\KeywordSpeech-16-16-mono-34secs
                              ... and 24 more
                             }
                    Folders: {
                             'C:\TEMP\Bdoc23a_2213998_3568\ib570499\18\segmented'
                             }
    AlternateFileSystemRoots: {}
               OutputDataType: 'double'
                       Labels: {}
      SupportedOutputFormats: ["wav"    "flac"    "ogg"    "opus"    "mp4"    "m4a"]
         DefaultOutputFormat: "wav"
```

Read a sample from the datastore and listen to it.

```
[audioIn,adsInfo] = read(adsSegmented);
sound(audioIn,adsInfo.SampleRate)
```

Supporting Function

```
function myWriter(audioIn,info,fileExtension)
    fs = info.ReadInfo.SampleRate;
    idx = detectSpeech(audioIn,fs);
    for ii = 1:size(idx,1)
        x = audioIn(idx(ii,1):idx(ii,2),:);
        filename = insertBefore(info.SuggestedOutputName,("."+fileExtension),string(ii));
        audiowrite(filename,x,fs)
    end
end
```

Clean Audio Data Set

Audio data sets, especially open-source audio data sets, might have inconsistent sampling rates, numbers of channels, or durations. They might also contain garbage data, such as clips that are labeled as containing speech but contain silence.

It is often a first step in machine learning and deep learning workflows to clean the audio data set. This is particularly important for the validation and test data sets. Common types of cleaning include resampling, converting to mono or stereo, trimming or expanding the duration of clips to a consistent length, removing periods of silence, removing background noise, or gain normalization.

In this example, you clean an audio data set so that all the files have a sample rate of 16 kHz, are mono, are in the FLAC format, and are normalized such that the max absolute magnitude of a signal is 1.

Create an `audioDatastore` object that points to the audio samples included with Audio Toolbox™.

```
folder = fullfile(matlabroot,"toolbox","audio","samples");
ads = audioDatastore(folder);
```

Define a function, `myTransform` on page 4-145, that applies a sequence of operations on the audio data. Create a transform datastore object that applies the cleaning operations.

```
adsTransform = transform(ads,@myTransform,IncludeInfo=true);
```

Call `writeall` on the transform datastore object to create the clean data set. Specify the format as `FLAC`. Write the data set to your current folder.

```
outputLocation = pwd;
writeall(adsTransform,outputLocation,OutputFormat="flac")
```

Create a new datastore object that points to the clean data set.

```
adsClean = audioDatastore(outputLocation,IncludeSubfolders=true);
```

`myTransform` Supporting Function

```
function [audioOut,adsInfo] = myTransform(audioIn,adsInfo)
    fs = adsInfo.SampleRate;
    desiredFs = 16e3;
```

```
    % Convert to mono
    x = mean(audioIn,2);

    % Resample to 16 kHz
    y = resample(x,desiredFs,fs);
    adsInfo.SampleRate = desiredFs;

    % Normalize so that the max absolute value of a signal is 1
    audioOut = y/max(abs(y));
end
```

Input Arguments

`ADS` — Audio datastore

`audioDatastore` object

Audio datastore, specified as an `audioDatastore` object.

`outputLocation` — Folder location to write data

character vector | string

Folder location to write data, specified as a character vector or string. You can specify a full or relative path in `outputLocation`.

Example: `outputLocation = "../../dir/data"`

Example: `outputLocation = "C:\Users\MyName\Desktop"`

Data Types: `char` | `string`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Example: `FolderLayout="flatten"`

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `'FolderLayout','flatten'`

`FolderLayout` — Layout of files in output folder

`"duplicate"` (default) | `"flatten"`

Layout of files in output folder, specified as `"duplicate"` or `"flatten"`.

- `"duplicate"` -– Replicate the folder structure of the data that the audio datastore points to. Specify the `FolderLayout` as `"duplicate"` to maintain correspondence between the input and output data sets.
- `"flatten"` -– Write all the files from the input to the specified output folder without any intermediate folders.

Data Types: `char` | `string`

`OutputFormat` — Output file format

`"wav"` (default) | `"flac"` | `"ogg"` | `"opus"` | `"mp4"` | `"m4a"`

Output file format, specified as `"wav"`, `"flac"`, `"ogg"`, `"opus"`, `"mp4"`, or `"m4a"`.

Data Types: `char` | `string`

`BitsPerSample` — Number of output bits per sample

`16` (default) | `8` | `24` | `32` | `64`

Number of output bits per sample, specified as an integer.

Dependencies

To enable this name-value pair argument, set `OutputFormat` to `"wav"` or `"flac"`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`BitRate` — Kilobits per second (kbit/s)

`128` (default) | `64` | `96` | `160` | `192` | `256` | `320`

Number of kilobits per second (kbit/s) used to compress audio files, specified as an integer. On Windows 7 or later, the only valid values are `96`, `128`, `160`, and `192`.

In general, a larger `BitRate` value results in higher compression quality.

Dependencies

To enable this name-value pair argument, set `OutputFormat` to `"m4a"` or `"mp4"`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`FilenamePrefix` — Prefix added to file name

character vector | string

Prefix added to file name, specified a character vector or string.

The `writeall` function adds the specified prefix to the output file names. For example, the following code adds today's date as the prefix to all the output file names:

```
prefixText = string(datetime("today"));
writeall(ADS,"C:\myFolder",FilenamePrefix=prefixText);
```

Data Types: `char` | `string`

`FilenameSuffix` — Suffix added to file name

character vector | string

Suffix added to the file name, specified as character vector or string. The file name suffix is applied before the file extension.

The `writeall` function adds the specified suffix to the output file names. For example, the following code adds the descriptive text `"clean"` as the suffix to all the output file names:

```
writeall(ADS,"C:\myFolder",FilenameSuffix="clean");
```

Data Types: `char` | `string`

`UseParallel` — Indicator to write in parallel

`false` (default) | `true`

Indicator to write in parallel, specified as `false` or `true`.

By default, the `writeall` function writes in serial. If you set `UseParallel` to `true`, then the `writeall` function writes the output files in parallel.

Note Writing in parallel requires Parallel Computing Toolbox™.

Data Types: `logical`

`WriteFcn` — Custom write function

function handle

Custom write function, specified as a function handle. The specified function is responsible for creating the output files. You can use `WriteFcn` to write data in a variety of formats, even if `writeall` does not directly support the output format.

Function Signature

The custom write function requires three input arguments: `audioIn`, `info`, and `suggestedOutputType`. The function can also accept additional inputs, such as name-value pairs, after the first three required inputs.

```
function myWriter(audioIn,info,suggestedOutputType,varargin)
```

- `audioIn` contains data read from the input datastore `ADS`.
- `info` is an object of type `matlab.io.datastore.WriteInfo` with fields listed in the table.

Field	Description	Type
`ReadInfo`	The second output of the `read` method.	`struct`
`SuggestedOutputName`	A fully qualified, globally unique file name that meets the location and naming requirements.	`string`
`Location`	The specified `outputLocation` passed to `writeall`.	`string`

- `suggestedOutputType` –- Suggested output file type.

Example Function

A simple write function that resamples audio to 44.1 kHz before writing.

```
function myWriter(data,info,~)
  fs = info.ReadInfo.SampleRate;
  desiredFs = 44.1e3;
  data = resample(data,desiredFs,fs);
  audiowrite(writeInfo.SuggestedOutputName,data,desiredFs);
end
```

To use `myWriter` as in the `writeall` function, use these commands:

```
ads = audioDatastore(location);
outputLocation = "C:/tmp/MyData";
writeall(ads,outputLocation,WriteFcn=@myWriter)
```

Data Types: `function_handle`

Version History

Introduced in R2020a

R2022b: Support for OPUS audio file format

The `audioDatastore` `writeall` function supports OPUS file format (`.opus`).

See Also

`audioDatastore`

transform

Transform audio datastore

Syntax

```
transformDatastore = transform(ADS,@fcn)
transformDatastore = transform(ADS,@fcn,Name,Value)
```

Description

`transformDatastore = transform(ADS,@fcn)` creates a new datastore that transforms output from the `read` function.

`transformDatastore = transform(ADS,@fcn,Name,Value)` specifies options using one or more `Name,Value` pair arguments.

Examples

Output Mono Audio from Datastore

Specify the file path to the audio samples included with Audio Toolbox™. Create an audio datastore that points to the specified folder.

```
folder = fullfile(matlabroot,'toolbox','audio','samples');
ADS = audioDatastore(folder);
```

Call `transform` to create a new datastore that mixes multichannel signals to mono.

```
ADSnew = transform(ADS,@(x)mean(x,2));
```

Read from the new datastore and confirm that it only outputs mono signals.

```
while hasdata(ADSnew)
    audio = read(ADSnew);
    fprintf('Number of channels = %d\n',size(audio,2))
end
```

```
Number of channels = 1
Number of channels = 1
Number of channels = 1
Number of channels = 1
Number of channels = 1
Number of channels = 1
Number of channels = 1
Number of channels = 1
Number of channels = 1
Number of channels = 1
Number of channels = 1
Number of channels = 1
Number of channels = 1
Number of channels = 1
```

```
Number of channels = 1
Number of channels = 1
Number of channels = 1
Number of channels = 1
Number of channels = 1
Number of channels = 1
Number of channels = 1
Number of channels = 1
Number of channels = 1
Number of channels = 1
Number of channels = 1
Number of channels = 1
Number of channels = 1
Number of channels = 1
Number of channels = 1
Number of channels = 1
Number of channels = 1
Number of channels = 1
Number of channels = 1
Number of channels = 1
Number of channels = 1
Number of channels = 1
```

Clip Audio to Five Seconds

The audio samples included with Audio Toolbox™ have varying durations. Use the `transform` function to customize the `read` function so that it outputs a random five second segment of the audio samples.

Specify the file path to the audio samples included with Audio Toolbox. Create an audio datastore that points to the specified folder.

```
folder = fullfile(matlabroot,'toolbox','audio','samples');
ADS = audioDatastore(folder);
```

Define a function to take as input the output of the `read` function. Make the function extract five seconds worth of data from the audio signal.

```
function [dataOut,info] = extractSegment(audioIn,info)
    [N,numChan] = size(audioIn);
    newN = round(info.SampleRate*5);
    if newN > N                                 % signal length < 5 seconds
        numPad = newN - N + 1;
        dataOut = [audioIn;zeros(numPad,numChan,'like',audioIn)];
    elseif newN < N                             % signal length > 5 seconds
        start = randi(N - newN + 1);
        dataOut = audioIn(start:start+newN-1,:);
    else                                        % signal length == 5 seconds
        dataOut = audioIn;
    end
end
```

Call `transform` to create a `TransformedDatastore` with `Transforms` set to the function you defined.

```
ADSnew = transform(ADS,@extractSegment,'IncludeInfo',true)

ADSnew =
  TransformedDatastore with properties:

      UnderlyingDatastores: {audioDatastore}
    SupportedOutputFormats: ["txt"    "csv"    "xlsx"    "xls"    "parquet"    "parq"    "png"
                Transforms: {@extractSegment}
               IncludeInfo: 1
```

Read the first three audio files and verify that the outputs are five second segments.

```
for i = 1:3
    [audio,info] = read(ADSnew);
    fprintf('Duration = %d seconds\n',size(audio,1)/info.SampleRate)
end

Duration = 5 seconds
Duration = 5 seconds
Duration = 5 seconds
```

Output Mel Spectrogram

Use `transform` to create an audio datastore that returns a mel spectrogram representation from the `read` function.

Specify the file path to the audio samples included with Audio Toolbox™. Create an audio datastore that points to the specified folder.

```
folder = fullfile(matlabroot,'toolbox','audio','samples');
ADS = audioDatastore(folder);
```

Define a function that transforms audio data from a time-domain representation to a log mel spectrogram. The function adds the additional outputs from the `melSpectrogram` function to the info struct output from reading the audio datastore.

```
function [dataOut,infoOut] = extractMelSpectrogram(audioIn,info)

    [S,F,T] = melSpectrogram(audioIn,info.SampleRate);

    dataOut = 10*log10(S+eps);
    infoOut = info;
    infoOut.CenterFrequencies = F;
    infoOut.TimeInstants = T;
end
```

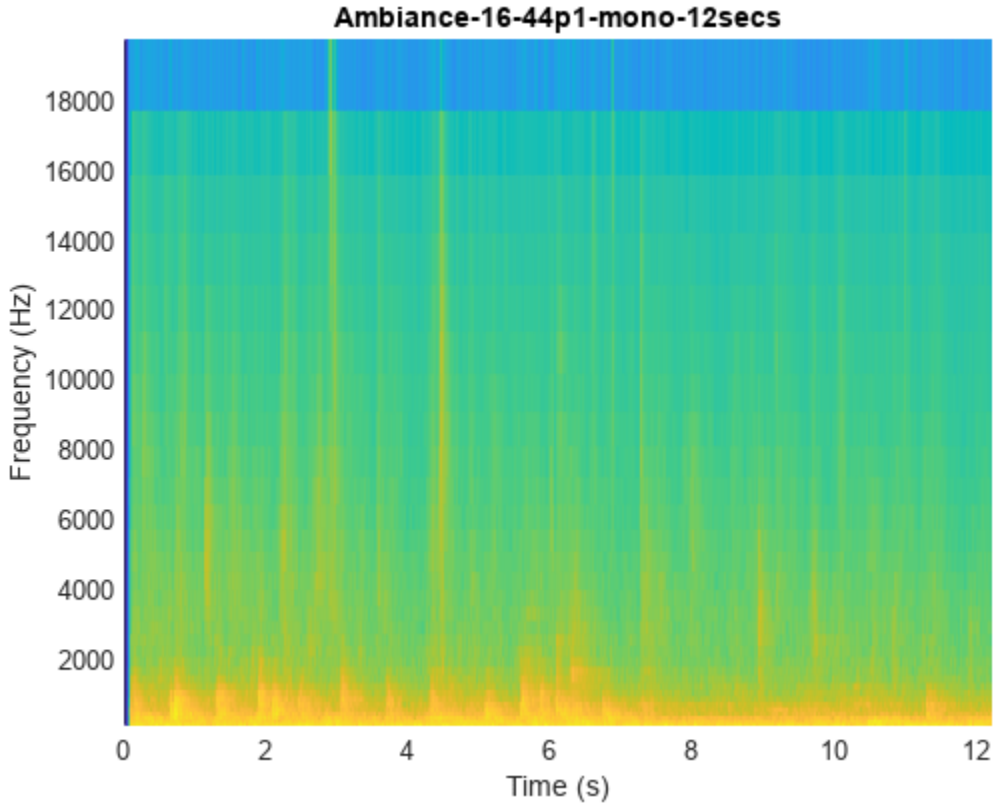
Call `transform` to create a `TransformedDatastore` with `Transforms` set to `extractMelSpectrogram`.

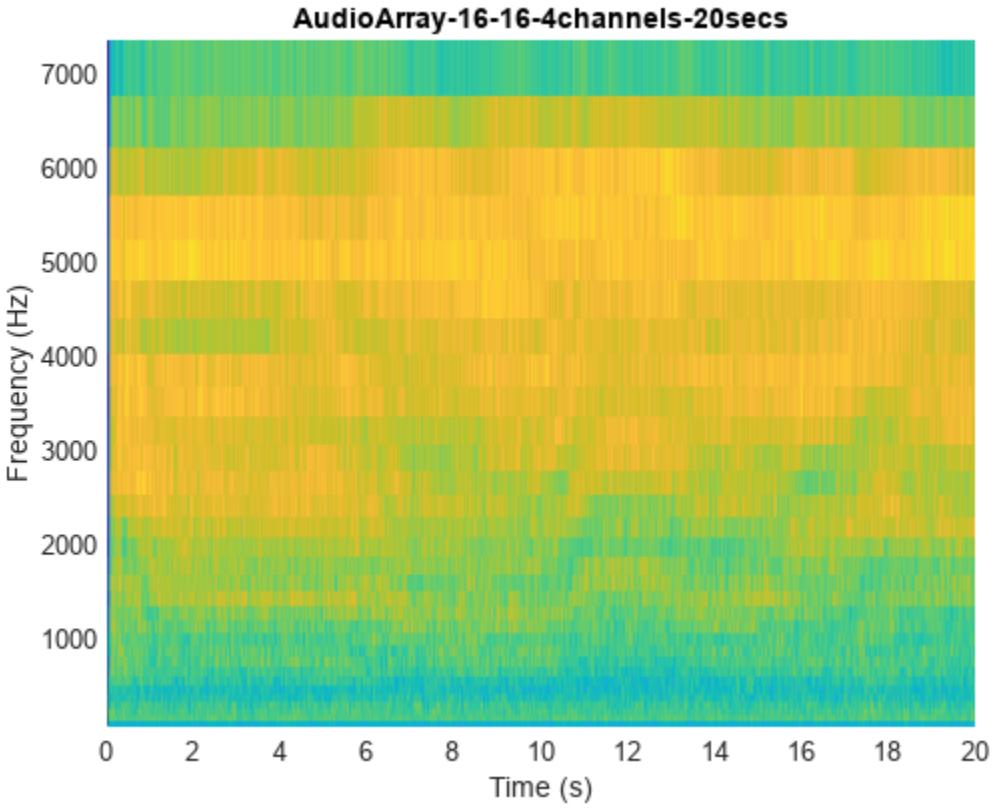
```
ADSnew = transform(ADS,@extractMelSpectrogram,'IncludeInfo',true)

ADSnew =
  TransformedDatastore with properties:

      UnderlyingDatastores: {audioDatastore}
```

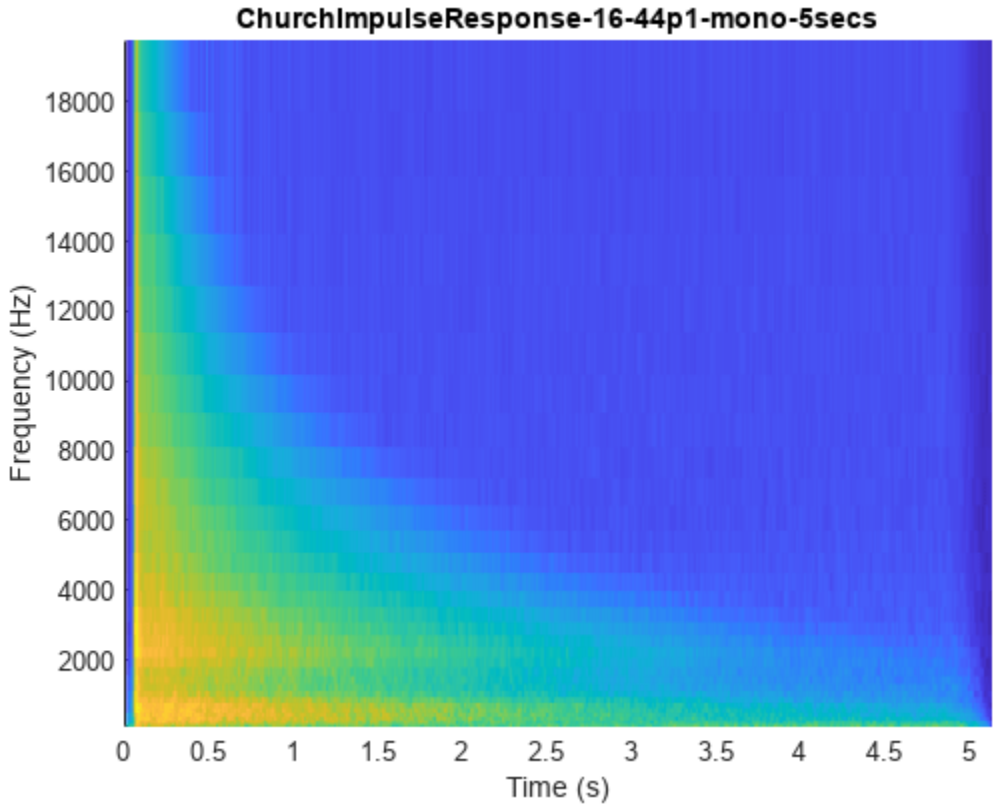
```
    SupportedOutputFormats: ["txt"    "csv"    "xlsx"    "xls"    "parquet"    "parq"    "png"
                Transforms: {@extractMelSpectrogram}
               IncludeInfo: 1
```

Read the first three audio files and plot the log mel spectrograms. If there are multiple channels, plot only the first channel.

```
for i = 1:3
    [melSpec,info] = read(ADSnew);

    figure(i)
    surf(info.TimeInstants,info.CenterFrequencies,melSpec(:,:,1),'EdgeColor','none');
    xlabel('Time (s)')
    ylabel('Frequency (Hz)')
    [~,name] = fileparts(info.FileName);
    title(name)
    axis([0 info.TimeInstants(end) info.CenterFrequencies(1) info.CenterFrequencies(end)])
    view([0,90])
end
```




AudioArray-16-16-4channels-20secs
Frequency (Hz)
7000
6000
5000
4000
3000
2000
1000
0
2
4
6
8
10
12
14
16
18
20
Time (s)



Output Spectral Shape Features

Use `transform` to create an audio datastore that returns feature vectors.

Specify the file path to the audio samples included with Audio Toolbox™. Create an audio datastore that points to the specified folder.

```
folder = fullfile(matlabroot,'toolbox','audio','samples');
ADS = audioDatastore(folder);
```

Define a function, `extractFeatureVector`, that transforms the audio data from a time-domain representation to feature vectors.

```
function [dataOut,info] = extractFeatureVector(audioIn,info)

    % Convert to frequency-domain representation
    windowLength = 256;
    overlapLength = 128;
    [~,f,~,S] = spectrogram(mean(audioIn,2), ...
                            hann(windowLength,"Periodic"), ...
                            overlapLength, ...
                            windowLength, ...
                            info.SampleRate, ...
                            "power", ...
```

```
                                "onesided");

    % Extract features
    [kurtosis,spread,centroid] = spectralKurtosis(S,f);
    skewness = spectralSkewness(S,f);
    crest    = spectralCrest(S,f);
    decrease = spectralDecrease(S,f);
    entropy  = spectralEntropy(S,f);
    flatness = spectralFlatness(S,f);
    flux     = spectralFlux(S,f);
    rolloff  = spectralRolloffPoint(S,f);
    slope    = spectralSlope(S,f);

    % Concatenate to create feature vectors
    dataOut = [kurtosis,spread,centroid,skewness,crest,decrease,entropy,flatness,flux,rolloff,slo

end
```

Call `transform` to create a `TransformedDatastore` with `Transforms` set to `extractFeatureVector`.

```
ADSnew = transform(ADS,@extractFeatureVector,'IncludeInfo',true)
```

```
ADSnew =

  TransformedDatastore with properties:

      UnderlyingDatastores: {audioDatastore}
    SupportedOutputFormats: ["txt"    "csv"    "xlsx"    "xls"    ...    ]
                Transforms: {@extractFeatureVector}
               IncludeInfo: 1
```

Call `read` to return the feature vectors for the audio over time.

```
featureMatrix = read(ADSnew);
[numFeatureVectors,numFeatures] = size(featureMatrix)
```

```
numFeatureVectors =

        4215


numFeatures =

    11
```

Apply Bandpass Filtering

Use `transform` to create an audio datastore that applies bandpass filtering before returning audio from the `read` function.

Specify the file path to the audio samples included with Audio Toolbox™. Create an audio datastore that points to the specified folder.

```
folder = fullfile(matlabroot,'toolbox','audio','samples');
ADS = audioDatastore(folder);
```

Define a function, `applyBandpassFilter`, that applies a bandpass filter with a passband between 1 and 15 kHz.

```
function [audioOut,info] = applyBandpassFilter(audioIn,info)

    audioOut = bandpass(audioIn,[1e3,15e3],info.SampleRate);

end
```

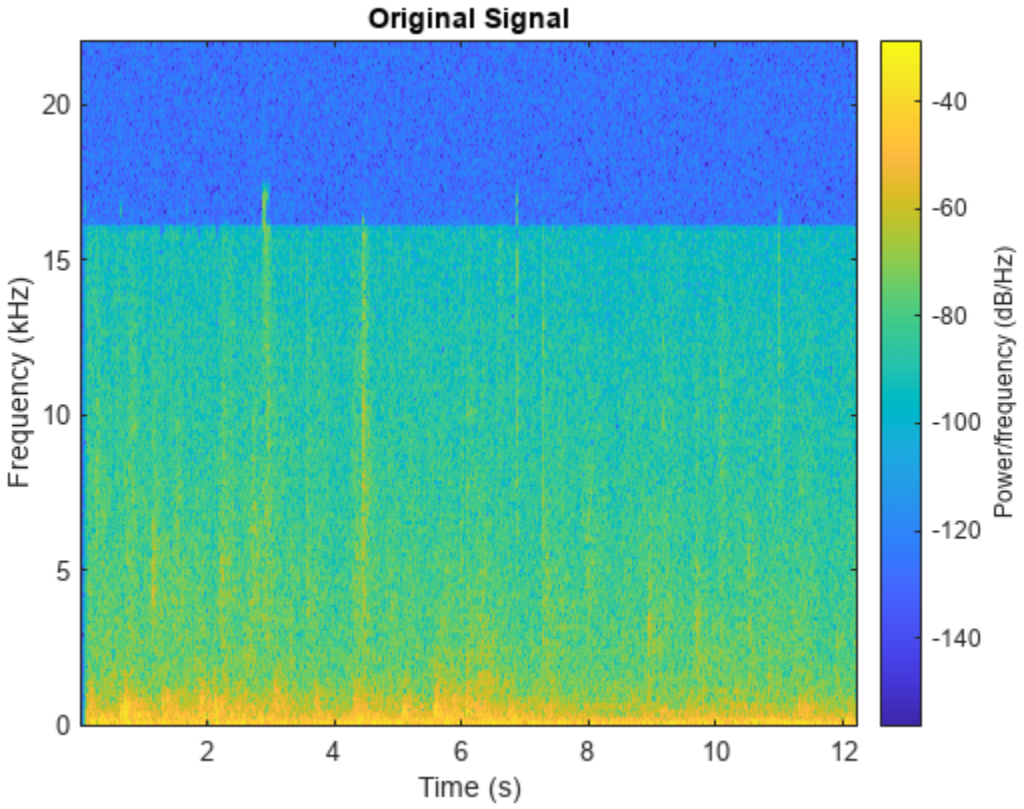
Call `transform` to create a `TransformedDatastore` with `Transforms` set to `applyBandpassFilter`.

```
ADSnew = transform(ADS,@applyBandpassFilter,'IncludeInfo',true)
```

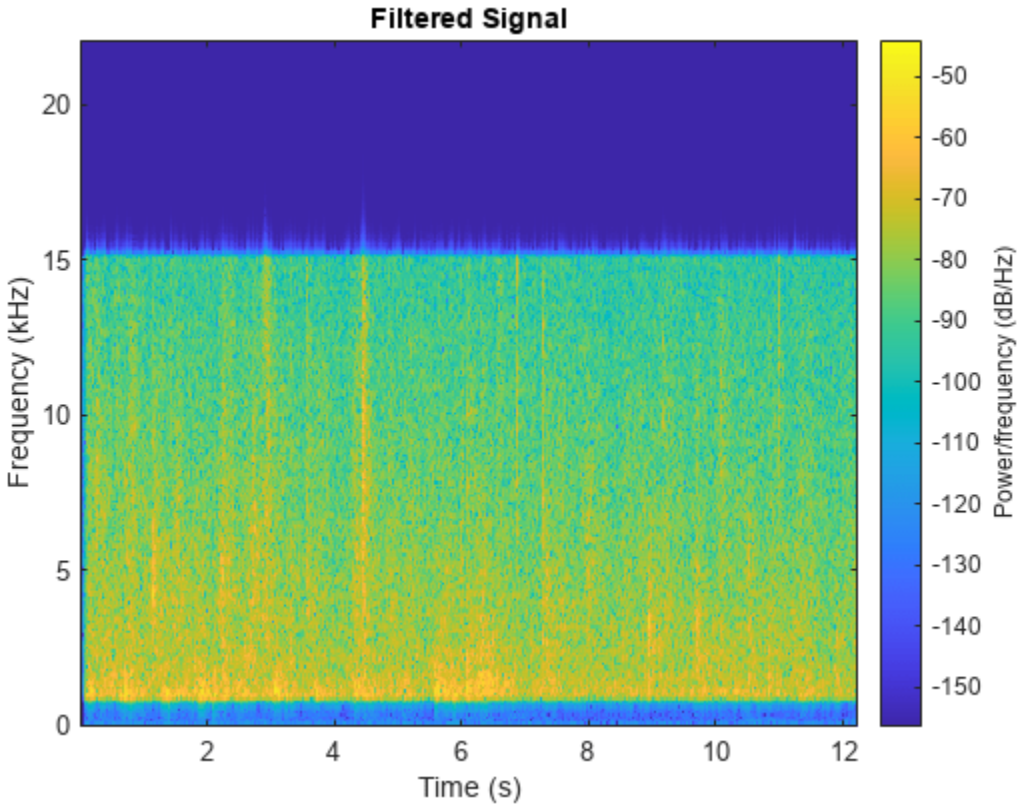
```
ADSnew =
  TransformedDatastore with properties:

      UnderlyingDatastores: {audioDatastore}
    SupportedOutputFormats: ["txt"    "csv"    "xlsx"    "xls"    "parquet"    "parq"    "png"
                Transforms: {@applyBandpassFilter}
               IncludeInfo: 1
```

Call `read` to return the bandpass filtered audio from the transform datastore. Call `read` to return the bandpass filtered audio from the original datastore. Plot the spectrograms to visualize the difference.

```
[audio1,info1] = read(ADS);
[audio2,info2] = read(ADSnew);

spectrogram(audio1,hann(512),256,512,info1.SampleRate,'yaxis')
title('Original Signal')
```




```
spectrogram(audio2,hann(512),256,512,info2.SampleRate,'yaxis')
title('Filtered Signal')
```



Input Arguments

ADS — Audio datastore

`audioDatastore` object

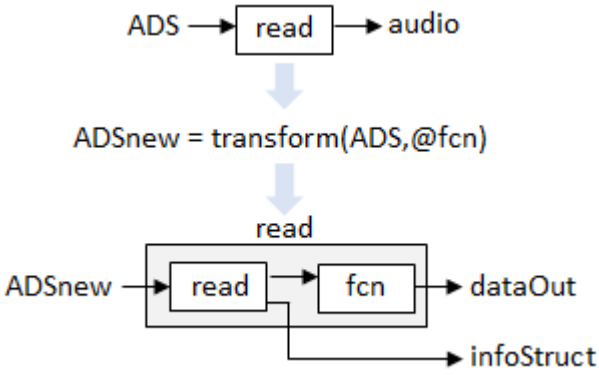
Audio datastore, specified as an `audioDatastore` object.

@fcn — Function that transforms data

function handle

Function that transforms data, specified as a function handle. The signature of the function depends on the `IncludeInfo` parameter.

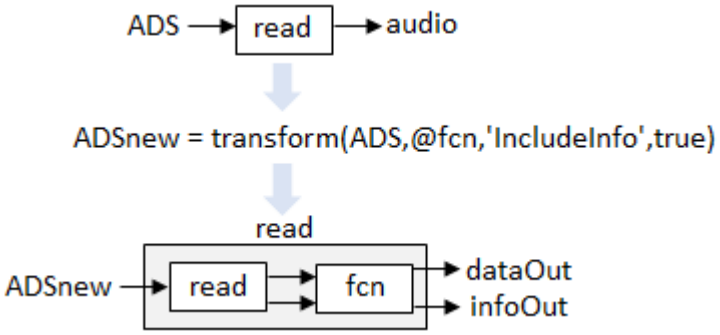
- If `IncludeInfo` is set to `false` (default), the function transforms the audio output from `read`. The info output from `read` is unaltered.



The transform function must have this signature:

```
function dataOut = fcn(audio)
...
end
```

- If `IncludeInfo` is set to `true`, the function transforms the audio output from `read`, and can use or modify the information returned from `read`.



The transform function must have this signature:

```
function [dataOut,infoOut] = fcn(audio,infoIn)
...
end
```

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `'IncludeInfo',tf`

`IncludeInfo` — Pass info through customized `read` function
`false` (default) | true

Pass info through the customized read function, specified as `true` or `false`. If `true`, the transform function can use or modify the information it gets from `read`. If unspecified, `IncludeInfo` defaults to `false`.

Data Types: `logical`

Output Arguments

`transformDatastore` — New datastore with customized `read`
`TransformedDatastore`

New datastore with customized `read`, returned as a `TransformedDatastore` with `UnderlyingDatastore` set to `ADS`, `Transforms` set to `fcn`, and `IncludeInfo` set to `true` or `false`.

Version History

Introduced in R2019a

See Also

`audioDatastore` | `combine` | `hasdata` | `preview` | `read` | `readall` | `reset`

combine

Combine data from multiple datastores

Syntax

```
ADSnew = combine(ADS1,ADS2,...,ADSN)
```

Description

`ADSnew = combine(ADS1,ADS2,...,ADSN)` combines two or more datastores by horizontally concatenating the data returned by `read` of the input datastores.

Examples

Combine Datastores

Create a datastore that maintains parity between the audio of the underlying datastores. Create two separate audio datastores, and then create a combined datastore representing the two underlying datastores.

Create a datastore `ads1` that points to the audio files included with Audio Toolbox.

```
folder = fullfile(matlabroot,'toolbox','audio','samples');
ads1 = audioDatastore(folder);
```

Create a second datastore `ads2` by adding noise to the audio in the `ads1`.

```
ads2 = transform(ads1,@(x) x + 0.01*randn(size(x)) );
```

Create a combined datastore from `ads1` and `ads2`.

```
adsCombined = combine(ads1,ads2);
```

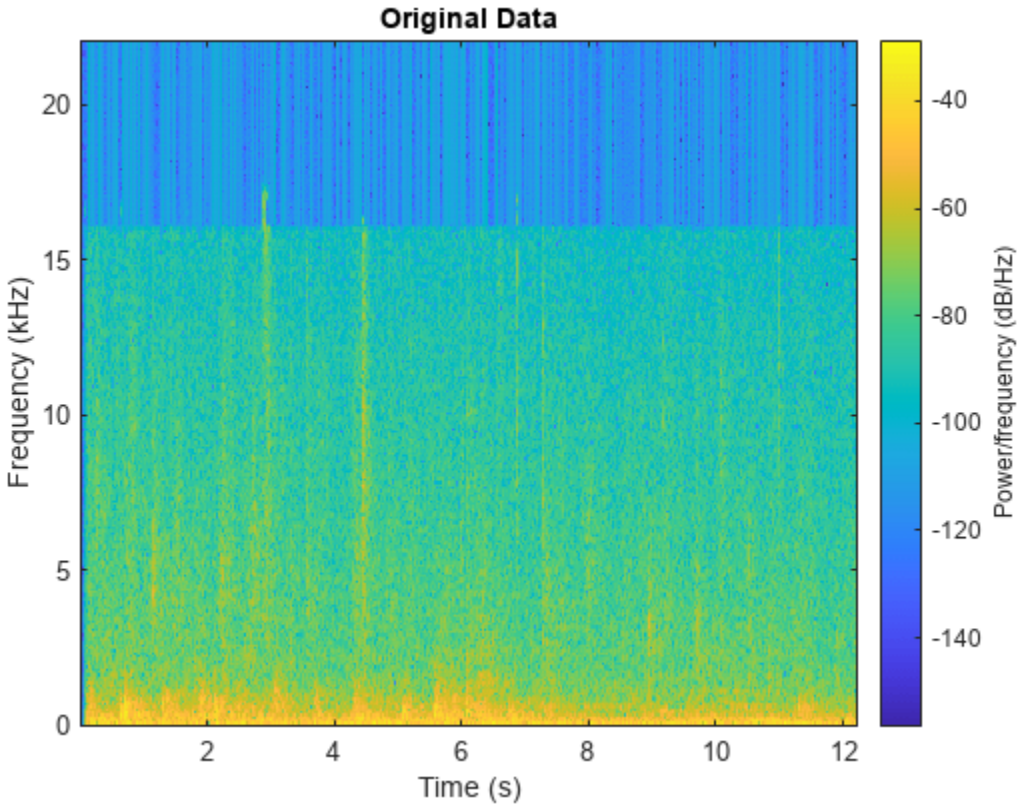
Read the first pair of audio files from the combined datastore. Each `read` operation on this combined datastore returns a pair of audio signals in a 1-by-2 cell array and a pair of info structs in a 1-by-2 cell array.

```
[dataOut,infoOut] = read(adsCombined)
```

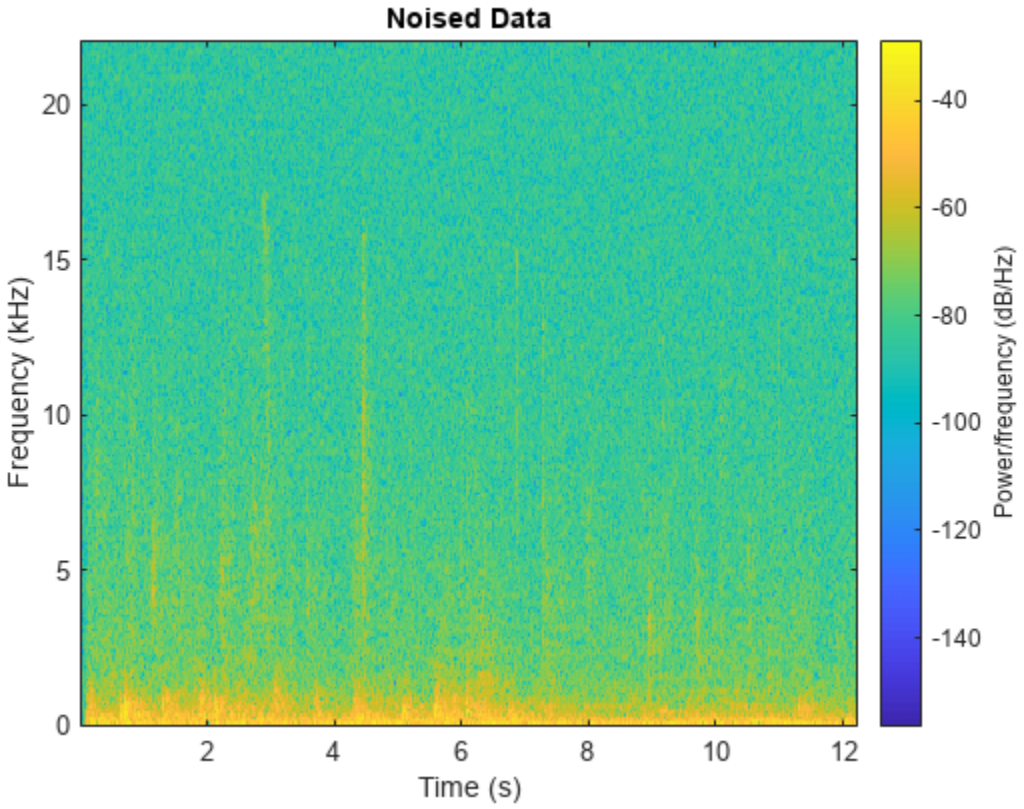
```
dataOut=1×2 cell array
    {539648x1 double}    {539648x1 double}


infoOut=1×2 cell array
    {1x1 struct}    {1x1 struct}
```

Plot the spectrograms of the first channels from both audio signals.

```
figure(1)
spectrogram(dataOut{1},hamming(512),256,512,infoOut{1}.SampleRate,'yaxis')
title('Original Data')
```




```
figure(2)
idx = size(dataOut,2)/2+1;
spectrogram(dataOut{2},hamming(512),256,512,infoOut{2}.SampleRate,'yaxis')
title('Noised Data')
```



Input Arguments

`ADS1,ADS2,...,ADSN` — Audio datastores to combine

`audioDatastore` objects

Audio datastores to combine, specified as two or more comma separated `audioDatastore` objects.

Output Arguments

`ADSnew` — New audio datastore with combined data

`audioDatastore` object

New audio datastore with combined data, returned as a `matlab.io.datastore.CombinedDatastore` object.

Calling `read` on the combined datastore returns a cell array containing the output of calling `read` on the individual datastores.

Version History

Introduced in R2019a

See Also

audioDatastore | transform | hasdata | preview | read | readall | reset

progress

Fraction of files read

Syntax

```
fractionRead = progress(ADS)
```

Description

`fractionRead = progress(ADS)` returns the fraction of files read in the datastore as a normalized value in the range [0,1].

Examples

Return Fraction of Files Read

Create an `audioDatastore` object `ADS`. Read a file from the datastore and then call `progress` to return the fraction of files read.

```
ADS = audioDatastore(fullfile(matlabroot,'toolbox','audio','samples'))
```

```
ADS =
  audioDatastore with properties:

                      Files: {
                             'B:\matlab\toolbox\audio\samples\Ambiance-16-44p1-mono-12secs.wav'
                             'B:\matlab\toolbox\audio\samples\AudioArray-16-16-4channels-20secs
                             ' ...\toolbox\audio\samples\ChurchImpulseResponse-16-44p1-mono-5se
                              ... and 33 more
                             }
                    Folders: {
                             'B:\matlab\toolbox\audio\samples'
                             }
    AlternateFileSystemRoots: {}
              OutputDataType: 'double'
                      Labels: {}
      SupportedOutputFormats: ["wav"    "flac"    "ogg"    "opus"    "mp4"    "m4a"]
         DefaultOutputFormat: "wav"
```

```
fractionOfFilesRead = progress(ADS)
```

```
fractionOfFilesRead = 0
```

```
data = read(ADS);
fractionOfFilesRead = progress(ADS)
```

```
fractionOfFilesRead = 0.0278
```

Input Arguments

`ADS` — Audio datastore
`audioDatastore` object

Specify `ADS` as an `audioDatastore` object.

Output Arguments

`fractionRead` — Fraction of files read
normalized value in the range [0,1]

Fraction of files read, returned as a normalized value in the range [0,1].

Data Types: `double`

Version History

Introduced in R2018b

See Also

`audioDatastore` | `hasdata`

Topics
“Train Speech Command Recognition Model Using Deep Learning”
“Speaker Identification Using Pitch and MFCC”
“Denoise Speech Using Deep Learning Networks”

numpartitions

Return estimate for reasonable number of partitions for parallel processing

Syntax

```
n = numpartitions(ADS)
n = numpartitions(ADS,pool)
```

Description

`n = numpartitions(ADS)` returns the default number of partitions for the datastore, `ADS`. The default number of partitions is the total number of files.

`n = numpartitions(ADS,pool)` returns a reasonable number of partitions to parallelize `ADS` over the parallel pool, based on the total number of files and the number of workers in pool. To parallelize datastore access, you must have Parallel Computing Toolbox installed.

Examples

Estimate Reasonable Number of Partitions for Audio Datastore

`numpartitions` returns a reasonable number of partitions for an audio datastore. You can use `numpartitions` as input to the `partition` function.

Specify the file path to the audio samples included with Audio Toolbox™. Create an audio datastore that points to the specified folder.

```
folder = fullfile(matlabroot,'toolbox','audio','samples');

ADS = audioDatastore(folder);
```

Use `numpartitions` to estimate a reasonable number of partitions for the audio datastore, `ADS`. By default, `numpartitions` returns the number of files the audio datastore points to.

```
n = numpartitions(ADS)
```

```
n = 36
```

Number of Partitions for Parallel Datastore Access

Partition a datastore to facilitate parallel access over the available parallel pool of workers.

Specify the file path to the audio samples included with Audio Toolbox™. Create an audio datastore that points to the specified folder.

```
folder = fullfile(matlabroot,'toolbox','audio','samples');
ADS = audioDatastore(folder);
```

Return an estimate for a reasonable number of partitions for parallel processing, given the current parallel pool.

```
pool = gcp;
n = numpartitions(ADS,pool);
```

Partition the audio datastore and read the data in each part.

```
parfor ii = 1:n
    subds = partition(ADS,n,ii);
    while hasdata(subds)
        data = read(subds);
    end
end
```

Input Arguments

`ADS` — Audio datastore

`audioDatastore` object

Specify `ADS` as an `audioDatastore` object.

`pool` — Parallel pool

parallel pool object

Parallel pool object.

Output Arguments

`n` — Number of partitions

positive integer

Number of partitions to parallelize datastore access over.

Version History

Introduced in R2018b

See Also

`audioDatastore` | `partition`

Topics

“Train Speech Command Recognition Model Using Deep Learning”
“Speaker Identification Using Pitch and MFCC”
“Denoise Speech Using Deep Learning Networks”

partition

Partition datastore and return on partitioned portion

Syntax

```
subADS = partition(ADS,numPartitions,index)
subADS = partition(ADS,'Files',index)
subADS = partition(ADS,'Files',filename)
```

Description

`subADS = partition(ADS,numPartitions,index)` partitions datastore `ADS` into the number of parts specified by `numPartitions` and returns the partition corresponding to the `index`.

`subADS = partition(ADS,'Files',index)` partitions the datastore by files and returns the partition corresponding to the file of index `index` in the `Files` property.

`subADS = partition(ADS,'Files',filename)` partitions the datastore by files and returns the partition corresponding to the file specified by `filename`.

Examples

Partition Datastore into Specific Number of Parts

Specify the file path to the audio samples included with Audio Toolbox™. Create an audio datastore that points to the specified folder.

```
folder = fullfile(matlabroot,'toolbox','audio','samples');
ADS = audioDatastore(folder)
```

```
ADS =
  audioDatastore with properties:

                      Files: {
                             'B:\matlab\toolbox\audio\samples\Ambiance-16-44p1-mono-12secs.wav'
                             'B:\matlab\toolbox\audio\samples\AudioArray-16-16-4channels-20secs
                             ' ...\toolbox\audio\samples\ChurchImpulseResponse-16-44p1-mono-5se
                              ... and 33 more
                             }
                    Folders: {
                             'B:\matlab\toolbox\audio\samples'
                             }
    AlternateFileSystemRoots: {}
              OutputDataType: 'double'
                      Labels: {}
      SupportedOutputFormats: ["wav"    "flac"    "ogg"    "opus"    "mp4"    "m4a"]
         DefaultOutputFormat: "wav"
```

Partition the datastore into three parts.

```
subADS1 = partition(ADS,3,1)

subADS1 =
  audioDatastore with properties:

                      Files: {
                             'B:\matlab\toolbox\audio\samples\Ambiance-16-44p1-mono-12secs.wav'
                             'B:\matlab\toolbox\audio\samples\AudioArray-16-16-4channels-20secs
                             ' ...\toolbox\audio\samples\ChurchImpulseResponse-16-44p1-mono-5se
                              ... and 9 more
                             }
                    Folders: {
                             'B:\matlab\toolbox\audio\samples'
                             }
    AlternateFileSystemRoots: {}
              OutputDataType: 'double'
                      Labels: {}
      SupportedOutputFormats: ["wav"    "flac"    "ogg"    "opus"    "mp4"    "m4a"]
         DefaultOutputFormat: "wav"


subADS2 = partition(ADS,3,2)

subADS2 =
  audioDatastore with properties:

                      Files: {
                             'B:\matlab\toolbox\audio\samples\JetAirplane-16-11p025-mono-16secs
                             'B:\matlab\toolbox\audio\samples\Laughter-16-8-mono-4secs.wav';
                             'B:\matlab\toolbox\audio\samples\MainStreetOne-16-16-mono-12secs.wa
                              ... and 9 more
                             }
                    Folders: {
                             'B:\matlab\toolbox\audio\samples'
                             }
    AlternateFileSystemRoots: {}
              OutputDataType: 'double'
                      Labels: {}
      SupportedOutputFormats: ["wav"    "flac"    "ogg"    "opus"    "mp4"    "m4a"]
         DefaultOutputFormat: "wav"


subADS3 = partition(ADS,3,3)

subADS3 =
  audioDatastore with properties:

                      Files: {
                             'B:\matlab\toolbox\audio\samples\RockGuitar-16-96-stereo-72secs.fla
                             'B:\matlab\toolbox\audio\samples\SingingAMajor-16-mono-18secs.ogg'
                             'B:\matlab\toolbox\audio\samples\SoftGuitar-44p1_mono-10mins.ogg'
                              ... and 9 more
                             }
                    Folders: {
                             'B:\matlab\toolbox\audio\samples'
                             }
    AlternateFileSystemRoots: {}
              OutputDataType: 'double'
                      Labels: {}
```

```
      SupportedOutputFormats: ["wav"    "flac"    "ogg"    "opus"    "mp4"    "m4a"]
         DefaultOutputFormat: "wav"
```

Partition Datastore into Default Number of Parts

Specify the file path to the audio samples included with Audio Toolbox™. Create an audio datastore that points to the specified folder.

```
folder = fullfile(matlabroot,'toolbox','audio','samples');
ADS = audioDatastore(folder);
```

Get the default number of partitions for `ADS`.

```
n = numpartitions(ADS);
```

Partition the datastore into the default number of partitions and return the datastore corresponding to the first partition.

```
subADS = partition(ADS,n,1);
```

Read the data in `subADS`.

```
while hasdata(subADS)
    data = read(subADS);
end
```

Partition Datastore by Files

Specify the file path to the audio samples included with Audio Toolbox™. Create an audio datastore that points to the specified folder.

```
folder = fullfile(matlabroot,'toolbox','audio','samples');
ADS = audioDatastore(folder);
```

Partition the datastore by files and return the part corresponding to the second file. `subds` contains one file.

```
subds = partition(ADS,'Files',2)
```

```
subds =
  audioDatastore with properties:

                      Files: {
                             'B:\matlab\toolbox\audio\samples\AudioArray-16-16-4channels-20secs
                             }
                    Folders: {
                             'B:\matlab\toolbox\audio\samples'
                             }
    AlternateFileSystemRoots: {}
              OutputDataType: 'double'
                      Labels: {}
      SupportedOutputFormats: ["wav"    "flac"    "ogg"    "opus"    "mp4"    "m4a"]
```

```
    DefaultOutputFormat: "wav"
```

Number of Partitions for Parallel Datastore Access

Partition a datastore to facilitate parallel access over the available parallel pool of workers.

Specify the file path to the audio samples included with Audio Toolbox™. Create an audio datastore that points to the specified folder.

```
folder = fullfile(matlabroot,'toolbox','audio','samples');
ADS = audioDatastore(folder);
```

Return an estimate for a reasonable number of partitions for parallel processing, given the current parallel pool.

```
pool = gcp;
n = numpartitions(ADS,pool);
```

Partition the audio datastore and read the data in each part.

```
parfor ii = 1:n
    subds = partition(ADS,n,ii);
    while hasdata(subds)
        data = read(subds);
    end
end
```

Input Arguments

`ADS` — Audio datastore

`audioDatastore` object

Audio datastore, specified as an `audioDatastore` object.

`numPartitions` — Number of partitions

positive integer

Number of partitions, specified as a positive integer. Use `numpartitions` to estimate a reasonable value for `numPartitions`.

Data Types: `double`

`index` — Index of sub-datastore

positive integer

Index of sub-datastore, specified as a positive integer in the range `[1,numPartitions]`.

Data Types: `double`

`filename` — File name

character vector

File name, specified as a character vector.

The value of `filename` must match exactly the file name contained in the `Files` property of the datastore.

Data Types: `char`

Output Arguments

`subADS` — Output audio datastore
`audioDatastore` object

Output audio datastore, returned as an `audioDatastore` object.

Version History

Introduced in R2018b

See Also

`audioDatastore` | `numpartitions`

Topics
“Train Speech Command Recognition Model Using Deep Learning”
“Speaker Identification Using Pitch and MFCC”
“Denoise Speech Using Deep Learning Networks”

countEachLabel

Count number of unique labels

Syntax

```
tbl = countEachLabel(ADS)
tbl = countEachLabel(ADS,'TableVariable',VariableName)
```

Description

`tbl = countEachLabel(ADS)` counts the number of times each unique label occurs in the datastore. In other words, it counts the number of files with each unique label. The output `tbl` is a table with variable names `Label` and `Count`.

`tbl = countEachLabel(ADS,'TableVariable',VariableName)` counts the number of times each unique label occurs in the datastore. When the datastore `Labels` property is specified by a table, you must specify `VariableName`. `VariableName` is the table variable (column) name you want to count.

Examples

Label Count

Specify the file path to the audio samples included with Audio Toolbox™.

```
folder = fullfile(matlabroot,'toolbox','audio','samples');
```

Create an audio datastore that points to the specified folder. Specify the `LabelSource` property as `foldernames`, so that the label associated with each file is set to the folder name that contains the file.

```
ads = audioDatastore(folder,'Labelsource','foldernames')
```

```
ads =
  audioDatastore with properties:

                      Files: {
                             'B:\matlab\toolbox\audio\samples\Ambiance-16-44p1-mono-12secs.wav'
                             'B:\matlab\toolbox\audio\samples\AudioArray-16-16-4channels-20secs
                             ' ...\toolbox\audio\samples\ChurchImpulseResponse-16-44p1-mono-5se
                              ... and 33 more
                             }
                    Folders: {
                             'B:\matlab\toolbox\audio\samples'
                             }
                     Labels: [samples; samples; samples ... and 33 more categorical]
    AlternateFileSystemRoots: {}
              OutputDataType: 'double'
      SupportedOutputFormats: ["wav"    "flac"    "ogg"    "opus"    "mp4"    "m4a"]
         DefaultOutputFormat: "wav"
```

Call `countEachLabel` to count the number of times each unique label occurs.

```
tbl = countEachLabel(ads)
```

```
tbl=1×2 table
     Label     Count
    _______    _____

    samples     36
```

Label Count when Labels Is Specified by Table

If the `Labels` property of an audio datastore is specified as a table, you must specify the table variable name when counting labels.

Specify the file path to the audio samples included with Audio Toolbox™.

```
folder = fullfile(matlabroot,'toolbox','audio','samples');
```

Create an audio datastore that points to the specified folder.

```
ADS = audioDatastore(folder)
```

```
ADS =
  audioDatastore with properties:

                      Files: {
                             'B:\matlab\toolbox\audio\samples\Ambiance-16-44p1-mono-12secs.wav'
                             'B:\matlab\toolbox\audio\samples\AudioArray-16-16-4channels-20secs
                             ' ...\toolbox\audio\samples\ChurchImpulseResponse-16-44p1-mono-5se
                              ... and 33 more
                             }
                    Folders: {
                             'B:\matlab\toolbox\audio\samples'
                             }
    AlternateFileSystemRoots: {}
              OutputDataType: 'double'
                      Labels: {}
      SupportedOutputFormats: ["wav"    "flac"    "ogg"    "opus"    "mp4"    "m4a"]
         DefaultOutputFormat: "wav"
```

The file names contain information about the files. Parse the file names to collect information about whether a file is mono or stereo and whether a file is longer than thirty seconds. Create a table containing the parsed information and then set the `Labels` property of the audio datastore to the label table.

```
numFiles = numel(ADS.Files);

numChannels = cell(numFiles,1);
isLong = cell(numFiles,1);

for i = 1:numFiles
    if ~isempty(strfind(ADS.Files{i},'mono'))
```

```
        numChannels{i} = 'mono';
    elseif ~isempty(strfind(ADS.Files{i},'stereo'))
        numChannels{i} = 'stereo';
    else
        numChannels{i} = 'unknown';
    end

    secs = str2double(regexp(ADS.Files{i}, '-(\d+)secs', 'tokens', 'once'));
    if secs > 30
        isLong{i} = true;
    elseif secs <= 30
        isLong{i} = false;
    else
        isLong{i} = 'unknown';
    end
end
labelTable = table(numChannels,isLong, ...
    'VariableNames',{'NumberOfChannels','IsLongerThan30Seconds'});

ADS.Labels = labelTable;
```

Call `countEachLabel` on the audio datastore and specify the `TableVariable` as `NumberOfChannels`. Call `countEachLabel` and specify the `TableVariable` as `IsLongerThan30Seconds`.

```
countNumberOfChannelLabels = countEachLabel(ADS,'TableVariable','NumberOfChannels')
```

```
countNumberOfChannelLabels=3×2 table
    NumberOfChannels    Count
    ________________    _____

        mono             24
        stereo           10
        unknown           2
```

```
countDurationLabels = countEachLabel(ADS,'TableVariable','IsLongerThan30Seconds')
```

```
countDurationLabels=3×2 table
    IsLongerThan30Seconds    Count
    _____________________    _____

           false              25
           true                6
           unknown             5
```

Input Arguments

ADS — Audio datastore

`audioDatastore` object

Specify `ADS` as an `audioDatastore` object.

VariableName — Label table variable name

character vector | string

Label table variable name, specified as a character vector or string that corresponds to a table variable of the `Label` property.

This syntax is required if the `Label` property of `audioDatastore` is specified by a table.

Data Types: `char` | `string`

Output Arguments

`tbl` — Table of label counts

two-column table

Table of label counts, returned as a two-column table containing the name of each label in `ADS` and the number of files associated with each label.

Data Types: `table`

Version History

Introduced in R2018b

See Also

`audioDatastore` | `splitEachLabel`

Topics

“Train Speech Command Recognition Model Using Deep Learning”
“Speaker Identification Using Pitch and MFCC”
“Denoise Speech Using Deep Learning Networks”

splitEachLabel

Splits datastore according to specified label proportions

Syntax

```
[ADS1,ADS2] = splitEachLabel(ADS,p)
[ADS1,...,ADSM] = splitEachLabel(ADS,p1,...,pN)
___ = splitEachLabel( ___ ,'randomized')
___ = splitEachLabel( ___ ,Name,Value)
```

Description

`[ADS1,ADS2] = splitEachLabel(ADS,p)` splits the audio files in `ADS` into two new datastores, `ADS1` and `ADS2`. The new datastore `ADS1` contains the first `p` files from each label ,and `ADS2` contains the remaining files from each label. `p` can be either a number between 0 and 1, exclusive, indicating the percentage of the files from each label to assign to `ADS1`, or an integer indicating the absolute number of files from each label to assign to `ADS1`.

`[ADS1,...,ADSM] = splitEachLabel(ADS,p1,...,pN)` splits the datastore into `N+1` new datastores. The new datastore `ADS1` contains the first `p1` files from each label, the next new datastore `ADS2` contains the next `p2` files, and so on. If `p1,…,pN` represent numbers of files, then their sum must be no more than the number of files in the smallest label in the original datastore, `ADS`.

`___ = splitEachLabel( ___ ,'randomized')` randomly assigns the specified proportion of files from each label to the new datastores.

`___ = splitEachLabel( ___ ,Name,Value)` specifies the properties of the new datastores using one or more name-value pair arguments. For example, you can specify which labels to split with `'Include','labelname'`.

Examples

Split by Fractions

Specify the file path to the audio samples included with Audio Toolbox™. Create an audio datastore that points to the specified folder.

```
folder = fullfile(matlabroot,'toolbox','audio','samples');
ADS = audioDatastore(folder,'FileExtensions','.wav');
```

Add the label `A` to the first half of the files, and the label `B` to the second half. If there are an odd number of files, assign the extra file the label `B`. Call `countEachLabel` to confirm that half of the files are labeled `A` and half the files are labeled `B`.

```
labels = [repmat({'A'},1,floor(numel(ADS.Files)/2)), ...
          repmat({'B'},1,ceil(numel(ADS.Files)/2))];
ADS.Labels = labels;

countEachLabel(ADS)
```

```
ans=2×2 table
    Label    Count
    _____    _____

      A       10
      B       10
```

Split ADS into two datastores, `ADS1` and `ADS2`, specifying that each new datastore contains fifty percent of each label and the corresponding files. Call `countEachLabel` to confirm that half of the files are labeled `A` and half of the files are labeled `B` for each of the new datastores.

```
[ADS1,ADS2] = splitEachLabel(ADS,0.5)
```

```
ADS1 =
  audioDatastore with properties:

                      Files: {
                             'B:\matlab\toolbox\audio\samples\Ambiance-16-44p1-mono-12secs.wav'
                             'B:\matlab\toolbox\audio\samples\AudioArray-16-16-4channels-20secs
                             ' ...\toolbox\audio\samples\ChurchImpulseResponse-16-44p1-mono-5se
                              ... and 7 more
                             }
                    Folders: {
                             'B:\matlab\toolbox\audio\samples'
                             }
                     Labels: {'A'; 'A'; 'A' ... and 7 more}
    AlternateFileSystemRoots: {}
              OutputDataType: 'double'
      SupportedOutputFormats: ["wav"    "flac"    "ogg"    "opus"    "mp4"    "m4a"]
         DefaultOutputFormat: "wav"


ADS2 =
  audioDatastore with properties:

                      Files: {
                             'B:\matlab\toolbox\audio\samples\Engine-16-44p1-stereo-20sec.wav';
                             'B:\matlab\toolbox\audio\samples\FemaleSpeech-16-8-mono-3secs.wav'
                             'B:\matlab\toolbox\audio\samples\Heli_16ch_ACN_SN3D.wav'
                              ... and 7 more
                             }
                    Folders: {
                             'B:\matlab\toolbox\audio\samples'
                             }
                     Labels: {'A'; 'A'; 'A' ... and 7 more}
    AlternateFileSystemRoots: {}
              OutputDataType: 'double'
      SupportedOutputFormats: ["wav"    "flac"    "ogg"    "opus"    "mp4"    "m4a"]
         DefaultOutputFormat: "wav"
```

```
ADS1count = countEachLabel(ADS1)
```

```
ADS1count=2×2 table
    Label    Count
    _____    _____
```

```
      A        5
      B        5
```

```
ADS2count = countEachLabel(ADS2)
```

```
ADS2count=2×2 table
    Label    Count
    _____    _____

      A        5
      B        5
```

Split by Number of Files

Specify the file path to the audio samples included with Audio Toolbox™. Create an audio datastore that points to the specified folder.

```
folder = fullfile(matlabroot,'toolbox','audio','samples');
ADS = audioDatastore(folder,'FileExtensions','.wav');
```

Add the label `A` to the first half of the files, and the label `B` to the second half. If there are an odd number of files, assign the extra file the label `B`. Call `countEachLabel` to confirm that half of the files are labeled `A` and half the files are labeled `B`.

```
labels = [repmat({'A'},1,floor(numel(ADS.Files)/2)), ...
          repmat({'B'},1,ceil(numel(ADS.Files)/2))];
ADS.Labels = labels;

countEachLabel(ADS)
```

```
ans=2×2 table
    Label    Count
    _____    _____

      A       10
      B       10
```

Split ADS into two datastores, `ADS1` and `ADS2`. Specify that `ADS1` contains four of each label and its corresponding file. `ADS2` contains the remaining labels and corresponding files. Call `countEachLabel` to confirm that `ADS1` contains four files labeled `A` and four files labeled `B`, and that `ADS2` contains the remaining labels.

```
[ADS1,ADS2] = splitEachLabel(ADS,4)
```

```
ADS1 =
  audioDatastore with properties:

                      Files: {
                             'B:\matlab\toolbox\audio\samples\Ambiance-16-44p1-mono-12secs.wav'
                             'B:\matlab\toolbox\audio\samples\AudioArray-16-16-4channels-20secs
                             ' ...\toolbox\audio\samples\ChurchImpulseResponse-16-44p1-mono-5se
                              ... and 5 more
                             }
```

```
                    Folders: {
                             'B:\matlab\toolbox\audio\samples'
                             }
                     Labels: {'A'; 'A'; 'A' ... and 5 more}
    AlternateFileSystemRoots: {}
              OutputDataType: 'double'
      SupportedOutputFormats: ["wav"    "flac"    "ogg"    "opus"    "mp4"    "m4a"]
         DefaultOutputFormat: "wav"


ADS2 =
  audioDatastore with properties:

                       Files: {
                             'B:\matlab\toolbox\audio\samples\Counting-16-44p1-mono-15secs.wav'
                             'B:\matlab\toolbox\audio\samples\Engine-16-44p1-stereo-20sec.wav';
                             'B:\matlab\toolbox\audio\samples\FemaleSpeech-16-8-mono-3secs.wav'
                              ... and 9 more
                             }
                    Folders: {
                             'B:\matlab\toolbox\audio\samples'
                             }
                     Labels: {'A'; 'A'; 'A' ... and 9 more}
    AlternateFileSystemRoots: {}
              OutputDataType: 'double'
      SupportedOutputFormats: ["wav"    "flac"    "ogg"    "opus"    "mp4"    "m4a"]
         DefaultOutputFormat: "wav"
```

```
ADS1count = countEachLabel(ADS1)
```

```
ADS1count=2×2 table
    Label    Count
    _____    _____

      A        4
      B        4
```

```
ADS2count = countEachLabel(ADS2)
```

```
ADS2count=2×2 table
    Label    Count
    _____    _____

      A        6
      B        6
```

Split Several Ways by Fractions

Specify the file path to the audio samples included with Audio Toolbox™. Create an audio datastore that points to the specified folder.

```
folder = fullfile(matlabroot,'toolbox','audio','samples');
ADS = audioDatastore(folder,'FileExtensions','.wav');
```

Add the label `A` to the first half of the files, and the label `B` to the second half. If there is an odd number of files, assign the extra file the label `B`. Call `countEachLabel` to confirm that half of the files are labeled `A` and half the files are labeled `B`.

```
labels = [repmat({'A'},1,floor(numel(ADS.Files)/2)), ...
          repmat({'B'},1,ceil(numel(ADS.Files)/2))];
ADS.Labels = labels;

countEachLabel(ADS)
```

```
ans=2×2 table
    Label    Count
    _____    _____

      A       10
      B       10
```

Split `ADS` into three new datastores, `ADS60`, `ADS10`, and `ADS30`. The first datastore, `ADS60`, contains the first 60% of files with the `A` label and the first 60% of files with the `B` label. `ADS10` contains the next 10% of files from each label. `ADS30` contains the remaining 30% of files from each label. If the percentage applied to a label does not result in a whole number of files, `splitEachLabel` rounds down to the nearest whole number.

```
[ADS60,ADS10,ADS30] = splitEachLabel(ADS,0.6,0.1)
```

```
ADS60 =
  audioDatastore with properties:

                      Files: {
                             'B:\matlab\toolbox\audio\samples\Ambiance-16-44p1-mono-12secs.wav'
                             'B:\matlab\toolbox\audio\samples\AudioArray-16-16-4channels-20secs
                             ' ...\toolbox\audio\samples\ChurchImpulseResponse-16-44p1-mono-5se
                              ... and 9 more
                             }
                    Folders: {
                             'B:\matlab\toolbox\audio\samples'
                             }
                     Labels: {'A'; 'A'; 'A' ... and 9 more}
    AlternateFileSystemRoots: {}
              OutputDataType: 'double'
      SupportedOutputFormats: ["wav"    "flac"    "ogg"    "opus"    "mp4"    "m4a"]
         DefaultOutputFormat: "wav"


ADS10 =
  audioDatastore with properties:

                      Files: {
                             'B:\matlab\toolbox\audio\samples\FemaleSpeech-16-8-mono-3secs.wav'
                             'B:\matlab\toolbox\audio\samples\TrainWhistle-16-44p1-mono-9secs.wa
                             }
                    Folders: {
                             'B:\matlab\toolbox\audio\samples'
                             }
                     Labels: {'A'; 'B'}
    AlternateFileSystemRoots: {}
              OutputDataType: 'double'
```

```
      SupportedOutputFormats: ["wav"    "flac"    "ogg"    "opus"    "mp4"    "m4a"]
         DefaultOutputFormat: "wav"


ADS30 =
  audioDatastore with properties:

                       Files: {
                              'B:\matlab\toolbox\audio\samples\Heli_16ch_ACN_SN3D.wav';
                              'B:\matlab\toolbox\audio\samples\JetAirplane-16-11p025-mono-16secs
                              'B:\matlab\toolbox\audio\samples\Laughter-16-8-mono-4secs.wav'
                               ... and 3 more
                              }
                     Folders: {
                              'B:\matlab\toolbox\audio\samples'
                              }
                      Labels: {'A'; 'A'; 'A' ... and 3 more}
    AlternateFileSystemRoots: {}
              OutputDataType: 'double'
      SupportedOutputFormats: ["wav"    "flac"    "ogg"    "opus"    "mp4"    "m4a"]
         DefaultOutputFormat: "wav"
```

Call `countEachLabel` to confirm the correct distribution of labels for each datastore.

```
countEachLabel(ADS60)
```

```
ans=2×2 table
    Label    Count
    _____    _____

      A        6
      B        6
```

```
countEachLabel(ADS10)
```

```
ans=2×2 table
    Label    Count
    _____    _____

      A        1
      B        1
```

```
countEachLabel(ADS30)
```

```
ans=2×2 table
    Label    Count
    _____    _____

      A        3
      B        3
```

Split Labels Several Ways by Number of Files

Specify the file path to the audio samples included with Audio Toolbox™. Create an audio datastore that points to the specified folder.

```
folder = fullfile(matlabroot,'toolbox','audio','samples');
ADS = audioDatastore(folder,'FileExtensions','.wav');
```

Add the label `A` to the first half of the files, and the label `B` to the second half. If there is an odd number of files, assign the extra file the label `B`. Call `countEachLabel` to confirm that half of the files are labeled `A` and half the files are labeled `B`.

```
labels = [repmat({'A'},1,floor(numel(ADS.Files)/2)), ...
          repmat({'B'},1,ceil(numel(ADS.Files)/2))];
ADS.Labels = labels;

countEachLabel(ADS)
```

```
ans=2×2 table
    Label    Count
    _____    _____

      A       10
      B       10
```

Split `ADS` into three new datastores, `ADS1`, `ADS2`, and `ADS3`. The first datastore, `ADS1`, contains the first file with the `A` label and the first file with the `B` label. `ADS2` contains the next file from each label. `ADS3` contains the remaining files from each label. If the percentage applied to a label does not result in a whole number of files, `splitEachLabel` rounds down to the nearest whole number.

```
[ADS1,ADS2,ADS3] = splitEachLabel(ADS,1,1)
```

```
ADS1 =
  audioDatastore with properties:

                      Files: {
                             'B:\matlab\toolbox\audio\samples\Ambiance-16-44p1-mono-12secs.wav'
                             'B:\matlab\toolbox\audio\samples\MainStreetOne-16-16-mono-12secs.wa
                             }
                    Folders: {
                             'B:\matlab\toolbox\audio\samples'
                             }
                     Labels: {'A'; 'B'}
    AlternateFileSystemRoots: {}
              OutputDataType: 'double'
      SupportedOutputFormats: ["wav"    "flac"    "ogg"    "opus"    "mp4"    "m4a"]
         DefaultOutputFormat: "wav"


ADS2 =
  audioDatastore with properties:

                      Files: {
                             'B:\matlab\toolbox\audio\samples\AudioArray-16-16-4channels-20secs
                             'B:\matlab\toolbox\audio\samples\NoisySpeech-16-22p5-mono-5secs.wav
                             }
                    Folders: {
```

```
                              'B:\matlab\toolbox\audio\samples'
                              }
                      Labels: {'A'; 'B'}
    AlternateFileSystemRoots: {}
              OutputDataType: 'double'
      SupportedOutputFormats: ["wav"    "flac"    "ogg"    "opus"    "mp4"    "m4a"]
         DefaultOutputFormat: "wav"


ADS3 =
  audioDatastore with properties:

                       Files: {
                              ' ...\toolbox\audio\samples\ChurchImpulseResponse-16-44p1-mono-5sec
                              'B:\matlab\toolbox\audio\samples\Click-16-44p1-mono-0.2secs.wav';
                              'B:\matlab\toolbox\audio\samples\Counting-16-44p1-mono-15secs.wav'
                               ... and 13 more
                              }
                     Folders: {
                              'B:\matlab\toolbox\audio\samples'
                              }
                      Labels: {'A'; 'A'; 'A' ... and 13 more}
    AlternateFileSystemRoots: {}
              OutputDataType: 'double'
      SupportedOutputFormats: ["wav"    "flac"    "ogg"    "opus"    "mp4"    "m4a"]
         DefaultOutputFormat: "wav"
```

Call `countEachLabel` to confirm the correct distribution of labels for each datastore.

```
countEachLabel(ADS1)
```

```
ans=2×2 table
    Label    Count
    _____    _____

      A        1
      B        1
```

```
countEachLabel(ADS2)
```

```
ans=2×2 table
    Label    Count
    _____    _____

      A        1
      B        1
```

```
countEachLabel(ADS3)
```

```
ans=2×2 table
    Label    Count
    _____    _____

      A        8
      B        8
```

Split Labels in Random Order

Specify the file path to the audio samples included with Audio Toolbox™. Create an audio datastore that points to the specified folder.

```
folder = fullfile(matlabroot,'toolbox','audio','samples');
ADS = audioDatastore(folder,'FileExtensions','.wav')
```

```
ADS =
  audioDatastore with properties:

                      Files: {
                             'B:\matlab\toolbox\audio\samples\Ambiance-16-44p1-mono-12secs.wav'
                             'B:\matlab\toolbox\audio\samples\AudioArray-16-16-4channels-20secs
                             ' ...\toolbox\audio\samples\ChurchImpulseResponse-16-44p1-mono-5se
                              ... and 17 more
                             }
                    Folders: {
                             'B:\matlab\toolbox\audio\samples'
                             }
    AlternateFileSystemRoots: {}
              OutputDataType: 'double'
                      Labels: {}
      SupportedOutputFormats: ["wav"    "flac"    "ogg"    "opus"    "mp4"    "m4a"]
         DefaultOutputFormat: "wav"
```

Add the label `A` to the first half of the files, and the label `B` to the second half. If there is an odd number of files, assign the extra file the label `B`. Call `countEachLabel` to confirm that half of the files are labeled `A` and half the files are labeled `B`.

```
labels = [repmat({'A'},1,floor(numel(ADS.Files)/2)), ...
          repmat({'B'},1,ceil(numel(ADS.Files)/2))];
ADS.Labels = labels;

countEachLabel(ADS)
```

```
ans=2×2 table
    Label    Count
    _____    _____

      A       10
      B       10
```

Create two new datastores from the files in `ADS` by randomly drawing from each label. The first datastore, `ADS1`, contains two random files with the `A` label and two random files with the `B` label. `ADS2` contains the remaining files from each label.

```
[ADS1,ADS2] = splitEachLabel(ADS,2,'randomized')
```

```
ADS1 =
  audioDatastore with properties:

                      Files: {
```

```
                                ' ...\toolbox\audio\samples\ChurchImpulseResponse-16-44p1-mono-5sec
                                'B:\matlab\toolbox\audio\samples\Engine-16-44p1-stereo-20sec.wav';
                                'B:\matlab\toolbox\audio\samples\MainStreetOne-16-16-mono-12secs.wa
                                 ... and 1 more
                                }
                       Folders: {
                                'B:\matlab\toolbox\audio\samples'
                                }
                        Labels: {'A'; 'A'; 'B' ... and 1 more}
    AlternateFileSystemRoots: {}
              OutputDataType: 'double'
      SupportedOutputFormats: ["wav"    "flac"    "ogg"    "opus"    "mp4"    "m4a"]
         DefaultOutputFormat: "wav"


ADS2 =
  audioDatastore with properties:

                        Files: {
                                'B:\matlab\toolbox\audio\samples\Ambiance-16-44p1-mono-12secs.wav'
                                'B:\matlab\toolbox\audio\samples\AudioArray-16-16-4channels-20secs
                                'B:\matlab\toolbox\audio\samples\Click-16-44p1-mono-0.2secs.wav'
                                 ... and 13 more
                                }
                       Folders: {
                                'B:\matlab\toolbox\audio\samples'
                                }
                        Labels: {'A'; 'A'; 'A' ... and 13 more}
    AlternateFileSystemRoots: {}
              OutputDataType: 'double'
      SupportedOutputFormats: ["wav"    "flac"    "ogg"    "opus"    "mp4"    "m4a"]
         DefaultOutputFormat: "wav"
```

Include and Exclude Specified Labels

Specify the file path to the audio samples included with Audio Toolbox™. Create an audio datastore that points to the specified folder.

```
folder = fullfile(matlabroot,'toolbox','audio','samples');
ADS = audioDatastore(folder,'FileExtensions','.wav')
```

```
ADS =

  audioDatastore with properties:

                        Files: {
                                'B:\matlab\toolbox\audio\samples\Ambiance-16-44p1-mono-12secs.wav'
                                'B:\matlab\toolbox\audio\samples\AudioArray-16-16-4channels-20secs
                                ' ...\toolbox\audio\samples\ChurchImpulseResponse-16-44p1-mono-5sec
                                 ... and 17 more
                                }
                       Folders: {
                                'B:\matlab\toolbox\audio\samples'
                                }
```

```
    AlternateFileSystemRoots: {}
              OutputDataType: 'double'
                      Labels: {}
      SupportedOutputFormats: ["wav"    "flac"    "ogg"    "opus"    ...    ]
         DefaultOutputFormat: "wav"
```

Add the label `A` to the first half of the files, and the label `B` to the second half. If there is an odd number of files, assign the extra file the label `B`. Call `countEachLabel` to confirm that half of the files are labeled `A` and half the files are labeled `B`.

```
labels = [repmat({'A'},1,floor(numel(ADS.Files)/2)), ...
          repmat({'B'},1,ceil(numel(ADS.Files)/2))];
ADS.Labels = labels;

countEachLabel(ADS)
```

```
ans =

  2x2 table

    Label    Count
    _____    _____

      A       10
      B       10
```

Create two new datastores from the files in `ADS`, including only the files with the `A` label. `ADS1` contains the first 70% of files with the `A` label, and `ADS2` contains the remaining 30% of labels with the `A` label.

```
[ADS1,ADS2] = splitEachLabel(ADS,0.7,'Include','A')
```

```
ADS1 =

  audioDatastore with properties:

                       Files: {
                              'B:\matlab\toolbox\audio\samples\Ambiance-16-44p1-mono-12secs.wav'
                              'B:\matlab\toolbox\audio\samples\AudioArray-16-16-4channels-20secs
                              ' ...\toolbox\audio\samples\ChurchImpulseResponse-16-44p1-mono-5se
                               ... and 4 more
                              }
                     Folders: {
                              'B:\matlab\toolbox\audio\samples'
                              }
                      Labels: {'A'; 'A'; 'A' ... and 4 more}
    AlternateFileSystemRoots: {}
              OutputDataType: 'double'
      SupportedOutputFormats: ["wav"    "flac"    "ogg"    "opus"    ...    ]
         DefaultOutputFormat: "wav"


ADS2 =
```

```
  audioDatastore with properties:

                      Files: {
                             'B:\matlab\toolbox\audio\samples\Heli_16ch_ACN_SN3D.wav';
                             'B:\matlab\toolbox\audio\samples\JetAirplane-16-11p025-mono-16secs
                             'B:\matlab\toolbox\audio\samples\Laughter-16-8-mono-4secs.wav'
                             }
                    Folders: {
                             'B:\matlab\toolbox\audio\samples'
                             }
                     Labels: {'A'; 'A'; 'A'}
    AlternateFileSystemRoots: {}
              OutputDataType: 'double'
      SupportedOutputFormats: ["wav"    "flac"    "ogg"    "opus"    ...    ]
         DefaultOutputFormat: "wav"
```

Equivalently, you can split only the A label by excluding the B label.

```
[ADS1,ADS2] = splitEachLabel(ADS,0.7,'Exclude','B')
```

```
ADS1 =

  audioDatastore with properties:

                      Files: {
                             'B:\matlab\toolbox\audio\samples\Ambiance-16-44p1-mono-12secs.wav'
                             'B:\matlab\toolbox\audio\samples\AudioArray-16-16-4channels-20secs
                             ' ...\toolbox\audio\samples\ChurchImpulseResponse-16-44p1-mono-5se
                              ... and 4 more
                             }
                    Folders: {
                             'B:\matlab\toolbox\audio\samples'
                             }
                     Labels: {'A'; 'A'; 'A' ... and 4 more}
    AlternateFileSystemRoots: {}
              OutputDataType: 'double'
      SupportedOutputFormats: ["wav"    "flac"    "ogg"    "opus"    ...    ]
         DefaultOutputFormat: "wav"


ADS2 =

  audioDatastore with properties:

                      Files: {
                             'B:\matlab\toolbox\audio\samples\Heli_16ch_ACN_SN3D.wav';
                             'B:\matlab\toolbox\audio\samples\JetAirplane-16-11p025-mono-16secs
                             'B:\matlab\toolbox\audio\samples\Laughter-16-8-mono-4secs.wav'
                             }
                    Folders: {
                             'B:\matlab\toolbox\audio\samples'
                             }
                     Labels: {'A'; 'A'; 'A'}
    AlternateFileSystemRoots: {}
              OutputDataType: 'double'
      SupportedOutputFormats: ["wav"    "flac"    "ogg"    "opus"    ...    ]
```

```
          DefaultOutputFormat: "wav"
```

Split Using Fraction and Label Table

Specify the file path to the audio samples included with Audio Toolbox™. Create an audio datastore that points to the specified folder.

```
folder = fullfile(matlabroot,'toolbox','audio','samples');
ADS = audioDatastore(folder)
```

```
ADS =
  audioDatastore with properties:

                      Files: {
                             'B:\matlab\toolbox\audio\samples\Ambiance-16-44p1-mono-12secs.wav'
                             'B:\matlab\toolbox\audio\samples\AudioArray-16-16-4channels-20secs
                             ' ...\toolbox\audio\samples\ChurchImpulseResponse-16-44p1-mono-5se
                              ... and 33 more
                             }
                    Folders: {
                             'B:\matlab\toolbox\audio\samples'
                             }
    AlternateFileSystemRoots: {}
              OutputDataType: 'double'
                      Labels: {}
      SupportedOutputFormats: ["wav"    "flac"    "ogg"    "opus"    "mp4"    "m4a"]
         DefaultOutputFormat: "wav"
```

Create a label table with two variables:

- `containsMusic` -- Can be either `true` or `false`.
- `instrument` -- Can be `Guitar`, `Drums`, or `Unknown`.

```
containsGuitar = contains(ADS.Files,'guitar','IgnoreCase',true);
containsDrums = contains(ADS.Files,'drum','IgnoreCase',true);
containsMusic = or(containsGuitar,containsDrums);

instrument = strings(size(ADS.Files));
instrument(:) = "Unknown";
instrument(containsGuitar) = "Guitar";
instrument(containsDrums) = "Drums";
```

Assign the label table to the `Labels` property of audio datastore to associate the rows of the label table with the rows of the datastore. Call `countEachLabel` to determine the incidences of `containsMusic` and `instrument`.

```
labels = table(containsMusic,instrument);
ADS.Labels = labels;

containsMusicCount = countEachLabel(ADS,'TableVariable','containsMusic')
```

```
containsMusicCount=2×2 table
    containsMusic    Count
    _____________    _____
```

```
        false         29
        true           7
```

```
instrumentCount = countEachLabel(ADS,'TableVariable','instrument')
```

```
instrumentCount=3×2 table
    instrument    Count
    __________    _____

     Drums          4
     Guitar         3
     Unknown       29
```

Split the datastore `ADS` into two, based on whether the audio file contains music. `ADS1` contains 70% of the audio files that contain music, and `ADS2` contains the rest. Call `countEachLabel` to verify that the ratio of `containsMusic == true` to `containsMusic == false` is preserved for the new datastores, within rounding.

```
[ADS1,ADS2] = splitEachLabel(ADS,0.7,'TableVariable','containsMusic');
ADS1_containsMusicCount = countEachLabel(ADS1,'TableVariable','containsMusic')
```

```
ADS1_containsMusicCount=2×2 table
    containsMusic    Count
    _____________    _____

        false         20
        true           5
```

```
ADS2_containsMusicCount = countEachLabel(ADS2,'TableVariable','containsMusic')
```

```
ADS2_containsMusicCount=2×2 table
    containsMusic    Count
    _____________    _____

        false          9
        true           2
```

Split the datastore `ADS` into two, based on the type of instrument present in the audio file. `ADS3` contains 25% of the audio files that have an instrument label, and `ADS4` contains the rest. Call `countEachLabel` to verify that the ratio of `instrument == "drums"` to `instrument == "guitar"` is preserved for the new datastores, within rounding.

```
[ADS3,ADS4] = splitEachLabel(ADS,0.25,'TableVariable','instrument');
ADS3_instrumentCount = countEachLabel(ADS3,'TableVariable','instrument')
```

```
ADS3_instrumentCount=3×2 table
    instrument    Count
    __________    _____

     Drums          1
     Guitar         1
     Unknown        7
```

```
ADS4_instrumentCount = countEachLabel(ADS4,'TableVariable','instrument')
```

```
ADS4_instrumentCount=3×2 table
    instrument    Count
    __________    _____

     Drums          3
     Guitar         2
     Unknown       22
```

Split by Number of Files and Label Table

Specify the file path to the audio samples included with Audio Toolbox™. Create an audio datastore that points to the specified folder.

```
folder = fullfile(matlabroot,'toolbox','audio','samples');
ADS = audioDatastore(folder);
```

Create a label table with two variables:

- `containsMusic` - Can be either `true` or `false`.
- `instrument` - Can be `Guitar`, `Drums`, or `Unknown`.

```
containsGuitar = contains(ADS.Files,'guitar','IgnoreCase',true);
containsDrums = contains(ADS.Files,'drum','IgnoreCase',true);
containsMusic = or(containsGuitar,containsDrums);

instrument = strings(size(ADS.Files));
instrument(:) = "Unknown";
instrument(containsGuitar) = "Guitar";
instrument(containsDrums) = "Drums";
```

Assign the label table to the `Labels` property of audio datastore to associate the rows of the label table with the rows of the datastore. Call `countEachLabel` to determine the incidences of `containsMusic` and `instrument`.

```
labels = table(containsMusic,instrument);
ADS.Labels = labels;

containsMusicCount = countEachLabel(ADS,'TableVariable','containsMusic')
```

```
containsMusicCount=2×2 table
    containsMusic    Count
    _____________    _____

        false         29
        true           7
```

```
instrumentCount = countEachLabel(ADS,'TableVariable','instrument');
```

Split the datastore `ADS` into two, based on whether the audio file contains music. `ADS1` contains 5 of each label under the table variable `containsMusic`, and `ADS2` contains the rest. Call `countEachLabel` to verify.

```
[ADS1,ADS2] = splitEachLabel(ADS,5,'TableVariable','containsMusic');
ADS1_containsMusicCount = countEachLabel(ADS1,'TableVariable','containsMusic')
```

```
ADS1_containsMusicCount=2×2 table
    containsMusic    Count
    _____________    _____

        false          5
        true           5
```

```
ADS2_containsMusicCount = countEachLabel(ADS2,'TableVariable','containsMusic')
```

```
ADS2_containsMusicCount=2×2 table
    containsMusic    Count
    _____________    _____

        false         24
        true           2
```

Split the datastore `ADS` into two, based on the type of instrument present in the audio file. `ADS3` contains 2 of each label under the table variable `instrument`, and `ADS4` contains the rest. Call `countEachLabel` to verify.

```
[ADS3,ADS4] = splitEachLabel(ADS,2,'TableVariable','instrument');
ADS3_instrumentCount = countEachLabel(ADS3,'TableVariable','instrument')
```

```
ADS3_instrumentCount=3×2 table
    instrument    Count
    __________    _____

     Drums          2
     Guitar         2
     Unknown        2
```

```
ADS4_instrumentCount = countEachLabel(ADS4,'TableVariable','instrument')
```

```
ADS4_instrumentCount=3×2 table
    instrument    Count
    __________    _____

     Drums          2
     Guitar         1
     Unknown       27
```

Input Arguments

ADS — Input audio datastore

`audioDatastore` object

Input audio datastore, specified as an `audioDatastore` object.

p — Proportion of files to split

scalar in interval (0,1) | positive integer scalar

Proportion of files to split, specified as a scalar in the interval (0,1), or a positive integer scalar.

If `p` is in the interval (0,1), it represents the percentage of the files from each label to assign to `ADS1`. If `p` represents a percentage, and it does not result in a whole number, then `splitEachLabel` rounds down to the nearest whole number.

If `p` is an integer, it represents the absolute number of files from each label to assign to `ADS1`. When `p` represents a number of files, there must be at least `p` files associated with each label.

Data Types: `double`

`p1,...,pN` — List of proportions
scalars in interval (0,1) | positive integer scalars

List of proportions, specified as scalars in the interval (0,1) or positive integer scalars.

If the proportions are in the interval (0,1), they represent the percentage of the files from each label to assign to the output datastores. When the proportions represent percentages, their sum must be no more than 1.

If the proportions are integers, they indicate the absolute number of files from each label to assign to the output datastores. When the proportions represent numbers of files, there must be enough files associated with each label to satisfy each proportion.

Data Types: `double`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `[ADS1,ADS2] = splitEachLabel(ADS,0.5,'Exclude','noisy')`

`Include` — Labels to include
categorical, logical, or numeric vector | cell array of character vectors | string array

Labels to include, specified as the comma-separated pair consisting of `'Include'` and a vector, cell array, or string array of label names with the same type as the `Labels` property. Each name must match one of the labels in the `Labels` property of the datastore.

This option cannot be used with the `'Exclude'` option.

`Exclude` — Labels to exclude
categorical, logical, or numeric vector | cell array of character vectors | string array

Labels to exclude, specified as the comma-separated pair consisting of `'Exclude'` and a vector, cell array, or string array of label names with the same type as the `Labels` property. Each name must match one of the labels in the `Labels` property of the datastore.

This option cannot be used with the `'Include'` option.

`TableVariable` — Label table variable name
char | string

Table variable name, specified as the comma-separated pair consisting of `'TableVariable'` and a character vector or string. When the `Labels` property of the audio datastore `ADS` is a table, you must use `'TableVariable'` to specify which label you are using to split.

Data Types: `char` | `string`

Output Arguments

`[ADS1,ADS2]` — Output audio datastores
`audioDatastore` objects

Output audio datastores, returned as `audioDatastore` objects. `ADS1` contains the specified proportion of files from each label in `ADS`, and `ADS2` contains the remaining files.

`[ADS1,...,ADSM]` — List of output audio datastores
`audioDatastore` objects

List of output audio datastores, returned as `audioDatastore` objects. The number of elements in the list is one more that the number of listed proportions. Each of the new datastores contains the proportion of each label in `ADS` defined by `p1,…,pN`. Any files left over are assigned to the *M*th datastore.

Version History

Introduced in R2018b

See Also

`audioDatastore` | `countEachLabel` | `subset`

Topics

“Train Speech Command Recognition Model Using Deep Learning”
“Speaker Identification Using Pitch and MFCC”
“Denoise Speech Using Deep Learning Networks”

preview

Read first file from datastore for preview

Syntax

```
data = preview(ADS)
```

Description

`data = preview(ADS)` always reads the first file from `ADS`. `preview` does not affect the state of `ADS`.

Examples

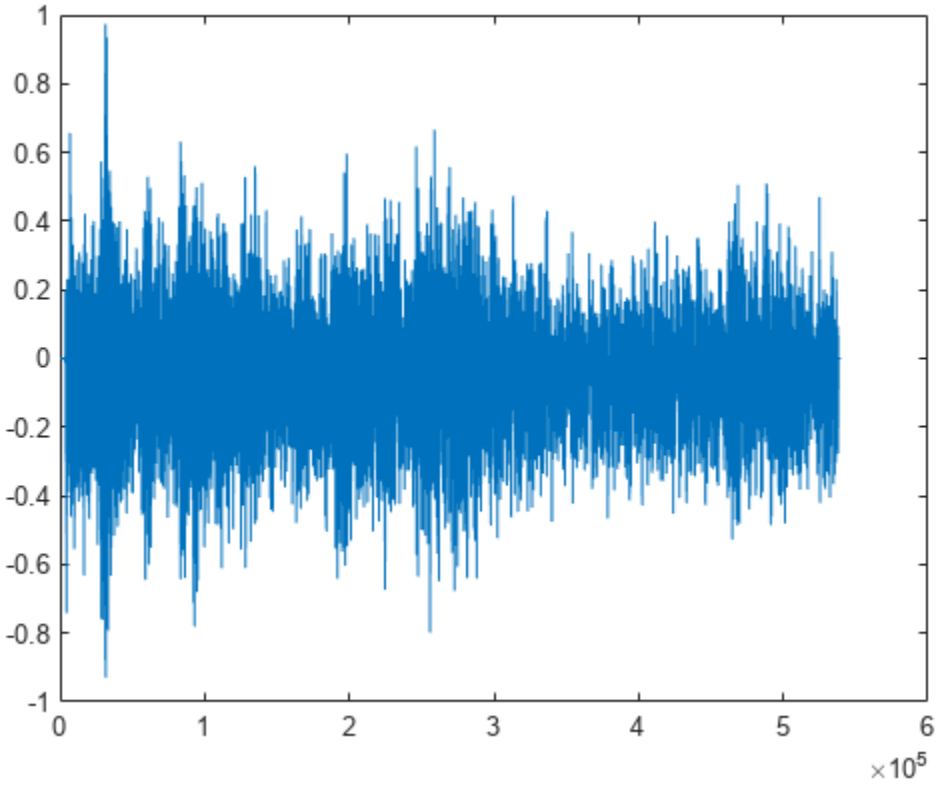
Preview Data in Audio Datastore

Specify the file path to the audio samples included with Audio Toolbox™. Create an audio datastore that points to the specified folder.

```
folder = fullfile(matlabroot,'toolbox','audio','samples');
ADS = audioDatastore(folder);
```

Preview the data in the audio datastore.

```
data = preview(ADS);
plot(data)
```



Input Arguments

ADS — Audio datastore
`audioDatastore` object

Specify `ADS` as an `audioDatastore` object.

Output Arguments

data — Subset of data
array of audio samples

Subset of data, returned as an array of audio samples.

Version History

Introduced in R2018b

See Also

`audioDatastore` | `hasdata`

Topics
“Train Speech Command Recognition Model Using Deep Learning”

“Speaker Identification Using Pitch and MFCC”
“Denoise Speech Using Deep Learning Networks”

subset

Create datastore with subset of files

Syntax

```
ADSsubset = subset(ADS,indices)
```

Description

`ADSsubset = subset(ADS,indices)` returns an audio datastore, `ADSsubset`, which contains a subset of the files in `ADS`.

Examples

Create Datastore with Subset Based on File Name

`subset` creates an audio datastore containing a subset of the files of the original datastore.

Specify the file path to the audio samples included with Audio Toolbox™. Create an audio datastore that points to the specified folder.

```
folder = fullfile(matlabroot,'toolbox','audio','samples');
ADS = audioDatastore(folder)
```

```
ADS =
  audioDatastore with properties:

                      Files: {
                             'B:\matlab\toolbox\audio\samples\Ambiance-16-44p1-mono-12secs.wav'
                             'B:\matlab\toolbox\audio\samples\AudioArray-16-16-4channels-20secs
                             ' ...\toolbox\audio\samples\ChurchImpulseResponse-16-44p1-mono-5se
                              ... and 33 more
                             }
                    Folders: {
                             'B:\matlab\toolbox\audio\samples'
                             }
    AlternateFileSystemRoots: {}
              OutputDataType: 'double'
                      Labels: {}
      SupportedOutputFormats: ["wav"    "flac"    "ogg"    "opus"    "mp4"    "m4a"]
         DefaultOutputFormat: "wav"
```

Create a logical vector indicating whether the file names in the audio datastore contain `'Guitar'`.

```
fileContainsGuitar = cellfun(@(c)contains(c,'Guitar'),ADS.Files)
```

```
fileContainsGuitar = 36x1 logical array

   0
   0
```

```
     0
     0
     0
     0
     0
     0
     0
     0
        ⋮
```

Call `subset` with the audio datastore and the indices corresponding to the files you want create a new audio datastore from.

```
ADSsubset = subset(ADS,fileContainsGuitar)
```

```
ADSsubset =
  audioDatastore with properties:

                      Files: {
                             'B:\matlab\toolbox\audio\samples\RockGuitar-16-44p1-stereo-72secs.w
                             'B:\matlab\toolbox\audio\samples\RockGuitar-16-96-stereo-72secs.fla
                             'B:\matlab\toolbox\audio\samples\SoftGuitar-44p1_mono-10mins.ogg'
                             }
                    Folders: {
                             'B:\matlab\toolbox\audio\samples'
                             }
    AlternateFileSystemRoots: {}
              OutputDataType: 'double'
                      Labels: {}
      SupportedOutputFormats: ["wav"    "flac"    "ogg"    "opus"    "mp4"    "m4a"]
         DefaultOutputFormat: "wav"
```

Create Datastore with Every Other File

Specify the file path to the audio samples included with Audio Toolbox™. Create an audio datastore that points to the specified folder.

```
folder = fullfile(matlabroot,'toolbox','audio','samples');
ADS = audioDatastore(folder)
```

```
ADS =
  audioDatastore with properties:

                      Files: {
                             'B:\matlab\toolbox\audio\samples\Ambiance-16-44p1-mono-12secs.wav'
                             'B:\matlab\toolbox\audio\samples\AudioArray-16-16-4channels-20secs
                             ' ...\toolbox\audio\samples\ChurchImpulseResponse-16-44p1-mono-5se
                              ... and 33 more
                             }
                    Folders: {
                             'B:\matlab\toolbox\audio\samples'
                             }
    AlternateFileSystemRoots: {}
              OutputDataType: 'double'
```

```
                    Labels: {}
      SupportedOutputFormats: ["wav"    "flac"    "ogg"    "opus"    "mp4"    "m4a"]
         DefaultOutputFormat: "wav"
```

Create an audio datastore containing every other file of the original datastore.

```
indices = 1:2:numel(ADS.Files);
ADSsubset = subset(ADS,indices)
```

```
ADSsubset =
  audioDatastore with properties:

                      Files: {
                             'B:\matlab\toolbox\audio\samples\Ambiance-16-44p1-mono-12secs.wav'
                             ' ...\toolbox\audio\samples\ChurchImpulseResponse-16-44p1-mono-5se
                             'B:\matlab\toolbox\audio\samples\Counting-16-44p1-mono-15secs.wav'
                              ... and 15 more
                             }
                    Folders: {
                             'B:\matlab\toolbox\audio\samples'
                             }
    AlternateFileSystemRoots: {}
              OutputDataType: 'double'
                      Labels: {}
      SupportedOutputFormats: ["wav"    "flac"    "ogg"    "opus"    "mp4"    "m4a"]
         DefaultOutputFormat: "wav"
```

Input Arguments

ADS — Audio datastore

`audioDatastore` object

Specify `ADS` as an `audioDatastore` object.

indices — Indices of files for subset

vector of indices | logical vector

Specify `indices` as:

- A vector containing the indices of files to be included in `ADSsubset`.
- A logical vector the same length as the number of files in `ADS`. If specifying indices as a logical vector, `true` indicates that the corresponding files are included in `ADSsubset`.

Data Types: `double` | `logical`

Output Arguments

ADSsubset — Subset of audio datastore

`audioDatastore` object

Subset of audio datastore, returned as an `audioDatastore` object.

Version History

Introduced in R2018b

See Also

`audioDatastore` | `splitEachLabel`

Topics

“Train Speech Command Recognition Model Using Deep Learning”
“Speaker Identification Using Pitch and MFCC”
“Denoise Speech Using Deep Learning Networks”

shuffle

Shuffle files in datastore

Syntax

```
shuffledADS = shuffle(ADS)
```

Description

`shuffledADS = shuffle(ADS)` creates a deep copy of the input datastore, `ADS`, and shuffles the files using `randperm`.

Examples

Shuffle Files

Create an `audioDatastore` object `ADS`. Shuffle the files to create a new datastore containing the same files in random order.

```
ADS = audioDatastore(fullfile(matlabroot,'toolbox','audio','samples'))
```

```
ADS =
  audioDatastore with properties:

                      Files: {
                             'B:\matlab\toolbox\audio\samples\Ambiance-16-44p1-mono-12secs.wav'
                             'B:\matlab\toolbox\audio\samples\AudioArray-16-16-4channels-20secs
                             ' ...\toolbox\audio\samples\ChurchImpulseResponse-16-44p1-mono-5se
                              ... and 33 more
                             }
                    Folders: {
                             'B:\matlab\toolbox\audio\samples'
                             }
    AlternateFileSystemRoots: {}
              OutputDataType: 'double'
                      Labels: {}
      SupportedOutputFormats: ["wav"    "flac"    "ogg"    "opus"    "mp4"    "m4a"]
         DefaultOutputFormat: "wav"
```

```
ADSshuffled = shuffle(ADS)
```

```
ADSshuffled =
  audioDatastore with properties:

                      Files: {
                             ' ...\matlab\toolbox\audio\samples\WashingMachine-16-44p1-stereo-10
                             'B:\matlab\toolbox\audio\samples\RockDrums-44p1-stereo-11secs.mp3'
                             'B:\matlab\toolbox\audio\samples\WashingMachine-16-8-mono-200secs.m
                              ... and 33 more
                             }
```

```
                  Folders: {
                           'B:\matlab\toolbox\audio\samples'
                           }
    AlternateFileSystemRoots: {}
              OutputDataType: 'double'
                      Labels: {}
      SupportedOutputFormats: ["wav"    "flac"    "ogg"    "opus"    "mp4"    "m4a"]
         DefaultOutputFormat: "wav"
```

Input Arguments

`ADS` — Input audio datastore

`audioDatastore` object

Input audio datastore, specified as an `audioDatastore` object.

Output Arguments

`shuffledADS` — Shuffled audio datastore

`audioDatastore` object

Shuffled audio datastore, returned as an `audioDatastore` object containing randomly ordered files from `ADS`.

Version History

Introduced in R2018b

See Also

`audioDatastore`

Topics

“Train Speech Command Recognition Model Using Deep Learning”
“Speaker Identification Using Pitch and MFCC”
“Denoise Speech Using Deep Learning Networks”

hasdata

Return true if there is more data in datastore

Syntax

```
tf = hasdata(ADS)
```

Description

`tf = hasdata(ADS)` returns logical `1` (`true`) if there is data available to read from the datastore specified by `ADS`. Otherwise, it returns logical `0` (`false`).

Examples

Keep Reading While Datastore Has Data

`hasdata` returns a logical scalar indicating whether or not there is unread data in the datastore. You can use `audioDatastore` to read files sequentially until all data is read.

Specify the file path to the audio samples included with Audio Toolbox™.

```
folder = fullfile(matlabroot,'toolbox','audio','samples');
```

Create an audio datastore that points to the specified folder.

```
ADS = audioDatastore(folder);
```

While the datastore has unread data, read from the datastore.

```
while hasdata(ADS)
    data = read(ADS);
end
```

Input Arguments

ADS — Audio datastore

`audioDatastore` object

Specify `ADS` as an `audioDatastore` object.

Output Arguments

tf — Indication if data is available to read

`true` | `false`

Indication is data is available to read from the datastore, returned as `true` or `false`.

Data Types: `logical`

Version History

Introduced in R2018b

See Also

`audioDatastore` | `read` | `progress`

Topics

“Train Speech Command Recognition Model Using Deep Learning”
“Speaker Identification Using Pitch and MFCC”
“Denoise Speech Using Deep Learning Networks”

reset

Reset datastore read pointer to start of data

Syntax

`reset(ADS)`

Description

`reset(ADS)` resets the datastore read pointer to the start of the data. Resetting allows re-reading from the same datastore.

Examples

Reset Audio Datastore to Initial State

Create an `audioDatastore` object `ADS`.

```
folder = fullfile(matlabroot,'toolbox','audio','samples');
ADS = audioDatastore(folder);
```

While the datastore has unread files, call `read` in a loop to read files sequentially.

```
while hasdata(ADS)
    data = read(ADS);
end
```

Reset the datastore to the state where no data has been read from it. Read the first file from the datastore.

```
reset(ADS)
data = read(ADS);
```

Input Arguments

ADS — Audio datastore

`audioDatastore` object

Specify `ADS` as an `audioDatastore` object.

Version History

Introduced in R2018b

See Also

`audioDatastore`

Topics
“Train Speech Command Recognition Model Using Deep Learning”
“Speaker Identification Using Pitch and MFCC”
“Denoise Speech Using Deep Learning Networks”

readall

Read all audio files from datastore

Syntax

```
data = readall(ADS)
data = readall(ADS,UseParallel=TF)
```

Description

`data = readall(ADS)` reads all audio files from the datastore.

If all the data in the datastore does not fit in memory, then `readall` returns an error.

`data = readall(ADS,UseParallel=TF)` reads the data in parallel if `TF` is `true` (requires Parallel Computing Toolbox).

Examples

Read All Data in Audio Datastore

Specify the file path to the audio samples included with Audio Toolbox™. Create an audio datastore that points to the specified folder.

```
folder = fullfile(matlabroot,'toolbox','audio','samples');
ADS = audioDatastore(folder);
```

Read all the data in the datastore.

```
readall(ADS)
```

```
ans=36×1 cell array
    { 539648x1  double}
    { 320512x4  double}
    { 227497x1  double}
    {   8000x1  double}
    { 685056x1  double}
    { 882688x2  double}
    {  24000x1  double}
    { 175104x1  double}
    {1115760x2  double}
    {1214832x2  double}
    { 263304x16 double}
    { 100868x1  double}
    { 180224x1  double}
    {  32768x1  double}
    { 192150x1  double}
    { 100352x1  double}
      ⋮
```

Input Arguments

ADS — Audio datastore

`audioDatastore` object

Specify `ADS` as an `audioDatastore` object.

TF — Read in parallel

`false` (default) | `true`

Read in parallel, specified as `true` or `false`. If you specify `true`, `readall` reads all data from the datastore using a pool of parallel workers (requires Parallel Computing Toolbox). For more information on parallel pools, see `parpool`. Parallel reading may result in improved performance when reading data.

Example: `readall(ds,UseParallel=true)`

Output Arguments

data — All audio files in audio datastore

cell array

All files in the audio datastore, returned as a cell array where each cell corresponds to a file.

Version History

Introduced in R2018b

See Also

`audioDatastore` | `read`

Topics

“Train Speech Command Recognition Model Using Deep Learning”
“Speaker Identification Using Pitch and MFCC”
“Denoise Speech Using Deep Learning Networks”

read

Read next consecutive audio file

Syntax

```
data = read(ADS)
[data,info] = read(ADS)
```

Description

`data = read(ADS)` returns audio extracted from the datastore. Each subsequent call to the `read` function continues reading from the endpoint of the previous call.

`[data,info] = read(ADS)` also returns information about the extracted audio data.

Examples

Read Data in Audio Datastore

Specify the file path to the audio samples included with Audio Toolbox™. Create an audio datastore that points to the specified folder.

```
folder = fullfile(matlabroot,'toolbox','audio','samples');
ADS = audioDatastore(folder);
```

While the audio datastore has unread files, read consecutive files from the datastore. Use `progress` to monitor the fraction of files read.

```
while hasdata(ADS)
    data = read(ADS);
    fprintf('Fraction of files read: %.2f\n',progress(ADS))
end
```

```
Fraction of files read: 0.03
Fraction of files read: 0.06
Fraction of files read: 0.08
Fraction of files read: 0.11
Fraction of files read: 0.14
Fraction of files read: 0.17
Fraction of files read: 0.19
Fraction of files read: 0.22
Fraction of files read: 0.25
Fraction of files read: 0.28
Fraction of files read: 0.31
Fraction of files read: 0.33
Fraction of files read: 0.36
Fraction of files read: 0.39
Fraction of files read: 0.42
Fraction of files read: 0.44
Fraction of files read: 0.47
Fraction of files read: 0.50
```

```
Fraction of files read: 0.53
Fraction of files read: 0.56
Fraction of files read: 0.58
Fraction of files read: 0.61
Fraction of files read: 0.64
Fraction of files read: 0.67
Fraction of files read: 0.69
Fraction of files read: 0.72
Fraction of files read: 0.75
Fraction of files read: 0.78
Fraction of files read: 0.81
Fraction of files read: 0.83
Fraction of files read: 0.86
Fraction of files read: 0.89
Fraction of files read: 0.92
Fraction of files read: 0.94
Fraction of files read: 0.97
Fraction of files read: 1.00
```

Return Information About Data

Specify the file path to the audio samples you want to include in the audio datastore. In this example, the samples are located on a local desktop. Create an audio datastore that points to the specified folder.

```
folder = 'C:\Users\audiouser\Desktop';
ADS = audioDatastore(folder,'LabelSource','foldernames');
```

When you read data from the datastore, you can additionally return information about the data as a structure. The information structure contains the file name, any labels associated with the file, and the sample rate of the file.

```
[data,info] = read(ADS);
info
```

```
info =

  struct with fields:

    SampleRate: 44100
      FileName: 'C:\Users\audiouser\Desktop\Turbine-16-44p1-mono-22secs.wav'
         Label: Desktop
```

Input Arguments

`ADS` — Audio datastore

`audioDatastore` object

Specify `ADS` as an `audioDatastore` object.

Output Arguments

`data` — Audio data

M-by-*N* matrix

Audio data, returned as a *M*-by-*N* matrix, where:

- *M* –- Total samples per channel in file.
- *N* –- Number of channels in file.

`info` — Information about audio data

struct

Information about audio data, returned as a struct with the following fields:

- `FileName` –- Name of the current file.
- `Label` –- All labels of the file.
- `SampleRate` –- Sample rate of the file.

Version History

Introduced in R2018b

See Also

`audioDatastore` | `hasdata` | `readall`

Topics

“Train Speech Command Recognition Model Using Deep Learning”
“Speaker Identification Using Pitch and MFCC”
“Denoise Speech Using Deep Learning Networks”

audioDatastore

Datastore for collection of audio files

Description

Use an `audioDatastore` object to manage a collection of audio files, where each individual audio file fits in memory, but the entire collection of audio files does not necessarily fit.

Creation

Syntax

```
ADS = audioDatastore(location)
ADS = audioDatastore(location,Name,Value)
```

Description

`ADS = audioDatastore(location)` creates a datastore `ADS` based on an audio file or collection of audio files in `location`.

`ADS = audioDatastore(location,Name,Value)` specifies additional properties using one or more name-value pair arguments.

Input Arguments

`location` — Files or folders to include in datastore

`FileSet` object | path | `DsFileSet` object

Files or folders included in the datastore, specified as a `FileSet` object, as file paths, or as a `DsFileSet` object.

- `FileSet` object — You can specify `location` as a `FileSet` object. Specifying the location as a `FileSet` object leads to a faster construction time for datastores compared to specifying a path or `DsFileSet` object. For more information, see `matlab.io.datastore.FileSet`.
- File path — You can specify a single file path as a character vector or string scalar. You can specify multiple file paths as a cell array of character vectors or a string array.
- `DsFileSet` object — You can specify a `DsFileSet` object. For more information, see `matlab.io.datastore.DsFileSet`.

Files or folders may be local or remote:

- Local files or folders — Specify local paths to files or folders. If the files are not in the current folder, then specify full or relative paths. Files within subfolders of the specified folder are not automatically included in the datastore. You can use the wildcard character (*) when specifying the local path. This character specifies that the datastore include all matching files or all files in the matching folders.
- Remote files or folders — Specify full paths to remote files or folders as a uniform resource locator (URL) of the form `hdfs:///path_to_file`. For more information, see “Work with Remote Data”.

When you specify a folder, the datastore includes only files with supported file formats and ignores files with any other format. To specify a custom list of file extensions to include in your datastore, see the `FileExtensions` property.

Example: `'song.wav'`

Example: `'../dir/music/song.wav'`

Example: `{'C:\dir\music\song.wav','C:\dir\speech\english.mp3'}`

Example: `'C:\dir\music\*.ogg'`

Data Types: `char` | `string` | `cell`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `ADS = audioDatastore('C:\dir\audiodata','FileExtensions','.ogg')`

`IncludeSubfolders` — Subfolder inclusion flag

`false` (default) | `true`

Subfolder inclusion flag, specified as the comma-separated pair consisting of `'IncludeSubfolders'` and `true` or `false`. Specify `true` to include all files and subfolders within each folder or `false` to include only the files within each folder.

If you do not specify `'IncludeSubfolders'`, then the default value is `false`.

Example: `'IncludeSubfolders',true`

Data Types: `logical` | `double`

`LabelSource` — Source providing label data

`'none'` (default) | `'foldernames'`

Source providing label data, specified as the comma-separated pair consisting of `'LabelSource'` and `'none'` or `'foldernames'`. If `'none'` is specified, then the `Labels` property is empty. If `'foldernames'` is specified, then labels are assigned according to the folder names and stored in the Labels property. You can later modify the labels by accessing the `Labels` property directly.

Data Types: `char` | `string`

`FileExtensions` — Audio file extensions

character vector | cell array of character vectors | string scalar | string array

Audio file extensions, specified as the comma-separated pair consisting of `'FileExtensions'` and a character vector, cell array of character vectors, string scalar, or string array. If you do not specify `'FileExtensions'`, then `audioDatastore` automatically includes all supported file types:

- .wav
- .avi
- .aif

- .aifc
- .aiff
- .mp3
- .au
- .snd
- .mp4
- .m4a
- .flac
- .ogg
- .mov
- .opus

Example: `'FileExtensions','.wav'`

Example: `'FileExtensions',{'.mp3','.mp4'}`

Data Types: `char` | `cell` | `string`

In addition to these name-value pairs, you also can specify any of the properties on this page as name-value pairs, except for the `Files` property.

Properties

`Files` — Files included in datastore

character vector | cell array of character vectors | string scalar | string array

Files included in the datastore, specified as a character vector, cell array of character vectors, string scalar, or string array. Each character vector or string is a full path to a file. The `location` argument in the `audioDatastore` defines `Files` when the datastore is created.

Data Types: `char` | `cell` | `string`

`Folders` — Folders used to create audio datastore

N-by-1 cell array of character vectors

This property is read-only.

Folders used to create the audio datastore, returned as an *N*-by-1 cell array of character vectors. Each row specifies a unique folder containing audio files that the `audioDatastore` object points to.

Data Types: `cell`

`Labels` — File labels

categorical, logical, or numeric vector | cell array | string array | table

File labels for the files in the datastore, specified as a vector, a cell array, a string array, or a table. The order of the labels in the array or table corresponds to the order of the associated files in the datastore.

If you specify `LabelSource` as `'foldernames'` when creating the `audioDatastore` object, then the label name for a file is the name of the folder containing it. If you do not specify `LabelSource` as `'foldernames'`, then `Labels` is an empty cell array or string array. If you change the `Files`

property after the datastore is created, then the `Labels` property is not automatically updated to incorporate the added fields.

Data Types: `categorical` | `cell` | `logical` | `double` | `single` | `string` | `table`

`OutputDataType` — Data type of output read
`'double'` (default) | `'native'`

Data type of the output, specified as `'double'` or `'native'`.

- `'double'` –– Double-precision normalized samples.
- `'native'` –– Native data type found in the file. Refer to `audioread` for more information about data types when `OutputDataType` is set to native.

The default value of this property is `'double'`.

Data Types: `char` | `string`

`AlternateFileSystemRoots` — Alternate file system root paths
string row vector | cell array of string vectors | cell array of character vectors

Alternate file system root paths, specified as a string row vector, a cell array of string vectors, or a cell array of character vectors. Use `AlternateFileSystemRoots` when you create a datastore on a local machine but must access and process data on another machine (possibly of a different operating system). Also, when processing data using Parallel Computing Toolbox and MATLAB Parallel Server™, and the data is stored on your local machines with a copy of the data available on different platform cloud or cluster machines, you must use `AlternateFileSystemRoots` to associate the root paths.

- To associate a set of root paths that are equivalent to one another, specify `AlternateFileSystemRoots` as a string vector. For example:

  ```
  ["Z:\datasets","/mynetwork/datasets"]
  ```
- To associate multiple sets of root paths that are equivalent for the datastore, specify `AlternateFileSystemRoots` as a cell array containing multiple rows, where each row represents a set of equivalent root paths. Specify each row in the cell array as either a string vector or a cell array of character vectors. For example:

  - Specify `AlternateFileSystemRoots` as a cell array of string vectors.

    ```
    {["Z:\datasets", "/mynetwork/datasets"]; ...
     ["Y:\datasets", "/mynetwork2/datasets","S:\datasets"]}
    ```
  - Alternatively, specify `AlternateFileSystemRoots` as a cell array of cell arrays of character vectors.

    ```
    {{'Z:\datasets', '/mynetwork/datasets'}; ...
     {'Y:\datasets", '/mynetwork2/datasets','S:\datasets'}}
    ```

The value of `AlternateFileSystemRoots` must satisfy these conditions:

- Contains one or more rows, where each row specifies a set of equivalent root paths.
- Each row specifies multiple root paths, and each root path must contain at least two characters.
- Root paths are unique and are not subfolders of one another.
- Contains at least one root path entry that points to the location of the files.

Data Types: `char` | `cell` | `string`

`SupportedOutputFormats` — Formats supported for writing audio files
`["wav","flac","ogg","opus","mp4","m4a"]`

This property is read-only.

Formats supported for writing audio files when using the `writeall` function, specified as `["wav","flac","ogg","opus","mp4","m4a"]`.

Data Types: `string`

`DefaultOutputFormat` — Default output audio file format
`"wav"` (default)

This property is read-only.

Default output format for writing audio files when using the `writeall` function, specified as `"wav"`.

Data Types: `string`

Object Functions

read	Read next consecutive audio file
readall	Read all audio files from datastore
reset	Reset datastore read pointer to start of data
hasdata	Return true if there is more data in datastore
shuffle	Shuffle files in datastore
subset	Create datastore with subset of files
preview	Read first file from datastore for preview
progress	Fraction of files read
splitEachLabel	Splits datastore according to specified label proportions
countEachLabel	Count number of unique labels
partition	Partition datastore and return on partitioned portion
numpartitions	Return estimate for reasonable number of partitions for parallel processing
combine	Combine data from multiple datastores
transform	Transform audio datastore
writeall	Write datastore to files
isPartitionable	Determine whether datastore is partitionable
isShuffleable	Determine whether datastore is shuffleable

Examples

Create Audio Datastore

Specify the file path to the audio samples included with Audio Toolbox™.

```
folder = fullfile(matlabroot,'toolbox','audio','samples');
```

Create an audio datastore that points to the specified folder.

```
ADS = audioDatastore(folder)
```

```
ADS =
  audioDatastore with properties:

                      Files: {
```

```
                                'B:\matlab\toolbox\audio\samples\Ambiance-16-44p1-mono-12secs.wav'
                                'B:\matlab\toolbox\audio\samples\AudioArray-16-16-4channels-20secs
                                ' ...\toolbox\audio\samples\ChurchImpulseResponse-16-44p1-mono-5se
                                 ... and 33 more
                                }
                      Folders: {
                                'B:\matlab\toolbox\audio\samples'
                                }
    AlternateFileSystemRoots: {}
              OutputDataType: 'double'
                      Labels: {}
      SupportedOutputFormats: ["wav"    "flac"    "ogg"    "opus"    "mp4"    "m4a"]
         DefaultOutputFormat: "wav"
```

Specify File Extensions to Include

Specify the file path to the audio samples included with Audio Toolbox™.

```
folder = fullfile(matlabroot,'toolbox','audio','samples');
```

Create an audio datastore that points to the `.ogg` files in the specified folder.

```
ADS = audioDatastore(folder,'FileExtension','.ogg')
```

```
ADS =
  audioDatastore with properties:

                       Files: {
                                'B:\matlab\toolbox\audio\samples\FemaleVolumeUp-16-mono-11secs.ogg
                                'B:\matlab\toolbox\audio\samples\Hey-16-mono-6secs.ogg';
                                'B:\matlab\toolbox\audio\samples\MaleVolumeUp-16-mono-6secs.ogg'
                                 ... and 3 more
                                }
                      Folders: {
                                'B:\matlab\toolbox\audio\samples'
                                }
    AlternateFileSystemRoots: {}
              OutputDataType: 'double'
                      Labels: {}
      SupportedOutputFormats: ["wav"    "flac"    "ogg"    "opus"    "mp4"    "m4a"]
         DefaultOutputFormat: "wav"
```

Version History

Introduced in R2018b

R2022b: Support for OPUS audio file format

The `audioDatastore` object supports reading and writing OPUS file format (`.opus`).

See Also

`datastore` | `mapreduce` | `tall`

Topics

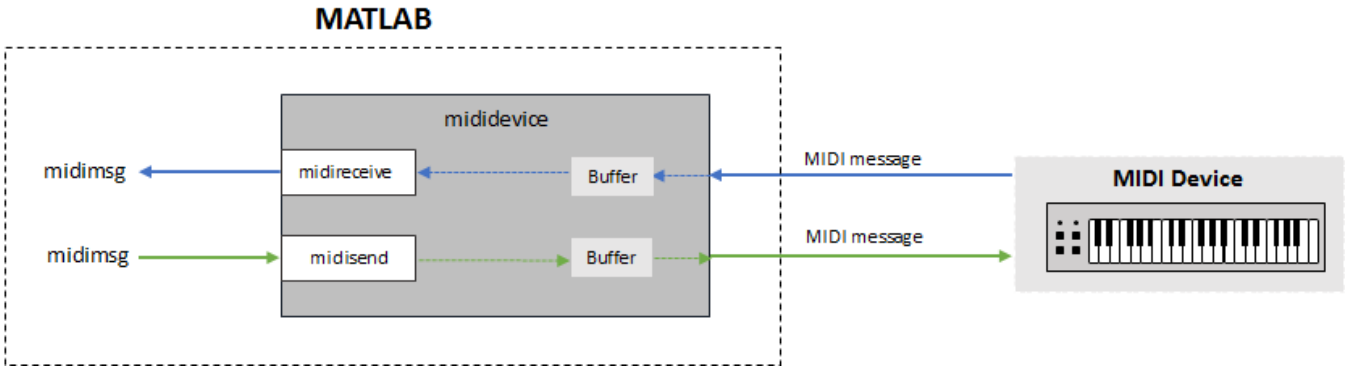
“Train Speech Command Recognition Model Using Deep Learning”
“Speaker Identification Using Pitch and MFCC”
“Denoise Speech Using Deep Learning Networks”

midimsg

Create MIDI message

Description

Create a MIDI message in MATLAB using `midimsg`. Create a MIDI device interface using `mididevice`. Send and receive messages using `midisend` and `midireceive`. When you create a MIDI message, you specify it as a MIDI message type.



For a tutorial on MIDI messages and interfacing with MIDI devices, see “MIDI Device Interface”.

Creation

Syntax

```
msg = midimsg('Note',channel,note,velocity,duration,timestamp)
msg = midimsg('NoteOn',channel,note,velocity,timestamp)
msg = midimsg('NoteOff',channel,note,velocity,timestamp)
msg = midimsg('ControlChange',channel,ccnumber,ccvalue,timestamp)
msg = midimsg('ProgramChange',channel,program,timestamp)
msg = midimsg('SystemExclusive',bytes,timestamp)
msg = midimsg('SystemExclusive',timestamp)
msg = midimsg('Data',bytes,timestamp)
msg = midimsg('EOX',timestamp)
msg = midimsg('TimingClock',timestamp)
msg = midimsg('Start',timestamp)
msg = midimsg('Continue',timestamp)
msg = midimsg('Stop',timestamp)
msg = midimsg('ActiveSensing',timestamp)
msg = midimsg('SystemReset',timestamp)
msg = midimsg('TuneRequest',timestamp)
msg = midimsg('MIDITimeCodeQuarterFrame',seq,value,timestamp)
msg = midimsg('SongPositionPointer',position,timestamp)
msg = midimsg('SongSelect',song,timestamp)
```

```
msg = midimsg('AllSoundOff',channel,timestamp)
msg = midimsg('ResetAllControllers',channel,timestamp)
msg = midimsg('LocalControl',channel,localcontrol,timestamp)
msg = midimsg('PolyOn',channel,timestamp)
msg = midimsg('MonoOn',channel,monoChannels,timestamp)
msg = midimsg('OmniOn',channel,timestamp)
msg = midimsg('OmniOff',channel,timestamp)
msg = midimsg('AllNotesOff',channel,timestamp)
msg = midimsg('PolyKeyPressure',channel,note,pressure,timestamp)
msg = midimsg('ChannelPressure',channel,pressure,timestamp)
msg = midimsg('PitchBend',channel,change,timestamp)
msg = midimsg
msg = midimsg(size)
msg = midimsg(0)
```

Description

`msg = midimsg('Note',channel,note,velocity,duration,timestamp)` returns two MIDI messages: `NoteOn` and `NoteOff`, with specified `Channel`, `Note`, `Velocity`, and `Timestamp` properties. The `Timestamp` property of the `NoteOff` message is determined as the `Timestamp` property of the `NoteOn` message plus the `duration`.

`msg = midimsg('NoteOn',channel,note,velocity,timestamp)` returns a `NoteOn` `midimsg`, with specified `Channel`, `Note`, `Velocity`, and `Timestamp` properties.

`msg = midimsg('NoteOff',channel,note,velocity,timestamp)` returns a `NoteOff` `midimsg`, with specified `Channel`, `Note`, `Velocity`, and `Timestamp` properties.

`msg = midimsg('ControlChange',channel,ccnumber,ccvalue,timestamp)` returns a `ControlChange` `midimsg`, with specified `Channel`, `CCNumber`, `CCValue`, and `Timestamp` properties.

`msg = midimsg('ProgramChange',channel,program,timestamp)` returns a `ProgramChange` `midimsg`, with specified `Channel`, `Program`, and `Timestamp` properties.

`msg = midimsg('SystemExclusive',bytes,timestamp)` returns a complete `SystemExclusive` message sequence, with specified `Timestamp` property.

`msg = midimsg('SystemExclusive',timestamp)` returns a `SystemExclusive` `midimsg`, with specified `Timestamp` property.

`msg = midimsg('Data',bytes,timestamp)` returns a `Data` `midimsg` for use in a System Exclusive message, with specified `MsgBytes` and `Timestamp` properties. `bytes` is specified as a scalar, vector, or multi-dimensional array of elements. Each element of `bytes` must be in the range [0,127].

`msg = midimsg('EOX',timestamp)` returns an `EOX` `midimsg`, with specified `Timestamp` property.

`msg = midimsg('TimingClock',timestamp)` returns a `TimingClock` `midimsg`, with specified `Timestamp` property.

`msg = midimsg('Start',timestamp)` returns a `Start` `midimsg`, with specified `Timestamp` property.

`msg = midimsg('Continue',timestamp)` returns a `Continue` `midimsg`, with specified `Timestamp` property.

`msg = midimsg('Stop',timestamp)` returns a `Stop` `midimsg`, with specified `Timestamp` property.

`msg = midimsg('ActiveSensing',timestamp)` returns a `ActiveSensing` `midimsg`, with specified `Timestamp` property.

`msg = midimsg('SystemReset',timestamp)` returns a `SystemReset` `midimsg`, with specified `Timestamp` property.

`msg = midimsg('TuneRequest',timestamp)` returns a `TuneRequest` `midimsg`, with specified `Timestamp` property.

`msg = midimsg('MIDITimeCodeQuarterFrame',seq,value,timestamp)` returns a `MIDITimeCodeQuarterFrame` `midimsg`, with specified `TimeCodeSequence`, `TimeCodeValue`, and `Timestamp` properties.

`msg = midimsg('SongPositionPointer',position,timestamp)` returns a `SongPositionPointer` `midimsg`, with specified `SongPosition` and `Timestamp` properties.

`msg = midimsg('SongSelect',song,timestamp)` returns a `SongSelect` `midimsg`, with specified `Song` and `Timestamp` properties.

`msg = midimsg('AllSoundOff',channel,timestamp)` returns a `AllSoundOff` `midimsg`, with specified `Channel` and `Timestamp` properties.

`msg = midimsg('ResetAllControllers',channel,timestamp)` returns a `ResetAllControllers` `midimsg`, with specified `Channel` and `Timestamp` properties.

`msg = midimsg('LocalControl',channel,localcontrol,timestamp)` returns a `LocalControl` `midimsg`, with specified `Channel`, `LocalControl`, and `Timestamp` properties.

`msg = midimsg('PolyOn',channel,timestamp)` returns a `PolyOn` `midimsg`, with specified `Channel` and `Timestamp` properties.

`msg = midimsg('MonoOn',channel,monoChannels,timestamp)` returns a `MonoOn` `midimsg`, with specified `Channel`, `MonoChannels`, and `Timestamp` properties.

`msg = midimsg('OmniOn',channel,timestamp)` returns an `OmniOn` `midimsg`, with specified `Channel` and `Timestamp` properties.

`msg = midimsg('OmniOff',channel,timestamp)` returns an `OmniOff` `midimsg`, with specified `Channel` and `Timestamp` properties.

`msg = midimsg('AllNotesOff',channel,timestamp)` returns an `AllNotesOff` `midimsg`, with specified `Channel` and `Timestamp` properties.

`msg = midimsg('PolyKeyPressure',channel,note,pressure,timestamp)` returns a `PolyKeyPressure` `midimsg`, with specified `Channel`, `Note`, `Pressure`, and `Timestamp` properties.

`msg = midimsg('ChannelPressure',channel,pressure,timestamp)` returns a `ChannelPressure` `midimsg`, with specified `Channel`, `Pressure`, and `Timestamp` properties.

`msg = midimsg('PitchBend',channel,change,timestamp)` returns a `PitchBend` `midimsg`, with specified `Channel`, `PitchChange`, and `Timestamp` properties.

`msg = midimsg` returns a scalar `midimsg` with all zero bytes. All zero bytes indicates a MIDI message with `Type` set to `Data`.

`msg = midimsg(size)` returns a `midimsg` array of `size` with all zero bytes.

`msg = midimsg(0)` returns an empty `midimsg`.

Note If `timestamp` is listed as an argument, it is optional and defaults to zero. The exception is the `'SystemExclusive',bytes,timestamp` syntax, in which case the timestamp argument is required.

Properties

Type — Type of MIDI message

`NoteOn` | `NoteOff` | `ControlChange` | `ProgramChange` | `SystemExclusive` | `Data` | `EOX` | ...

This property is read-only.

Type of MIDI message, returned as one of the following `midimsgtype` enumeration values:

`NoteOn`	`Data`	`Stop`	`SongPositionPointer`	`PolyOn`	`PolyKeyPressure`
`NoteOff`	`EOX`	`ActiveSensing`	`SongSelect`	`MonoOn`	`ChannelPressure`
`ControlChange`	`TimingClock`	`SystemReset`	`AllSoundOff`	`OmniOn`	`PitchBendChange`
`ProgramChange`	`Start`	`TuneRequest`	`ResetAllControllers`	`OmniOff`	`Undefined`
`SystemExclusive`	`Continue`	`MIDITimeCodeQuarterFrame`	`LocalControl`	`AllNotesOff`	

You can specify the type of MIDI message during creation as a character vector, string, or member of the `midimsgtype` enumeration.

For example, the following create equivalent MIDI messages:

- `midimsg('SongPositionPointer',1)`
- `midimsg("SongPositionPointer",1)`
- `midimsg(midimsgtype.SongPositionPointer,1)`

NumMsgBytes — Number of bytes in MIDI message

scalar | vector | array

This property is read-only.

Number of bytes in the MIDI message, returned as a scalar, vector, or array the same size as `msg`.

Data Types: `double`

`MsgBytes` — Actual bytes of constructed MIDI message (decimal)

scalar | vector | array

This property is read-only.

Actual bytes of the constructed MIDI message in decimal, returned as a scalar, vector, or array the same size as `msg`.

Data Types: `uint8`

`Timestamp` — Location in time for MIDI message

scalar | vector | array

Location in time for the MIDI message, specified as a scalar, vector, or array the same size as `msg`.

You can specify the timestamp as any numeric value. However, the timestamp is always stored and returned as type `double`.

For more on how MIDI timestamps are implemented in Audio Toolbox, see “MIDI Message Timing”.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`Channel` — MIDI channel to which message is addressed

integer in the range [1,16]

MIDI channel to which message is addressed, specified as an integer in the range [1,16].

Dependencies

This property is valid only for `NoteOn`, `NoteOff`, `PolyKeyPressure`, `AllSoundOff`, `ResetAllControllers`, `LocalControl`, `AllNotesOff`, `OmniOn`, `OmniOff`, `MonoOn`, `PolyOn`, `ControlChange`, `ProgramChange`, `ChannelPressure`, and `PitchBend` `midimsg` objects.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`Note` — MIDI note number

integer in the range [0,127]

MIDI note number, specified as an integer in the range [0,127]. The MIDI specification defines note number 60 as Middle C, and all other notes are relative. MIDI devices and software define the mapping between a note and a MIDI note number. If Middle C is arbitrarily assumed to be C5 for the target MIDI hardware or software, the following table maps between MIDI note numbers and notes:

	Note Numbers											
Octave	C	C#	D	D#	E	F	F#	G	G#	A	A#	B
0	0	1	2	3	4	5	6	7	8	9	10	11
1	12	13	14	15	16	17	18	19	20	21	22	23
2	24	25	26	27	28	29	30	31	32	33	34	35
3	36	37	38	39	40	41	42	43	44	45	46	47
4	48	49	50	51	52	53	54	55	56	57	58	59
5	60	61	62	63	64	65	66	67	68	69	70	71
6	72	73	74	75	76	77	78	79	80	81	82	83
7	84	85	86	87	88	89	90	91	92	93	94	95
8	96	97	98	99	100	101	102	103	104	105	106	107
9	108	109	110	111	112	113	114	115	116	117	118	119
10	120	121	122	123	124	125	126	127				

Dependencies

This property is valid only for `NoteOn`, `NoteOff`, and `PolyKeyPressure` `midimsg` objects.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`Velocity` — Velocity of MIDI message

integer in the range [0,127]

Velocity of MIDI message, specified as a scalar integer in the range [0,127]. Velocity describes how fast, or "hard," a note is played. A higher number corresponds to faster velocity.

Dependencies

This property is valid only for `NoteOn` and `NoteOff` `midimsg` objects.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`KeyPressure` — Key pressure

integer in the range [0,127]

Key pressure, specified as a scalar integer in the range [0,127]. Key pressure applies *aftertouch* to an individual note. For example, on a keyboard, key pressure describes the pressure applied to a key after that key has been struck (after a `NoteOn` message is sent). You can use `KeyPressure` to add expression to held notes.

Dependencies

This property is valid only for `PolyKeyPressure` `midimsg` objects.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`LocalControl` — Enable local control

`true` | `false`

Enable local control, specified as `true` or `false`. When local control is set to `false`, all devices on a given channel respond only to data received over MIDI.

Dependencies

This property is valid only for `LocalControl` `midimsg` objects.

Data Types: `logical`

`MonoChannels` — Channels for `MonoOn` messages

integer in the range [0,16]

Channels for `MonoOn` messages, specified as a scalar integer in the range [0,16].

Dependencies

This property is valid only for `MonoOn` `midimsg` objects.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`CCNumber` — Control change number

integer in the range [0,119]

Control change number, specified as an integer in the range [0,119].

Dependencies

This property is valid only for `ControlChange` `midimsg` objects.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`CCValue` — Control change value

integer in the range [0,127]

Control change value, specified as an integer in the range [0,127].

Dependencies

This property is valid only for `ControlChange` `midimsg` objects.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`Program` — Program number to switch to

integer in the range [0,127]

Program number to switch to, specified as an integer in the range [0,127].

Dependencies

This property is valid only for `ProgramChange` `midimsg` objects.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`ChannelPressure` — Channel pressure

integer in the range [0,127]

Channel pressure, specified as an integer in the range [0,127]. Key pressure applies *aftertouch* to all notes in a channel.

Dependencies

This property is valid only for `ChannelPressure` `midimsg` objects.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`PitchChange` — Amount of pitch change to apply

integer in the range [0,16383]

Amount of pitch change to apply, specified as an integer in the range [0,16383]. The center position (no effect) is `8192`. Sensitivity is a function of the receiver.

Dependencies

This property is valid only for `PitchBend` `midimsg` objects.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`TimeCodeSequence` — Sequence number

integer in the range [0,7]

Sequence number, specified as an integer in the range [0,7].

Dependencies

This property is valid only for `MIDITimeCodeQuarterFrame` `midimsg` objects.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`TimeCodeValue` — Time code value

integer in the range [0,15]

Time code value, specified as an integer in the range [0,15].

Dependencies

This property is valid only for `MIDITimeCodeQuarterFrame` `midimsg` objects.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`SongPosition` — Position in song to go to

integer in the range [0,16383]

Position in song to go to, specified as an integer in the range [0,16383].

Dependencies

This property is valid only for `SongPositionPointer` `midimsg` objects.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`Song` — Song number to switch to

integer in the range [0,127]

Song number to switch to, specified as an integer in the range [0,127].

Dependencies

This property is valid only for `SongSelect` `midimsg` objects.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Examples

Create Note Messages

You can create MIDI note messages using the `NoteOn` and `NoteOff` `midimsg` objects. A `NoteOn` message indicates that a note should begin playing. A `NoteOff` message indicates that a note should stop playing. Alternatively, you can send a second `NoteOn` message with velocity set to zero to indicate that the note should stop playing. The Audio Toolbox® provides a convenience syntax to create pairs of note on and note off messages.

Create a pair of MIDI messages to indicate a Note On and Note Off sequence using the `Note` convenience syntax. Specify that the note starts after one second, and has a duration of two seconds.

```
channel = 1;
note = 60;
velocity = 64;
duration = 2;
timestamp = 1;
msgs = midimsg('Note',channel,note,velocity,duration,timestamp)
```

```
msgs =
  MIDI message:
    NoteOn          Channel: 1  Note: 60  Velocity: 64  Timestamp: 1  [ 90 3C 40 ]
    NoteOn          Channel: 1  Note: 60  Velocity: 0   Timestamp: 3  [ 90 3C 00 ]
```

Two `midimsg` objects are created and returned as an array. The `Note` syntax returns the note off message as a `NoteOn` `midimsg` object with `Velocity` set to zero.

To create Note On and Note Off messages separately, create two `NoteOn` messages and concatenate them.

```
msgs = [midimsg('NoteOn',channel,note,velocity,timestamp), ...
        midimsg('NoteOn',channel,note,0,3)]
```

```
msgs =
  MIDI message:
    NoteOn          Channel: 1  Note: 60  Velocity: 64  Timestamp: 1  [ 90 3C 40 ]
    NoteOn          Channel: 1  Note: 60  Velocity: 0   Timestamp: 3  [ 90 3C 00 ]
```

You can also specify the Note Off using a `NoteOff` `midimsg` object. Using the `NoteOff` syntax enables you to specify a release velocity.

```
    msgs = [midimsg('NoteOn',channel,note,velocity,timestamp), ...
            midimsg('NoteOff',channel,note,velocity,3)]
```

```
msgs =
  MIDI message:
    NoteOn          Channel: 1  Note: 60  Velocity: 64  Timestamp: 1  [ 90 3C 40 ]
    NoteOff         Channel: 1  Note: 60  Velocity: 64  Timestamp: 3  [ 80 3C 40 ]
```

Control Change Messages for Control Surfaces

To create a control change message, specify the `midimsg` `Type` as `ControlChange` and set the required parameters: `Channel`, `CCNumber`, and `CCValue`. To determine the channel and control number assigned to your MIDI control surface, use `midiid`. Enter `midiid` at the Command Prompt and then move the control you want to identify.

```
[ccInfo,deviceName] = midiid;

Move the control you wish to identify; type ^C to abort.
Waiting for control message... done
```

`midiid` returns the control change number and channel as a single number according to the following formula: `ccInfo = (Channel*1000 + CCNumber)`. Define a MIDI Control Change message to move the identified controller. Your MIDI Control Surface must be bidirectional to receive Control Change messages.

```
channel = floor(ccInfo/1000);
ccnumber = ccInfo - channel*1000;
ccvalue = 1;
msg = midimsg('ControlChange',channel,ccnumber,ccvalue)

msg =
  MIDI message:
    ControlChange   Channel: 1  CCNumber: 16  CCValue: 1   Timestamp: 0  [ B0 10 01 ]
```

Create a `mididevice` object using the `deviceName` identified using `midiid`. Send the MIDI message to your device.

```
device = mididevice(deviceName);
midisend(device,msg);
```

Create a Program Change Message

Program Change messages, sometimes called "patch change" messages, specify how notes are interpreted. For example, a Program Change message can specify the instrument being played. To create a `ProgramChange` `midimsg` object, specify the `midimsg` type as `ProgramChange`, and the required property values: `Channel` and `Program`.

```
channel = 4;
program = 7;
msg = midimsg('ProgramChange',channel,program)

msg =
  MIDI message:
    ProgramChange   Channel: 4  Program: 7   Timestamp: 0  [ C3 07 ]
```

Create a System Exclusive Message

System Exclusive messages are defined by a sequence of `midimsg` objects: `SystemExclusive`, `Data`, and `EOX`. To create a System Exclusive sequence, specify the `SystemExclusive` `midimsg` type during creation and then specify the bytes of the message. This syntax requires a timestamp.

```
bytes = [0 1 2];
timestamp = 0;
msg = midimsg('SystemExclusive',bytes,timestamp)

msg =
  MIDI message:
    SystemExclusive Timestamp: 0  [ F0 ]
```

```
    Data            Timestamp: 0  [ 00 01 02 ]
    EOX             Timestamp: 0  [ F7 ]
```

You can also create the `SystemExclusive`, `Data`, and `EOX` `midimsg` objects individually. For example, the following `midimsg` array is the same as the preceding.

```
msg = [midimsg('SystemExclusive',timestamp), ...
       midimsg('Data',bytes,timestamp), ...
       midimsg('EOX',timestamp)]
```

```
msg =
  MIDI message:
    SystemExclusive Timestamp: 0  [ F0 ]
    Data            Timestamp: 0  [ 00 01 02 ]
    EOX             Timestamp: 0  [ F7 ]
```

Create a Scalar Default MIDI Message

The default MIDI message is a scalar with all zero bytes, and `Type` is `Data`.

```
msg = midimsg
```

```
msg =
  MIDI message:
    Data            Timestamp: 0  [ 00 00 00 00 00 00 00 00 ]
```

Preallocate Array of MIDI Messages

You can create a MIDI message array by specifying the size by a scalar or row vector.

If you specify the size as a scalar *M*, `midimsg` returns an *M*-by-*M* array with all zero bytes.

```
msg = midimsg(2)
```

```
msg =
  MIDI message:
    Data            Timestamp: 0  [ 00 00 00 00 00 00 00 00 ]
    Data            Timestamp: 0  [ 00 00 00 00 00 00 00 00 ]
    Data            Timestamp: 0  [ 00 00 00 00 00 00 00 00 ]
    Data            Timestamp: 0  [ 00 00 00 00 00 00 00 00 ]
```

An array of MIDI messages is always displayed vertically in order of their linear indexing. You can refer to individual elements of the array by specifying its position in each dimension, or by its linear index. For example, change the `Timestamp` of the third element from `0` to `2 using linear indexing`, and then from 2 to 3 using first dimensional indexing.

```
msg(3).Timestamp = 2
```

```
msg =
  MIDI message:
    Data            Timestamp: 0  [ 00 00 00 00 00 00 00 00 ]
    Data            Timestamp: 0  [ 00 00 00 00 00 00 00 00 ]
    Data            Timestamp: 2  [ 00 00 00 00 00 00 00 00 ]
    Data            Timestamp: 0  [ 00 00 00 00 00 00 00 00 ]
```

```
msg(1,2).Timestamp = 3

msg =
  MIDI message:
    Data            Timestamp: 0  [ 00 00 00 00 00 00 00 00 ]
    Data            Timestamp: 0  [ 00 00 00 00 00 00 00 00 ]
    Data            Timestamp: 3  [ 00 00 00 00 00 00 00 00 ]
    Data            Timestamp: 0  [ 00 00 00 00 00 00 00 00 ]
```

You can also specify nonsymmetric arrays. If you specify the size as a row vector of two or more elements, `midimsg` returns an *M*-by-*N*-by-...-*X* multidimensional array. For example, to specify a three dimensional array with each dimension having a different number of elements, specify the size as a row vector of three elements.

```
msg = midimsg([2,1,3])

msg =
  MIDI message:
    Data            Timestamp: 0  [ 00 00 00 00 00 00 00 00 ]
    Data            Timestamp: 0  [ 00 00 00 00 00 00 00 00 ]
    Data            Timestamp: 0  [ 00 00 00 00 00 00 00 00 ]
    Data            Timestamp: 0  [ 00 00 00 00 00 00 00 00 ]
    Data            Timestamp: 0  [ 00 00 00 00 00 00 00 00 ]
    Data            Timestamp: 0  [ 00 00 00 00 00 00 00 00 ]

size(msg)

ans = 1×3

     2     1     3
```

Create Empty MIDI Message

```
msg = midimsg(0)

msg =

  empty MIDI message array
```

Manipulate Array of MIDI Messages

In this example, you create an array of MIDI messages, and then index into the array in a loop to define a melody.

Create a 22-by-1 array of MIDI messages with all zero data.

```
msgArray = midimsg([22,1]);
```

To create a melody, create MIDI `NoteOn` and `NoteOff` messages by indexing in a loop. Display the result.

```
melody = [60,65,60,57,55,53,60,65,60,67,60];
for i = 1:numel(melody)
```

```
    idx = (2*i-1):(2*i);
    msgArray(idx) = midimsg('Note',1,melody(i),50,0.5,i);
end
msgArray
```

```
msgArray =
  MIDI message:
    NoteOn          Channel: 1  Note: 60  Velocity: 50  Timestamp: 1  [ 90 3C 32 ]
    NoteOn          Channel: 1  Note: 60  Velocity: 0   Timestamp: 1.5  [ 90 3C 00 ]
    NoteOn          Channel: 1  Note: 65  Velocity: 50  Timestamp: 2  [ 90 41 32 ]
    NoteOn          Channel: 1  Note: 65  Velocity: 0   Timestamp: 2.5  [ 90 41 00 ]
    NoteOn          Channel: 1  Note: 60  Velocity: 50  Timestamp: 3  [ 90 3C 32 ]
    NoteOn          Channel: 1  Note: 60  Velocity: 0   Timestamp: 3.5  [ 90 3C 00 ]
    NoteOn          Channel: 1  Note: 57  Velocity: 50  Timestamp: 4  [ 90 39 32 ]
    NoteOn          Channel: 1  Note: 57  Velocity: 0   Timestamp: 4.5  [ 90 39 00 ]
    NoteOn          Channel: 1  Note: 55  Velocity: 50  Timestamp: 5  [ 90 37 32 ]
    NoteOn          Channel: 1  Note: 55  Velocity: 0   Timestamp: 5.5  [ 90 37 00 ]
    NoteOn          Channel: 1  Note: 53  Velocity: 50  Timestamp: 6  [ 90 35 32 ]
    NoteOn          Channel: 1  Note: 53  Velocity: 0   Timestamp: 6.5  [ 90 35 00 ]
    NoteOn          Channel: 1  Note: 60  Velocity: 50  Timestamp: 7  [ 90 3C 32 ]
    NoteOn          Channel: 1  Note: 60  Velocity: 0   Timestamp: 7.5  [ 90 3C 00 ]
    NoteOn          Channel: 1  Note: 65  Velocity: 50  Timestamp: 8  [ 90 41 32 ]
    NoteOn          Channel: 1  Note: 65  Velocity: 0   Timestamp: 8.5  [ 90 41 00 ]
    NoteOn          Channel: 1  Note: 60  Velocity: 50  Timestamp: 9  [ 90 3C 32 ]
    NoteOn          Channel: 1  Note: 60  Velocity: 0   Timestamp: 9.5  [ 90 3C 00 ]
    NoteOn          Channel: 1  Note: 67  Velocity: 50  Timestamp: 10  [ 90 43 32 ]
    NoteOn          Channel: 1  Note: 67  Velocity: 0   Timestamp: 10.5  [ 90 43 00 ]
    NoteOn          Channel: 1  Note: 60  Velocity: 50  Timestamp: 11  [ 90 3C 32 ]
    NoteOn          Channel: 1  Note: 60  Velocity: 0   Timestamp: 11.5  [ 90 3C 00 ]
```

The order of the MIDI messages in the array is only important for readability. When you send MIDI messages using a `mididevice` object, the `mididevice` object reorders your MIDI messages according to their timestamps and sends them in chronological order. Create a `PitchBend` MIDI message to bend the fourth note downward and add it to the MIDI message array. For readability, sort the MIDI message array by `Timestamp`.

```
msg = midimsg('PitchBend',1,7192,4.01);
msgArray = [msgArray;msg]
```

```
msgArray =
  MIDI message:
    NoteOn          Channel: 1  Note: 60  Velocity: 50  Timestamp: 1  [ 90 3C 32 ]
    NoteOn          Channel: 1  Note: 60  Velocity: 0   Timestamp: 1.5  [ 90 3C 00 ]
    NoteOn          Channel: 1  Note: 65  Velocity: 50  Timestamp: 2  [ 90 41 32 ]
    NoteOn          Channel: 1  Note: 65  Velocity: 0   Timestamp: 2.5  [ 90 41 00 ]
    NoteOn          Channel: 1  Note: 60  Velocity: 50  Timestamp: 3  [ 90 3C 32 ]
    NoteOn          Channel: 1  Note: 60  Velocity: 0   Timestamp: 3.5  [ 90 3C 00 ]
    NoteOn          Channel: 1  Note: 57  Velocity: 50  Timestamp: 4  [ 90 39 32 ]
    NoteOn          Channel: 1  Note: 57  Velocity: 0   Timestamp: 4.5  [ 90 39 00 ]
    NoteOn          Channel: 1  Note: 55  Velocity: 50  Timestamp: 5  [ 90 37 32 ]
    NoteOn          Channel: 1  Note: 55  Velocity: 0   Timestamp: 5.5  [ 90 37 00 ]
    NoteOn          Channel: 1  Note: 53  Velocity: 50  Timestamp: 6  [ 90 35 32 ]
    NoteOn          Channel: 1  Note: 53  Velocity: 0   Timestamp: 6.5  [ 90 35 00 ]
    NoteOn          Channel: 1  Note: 60  Velocity: 50  Timestamp: 7  [ 90 3C 32 ]
    NoteOn          Channel: 1  Note: 60  Velocity: 0   Timestamp: 7.5  [ 90 3C 00 ]
    NoteOn          Channel: 1  Note: 65  Velocity: 50  Timestamp: 8  [ 90 41 32 ]
    NoteOn          Channel: 1  Note: 65  Velocity: 0   Timestamp: 8.5  [ 90 41 00 ]
    NoteOn          Channel: 1  Note: 60  Velocity: 50  Timestamp: 9  [ 90 3C 32 ]
```

```
    NoteOn          Channel: 1  Note: 60  Velocity: 0   Timestamp: 9.5  [ 90 3C 00 ]
    NoteOn          Channel: 1  Note: 67  Velocity: 50  Timestamp: 10  [ 90 43 32 ]
    NoteOn          Channel: 1  Note: 67  Velocity: 0   Timestamp: 10.5  [ 90 43 00 ]
    NoteOn          Channel: 1  Note: 60  Velocity: 50  Timestamp: 11  [ 90 3C 32 ]
    NoteOn          Channel: 1  Note: 60  Velocity: 0   Timestamp: 11.5  [ 90 3C 00 ]
    PitchBend       Channel: 1  PitchChange: 7192 Timestamp: 4.01  [ E0 18 38 ]
```

```
timeStamps = [msgArray.Timestamp];
[~,idx] = sort(timeStamps);

msgArray = msgArray(idx)
```

```
msgArray =
  MIDI message:
    NoteOn          Channel: 1  Note: 60  Velocity: 50  Timestamp: 1  [ 90 3C 32 ]
    NoteOn          Channel: 1  Note: 60  Velocity: 0   Timestamp: 1.5  [ 90 3C 00 ]
    NoteOn          Channel: 1  Note: 65  Velocity: 50  Timestamp: 2  [ 90 41 32 ]
    NoteOn          Channel: 1  Note: 65  Velocity: 0   Timestamp: 2.5  [ 90 41 00 ]
    NoteOn          Channel: 1  Note: 60  Velocity: 50  Timestamp: 3  [ 90 3C 32 ]
    NoteOn          Channel: 1  Note: 60  Velocity: 0   Timestamp: 3.5  [ 90 3C 00 ]
    NoteOn          Channel: 1  Note: 57  Velocity: 50  Timestamp: 4  [ 90 39 32 ]
    PitchBend       Channel: 1  PitchChange: 7192 Timestamp: 4.01  [ E0 18 38 ]
    NoteOn          Channel: 1  Note: 57  Velocity: 0   Timestamp: 4.5  [ 90 39 00 ]
    NoteOn          Channel: 1  Note: 55  Velocity: 50  Timestamp: 5  [ 90 37 32 ]
    NoteOn          Channel: 1  Note: 55  Velocity: 0   Timestamp: 5.5  [ 90 37 00 ]
    NoteOn          Channel: 1  Note: 53  Velocity: 50  Timestamp: 6  [ 90 35 32 ]
    NoteOn          Channel: 1  Note: 53  Velocity: 0   Timestamp: 6.5  [ 90 35 00 ]
    NoteOn          Channel: 1  Note: 60  Velocity: 50  Timestamp: 7  [ 90 3C 32 ]
    NoteOn          Channel: 1  Note: 60  Velocity: 0   Timestamp: 7.5  [ 90 3C 00 ]
    NoteOn          Channel: 1  Note: 65  Velocity: 50  Timestamp: 8  [ 90 41 32 ]
    NoteOn          Channel: 1  Note: 65  Velocity: 0   Timestamp: 8.5  [ 90 41 00 ]
    NoteOn          Channel: 1  Note: 60  Velocity: 50  Timestamp: 9  [ 90 3C 32 ]
    NoteOn          Channel: 1  Note: 60  Velocity: 0   Timestamp: 9.5  [ 90 3C 00 ]
    NoteOn          Channel: 1  Note: 67  Velocity: 50  Timestamp: 10  [ 90 43 32 ]
    NoteOn          Channel: 1  Note: 67  Velocity: 0   Timestamp: 10.5  [ 90 43 00 ]
    NoteOn          Channel: 1  Note: 60  Velocity: 50  Timestamp: 11  [ 90 3C 32 ]
    NoteOn          Channel: 1  Note: 60  Velocity: 0   Timestamp: 11.5  [ 90 3C 00 ]
```

Version History

Introduced in R2018a

See Also

`parameterTuner` | **Audio Test Bench** | `midisend` | `midireceive` | `mididevice`

Topics

“MIDI Device Interface”

External Websites

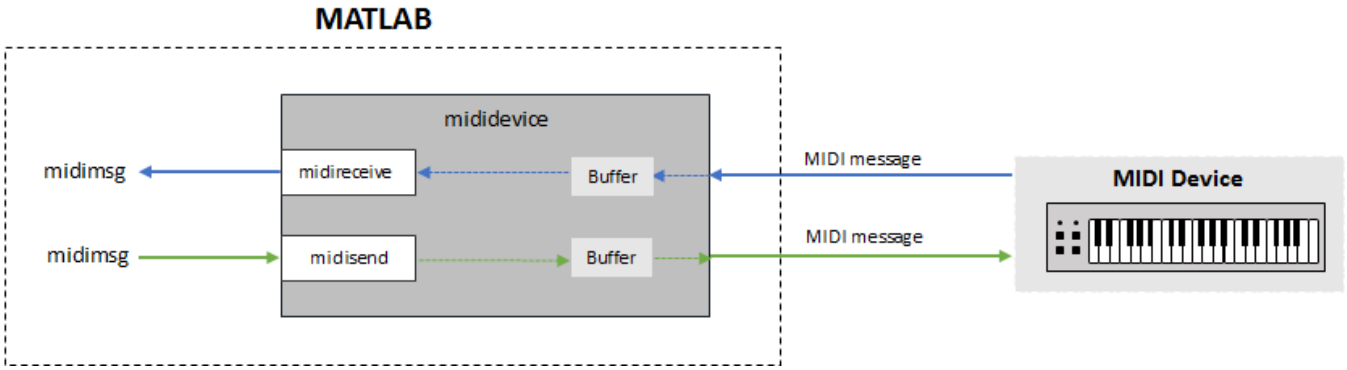
MIDI Manufacturers Association

mididevice

Send and receive MIDI messages

Description

Interface to a MIDI device in MATLAB using `mididevice`. Package MIDI messages using `midimsg`. Send and receive messages using `midisend` and `midireceive`. Use `mididevinfo` to query your system for available MIDI devices.



For a tutorial on interfacing with MIDI devices, see “MIDI Device Interface”.

Creation

Syntax

```
device = mididevice(deviceNameOrID)
device = mididevice('Input',inDeviceNameOrID)
device = mididevice('Output',outDeviceNameOrID)
device = mididevice('Input',inDeviceNameOrID,'Output',outDeviceNameOrID)
```

Description

`device = mididevice(deviceNameOrID)` returns an interface to the MIDI device specified by `deviceNameOrID`. If the MIDI device supports MIDI in and MIDI out, then `device` also supports MIDI in and MIDI out.

`device = mididevice('Input',inDeviceNameOrID)` returns an input interface to the MIDI input device, `inDeviceNameOrID`.

`device = mididevice('Output',outDeviceNameOrID)` returns an output interface to the MIDI output device, `outDeviceNameOrID`.

`device = mididevice('Input',inDeviceNameOrID,'Output',outDeviceNameOrID)` returns a MIDI I/O interface, where input is received from `inDeviceNameOrID` and output is sent to `outDeviceNameOrID`.

Properties

`Input` — Input device name associated with `mididevice`

empty `char` array (default)

This property is read-only.

Input device name attached to your `mididevice` object, returned as a character array.

`Input` is set during the creation of the `mididevice` object and cannot be modified later.

Data Types: `char`

`Output` — Output device name associated with `mididevice`

empty `char` array (default)

This property is read-only.

Output device name attached to your `mididevice` object, returned as a character array

`Output` is set during the creation of the `mididevice` object and cannot be modified later.

Data Types: `char`

`InputID` — Input device ID associated with `mididevice`

`-1` (default)

This property is read-only.

Unique MIDI input device ID attached to your `mididevice` object, returned as a scalar double. If your system includes different MIDI devices with the same name, using the device ID removes ambiguity.

`InputID` is set during the creation of the `mididevice` object and cannot be modified later.

Data Types: `double`

`OutputID` — Output device name associated with `mididevice`

`-1` (default)

This property is read-only.

Unique MIDI output device ID attached to your `mididevice` object, returned as a scalar double. If your system includes different MIDI devices with the same name, using the device ID removes ambiguity.

`OutputID` is set during the creation of the `mididevice` object and cannot be modified later.

Data Types: `double`

Object Functions

midisend	Send MIDI message to MIDI device
midireceive	Receive MIDI message from MIDI device
hasdata	Determine if data is available to read from MIDI device

Examples

Connect Input and Output to Single MIDI Device

Query your system for available MIDI devices.

```
mididevinfo
```

```
  MIDI devices available:
  ID  Direction  Interface   Name
   0   output    MMSystem   'Microsoft MIDI Mapper'
   1    input    MMSystem   'USB MIDI Interface '
   2   output    MMSystem   'Microsoft GS Wavetable Synth'
   3   output    MMSystem   'USB MIDI Interface '
```

Create a MIDI device object to interface with your selected device. If you specify a single MIDI device object, and it is capable of both input and output, `mididevice` connects to both the input and output.

```
device = mididevice('USB MIDI Interface ')
```

```
device =
  mididevice connected to
     Input: 'USB MIDI Interface ' (1)
    Output: 'USB MIDI Interface ' (3)
```

Connect Input to MIDI Device

Query your system for MIDI devices.

```
mididevinfo
```

```
  MIDI devices available:
  ID  Direction  Interface   Name
   0   output    MMSystem   'Microsoft MIDI Mapper'
   1    input    MMSystem   'USB MIDI Interface '
   2   output    MMSystem   'Microsoft GS Wavetable Synth'
   3   output    MMSystem   'USB MIDI Interface '
```

Create a MIDI device object to interface with your selected input device. As soon as you create the MIDI device object, it begins listening for MIDI messages and storing them in a buffer.

```
device = mididevice('Input','USB MIDI Interface ');
```

Connect Output to MIDI Device

Query your system for available MIDI devices.

```
mididevinfo
```

```
  MIDI devices available:
  ID  Direction  Interface   Name
   0   output    MMSystem   'Microsoft MIDI Mapper'
```

```
   1    input    MMSystem   'USB MIDI Interface '
   2   output    MMSystem   'Microsoft GS Wavetable Synth'
   3   output    MMSystem   'USB MIDI Interface '
```

Create a MIDI device object to interface with your selected output device.

```
device = mididevice('Output','USB MIDI Interface ')
```

```
device =
  mididevice connected to
    Output: 'USB MIDI Interface ' (3)
```

Connect Input and Output to Different MIDI Devices

Query your system for available MIDI devices.

```
mididevinfo
```

```
  MIDI devices available:
  ID  Direction  Interface   Name
   0   output    MMSystem   'Microsoft MIDI Mapper'
   1    input    MMSystem   'USB MIDI Interface '
   2   output    MMSystem   'Microsoft GS Wavetable Synth'
   3   output    MMSystem   'USB MIDI Interface '
```

Create a MIDI device object that receives data from one device and sends data to another device. In this example, the MIDI device object receives MIDI messages from the `'USB MIDI Interface '` device and sends data to the `'Microsoft GS Wavetable Synth'` virtual output device. To avoid ambiguity, the MIDI devices are specified by the device IDs.

```
device = mididevice('Input',1,'Output',2)
```

```
device =
  mididevice connected to
     Input: 'USB MIDI Interface ' (1)
    Output: 'Microsoft GS Wavetable Synth' (2)
```

Version History

Introduced in R2018a

See Also

`parameterTuner` | **Audio Test Bench** | `midisend` | `midireceive` | `mididevinfo` | `midimsg`

Topics

“MIDI Device Interface”

External Websites

MIDI Manufacturers Association

hasdata

Determine if data is available to read from MIDI device

Syntax

```
tf = hasdata(device)
```

Description

`tf = hasdata(device)` returns logical `1` (`true`) if there is data available to read from the `mididevice` specified by `device`. Otherwise, it returns logical `0` (`false`).

Examples

Determine if Data Is Available to Receive

Create a `mididevice` object to interface with your MIDI device. Query your system for available MIDI devices.

```
mididevinfo
```

```
  MIDI devices available:
  ID  Direction  Interface   Name
   0   output    MMSystem   'Microsoft MIDI Mapper'
   1    input    MMSystem   'nanoKONTROL2'
   2    input    MMSystem   'USB Uno MIDI Interface'
   3   output    MMSystem   'Microsoft GS Wavetable Synth'
   4   output    MMSystem   'nanoKONTROL2'
   5   output    MMSystem   'USB Uno MIDI Interface'
```

```
device = mididevice('USB Uno MIDI Interface')
```

```
device =
  mididevice connected to
     Input: 'USB Uno MIDI Interface' (2)
    Output: 'USB Uno MIDI Interface' (5)
```

As soon as your `mididevice` object is created, it begins listening for MIDI messages and storing them in a buffer. When you call `midireceive`, MIDI messages are retrieved from the buffer and returned. You can use `hasdata` to query whether your `mididevice` object buffer contains unread MIDI messages.

```
hasdata(device)
```

```
ans = logical
   0
```

Input Arguments

`device` — `mididevice` object

`mididevice` object

Specify `device` as an object created by `mididevice`.

Version History

Introduced in R2018a

See Also

`midisend` | `mididevice` | `mididevinfo` | `midimsg`

Topics

“MIDI Device Interface”

External Websites

MIDI Manufacturers Association

midireceive

Receive MIDI message from MIDI device

Syntax

```
msgs = midireceive(device)
msgs = midireceive(device,maxmsgs)
```

Description

`msgs = midireceive(device)` returns the MIDI messages, `msgs`, received from a MIDI device using the MIDI device interface, `device`.

`msgs = midireceive(device,maxmsgs)` specifies the maximum number of MIDI messages to return as `maxmsgs`.

Examples

Receive MIDI Messages

To determine what MIDI devices are attached to your MIDI input ports, call `mididevinfo`. Use the `availableDevices` struct to specify a valid MIDI device to create a `mididevice` object.

```
availableDevices = mididevinfo;
device = mididevice(availableDevices.input(1).ID);
```

Once your MIDI `device` object is created, it begins listening to MIDI messages from your specified device and storing them in a buffer. To get all MIDI messages in the buffer, call `midireceive`. In this example, several keys on a MIDI keyboard are played.

```
msgs = midireceive(device)
```

```
msgs =

  MIDI message:
    NoteOn          Channel: 1  Note: 52  Velocity: 64  Timestamp: 3.94  [ 90 34 40 ]
    NoteOn          Channel: 1  Note: 52  Velocity: 0   Timestamp: 4.179  [ 90 34 00 ]
    NoteOn          Channel: 1  Note: 48  Velocity: 64  Timestamp: 4.19  [ 90 30 40 ]
    NoteOn          Channel: 1  Note: 47  Velocity: 64  Timestamp: 4.382  [ 90 2F 40 ]
    NoteOn          Channel: 1  Note: 48  Velocity: 0   Timestamp: 4.459  [ 90 30 00 ]
    NoteOn          Channel: 1  Note: 48  Velocity: 64  Timestamp: 4.59  [ 90 30 40 ]
    NoteOn          Channel: 1  Note: 47  Velocity: 0   Timestamp: 4.776  [ 90 2F 00 ]
    NoteOn          Channel: 1  Note: 50  Velocity: 64  Timestamp: 4.788  [ 90 32 40 ]
    NoteOn          Channel: 1  Note: 47  Velocity: 64  Timestamp: 4.802  [ 90 2F 40 ]
    NoteOn          Channel: 1  Note: 52  Velocity: 64  Timestamp: 4.831  [ 90 34 40 ]
    NoteOn          Channel: 1  Note: 47  Velocity: 0   Timestamp: 4.84  [ 90 2F 00 ]
    NoteOn          Channel: 1  Note: 48  Velocity: 0   Timestamp: 4.912  [ 90 30 00 ]
    NoteOn          Channel: 1  Note: 52  Velocity: 0   Timestamp: 4.953  [ 90 34 00 ]
    NoteOn          Channel: 1  Note: 50  Velocity: 0   Timestamp: 5.079  [ 90 32 00 ]
```

Reading from the buffer clears the data. For example, if no more MIDI messages are sent, and the buffer is reread, `midireceive` returns an empty MIDI message.

```
msgs = midireceive(device)
```

```
msgs =

  empty MIDI message array
```

Receive Limited Number of MIDI Messages

Query your system for available output from MIDI devices. Specify that the output of a MIDI device is connected to the input of your `mididevice` object.

```
mididevinfo
```

```
  MIDI devices available:
  ID  Direction  Interface   Name
   0   output    MMSystem   'Microsoft MIDI Mapper'
   1    input    MMSystem   'USB MIDI Interface '
   2   output    MMSystem   'Microsoft GS Wavetable Synth'
   3   output    MMSystem   'USB MIDI Interface '
```

```
device = mididevice('Input','USB MIDI Interface ');
```

Once your MIDI `device` object is created, it begins listening to MIDI messages from your specified device and storing them in a buffer. To get a limited number of MIDI messages from the buffer, call `midireceive` and specify the maximum number of messages to return. In this example, five keys are played on a MIDI device. A maximum of four MIDI messages are received at each call to `midireceive`.

```
midireceive(device,4)
```

```
ans =

  MIDI message:
    NoteOn          Channel: 1  Note: 36  Velocity: 64  Timestamp: 2929.71  [ 90 24 40 ]
    NoteOn          Channel: 1  Note: 36  Velocity: 0   Timestamp: 2929.91  [ 90 24 00 ]
    NoteOn          Channel: 1  Note: 37  Velocity: 64  Timestamp: 2930.43  [ 90 25 40 ]
    NoteOn          Channel: 1  Note: 37  Velocity: 0   Timestamp: 2930.59  [ 90 25 00 ]
```

```
midireceive(device,4)
```

```
ans =

  MIDI message:
    NoteOn          Channel: 1  Note: 38  Velocity: 64  Timestamp: 2931.16  [ 90 26 40 ]
    NoteOn          Channel: 1  Note: 38  Velocity: 0   Timestamp: 2931.32  [ 90 26 00 ]
    NoteOn          Channel: 1  Note: 39  Velocity: 64  Timestamp: 2931.87  [ 90 27 40 ]
    NoteOn          Channel: 1  Note: 39  Velocity: 0   Timestamp: 2932.01  [ 90 27 00 ]
```

```
midireceive(device,4)
```

```
ans =

  MIDI message:
    NoteOn          Channel: 1  Note: 40  Velocity: 64  Timestamp: 2932.52  [ 90 28 40 ]
    NoteOn          Channel: 1  Note: 40  Velocity: 0   Timestamp: 2932.66  [ 90 28 00 ]
```

Input Arguments

`device` — Object of `mididevice`

object of `mididevice`

Specify `device` as an object created by `mididevice`.

`maxmsgs` — Maximum number of messages to return
positive integer scalar

Maximum number of messages to return, specified as a positive integer scalar.

Data Types: `single` | `double`

Output Arguments

`msgs` — Object of `midimsg`
scalar | column vector

Object of `midimsg`, returned as a scalar or column vector. The number of MIDI messages in the `mididevice` buffer and `maxmsgs` determine the size of `msgs`.

Version History

Introduced in R2018a

See Also

`midisend` | `mididevice` | `mididevinfo` | `midimsg`

Topics
“MIDI Device Interface”

External Websites
MIDI Manufacturers Association

midisend

Send MIDI message to MIDI device

Syntax

```
midisend(device,msg)
midisend(device,varargin)
```

Description

`midisend(device,msg)` sends the MIDI message, `msg`, to a MIDI device using the MIDI device interface, `device`.

`midisend(device,varargin)` creates MIDI messages using `varargin` and then sends the MIDI messages. The `varargin` syntax is for convenience and includes a call to `midimsg` with the call to `midisend`.

Examples

Send MIDI Messages to Device

Query your system for available MIDI device output ports. Use the `availableDevices` struct to specify a valid MIDI device and create a `mididevice` object.

```
availableDevices = mididevinfo;
device = mididevice(availableDevices.output(2).ID);
```

Create a pair of `NoteOn` messages (to indicate Note On and Note Off) and send them to your selected MIDI device.

```
msgs = midimsg('Note',1,48,64,0.25);
midisend(device,msgs)
```

Define and Send MIDI Messages to Device

`midisend` enables you to combine the definition and sending of a `midimsg` into a single function call. Send middle C on channel 3 with velocity 64.

```
mididevinfo
```

```
  MIDI devices available:
  ID  Direction  Interface   Name
   0   output    MMSystem   'Microsoft MIDI Mapper'
   1    input    MMSystem   'nanoKONTROL2'
   2    input    MMSystem   'USB Uno MIDI Interface'
   3   output    MMSystem   'Microsoft GS Wavetable Synth'
   4   output    MMSystem   'nanoKONTROL2'
   5   output    MMSystem   'USB Uno MIDI Interface'
```

```
device = mididevice('USB Uno MIDI Interface')
```

```
device =
  mididevice connected to
     Input: 'USB Uno MIDI Interface' (2)
    Output: 'USB Uno MIDI Interface' (5)
```

```
midisend(device,'NoteOn',3,60,64)
```

Compile and Play MIDI Messages

Get the name of an available output MIDI device on your system.

```
mInfo = mididevinfo;
```

```
Disregard cmd.exe warnings about UNC directory pathnames.
Disregard cmd.exe warnings about UNC directory pathnames.
```

```
midiDeviceName = mInfo.output(1).Name;
```

Create a `mididevice` object.

```
device = mididevice(midiDeviceName);
```

Create a MIDI message array.

```
msgs = [];
for ii = 1:8
    msgs = [msgs;midimsg('Note',1,20+8*ii,64,1,ii)];
end
```

To listen to the MIDI messages, send the MIDI messages to your device.

```
midisend(device,msgs)
```

To compile the previous steps, encapsulate the code in a function and then call `mcc`.

```
function playMusic1()
    mInfo = mididevinfo;
    midiDeviceName = mInfo.output(1).Name;
    device = mididevice(midiDeviceName);

    msgs = [];
    for ii = 1:8
        msgs = [msgs;midimsg('Note',1,20+8*ii,64,1,ii)];
    end

    midisend(device,msgs)
end
```

```
mcc playMusic1 -m -w disable
```

Execute the compiled code. You will not hear any sound. This is because the executable opened, sent the MIDI messages to the queue, and then closed, aborting its commands before the MIDI messages had a chance to play.

```
!playMusic1.exe
```

To keep the executable open long enough for the MIDI messages to play, add a pause to the executable. Set the duration of the pause to equal the duration of the MIDI messages.

```
function playMusic2()
    mInfo = mididevinfo;
    midiDeviceName = mInfo.output(1).Name;
    device = mididevice(midiDeviceName);

    msgs = [];
    for ii = 1:8
        msgs = [msgs;midimsg('Note',1,20+8*ii,64,1,ii)];
    end

    midisend(device,msgs)
    pause(msgs(end).Timestamp)
end

mcc playMusic2 -m -w disable
```

Play the compiled executable. The sound that plays through your MIDI device is the same as the uncompiled version.

```
!playMusic2.exe
```

Input Arguments

`device` — Object of `mididevice`

scalar

Specify `device` as an object created by `mididevice`.

`msg` — Object of `midimsg`

scalar | vector | array

Specify `msg` as an object created by `midimsg`.

`varargin` — Variable number of arguments describing MIDI message

`midimsg` input arguments

Specify `varargin` as a valid combination of arguments that can construct a MIDI message. See `midimsg` for a description of valid arguments.

Version History

Introduced in R2018a

See Also

`midireceive` | `mididevice` | `mididevinfo` | `midimsg`

Topics

“MIDI Device Interface”

External Websites
MIDI Manufacturers Association

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 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 “Audio Plugins in MATLAB”.

The `audioPlugin` class is a `handle` class.

Methods

Public Methods

setLatencyInSamples	Set latency in samples reported to DAW
getSampleRate	Get sample rate at which the plugin is run
setSampleRate	Set sample rate at which the plugin is run

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
```

Design Valid Audio Plugin That Uses `setLatencyInSamples`

Design an `audioPlugin` class that uses the `setLatencyInSamples` method to report the latency of the plugin. The plugin in this example, `simpleDelay`, delays the audio signal by a fixed integer and reports the delay to the host application.

```
classdef simpleDelay < audioPlugin
    % simpleDelay Add delay to audio signal
    %   This plugin adds a 100 sample delay to the audio input and reports
    %   the latency to the host application.
    properties (Access = private)
        Delay
    end
    methods
        function plugin = simpleDelay
            plugin.Delay = dsp.Delay(100);
        end
        function out = process(plugin,in)
            out = plugin.Delay(in);
        end
        function reset(plugin)
            setLatencyInSamples(plugin,100)
        end
    end
end
```

This example is intended to show the pattern for using `setLatencyInSamples`. For a detailed use-case, see `audiopluginexample.FastConvolver` in the "Audio Plugin Example Gallery".

Version History

Introduced in R2016a

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`audioPluginParameter` | `generateAudioPlugin` | `validateAudioPlugin` | `audioPluginInterface` | `audioPluginSource` | `audioPluginConfig` | `parameterTuner` | **Audio Test Bench**

Topics
“Design an Audio Plugin”
“Audio Plugins in MATLAB”
“Audio Plugin Example Gallery”

setLatencyInSamples

Class: `audioPlugin`

Set latency in samples reported to DAW

Syntax

```
setLatencyInSamples(myAudioPlugin,latency)
```

Description

`setLatencyInSamples(myAudioPlugin,latency)` sets the latency, in samples, that `myAudioPlugin` reports to a digital audio workstation (DAW) or other host application. Specify `latency` as a positive integer.

Note Latency is reported to a host application when the `reset` method is called. As a best practice, call `setLatencyInSamples` in the `reset` method of your `audioPlugin` class.

Version History

Introduced in R2020b

See Also

`audioPlugin`

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`.

Version History

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.

Version History

Introduced in R2016a

audioPluginConfig

Specify coder configuration of audio plugin

Description

The `audioPluginConfig` object enables you to validate and generate audio plugins that use deep learning pretrained networks. This object also allows you to pass code replacement libraries to the `generateAudioPlugin` function.

Creation

Syntax

```
obj = audioPluginConfig(Name,Value)
```

Description

`obj = audioPluginConfig(Name,Value)` creates an object that describes the coder configuration for your audio plugin. Use name-value arguments to specify the properties of the object.

This object generates a constant property called `PluginConfig` for audio plugin classes. Use the `audioPluginConfig` object if your plugin uses deep learning networks or a code replacement library.

Properties

`DeepLearningConfig` — Deep learning library configuration
`[]` (default) | `coder.DeepLearningConfig("none")` | `coder.DeepLearningConfig("mkldnn")`

Deep learning library configuration, specified as an empty array (`[]`), `coder.DeepLearningConfig("none")`, or `coder.DeepLearningConfig("mkldnn")`.

You can also use the `generateAudioPlugin` user interface (UI) to specify the deep learning library for plugin generation.

Value	`generateAudioPlugin` UI Setting	Description
`[]`	Set **Deep learning library** to `None`	Do not use a deep learning library.
`coder.DeepLearningConfig("none")`	Set **Deep learning library** to `Plain C`	Generate code that does not use any third-party library.

Value	generateAudioPlugin UI Setting	Description
`coder.DeepLearningConfig("mkldnn")`	Set **Deep learning library** to `Intel MKL-DNN`	Generate code that uses the Intel Math Kernel Library for Deep Neural Networks (Intel MKL-DNN). This option does not work on Macintosh platforms using ARM® processors. This option is not supported with the `-win32` option of the `generateAudioPlugin` function. • On Intel Macintosh platforms, `generateAudioPlugin` packages the required libraries (`libdnnl.1.4.dylib`, `libdnnl.1.dylib`, `libdnnl.dylib`, and `libomp.dylib`) within the generated plugin bundle. The path to required the libraries is set to the `INTEL_MKLDNN` environment variable. You must install the libraries. To distribute the generated plugin, you must have licenses to distribute the Intel MKL-DNN and OpenMP libraries. • On Microsoft Windows platforms, `generateAudioPlugin` creates upon compilation a folder named *`pluginName`*`_juceproject_NetworkWeights` in the build directory. The folder contains the network weight files that are read by the generated plugin. When you add the generated plugin to a third-party DAW, you must copy the generated folder along with the plugin binary to your DAW's plugin location. The generated plugin will work in a DAW only if the required library MKL-DNN is visible to the DAW. To make the MKL-DNN library visible to a DAW, you must add the path to the MKL-DNN library to the Windows environment variable `PATH`. • On Linux platforms, `generateAudioPlugin` creates upon compilation a folder named `.MWPluginData/`*`pluginName`*`_juceproject_NetworkWeights` in your home directory. The folder contains the network weight files that are read by the generated plugin. The generated plugin will work in a DAW only if the required library MKL-DNN is visible to the DAW. To make the MKL-DNN library visible to a DAW, you must keep the MKL-DNN library in the `/usr/lib` directory or in the `/usr/local/lib` directory.

Value	generateAudioPlugin UI Setting	Description
		For more information about installing the MKL-DNN library and setting the related environment variables, see “Prerequisites for Deep Learning with MATLAB Coder” (MATLAB Coder). This option is not supported in MATLAB Online.

You must have MATLAB Coder Interface for Deep Learning installed to use this property unless you choose the `[]` option. For more information, see `coder.DeepLearningConfig`.

`CodeReplacementLibrary` — Code replacement library configuration
`""` (default) | `"none"` | `"Intel AVX (Windows)"` | `"DSP Intel AVX2-FMA (Windows)"` | `"DSP Intel AVX2-FMA (Linux)"` | `"DSP Intel AVX2-FMA (Mac)"`

Code replacement library configuration, specified as an empty string (`""`), `"none"`, `"Intel AVX (Windows)"`, `"DSP Intel AVX2-FMA (Windows)"`, `"DSP Intel AVX2-FMA (Linux)"`, or `"DSP Intel AVX2-FMA (Mac)"`.

You can also use the `generateAudioPlugin` user interface (UI) to specify the code replacement library for plugin generation.

Value	generateAudioPlugin UI Setting	Description
`""` or `"none"`	Set **Code replacement library** to `None`	Do not use a code replacement library.
`"Intel AVX (Windows)"`	Set **Code replacement library** to `Intel AVX (Windows)`	Generate code that uses the Intel AVX code replacement library. This option works only on Windows platforms. This option is not supported with the `-win32` option of the `generateAudioPlugin` function.
`"DSP Intel AVX2-FMA (Windows)"`	Set **Code replacement library** to `DSP Intel AVX2-FMA (Windows)`	Generate code that uses the Intel DSP AVX2-FMA code replacement library. This option works only on Windows platforms. This option is not supported with the `-win32` option of the `generateAudioPlugin` function.
`"DSP Intel AVX2-FMA (Linux)"`	Set **Code replacement library** to `DSP Intel AVX2-FMA (Linux)`	Generate a JUCE project that uses the Intel DSP AVX2-FMA code replacement library for Linux platforms. This option works only with the `-juceproject` option of the `generateAudioPlugin` function.
`"DSP Intel AVX2-FMA (Mac)"`	Set **Code replacement library** to `DSP Intel AVX2-FMA (Mac)`	Generate code that uses the Intel DSP AVX2-FMA code replacement library. This option works only on Intel Mac platforms.

You must have Embedded Coder® installed to use this property. For more information about code replacement libraries, see “What Is Code Replacement Customization?” (Embedded Coder). For more information on using the DSP AVX2-FMA code replacement libraries with System objects, see “System objects in DSP System Toolbox that Support SIMD Code Generation”.

Examples

Audio Configuration Information for Plugin Class Definition

Create the source file for a plugin class, `MyAudioPlugin`, that uses the Intel AVX code replacement library for Windows. Add a processing function to the class.

```
classdef MyAudioPlugin < audioPlugin
    properties (Constant)
        PluginConfig = audioPluginConfig( ...
            'DeepLearningConfig',coder.DeepLearningConfig('none'), ...
            'CodeReplacementLibrary','Intel AVX (Windows)');
    end
    methods
        function out = process(~,in)
            out = in;
        end
    end
end
```

To validate the plugin, use the `validateAudioPlugin` function. To generate the plugin, use the `generateAudioPlugin` function.

Audio Configuration Information on the Command Line

Create a `DeepLearningConfigBase` configuration object that generates code that does not use any third-party library. Use the `audioPluginConfig` object to specify a plugin that incorporates the previous property and uses the Intel AVX code replacement library for Windows. Generate the audio plugin.

```
dlcfg = coder.DeepLearningConfig('none');
cfg = audioPluginConfig( ...
    'DeepLearningConfig',dlcfg, ...
    'CodeReplacementLibrary','Intel AVX (Windows)');
generateAudioPlugin -audioconfig cfg MyAudioPlugin
```

Version History

Introduced in R2021b

See Also

Functions

`generateAudioPlugin` | `validateAudioPlugin`

Objects

`audioPlugin` | `audioPluginInterface` | `audioPluginParameter` | `audioPluginSource` | `coder.DeepLearningConfig`

Apps

Audio Test Bench

Topics
“Design an Audio Plugin”
“Audio Plugins in MATLAB”
“Audio Plugin Example Gallery”

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 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.

The `audioPluginSource` class is a `handle` class.

Methods

Public Methods

getSamplesPerFrame	Get frame size returned by the plugin
setSamplesPerFrame	Set frame size returned by the plugin (MATLAB environment only)

Inherited Methods

setLatencyInSamples	Set latency in samples reported to DAW
getSampleRate	Get sample rate at which the plugin is run
setSampleRate	Set sample rate at which the plugin is run

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.

Version History

Introduced in R2016a

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`validateAudioPlugin` | `generateAudioPlugin` | `audioPluginParameter` | `audioPluginInterface` | `audioPlugin` | `audioPluginConfig` | `parameterTuner` | **Audio Test Bench**

Topics

“Audio Plugins in MATLAB”
“Audio Plugin Example Gallery”
“Hierarchies of Classes — Concepts”

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.

Version History

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.

Version History

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”.

The `externalAudioPlugin` class is a `handle` class.

Methods

Public 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

setLatencyInSamples	Set latency in samples reported to DAW
getSampleRate	Get sample rate at which the plugin is run
setSampleRate	Set sample rate at which the plugin is run

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)
```

Use `info` to return information about the hosted plugin.

```
info(hostedPlugin)
```

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)
```

Alternatively, you can use `getParameter` to return the normalized value of a single parameter.

```
parameterIndex = 5;
parameterValue = getParameter(hostedPlugin,parameterIndex)
```

Run External Plugin in MATLAB

Load a VST audio plugin into MATLAB™ by specifying its full path. If you are using a Mac, replace the `.dll` file extension with `.vst`.

```
pluginPath = fullfile(matlabroot,'toolbox','audio','samples','ParametricEqualizer.dll');
hostedPlugin = loadAudioPlugin(pluginPath);
```

Create input and output objects for an audio stream loop that reads from a file and writes to your audio device. Set the sample rate of the hosted plugin to the sample rate of the input to the plugin.

```
fileReader = dsp.AudioFileReader('FunkyDrums-44p1-stereo-25secs.mp3');
deviceWriter = audioDeviceWriter('SampleRate',fileReader.SampleRate);
setSampleRate(hostedPlugin,fileReader.SampleRate);
```

Set the `MediumPeakGain` property to -20 dB.

```
hostedPlugin.MediumPeakGain = -20;
```

Use the hosted plugin to process the audio file in an audio stream loop. Sweep the medium peak gain upward in the loop to hear the effect.

```
while hostedPlugin.MediumPeakGain < 19
    hostedPlugin.MediumPeakGain = hostedPlugin.MediumPeakGain + 0.04;
    x = fileReader();
    y = process(hostedPlugin,x);
    deviceWriter(y);
end

release(fileReader)
release(deviceWriter)
```

Limitations

- Saving an external plugin as a MAT-file and then loading it preserves the external settings and parameters of the plugin but does not preserve its internal state or memory. Do not save and load your plugins when you are processing audio.

Version History

Introduced in R2016b

See Also

`externalAudioPluginSource` | `audioPluginSource` | `audioPlugin` | `loadAudioPlugin` | `parameterTuner` | **Audio Test Bench**

Topics

“Host External Audio Plugins”
“Hierarchies of Classes — Concepts”

dispParameter

Class: `externalAudioPlugin`

Display information of single or multiple parameters

Syntax

```
dispParameter(hostedPlugin)
dispParameter(hostedPlugin,parameter)
```

Description

`dispParameter(hostedPlugin)` displays all parameters and associated indices, values, displayed values, and display labels. For example:

```
dispParameter(hostedPlugin)
```

```
               Parameter    Value    Display
             _____________________________________

     1               Wet:   1.0000    +0.0 dB
     2               Dry:   1.0000    +0.0 dB
     3        1: Enabled:   1.0000      ON
     4         1: Length:   0.0000     0.0 ms
     5         1: Length:   0.0156    4.00 8N
     6       1: Feedback:   0.0000    -inf dB
     7        1: Lowpass:   1.0000   20000 Hz
     8         1: Hipass:   0.0000       0 Hz
     9     1: Resolution:   1.0000      24 bits
    10   1: Stereo width:   1.0000    1.00
    11         1: Volume:   1.0000    +0.0 dB
    12            1: Pan:   0.5000     0.0 %
```

The `Value` column corresponds to the normalized parameter value. Generally, the normalized parameter value represents the position of a UI widget or MIDI controller. The `Display` column corresponds to an internal parameter value used for processing. The `Value` and `Display` are related by an unknown mapping that is internal to the hosted plugin.

`dispParameter(hostedPlugin,parameter)` displays a subset of parameters. You can specify a parameter by its name as a character vector, string, or as a vector of one or more parameter indices. For example:

- `dispParameter(hostedPlugin,'Gain')` displays information about the `'Gain'` parameter of `hostedPlugin`.
- `dispParameter(hostedPlugin,[1,3])` displays information about parameters specified by indices 1 and 3.

Version History

Introduced in R2016b

getParameter

Class: `externalAudioPlugin`

Get normalized value and information about parameter

Syntax

```
value = getParameter(hostedPlugin,parameter)
[value, parameterInformation] = getParameter(hostedPlugin,parameter)
```

Description

`value = getParameter(hostedPlugin,parameter)` returns the normalized value of the `parameter` of `hostedPlugin`. You can specify a parameter by its name as a character vector, string, or by its index. For example:

- `getParameter(hostedPlugin,'Gain')` returns the normalized value of the hosted plugin parameter named `'Gain'`. If the parameter name is not unique, `getParameter` returns an error.
- `getParameter(hostedPlugin,2)` returns information about the parameter specified by index 2.

`[value, parameterInformation] = getParameter(hostedPlugin,parameter)` returns a structure containing additional information about the specified parameter of the hosted plugin.

Field	Description
`DisplayName`	Display name or prompt of the plugin parameter, returned as a character vector. The display name is intended for display on the plugin 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.
`Label`	Label intended for display with `DisplayValue` on the plugin UI, returned as a character vector. Typical labels include `dB` and `Hz`.

Version History

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, VST 3, and AU.
`InputChannels`	Number of channels passed to the processing function of the plugin.
`OutputChannels`	Number of channels returned from the processing function of the plugin.
`NumParams`	Total number of plugin parameters.
`PluginPath`	Path specified when plugin is loaded using `loadAudioPlugin`.
`VendorName`	Name of the plugin creator.
`VendorVersion`	Version number. Typically used to track plugin releases.
`UniqueID`	Unique identifier of plugin used for recognition in certain digital audio workstation (DAW) environments.

Version History

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.

Version History

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.

Version History

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”.

The `externalAudioPluginSource` class is a `handle` class.

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

setLatencyInSamples	Set latency in samples reported to DAW
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)

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)
```

Use `info` to return information about the hosted plugin.

```
info(hostedSourcePlugin)
```

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)
```

Alternatively, you can use `getParameter` to return the normalized value of a single parameter.

```
getParameter(hostedSourcePlugin,1)
```

Run External Source Plugin in MATLAB

Load a VST audio source plugin into MATLAB™ by specifying its full path. If you are using a Mac, replace the `.dll` file extension with `.vst`.

```
pluginPath = fullfile(matlabroot,'toolbox','audio','samples','oscillator.dll');
hostedSourcePlugin = loadAudioPlugin(pluginPath);
```

Set the `Amplitude` property to 0.5. Set the `Frequency` property to 16 kHz.

```
hostedSourcePlugin.Amplitude = 0.5;
hostedSourcePlugin.Frequency = 16000;
```

Set the sample rate at which to run the plugin. Create an output object to write to your audio device.

```
setSampleRate(hostedSourcePlugin,44100);
deviceWriter = audioDeviceWriter('SampleRate',44100);
```

Use the hosted source plugin to output an audio stream. The processing in the audio stream loop ramps the frequency parameter down and then up.

```
k = 1;
for i = 1:1000
    hostedSourcePlugin.Frequency = hostedSourcePlugin.Frequency - 30*k;
    y = process(hostedSourcePlugin);
    deviceWriter(y);
    if (hostedSourcePlugin.Frequency - 30 <= 0.1) || (hostedSourcePlugin.Frequency + 30 >= 20e3)
        k = -1*k;
    end
end

release(deviceWriter)
```

Limitations

- Saving an external plugin as a MAT-file and then loading it preserves the external settings and parameters of the plugin but does not preserve its internal state or memory. Do not save and load your plugins when you are processing audio.

Version History

Introduced in R2016b

See Also

`parameterTuner` | **Audio Test Bench** | `loadAudioPlugin` | `audioPlugin` | `audioPluginSource` | `externalAudioPlugin`

Topics

“Host External Audio Plugins”
“Hierarchies of Classes — Concepts”

ivectorSystem

Create i-vector system

Description

i-vectors are compact statistical representations of identity extracted from audio signals. `ivectorSystem` creates a trainable i-vector system to extract i-vectors and perform classification tasks such as speaker recognition, speaker diarization, and sound classification. You can also determine thresholds for open set tasks and enroll labels into the system for both open and closed set classification.

Creation

Syntax

```
ivs = ivectorSystem
ivs = ivectorSystem(Name=Value)
```

Description

`ivs = ivectorSystem` creates a default i-vector system. You can train the i-vector system to extract i-vectors and perform classification tasks.

`ivs = ivectorSystem(Name=Value)` specifies nondefault properties for `ivs` using one or more name-value arguments.

Properties

`InputType` — Type of input
`"audio"` (default) | `"features"`

Input type, specified as `"audio"` or `"features"`.

- `"audio"` –– The i-vector system accepts mono audio signals as input. The audio data is processed to extract 20 mel frequency cepstral coefficients (MFCCs), delta MFCCs, and delta-delta MFCCs for 60 coefficients per frame.

  If `InputType` is set to `"audio"` when the i-vector system is created, the training data can be:

  - A cell array of single-channel audio signals, each specified as a column vector with underlying type `single` or `double`.
  - An `audioDatastore` object or a `signalDatastore` object that points to a data set of mono audio signals.
  - A `TransformedDatastore` with an underlying `audioDatastore` or `signalDatastore` that points to a data set of mono audio signals. The output from calls to `read` from the transform datastore must be mono audio signals with underlying data type `single` or `double`.

- `"features"` –– The i-vector accepts pre-extracted audio features as input.

  If `InputType` is set to `"features"` when the i-vector system is created, the training data can be:

  - A cell array of matrices with underlying type `single` or `double`. The matrices must consist of audio features where the number of features (columns) is locked the first time `trainExtractor` is called and the number of hops (rows) is variable-sized. The number of features input in any subsequent calls to any of the object functions must be equal to the number of features used when calling `trainExtractor`.
  - A `TransformedDatastore` object with an underlying `audioDatastore` or `signalDatastore` whose `read` function has output as described in the previous bullet.
  - A `signalDatastore` object whose `read` function has output as described in the first bullet.

Example: `ivs = ivectorSystem(InputType="audio")`

Data Types: `char` | `string`

`SampleRate` — Sample rate of audio input in Hz
`16000` (default) | positive scalar

Sample rate of the audio input in Hz, specified as a positive scalar.

Note The `"SampleRate"` property applies only when `InputType` is set to `"audio"`.

Example: `ivs = ivectorSystem(InputType="audio",SampleRate=48000)`

Data Types: `single` | `double`

`DetectSpeech` — Apply speech detection
`true` (default) | `false`

Apply speech detection, specified as `true` or `false`. With `DetectSpeech` set to `true`, the i-vector system extracts features only from regions where speech is detected.

Note The `DetectSpeech` property applies only when `InputType` is set to `"audio"`.

`ivectorSystem` uses the `detectSpeech` function to detect regions of speech.

Example: `ivs = ivectorSystem(InputType="audio",DetectSpeech=true)`

Data Types: `logical` | `single` | `double`

`Verbose` — Display training progress
`true` (default) | `false`

Display training progress, specified as `true` or `false`. With `Verbose` set to `true`, the i-vector system displays the training progress in the command window or the Live Editor.

Tip To toggle between verbose and non-verbose behavior, use dot notation to set the `Verbose` property between object function calls.

Example: `ivs = ivectorSystem(InputType="audio",Verbose=false)`

Data Types: `logical` | `single` | `double`

`EnrolledLabels` — Table containing enrolled labels

`0`-by-2 table (default)

This property is read-only.

Table containing enrolled labels, specified as a table. Table row names correspond to labels and column names correspond to the template i-vector and the number of individual i-vectors used to generate the template i-vector. The number of i-vectors used to generate the template i-vector may be viewed as a measure of confidence in the template.

- Use `enroll` to enroll new labels or update existing labels.
- Use `unenroll` to remove labels from the system.

Data Types: `table`

Object Functions

trainExtractor	Train i-vector extractor
trainClassifier	Train i-vector classifier
calibrate	Train i-vector system calibrator
enroll	Enroll labels
unenroll	Unenroll labels
detectionErrorTradeoff	Evaluate binary classification system
verify	Verify label
identify	Identify label
ivector	Extract i-vector
info	Return training configuration and data info
addInfoHeader	Add custom information about i-vector system
release	Allow property values and input characteristics to change

Examples

Train Speaker Verification System

Use the Pitch Tracking Database from Graz University of Technology (PTDB-TUG) [1] on page 4-288. The data set consists of 20 English native speakers reading 2342 phonetically rich sentences from the TIMIT corpus. Download and extract the data set.

```
downloadFolder = matlab.internal.examples.downloadSupportFile("audio","ptdb-tug.zip");
dataFolder = tempdir;
unzip(downloadFolder,dataFolder)
dataset = fullfile(dataFolder,"ptdb-tug");
```

Create an `audioDatastore` object that points to the data set. The data set was originally intended for use in pitch-tracking training and evaluation and includes laryngograph readings and baseline pitch decisions. Use only the original audio recordings.

```
ads = audioDatastore([fullfile(dataset,"SPEECH DATA","FEMALE","MIC"),fullfile(dataset,"SPEECH DA
                      IncludeSubfolders=true, ...
                      FileExtensions=".wav");
```

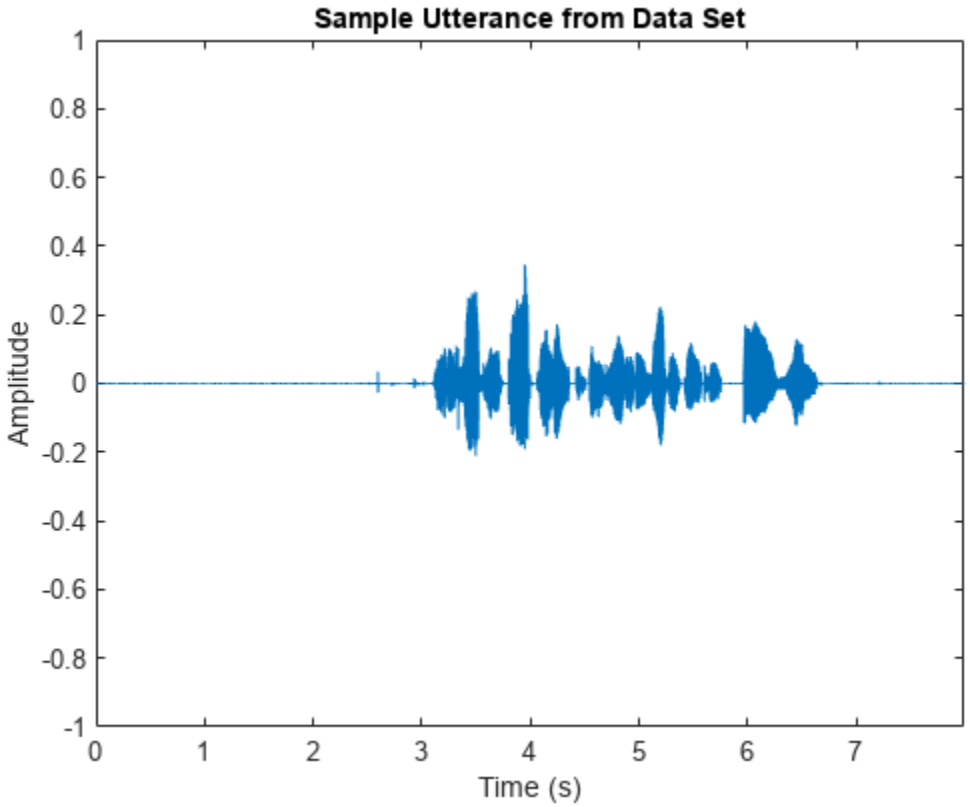
The file names contain the speaker IDs. Decode the file names to set the labels in the `audioDatastore` object.

```
ads.Labels = extractBetween(ads.Files,"mic_","_");
countEachLabel(ads)
```

```
ans=20×2 table
    Label    Count
    _____    _____

     F01      236
     F02      236
     F03      236
     F04      236
     F05      236
     F06      236
     F07      236
     F08      234
     F09      236
     F10      236
     M01      236
     M02      236
     M03      236
     M04      236
     M05      236
     M06      236
      ⋮
```

Read an audio file from the data set, listen to it, and plot it.

```
[audioIn,audioInfo] = read(ads);
fs = audioInfo.SampleRate;

t = (0:size(audioIn,1)-1)/fs;
sound(audioIn,fs)
plot(t,audioIn)
xlabel("Time (s)")
ylabel("Amplitude")
axis([0 t(end) -1 1])
title("Sample Utterance from Data Set")
```



Separate the `audioDatastore` object into four: one for training, one for enrollment, one to evaluate the detection-error tradeoff, and one for testing. The training set contains 16 speakers. The enrollment, detection-error tradeoff, and test sets contain the other four speakers.

```
speakersToTest = categorical(["M01","M05","F01","F05"]);

adsTrain = subset(ads,~ismember(ads.Labels,speakersToTest));

ads = subset(ads,ismember(ads.Labels,speakersToTest));
[adsEnroll,adsTest,adsDET] = splitEachLabel(ads,3,1);
```

Display the label distributions of the `audioDatastore` objects.

```
countEachLabel(adsTrain)
```

```
ans=16×2 table
    Label    Count
    _____    _____

     F02      236
     F03      236
     F04      236
     F06      236
     F07      236
     F08      234
     F09      236
     F10      236
     M02      236
```

```
     M03      236
     M04      236
     M06      236
     M07      236
     M08      236
     M09      236
     M10      236
```

```
countEachLabel(adsEnroll)
```

```
ans=4×2 table
    Label    Count
    _____    _____

     F01       3
     F05       3
     M01       3
     M05       3
```

```
countEachLabel(adsTest)
```

```
ans=4×2 table
    Label    Count
    _____    _____

     F01       1
     F05       1
     M01       1
     M05       1
```

```
countEachLabel(adsDET)
```

```
ans=4×2 table
    Label    Count
    _____    _____

     F01      232
     F05      232
     M01      232
     M05      232
```

Create an i-vector system. By default, the i-vector system assumes the input to the system is mono audio signals.

```
speakerVerification = ivectorSystem(SampleRate=fs)
```

```
speakerVerification =
  ivectorSystem with properties:

         InputType: 'audio'
        SampleRate: 48000
      DetectSpeech: 1
           Verbose: 1
    EnrolledLabels: [0×2 table]
```

To train the extractor of the i-vector system, call `trainExtractor`. Specify the number of universal background model (UBM) components as 128 and the number of expectation maximization iterations as 5. Specify the total variability space (TVS) rank as 64 and the number of iterations as 3.

```
trainExtractor(speakerVerification,adsTrain, ...
    UBMNumComponents=128,UBMNumIterations=5, ...
    TVSRank=64,TVSNumIterations=3)
```

```
Calculating standardization factors ....done.
Training universal background model ........done.
Training total variability space ......done.
i-vector extractor training complete.
```

To train the classifier of the i-vector system, use `trainClassifier`. To reduce dimensionality of the i-vectors, specify the number of eigenvectors in the projection matrix as 16. Specify the number of dimensions in the probabilistic linear discriminant analysis (PLDA) model as 16, and the number of iterations as 3.

```
trainClassifier(speakerVerification,adsTrain,adsTrain.Labels, ...
    NumEigenvectors=16, ...
    PLDANumDimensions=16,PLDANumIterations=3)
```

```
Extracting i-vectors ...done.
Training projection matrix .....done.
Training PLDA model ......done.
i-vector classifier training complete.
```

To calibrate the system so that scores can be interpreted as a measure of confidence in a positive decision, use `calibrate`.

```
calibrate(speakerVerification,adsTrain,adsTrain.Labels)
```

```
Extracting i-vectors ...done.
Calibrating CSS scorer ...done.
Calibrating PLDA scorer ...done.
Calibration complete.
```

To inspect parameters used previously to train the i-vector system, use `info`.

```
info(speakerVerification)
```

```
i-vector system input
  Input feature vector length: 60
  Input data type: double

trainExtractor
  Train signals: 3774
  UBMNumComponents: 128
  UBMNumIterations: 5
  TVSRank: 64
  TVSNumIterations: 3

trainClassifier
  Train signals: 3774
  Train labels: F02 (236), F03 (236) ... and 14 more
  NumEigenvectors: 16
  PLDANumDimensions: 16
  PLDANumIterations: 3
```

```
calibrate
  Calibration signals: 3774
  Calibration labels: F02 (236), F03 (236) ... and 14 more
```

Split the enrollment set.

```
[adsEnrollPart1,adsEnrollPart2] = splitEachLabel(adsEnroll,1,2);
```

To enroll speakers in the i-vector system, call `enroll`.

```
enroll(speakerVerification,adsEnrollPart1,adsEnrollPart1.Labels)
```

```
Extracting i-vectors ...done.
Enrolling i-vectors .......done.
Enrollment complete.
```

When you enroll speakers, the read-only `EnrolledLabels` property is updated with the enrolled labels and corresponding template i-vectors. The table also keeps track of the number of signals used to create the template i-vector. Generally, using more signals results in a better template.

```
speakerVerification.EnrolledLabels
```

```
ans=4×2 table
              ivector       NumSamples
            ____________    __________

    F01    {16×1 double}        1
    F05    {16×1 double}        1
    M01    {16×1 double}        1
    M05    {16×1 double}        1
```

Enroll the second part of the enrollment set and then view the enrolled labels table again. The i-vector templates and the number of samples are updated.

```
enroll(speakerVerification,adsEnrollPart2,adsEnrollPart2.Labels)
```

```
Extracting i-vectors ...done.
Enrolling i-vectors .......done.
Enrollment complete.
```

```
speakerVerification.EnrolledLabels
```

```
ans=4×2 table
              ivector       NumSamples
            ____________    __________

    F01    {16×1 double}        3
    F05    {16×1 double}        3
    M01    {16×1 double}        3
    M05    {16×1 double}        3
```

To evaluate the i-vector system and determine a decision threshold for speaker verification, call `detectionErrorTradeoff`.

```
[results, eerThreshold] = detectionErrorTradeoff(speakerVerification,adsDET,adsDET.Labels);
```

```
Extracting i-vectors ...done.
Scoring i-vector pairs ...done.
Detection error tradeoff evaluation complete.
```

The first output from `detectionErrorTradeoff` is a structure with two fields: CSS and PLDA. Each field contains a table. Each row of the table contains a possible decision threshold for speaker verification tasks, and the corresponding false alarm rate (FAR) and false rejection rate (FRR). The FAR and FRR are determined using the enrolled speaker labels and the data input to the `detectionErrorTradeoff` function.

`results`

```
results = struct with fields:
    PLDA: [1000×3 table]
     CSS: [1000×3 table]
```

`results.CSS`

```
ans=1000×3 table
    Threshold       FAR      FRR
    __________    _______    ___

    2.3259e-10          1     0
    2.3965e-10    0.99964     0
    2.4693e-10    0.99928     0
    2.5442e-10    0.99928     0
    2.6215e-10    0.99928     0
     2.701e-10    0.99928     0
     2.783e-10    0.99928     0
    2.8675e-10    0.99928     0
    2.9545e-10    0.99928     0
    3.0442e-10    0.99928     0
    3.1366e-10    0.99928     0
    3.2318e-10    0.99928     0
    3.3299e-10    0.99928     0
     3.431e-10    0.99928     0
    3.5352e-10    0.99928     0
    3.6425e-10    0.99892     0
      ⋮
```

`results.PLDA`

```
ans=1000×3 table
    Threshold       FAR      FRR
    __________    _______    ___

    3.2661e-40          1     0
    3.6177e-40    0.99964     0
    4.0072e-40    0.99964     0
    4.4387e-40    0.99964     0
    4.9166e-40    0.99964     0
    5.4459e-40    0.99964     0
    6.0322e-40    0.99964     0
    6.6817e-40    0.99964     0
    7.4011e-40    0.99964     0
     8.198e-40    0.99964     0
    9.0806e-40    0.99964     0
```

```
    1.0058e-39    0.99964     0
    1.1141e-39    0.99964     0
    1.2341e-39    0.99964     0
    1.3669e-39    0.99964     0
    1.5141e-39    0.99964     0
      ⋮
```

The second output from `detectionErrorTradeoff` is a structure with two fields: `CSS` and `PLDA`. The corresponding value is the decision threshold that results in the equal error rate (when FAR and FRR are equal).

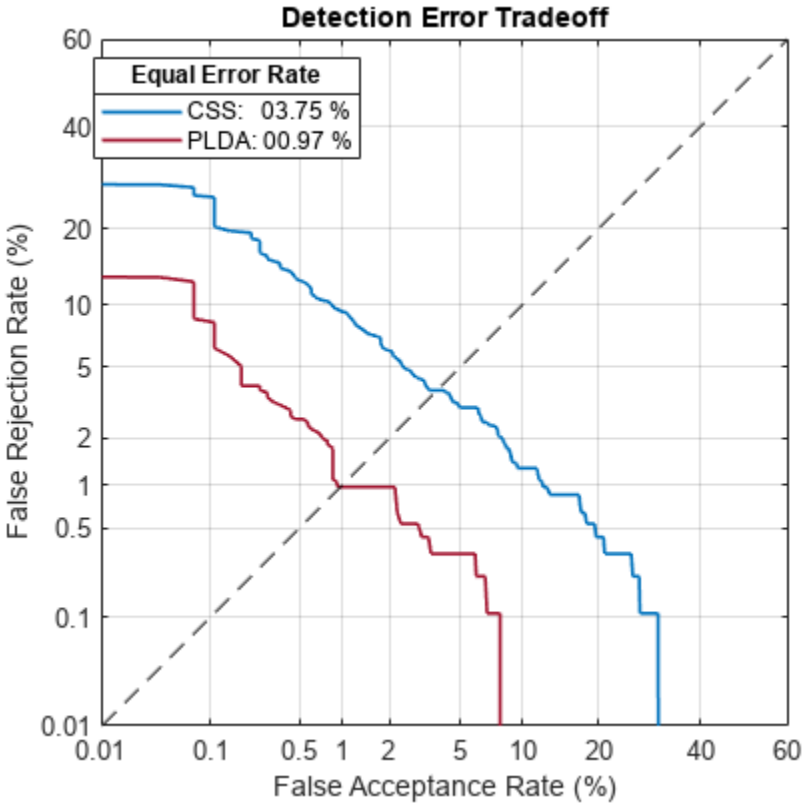
```
eerThreshold
```

```
eerThreshold = struct with fields:
    PLDA: 0.0398
     CSS: 0.9369
```

The first time you call `detectionErrorTradeoff`, you must provide data and corresponding labels to evaluate. Subsequently, you can get the same information, or a different analysis using the same underlying data, by calling `detectionErrorTradeoff` without data and labels.

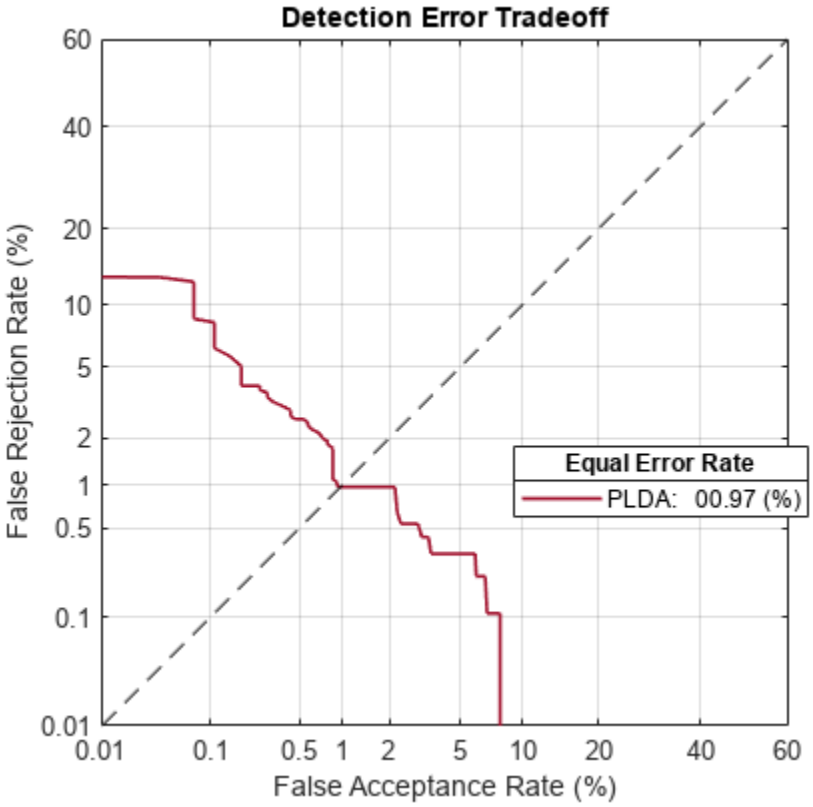
Call `detectionErrorTradeoff` a second time with no data arguments or output arguments to visualize the detection-error tradeoff.

```
detectionErrorTradeoff(speakerVerification)
```



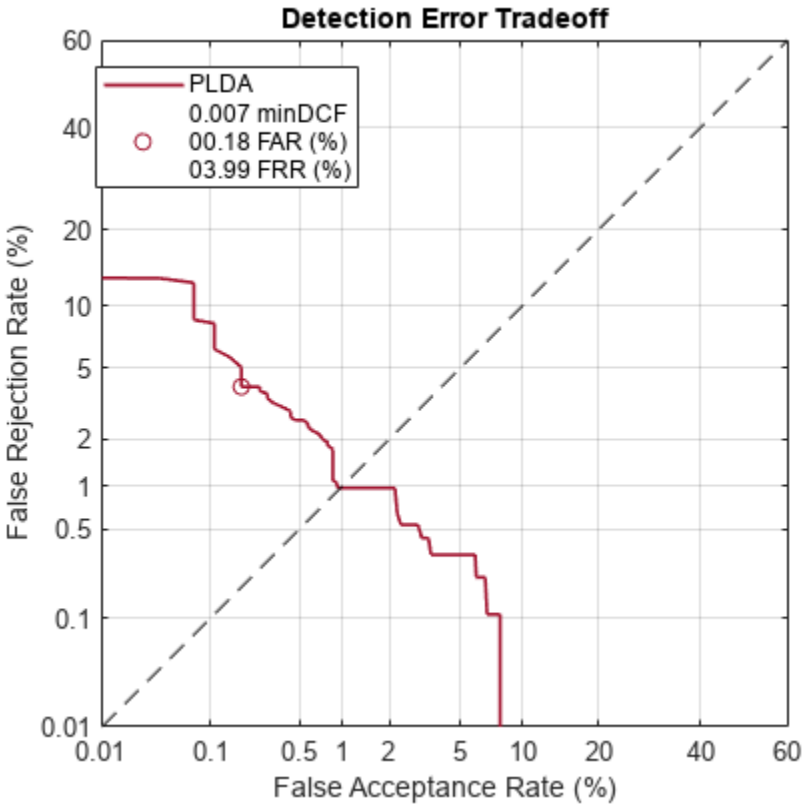
Call `detectionErrorTradeoff` again. This time, visualize only the detection-error tradeoff for the PLDA scorer.

```
detectionErrorTradeoff(speakerVerification,Scorer="plda")
```



Depending on your application, you may want to use a threshold that weights the error cost of a false alarm higher or lower than the error cost of a false rejection. You may also be using data that is not representative of the prior probability of the speaker being present. You can use the `minDCF` parameter to specify custom costs and prior probability. Call `detectionErrorTradeoff` again, this time specify the cost of a false rejection as 1, the cost of a false acceptance as 2, and the prior probability that a speaker is present as 0.1.

```
costFR = 1;
costFA = 2;
priorProb = 0.1;
detectionErrorTradeoff(speakerVerification,Scorer="plda",minDCF=[costFR,costFA,priorProb])
```



Call `detectionErrorTradeoff` again. This time, get the `minDCF` threshold for the PLDA scorer and the parameters of the detection cost function.

```
[~,minDCFThreshold] = detectionErrorTradeoff(speakerVerification,Scorer="plda",minDCF=[costFR,cos
```

```
minDCFThreshold = 0.4709
```

Test Speaker Verification System

Read a signal from the test set.

```
adsTest = shuffle(adsTest);
[audioIn,audioInfo] = read(adsTest);
knownSpeakerID = audioInfo.Label
```

```
knownSpeakerID = 1×1 cell array
    {'F01'}
```

To perform speaker verification, call `verify` with the audio signal and specify the speaker ID, a scorer, and a threshold for the scorer. The `verify` function returns a logical value indicating whether a speaker identity is accepted or rejected, and a score indicating the similarity of the input audio and the template i-vector corresponding to the enrolled label.

```
[tf,score] = verify(speakerVerification,audioIn,knownSpeakerID,"plda",eerThreshold.PLDA);
if tf
    fprintf('Success!\nSpeaker accepted.\nSimilarity score = %0.2f\n\n',score)
else
```

```
    fprinf('Failure!\nSpeaker rejected.\nSimilarity score = %0.2f\n\n',score)
end
```

```
Success!
Speaker accepted.
Similarity score = 1.00
```

Call speaker verification again. This time, specify an incorrect speaker ID.

```
possibleSpeakers = speakerVerification.EnrolledLabels.Properties.RowNames;
imposterIdx = find(~ismember(possibleSpeakers,knownSpeakerID));
imposter = possibleSpeakers(imposterIdx(randperm(numel(imposterIdx),1)))
```

```
imposter = 1×1 cell array
    {'M05'}
```

```
[tf,score] = verify(speakerVerification,audioIn,imposter,"plda",eerThreshold.PLDA);
if tf
    fprintf('Failure!\nSpeaker accepted.\nSimilarity score = %0.2f\n\n',score)
else
    fprintf('Success!\nSpeaker rejected.\nSimilarity score = %0.2f\n\n',score)
end
```

```
Success!
Speaker rejected.
Similarity score = 0.00
```

References

[1] Signal Processing and Speech Communication Laboratory. https://www.spsc.tugraz.at/databases-and-tools/ptdb-tug-pitch-tracking-database-from-graz-university-of-technology.html. Accessed 12 Dec. 2019.

Train Speaker Identification System

Use the Census Database (also known as AN4 Database) from the CMU Robust Speech Recognition Group [1] on page 4-291. The data set contains recordings of male and female subjects speaking words and numbers. The helper function in this example downloads the data set for you and converts the raw files to FLAC, and returns two `audioDatastore` objects containing the training set and test set. By default, the data set is reduced so that the example runs quickly. You can use the full data set by setting `ReduceDataset` to false.

```
[adsTrain,adsTest] = HelperAN4Download(ReduceDataset=true);
```

Split the test data set into enroll and test sets. Use two utterances for enrollment and the remaining for the test set. Generally, the more utterances you use for enrollment, the better the performance of the system. However, most practical applications are limited to a small set of enrollment utterances.

```
[adsEnroll,adsTest] = splitEachLabel(adsTest,2);
```

Inspect the distribution of speakers in the training, test, and enroll sets. The speakers in the training set do not overlap with the speakers in the test and enroll sets.

```
summary(adsTrain.Labels)
```

```
     fejs      13
     fmjd      13
     fsrb      13
     ftmj      13
     fwxs      12
     mcen      13
     mrcb      13
     msjm      13
     msjr      13
     msmn       9

summary(adsEnroll.Labels)

     fvap      2
     marh      2

summary(adsTest.Labels)

     fvap      11
     marh      11
```

Create an i-vector system that accepts feature input.

```
fs = 16e3;
iv = ivectorSystem(SampleRate=fs,InputType="features");
```

Create an `audioFeatureExtractor` object to extract the gammatone cepstral coefficients (GTCC), the delta GTCC, the delta-delta GTCC, and the pitch from 50 ms periodic Hann windows with 45 ms overlap.

```
afe = audioFeatureExtractor(gtcc=true,gtccDelta=true,gtccDeltaDelta=true,pitch=true,SampleRate=fs
afe.Window = hann(round(0.05*fs),"periodic");
afe.OverlapLength = round(0.045*fs);
afe

afe =
  audioFeatureExtractor with properties:

   Properties
                     Window: [800×1 double]
              OverlapLength: 720
                 SampleRate: 16000
                  FFTLength: []
    SpectralDescriptorInput: 'linearSpectrum'
        FeatureVectorLength: 40

   Enabled Features
     gtcc, gtccDelta, gtccDeltaDelta, pitch

   Disabled Features
     linearSpectrum, melSpectrum, barkSpectrum, erbSpectrum, mfcc, mfccDelta
     mfccDeltaDelta, spectralCentroid, spectralCrest, spectralDecrease, spectralEntropy, spectral
     spectralFlux, spectralKurtosis, spectralRolloffPoint, spectralSkewness, spectralSlope, spect
     harmonicRatio, zerocrossrate, shortTimeEnergy


   To extract a feature, set the corresponding property to true.
   For example, obj.mfcc = true, adds mfcc to the list of enabled features.
```

Create transformed datastores by adding feature extraction to the `read` function of `adsTrain` and `adsEnroll`.

```
trainLabels = adsTrain.Labels;
adsTrain = transform(adsTrain,@(x)extract(afe,x));
enrollLabels = adsEnroll.Labels;
adsEnroll = transform(adsEnroll,@(x)extract(afe,x));
```

Train both the extractor and classifier using the training set.

```
trainExtractor(iv,adsTrain, ...
    UBMNumComponents=64, ...
    UBMNumIterations=5, ...
    TVSRank=32, ...
    TVSNumIterations=3);
```

```
Calculating standardization factors ....done.
Training universal background model ........done.
Training total variability space ......done.
i-vector extractor training complete.
```

```
trainClassifier(iv,adsTrain,trainLabels, ...
    NumEigenvectors=16, ...
    ...
    PLDANumDimensions=16, ...
    PLDANumIterations=5);
```

```
Extracting i-vectors ...done.
Training projection matrix .....done.
Training PLDA model ........done.
i-vector classifier training complete.
```

To calibrate the system so that scores can be interpreted as a measure of confidence in a positive decision, use `calibrate`.

```
calibrate(iv,adsTrain,trainLabels)
```

```
Extracting i-vectors ...done.
Calibrating CSS scorer ...done.
Calibrating PLDA scorer ...done.
Calibration complete.
```

Enroll the speakers from the enrollment set.

```
enroll(iv,adsEnroll,enrollLabels)
```

```
Extracting i-vectors ...done.
Enrolling i-vectors .....done.
Enrollment complete.
```

Evaluate the file-level prediction accuracy on the test set.

```
numCorrect = 0;
reset(adsTest)
for index = 1:numel(adsTest.Files)
    features = extract(afe,read(adsTest));

    results = identify(iv,features);
```

```
    trueLabel = adsTest.Labels(index);
    predictedLabel = results.Label(1);
    isPredictionCorrect = trueLabel==predictedLabel;

    numCorrect = numCorrect + isPredictionCorrect;
end
display("File Accuracy: " + round(100*numCorrect/numel(adsTest.Files),2) + " (%)")
```

```
    "File Accuracy: 100 (%)"
```

References

[1] "CMU Sphinx Group - Audio Databases." http://www.speech.cs.cmu.edu/databases/an4/. Accessed 19 Dec. 2019.

Train Environmental Sound Classification System

Download and unzip the environment sound classification data set. This data set consists of recordings labeled as one of 10 different audio sound classes (ESC-10).

```
loc = matlab.internal.examples.downloadSupportFile("audio","ESC-10.zip");
unzip(loc,pwd)
```

Create an `audioDatastore` object to manage the data and split it into training and validation sets. Call `countEachLabel` to display the distribution of sound classes and the number of unique labels.

```
ads = audioDatastore(pwd,IncludeSubfolders=true,LabelSource="foldernames");
countEachLabel(ads)
```

```
ans=10×2 table
        Label         Count
    ______________    _____

    chainsaw           40
    clock_tick         40
    crackling_fire     40
    crying_baby        40
    dog                40
    helicopter         40
    rain               40
    rooster            38
    sea_waves          40
    sneezing           40
```

Listen to one of the files.

```
[audioIn,audioInfo] = read(ads);
fs = audioInfo.SampleRate;
sound(audioIn,fs)
audioInfo.Label
```

```
ans = categorical
     chainsaw
```

Split the datastore into training and test sets.

```
[adsTrain,adsTest] = splitEachLabel(ads,0.8);
```

Create an `audioFeatureExtractor` to extract all possible features from the audio.

```
afe = audioFeatureExtractor(SampleRate=fs, ...
    Window=hamming(round(0.03*fs),"periodic"), ...
    OverlapLength=round(0.02*fs));
params = info(afe,"all");
params = structfun(@(x)true,params,UniformOutput=false);
set(afe,params);
afe
```

```
afe =
  audioFeatureExtractor with properties:

   Properties
                     Window: [1323×1 double]
              OverlapLength: 882
                 SampleRate: 44100
                  FFTLength: []
    SpectralDescriptorInput: 'linearSpectrum'
        FeatureVectorLength: 862

   Enabled Features
     linearSpectrum, melSpectrum, barkSpectrum, erbSpectrum, mfcc, mfccDelta
     mfccDeltaDelta, gtcc, gtccDelta, gtccDeltaDelta, spectralCentroid, spectralCrest
     spectralDecrease, spectralEntropy, spectralFlatness, spectralFlux, spectralKurtosis, spectra
     spectralSkewness, spectralSlope, spectralSpread, pitch, harmonicRatio, zerocrossrate
     shortTimeEnergy

   Disabled Features
     none


   To extract a feature, set the corresponding property to true.
   For example, obj.mfcc = true, adds mfcc to the list of enabled features.
```

Create two directories in your current folder: train and test. Extract features from the training and the test data sets and write the features as MAT files to the respective directories. Pre-extracting features can save time when you want to evaluate different feature combinations or training configurations.

```
if ~isdir("train")
    mkdir("train")
    mkdir("test")

    outputType = ".mat";
    writeall(adsTrain,"train",WriteFcn=@(x,y,z)writeFeatures(x,y,z,afe))
    writeall(adsTest,"test",WriteFcn=@(x,y,z)writeFeatures(x,y,z,afe))
end
```

Create signal datastores to point to the audio features.

```
sdsTrain = signalDatastore("train",IncludeSubfolders=true);
sdsTest = signalDatastore("test",IncludeSubfolders=true);
```

Create label arrays that are in the same order as the `signalDatastore` files.

```
labelsTrain = categorical(extractBetween(sdsTrain.Files,"ESC-10"+filesep,filesep));
labelsTest = categorical(extractBetween(sdsTest.Files,"ESC-10"+filesep,filesep));
```

Create a transform datastore from the signal datastores to isolate and use only the desired features. You can use the output from `info` on the `audioFeatureExtractor` to map your chosen features to the index in the features matrix. You can experiment with the example by choosing different features.

```
featureIndices = info(afe)
```

```
featureIndices = struct with fields:
          linearSpectrum: [1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 2
             melSpectrum: [663 664 665 666 667 668 669 670 671 672 673 674 675 676 677 678 679 68
            barkSpectrum: [695 696 697 698 699 700 701 702 703 704 705 706 707 708 709 710 711 71
             erbSpectrum: [727 728 729 730 731 732 733 734 735 736 737 738 739 740 741 742 743 74
                    mfcc: [770 771 772 773 774 775 776 777 778 779 780 781 782]
               mfccDelta: [783 784 785 786 787 788 789 790 791 792 793 794 795]
          mfccDeltaDelta: [796 797 798 799 800 801 802 803 804 805 806 807 808]
                    gtcc: [809 810 811 812 813 814 815 816 817 818 819 820 821]
               gtccDelta: [822 823 824 825 826 827 828 829 830 831 832 833 834]
          gtccDeltaDelta: [835 836 837 838 839 840 841 842 843 844 845 846 847]
        spectralCentroid: 848
           spectralCrest: 849
        spectralDecrease: 850
         spectralEntropy: 851
        spectralFlatness: 852
            spectralFlux: 853
        spectralKurtosis: 854
    spectralRolloffPoint: 855
        spectralSkewness: 856
           spectralSlope: 857
          spectralSpread: 858
                   pitch: 859
           harmonicRatio: 860
           zerocrossrate: 861
         shortTimeEnergy: 862
```

```
idxToUse = [...
    featureIndices.harmonicRatio ...
    ,featureIndices.spectralRolloffPoint ...
    ,featureIndices.spectralFlux ...
    ,featureIndices.spectralSlope ...
    ];
tdsTrain = transform(sdsTrain,@(x)x(:,idxToUse));
tdsTest = transform(sdsTest,@(x)x(:,idxToUse));
```

Create an i-vector system that accepts feature input.

```
soundClassifier = ivectorSystem(InputType="features");
```

Train the extractor and classifier using the training set.

```
trainExtractor(soundClassifier,tdsTrain,UBMNumComponents=128,TVSRank=64);
```

```
Calculating standardization factors ....done.
Training universal background model .....done.
Training total variability space ......done.
i-vector extractor training complete.
```

```
trainClassifier(soundClassifier,tdsTrain,labelsTrain,NumEigenvectors=32,PLDANumIterations=0)
```

```
Extracting i-vectors ...done.
Training projection matrix .....done.
i-vector classifier training complete.
```

Enroll the labels from the training set to create i-vector templates for each of the environmental sounds.

```
enroll(soundClassifier,tdsTrain,labelsTrain)
```

```
Extracting i-vectors ...done.
Enrolling i-vectors .............done.
Enrollment complete.
```

Calibrate the i-vector system.

```
calibrate(soundClassifier,tdsTrain,labelsTrain)
```

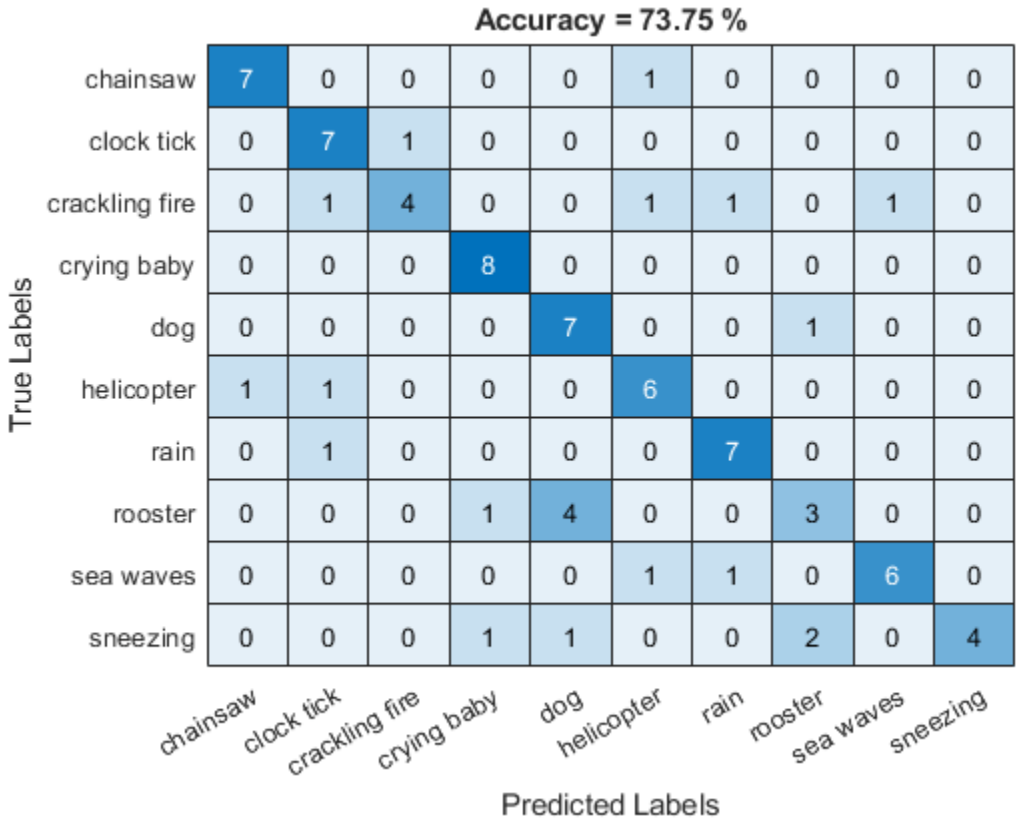
```
Extracting i-vectors ...done.
Calibrating CSS scorer ...done.
Calibration complete.
```

Use the `identify` function on the test set to return the system's inferred label.

```
inferredLabels = labelsTest;
inferredLabels(:) = inferredLabels(1);
for ii = 1:numel(labelsTest)
    features = read(tdsTest);
    tableOut = identify(soundClassifier,features,"css",NumCandidates=1);
    inferredLabels(ii) = tableOut.Label(1);
end
```

Create a confusion matrix to visualize performance on the test set.

```
uniqueLabels = unique(labelsTest);
cm = zeros(numel(uniqueLabels),numel(uniqueLabels));
for ii = 1:numel(uniqueLabels)
    for jj = 1:numel(uniqueLabels)
        cm(ii,jj) = sum((labelsTest==uniqueLabels(ii)) & (inferredLabels==uniqueLabels(jj)));
    end
end
labelStrings = replace(string(uniqueLabels),"_"," ");
heatmap(labelStrings,labelStrings,cm)
colorbar off
ylabel("True Labels")
xlabel("Predicted Labels")
accuracy = mean(inferredLabels==labelsTest);
title(sprintf("Accuracy = %0.2f %%",accuracy*100))
```



Release the i-vector system.

```
release(soundClassifier)
```

Supporting Functions

```
function writeFeatures(audioIn,info,~,afe)
    % Convet to single-precision
    audioIn = single(audioIn);

    % Extract features
    features = extract(afe,audioIn);

    % Replace the file extension of the suggested output name with MAT.
    filename = strrep(info.SuggestedOutputName,".wav",".mat");

    % Save the MFCC coefficients to the MAT file.
    save(filename,"features")
end
```

Train Acoustic Fault Recognition System

Download and unzip the air compressor data set [1] on page 4-298. This data set consists of recordings from air compressors in a healthy state or one of seven faulty states.

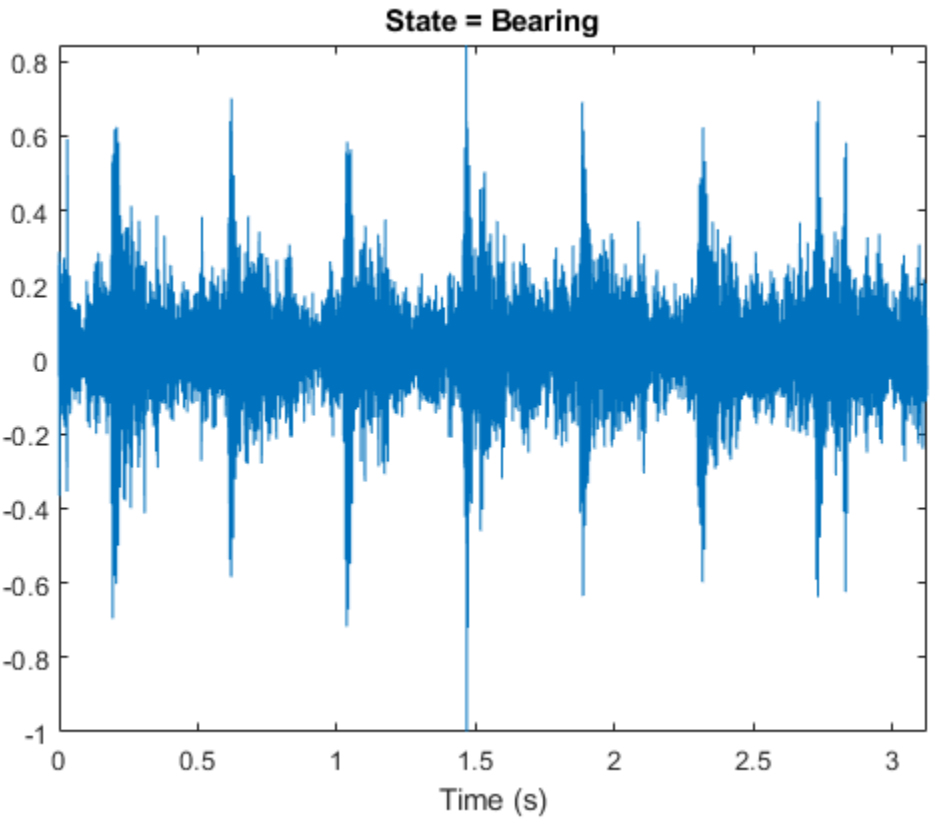
```
loc = matlab.internal.examples.downloadSupportFile("audio", ...
    "AirCompressorDataset/AirCompressorDataset.zip");
unzip(loc,pwd)
```

Create an `audioDatastore` object to manage the data and split it into training and validation sets.

```
ads = audioDatastore(pwd,IncludeSubfolders=true,LabelSource="foldernames");

[adsTrain,adsTest] = splitEachLabel(ads,0.8,0.2);
```

Read an audio file from the datastore and save the sample rate. Listen to the audio signal and plot the signal in the time domain.

```
[x,fileInfo] = read(adsTrain);
fs = fileInfo.SampleRate;

sound(x,fs)

t = (0:size(x,1)-1)/fs;
plot(t,x)
xlabel("Time (s)")
title("State = " + string(fileInfo.Label))
axis tight
```



Create an i-vector system with `DetectSpeech` set to `false`. Turn off the verbose behavior.

```
faultRecognizer = ivectorSystem(SampleRate=fs,DetectSpeech=false, ...
    Verbose=false)
```

```
faultRecognizer =
  ivectorSystem with properties:

         InputType: 'audio'
        SampleRate: 16000
      DetectSpeech: 0
           Verbose: 0
    EnrolledLabels: [0×2 table]
```

Train the i-vector extractor and the i-vector classifier using the training datastore.

```
trainExtractor(faultRecognizer,adsTrain, ...
    UBMNumComponents=80, ...
    UBMNumIterations=3, ...
    ...
    TVSRank=40, ...
    TVSNumIterations=3)

trainClassifier(faultRecognizer,adsTrain,adsTrain.Labels, ...
    NumEigenvectors=7, ...
    ...
    PLDANumDimensions=32, ...
    PLDANumIterations=5)
```

Calibrate the scores output by `faultRecognizer` so they can be interpreted as a measure of confidence in a positive decision. Turn the verbose behavior back on. Enroll all of the labels from the training set.

```
calibrate(faultRecognizer,adsTrain,adsTrain.Labels)

faultRecognizer.Verbose = true;

enroll(faultRecognizer,adsTrain,adsTrain.Labels)
```

```
Extracting i-vectors ...done.
Enrolling i-vectors ..........done.
Enrollment complete.
```

Use the read-only property `EnrolledLabels` to view the enrolled labels and the corresponding i-vector templates.

```
faultRecognizer.EnrolledLabels
```

```
ans=8×2 table
                 ivector       NumSamples
               ____________    __________

    Bearing    {7×1 double}       180
    Flywheel   {7×1 double}       180
    Healthy    {7×1 double}       180
    LIV        {7×1 double}       180
    LOV        {7×1 double}       180
    NRV        {7×1 double}       180
    Piston     {7×1 double}       180
    Riderbelt  {7×1 double}       180
```

Use the `identify` function with the PLDA scorer to predict the condition of machines in the test set. The `identify` function returns a table of possible labels sorted in descending order of confidence.

```
[audioIn,audioInfo] = read(adsTest);
trueLabel = audioInfo.Label
```

```
trueLabel = categorical
     Bearing
```

```
predictedLabels = identify(faultRecognizer,audioIn,"plda")
```

```
predictedLabels=8×2 table
      Label         Score
    _________    __________

    Bearing          0.99997
    Flywheel      2.265e-05
    Piston       8.6076e-08
    LIV          1.4237e-15
    NRV          4.5529e-16
    Riderbelt    3.7359e-16
    LOV          6.3025e-19
    Healthy      4.2094e-30
```

By default, the `identify` function returns all possible candidate labels and their corresponding scores. Use `NumCandidates` to reduce the number of candidates returned.

```
results = identify(faultRecognizer,audioIn,"plda",NumCandidates=3)
```

```
results=3×2 table
     Label         Score
    ________    __________

    Bearing          0.99997
    Flywheel      2.265e-05
    Piston       8.6076e-08
```

References

[1] Verma, Nishchal K., et al. “Intelligent Condition Based Monitoring Using Acoustic Signals for Air Compressors.” *IEEE Transactions on Reliability*, vol. 65, no. 1, Mar. 2016, pp. 291–309. *DOI.org (Crossref)*, doi:10.1109/TR.2015.2459684.

Train Speech Emotion Recognition System

Download the Berlin Database of Emotional Speech [1] on page 4-306. The database contains 535 utterances spoken by 10 actors intended to convey one of the following emotions: anger, boredom, disgust, anxiety/fear, happiness, sadness, or neutral. The emotions are text independent.

```
url = "http://emodb.bilderbar.info/download/download.zip";
downloadFolder = tempdir;
```

```
datasetFolder = fullfile(downloadFolder,"Emo-DB");

if ~exist(datasetFolder,"dir")
    disp("Downloading Emo-DB (40.5 MB) ...")
    unzip(url,datasetFolder)
end
```

Create an `audioDatastore` that points to the audio files.

```
ads = audioDatastore(fullfile(datasetFolder,"wav"));
```

The file names are codes indicating the speaker id, text spoken, emotion, and version. The website contains a key for interpreting the code and additional information about the speakers such as gender and age. Create a table with the variables `Speaker` and `Emotion`. Decode the file names into the table.

```
filepaths = ads.Files;
emotionCodes = cellfun(@(x)x(end-5),filepaths,"UniformOutput",false);
emotions = replace(emotionCodes,{'W','L','E','A','F','T','N'}, ...
    {'Anger','Boredom','Disgust','Anxiety','Happiness','Sadness','Neutral'});

speakerCodes = cellfun(@(x)x(end-10:end-9),filepaths,"UniformOutput",false);
labelTable = table(categorical(speakerCodes),categorical(emotions),VariableNames=["Speaker","Emo
summary(labelTable)
```

```
Variables:

    Speaker: 535×1 categorical

        Values:

            03       49
            08       58
            09       43
            10       38
            11       55
            12       35
            13       61
            14       69
            15       56
            16       71

    Emotion: 535×1 categorical

        Values:

            Anger          127
            Anxiety         69
            Boredom         81
            Disgust         46
            Happiness       71
            Neutral         79
            Sadness         62
```

`labelTable` is in the same order as the files in `audioDatastore`. Set the `Labels` property of the `audioDatastore` to `labelTable`.

```
ads.Labels = labelTable;
```

Read a signal from the datastore and listen to it. Display the speaker ID and emotion of the audio signal.

```
[audioIn,audioInfo] = read(ads);
fs = audioInfo.SampleRate;
sound(audioIn,fs)
audioInfo.Label
```

```
ans=1×2 table
    Speaker     Emotion
    _______    _________

      03       Happiness
```

Split the datastore into a training set and a test set. Assign two speakers to the test set and the remaining to the training set.

```
testSpeakerIdx = ads.Labels.Speaker=="12" | ads.Labels.Speaker=="13";
adsTrain = subset(ads,~testSpeakerIdx);
adsTest = subset(ads,testSpeakerIdx);
```

Read all the training and testing audio data into cell arrays. If your data can fit in memory, training is usually faster to input cell arrays to an i-vector system rather than datastores.

```
trainSet = readall(adsTrain);
trainLabels = adsTrain.Labels.Emotion;
testSet = readall(adsTest);
testLabels = adsTest.Labels.Emotion;
```

Create an i-vector system that does not apply speech detection. When `DetectSpeech` is set to `true` (the default), only regions of detected speech are used to train the i-vector system. When `DetectSpeech` is set to `false`, the entire input audio is used to train the i-vector system. The usefulness of applying speech detection depends on the data input to the system.

```
emotionRecognizer = ivectorSystem(SampleRate=fs,DetectSpeech=false)
```

```
emotionRecognizer =
  ivectorSystem with properties:

         InputType: 'audio'
        SampleRate: 16000
      DetectSpeech: 0
           Verbose: 1
    EnrolledLabels: [0×2 table]
```

Call `trainExtractor` using the training set.

```
rng default
trainExtractor(emotionRecognizer,trainSet, ...
    UBMNumComponents = 256 , ...
    UBMNumIterations = 5 , ...
    ...
```

```
    TVSRank = 128 , ...
    TVSNumIterations = 5 );
```

```
Calculating standardization factors .....done.
Training universal background model ........done.
Training total variability space ........done.
i-vector extractor training complete.
```

Copy the emotion recognition system for use later in the example.

```
sentimentRecognizer = copy(emotionRecognizer);
```

Call `trainClassifier` using the training set.

```
rng default
trainClassifier(emotionRecognizer,trainSet,trainLabels, ...
    NumEigenvectors = 32 , ...
    ...
    PLDANumDimensions = 16 , ...
    PLDANumIterations = 10 );
```

```
Extracting i-vectors ...done.
Training projection matrix .....done.
Training PLDA model .............done.
i-vector classifier training complete.
```

Call `calibrate` using the training set. In practice, the calibration set should be different than the training set.

```
calibrate(emotionRecognizer,trainSet,trainLabels)
```

```
Extracting i-vectors ...done.
Calibrating CSS scorer ...done.
Calibrating PLDA scorer ...done.
Calibration complete.
```

Enroll the training labels into the i-vector system.

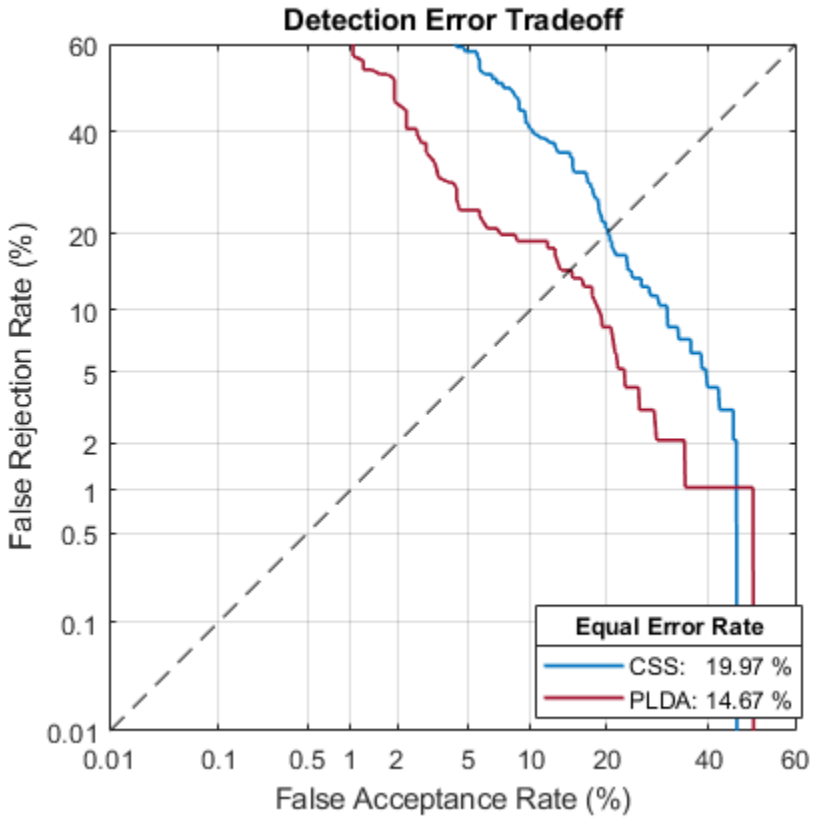
```
enroll(emotionRecognizer,trainSet,trainLabels)
```

```
Extracting i-vectors ...done.
Enrolling i-vectors ..........done.
Enrollment complete.
```

You can use `detectionErrorTradeoff` as a quick sanity check on the performance of a multilabel closed-set classification system. However, `detectionErrorTradeoff` provides information more suitable to open-set binary classification problems, for example, speaker verification tasks.

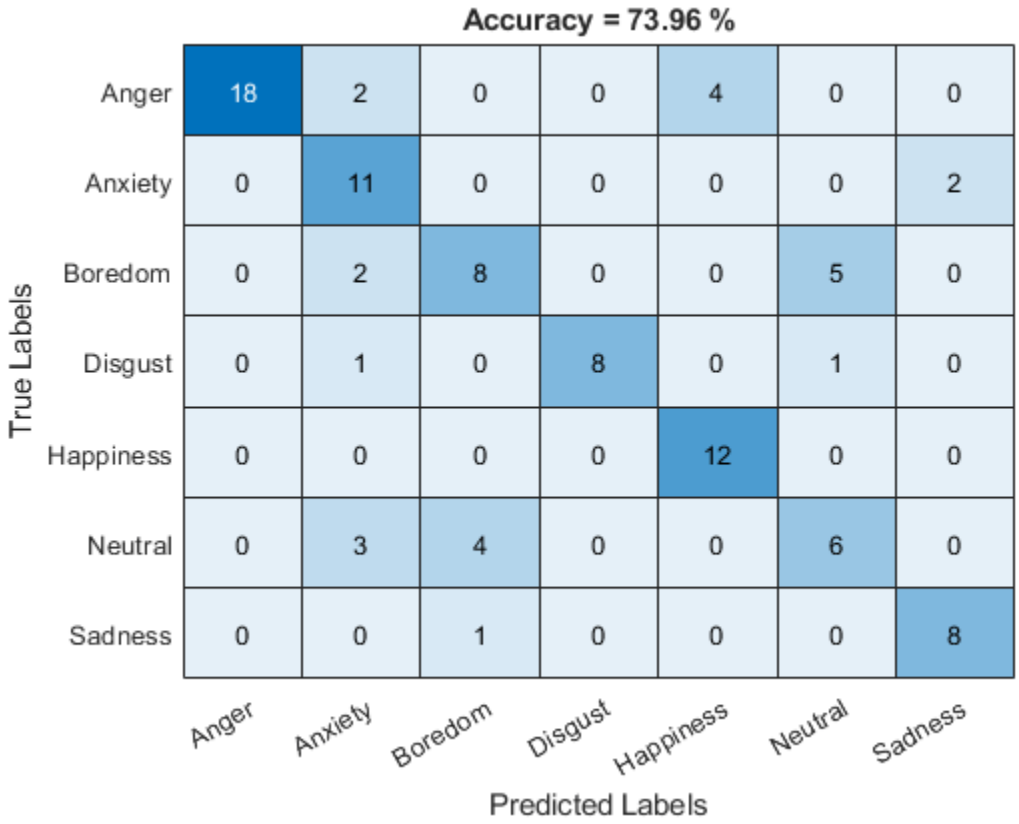
```
detectionErrorTradeoff(emotionRecognizer,testSet,testLabels)
```

```
Extracting i-vectors ...done.
Scoring i-vector pairs ...done.
Detection error tradeoff evaluation complete.
```



For a more detailed view of the i-vector system's performance in a multilabel closed set application, you can use the `identify` function and create a confusion matrix. The confusion matrix enables you to identify which emotions are misidentified and what they are misidentified as. Use the supporting function `plotConfusion` to display the results.

```
trueLabels = testLabels;
predictedLabels = trueLabels;

scorer = plda;
for ii = 1:numel(testSet)
    tableOut = identify(emotionRecognizer,testSet{ii},scorer);
    predictedLabels(ii) = tableOut.Label(1);
end

plotConfusion(trueLabels,predictedLabels)
```



Call `info` to inspect how `emotionRecognizer` was trained and evaluated.

```
info(emotionRecognizer)
```

```
i-vector system input
  Input feature vector length: 60
  Input data type: double

trainExtractor
  Train signals: 439
  UBMNumComponents: 256
  UBMNumIterations: 5
  TVSRank: 128
  TVSNumIterations: 5

trainClassifier
  Train signals: 439
  Train labels: Anger (103), Anxiety (56) ... and 5 more
  NumEigenvectors: 32
  PLDANumDimensions: 16
  PLDANumIterations: 10

calibrate
  Calibration signals: 439
  Calibration labels: Anger (103), Anxiety (56) ... and 5 more

detectionErrorTradeoff
```

```
  Evaluation signals: 96
  Evaluation labels: Anger (24), Anxiety (13) ... and 5 more
```

Next, modify the i-vector system to recognize emotions as positive, neutral, or negative. Update the labels to only include the categories negative, positive, and categorical.

```
trainLabelsSentiment = trainLabels;
trainLabelsSentiment(ismember(trainLabels,categorical(["Anger","Anxiety","Boredom","Sadness","Dis
trainLabelsSentiment(ismember(trainLabels,categorical("Happiness"))) = categorical("Postive");
trainLabelsSentiment = removecats(trainLabelsSentiment);

testLabelsSentiment = testLabels;
testLabelsSentiment(ismember(testLabels,categorical(["Anger","Anxiety","Boredom","Sadness","Disgu
testLabelsSentiment(ismember(testLabels,categorical("Happiness"))) = categorical("Postive");
testLabelsSentiment = removecats(testLabelsSentiment);
```

Train the i-vector system classifier using the updated labels. You do not need to retrain the extractor. Recalibrate the system.

```
rng default
trainClassifier(sentimentRecognizer,trainSet,trainLabelsSentiment, ...
    NumEigenvectors = 64 , ...
    ...
    PLDANumDimensions = 32 , ...
    PLDANumIterations = 10 );
```

```
Extracting i-vectors ...done.
Training projection matrix .....done.
Training PLDA model .............done.
i-vector classifier training complete.
```

```
calibrate(sentimentRecognizer,trainSet,trainLabels)
```

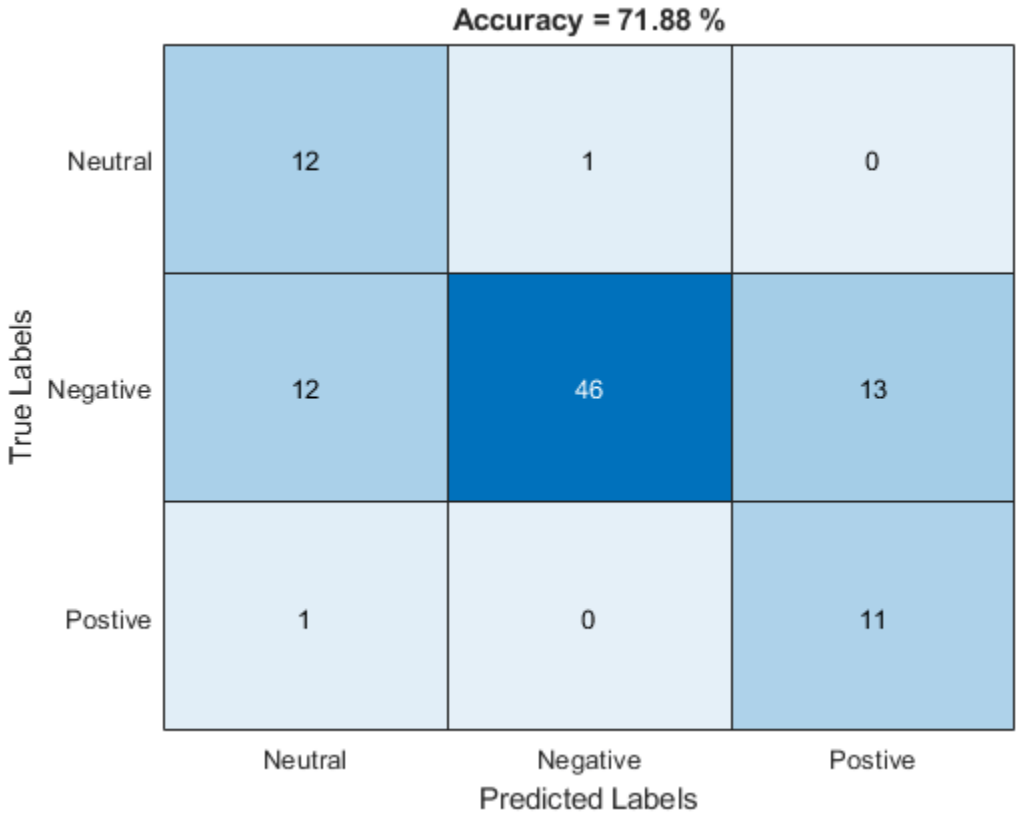
```
Extracting i-vectors ...done.
Calibrating CSS scorer ...done.
Calibrating PLDA scorer ...done.
Calibration complete.
```

Enroll the training labels into the system and then plot the confusion matrix for the test set.

```
enroll(sentimentRecognizer,trainSet,trainLabelsSentiment)
```

```
Extracting i-vectors ...done.
Enrolling i-vectors ......done.
Enrollment complete.
```

```
trueLabels = testLabelsSentiment;
predictedLabels = trueLabels;
scorer = plda ;
for ii = 1:numel(testSet)
    tableOut = identify(sentimentRecognizer,testSet{ii},scorer);
    predictedLabels(ii) = tableOut.Label(1);
end

plotConfusion(trueLabels,predictedLabels)
```



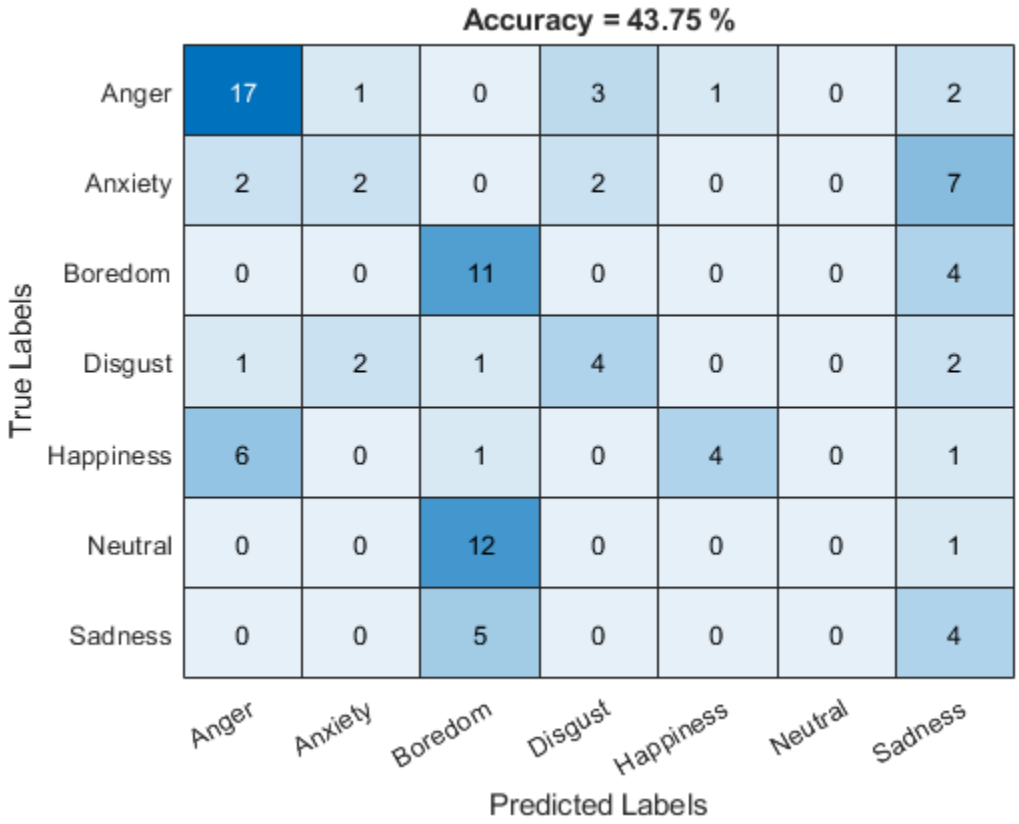
An i-vector system does not require the labels used to train the classifier to be equal to the enrolled labels.

Unenroll the sentiment labels from the system and then enroll the original emotion categories in the system. Analyze the system's classification performance.

```
unenroll(sentimentRecognizer)
enroll(sentimentRecognizer,trainSet,trainLabels)
```

```
Extracting i-vectors ...done.
Enrolling i-vectors ..........done.
Enrollment complete.
```

```
trueLabels = testLabels;
predictedLabels = trueLabels;
scorer = plda;
for ii = 1:numel(testSet)
    tableOut = identify(sentimentRecognizer,testSet{ii},scorer);
    predictedLabels(ii) = tableOut.Label(1);
end

plotConfusion(trueLabels,predictedLabels)
```



Supporting Functions

```
function plotConfusion(trueLabels,predictedLabels)
uniqueLabels = unique(trueLabels);
cm = zeros(numel(uniqueLabels),numel(uniqueLabels));
for ii = 1:numel(uniqueLabels)
    for jj = 1:numel(uniqueLabels)
        cm(ii,jj) = sum((trueLabels==uniqueLabels(ii)) & (predictedLabels==uniqueLabels(jj)));
    end
end

heatmap(uniqueLabels,uniqueLabels,cm)
colorbar off
ylabel('True Labels')
xlabel('Predicted Labels')
accuracy = mean(trueLabels==predictedLabels);
title(sprintf("Accuracy = %0.2f %%",accuracy*100))
end
```

References

[1] Burkhardt, F., A. Paeschke, M. Rolfes, W.F. Sendlmeier, and B. Weiss, "A Database of German Emotional Speech." In Proceedings Interspeech 2005. Lisbon, Portugal: International Speech Communication Association, 2005.

Train Word Recognition System

An i-vector system consists of a trainable front end that learns how to extract i-vectors based on unlabeled data, and a trainable backend that learns how to classify i-vectors based on labeled data. In this example, you apply an i-vector system to the task of word recognition. First, evaluate the accuracy of the i-vector system using the classifiers included in a traditional i-vector system: probabilistic linear discriminant analysis (PLDA) and cosine similarity scoring (CSS). Next, evaluate the accuracy of the system if you replace the classifier with bidirectional long short-term memory (BiLSTM) network or a K-nearest neighbors classifier.

Create Training and Validation Sets

Download the Free Spoken Digit Dataset (FSDD) [1] on page 4-312. FSDD consists of short audio files with spoken digits (0-9).

```
loc = matlab.internal.examples.downloadSupportFile("audio","FSDD.zip");
unzip(loc,pwd)
```

Create an `audioDatastore` to point to the recordings. Get the sample rate of the data set.

```
ads = audioDatastore(pwd,IncludeSubfolders=true);
[~,adsInfo] = read(ads);
fs = adsInfo.SampleRate;
```

The first element of the file names is the digit spoken in the file. Get the first element of the file names, convert them to categorical, and then set the `Labels` property of the `audioDatastore`.

```
[~,filenames] = cellfun(@(x)fileparts(x),ads.Files,UniformOutput=false);
ads.Labels = categorical(string(cellfun(@(x)x(1),filenames)));
```

To split the datastore into a development set and a validation set, use `splitEachLabel`. Allocate 80% of the data for development and the remaining 20% for validation.

```
[adsTrain,adsValidation] = splitEachLabel(ads,0.8);
```

Evaluate Traditional i-vector Backend Performance

Create an i-vector system that expects audio input at a sample rate of 8 kHz and does not perform speech detection.

```
wordRecognizer = ivectorSystem(DetectSpeech=false,SampleRate=fs)
```

```
wordRecognizer =
  ivectorSystem with properties:

         InputType: 'audio'
        SampleRate: 8000
      DetectSpeech: 0
           Verbose: 1
    EnrolledLabels: [0×2 table]
```

Train the i-vector extractor using the data in the training set.

```
trainExtractor(wordRecognizer,adsTrain, ...
    UBMNumComponents=64, ...
    UBMNumIterations=5, ...
    ...
```

```
    TVSRank=32, ...
    TVSNumIterations=5);
```

```
Calculating standardization factors ....done.
Training universal background model ........done.
Training total variability space ........done.
i-vector extractor training complete.
```

Train the i-vector classifier using the data in the training data set and the corresponding labels.

```
trainClassifier(wordRecognizer,adsTrain,adsTrain.Labels, ...
    NumEigenvectors=10, ...
    ...
    PLDANumDimensions=10, ...
    PLDANumIterations=5);
```

```
Extracting i-vectors ...done.
Training projection matrix .....done.
Training PLDA model ........done.
i-vector classifier training complete.
```

Calibrate the scores output by `wordRecognizer` so they can be interpreted as a measure of confidence in a positive decision. Enroll labels into the system using the entire training set.

```
calibrate(wordRecognizer,adsTrain,adsTrain.Labels)
```

```
Extracting i-vectors ...done.
Calibrating CSS scorer ...done.
Calibrating PLDA scorer ...done.
Calibration complete.
```

```
enroll(wordRecognizer,adsTrain,adsTrain.Labels)
```

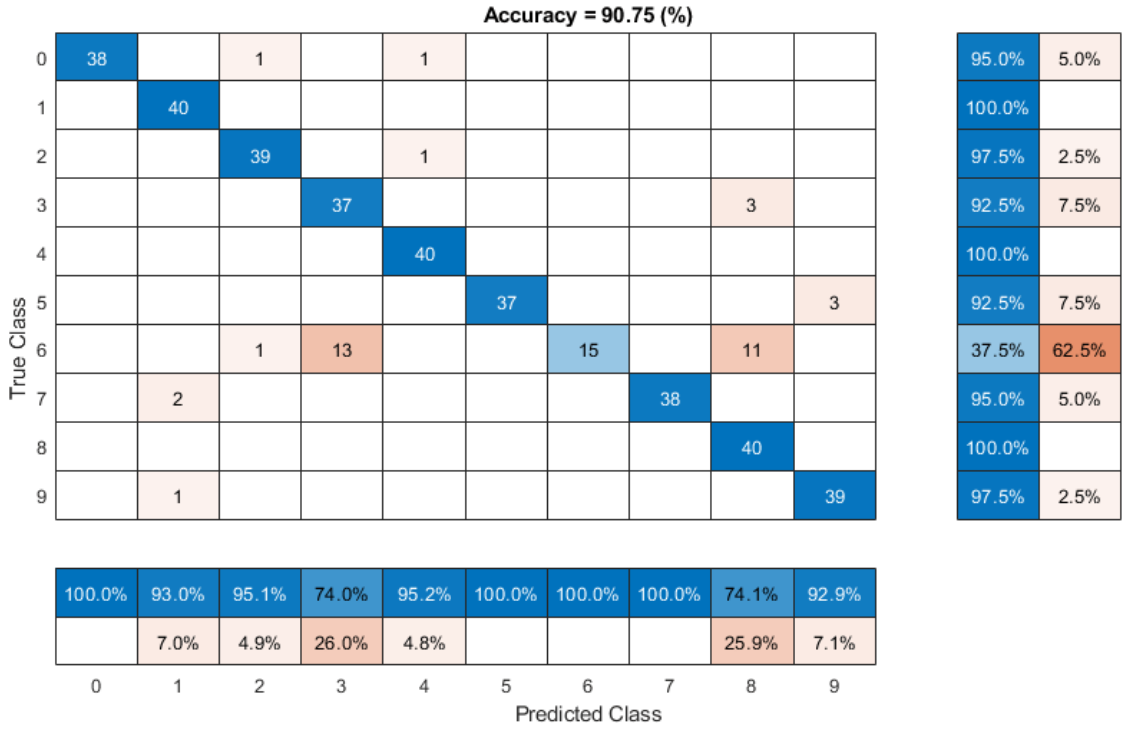
```
Extracting i-vectors ...done.
Enrolling i-vectors .............done.
Enrollment complete.
```

In a loop, read audio from the validation datastore, identify the most-likely word present according to the specified scorer, and save the prediction for analysis.

```
trueLabels = adsValidation.Labels;
predictedLabels = trueLabels;

reset(adsValidation)

scorer = plda;
for ii = 1:numel(trueLabels)

    audioIn = read(adsValidation);

    to = identify(wordRecognizer,audioIn,scorer);

    predictedLabels(ii) = to.Label(1);

end
```

Display a confusion chart of the i-vector system's performance on the validation set.

```
figure(Units="normalized",Position=[0.2 0.2 0.5 0.5])
confusionchart(trueLabels,predictedLabels, ...
    ColumnSummary="column-normalized", ...
    RowSummary="row-normalized", ...
    Title=sprintf('Accuracy = %0.2f (%%)',100*mean(predictedLabels==trueLabels)))
```



Evaluate Deep Learning Backend Performance

Next, train a fully-connected network using i-vectors as input.

```
ivectorsTrain = (ivector(wordRecognizer,adsTrain))';
ivectorsValidation = (ivector(wordRecognizer,adsValidation))';
```

Define a fully connected network.

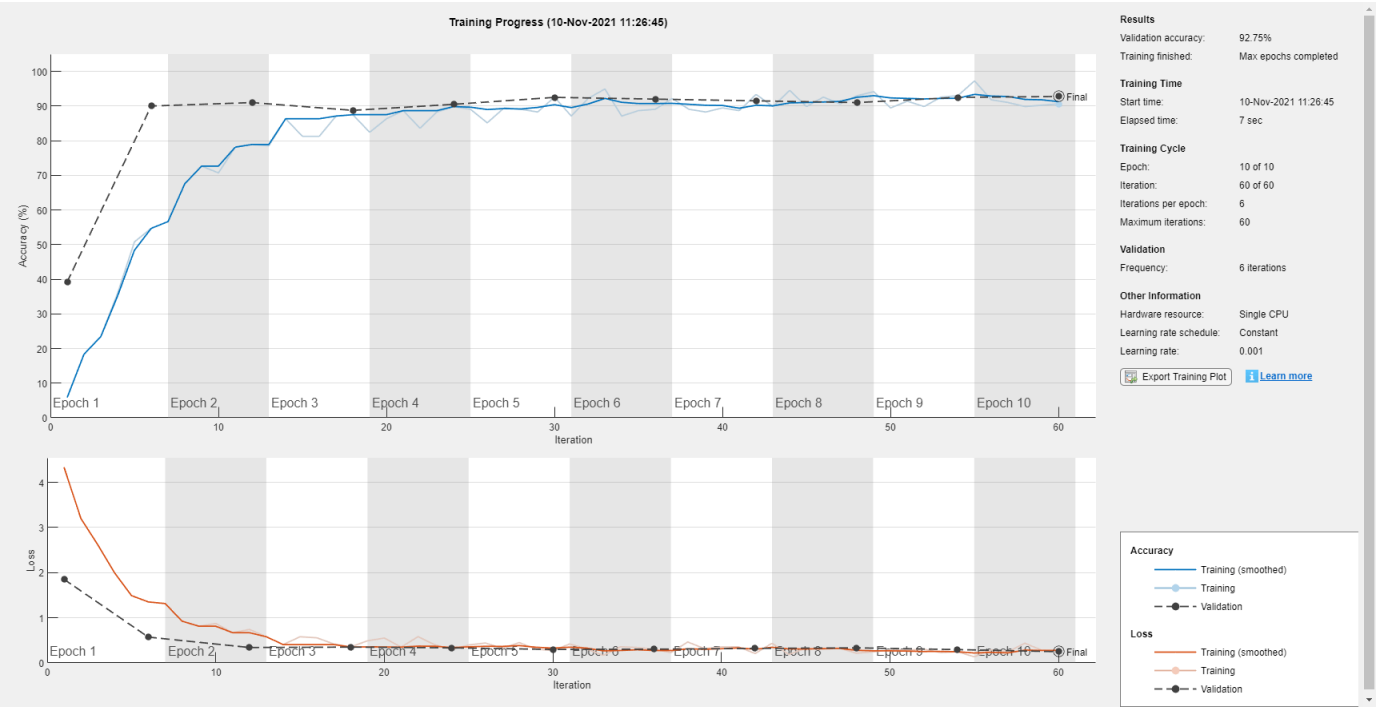
```
layers = [ ...
    featureInputLayer(size(ivectorsTrain,2),Normalization="none")
    fullyConnectedLayer(128)
    dropoutLayer(0.4)
    fullyConnectedLayer(256)
    dropoutLayer(0.4)
    fullyConnectedLayer(256)
    dropoutLayer(0.4)
    fullyConnectedLayer(128)
    dropoutLayer(0.4)
    fullyConnectedLayer(numel(unique(adsTrain.Labels)))
    softmaxLayer
    classificationLayer];
```

Define training parameters.

```
miniBatchSize = 256;
validationFrequency = floor(numel(adsTrain.Labels)/miniBatchSize);
options = trainingOptions("adam", ...
    MaxEpochs=10, ...
    MiniBatchSize=miniBatchSize, ...
    Plots="training-progress", ...
    Verbose=false, ...
    Shuffle="every-epoch", ...
    ValidationData={ivectorsValidation,adsValidation.Labels}, ...
    ValidationFrequency=validationFrequency);
```

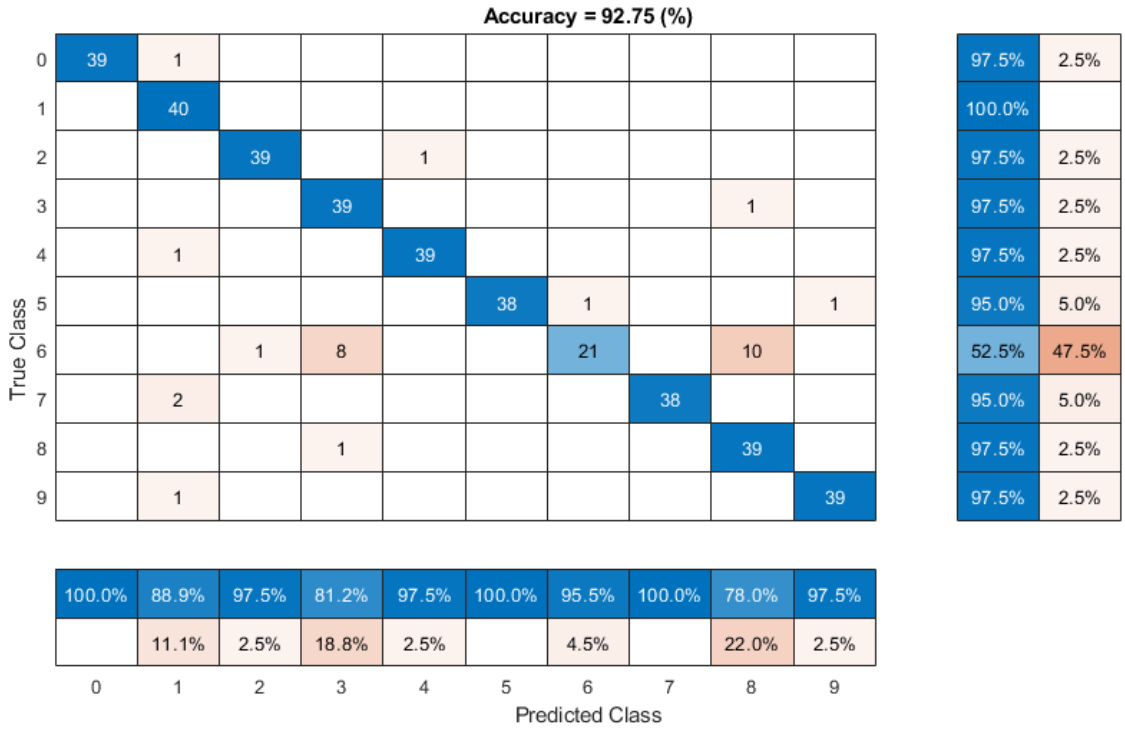
Train the network.

```
net = trainNetwork(ivectorsTrain,adsTrain.Labels,layers,options);
```



Evaluate the performance of the deep learning backend using a confusion chart.

```
predictedLabels = classify(net,ivectorsValidation);
trueLabels = adsValidation.Labels;

figure(Units="normalized",Position=[0.2 0.2 0.5 0.5])
confusionchart(trueLabels,predictedLabels, ...
    ColumnSummary="column-normalized", ...
    RowSummary="row-normalized", ...
    Title=sprintf('Accuracy = %0.2f (%%)',100*mean(predictedLabels==trueLabels)))
```



Evaluate KNN Backend Performance

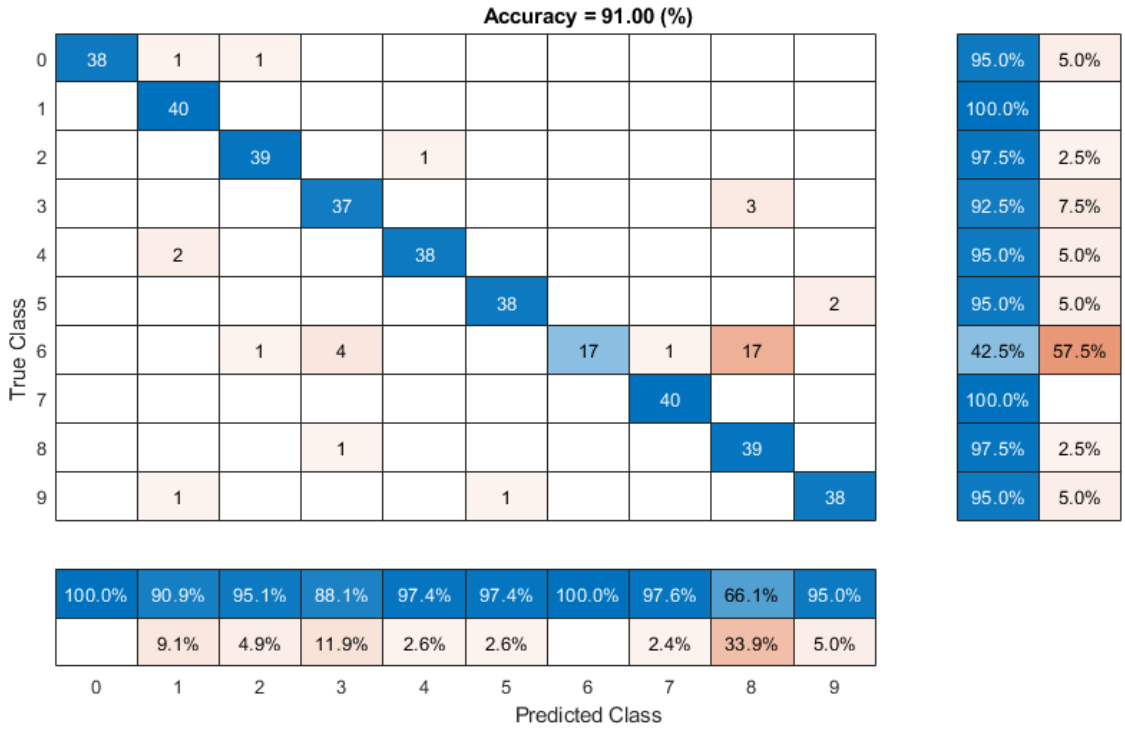
Train and evaluate i-vectors with a *k*-nearest neighbor (KNN) backend.

Use `fitcknn` to train a KNN model.

```
classificationKNN = fitcknn(...
    ivectorsTrain, ...
    adsTrain.Labels, ...
    Distance="Euclidean", ...
    Exponent=[], ...
    NumNeighbors=10, ...
    DistanceWeight="SquaredInverse", ...
    Standardize=true, ...
    ClassNames=unique(adsTrain.Labels));
```

Evaluate the KNN backend.

```
predictedLabels = predict(classificationKNN,ivectorsValidation);
trueLabels = adsValidation.Labels;

figure(Units="normalized",Position=[0.2 0.2 0.5 0.5])
confusionchart(trueLabels,predictedLabels, ...
    ColumnSummary="column-normalized", ...
    RowSummary="row-normalized", ...
    Title=sprintf('Accuracy = %0.2f (%%)',100*mean(predictedLabels==trueLabels)))
```



References

[1] Jakobovski. "Jakobovski/Free-Spoken-Digit-Dataset." GitHub, May 30, 2019. `https://github.com/Jakobovski/free-spoken-digit-dataset`.

Version History

Introduced in R2021a

References

[1] Reynolds, Douglas A., et al. “Speaker Verification Using Adapted Gaussian Mixture Models.” *Digital Signal Processing*, vol. 10, no. 1–3, Jan. 2000, pp. 19–41. *DOI.org (Crossref)*, doi:10.1006/dspr.1999.0361.

[2] Kenny, Patrick, et al. “Joint Factor Analysis Versus Eigenchannels in Speaker Recognition.” *IEEE Transactions on Audio, Speech and Language Processing*, vol. 15, no. 4, May 2007, pp. 1435–47. *DOI.org (Crossref)*, doi:10.1109/TASL.2006.881693.

[3] Kenny, P., et al. “A Study of Interspeaker Variability in Speaker Verification.” *IEEE Transactions on Audio, Speech, and Language Processing*, vol. 16, no. 5, July 2008, pp. 980–88. *DOI.org (Crossref)*, doi:10.1109/TASL.2008.925147.

[4] Dehak, Najim, et al. “Front-End Factor Analysis for Speaker Verification.” *IEEE Transactions on Audio, Speech, and Language Processing*, vol. 19, no. 4, May 2011, pp. 788–98. *DOI.org (Crossref)*, doi:10.1109/TASL.2010.2064307.

[5] Matejka, Pavel, Ondrej Glembek, Fabio Castaldo, M. J. Alam, Oldrich Plchot, Patrick Kenny, Lukas Burget, and Jan Cernocky. “Full-Covariance UBM and Heavy-Tailed PLDA in i-Vector Speaker Verification.” *2011 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 2011. https://doi.org/10.1109/icassp.2011.5947436.

[6] Snyder, David, et al. “X-Vectors: Robust DNN Embeddings for Speaker Recognition.” *2018 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, IEEE, 2018, pp. 5329–33. *DOI.org (Crossref)*, doi:10.1109/ICASSP.2018.8461375.

[7] Signal Processing and Speech Communication Laboratory. Accessed December 12, 2019. https://www.spsc.tugraz.at/databases-and-tools/ptdb-tug-pitch-tracking-database-from-graz-university-of-technology.html.

[8] Variani, Ehsan, et al. “Deep Neural Networks for Small Footprint Text-Dependent Speaker Verification.” *2014 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, IEEE, 2014, pp. 4052–56. *DOI.org (Crossref)*, doi:10.1109/ICASSP.2014.6854363.

[9] Dehak, Najim, Réda Dehak, James R. Glass, Douglas A. Reynolds and Patrick Kenny. “Cosine Similarity Scoring without Score Normalization Techniques.” *Odyssey* (2010).

[10] Verma, Pulkit, and Pradip K. Das. “I-Vectors in Speech Processing Applications: A Survey.” *International Journal of Speech Technology*, vol. 18, no. 4, Dec. 2015, pp. 529–46. *DOI.org (Crossref)*, doi:10.1007/s10772-015-9295-3.

[11] D. García-Romero and C. Espy-Wilson, “Analysis of I-vector Length Normalization in Speaker Recognition Systems.” *Interspeech*, 2011, pp. 249–252.

[12] Kenny, Patrick. "Bayesian Speaker Verification with Heavy-Tailed Priors". *Odyssey 2010 - The Speaker and Language Recognition Workshop*, Brno, Czech Republic, 2010.

[13] Sizov, Aleksandr, Kong Aik Lee, and Tomi Kinnunen. “Unifying Probabilistic Linear Discriminant Analysis Variants in Biometric Authentication.” *Lecture Notes in Computer Science Structural, Syntactic, and Statistical Pattern Recognition*, 2014, 464–75. https://doi.org/10.1007/978-3-662-44415-3_47.

[14] Rajan, Padmanabhan, Anton Afanasyev, Ville Hautamäki, and Tomi Kinnunen. “From Single to Multiple Enrollment I-Vectors: Practical PLDA Scoring Variants for Speaker Verification.” *Digital Signal Processing* 31 (August), 2014, pp. 93–101. https://doi.org/10.1016/j.dsp.2014.05.001.

See Also

`audioDatastore` | `audioFeatureExtractor` | `audioDataAugmenter` | `speakerRecognition`

Topics

“i-vector Score Normalization”
“i-vector Score Calibration”

trainExtractor

Train i-vector extractor

Syntax

```
trainExtractor(ivs,data)
trainExtractor(ivs,data,Name=Value)
```

Description

`trainExtractor(ivs,data)` trains the `ivectorSystem` object `ivs` to extract i-vectors using training data.

`trainExtractor(ivs,data,Name=Value)` specifies options using one or more name-value arguments. For example, `trainExtractor(ivs,data,UBMNumComponents=A)` specifies the maximum number of Gaussian components used to train the universal background model (UBM).

Examples

Train Speaker Verification System

Use the Pitch Tracking Database from Graz University of Technology (PTDB-TUG) [1] on page 4-324. The data set consists of 20 English native speakers reading 2342 phonetically rich sentences from the TIMIT corpus. Download and extract the data set.

```
downloadFolder = matlab.internal.examples.downloadSupportFile("audio","ptdb-tug.zip");
dataFolder = tempdir;
unzip(downloadFolder,dataFolder)
dataset = fullfile(dataFolder,"ptdb-tug");
```

Create an `audioDatastore` object that points to the data set. The data set was originally intended for use in pitch-tracking training and evaluation and includes laryngograph readings and baseline pitch decisions. Use only the original audio recordings.

```
ads = audioDatastore([fullfile(dataset,"SPEECH DATA","FEMALE","MIC"),fullfile(dataset,"SPEECH DA
                      IncludeSubfolders=true, ...
                      FileExtensions=".wav");
```

The file names contain the speaker IDs. Decode the file names to set the labels in the `audioDatastore` object.

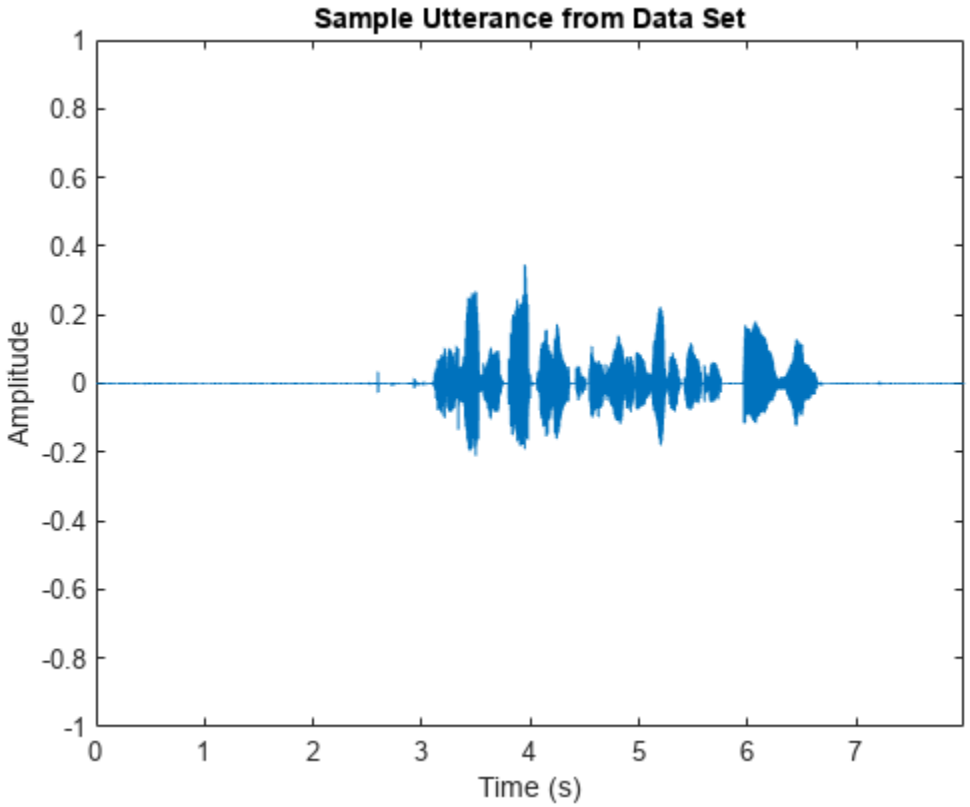
```
ads.Labels = extractBetween(ads.Files,"mic_","_");
countEachLabel(ads)
```

```
ans=20×2 table
    Label    Count
    _____    _____

     F01      236
     F02      236
```

```
F03      236
F04      236
F05      236
F06      236
F07      236
F08      234
F09      236
F10      236
M01      236
M02      236
M03      236
M04      236
M05      236
M06      236
  ⋮
```

Read an audio file from the data set, listen to it, and plot it.

```
[audioIn,audioInfo] = read(ads);
fs = audioInfo.SampleRate;

t = (0:size(audioIn,1)-1)/fs;
sound(audioIn,fs)
plot(t,audioIn)
xlabel("Time (s)")
ylabel("Amplitude")
axis([0 t(end) -1 1])
title("Sample Utterance from Data Set")
```



Separate the `audioDatastore` object into four: one for training, one for enrollment, one to evaluate the detection-error tradeoff, and one for testing. The training set contains 16 speakers. The enrollment, detection-error tradeoff, and test sets contain the other four speakers.

```
speakersToTest = categorical(["M01","M05","F01","F05"]);

adsTrain = subset(ads,~ismember(ads.Labels,speakersToTest));

ads = subset(ads,ismember(ads.Labels,speakersToTest));
[adsEnroll,adsTest,adsDET] = splitEachLabel(ads,3,1);
```

Display the label distributions of the `audioDatastore` objects.

```
countEachLabel(adsTrain)
```

```
ans=16×2 table
    Label    Count
    _____    _____

     F02      236
     F03      236
     F04      236
     F06      236
     F07      236
     F08      234
     F09      236
     F10      236
     M02      236
```

```
     M03      236
     M04      236
     M06      236
     M07      236
     M08      236
     M09      236
     M10      236
```

```
countEachLabel(adsEnroll)
```

```
ans=4×2 table
    Label    Count
    _____    _____

     F01       3
     F05       3
     M01       3
     M05       3
```

```
countEachLabel(adsTest)
```

```
ans=4×2 table
    Label    Count
    _____    _____

     F01       1
     F05       1
     M01       1
     M05       1
```

```
countEachLabel(adsDET)
```

```
ans=4×2 table
    Label    Count
    _____    _____

     F01      232
     F05      232
     M01      232
     M05      232
```

Create an i-vector system. By default, the i-vector system assumes the input to the system is mono audio signals.

```
speakerVerification = ivectorSystem(SampleRate=fs)
```

```
speakerVerification =
  ivectorSystem with properties:

         InputType: 'audio'
        SampleRate: 48000
      DetectSpeech: 1
           Verbose: 1
    EnrolledLabels: [0×2 table]
```

To train the extractor of the i-vector system, call `trainExtractor`. Specify the number of universal background model (UBM) components as 128 and the number of expectation maximization iterations as 5. Specify the total variability space (TVS) rank as 64 and the number of iterations as 3.

```
trainExtractor(speakerVerification,adsTrain, ...
    UBMNumComponents=128,UBMNumIterations=5, ...
    TVSRank=64,TVSNumIterations=3)
```

```
Calculating standardization factors ....done.
Training universal background model ........done.
Training total variability space ......done.
i-vector extractor training complete.
```

To train the classifier of the i-vector system, use `trainClassifier`. To reduce dimensionality of the i-vectors, specify the number of eigenvectors in the projection matrix as 16. Specify the number of dimensions in the probabilistic linear discriminant analysis (PLDA) model as 16, and the number of iterations as 3.

```
trainClassifier(speakerVerification,adsTrain,adsTrain.Labels, ...
    NumEigenvectors=16, ...
    PLDANumDimensions=16,PLDANumIterations=3)
```

```
Extracting i-vectors ...done.
Training projection matrix .....done.
Training PLDA model ......done.
i-vector classifier training complete.
```

To calibrate the system so that scores can be interpreted as a measure of confidence in a positive decision, use `calibrate`.

```
calibrate(speakerVerification,adsTrain,adsTrain.Labels)
```

```
Extracting i-vectors ...done.
Calibrating CSS scorer ...done.
Calibrating PLDA scorer ...done.
Calibration complete.
```

To inspect parameters used previously to train the i-vector system, use `info`.

```
info(speakerVerification)
```

```
i-vector system input
  Input feature vector length: 60
  Input data type: double

trainExtractor
  Train signals: 3774
  UBMNumComponents: 128
  UBMNumIterations: 5
  TVSRank: 64
  TVSNumIterations: 3

trainClassifier
  Train signals: 3774
  Train labels: F02 (236), F03 (236) ... and 14 more
  NumEigenvectors: 16
  PLDANumDimensions: 16
  PLDANumIterations: 3
```

```
calibrate
  Calibration signals: 3774
  Calibration labels: F02 (236), F03 (236) ... and 14 more
```

Split the enrollment set.

```
[adsEnrollPart1,adsEnrollPart2] = splitEachLabel(adsEnroll,1,2);
```

To enroll speakers in the i-vector system, call `enroll`.

```
enroll(speakerVerification,adsEnrollPart1,adsEnrollPart1.Labels)
```

```
Extracting i-vectors ...done.
Enrolling i-vectors .......done.
Enrollment complete.
```

When you enroll speakers, the read-only `EnrolledLabels` property is updated with the enrolled labels and corresponding template i-vectors. The table also keeps track of the number of signals used to create the template i-vector. Generally, using more signals results in a better template.

```
speakerVerification.EnrolledLabels
```

```
ans=4×2 table
              ivector       NumSamples
            ____________    __________

    F01    {16×1 double}        1
    F05    {16×1 double}        1
    M01    {16×1 double}        1
    M05    {16×1 double}        1
```

Enroll the second part of the enrollment set and then view the enrolled labels table again. The i-vector templates and the number of samples are updated.

```
enroll(speakerVerification,adsEnrollPart2,adsEnrollPart2.Labels)
```

```
Extracting i-vectors ...done.
Enrolling i-vectors .......done.
Enrollment complete.
```

```
speakerVerification.EnrolledLabels
```

```
ans=4×2 table
              ivector       NumSamples
            ____________    __________

    F01    {16×1 double}        3
    F05    {16×1 double}        3
    M01    {16×1 double}        3
    M05    {16×1 double}        3
```

To evaluate the i-vector system and determine a decision threshold for speaker verification, call `detectionErrorTradeoff`.

```
[results, eerThreshold] = detectionErrorTradeoff(speakerVerification,adsDET,adsDET.Labels);
```

```
Extracting i-vectors ...done.
Scoring i-vector pairs ...done.
Detection error tradeoff evaluation complete.
```

The first output from `detectionErrorTradeoff` is a structure with two fields: CSS and PLDA. Each field contains a table. Each row of the table contains a possible decision threshold for speaker verification tasks, and the corresponding false alarm rate (FAR) and false rejection rate (FRR). The FAR and FRR are determined using the enrolled speaker labels and the data input to the `detectionErrorTradeoff` function.

`results`

```
results = struct with fields:
    PLDA: [1000×3 table]
     CSS: [1000×3 table]
```

`results.CSS`

```
ans=1000×3 table
    Threshold       FAR      FRR
    __________    _______    ___

    2.3259e-10          1     0
    2.3965e-10    0.99964     0
    2.4693e-10    0.99928     0
    2.5442e-10    0.99928     0
    2.6215e-10    0.99928     0
     2.701e-10    0.99928     0
     2.783e-10    0.99928     0
    2.8675e-10    0.99928     0
    2.9545e-10    0.99928     0
    3.0442e-10    0.99928     0
    3.1366e-10    0.99928     0
    3.2318e-10    0.99928     0
    3.3299e-10    0.99928     0
     3.431e-10    0.99928     0
    3.5352e-10    0.99928     0
    3.6425e-10    0.99892     0
      ⋮
```

`results.PLDA`

```
ans=1000×3 table
    Threshold       FAR      FRR
    __________    _______    ___

    3.2661e-40          1     0
    3.6177e-40    0.99964     0
    4.0072e-40    0.99964     0
    4.4387e-40    0.99964     0
    4.9166e-40    0.99964     0
    5.4459e-40    0.99964     0
    6.0322e-40    0.99964     0
    6.6817e-40    0.99964     0
    7.4011e-40    0.99964     0
     8.198e-40    0.99964     0
    9.0806e-40    0.99964     0
```

```
    1.0058e-39    0.99964     0
    1.1141e-39    0.99964     0
    1.2341e-39    0.99964     0
    1.3669e-39    0.99964     0
    1.5141e-39    0.99964     0
      ⋮
```

The second output from `detectionErrorTradeoff` is a structure with two fields: `CSS` and `PLDA`. The corresponding value is the decision threshold that results in the equal error rate (when FAR and FRR are equal).

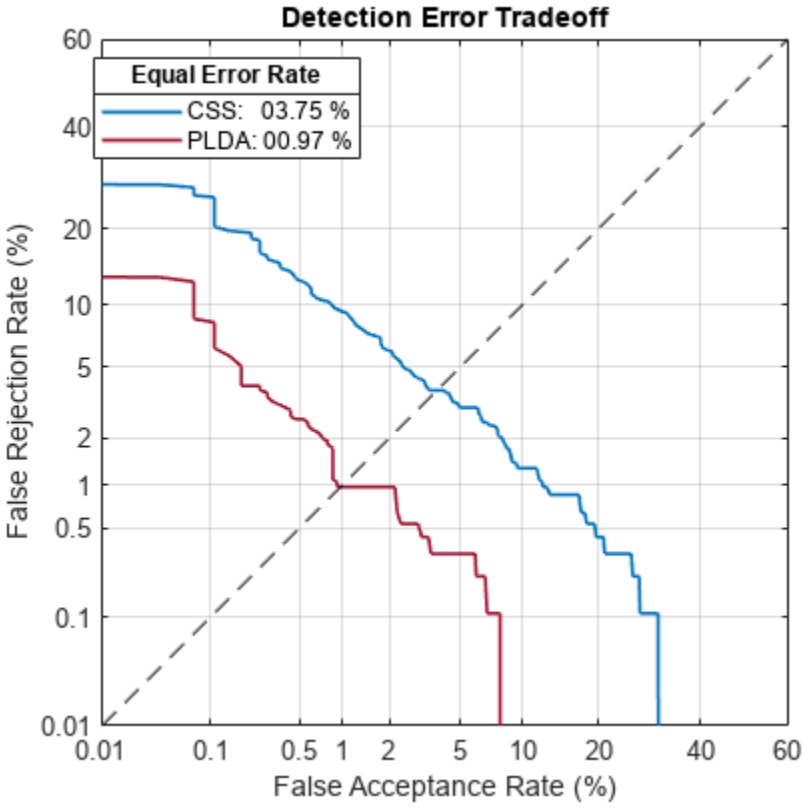
```
eerThreshold
```

```
eerThreshold = struct with fields:
    PLDA: 0.0398
     CSS: 0.9369
```

The first time you call `detectionErrorTradeoff`, you must provide data and corresponding labels to evaluate. Subsequently, you can get the same information, or a different analysis using the same underlying data, by calling `detectionErrorTradeoff` without data and labels.

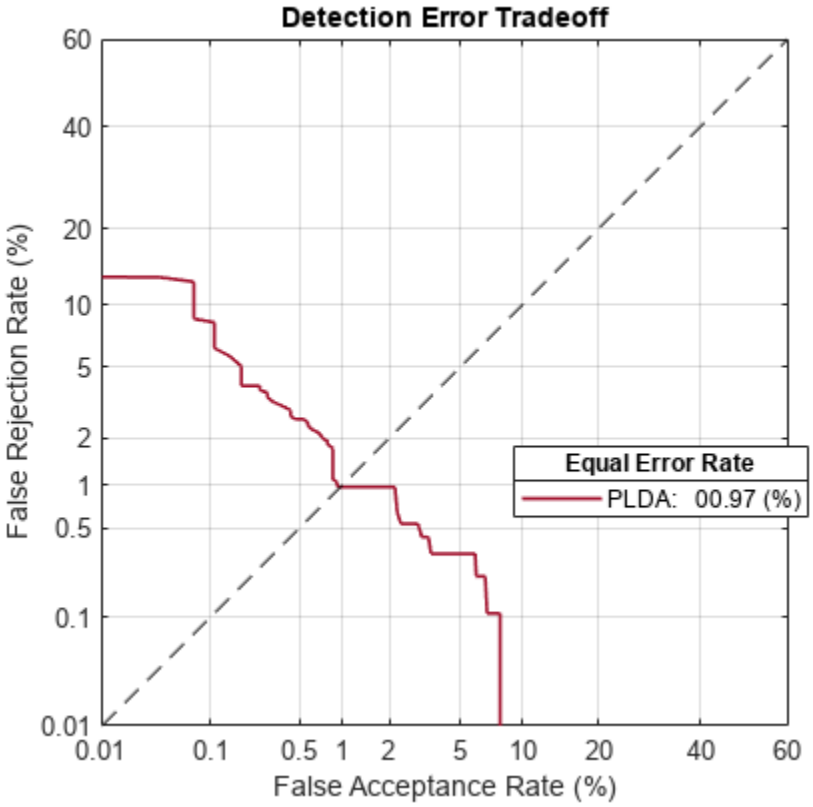
Call `detectionErrorTradeoff` a second time with no data arguments or output arguments to visualize the detection-error tradeoff.

```
detectionErrorTradeoff(speakerVerification)
```



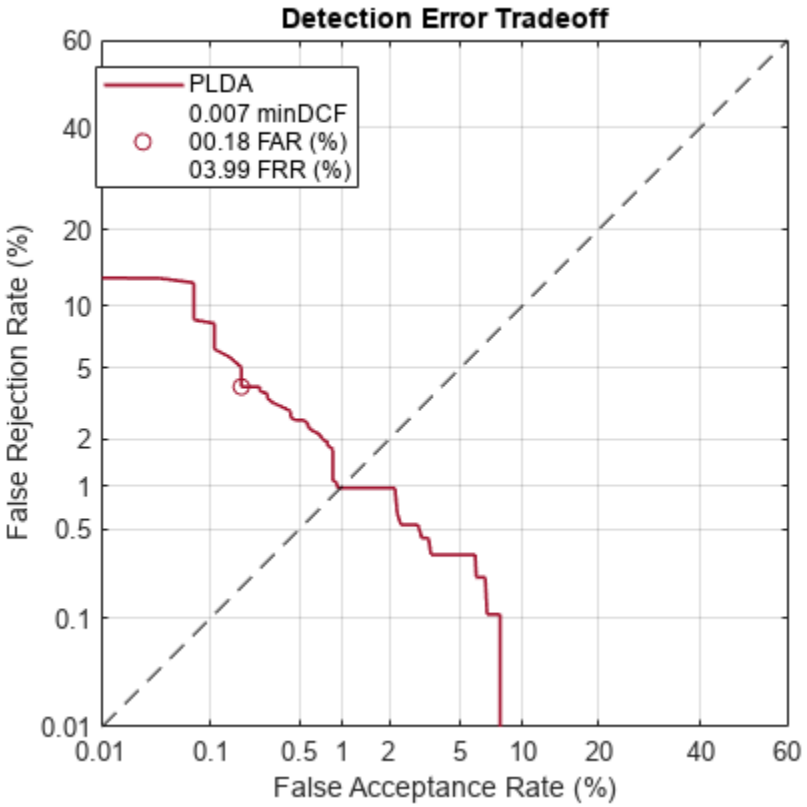
Call `detectionErrorTradeoff` again. This time, visualize only the detection-error tradeoff for the PLDA scorer.

```
detectionErrorTradeoff(speakerVerification,Scorer="plda")
```



Depending on your application, you may want to use a threshold that weights the error cost of a false alarm higher or lower than the error cost of a false rejection. You may also be using data that is not representative of the prior probability of the speaker being present. You can use the `minDCF` parameter to specify custom costs and prior probability. Call `detectionErrorTradeoff` again, this time specify the cost of a false rejection as 1, the cost of a false acceptance as 2, and the prior probability that a speaker is present as 0.1.

```
costFR = 1;
costFA = 2;
priorProb = 0.1;
detectionErrorTradeoff(speakerVerification,Scorer="plda",minDCF=[costFR,costFA,priorProb])
```



Call `detectionErrorTradeoff` again. This time, get the `minDCF` threshold for the PLDA scorer and the parameters of the detection cost function.

```
[~,minDCFThreshold] = detectionErrorTradeoff(speakerVerification,Scorer="plda",minDCF=[costFR,cos
```

```
minDCFThreshold = 0.4709
```

Test Speaker Verification System

Read a signal from the test set.

```
adsTest = shuffle(adsTest);
[audioIn,audioInfo] = read(adsTest);
knownSpeakerID = audioInfo.Label
```

```
knownSpeakerID = 1×1 cell array
    {'F01'}
```

To perform speaker verification, call `verify` with the audio signal and specify the speaker ID, a scorer, and a threshold for the scorer. The `verify` function returns a logical value indicating whether a speaker identity is accepted or rejected, and a score indicating the similarity of the input audio and the template i-vector corresponding to the enrolled label.

```
[tf,score] = verify(speakerVerification,audioIn,knownSpeakerID,"plda",eerThreshold.PLDA);
if tf
    fprintf('Success!\nSpeaker accepted.\nSimilarity score = %0.2f\n\n',score)
else
```

```
    fprinf('Failure!\nSpeaker rejected.\nSimilarity score = %0.2f\n\n',score)
end
```

```
Success!
Speaker accepted.
Similarity score = 1.00
```

Call speaker verification again. This time, specify an incorrect speaker ID.

```
possibleSpeakers = speakerVerification.EnrolledLabels.Properties.RowNames;
imposterIdx = find(~ismember(possibleSpeakers,knownSpeakerID));
imposter = possibleSpeakers(imposterIdx(randperm(numel(imposterIdx),1)))
```

```
imposter = 1×1 cell array
    {'M05'}
```

```
[tf,score] = verify(speakerVerification,audioIn,imposter,"plda",eerThreshold.PLDA);
if tf
    fprintf('Failure!\nSpeaker accepted.\nSimilarity score = %0.2f\n\n',score)
else
    fprintf('Success!\nSpeaker rejected.\nSimilarity score = %0.2f\n\n',score)
end
```

```
Success!
Speaker rejected.
Similarity score = 0.00
```

References

[1] Signal Processing and Speech Communication Laboratory. https://www.spsc.tugraz.at/databases-and-tools/ptdb-tug-pitch-tracking-database-from-graz-university-of-technology.html. Accessed 12 Dec. 2019.

Input Arguments

`ivs` — i-vector system

`ivectorSystem` object

i-vector system, specified as an object of type `ivectorSystem`.

`data` — Training data for i-vector system

cell array | `audioDatastore` | `signalDatastore` | `TransformedDatastore`

Training data for an i-vector system, specified as a cell array or as an `audioDatastore`, `signalDatastore`, or `TransformedDatastore` object.

- If `InputType` is set to `"audio"` when the i-vector system is created, specify `data` as one of these:
  - A cell array of single-channel audio signals, each specified as a column vector with underlying type `single` or `double`.
  - An `audioDatastore` object or a `signalDatastore` object that points to a data set of mono audio signals.
  - A `TransformedDatastore` with an underlying `audioDatastore` or `signalDatastore` that points to a data set of mono audio signals. The output from calls to `read` from the transform datastore must be mono audio signals with underlying data type `single` or `double`.

- If `InputType` is set to `"features"` when the i-vector system is created, specify `data` as one of these:
  - A cell array of matrices with underlying type `single` or `double`. The matrices must consist of audio features where the number of features (columns) is locked the first time `trainExtractor` is called and the number of hops (rows) is variable-sized. The number of features input in any subsequent calls to any of the object functions must be equal to the number of features used when calling `trainExtractor`.
  - A `TransformedDatastore` object with an underlying `audioDatastore` or `signalDatastore` whose `read` function has output as described in the previous bullet.
  - A `signalDatastore` object whose `read` function has output as described in the first bullet.

Data Types: `cell` | `audioDatastore` | `signalDatastore`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `trainExtractor(ivs,data,UBMNumIterations=B)`

`UBMNumComponents` — Maximum number of Gaussian components
32 (default) | positive integer

Maximum number of Gaussian components used to train the UBM, specified as a positive integer. The algorithm trims unused components determined during training to avoid numerical issues.

Example: `trainExtractor(ivs,data,UBMNumComponents=40)`

Data Types: `single` | `double`

`UBMNumIterations` — Number of expectation-maximization iterations
2 (default) | positive integer

Number of expectation-maximization iterations used to train the UBM, specified as a positive integer.

Example: `trainExtractor(ivs,data,UBMNumIterations=5)`

Data Types: `single` | `double`

`TVSRank` — Maximum rank of total variability space
16 (default) | positive integer

Maximum rank of the total variability space (TVS) trained to extract i-vectors, specified as a positive integer.

Example: `trainExtractor(ivs,data,TVSRank=24)`

Data Types: `single` | `double`

`TVSNumIterations` — Number of expectation-maximization iterations
3 (default) | positive integer

Number of expectation-maximization iterations used to train the TVS, specified as a positive integer.

Example: `trainExtractor(ivs,data,TVSNumIterations=5)`

Data Types: `single` | `double`

`ExecutionEnvironment` — Hardware resource for execution
`"auto"` (default) | `"cpu"` | `"gpu"` | `"multi-gpu"` | `"parallel"`

Hardware resource for execution, specified as one of these:

- `"auto"` — Use the GPU if it is available. Otherwise, use the CPU.
- `"cpu"` — Use the CPU.
- `"gpu"` — Use the GPU. This option requires Parallel Computing Toolbox.
- `"multi-gpu"` — Use multiple GPUs on one machine, using a local parallel pool based on your default cluster profile. If there is no current parallel pool, the software starts a parallel pool with pool size equal to the number of available GPUs. This option requires Parallel Computing Toolbox.
- `"parallel"` — Use a local or remote parallel pool based on your default cluster profile. If there is no current parallel pool, the software starts one using the default cluster profile. If the pool has access to GPUs, then only workers with a unique GPU perform training computation. If the pool does not have GPUs, then the training takes place on all available CPU workers. This option requires Parallel Computing Toolbox.

Data Types: `char` | `string`

`DispatchInBackground` — Option to use prefetch queuing
`false` (default) | `true`

Option to use prefetch queuing when reading from a datastore, specified as a logical value. This argument requires Parallel Computing Toolbox.

Data Types: `logical`

Version History

Introduced in R2021a

See Also

`trainClassifier` | `calibrate` | `enroll` | `unenroll` | `detectionErrorTradeoff` | `verify` | `identify` | `ivector` | `info` | `addInfoHeader` | `release` | `ivectorSystem` | `speakerRecognition`

trainClassifier

Train i-vector classifier

Syntax

```
trainClassifier(ivs,data,labels)
trainClassifier(ivs,data,labels,Name=Value)
```

Description

`trainClassifier(ivs,data,labels)` trains the `ivectorSystem` object `ivs` to classify i-vectors as `labels`.

`trainClassifier(ivs,data,labels,Name=Value)` specifies options using one or more name-value arguments. For example, `trainClassifier(ivs,data,labels,NumEigenvectors=A)` specifies the number of eigenvectors used to perform dimensionality reduction.

Examples

Train Speaker Verification System

Use the Pitch Tracking Database from Graz University of Technology (PTDB-TUG) [1] on page 4-337. The data set consists of 20 English native speakers reading 2342 phonetically rich sentences from the TIMIT corpus. Download and extract the data set.

```
downloadFolder = matlab.internal.examples.downloadSupportFile("audio","ptdb-tug.zip");
dataFolder = tempdir;
unzip(downloadFolder,dataFolder)
dataset = fullfile(dataFolder,"ptdb-tug");
```

Create an `audioDatastore` object that points to the data set. The data set was originally intended for use in pitch-tracking training and evaluation and includes laryngograph readings and baseline pitch decisions. Use only the original audio recordings.

```
ads = audioDatastore([fullfile(dataset,"SPEECH DATA","FEMALE","MIC"),fullfile(dataset,"SPEECH DA
                      IncludeSubfolders=true, ...
                      FileExtensions=".wav");
```

The file names contain the speaker IDs. Decode the file names to set the labels in the `audioDatastore` object.

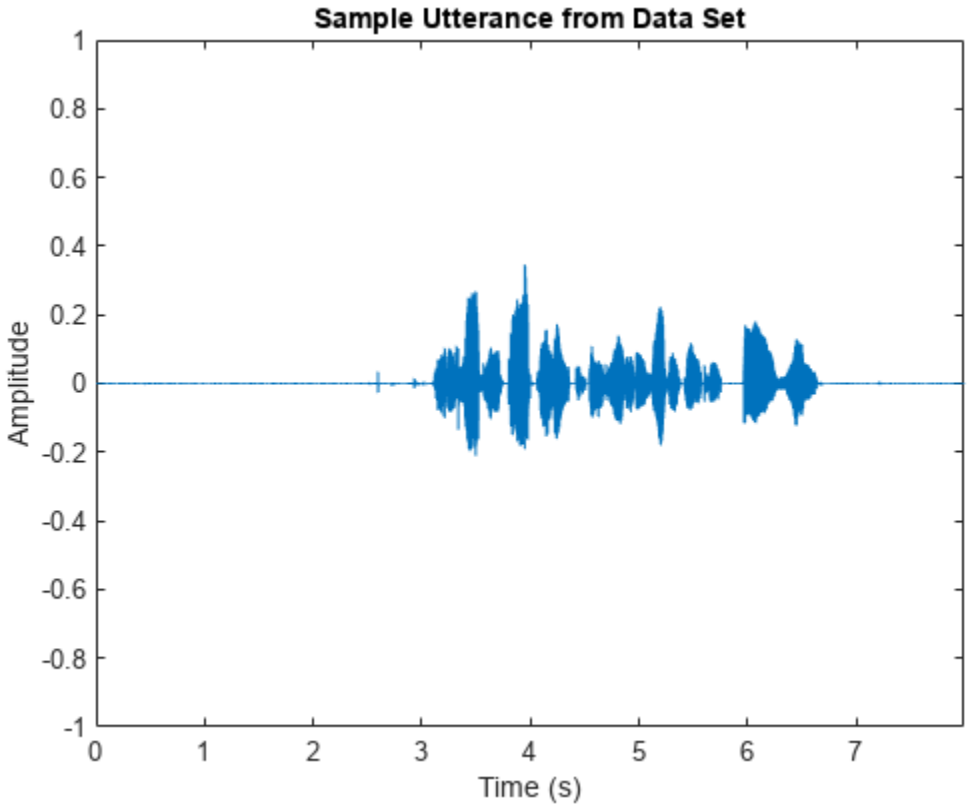
```
ads.Labels = extractBetween(ads.Files,"mic_","_");
countEachLabel(ads)
```

```
ans=20×2 table
    Label    Count
    _____    _____

     F01      236
     F02      236
```

```
F03      236
F04      236
F05      236
F06      236
F07      236
F08      234
F09      236
F10      236
M01      236
M02      236
M03      236
M04      236
M05      236
M06      236
  ⋮
```

Read an audio file from the data set, listen to it, and plot it.

```
[audioIn,audioInfo] = read(ads);
fs = audioInfo.SampleRate;

t = (0:size(audioIn,1)-1)/fs;
sound(audioIn,fs)
plot(t,audioIn)
xlabel("Time (s)")
ylabel("Amplitude")
axis([0 t(end) -1 1])
title("Sample Utterance from Data Set")
```



Separate the `audioDatastore` object into four: one for training, one for enrollment, one to evaluate the detection-error tradeoff, and one for testing. The training set contains 16 speakers. The enrollment, detection-error tradeoff, and test sets contain the other four speakers.

```
speakersToTest = categorical(["M01","M05","F01","F05"]);

adsTrain = subset(ads,~ismember(ads.Labels,speakersToTest));

ads = subset(ads,ismember(ads.Labels,speakersToTest));
[adsEnroll,adsTest,adsDET] = splitEachLabel(ads,3,1);
```

Display the label distributions of the `audioDatastore` objects.

```
countEachLabel(adsTrain)
```

```
ans=16×2 table
    Label    Count
    _____    _____

     F02      236
     F03      236
     F04      236
     F06      236
     F07      236
     F08      234
     F09      236
     F10      236
     M02      236
```

```
     M03      236
     M04      236
     M06      236
     M07      236
     M08      236
     M09      236
     M10      236
```

```
countEachLabel(adsEnroll)
```

```
ans=4×2 table
    Label    Count
    _____    _____

     F01       3
     F05       3
     M01       3
     M05       3
```

```
countEachLabel(adsTest)
```

```
ans=4×2 table
    Label    Count
    _____    _____

     F01       1
     F05       1
     M01       1
     M05       1
```

```
countEachLabel(adsDET)
```

```
ans=4×2 table
    Label    Count
    _____    _____

     F01      232
     F05      232
     M01      232
     M05      232
```

Create an i-vector system. By default, the i-vector system assumes the input to the system is mono audio signals.

```
speakerVerification = ivectorSystem(SampleRate=fs)
```

```
speakerVerification =
  ivectorSystem with properties:

         InputType: 'audio'
        SampleRate: 48000
      DetectSpeech: 1
           Verbose: 1
    EnrolledLabels: [0×2 table]
```

To train the extractor of the i-vector system, call `trainExtractor`. Specify the number of universal background model (UBM) components as 128 and the number of expectation maximization iterations as 5. Specify the total variability space (TVS) rank as 64 and the number of iterations as 3.

```
trainExtractor(speakerVerification,adsTrain, ...
    UBMNumComponents=128,UBMNumIterations=5, ...
    TVSRank=64,TVSNumIterations=3)
```

```
Calculating standardization factors ....done.
Training universal background model ........done.
Training total variability space ......done.
i-vector extractor training complete.
```

To train the classifier of the i-vector system, use `trainClassifier`. To reduce dimensionality of the i-vectors, specify the number of eigenvectors in the projection matrix as 16. Specify the number of dimensions in the probabilistic linear discriminant analysis (PLDA) model as 16, and the number of iterations as 3.

```
trainClassifier(speakerVerification,adsTrain,adsTrain.Labels, ...
    NumEigenvectors=16, ...
    PLDANumDimensions=16,PLDANumIterations=3)
```

```
Extracting i-vectors ...done.
Training projection matrix .....done.
Training PLDA model ......done.
i-vector classifier training complete.
```

To calibrate the system so that scores can be interpreted as a measure of confidence in a positive decision, use `calibrate`.

```
calibrate(speakerVerification,adsTrain,adsTrain.Labels)
```

```
Extracting i-vectors ...done.
Calibrating CSS scorer ...done.
Calibrating PLDA scorer ...done.
Calibration complete.
```

To inspect parameters used previously to train the i-vector system, use `info`.

```
info(speakerVerification)
```

```
i-vector system input
  Input feature vector length: 60
  Input data type: double

trainExtractor
  Train signals: 3774
  UBMNumComponents: 128
  UBMNumIterations: 5
  TVSRank: 64
  TVSNumIterations: 3

trainClassifier
  Train signals: 3774
  Train labels: F02 (236), F03 (236) ... and 14 more
  NumEigenvectors: 16
  PLDANumDimensions: 16
  PLDANumIterations: 3
```

```
calibrate
  Calibration signals: 3774
  Calibration labels: F02 (236), F03 (236) ... and 14 more
```

Split the enrollment set.

```
[adsEnrollPart1,adsEnrollPart2] = splitEachLabel(adsEnroll,1,2);
```

To enroll speakers in the i-vector system, call `enroll`.

```
enroll(speakerVerification,adsEnrollPart1,adsEnrollPart1.Labels)
```

```
Extracting i-vectors ...done.
Enrolling i-vectors .......done.
Enrollment complete.
```

When you enroll speakers, the read-only `EnrolledLabels` property is updated with the enrolled labels and corresponding template i-vectors. The table also keeps track of the number of signals used to create the template i-vector. Generally, using more signals results in a better template.

```
speakerVerification.EnrolledLabels
```

```
ans=4×2 table
              ivector       NumSamples
            ____________    __________

    F01    {16×1 double}        1
    F05    {16×1 double}        1
    M01    {16×1 double}        1
    M05    {16×1 double}        1
```

Enroll the second part of the enrollment set and then view the enrolled labels table again. The i-vector templates and the number of samples are updated.

```
enroll(speakerVerification,adsEnrollPart2,adsEnrollPart2.Labels)
```

```
Extracting i-vectors ...done.
Enrolling i-vectors .......done.
Enrollment complete.
```

```
speakerVerification.EnrolledLabels
```

```
ans=4×2 table
              ivector       NumSamples
            ____________    __________

    F01    {16×1 double}        3
    F05    {16×1 double}        3
    M01    {16×1 double}        3
    M05    {16×1 double}        3
```

To evaluate the i-vector system and determine a decision threshold for speaker verification, call `detectionErrorTradeoff`.

```
[results, eerThreshold] = detectionErrorTradeoff(speakerVerification,adsDET,adsDET.Labels);
```

```
Extracting i-vectors ...done.
Scoring i-vector pairs ...done.
Detection error tradeoff evaluation complete.
```

The first output from `detectionErrorTradeoff` is a structure with two fields: CSS and PLDA. Each field contains a table. Each row of the table contains a possible decision threshold for speaker verification tasks, and the corresponding false alarm rate (FAR) and false rejection rate (FRR). The FAR and FRR are determined using the enrolled speaker labels and the data input to the `detectionErrorTradeoff` function.

`results`

```
results = struct with fields:
    PLDA: [1000×3 table]
     CSS: [1000×3 table]
```

`results.CSS`

```
ans=1000×3 table
    Threshold       FAR      FRR
    __________    _______    ___

    2.3259e-10          1     0
    2.3965e-10    0.99964     0
    2.4693e-10    0.99928     0
    2.5442e-10    0.99928     0
    2.6215e-10    0.99928     0
     2.701e-10    0.99928     0
     2.783e-10    0.99928     0
    2.8675e-10    0.99928     0
    2.9545e-10    0.99928     0
    3.0442e-10    0.99928     0
    3.1366e-10    0.99928     0
    3.2318e-10    0.99928     0
    3.3299e-10    0.99928     0
     3.431e-10    0.99928     0
    3.5352e-10    0.99928     0
    3.6425e-10    0.99892     0
      ⋮
```

`results.PLDA`

```
ans=1000×3 table
    Threshold       FAR      FRR
    __________    _______    ___

    3.2661e-40          1     0
    3.6177e-40    0.99964     0
    4.0072e-40    0.99964     0
    4.4387e-40    0.99964     0
    4.9166e-40    0.99964     0
    5.4459e-40    0.99964     0
    6.0322e-40    0.99964     0
    6.6817e-40    0.99964     0
    7.4011e-40    0.99964     0
     8.198e-40    0.99964     0
    9.0806e-40    0.99964     0
```

```
    1.0058e-39    0.99964     0
    1.1141e-39    0.99964     0
    1.2341e-39    0.99964     0
    1.3669e-39    0.99964     0
    1.5141e-39    0.99964     0
      ⋮
```

The second output from `detectionErrorTradeoff` is a structure with two fields: `CSS` and `PLDA`. The corresponding value is the decision threshold that results in the equal error rate (when FAR and FRR are equal).

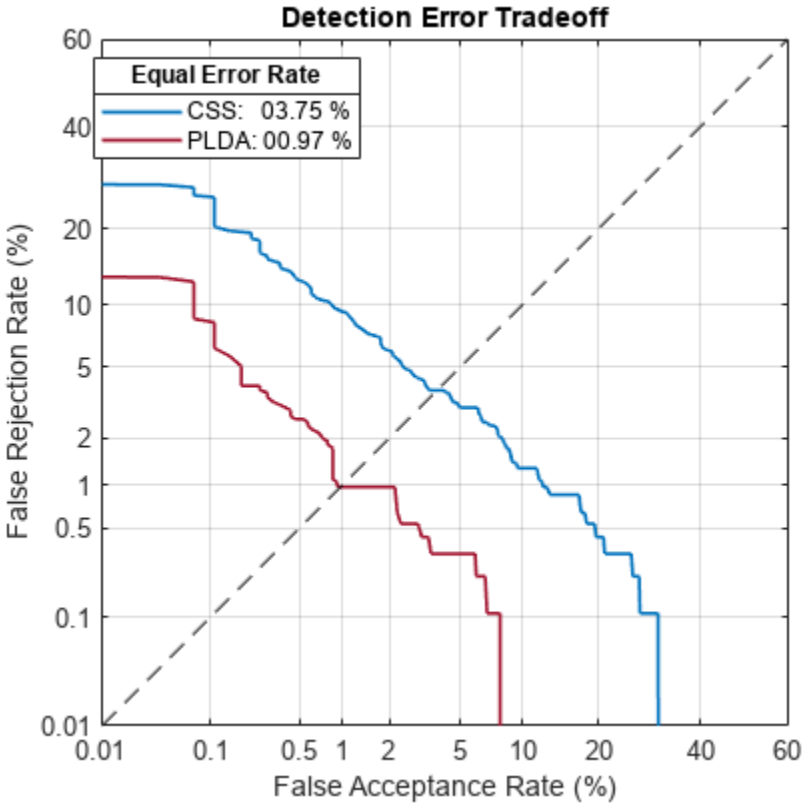
```
eerThreshold
```

```
eerThreshold = struct with fields:
    PLDA: 0.0398
     CSS: 0.9369
```

The first time you call `detectionErrorTradeoff`, you must provide data and corresponding labels to evaluate. Subsequently, you can get the same information, or a different analysis using the same underlying data, by calling `detectionErrorTradeoff` without data and labels.

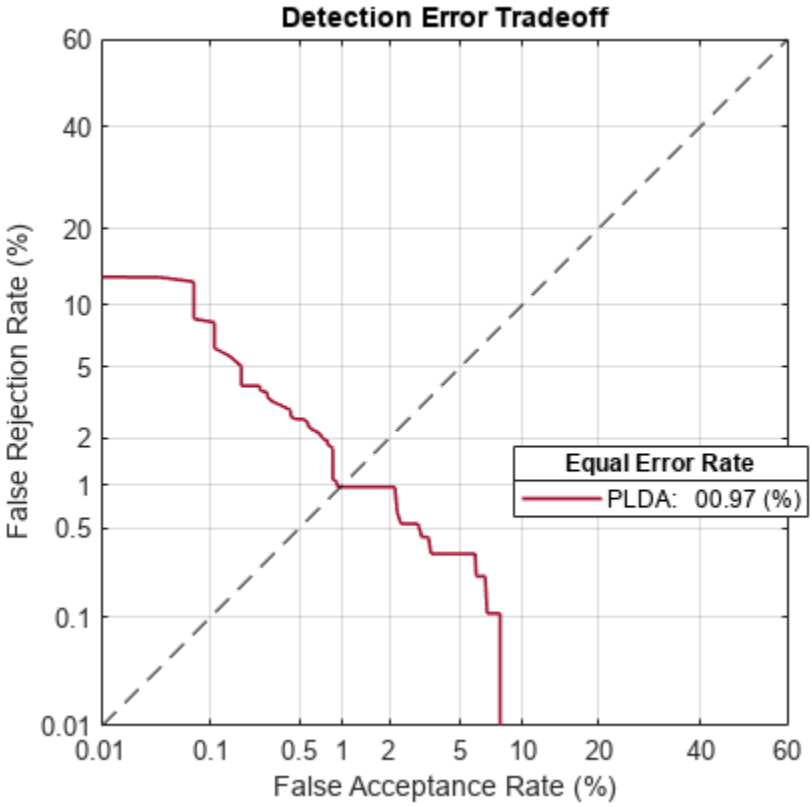
Call `detectionErrorTradeoff` a second time with no data arguments or output arguments to visualize the detection-error tradeoff.

```
detectionErrorTradeoff(speakerVerification)
```



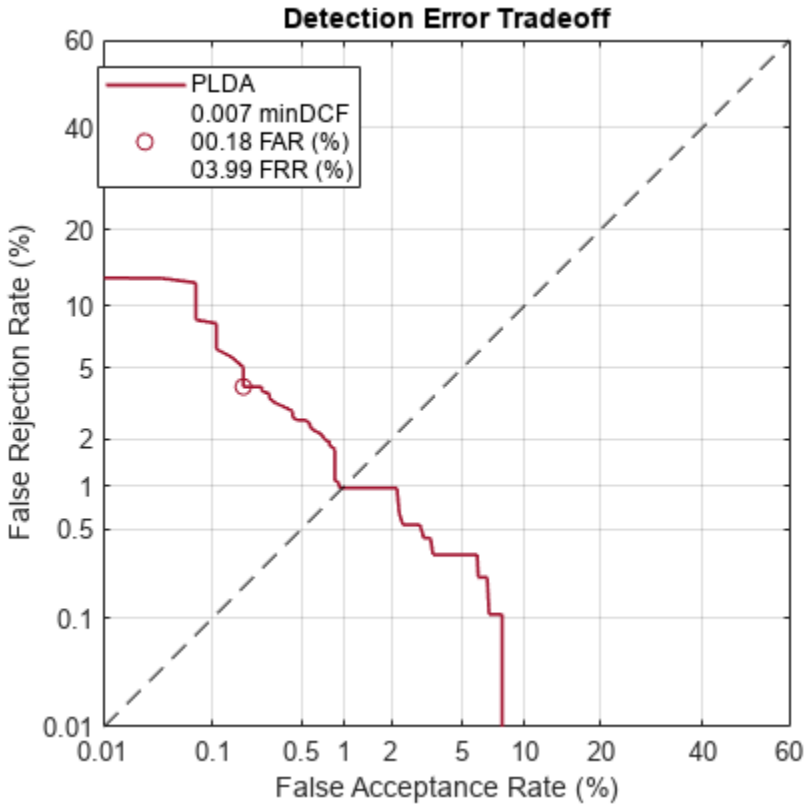
Call `detectionErrorTradeoff` again. This time, visualize only the detection-error tradeoff for the PLDA scorer.

```
detectionErrorTradeoff(speakerVerification,Scorer="plda")
```



Depending on your application, you may want to use a threshold that weights the error cost of a false alarm higher or lower than the error cost of a false rejection. You may also be using data that is not representative of the prior probability of the speaker being present. You can use the `minDCF` parameter to specify custom costs and prior probability. Call `detectionErrorTradeoff` again, this time specify the cost of a false rejection as 1, the cost of a false acceptance as 2, and the prior probability that a speaker is present as 0.1.

```
costFR = 1;
costFA = 2;
priorProb = 0.1;
detectionErrorTradeoff(speakerVerification,Scorer="plda",minDCF=[costFR,costFA,priorProb])
```



Call `detectionErrorTradeoff` again. This time, get the `minDCF` threshold for the PLDA scorer and the parameters of the detection cost function.

```
[~,minDCFThreshold] = detectionErrorTradeoff(speakerVerification,Scorer="plda",minDCF=[costFR,cos
```

```
minDCFThreshold = 0.4709
```

Test Speaker Verification System

Read a signal from the test set.

```
adsTest = shuffle(adsTest);
[audioIn,audioInfo] = read(adsTest);
knownSpeakerID = audioInfo.Label
```

```
knownSpeakerID = 1×1 cell array
    {'F01'}
```

To perform speaker verification, call `verify` with the audio signal and specify the speaker ID, a scorer, and a threshold for the scorer. The `verify` function returns a logical value indicating whether a speaker identity is accepted or rejected, and a score indicating the similarity of the input audio and the template i-vector corresponding to the enrolled label.

```
[tf,score] = verify(speakerVerification,audioIn,knownSpeakerID,"plda",eerThreshold.PLDA);
if tf
    fprintf('Success!\nSpeaker accepted.\nSimilarity score = %0.2f\n\n',score)
else
```

```
    fprinf('Failure!\nSpeaker rejected.\nSimilarity score = %0.2f\n\n',score)
end
```

```
Success!
Speaker accepted.
Similarity score = 1.00
```

Call speaker verification again. This time, specify an incorrect speaker ID.

```
possibleSpeakers = speakerVerification.EnrolledLabels.Properties.RowNames;
imposterIdx = find(~ismember(possibleSpeakers,knownSpeakerID));
imposter = possibleSpeakers(imposterIdx(randperm(numel(imposterIdx),1)))
```

```
imposter = 1×1 cell array
    {'M05'}
```

```
[tf,score] = verify(speakerVerification,audioIn,imposter,"plda",eerThreshold.PLDA);
if tf
    fprintf('Failure!\nSpeaker accepted.\nSimilarity score = %0.2f\n\n',score)
else
    fprintf('Success!\nSpeaker rejected.\nSimilarity score = %0.2f\n\n',score)
end
```

```
Success!
Speaker rejected.
Similarity score = 0.00
```

References

[1] Signal Processing and Speech Communication Laboratory. https://www.spsc.tugraz.at/databases-and-tools/ptdb-tug-pitch-tracking-database-from-graz-university-of-technology.html. Accessed 12 Dec. 2019.

Input Arguments

`ivs` — i-vector system

`ivectorSystem` object

i-vector system, specified as an object of type `ivectorSystem`.

`data` — Training data for i-vector system

cell array | `audioDatastore` | `signalDatastore` | `TransformedDatastore`

Training data for an i-vector system, specified as a cell array or as an `audioDatastore`, `signalDatastore`, or `TransformedDatastore` object.

- If `InputType` is set to `"audio"` when the i-vector system is created, specify `data` as one of these:
  - A cell array of single-channel audio signals, each specified as a column vector with underlying type `single` or `double`.
  - An `audioDatastore` object or a `signalDatastore` object that points to a data set of mono audio signals.
  - A `TransformedDatastore` with an underlying `audioDatastore` or `signalDatastore` that points to a data set of mono audio signals. The output from calls to `read` from the transform datastore must be mono audio signals with underlying data type `single` or `double`.

- If `InputType` is set to `"features"` when the i-vector system is created, specify `data` as one of these:
  - A cell array of matrices with underlying type `single` or `double`. The matrices must consist of audio features where the number of features (columns) is locked the first time `trainExtractor` is called and the number of hops (rows) is variable-sized. The number of features input in any subsequent calls to any of the object functions must be equal to the number of features used when calling `trainExtractor`.
  - A `TransformedDatastore` object with an underlying `audioDatastore` or `signalDatastore` whose `read` function has output as described in the previous bullet.
  - A `signalDatastore` object whose `read` function has output as described in the first bullet.

Data Types: `cell` | `audioDatastore` | `signalDatastore`

`labels` — Classification labels
categorical array | cell array | string array

Classification labels used by the i-vector system, specified as one of the following:

- A categorical array
- A cell array of character vectors
- A string array

Note The number of audio signals in `data` must match the number of `labels`.

Data Types: `categorical` | `cell` | `string`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `trainClassifier(ivs,data,labels,PLDANumIterations=D)`

`NumEignevectors` — Number of eigenvectors
16 (default) | positive integer

The number of eigenvectors used to perform dimensionality reduction, specified as a positive integer.

Example: `trainClassifier(ivs,data,labels,NumEigenvectors=18)`

Data Types: `single` | `double`

`PLDANumIterations` — Number of expectation-maximization iterations
5 (default) | positive integer

The number of expectation-maximization iterations used to train the probabilistic linear discriminant analysis (PLDA) model, specified as a positive integer.

Example: `trainClassifier(ivs,data,labels,PLDANumIterations=3)`

Data Types: `single` | `double`

`PLDANumDimensions` — Maximum number of PLDA dimensions
16 (default) | positive integer

The maximum number of dimensions for the PLDA model, specified as a positive integer.

Example: `trainClassifier(ivs,data,labels,PLDANumDimensions=10)`

Note `PLDANumDimensions` must be less than or equal to the rank of the total variability matrix.

Data Types: `single` | `double`

`ExecutionEnvironment` — Hardware resource for execution
`"auto"` (default) | `"cpu"` | `"gpu"` | `"multi-gpu"` | `"parallel"`

Hardware resource for execution, specified as one of these:

- `"auto"` — Use the GPU if it is available. Otherwise, use the CPU.
- `"cpu"` — Use the CPU.
- `"gpu"` — Use the GPU. This option requires Parallel Computing Toolbox.
- `"multi-gpu"` — Use multiple GPUs on one machine, using a local parallel pool based on your default cluster profile. If there is no current parallel pool, the software starts a parallel pool with pool size equal to the number of available GPUs. This option requires Parallel Computing Toolbox.
- `"parallel"` — Use a local or remote parallel pool based on your default cluster profile. If there is no current parallel pool, the software starts one using the default cluster profile. If the pool has access to GPUs, then only workers with a unique GPU perform training computation. If the pool does not have GPUs, then the training takes place on all available CPU workers. This option requires Parallel Computing Toolbox.

Data Types: `char` | `string`

`DispatchInBackground` — Option to use prefetch queuing
`false` (default) | `true`

Option to use prefetch queuing when reading from a datastore, specified as a logical value. This argument requires Parallel Computing Toolbox.

Data Types: `logical`

Version History

Introduced in R2021a

See Also

`trainExtractor` | `calibrate` | `enroll` | `unenroll` | `detectionErrorTradeoff` | `verify` | `identify` | `ivector` | `info` | `addInfoHeader` | `release` | `ivectorSystem` | `speakerRecognition`

calibrate

Train i-vector system calibrator

Syntax

```
calibrate(ivs,data,labels)
calibrate(ivs,data,labels,Name=Value)
```

Description

`calibrate(ivs,data,labels)` calibrates scores output from by i-vector system `ivs`. The calibration scheme is determined using the calibration data and associated labels. A properly calibrated system outputs scores that can be interpreted as confidence in a positive decision.

`calibrate(ivs,data,labels,Name=Value)` specifies additional options using name-value arguments. You can choose the hardware resource to extract i-vectors and whether to use prefetch queuing when reading from a datastore.

Examples

Train Acoustic Fault Recognition System

Download and unzip the air compressor data set [1] on page 4-343. This data set consists of recordings from air compressors in a healthy state or one of seven faulty states.

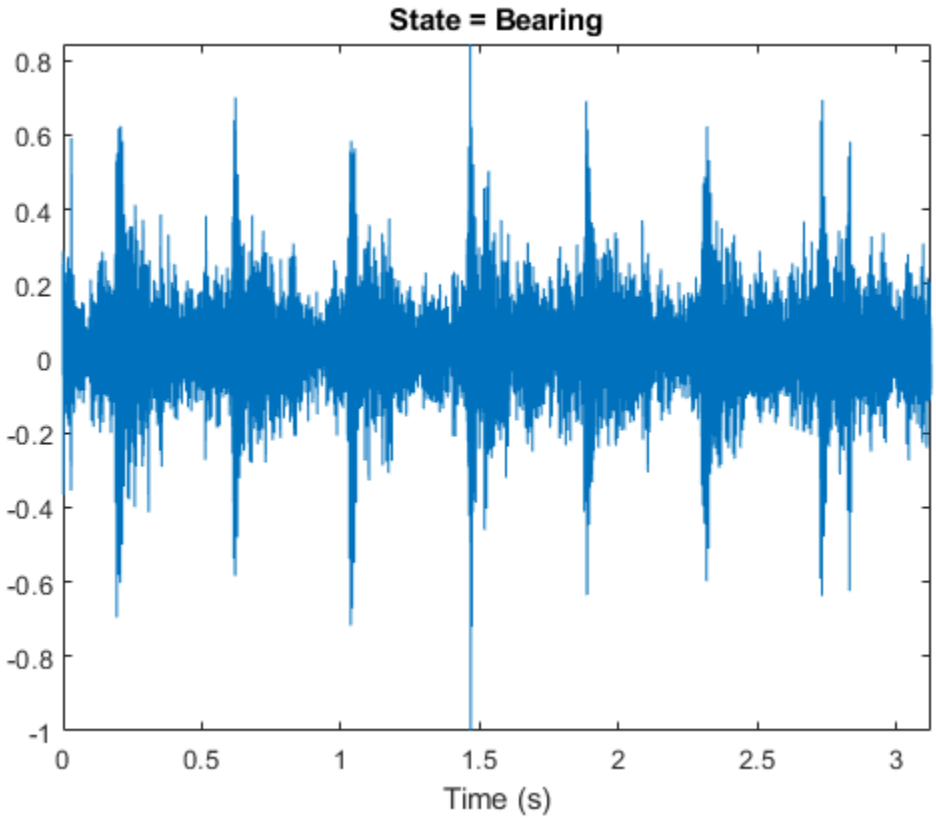
```
loc = matlab.internal.examples.downloadSupportFile("audio", ...
    "AirCompressorDataset/AirCompressorDataset.zip");
unzip(loc,pwd)
```

Create an `audioDatastore` object to manage the data and split it into training and validation sets.

```
ads = audioDatastore(pwd,IncludeSubfolders=true,LabelSource="foldernames");

[adsTrain,adsTest] = splitEachLabel(ads,0.8,0.2);
```

Read an audio file from the datastore and save the sample rate. Listen to the audio signal and plot the signal in the time domain.

```
[x,fileInfo] = read(adsTrain);
fs = fileInfo.SampleRate;

sound(x,fs)

t = (0:size(x,1)-1)/fs;
plot(t,x)
xlabel("Time (s)")
title("State = " + string(fileInfo.Label))
axis tight
```



Create an i-vector system with `DetectSpeech` set to `false`. Turn off the verbose behavior.

```
faultRecognizer = ivectorSystem(SampleRate=fs,DetectSpeech=false, ...
    Verbose=false)
```

```
faultRecognizer =
  ivectorSystem with properties:

         InputType: 'audio'
        SampleRate: 16000
      DetectSpeech: 0
           Verbose: 0
    EnrolledLabels: [0×2 table]
```

Train the i-vector extractor and the i-vector classifier using the training datastore.

```
trainExtractor(faultRecognizer,adsTrain, ...
    UBMNumComponents=80, ...
    UBMNumIterations=3, ...
    ...
    TVSRank=40, ...
    TVSNumIterations=3)

trainClassifier(faultRecognizer,adsTrain,adsTrain.Labels, ...
    NumEigenvectors=7, ...
    ...
```

```
    PLDANumDimensions=32, ...
    PLDANumIterations=5)
```

Calibrate the scores output by `faultRecognizer` so they can be interpreted as a measure of confidence in a positive decision. Turn the verbose behavior back on. Enroll all of the labels from the training set.

```
calibrate(faultRecognizer,adsTrain,adsTrain.Labels)

faultRecognizer.Verbose = true;

enroll(faultRecognizer,adsTrain,adsTrain.Labels)
```

```
Extracting i-vectors ...done.
Enrolling i-vectors ..........done.
Enrollment complete.
```

Use the read-only property `EnrolledLabels` to view the enrolled labels and the corresponding i-vector templates.

```
faultRecognizer.EnrolledLabels
```

```
ans=8×2 table
                  ivector       NumSamples
                ____________    __________

    Bearing     {7×1 double}       180
    Flywheel    {7×1 double}       180
    Healthy     {7×1 double}       180
    LIV         {7×1 double}       180
    LOV         {7×1 double}       180
    NRV         {7×1 double}       180
    Piston      {7×1 double}       180
    Riderbelt   {7×1 double}       180
```

Use the `identify` function with the PLDA scorer to predict the condition of machines in the test set. The `identify` function returns a table of possible labels sorted in descending order of confidence.

```
[audioIn,audioInfo] = read(adsTest);
trueLabel = audioInfo.Label
```

```
trueLabel = categorical
     Bearing
```

```
predictedLabels = identify(faultRecognizer,audioIn,"plda")
```

```
predictedLabels=8×2 table
      Label        Score
    _________    _________

    Bearing        0.99997
    Flywheel      2.265e-05
    Piston       8.6076e-08
    LIV          1.4237e-15
    NRV          4.5529e-16
    Riderbelt    3.7359e-16
    LOV          6.3025e-19
```

```
    Healthy      4.2094e-30
```

By default, the `identify` function returns all possible candidate labels and their corresponding scores. Use `NumCandidates` to reduce the number of candidates returned.

```
results = identify(faultRecognizer,audioIn,"plda",NumCandidates=3)
```

```
results=3×2 table
     Label        Score
    ________    __________

    Bearing        0.99997
    Flywheel     2.265e-05
    Piston       8.6076e-08
```

References

[1] Verma, Nishchal K., et al. “Intelligent Condition Based Monitoring Using Acoustic Signals for Air Compressors.” *IEEE Transactions on Reliability*, vol. 65, no. 1, Mar. 2016, pp. 291–309. *DOI.org (Crossref)*, doi:10.1109/TR.2015.2459684.

Input Arguments

`ivs` — i-vector system

`ivectorSystem` object

i-vector system, specified as an object of type `ivectorSystem`.

`data` — Training data for i-vector system

cell array | `audioDatastore` | `signalDatastore` | `TransformedDatastore`

Training data for an i-vector system, specified as a cell array or as an `audioDatastore`, `signalDatastore`, or `TransformedDatastore` object.

- If `InputType` is set to `"audio"` when the i-vector system is created, specify `data` as one of these:
  - A cell array of single-channel audio signals, each specified as a column vector with underlying type `single` or `double`.
  - An `audioDatastore` object or a `signalDatastore` object that points to a data set of mono audio signals.
  - A `TransformedDatastore` with an underlying `audioDatastore` or `signalDatastore` that points to a data set of mono audio signals. The output from calls to `read` from the transform datastore must be mono audio signals with underlying data type `single` or `double`.
- If `InputType` is set to `"features"` when the i-vector system is created, specify `data` as one of these:
  - A cell array of matrices with underlying type `single` or `double`. The matrices must consist of audio features where the number of features (columns) is locked the first time `trainExtractor` is called and the number of hops (rows) is variable-sized. The number of features input in any subsequent calls to any of the object functions must be equal to the number of features used when calling `trainExtractor`.

- A `TransformedDatastore` object with an underlying `audioDatastore` or `signalDatastore` whose `read` function has output as described in the previous bullet.
- A `signalDatastore` object whose `read` function has output as described in the first bullet.

Data Types: `cell` | `audioDatastore` | `signalDatastore`

`labels` — Classification labels
character vector | string | cell array | string array | categorical array

Classification labels used by an i-vector system, specified as one of these:

- A categorical array
- A cell array of character vectors
- A string array

Data Types: `categorical` | `cell` | `string`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `calibrate(ivs,data,labels,DispatchInBackground=true)`

`ExecutionEnvironment` — Hardware resource for execution
`"auto"` (default) | `"cpu"` | `"gpu"` | `"multi-gpu"` | `"parallel"`

Hardware resource for execution, specified as one of these:

- `"auto"` — Use the GPU if it is available. Otherwise, use the CPU.
- `"cpu"` — Use the CPU.
- `"gpu"` — Use the GPU. This option requires Parallel Computing Toolbox.
- `"multi-gpu"` — Use multiple GPUs on one machine, using a local parallel pool based on your default cluster profile. If there is no current parallel pool, the software starts a parallel pool with pool size equal to the number of available GPUs. This option requires Parallel Computing Toolbox.
- `"parallel"` — Use a local or remote parallel pool based on your default cluster profile. If there is no current parallel pool, the software starts one using the default cluster profile. If the pool has access to GPUs, then only workers with a unique GPU perform training computation. If the pool does not have GPUs, then the training takes place on all available CPU workers. This option requires Parallel Computing Toolbox.

Data Types: `char` | `string`

`DispatchInBackground` — Option to use prefetch queuing
`false` (default) | `true`

Option to use prefetch queuing when reading from a datastore, specified as a logical value. This argument requires Parallel Computing Toolbox.

Data Types: `logical`

Version History

Introduced in R2022a

See Also

`trainExtractor` | `trainClassifier` | `enroll` | `unenroll` | `detectionErrorTradeoff` | `verify` | `identify` | `ivector` | `info` | `addInfoHeader` | `release` | `ivectorSystem` | `speakerRecognition`

enroll

Enroll labels

Syntax

```
enroll(ivs,data,labels)
enroll(ivs,data,labels,Name,Value)
```

Description

`enroll(ivs,data,labels)` enrolls the data and corresponding labels into the i-vector system `ivs`.

`enroll(ivs,data,labels,Name,Value)` specifies additional options using name-value arguments. You can choose the hardware resource to extract i-vectors and whether to use prefetch queuing when reading from a datastore.

Examples

Train Speaker Verification System

Use the Pitch Tracking Database from Graz University of Technology (PTDB-TUG) [1] on page 4-356. The data set consists of 20 English native speakers reading 2342 phonetically rich sentences from the TIMIT corpus. Download and extract the data set.

```
downloadFolder = matlab.internal.examples.downloadSupportFile("audio","ptdb-tug.zip");
dataFolder = tempdir;
unzip(downloadFolder,dataFolder)
dataset = fullfile(dataFolder,"ptdb-tug");
```

Create an `audioDatastore` object that points to the data set. The data set was originally intended for use in pitch-tracking training and evaluation and includes laryngograph readings and baseline pitch decisions. Use only the original audio recordings.

```
ads = audioDatastore([fullfile(dataset,"SPEECH DATA","FEMALE","MIC"),fullfile(dataset,"SPEECH DA
                      IncludeSubfolders=true, ...
                      FileExtensions=".wav");
```

The file names contain the speaker IDs. Decode the file names to set the labels in the `audioDatastore` object.

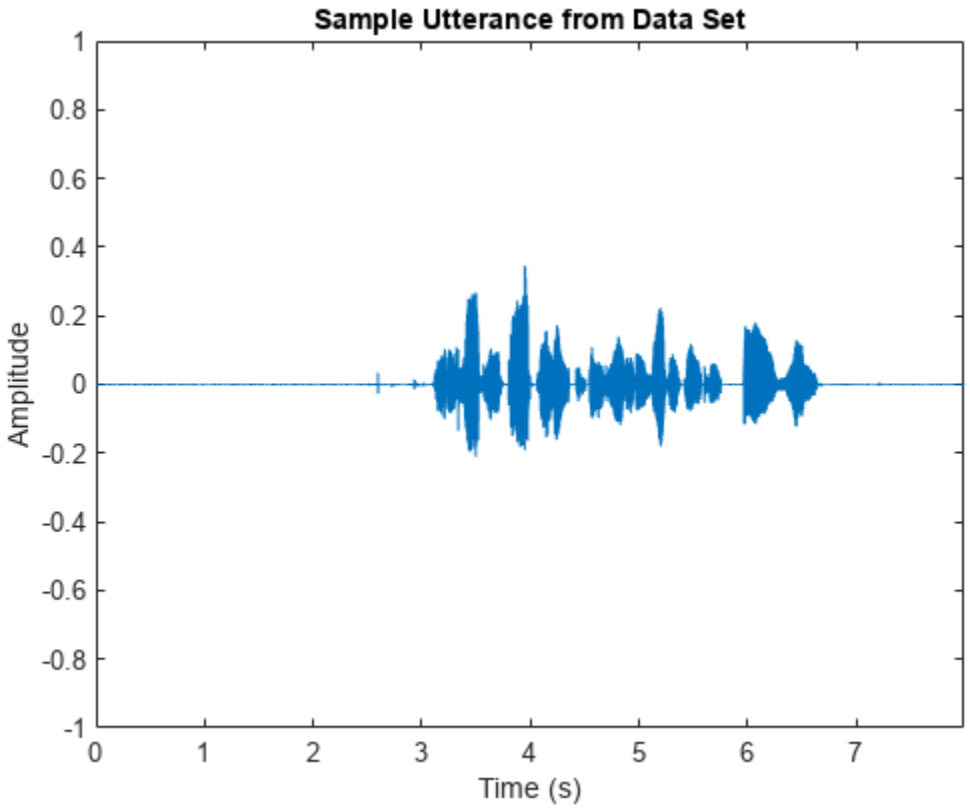
```
ads.Labels = extractBetween(ads.Files,"mic_","_");
countEachLabel(ads)
```

```
ans=20×2 table
    Label    Count
    _____    _____

     F01      236
     F02      236
     F03      236
     F04      236
```

```
F05      236
F06      236
F07      236
F08      234
F09      236
F10      236
M01      236
M02      236
M03      236
M04      236
M05      236
M06      236
  ⋮
```

Read an audio file from the data set, listen to it, and plot it.

```
[audioIn,audioInfo] = read(ads);
fs = audioInfo.SampleRate;

t = (0:size(audioIn,1)-1)/fs;
sound(audioIn,fs)
plot(t,audioIn)
xlabel("Time (s)")
ylabel("Amplitude")
axis([0 t(end) -1 1])
title("Sample Utterance from Data Set")
```



Separate the `audioDatastore` object into four: one for training, one for enrollment, one to evaluate the detection-error tradeoff, and one for testing. The training set contains 16 speakers. The enrollment, detection-error tradeoff, and test sets contain the other four speakers.

```
speakersToTest = categorical(["M01","M05","F01","F05"]);

adsTrain = subset(ads,~ismember(ads.Labels,speakersToTest));

ads = subset(ads,ismember(ads.Labels,speakersToTest));
[adsEnroll,adsTest,adsDET] = splitEachLabel(ads,3,1);
```

Display the label distributions of the `audioDatastore` objects.

```
countEachLabel(adsTrain)
```

```
ans=16×2 table
    Label    Count
    _____    _____

     F02      236
     F03      236
     F04      236
     F06      236
     F07      236
     F08      234
     F09      236
     F10      236
     M02      236
     M03      236
     M04      236
     M06      236
     M07      236
     M08      236
     M09      236
     M10      236
```

```
countEachLabel(adsEnroll)
```

```
ans=4×2 table
    Label    Count
    _____    _____

     F01       3
     F05       3
     M01       3
     M05       3
```

```
countEachLabel(adsTest)
```

```
ans=4×2 table
    Label    Count
    _____    _____

     F01       1
     F05       1
     M01       1
```

```
     M05       1
```

```
countEachLabel(adsDET)
```

```
ans=4×2 table
    Label    Count
    _____    _____

     F01      232
     F05      232
     M01      232
     M05      232
```

Create an i-vector system. By default, the i-vector system assumes the input to the system is mono audio signals.

```
speakerVerification = ivectorSystem(SampleRate=fs)
```

```
speakerVerification =
  ivectorSystem with properties:

         InputType: 'audio'
        SampleRate: 48000
      DetectSpeech: 1
           Verbose: 1
    EnrolledLabels: [0×2 table]
```

To train the extractor of the i-vector system, call `trainExtractor`. Specify the number of universal background model (UBM) components as 128 and the number of expectation maximization iterations as 5. Specify the total variability space (TVS) rank as 64 and the number of iterations as 3.

```
trainExtractor(speakerVerification,adsTrain, ...
    UBMNumComponents=128,UBMNumIterations=5, ...
    TVSRank=64,TVSNumIterations=3)
```

```
Calculating standardization factors ....done.
Training universal background model ........done.
Training total variability space ......done.
i-vector extractor training complete.
```

To train the classifier of the i-vector system, use `trainClassifier`. To reduce dimensionality of the i-vectors, specify the number of eigenvectors in the projection matrix as 16. Specify the number of dimensions in the probabilistic linear discriminant analysis (PLDA) model as 16, and the number of iterations as 3.

```
trainClassifier(speakerVerification,adsTrain,adsTrain.Labels, ...
    NumEigenvectors=16, ...
    PLDANumDimensions=16,PLDANumIterations=3)
```

```
Extracting i-vectors ...done.
Training projection matrix .....done.
Training PLDA model ......done.
i-vector classifier training complete.
```

To calibrate the system so that scores can be interpreted as a measure of confidence in a positive decision, use `calibrate`.

```
calibrate(speakerVerification,adsTrain,adsTrain.Labels)
```

```
Extracting i-vectors ...done.
Calibrating CSS scorer ...done.
Calibrating PLDA scorer ...done.
Calibration complete.
```

To inspect parameters used previously to train the i-vector system, use `info`.

```
info(speakerVerification)
```

```
i-vector system input
  Input feature vector length: 60
  Input data type: double

trainExtractor
  Train signals: 3774
  UBMNumComponents: 128
  UBMNumIterations: 5
  TVSRank: 64
  TVSNumIterations: 3

trainClassifier
  Train signals: 3774
  Train labels: F02 (236), F03 (236) ... and 14 more
  NumEigenvectors: 16
  PLDANumDimensions: 16
  PLDANumIterations: 3

calibrate
  Calibration signals: 3774
  Calibration labels: F02 (236), F03 (236) ... and 14 more
```

Split the enrollment set.

```
[adsEnrollPart1,adsEnrollPart2] = splitEachLabel(adsEnroll,1,2);
```

To enroll speakers in the i-vector system, call `enroll`.

```
enroll(speakerVerification,adsEnrollPart1,adsEnrollPart1.Labels)
```

```
Extracting i-vectors ...done.
Enrolling i-vectors .......done.
Enrollment complete.
```

When you enroll speakers, the read-only `EnrolledLabels` property is updated with the enrolled labels and corresponding template i-vectors. The table also keeps track of the number of signals used to create the template i-vector. Generally, using more signals results in a better template.

```
speakerVerification.EnrolledLabels
```

```
ans=4×2 table
              ivector       NumSamples
            ____________    __________

    F01    {16×1 double}        1
    F05    {16×1 double}        1
    M01    {16×1 double}        1
```

```
    M05    {16×1 double}        1
```

Enroll the second part of the enrollment set and then view the enrolled labels table again. The i-vector templates and the number of samples are updated.

```
enroll(speakerVerification,adsEnrollPart2,adsEnrollPart2.Labels)
```

```
Extracting i-vectors ...done.
Enrolling i-vectors .......done.
Enrollment complete.
```

```
speakerVerification.EnrolledLabels
```

```
ans=4×2 table
              ivector       NumSamples
           _____________    __________

    F01    {16×1 double}        3
    F05    {16×1 double}        3
    M01    {16×1 double}        3
    M05    {16×1 double}        3
```

To evaluate the i-vector system and determine a decision threshold for speaker verification, call `detectionErrorTradeoff`.

```
[results, eerThreshold] = detectionErrorTradeoff(speakerVerification,adsDET,adsDET.Labels);
```

```
Extracting i-vectors ...done.
Scoring i-vector pairs ...done.
Detection error tradeoff evaluation complete.
```

The first output from `detectionErrorTradeoff` is a structure with two fields: CSS and PLDA. Each field contains a table. Each row of the table contains a possible decision threshold for speaker verification tasks, and the corresponding false alarm rate (FAR) and false rejection rate (FRR). The FAR and FRR are determined using the enrolled speaker labels and the data input to the `detectionErrorTradeoff` function.

```
results
```

```
results = struct with fields:
    PLDA: [1000×3 table]
     CSS: [1000×3 table]
```

```
results.CSS
```

```
ans=1000×3 table
    Threshold       FAR      FRR
    __________    _______    ___

    2.3259e-10          1     0
    2.3965e-10    0.99964     0
    2.4693e-10    0.99928     0
    2.5442e-10    0.99928     0
    2.6215e-10    0.99928     0
     2.701e-10    0.99928     0
     2.783e-10    0.99928     0
```

```
    2.8675e-10    0.99928     0
    2.9545e-10    0.99928     0
    3.0442e-10    0.99928     0
    3.1366e-10    0.99928     0
    3.2318e-10    0.99928     0
    3.3299e-10    0.99928     0
     3.431e-10    0.99928     0
    3.5352e-10    0.99928     0
    3.6425e-10    0.99892     0
      ⋮
```

```
results.PLDA
```

```
ans=1000×3 table
    Threshold       FAR      FRR
    __________    _______    ___

    3.2661e-40          1     0
    3.6177e-40    0.99964     0
    4.0072e-40    0.99964     0
    4.4387e-40    0.99964     0
    4.9166e-40    0.99964     0
    5.4459e-40    0.99964     0
    6.0322e-40    0.99964     0
    6.6817e-40    0.99964     0
    7.4011e-40    0.99964     0
     8.198e-40    0.99964     0
    9.0806e-40    0.99964     0
    1.0058e-39    0.99964     0
    1.1141e-39    0.99964     0
    1.2341e-39    0.99964     0
    1.3669e-39    0.99964     0
    1.5141e-39    0.99964     0
      ⋮
```

The second output from `detectionErrorTradeoff` is a structure with two fields: `CSS` and `PLDA`. The corresponding value is the decision threshold that results in the equal error rate (when FAR and FRR are equal).

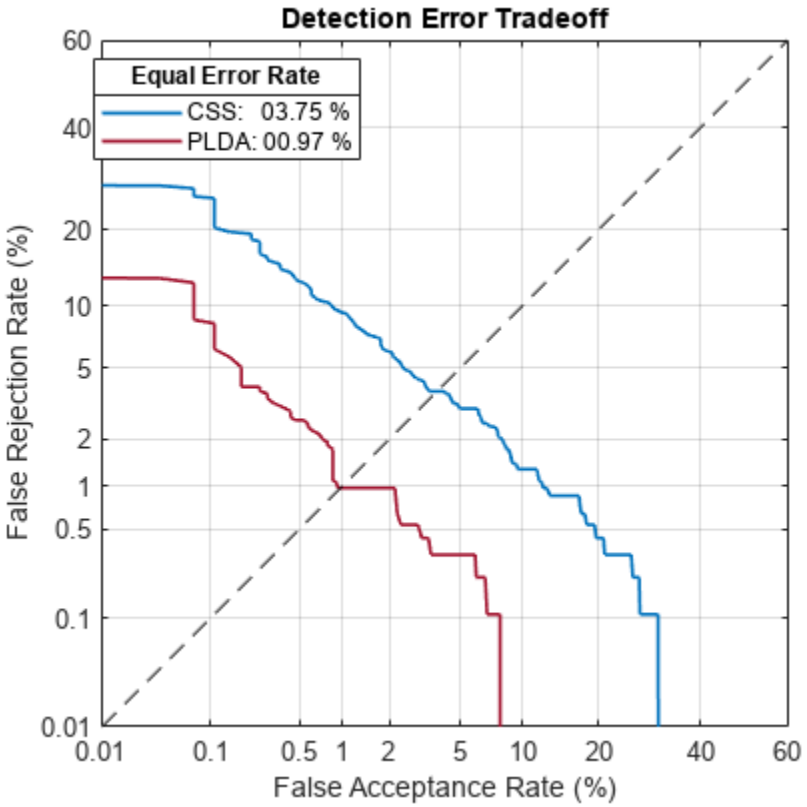
```
eerThreshold
```

```
eerThreshold = struct with fields:
    PLDA: 0.0398
     CSS: 0.9369
```

The first time you call `detectionErrorTradeoff`, you must provide data and corresponding labels to evaluate. Subsequently, you can get the same information, or a different analysis using the same underlying data, by calling `detectionErrorTradeoff` without data and labels.

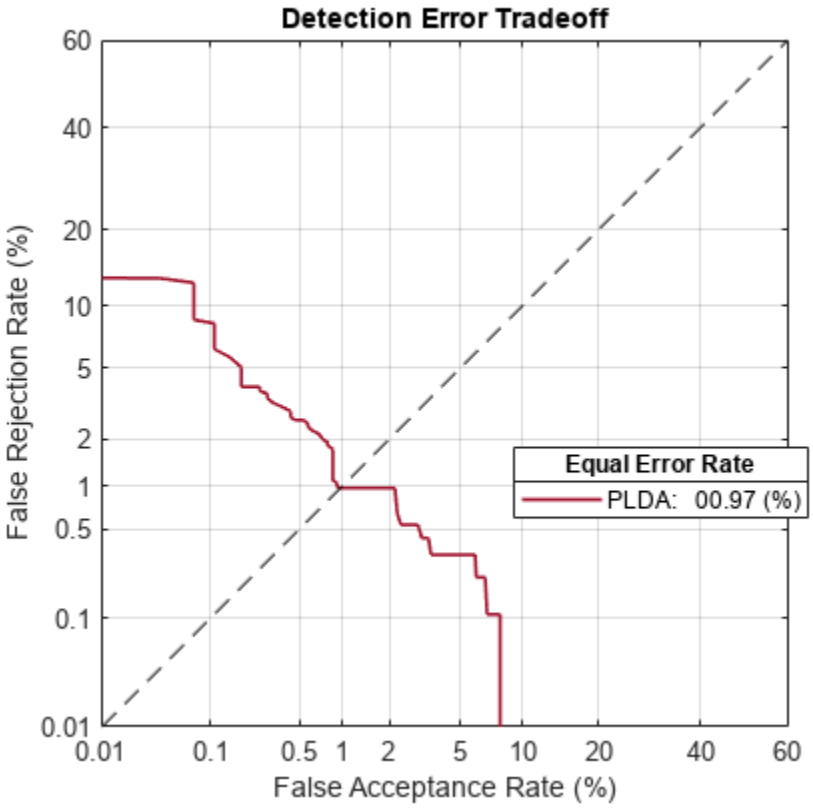
Call `detectionErrorTradeoff` a second time with no data arguments or output arguments to visualize the detection-error tradeoff.

```
detectionErrorTradeoff(speakerVerification)
```



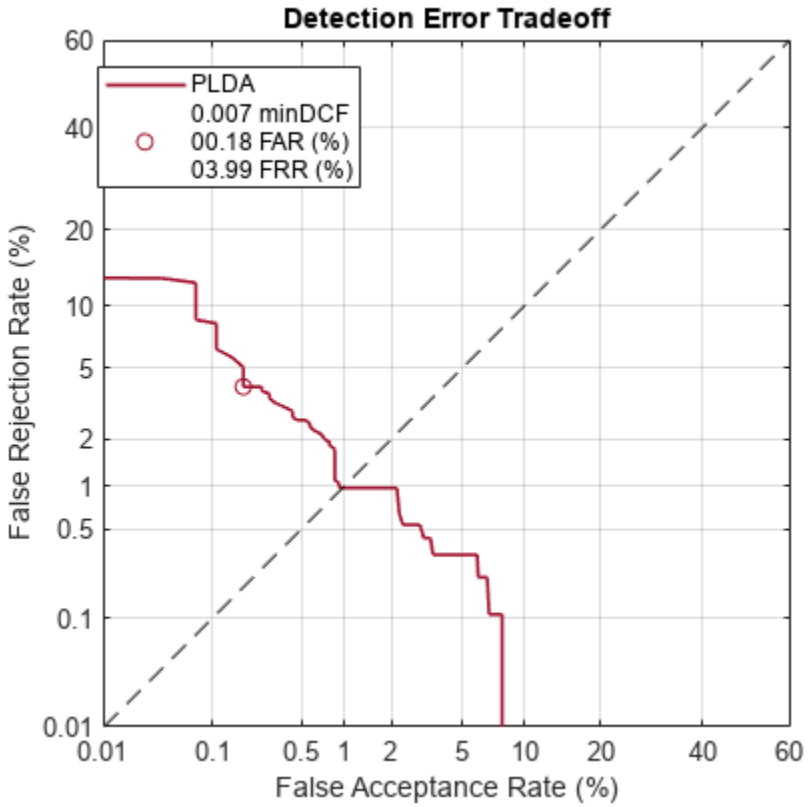
Call `detectionErrorTradeoff` again. This time, visualize only the detection-error tradeoff for the PLDA scorer.

```
detectionErrorTradeoff(speakerVerification,Scorer="plda")
```



Depending on your application, you may want to use a threshold that weights the error cost of a false alarm higher or lower than the error cost of a false rejection. You may also be using data that is not representative of the prior probability of the speaker being present. You can use the `minDCF` parameter to specify custom costs and prior probability. Call `detectionErrorTradeoff` again, this time specify the cost of a false rejection as 1, the cost of a false acceptance as 2, and the prior probability that a speaker is present as 0.1.

```
costFR = 1;
costFA = 2;
priorProb = 0.1;
detectionErrorTradeoff(speakerVerification,Scorer="plda",minDCF=[costFR,costFA,priorProb])
```



Call `detectionErrorTradeoff` again. This time, get the `minDCF` threshold for the PLDA scorer and the parameters of the detection cost function.

```
[~,minDCFThreshold] = detectionErrorTradeoff(speakerVerification,Scorer="plda",minDCF=[costFR,cos
```

```
minDCFThreshold = 0.4709
```

Test Speaker Verification System

Read a signal from the test set.

```
adsTest = shuffle(adsTest);
[audioIn,audioInfo] = read(adsTest);
knownSpeakerID = audioInfo.Label
```

```
knownSpeakerID = 1×1 cell array
    {'F01'}
```

To perform speaker verification, call `verify` with the audio signal and specify the speaker ID, a scorer, and a threshold for the scorer. The `verify` function returns a logical value indicating whether a speaker identity is accepted or rejected, and a score indicating the similarity of the input audio and the template i-vector corresponding to the enrolled label.

```
[tf,score] = verify(speakerVerification,audioIn,knownSpeakerID,"plda",eerThreshold.PLDA);
if tf
    fprintf('Success!\nSpeaker accepted.\nSimilarity score = %0.2f\n\n',score)
else
```

```
    fprinf('Failure!\nSpeaker rejected.\nSimilarity score = %0.2f\n\n',score)
end
```

```
Success!
Speaker accepted.
Similarity score = 1.00
```

Call speaker verification again. This time, specify an incorrect speaker ID.

```
possibleSpeakers = speakerVerification.EnrolledLabels.Properties.RowNames;
imposterIdx = find(~ismember(possibleSpeakers,knownSpeakerID));
imposter = possibleSpeakers(imposterIdx(randperm(numel(imposterIdx),1)))
```

```
imposter = 1×1 cell array
    {'M05'}
```

```
[tf,score] = verify(speakerVerification,audioIn,imposter,"plda",eerThreshold.PLDA);
if tf
    fprintf('Failure!\nSpeaker accepted.\nSimilarity score = %0.2f\n\n',score)
else
    fprintf('Success!\nSpeaker rejected.\nSimilarity score = %0.2f\n\n',score)
end
```

```
Success!
Speaker rejected.
Similarity score = 0.00
```

References

[1] Signal Processing and Speech Communication Laboratory. https://www.spsc.tugraz.at/databases-and-tools/ptdb-tug-pitch-tracking-database-from-graz-university-of-technology.html. Accessed 12 Dec. 2019.

Input Arguments

`ivs` — i-vector system

`ivectorSystem` object

i-vector system, specified as an object of type `ivectorSystem`.

`data` — Labeled enrollment data

column vector | cell array | `audioDatastore` | `signalDatastore` | `TransformedDatastore`

Labeled enrollment data, specified as a cell array or as an `audioDatastore`, `signalDatastore`, or `TransformedDatastore` object.

- If `InputType` is set to `"audio"` when the i-vector system is created, specify `data` as one of these:
  - A column vector with underlying type `single` or `double`.
  - A cell array of single-channel audio signals, each specified as a column vector with underlying type `single` or `double`.
  - An `audioDatastore` object or a `signalDatastore` object that points to a data set of mono audio signals.

  - A `TransformedDatastore` with an underlying `audioDatastore` or `signalDatastore` that points to a data set of mono audio signals. The output from calls to `read` from the transform datastore must be mono audio signals with underlying data type `single` or `double`.
- If `InputType` is set to `"features"` when the i-vector system is created, specify `data` as one of these:
  - A cell array of matrices with underlying type `single` or `double`. The matrices must consist of audio features where the number of features (columns) is locked the first time `trainExtractor` is called and the number of hops (rows) is variable-sized. The number of features input in any subsequent calls to any of the object functions must be equal to the number of features used when calling `trainExtractor`.
  - A `TransformedDatastore` object with an underlying `audioDatastore` or `signalDatastore` whose `read` function has output as described in the previous bullet.
  - A `signalDatastore` object whose `read` function has output as described in the first bullet.

Data Types: `cell` | `audioDatastore` | `signalDatastore`

`labels` — Classification labels
character vector | string | cell array | string array | categorical array

Classification labels used by an i-vector system, specified as one of these:

- A categorical array
- A cell array of character vectors
- A string array

Data Types: `categorical` | `cell` | `string`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `enroll(ivs,data,labels,DispatchInBackground=true)`

`ExecutionEnvironment` — Hardware resource for execution
`"auto"` (default) | `"cpu"` | `"gpu"` | `"multi-gpu"` | `"parallel"`

Hardware resource for execution, specified as one of these:

- `"auto"` — Use the GPU if it is available. Otherwise, use the CPU.
- `"cpu"` — Use the CPU.
- `"gpu"` — Use the GPU. This option requires Parallel Computing Toolbox.
- `"multi-gpu"` — Use multiple GPUs on one machine, using a local parallel pool based on your default cluster profile. If there is no current parallel pool, the software starts a parallel pool with pool size equal to the number of available GPUs. This option requires Parallel Computing Toolbox.
- `"parallel"` — Use a local or remote parallel pool based on your default cluster profile. If there is no current parallel pool, the software starts one using the default cluster profile. If the pool has access to GPUs, then only workers with a unique GPU perform training computation. If the pool

does not have GPUs, then the training takes place on all available CPU workers. This option requires Parallel Computing Toolbox.

Data Types: `char` | `string`

`DispatchInBackground` — Option to use prefetch queuing
`false` (default) | `true`

Option to use prefetch queuing when reading from a datastore, specified as a logical value. This argument requires Parallel Computing Toolbox.

Data Types: `logical`

Version History

Introduced in R2021a

See Also

`trainExtractor` | `trainClassifier` | `calibrate` | `unenroll` | `detectionErrorTradeoff` | `verify` | `identify` | `ivector` | `info` | `addInfoHeader` | `release` | `ivectorSystem` | `speakerRecognition`

unenroll

Unenroll labels

Syntax

```
unenroll(ivs)
unenroll(ivs,labels)
```

Description

`unenroll(ivs)` unenrolls all labels and corresponding i-vectors from the i-vector system `ivs`.

`unenroll(ivs,labels)` unenrolls the specified labels and corresponding i-vectors from the i-vector system `ivs`.

Examples

Train Speech Emotion Recognition System

Download the Berlin Database of Emotional Speech [1] on page 4-367. The database contains 535 utterances spoken by 10 actors intended to convey one of the following emotions: anger, boredom, disgust, anxiety/fear, happiness, sadness, or neutral. The emotions are text independent.

```
url = "http://emodb.bilderbar.info/download/download.zip";
downloadFolder = tempdir;
datasetFolder = fullfile(downloadFolder,"Emo-DB");

if ~exist(datasetFolder,"dir")
    disp("Downloading Emo-DB (40.5 MB) ...")
    unzip(url,datasetFolder)
end
```

Create an `audioDatastore` that points to the audio files.

```
ads = audioDatastore(fullfile(datasetFolder,"wav"));
```

The file names are codes indicating the speaker id, text spoken, emotion, and version. The website contains a key for interpreting the code and additional information about the speakers such as gender and age. Create a table with the variables `Speaker` and `Emotion`. Decode the file names into the table.

```
filepaths = ads.Files;
emotionCodes = cellfun(@(x)x(end-5),filepaths,"UniformOutput",false);
emotions = replace(emotionCodes,{'W','L','E','A','F','T','N'}, ...
    {'Anger','Boredom','Disgust','Anxiety','Happiness','Sadness','Neutral'});

speakerCodes = cellfun(@(x)x(end-10:end-9),filepaths,"UniformOutput",false);
labelTable = table(categorical(speakerCodes),categorical(emotions),VariableNames=["Speaker","Emo
summary(labelTable)
```

```
Variables:
```

```
    Speaker: 535×1 categorical

        Values:

            03       49
            08       58
            09       43
            10       38
            11       55
            12       35
            13       61
            14       69
            15       56
            16       71

    Emotion: 535×1 categorical

        Values:

            Anger          127
            Anxiety         69
            Boredom         81
            Disgust         46
            Happiness       71
            Neutral         79
            Sadness         62
```

`labelTable` is in the same order as the files in `audioDatastore`. Set the `Labels` property of the `audioDatastore` to `labelTable`.

```
ads.Labels = labelTable;
```

Read a signal from the datastore and listen to it. Display the speaker ID and emotion of the audio signal.

```
[audioIn,audioInfo] = read(ads);
fs = audioInfo.SampleRate;
sound(audioIn,fs)
audioInfo.Label
```

```
ans=1×2 table
    Speaker     Emotion
    _______    _________

      03       Happiness
```

Split the datastore into a training set and a test set. Assign two speakers to the test set and the remaining to the training set.

```
testSpeakerIdx = ads.Labels.Speaker=="12" | ads.Labels.Speaker=="13";
adsTrain = subset(ads,~testSpeakerIdx);
adsTest = subset(ads,testSpeakerIdx);
```

Read all the training and testing audio data into cell arrays. If your data can fit in memory, training is usually faster to input cell arrays to an i-vector system rather than datastores.

```
trainSet = readall(adsTrain);
trainLabels = adsTrain.Labels.Emotion;
```

```
testSet = readall(adsTest);
testLabels = adsTest.Labels.Emotion;
```

Create an i-vector system that does not apply speech detection. When `DetectSpeech` is set to `true` (the default), only regions of detected speech are used to train the i-vector system. When `DetectSpeech` is set to `false`, the entire input audio is used to train the i-vector system. The usefulness of applying speech detection depends on the data input to the system.

```
emotionRecognizer = ivectorSystem(SampleRate=fs,DetectSpeech=false)
```

```
emotionRecognizer =
  ivectorSystem with properties:

         InputType: 'audio'
        SampleRate: 16000
      DetectSpeech: 0
           Verbose: 1
    EnrolledLabels: [0×2 table]
```

Call `trainExtractor` using the training set.

```
rng default
trainExtractor(emotionRecognizer,trainSet, ...
    UBMNumComponents = 256, ...
    UBMNumIterations = 5, ...
    ...
    TVSRank = 128, ...
    TVSNumIterations = 5);
```

```
Calculating standardization factors .....done.
Training universal background model ........done.
Training total variability space ........done.
i-vector extractor training complete.
```

Copy the emotion recognition system for use later in the example.

```
sentimentRecognizer = copy(emotionRecognizer);
```

Call `trainClassifier` using the training set.

```
rng default
trainClassifier(emotionRecognizer,trainSet,trainLabels, ...
    NumEigenvectors = 32, ...
    ...
    PLDANumDimensions = 16, ...
    PLDANumIterations = 10);
```

```
Extracting i-vectors ...done.
Training projection matrix .....done.
Training PLDA model .............done.
i-vector classifier training complete.
```

Call `calibrate` using the training set. In practice, the calibration set should be different than the training set.

```
calibrate(emotionRecognizer,trainSet,trainLabels)
```

```
Extracting i-vectors ...done.
Calibrating CSS scorer ...done.
Calibrating PLDA scorer ...done.
Calibration complete.
```

Enroll the training labels into the i-vector system.

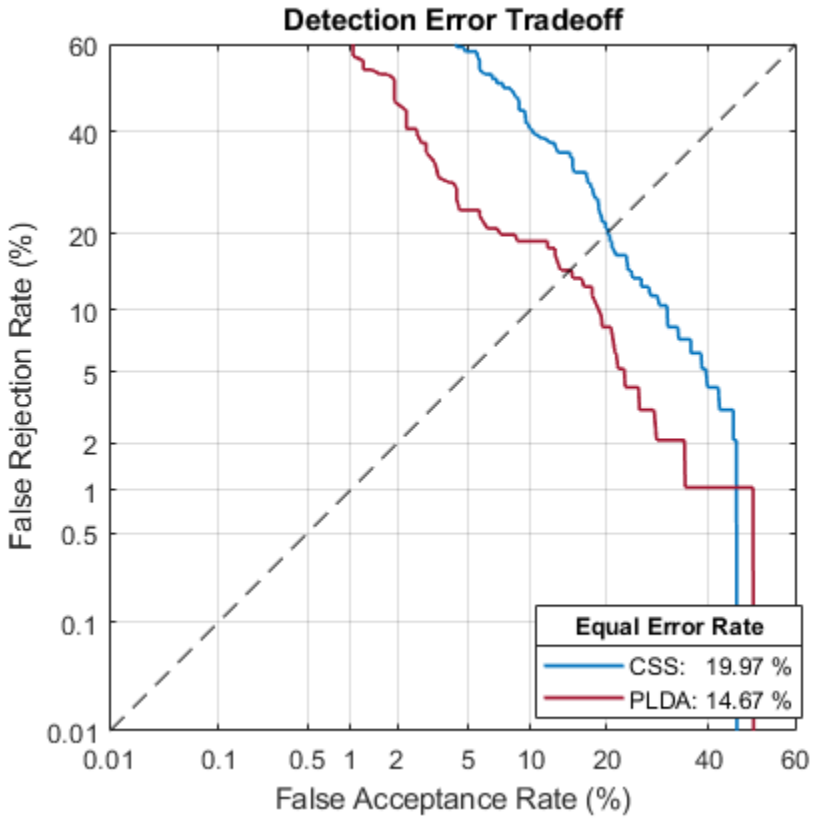
```
enroll(emotionRecognizer,trainSet,trainLabels)
```

```
Extracting i-vectors ...done.
Enrolling i-vectors ..........done.
Enrollment complete.
```

You can use `detectionErrorTradeoff` as a quick sanity check on the performance of a multilabel closed-set classification system. However, `detectionErrorTradeoff` provides information more suitable to open-set binary classification problems, for example, speaker verification tasks.

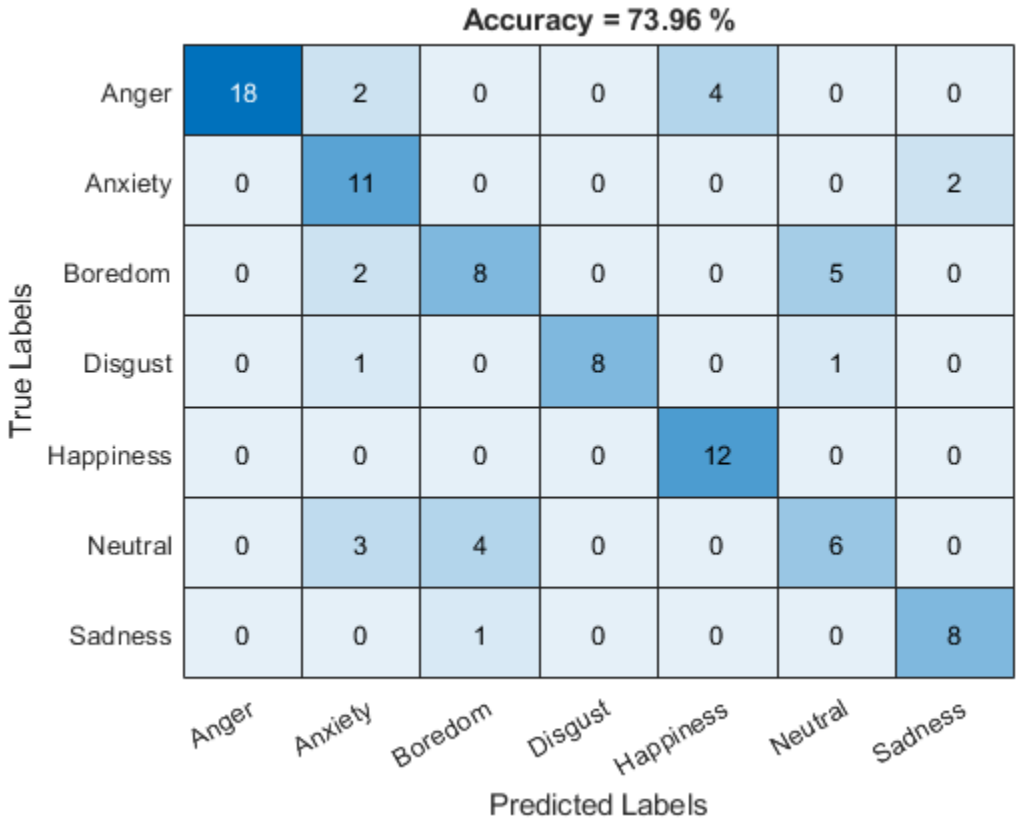
```
detectionErrorTradeoff(emotionRecognizer,testSet,testLabels)
```

```
Extracting i-vectors ...done.
Scoring i-vector pairs ...done.
Detection error tradeoff evaluation complete.
```



For a more detailed view of the i-vector system's performance in a multilabel closed set application, you can use the `identify` function and create a confusion matrix. The confusion matrix enables you to identify which emotions are misidentified and what they are misidentified as. Use the supporting function `plotConfusion` to display the results.

```
trueLabels = testLabels;
predictedLabels = trueLabels;

scorer = plda;
for ii = 1:numel(testSet)
    tableOut = identify(emotionRecognizer,testSet{ii},scorer);
    predictedLabels(ii) = tableOut.Label(1);
end

plotConfusion(trueLabels,predictedLabels)
```



Call `info` to inspect how `emotionRecognizer` was trained and evaluated.

```
info(emotionRecognizer)
```

```
i-vector system input
  Input feature vector length: 60
  Input data type: double

trainExtractor
  Train signals: 439
  UBMNumComponents: 256
  UBMNumIterations: 5
```

```
  TVSRank: 128
  TVSNumIterations: 5

trainClassifier
  Train signals: 439
  Train labels: Anger (103), Anxiety (56) ... and 5 more
  NumEigenvectors: 32
  PLDANumDimensions: 16
  PLDANumIterations: 10

calibrate
  Calibration signals: 439
  Calibration labels: Anger (103), Anxiety (56) ... and 5 more

detectionErrorTradeoff
  Evaluation signals: 96
  Evaluation labels: Anger (24), Anxiety (13) ... and 5 more
```

Next, modify the i-vector system to recognize emotions as positive, neutral, or negative. Update the labels to only include the categories negative, positive, and categorical.

```
trainLabelsSentiment = trainLabels;
trainLabelsSentiment(ismember(trainLabels,categorical(["Anger","Anxiety","Boredom","Sadness","Dis
trainLabelsSentiment(ismember(trainLabels,categorical("Happiness"))) = categorical("Postive");
trainLabelsSentiment = removecats(trainLabelsSentiment);

testLabelsSentiment = testLabels;
testLabelsSentiment(ismember(testLabels,categorical(["Anger","Anxiety","Boredom","Sadness","Disgu
testLabelsSentiment(ismember(testLabels,categorical("Happiness"))) = categorical("Postive");
testLabelsSentiment = removecats(testLabelsSentiment);
```

Train the i-vector system classifier using the updated labels. You do not need to retrain the extractor. Recalibrate the system.

```
rng default
trainClassifier(sentimentRecognizer,trainSet,trainLabelsSentiment, ...
    NumEigenvectors = 64 , ...
    ...
    PLDANumDimensions = 32 , ...
    PLDANumIterations = 10 );
```

```
Extracting i-vectors ...done.
Training projection matrix .....done.
Training PLDA model .............done.
i-vector classifier training complete.
```

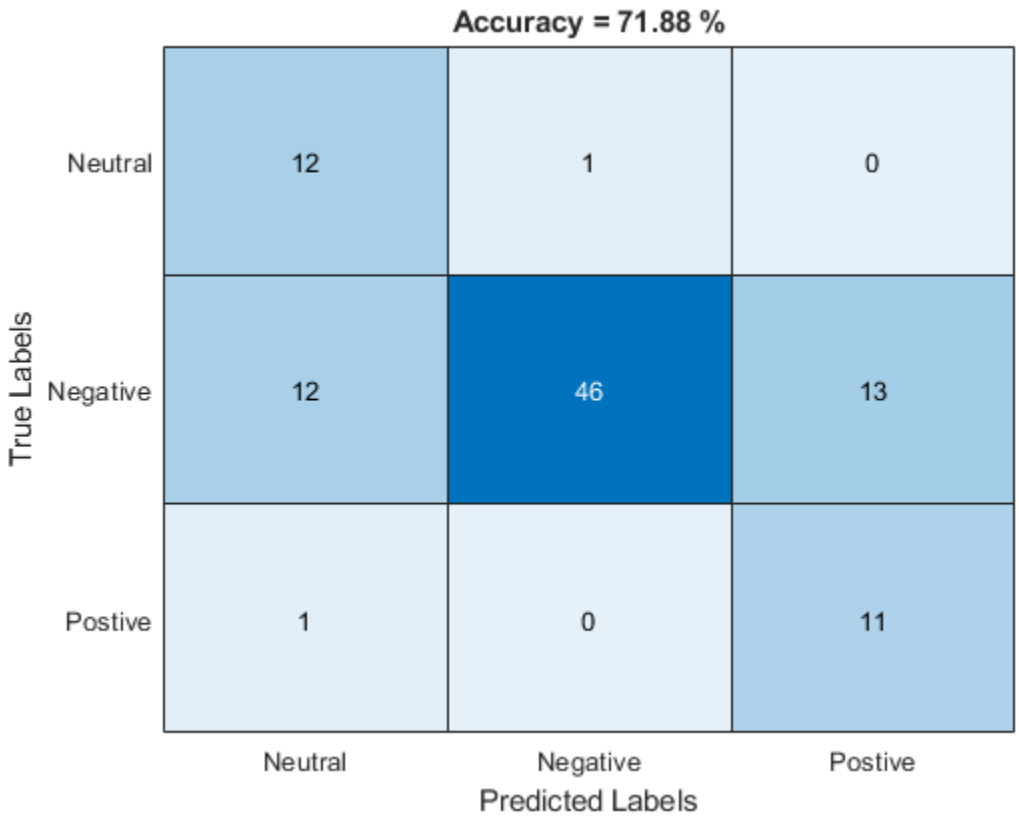
```
calibrate(sentimentRecognizer,trainSet,trainLabels)
```

```
Extracting i-vectors ...done.
Calibrating CSS scorer ...done.
Calibrating PLDA scorer ...done.
Calibration complete.
```

Enroll the training labels into the system and then plot the confusion matrix for the test set.

```
enroll(sentimentRecognizer,trainSet,trainLabelsSentiment)
```

```
Extracting i-vectors ...done.
Enrolling i-vectors ......done.
Enrollment complete.

trueLabels = testLabelsSentiment;
predictedLabels = trueLabels;
scorer = plda;
for ii = 1:numel(testSet)
    tableOut = identify(sentimentRecognizer,testSet{ii},scorer);
    predictedLabels(ii) = tableOut.Label(1);
end

plotConfusion(trueLabels,predictedLabels)
```

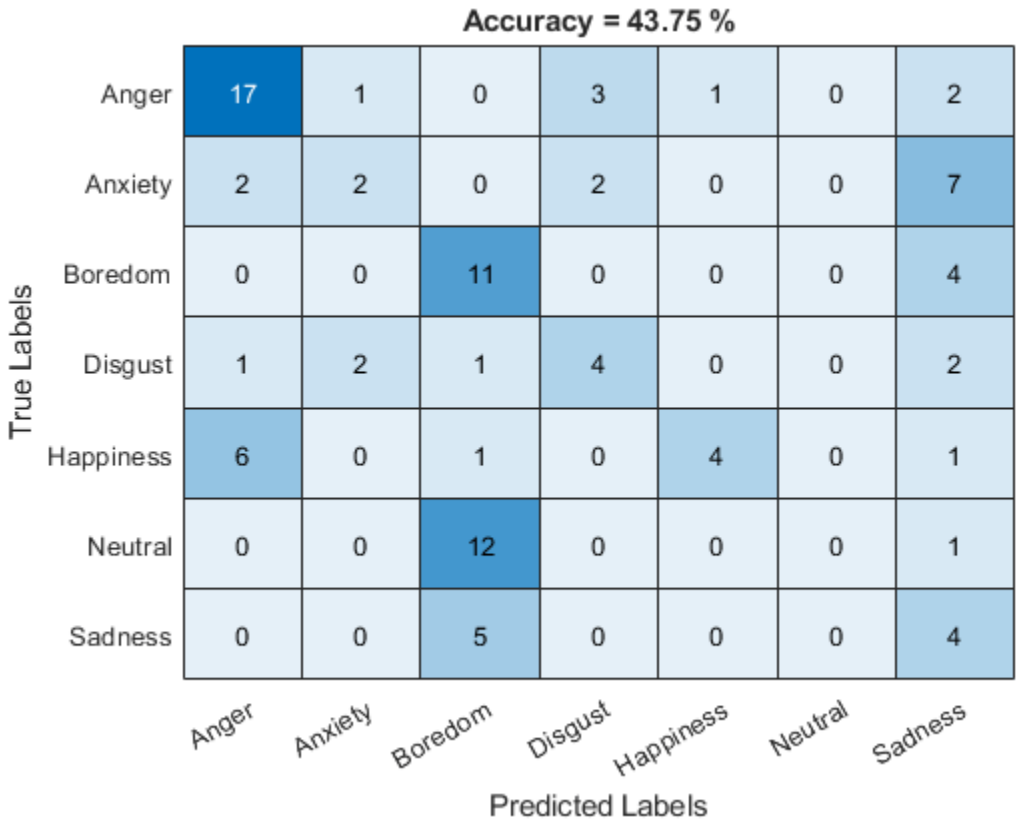


An i-vector system does not require the labels used to train the classifier to be equal to the enrolled labels.

Unenroll the sentiment labels from the system and then enroll the original emotion categories in the system. Analyze the system's classification performance.

```
unenroll(sentimentRecognizer)
enroll(sentimentRecognizer,trainSet,trainLabels)

Extracting i-vectors ...done.
Enrolling i-vectors ..........done.
Enrollment complete.
```

```
trueLabels = testLabels;
predictedLabels = trueLabels;
scorer = plda;
for ii = 1:numel(testSet)
    tableOut = identify(sentimentRecognizer,testSet{ii},scorer);
    predictedLabels(ii) = tableOut.Label(1);
end

plotConfusion(trueLabels,predictedLabels)
```



Supporting Functions

```
function plotConfusion(trueLabels,predictedLabels)
uniqueLabels = unique(trueLabels);
cm = zeros(numel(uniqueLabels),numel(uniqueLabels));
for ii = 1:numel(uniqueLabels)
    for jj = 1:numel(uniqueLabels)
        cm(ii,jj) = sum((trueLabels==uniqueLabels(ii)) & (predictedLabels==uniqueLabels(jj)));
    end
end

heatmap(uniqueLabels,uniqueLabels,cm)
colorbar off
ylabel('True Labels')
xlabel('Predicted Labels')
accuracy = mean(trueLabels==predictedLabels);
```

```
title(sprintf("Accuracy = %0.2f %%",accuracy*100))
end
```

References

[1] Burkhardt, F., A. Paeschke, M. Rolfes, W.F. Sendlmeier, and B. Weiss, "A Database of German Emotional Speech." In Proceedings Interspeech 2005. Lisbon, Portugal: International Speech Communication Association, 2005.

Input Arguments

`ivs` — i-vector system

`ivectorSystem` object

i-vector system, specified as an object of type `ivectorSystem`.

`labels` — Classification labels

character vector | string | cell array | string array | categorical array

Classification labels used by an i-vector system, specified as one of these:

- A categorical array
- A cell array of character vectors
- A string array

Data Types: `categorical` | `cell` | `string`

Version History

Introduced in R2021a

See Also

`trainExtractor` | `trainClassifier` | `calibrate` | `enroll` | `detectionErrorTradeoff` | `verify` | `identify` | `ivector` | `info` | `addInfoHeader` | `release` | `ivectorSystem` | `speakerRecognition`

detectionErrorTradeoff

Evaluate binary classification system

Syntax

```
results = detectionErrorTradeoff(ivs,data,labels)
results = detectionErrorTradeoff(ivs)

[results,threshold] = detectionErrorTradeoff( ___ )

[ ___ ] = detectionErrorTradeoff( ___ ,Name,Value)

detectionErrorTradeoff( ___ )
```

Description

`results = detectionErrorTradeoff(ivs,data,labels)` computes detection error tradeoff of the i-vector system `ivs` for the enrolled labels and the specified data.

`results = detectionErrorTradeoff(ivs)` returns a structure containing previously calculated results of threshold sweeping for probabilistic linear discriminant analysis (PLDA) and cosine similarity scoring (CSS).

`[results,threshold] = detectionErrorTradeoff( ___ )` also returns the threshold corresponding to the equal error rate.

`[ ___ ] = detectionErrorTradeoff( ___ ,Name,Value)` specifies additional options using name-value arguments. For example, you can choose the scorer results and the hardware resource for extracting i-vectors.

`detectionErrorTradeoff( ___ )` with no output arguments plots the equal error rate and the detection error tradeoff.

Examples

Train Speaker Verification System

Use the Pitch Tracking Database from Graz University of Technology (PTDB-TUG) [1] on page 4-378. The data set consists of 20 English native speakers reading 2342 phonetically rich sentences from the TIMIT corpus. Download and extract the data set.

```
downloadFolder = matlab.internal.examples.downloadSupportFile("audio","ptdb-tug.zip");
dataFolder = tempdir;
unzip(downloadFolder,dataFolder)
dataset = fullfile(dataFolder,"ptdb-tug");
```

Create an `audioDatastore` object that points to the data set. The data set was originally intended for use in pitch-tracking training and evaluation and includes laryngograph readings and baseline pitch decisions. Use only the original audio recordings.

```
ads = audioDatastore([fullfile(dataset,"SPEECH DATA","FEMALE","MIC"),fullfile(dataset,"SPEECH DA
                      IncludeSubfolders=true, ...
                      FileExtensions=".wav");
```

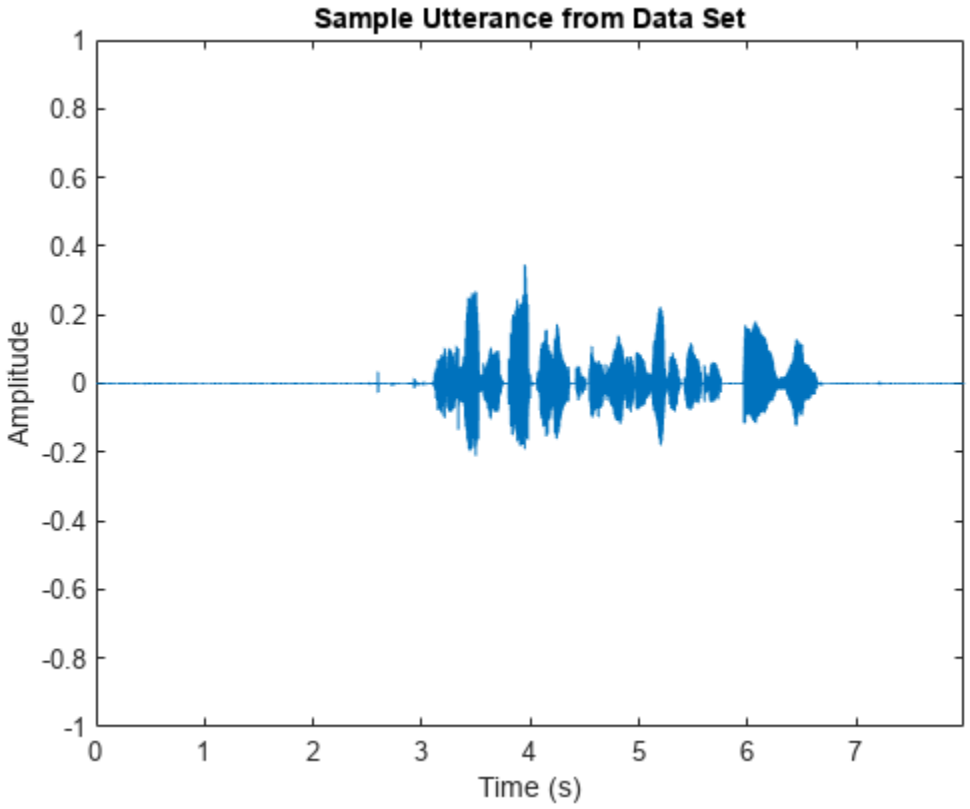
The file names contain the speaker IDs. Decode the file names to set the labels in the `audioDatastore` object.

```
ads.Labels = extractBetween(ads.Files,"mic_","_");
countEachLabel(ads)
```

```
ans=20×2 table
    Label    Count
    _____    _____

     F01      236
     F02      236
     F03      236
     F04      236
     F05      236
     F06      236
     F07      236
     F08      234
     F09      236
     F10      236
     M01      236
     M02      236
     M03      236
     M04      236
     M05      236
     M06      236
      ⋮
```

Read an audio file from the data set, listen to it, and plot it.

```
[audioIn,audioInfo] = read(ads);
fs = audioInfo.SampleRate;

t = (0:size(audioIn,1)-1)/fs;
sound(audioIn,fs)
plot(t,audioIn)
xlabel("Time (s)")
ylabel("Amplitude")
axis([0 t(end) -1 1])
title("Sample Utterance from Data Set")
```



Separate the `audioDatastore` object into four: one for training, one for enrollment, one to evaluate the detection-error tradeoff, and one for testing. The training set contains 16 speakers. The enrollment, detection-error tradeoff, and test sets contain the other four speakers.

```
speakersToTest = categorical(["M01","M05","F01","F05"]);

adsTrain = subset(ads,~ismember(ads.Labels,speakersToTest));

ads = subset(ads,ismember(ads.Labels,speakersToTest));
[adsEnroll,adsTest,adsDET] = splitEachLabel(ads,3,1);
```

Display the label distributions of the `audioDatastore` objects.

```
countEachLabel(adsTrain)
```

```
ans=16×2 table
    Label    Count
    _____    _____

     F02      236
     F03      236
     F04      236
     F06      236
     F07      236
     F08      234
     F09      236
     F10      236
     M02      236
```

```
     M03      236
     M04      236
     M06      236
     M07      236
     M08      236
     M09      236
     M10      236
```

```
countEachLabel(adsEnroll)
```

```
ans=4×2 table
    Label    Count
    _____    _____

     F01       3
     F05       3
     M01       3
     M05       3
```

```
countEachLabel(adsTest)
```

```
ans=4×2 table
    Label    Count
    _____    _____

     F01       1
     F05       1
     M01       1
     M05       1
```

```
countEachLabel(adsDET)
```

```
ans=4×2 table
    Label    Count
    _____    _____

     F01      232
     F05      232
     M01      232
     M05      232
```

Create an i-vector system. By default, the i-vector system assumes the input to the system is mono audio signals.

```
speakerVerification = ivectorSystem(SampleRate=fs)
```

```
speakerVerification =
  ivectorSystem with properties:

         InputType: 'audio'
        SampleRate: 48000
      DetectSpeech: 1
           Verbose: 1
    EnrolledLabels: [0×2 table]
```

To train the extractor of the i-vector system, call `trainExtractor`. Specify the number of universal background model (UBM) components as 128 and the number of expectation maximization iterations as 5. Specify the total variability space (TVS) rank as 64 and the number of iterations as 3.

```
trainExtractor(speakerVerification,adsTrain, ...
    UBMNumComponents=128,UBMNumIterations=5, ...
    TVSRank=64,TVSNumIterations=3)
```

```
Calculating standardization factors ....done.
Training universal background model ........done.
Training total variability space ......done.
i-vector extractor training complete.
```

To train the classifier of the i-vector system, use `trainClassifier`. To reduce dimensionality of the i-vectors, specify the number of eigenvectors in the projection matrix as 16. Specify the number of dimensions in the probabilistic linear discriminant analysis (PLDA) model as 16, and the number of iterations as 3.

```
trainClassifier(speakerVerification,adsTrain,adsTrain.Labels, ...
    NumEigenvectors=16, ...
    PLDANumDimensions=16,PLDANumIterations=3)
```

```
Extracting i-vectors ...done.
Training projection matrix .....done.
Training PLDA model ......done.
i-vector classifier training complete.
```

To calibrate the system so that scores can be interpreted as a measure of confidence in a positive decision, use `calibrate`.

```
calibrate(speakerVerification,adsTrain,adsTrain.Labels)
```

```
Extracting i-vectors ...done.
Calibrating CSS scorer ...done.
Calibrating PLDA scorer ...done.
Calibration complete.
```

To inspect parameters used previously to train the i-vector system, use `info`.

```
info(speakerVerification)
```

```
i-vector system input
  Input feature vector length: 60
  Input data type: double

trainExtractor
  Train signals: 3774
  UBMNumComponents: 128
  UBMNumIterations: 5
  TVSRank: 64
  TVSNumIterations: 3

trainClassifier
  Train signals: 3774
  Train labels: F02 (236), F03 (236) ... and 14 more
  NumEigenvectors: 16
  PLDANumDimensions: 16
  PLDANumIterations: 3
```

```
calibrate
  Calibration signals: 3774
  Calibration labels: F02 (236), F03 (236) ... and 14 more
```

Split the enrollment set.

```
[adsEnrollPart1,adsEnrollPart2] = splitEachLabel(adsEnroll,1,2);
```

To enroll speakers in the i-vector system, call `enroll`.

```
enroll(speakerVerification,adsEnrollPart1,adsEnrollPart1.Labels)
```

```
Extracting i-vectors ...done.
Enrolling i-vectors .......done.
Enrollment complete.
```

When you enroll speakers, the read-only `EnrolledLabels` property is updated with the enrolled labels and corresponding template i-vectors. The table also keeps track of the number of signals used to create the template i-vector. Generally, using more signals results in a better template.

```
speakerVerification.EnrolledLabels
```

```
ans=4×2 table
              ivector       NumSamples
            ____________    __________

    F01    {16×1 double}        1
    F05    {16×1 double}        1
    M01    {16×1 double}        1
    M05    {16×1 double}        1
```

Enroll the second part of the enrollment set and then view the enrolled labels table again. The i-vector templates and the number of samples are updated.

```
enroll(speakerVerification,adsEnrollPart2,adsEnrollPart2.Labels)
```

```
Extracting i-vectors ...done.
Enrolling i-vectors .......done.
Enrollment complete.
```

```
speakerVerification.EnrolledLabels
```

```
ans=4×2 table
              ivector       NumSamples
            ____________    __________

    F01    {16×1 double}        3
    F05    {16×1 double}        3
    M01    {16×1 double}        3
    M05    {16×1 double}        3
```

To evaluate the i-vector system and determine a decision threshold for speaker verification, call `detectionErrorTradeoff`.

```
[results, eerThreshold] = detectionErrorTradeoff(speakerVerification,adsDET,adsDET.Labels);
```

```
Extracting i-vectors ...done.
Scoring i-vector pairs ...done.
Detection error tradeoff evaluation complete.
```

The first output from `detectionErrorTradeoff` is a structure with two fields: CSS and PLDA. Each field contains a table. Each row of the table contains a possible decision threshold for speaker verification tasks, and the corresponding false alarm rate (FAR) and false rejection rate (FRR). The FAR and FRR are determined using the enrolled speaker labels and the data input to the `detectionErrorTradeoff` function.

`results`

```
results = struct with fields:
    PLDA: [1000×3 table]
     CSS: [1000×3 table]
```

`results.CSS`

```
ans=1000×3 table
    Threshold       FAR      FRR
    __________    _______    ___

    2.3259e-10          1     0
    2.3965e-10    0.99964     0
    2.4693e-10    0.99928     0
    2.5442e-10    0.99928     0
    2.6215e-10    0.99928     0
     2.701e-10    0.99928     0
     2.783e-10    0.99928     0
    2.8675e-10    0.99928     0
    2.9545e-10    0.99928     0
    3.0442e-10    0.99928     0
    3.1366e-10    0.99928     0
    3.2318e-10    0.99928     0
    3.3299e-10    0.99928     0
     3.431e-10    0.99928     0
    3.5352e-10    0.99928     0
    3.6425e-10    0.99892     0
      ⋮
```

`results.PLDA`

```
ans=1000×3 table
    Threshold       FAR      FRR
    __________    _______    ___

    3.2661e-40          1     0
    3.6177e-40    0.99964     0
    4.0072e-40    0.99964     0
    4.4387e-40    0.99964     0
    4.9166e-40    0.99964     0
    5.4459e-40    0.99964     0
    6.0322e-40    0.99964     0
    6.6817e-40    0.99964     0
    7.4011e-40    0.99964     0
     8.198e-40    0.99964     0
    9.0806e-40    0.99964     0
```

```
    1.0058e-39    0.99964     0
    1.1141e-39    0.99964     0
    1.2341e-39    0.99964     0
    1.3669e-39    0.99964     0
    1.5141e-39    0.99964     0
      ⋮
```

The second output from `detectionErrorTradeoff` is a structure with two fields: `CSS` and `PLDA`. The corresponding value is the decision threshold that results in the equal error rate (when FAR and FRR are equal).

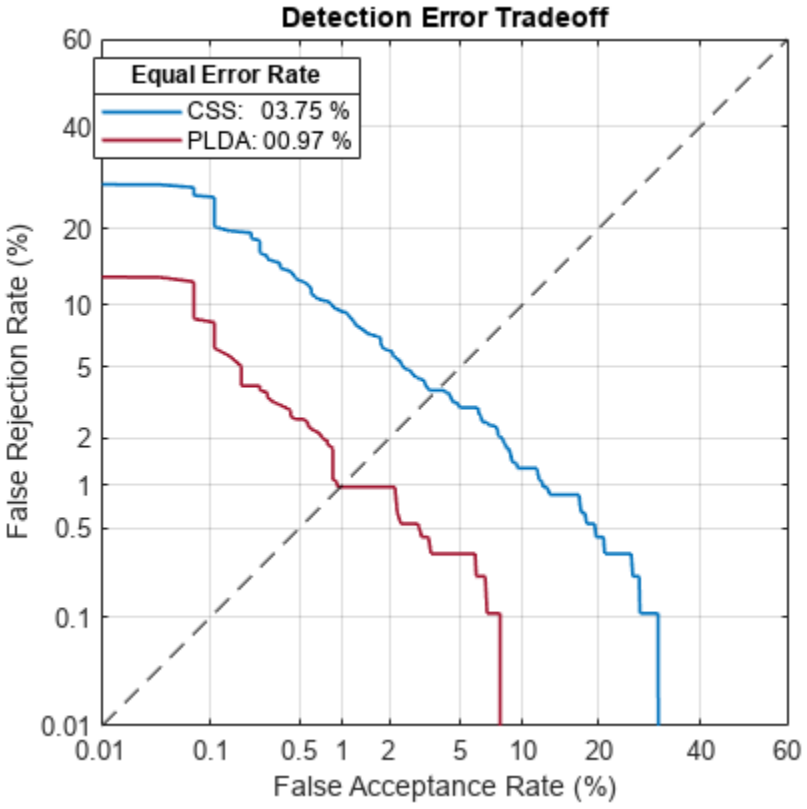
```
eerThreshold
```

```
eerThreshold = struct with fields:
    PLDA: 0.0398
     CSS: 0.9369
```

The first time you call `detectionErrorTradeoff`, you must provide data and corresponding labels to evaluate. Subsequently, you can get the same information, or a different analysis using the same underlying data, by calling `detectionErrorTradeoff` without data and labels.

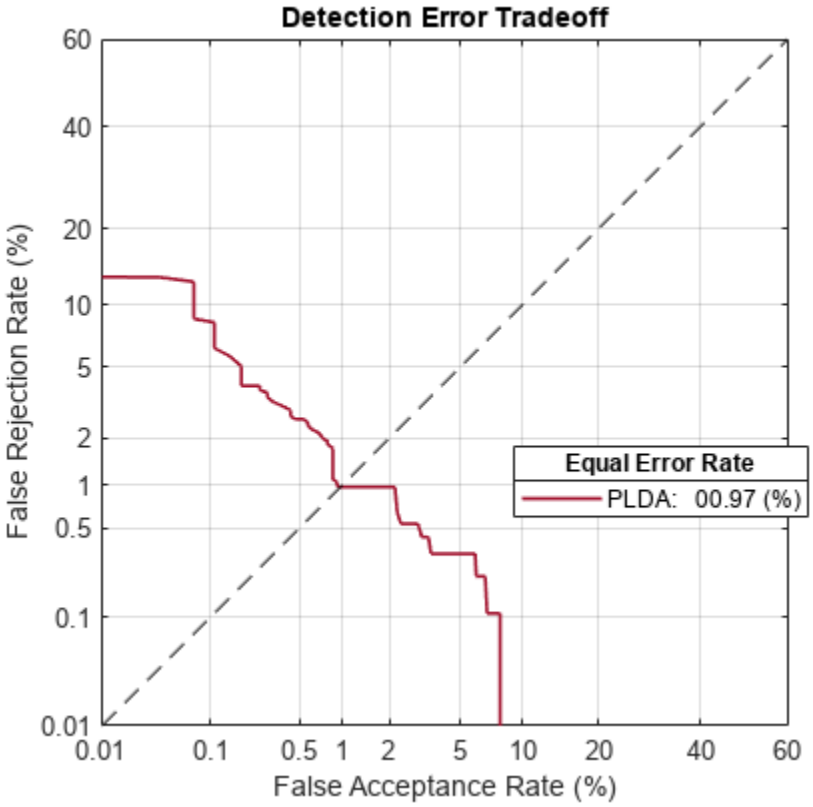
Call `detectionErrorTradeoff` a second time with no data arguments or output arguments to visualize the detection-error tradeoff.

```
detectionErrorTradeoff(speakerVerification)
```



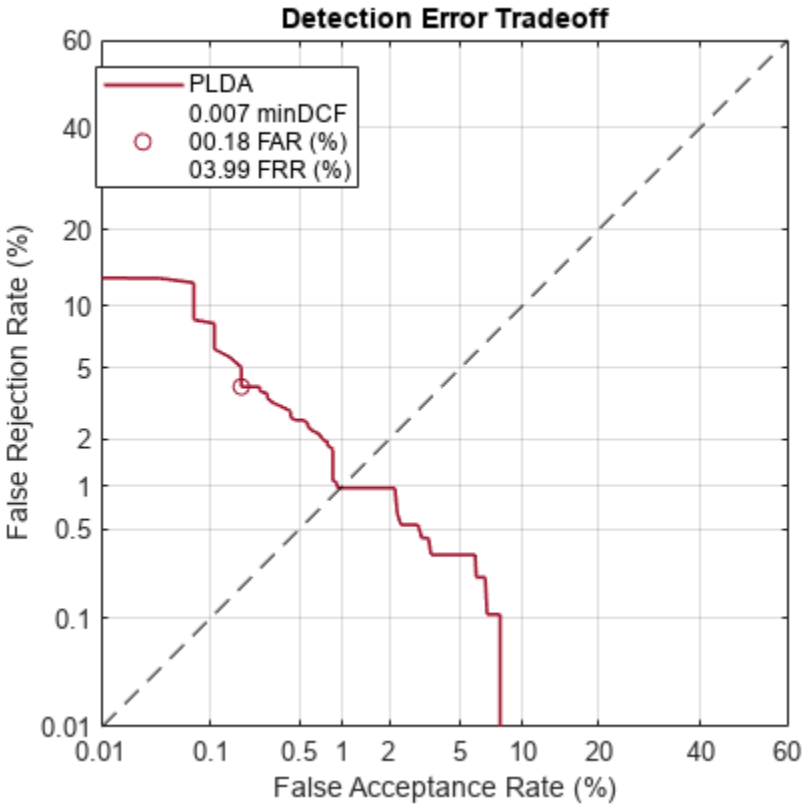
Call `detectionErrorTradeoff` again. This time, visualize only the detection-error tradeoff for the PLDA scorer.

```
detectionErrorTradeoff(speakerVerification,Scorer="plda")
```



Depending on your application, you may want to use a threshold that weights the error cost of a false alarm higher or lower than the error cost of a false rejection. You may also be using data that is not representative of the prior probability of the speaker being present. You can use the `minDCF` parameter to specify custom costs and prior probability. Call `detectionErrorTradeoff` again, this time specify the cost of a false rejection as 1, the cost of a false acceptance as 2, and the prior probability that a speaker is present as 0.1.

```
costFR = 1;
costFA = 2;
priorProb = 0.1;
detectionErrorTradeoff(speakerVerification,Scorer="plda",minDCF=[costFR,costFA,priorProb])
```



Call `detectionErrorTradeoff` again. This time, get the `minDCF` threshold for the PLDA scorer and the parameters of the detection cost function.

```
[~,minDCFThreshold] = detectionErrorTradeoff(speakerVerification,Scorer="plda",minDCF=[costFR,cos
```

```
minDCFThreshold = 0.4709
```

Test Speaker Verification System

Read a signal from the test set.

```
adsTest = shuffle(adsTest);
[audioIn,audioInfo] = read(adsTest);
knownSpeakerID = audioInfo.Label
```

```
knownSpeakerID = 1×1 cell array
    {'F01'}
```

To perform speaker verification, call `verify` with the audio signal and specify the speaker ID, a scorer, and a threshold for the scorer. The `verify` function returns a logical value indicating whether a speaker identity is accepted or rejected, and a score indicating the similarity of the input audio and the template i-vector corresponding to the enrolled label.

```
[tf,score] = verify(speakerVerification,audioIn,knownSpeakerID,"plda",eerThreshold.PLDA);
if tf
    fprintf('Success!\nSpeaker accepted.\nSimilarity score = %0.2f\n\n',score)
else
```

```
    fprinf('Failure!\nSpeaker rejected.\nSimilarity score = %0.2f\n\n',score)
end
```

```
Success!
Speaker accepted.
Similarity score = 1.00
```

Call speaker verification again. This time, specify an incorrect speaker ID.

```
possibleSpeakers = speakerVerification.EnrolledLabels.Properties.RowNames;
imposterIdx = find(~ismember(possibleSpeakers,knownSpeakerID));
imposter = possibleSpeakers(imposterIdx(randperm(numel(imposterIdx),1)))
```

```
imposter = 1×1 cell array
    {'M05'}
```

```
[tf,score] = verify(speakerVerification,audioIn,imposter,"plda",eerThreshold.PLDA);
if tf
    fprintf('Failure!\nSpeaker accepted.\nSimilarity score = %0.2f\n\n',score)
else
    fprintf('Success!\nSpeaker rejected.\nSimilarity score = %0.2f\n\n',score)
end
```

```
Success!
Speaker rejected.
Similarity score = 0.00
```

References

[1] Signal Processing and Speech Communication Laboratory. https://www.spsc.tugraz.at/databases-and-tools/ptdb-tug-pitch-tracking-database-from-graz-university-of-technology.html. Accessed 12 Dec. 2019.

Input Arguments

`ivs` — i-vector system

`ivectorSystem` object

i-vector system, specified as an object of type `ivectorSystem`.

`data` — Labeled evaluation data

cell array | `audioDatastore` | `signalDatastore` | `TransformedDatastore`

Labeled evaluation data, specified as a cell array or as an `audioDatastore`, `signalDatastore`, or `TransformedDatastore` object.

- If `InputType` is set to `"audio"` when the i-vector system is created, specify `data` as one of these:
  - A cell array of single-channel audio signals, each specified as a column vector with underlying type `single` or `double`.
  - An `audioDatastore` object or a `signalDatastore` object that points to a data set of mono audio signals.
  - A `TransformedDatastore` with an underlying `audioDatastore` or `signalDatastore` that points to a data set of mono audio signals. The output from calls to `read` from the transform datastore must be mono audio signals with underlying data type `single` or `double`.

- If `InputType` is set to `"features"` when the i-vector system is created, specify `data` as one of these:
  - A cell array of matrices with underlying type `single` or `double`. The matrices must consist of audio features where the number of features (columns) is locked the first time `trainExtractor` is called and the number of hops (rows) is variable-sized. The number of features input in any subsequent calls to any of the object functions must be equal to the number of features used when calling `trainExtractor`.
  - A `TransformedDatastore` object with an underlying `audioDatastore` or `signalDatastore` whose `read` function has output as described in the previous bullet.
  - A `signalDatastore` object whose `read` function has output as described in the first bullet.

Data Types: `cell` | `audioDatastore` | `signalDatastore`

`labels` — Classification labels
categorical array | cell array | string array

Classification labels used by an i-vector system, specified as one of the following:

- A categorical array
- A cell array of character vectors
- A string array

Note The number of audio signals in `data` must match the number of `labels`.

Data Types: `categorical` | `cell` | `string`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `detectionErrorTradeoff(ivs,Scorer="css")`

`Scorer` — Scorer results
`"all"` (default) | `"plda"` | `"css"`

Scorer results returned from an i-vector system, specified as `"plda"`, which corresponds to probabilistic linear discriminant analysis (PLDA), `"css"`, which corresponds to cosine similarity score (CSS), or `"all"`.

Data Types: `char` | `string`

`minDCF` — Parameters of detection cost function
three-element vector

Parameters of the detection cost function, specified as a three-element vector consisting of the cost of a false rejection, the cost of a false acceptance, and the prior probability of an enrolled label being present, in that order.

When you specify parameters of a detection cost function, `detectionErrorTradeoff` returns the `threshold` corresponding to the minimum of the detection cost function [1]. The detection cost function is defined as

$$C_{\text{det}}(P_{\text{FR}},P_{\text{FA}}) = C_{\text{FR}} \times P_{\text{FR}} \times P_{\text{present}} + C_{\text{FA}} \times P_{\text{FA}} \times (1 - P_{\text{present}}),$$

where

- C_{det} — Detection cost function
- C_{FR} — Cost of a false rejection
- C_{FA} — Cost of a false acceptance
- P_{present} — Prior probability of an enrolled label being present
- P_{FR} — Observed probability of a false rejection given the `data` input to `detectionErrorTradeoff`
- P_{FA} — Observed probability of a false acceptance given the `data` input to `detectionErrorTradeoff`

Data Types: `single` | `double`

`ExecutionEnvironment` — Hardware resource for execution
`"auto"` (default) | `"cpu"` | `"gpu"` | `"multi-gpu"` | `"parallel"`

Hardware resource for execution, specified as one of these:

- `"auto"` — Use the GPU if it is available. Otherwise, use the CPU.
- `"cpu"` — Use the CPU.
- `"gpu"` — Use the GPU. This option requires Parallel Computing Toolbox.
- `"multi-gpu"` — Use multiple GPUs on one machine, using a local parallel pool based on your default cluster profile. If there is no current parallel pool, the software starts a parallel pool with pool size equal to the number of available GPUs. This option requires Parallel Computing Toolbox.
- `"parallel"` — Use a local or remote parallel pool based on your default cluster profile. If there is no current parallel pool, the software starts one using the default cluster profile. If the pool has access to GPUs, then only workers with a unique GPU perform training computation. If the pool does not have GPUs, then the training takes place on all available CPU workers. This option requires Parallel Computing Toolbox.

Data Types: `char` | `string`

`DispatchInBackground` — Option to use prefetch queuing
`false` (default) | `true`

Option to use prefetch queuing when reading from a datastore, specified as a logical value. This argument requires Parallel Computing Toolbox.

Data Types: `logical`

Output Arguments

`results` — FAR and FRR per threshold tested
structure | table

FAR and FRR per threshold tested, returned as a structure or a table.

- If `Scorer` is specified as `"all"`, then `results` is returned as a structure with fields `PLDA` and `CSS` and values containing tables. Each table has three variables: `Threshold`, `FAR`, and `FRR`.
- If `Scorer` is specified as `"plda"` or `"css"`, then `results` is returned as a table corresponding to the specified scorer.

Data Types: `struct` | `table`

`threshold` — Threshold corresponding to equal error rate
scalar | structure

Threshold corresponding to the equal error rate (EER) or minimum of the detection cost function (minDCF), returned as a scalar or a structure. If the `minDCF` is specified, then `threshold` corresponds to `minDCF`. Otherwise, `threshold` corresponds to the EER.

- If `Scorer` is specified as `"all"`, then `threshold` is returned as a structure with fields `PLDA` and `CSS` and values equal to the respective thresholds.
- If `Scorer` is specified as `"plda"` or `"css"`, then `threshold` is returned as a scalar corresponding to the specified scorer.

Data Types: `single` | `double` | `struct`

Version History

Introduced in R2021a

References

[1] Leeuwen, David A. van, and Niko Brümmer. “An Introduction to Application-Independent Evaluation of Speaker Recognition Systems.” In *Speaker Classification I*, edited by Christian Müller, 4343:330–53. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007. https://doi.org/10.1007/978-3-540-74200-5_19.

See Also

`trainExtractor` | `trainClassifier` | `calibrate` | `unenroll` | `enroll` | `verify` | `identify` | `ivector` | `info` | `addInfoHeader` | `release` | `ivectorSystem` | `speakerRecognition`

verify

Verify label

Syntax

```
tf = verify(ivs,data,label)
tf = verify(ivs,data,label,scorer)
tf = verify(ivs,data,label,scorer,threshold)

[tf,score] = verify( ___ )
```

Description

`tf = verify(ivs,data,label)` returns `true` if i-vector system `ivs` finds that `data` corresponds to `label` and `false` otherwise.

`tf = verify(ivs,data,label,scorer)` specifies the scorer used for verification.

`tf = verify(ivs,data,label,scorer,threshold)` specifies the decision threshold used for the score.

`[tf,score] = verify( ___ )` also returns a `score` indicating the similarity between the i-vector derived from the `data` and the i-vector template corresponding to the `label`.

Examples

Train Speaker Verification System

Use the Pitch Tracking Database from Graz University of Technology (PTDB-TUG) [1] on page 4-392. The data set consists of 20 English native speakers reading 2342 phonetically rich sentences from the TIMIT corpus. Download and extract the data set.

```
downloadFolder = matlab.internal.examples.downloadSupportFile("audio","ptdb-tug.zip");
dataFolder = tempdir;
unzip(downloadFolder,dataFolder)
dataset = fullfile(dataFolder,"ptdb-tug");
```

Create an `audioDatastore` object that points to the data set. The data set was originally intended for use in pitch-tracking training and evaluation and includes laryngograph readings and baseline pitch decisions. Use only the original audio recordings.

```
ads = audioDatastore([fullfile(dataset,"SPEECH DATA","FEMALE","MIC"),fullfile(dataset,"SPEECH DA
                      IncludeSubfolders=true, ...
                      FileExtensions=".wav");
```

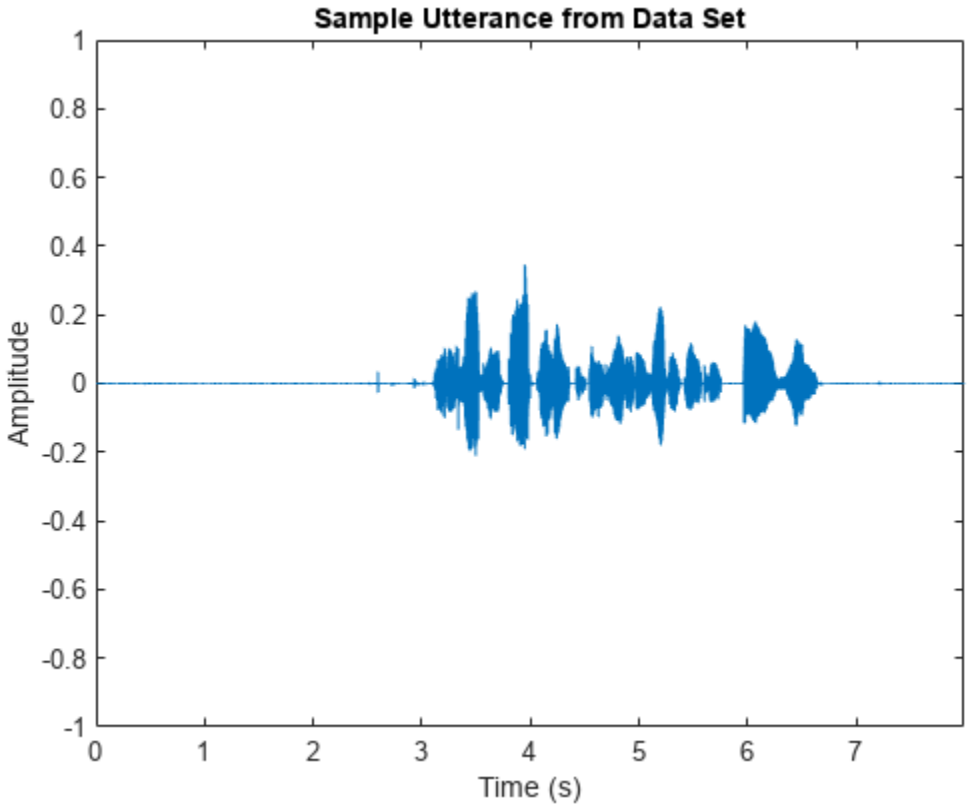
The file names contain the speaker IDs. Decode the file names to set the labels in the `audioDatastore` object.

```
ads.Labels = extractBetween(ads.Files,"mic_","_");
countEachLabel(ads)
```

```
ans=20×2 table
    Label    Count
    _____    _____

     F01      236
     F02      236
     F03      236
     F04      236
     F05      236
     F06      236
     F07      236
     F08      234
     F09      236
     F10      236
     M01      236
     M02      236
     M03      236
     M04      236
     M05      236
     M06      236
      ⋮
```

Read an audio file from the data set, listen to it, and plot it.

```
[audioIn,audioInfo] = read(ads);
fs = audioInfo.SampleRate;

t = (0:size(audioIn,1)-1)/fs;
sound(audioIn,fs)
plot(t,audioIn)
xlabel("Time (s)")
ylabel("Amplitude")
axis([0 t(end) -1 1])
title("Sample Utterance from Data Set")
```



Separate the `audioDatastore` object into four: one for training, one for enrollment, one to evaluate the detection-error tradeoff, and one for testing. The training set contains 16 speakers. The enrollment, detection-error tradeoff, and test sets contain the other four speakers.

```
speakersToTest = categorical(["M01","M05","F01","F05"]);

adsTrain = subset(ads,~ismember(ads.Labels,speakersToTest));

ads = subset(ads,ismember(ads.Labels,speakersToTest));
[adsEnroll,adsTest,adsDET] = splitEachLabel(ads,3,1);
```

Display the label distributions of the `audioDatastore` objects.

```
countEachLabel(adsTrain)
```

```
ans=16×2 table
    Label    Count
    _____    _____

     F02      236
     F03      236
     F04      236
     F06      236
     F07      236
     F08      234
     F09      236
     F10      236
     M02      236
```

```
     M03      236
     M04      236
     M06      236
     M07      236
     M08      236
     M09      236
     M10      236
```

```
countEachLabel(adsEnroll)
```

```
ans=4×2 table
    Label    Count
    _____    _____

     F01       3
     F05       3
     M01       3
     M05       3
```

```
countEachLabel(adsTest)
```

```
ans=4×2 table
    Label    Count
    _____    _____

     F01       1
     F05       1
     M01       1
     M05       1
```

```
countEachLabel(adsDET)
```

```
ans=4×2 table
    Label    Count
    _____    _____

     F01      232
     F05      232
     M01      232
     M05      232
```

Create an i-vector system. By default, the i-vector system assumes the input to the system is mono audio signals.

```
speakerVerification = ivectorSystem(SampleRate=fs)
```

```
speakerVerification =
  ivectorSystem with properties:

         InputType: 'audio'
        SampleRate: 48000
      DetectSpeech: 1
           Verbose: 1
    EnrolledLabels: [0×2 table]
```

To train the extractor of the i-vector system, call `trainExtractor`. Specify the number of universal background model (UBM) components as 128 and the number of expectation maximization iterations as 5. Specify the total variability space (TVS) rank as 64 and the number of iterations as 3.

```
trainExtractor(speakerVerification,adsTrain, ...
    UBMNumComponents=128,UBMNumIterations=5, ...
    TVSRank=64,TVSNumIterations=3)
```

```
Calculating standardization factors ....done.
Training universal background model ........done.
Training total variability space ......done.
i-vector extractor training complete.
```

To train the classifier of the i-vector system, use `trainClassifier`. To reduce dimensionality of the i-vectors, specify the number of eigenvectors in the projection matrix as 16. Specify the number of dimensions in the probabilistic linear discriminant analysis (PLDA) model as 16, and the number of iterations as 3.

```
trainClassifier(speakerVerification,adsTrain,adsTrain.Labels, ...
    NumEigenvectors=16, ...
    PLDANumDimensions=16,PLDANumIterations=3)
```

```
Extracting i-vectors ...done.
Training projection matrix .....done.
Training PLDA model ......done.
i-vector classifier training complete.
```

To calibrate the system so that scores can be interpreted as a measure of confidence in a positive decision, use `calibrate`.

```
calibrate(speakerVerification,adsTrain,adsTrain.Labels)
```

```
Extracting i-vectors ...done.
Calibrating CSS scorer ...done.
Calibrating PLDA scorer ...done.
Calibration complete.
```

To inspect parameters used previously to train the i-vector system, use `info`.

```
info(speakerVerification)
```

```
i-vector system input
  Input feature vector length: 60
  Input data type: double

trainExtractor
  Train signals: 3774
  UBMNumComponents: 128
  UBMNumIterations: 5
  TVSRank: 64
  TVSNumIterations: 3

trainClassifier
  Train signals: 3774
  Train labels: F02 (236), F03 (236) ... and 14 more
  NumEigenvectors: 16
  PLDANumDimensions: 16
  PLDANumIterations: 3
```

```
calibrate
  Calibration signals: 3774
  Calibration labels: F02 (236), F03 (236) ... and 14 more
```

Split the enrollment set.

```
[adsEnrollPart1,adsEnrollPart2] = splitEachLabel(adsEnroll,1,2);
```

To enroll speakers in the i-vector system, call `enroll`.

```
enroll(speakerVerification,adsEnrollPart1,adsEnrollPart1.Labels)
```

```
Extracting i-vectors ...done.
Enrolling i-vectors .......done.
Enrollment complete.
```

When you enroll speakers, the read-only `EnrolledLabels` property is updated with the enrolled labels and corresponding template i-vectors. The table also keeps track of the number of signals used to create the template i-vector. Generally, using more signals results in a better template.

```
speakerVerification.EnrolledLabels
```

```
ans=4×2 table
              ivector       NumSamples
            ____________    __________

    F01    {16×1 double}        1
    F05    {16×1 double}        1
    M01    {16×1 double}        1
    M05    {16×1 double}        1
```

Enroll the second part of the enrollment set and then view the enrolled labels table again. The i-vector templates and the number of samples are updated.

```
enroll(speakerVerification,adsEnrollPart2,adsEnrollPart2.Labels)
```

```
Extracting i-vectors ...done.
Enrolling i-vectors .......done.
Enrollment complete.
```

```
speakerVerification.EnrolledLabels
```

```
ans=4×2 table
              ivector       NumSamples
            ____________    __________

    F01    {16×1 double}        3
    F05    {16×1 double}        3
    M01    {16×1 double}        3
    M05    {16×1 double}        3
```

To evaluate the i-vector system and determine a decision threshold for speaker verification, call `detectionErrorTradeoff`.

```
[results, eerThreshold] = detectionErrorTradeoff(speakerVerification,adsDET,adsDET.Labels);
```

```
Extracting i-vectors ...done.
Scoring i-vector pairs ...done.
Detection error tradeoff evaluation complete.
```

The first output from `detectionErrorTradeoff` is a structure with two fields: CSS and PLDA. Each field contains a table. Each row of the table contains a possible decision threshold for speaker verification tasks, and the corresponding false alarm rate (FAR) and false rejection rate (FRR). The FAR and FRR are determined using the enrolled speaker labels and the data input to the `detectionErrorTradeoff` function.

`results`

```
results = struct with fields:
    PLDA: [1000×3 table]
     CSS: [1000×3 table]
```

`results.CSS`

```
ans=1000×3 table
    Threshold       FAR      FRR
    __________    _______    ___

    2.3259e-10          1     0
    2.3965e-10    0.99964     0
    2.4693e-10    0.99928     0
    2.5442e-10    0.99928     0
    2.6215e-10    0.99928     0
     2.701e-10    0.99928     0
     2.783e-10    0.99928     0
    2.8675e-10    0.99928     0
    2.9545e-10    0.99928     0
    3.0442e-10    0.99928     0
    3.1366e-10    0.99928     0
    3.2318e-10    0.99928     0
    3.3299e-10    0.99928     0
     3.431e-10    0.99928     0
    3.5352e-10    0.99928     0
    3.6425e-10    0.99892     0
      ⋮
```

`results.PLDA`

```
ans=1000×3 table
    Threshold       FAR      FRR
    __________    _______    ___

    3.2661e-40          1     0
    3.6177e-40    0.99964     0
    4.0072e-40    0.99964     0
    4.4387e-40    0.99964     0
    4.9166e-40    0.99964     0
    5.4459e-40    0.99964     0
    6.0322e-40    0.99964     0
    6.6817e-40    0.99964     0
    7.4011e-40    0.99964     0
     8.198e-40    0.99964     0
    9.0806e-40    0.99964     0
```

```
    1.0058e-39    0.99964     0
    1.1141e-39    0.99964     0
    1.2341e-39    0.99964     0
    1.3669e-39    0.99964     0
    1.5141e-39    0.99964     0
      ⋮
```

The second output from `detectionErrorTradeoff` is a structure with two fields: `CSS` and `PLDA`. The corresponding value is the decision threshold that results in the equal error rate (when FAR and FRR are equal).

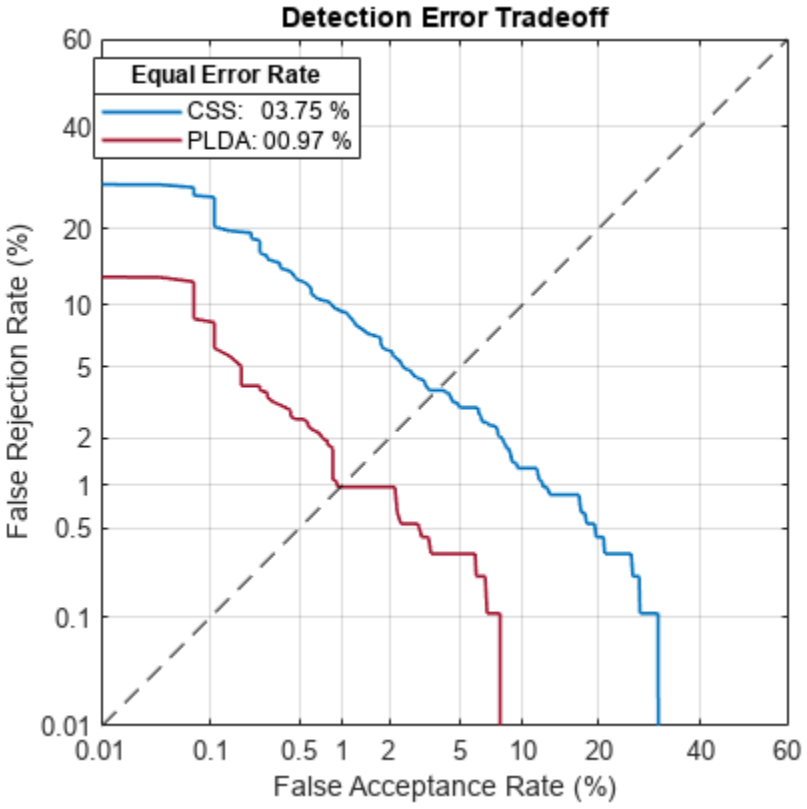
```
eerThreshold
```

```
eerThreshold = struct with fields:
    PLDA: 0.0398
     CSS: 0.9369
```

The first time you call `detectionErrorTradeoff`, you must provide data and corresponding labels to evaluate. Subsequently, you can get the same information, or a different analysis using the same underlying data, by calling `detectionErrorTradeoff` without data and labels.

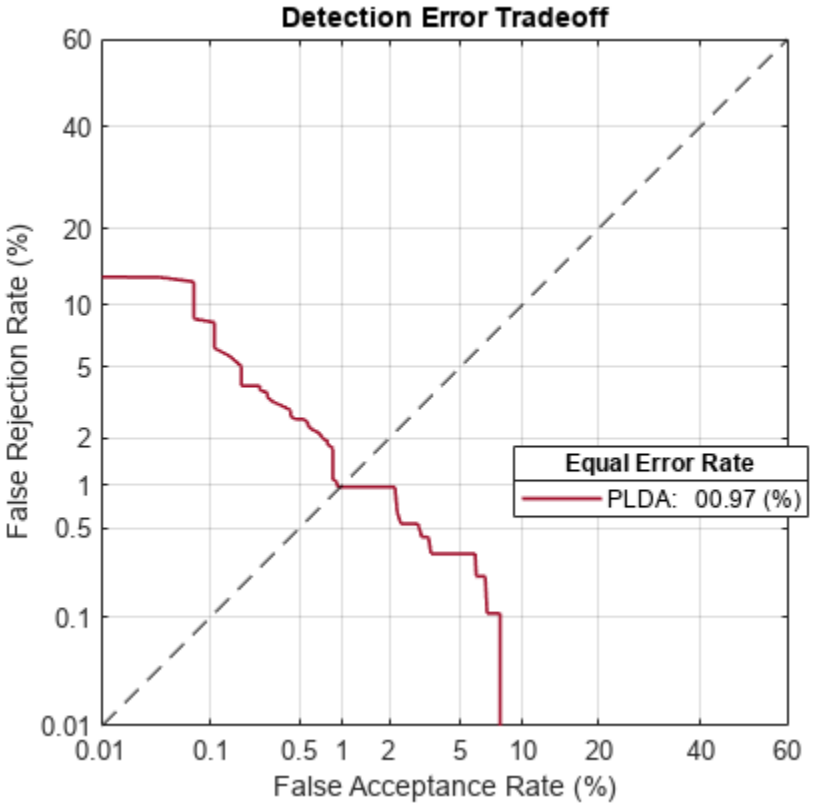
Call `detectionErrorTradeoff` a second time with no data arguments or output arguments to visualize the detection-error tradeoff.

```
detectionErrorTradeoff(speakerVerification)
```



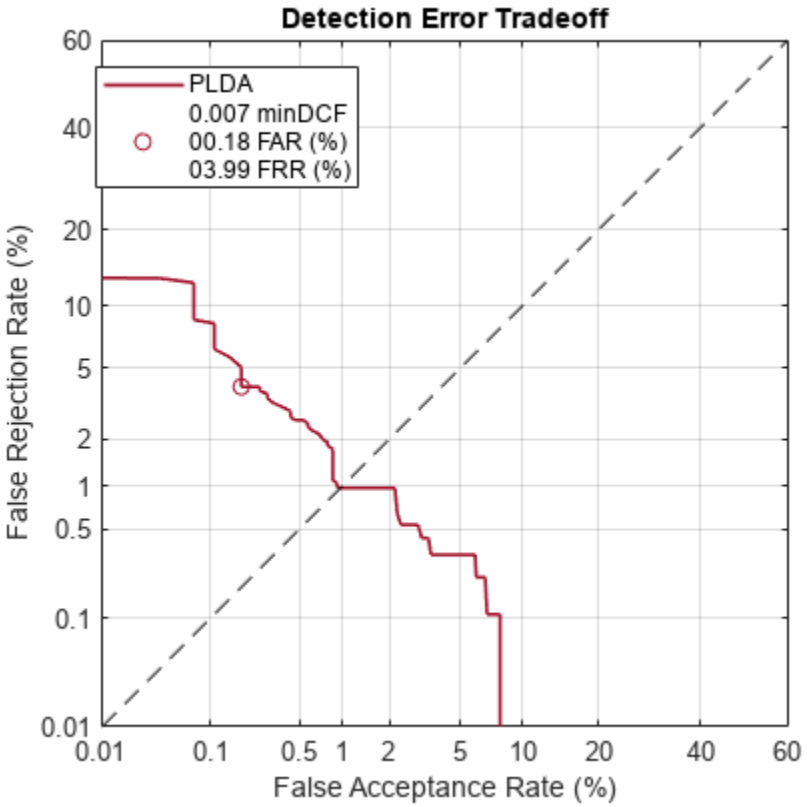
Call `detectionErrorTradeoff` again. This time, visualize only the detection-error tradeoff for the PLDA scorer.

```
detectionErrorTradeoff(speakerVerification,Scorer="plda")
```



Depending on your application, you may want to use a threshold that weights the error cost of a false alarm higher or lower than the error cost of a false rejection. You may also be using data that is not representative of the prior probability of the speaker being present. You can use the `minDCF` parameter to specify custom costs and prior probability. Call `detectionErrorTradeoff` again, this time specify the cost of a false rejection as 1, the cost of a false acceptance as 2, and the prior probability that a speaker is present as 0.1.

```
costFR = 1;
costFA = 2;
priorProb = 0.1;
detectionErrorTradeoff(speakerVerification,Scorer="plda",minDCF=[costFR,costFA,priorProb])
```



Call `detectionErrorTradeoff` again. This time, get the `minDCF` threshold for the PLDA scorer and the parameters of the detection cost function.

```
[~,minDCFThreshold] = detectionErrorTradeoff(speakerVerification,Scorer="plda",minDCF=[costFR,cos
```

```
minDCFThreshold = 0.4709
```

Test Speaker Verification System

Read a signal from the test set.

```
adsTest = shuffle(adsTest);
[audioIn,audioInfo] = read(adsTest);
knownSpeakerID = audioInfo.Label
```

```
knownSpeakerID = 1×1 cell array
    {'F01'}
```

To perform speaker verification, call `verify` with the audio signal and specify the speaker ID, a scorer, and a threshold for the scorer. The `verify` function returns a logical value indicating whether a speaker identity is accepted or rejected, and a score indicating the similarity of the input audio and the template i-vector corresponding to the enrolled label.

```
[tf,score] = verify(speakerVerification,audioIn,knownSpeakerID,"plda",eerThreshold.PLDA);
if tf
    fprintf('Success!\nSpeaker accepted.\nSimilarity score = %0.2f\n\n',score)
else
```

```
    fprinf('Failure!\nSpeaker rejected.\nSimilarity score = %0.2f\n\n',score)
end
```

```
Success!
Speaker accepted.
Similarity score = 1.00
```

Call speaker verification again. This time, specify an incorrect speaker ID.

```
possibleSpeakers = speakerVerification.EnrolledLabels.Properties.RowNames;
imposterIdx = find(~ismember(possibleSpeakers,knownSpeakerID));
imposter = possibleSpeakers(imposterIdx(randperm(numel(imposterIdx),1)))
```

```
imposter = 1×1 cell array
    {'M05'}
```

```
[tf,score] = verify(speakerVerification,audioIn,imposter,"plda",eerThreshold.PLDA);
if tf
    fprintf('Failure!\nSpeaker accepted.\nSimilarity score = %0.2f\n\n',score)
else
    fprintf('Success!\nSpeaker rejected.\nSimilarity score = %0.2f\n\n',score)
end
```

```
Success!
Speaker rejected.
Similarity score = 0.00
```

References

[1] Signal Processing and Speech Communication Laboratory. https://www.spsc.tugraz.at/databases-and-tools/ptdb-tug-pitch-tracking-database-from-graz-university-of-technology.html. Accessed 12 Dec. 2019.

Input Arguments

`ivs` — i-vector system

`ivectorSystem` object

i-vector system, specified as an object of type `ivectorSystem`.

`data` — Data to verify

column vector | matrix

Data to verify, specified as a column vector representing a single-channel (mono) audio signal or a matrix of audio features.

- If `InputType` is set to `"audio"` when the i-vector system is created, `data` must be a column vector with underlying type `single` or `double`.
- If `InputType` is set to `"features"` when the i-vector system is created, `data` must be a matrix with underlying type `single` or `double`. The matrix must consist of audio features where the number of features (columns) is locked the first time `trainExtractor` is called and the number of hops (rows) is variable-sized.

Data Types: `single` | `double`

`label` — Label to verify

categorical scalar | character vector | string

Label to verify, specified as a categorical scalar, a character vector, or a string.

Data Types: `categorical` | `char` | `string`

`scorer` — Scoring algorithm

`"plda"` | `"css"`

Scoring algorithm used by the i-vector system, specified as `"plda"`, which corresponds to probabilistic linear discriminant analysis (PLDA), or `"css"`, which corresponds to cosine similarity score (CSS). If the PLDA model was trained by `trainClassifier`, the default scorer is `"plda"`. Otherwise, the default scorer is `"css"`.

Data Types: `char` | `string`

`threshold` — Decision threshold

scalar

Decision threshold applied to the similarity score, specified as a scalar. The default decision threshold is the equal error rate of the scorer determined by calling `detectionErrorTradeoff`. If `detectionErrorTradeoff` is not called, then you must define the threshold.

Data Types: `single` | `double`

Output Arguments

`tf` — Correspondence indicator

`true` | `false`

Correspondence indicator, returned as a logical.

- If the i-vector system finds that `data` corresponds to `label`, `tf` is returned as `true`.
- If the i-vector system finds that `data` does not correspond to `label`, `tf` is returned as `false`.

Data Types: `logical`

`score` — Similarity score

scalar

Score indicating the similarity between the i-vector derived from `data` and the i-vector corresponding to `label`, returned as a scalar.

Data Types: `double`

Version History

Introduced in R2021a

R2022a: `verify` throws warning if scores are not calibrated

Behavior changed in R2022a

Starting in R2022a, the `verify` function throws a warning if it is called with two output arguments and the scores from the i-vector system are not calibrated. Use `calibrate` to calibrate the scores.

See Also

`trainExtractor` | `trainClassifier` | `calibrate` | `unenroll` | `enroll` | `detectionErrorTradeoff` | `identify` | `ivector` | `info` | `addInfoHeader` | `release` | `ivectorSystem` | `speakerRecognition`

identify

Identify label

Syntax

```
tableOut = identify(ivs,data)
tableOut = identify(ivs,data,scorer)

tableOut = identify( ___ ,NumCandidates=N)
```

Description

`tableOut = identify(ivs,data)` identifies the label corresponding to the `data`.

`tableOut = identify(ivs,data,scorer)` specifies the scorer used to perform identification.

`tableOut = identify( ___ ,NumCandidates=N)` specifies the number of candidates to return in `tableOut`.

Examples

Train Speaker Identification System

Use the Census Database (also known as AN4 Database) from the CMU Robust Speech Recognition Group [1] on page 4-398. The data set contains recordings of male and female subjects speaking words and numbers. The helper function in this example downloads the data set for you and converts the raw files to FLAC, and returns two `audioDatastore` objects containing the training set and test set. By default, the data set is reduced so that the example runs quickly. You can use the full data set by setting `ReduceDataset` to false.

```
[adsTrain,adsTest] = HelperAN4Download(ReduceDataset=true);
```

Split the test data set into enroll and test sets. Use two utterances for enrollment and the remaining for the test set. Generally, the more utterances you use for enrollment, the better the performance of the system. However, most practical applications are limited to a small set of enrollment utterances.

```
[adsEnroll,adsTest] = splitEachLabel(adsTest,2);
```

Inspect the distribution of speakers in the training, test, and enroll sets. The speakers in the training set do not overlap with the speakers in the test and enroll sets.

```
summary(adsTrain.Labels)
```

```
     fejs      13
     fmjd      13
     fsrb      13
     ftmj      13
     fwxs      12
     mcen      13
     mrcb      13
```

```
     msjm      13
     msjr      13
     msmn       9

summary(adsEnroll.Labels)

     fvap      2
     marh      2

summary(adsTest.Labels)

     fvap      11
     marh      11
```

Create an i-vector system that accepts feature input.

```
fs = 16e3;
iv = ivectorSystem(SampleRate=fs,InputType="features");
```

Create an `audioFeatureExtractor` object to extract the gammatone cepstral coefficients (GTCC), the delta GTCC, the delta-delta GTCC, and the pitch from 50 ms periodic Hann windows with 45 ms overlap.

```
afe = audioFeatureExtractor(gtcc=true,gtccDelta=true,gtccDeltaDelta=true,pitch=true,SampleRate=fs
afe.Window = hann(round(0.05*fs),"periodic");
afe.OverlapLength = round(0.045*fs);
afe

afe =
  audioFeatureExtractor with properties:

   Properties
                     Window: [800×1 double]
              OverlapLength: 720
                 SampleRate: 16000
                  FFTLength: []
    SpectralDescriptorInput: 'linearSpectrum'
        FeatureVectorLength: 40

   Enabled Features
     gtcc, gtccDelta, gtccDeltaDelta, pitch

   Disabled Features
     linearSpectrum, melSpectrum, barkSpectrum, erbSpectrum, mfcc, mfccDelta
     mfccDeltaDelta, spectralCentroid, spectralCrest, spectralDecrease, spectralEntropy, spectral
     spectralFlux, spectralKurtosis, spectralRolloffPoint, spectralSkewness, spectralSlope, spect
     harmonicRatio, zerocrossrate, shortTimeEnergy


   To extract a feature, set the corresponding property to true.
   For example, obj.mfcc = true, adds mfcc to the list of enabled features.
```

Create transformed datastores by adding feature extraction to the `read` function of `adsTrain` and `adsEnroll`.

```
trainLabels = adsTrain.Labels;
adsTrain = transform(adsTrain,@(x)extract(afe,x));
```

```
enrollLabels = adsEnroll.Labels;
adsEnroll = transform(adsEnroll,@(x)extract(afe,x));
```

Train both the extractor and classifier using the training set.

```
trainExtractor(iv,adsTrain, ...
    UBMNumComponents=64, ...
    UBMNumIterations=5, ...
    TVSRank=32, ...
    TVSNumIterations=3);
```

```
Calculating standardization factors ....done.
Training universal background model ........done.
Training total variability space ......done.
i-vector extractor training complete.
```

```
trainClassifier(iv,adsTrain,trainLabels, ...
    NumEigenvectors=16, ...
    ...
    PLDANumDimensions=16, ...
    PLDANumIterations=5);
```

```
Extracting i-vectors ...done.
Training projection matrix .....done.
Training PLDA model ........done.
i-vector classifier training complete.
```

To calibrate the system so that scores can be interpreted as a measure of confidence in a positive decision, use `calibrate`.

```
calibrate(iv,adsTrain,trainLabels)
```

```
Extracting i-vectors ...done.
Calibrating CSS scorer ...done.
Calibrating PLDA scorer ...done.
Calibration complete.
```

Enroll the speakers from the enrollment set.

```
enroll(iv,adsEnroll,enrollLabels)
```

```
Extracting i-vectors ...done.
Enrolling i-vectors .....done.
Enrollment complete.
```

Evaluate the file-level prediction accuracy on the test set.

```
numCorrect = 0;
reset(adsTest)
for index = 1:numel(adsTest.Files)
    features = extract(afe,read(adsTest));

    results = identify(iv,features);

    trueLabel = adsTest.Labels(index);
    predictedLabel = results.Label(1);
    isPredictionCorrect = trueLabel==predictedLabel;

    numCorrect = numCorrect + isPredictionCorrect;
```

```
end
display("File Accuracy: " + round(100*numCorrect/numel(adsTest.Files),2) + " (%)")
```

```
    "File Accuracy: 100 (%)"
```

References

[1] "CMU Sphinx Group - Audio Databases." http://www.speech.cs.cmu.edu/databases/an4/. Accessed 19 Dec. 2019.

Input Arguments

`ivs` — i-vector system

`ivectorSystem` object

i-vector system, specified as an object of type `ivectorSystem`.

`data` — Data to identify

column vector | matrix

Data to identify, specified as a column vector representing a single-channel (mono) audio signal or a matrix of audio features.

- If `InputType` is set to `"audio"` when the i-vector system is created, `data` must be a column vector with underlying type `single` or `double`.
- If `InputType` is set to `"features"` when the i-vector system is created, `data` must be a matrix with underlying type `single` or `double`. The matrix must consist of audio features where the number of features (columns) is locked the first time `trainExtractor` is called and the number of hops (rows) is variable-sized.

Data Types: `single` | `double`

`scorer` — Scoring algorithm

`"plda"` | `"css"`

Scoring algorithm used by the i-vector system, specified as `"plda"`, which corresponds to probabilistic linear discriminant analysis (PLDA), or `"css"`, which corresponds to cosine similarity score (CSS).

To use `"plda"`, you must train the PLDA model using `trainClassifier`. If the PLDA model has been trained, then `scorer` defaults to `"plda"`. Otherwise, the scorer defaults to `"css"`.

Data Types: `char` | `string`

`N` — Number of candidates

positive scalar

Number of candidates to return in `tableOut`, specified as a positive scalar.

Note If you request a number of candidates greater than the number of `labels` enrolled in the i-vector system, then all candidates are returned. If unspecified, the number of candidates defaults to the number of enrolled `labels`.

Data Types: `single` | `double`

Output Arguments

`tableOut` — Score table

table

Candidate labels and corresponding scores, returned as a table. The number of rows of `tableOut` is equal to `N`, the number of candidates. The candidates are sorted in order of confidence.

Data Types: `table`

Version History

Introduced in R2021a

R2022a: `identify` throws warning if scores are not calibrated

Behavior changed in R2022a

Starting in R2022a, the `identify` function throws a warning if the scores from the i-vector system are not calibrated. Use `calibrate` to calibrate the scores.

See Also

`trainExtractor` | `trainClassifier` | `calibrate` | `unenroll` | `enroll` | `detectionErrorTradeoff` | `verify` | `ivector` | `info` | `addInfoHeader` | `release` | `ivectorSystem` | `speakerRecognition`

ivector

Extract i-vector

Syntax

```
w = ivector(ivs,data)
```

```
w = ivector(ivs,data,Name,Value)
```

Description

`w = ivector(ivs,data)` extracts i-vectors from the input `data`.

`w = ivector(ivs,data,Name,Value)` specifies additional options using name-value arguments. You can choose the hardware resource for extracting i-vectors and whether to apply the projection matrix from `trainClassifier`.

Examples

Train Word Recognition System

An i-vector system consists of a trainable front end that learns how to extract i-vectors based on unlabeled data, and a trainable backend that learns how to classify i-vectors based on labeled data. In this example, you apply an i-vector system to the task of word recognition. First, evaluate the accuracy of the i-vector system using the classifiers included in a traditional i-vector system: probabilistic linear discriminant analysis (PLDA) and cosine similarity scoring (CSS). Next, evaluate the accuracy of the system if you replace the classifier with bidirectional long short-term memory (BiLSTM) network or a K-nearest neighbors classifier.

Create Training and Validation Sets

Download the Free Spoken Digit Dataset (FSDD) [1] on page 4-406. FSDD consists of short audio files with spoken digits (0-9).

```
loc = matlab.internal.examples.downloadSupportFile("audio","FSDD.zip");
unzip(loc,pwd)
```

Create an `audioDatastore` to point to the recordings. Get the sample rate of the data set.

```
ads = audioDatastore(pwd,IncludeSubfolders=true);
[~,adsInfo] = read(ads);
fs = adsInfo.SampleRate;
```

The first element of the file names is the digit spoken in the file. Get the first element of the file names, convert them to categorical, and then set the `Labels` property of the `audioDatastore`.

```
[~,filenames] = cellfun(@(x)fileparts(x),ads.Files,UniformOutput=false);
ads.Labels = categorical(string(cellfun(@(x)x(1),filenames)));
```

To split the datastore into a development set and a validation set, use `splitEachLabel`. Allocate 80% of the data for development and the remaining 20% for validation.

```
[adsTrain,adsValidation] = splitEachLabel(ads,0.8);
```

Evaluate Traditional i-vector Backend Performance

Create an i-vector system that expects audio input at a sample rate of 8 kHz and does not perform speech detection.

```
wordRecognizer = ivectorSystem(DetectSpeech=false,SampleRate=fs)
```

```
wordRecognizer =
  ivectorSystem with properties:

         InputType: 'audio'
        SampleRate: 8000
      DetectSpeech: 0
           Verbose: 1
    EnrolledLabels: [0×2 table]
```

Train the i-vector extractor using the data in the training set.

```
trainExtractor(wordRecognizer,adsTrain, ...
    UBMNumComponents=64, ...
    UBMNumIterations=5, ...
    ...
    TVSRank=32, ...
    TVSNumIterations=5);
```

```
Calculating standardization factors ....done.
Training universal background model ........done.
Training total variability space ........done.
i-vector extractor training complete.
```

Train the i-vector classifier using the data in the training data set and the corresponding labels.

```
trainClassifier(wordRecognizer,adsTrain,adsTrain.Labels, ...
    NumEigenvectors=10, ...
    ...
    PLDANumDimensions=10, ...
    PLDANumIterations=5);
```

```
Extracting i-vectors ...done.
Training projection matrix .....done.
Training PLDA model ........done.
i-vector classifier training complete.
```

Calibrate the scores output by `wordRecognizer` so they can be interpreted as a measure of confidence in a positive decision. Enroll labels into the system using the entire training set.

```
calibrate(wordRecognizer,adsTrain,adsTrain.Labels)
```

```
Extracting i-vectors ...done.
Calibrating CSS scorer ...done.
Calibrating PLDA scorer ...done.
Calibration complete.
```

```
enroll(wordRecognizer,adsTrain,adsTrain.Labels)
```

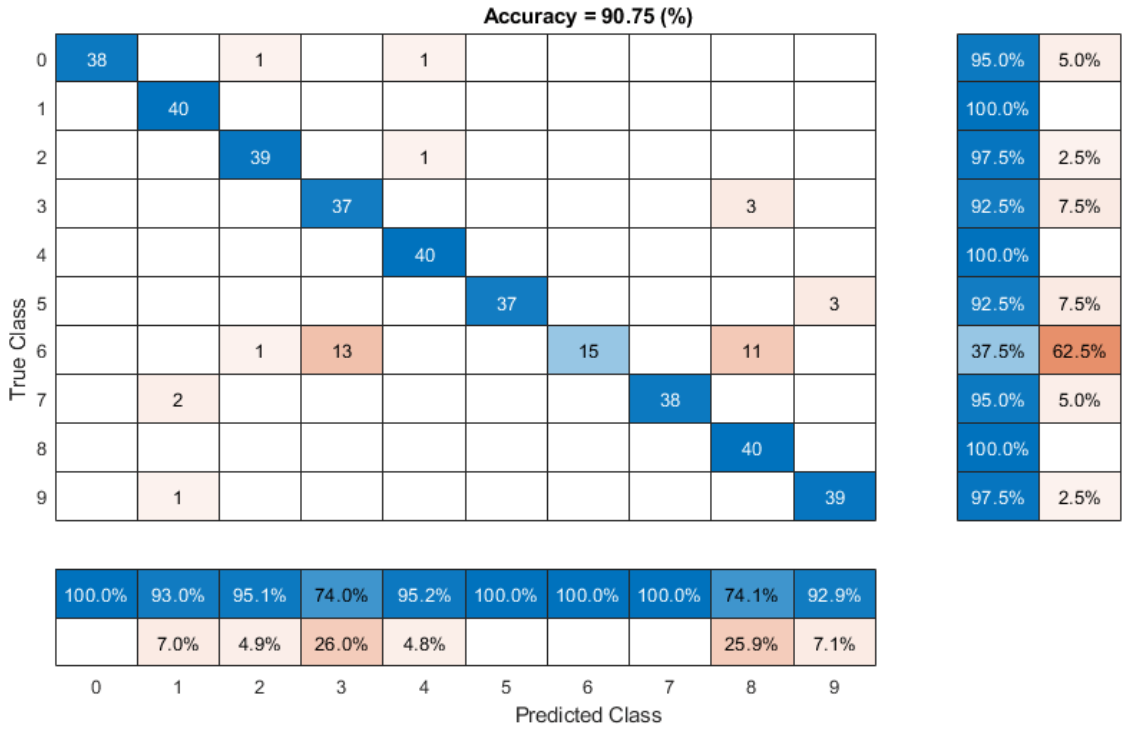
```
Extracting i-vectors ...done.
Enrolling i-vectors .............done.
Enrollment complete.
```

In a loop, read audio from the validation datastore, identify the most-likely word present according to the specified scorer, and save the prediction for analysis.

```
trueLabels = adsValidation.Labels;
predictedLabels = trueLabels;

reset(adsValidation)

scorer = plda;
for ii = 1:numel(trueLabels)

    audioIn = read(adsValidation);

    to = identify(wordRecognizer,audioIn,scorer);

    predictedLabels(ii) = to.Label(1);

end
```

Display a confusion chart of the i-vector system's performance on the validation set.

```
figure(Units="normalized",Position=[0.2 0.2 0.5 0.5])
confusionchart(trueLabels,predictedLabels, ...
    ColumnSummary="column-normalized", ...
    RowSummary="row-normalized", ...
    Title=sprintf('Accuracy = %0.2f (%%)',100*mean(predictedLabels==trueLabels)))
```



Evaluate Deep Learning Backend Performance

Next, train a fully-connected network using i-vectors as input.

```
ivectorsTrain = (ivector(wordRecognizer,adsTrain))';
ivectorsValidation = (ivector(wordRecognizer,adsValidation))';
```

Define a fully connected network.

```
layers = [ ...
    featureInputLayer(size(ivectorsTrain,2),Normalization="none")
    fullyConnectedLayer(128)
    dropoutLayer(0.4)
    fullyConnectedLayer(256)
    dropoutLayer(0.4)
    fullyConnectedLayer(256)
    dropoutLayer(0.4)
    fullyConnectedLayer(128)
    dropoutLayer(0.4)
    fullyConnectedLayer(numel(unique(adsTrain.Labels)))
    softmaxLayer
    classificationLayer];
```

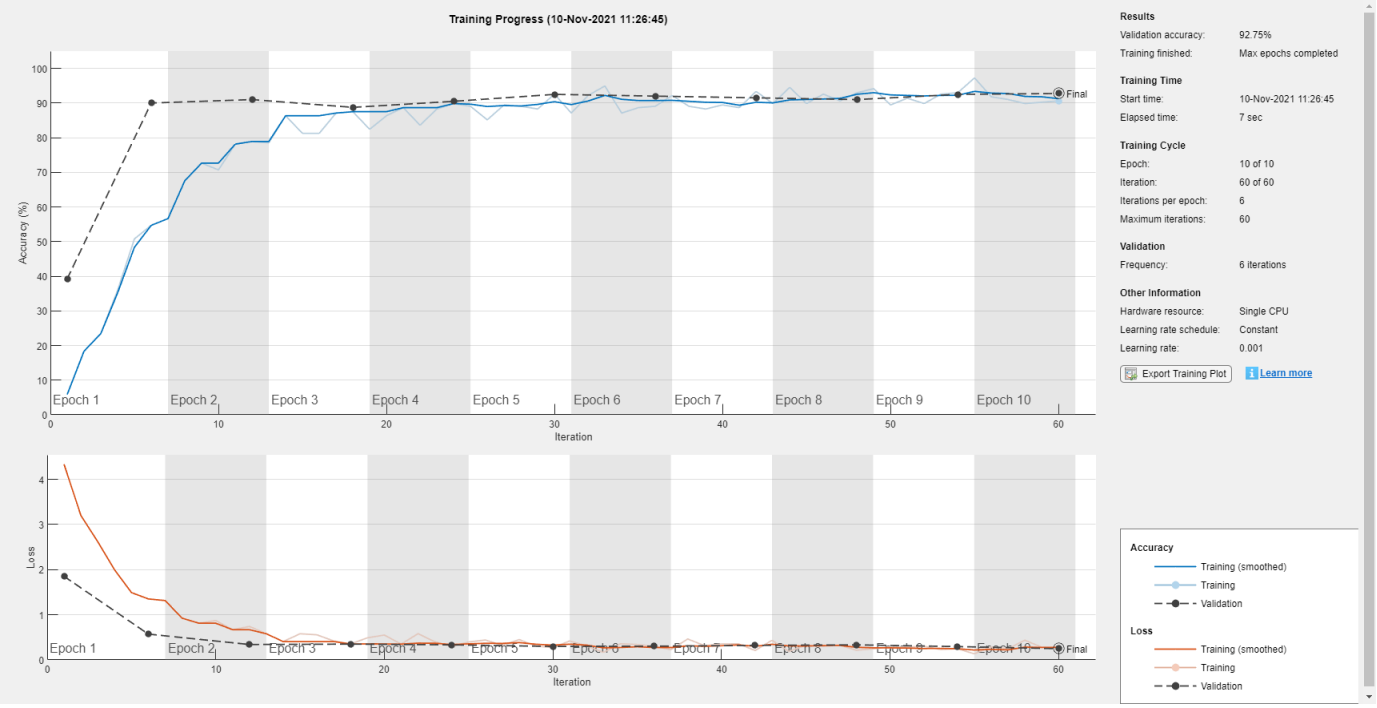
Define training parameters.

```
miniBatchSize = 256;
validationFrequency = floor(numel(adsTrain.Labels)/miniBatchSize);
options = trainingOptions("adam", ...
    MaxEpochs=10, ...
    MiniBatchSize=miniBatchSize, ...
```

```
    Plots="training-progress", ...
    Verbose=false, ...
    Shuffle="every-epoch", ...
    ValidationData={ivectorsValidation,adsValidation.Labels}, ...
    ValidationFrequency=validationFrequency);
```

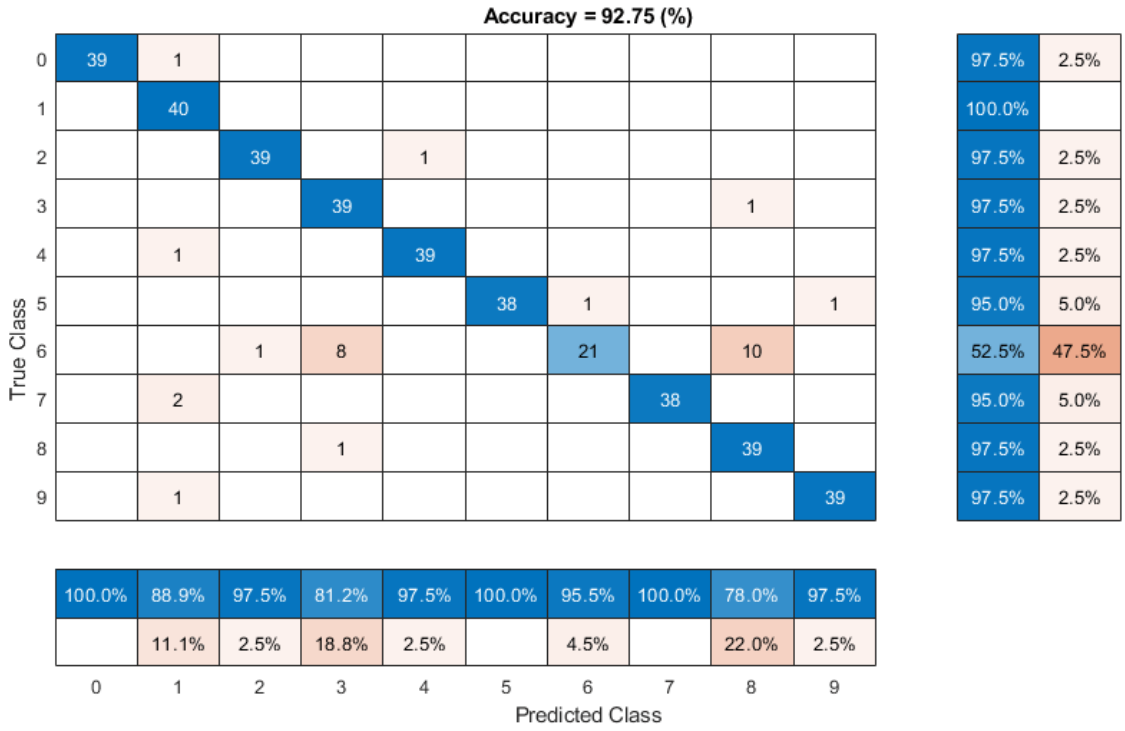
Train the network.

```
net = trainNetwork(ivectorsTrain,adsTrain.Labels,layers,options);
```



Evaluate the performance of the deep learning backend using a confusion chart.

```
predictedLabels = classify(net,ivectorsValidation);
trueLabels = adsValidation.Labels;

figure(Units="normalized",Position=[0.2 0.2 0.5 0.5])
confusionchart(trueLabels,predictedLabels, ...
    ColumnSummary="column-normalized", ...
    RowSummary="row-normalized", ...
    Title=sprintf('Accuracy = %0.2f (%%)',100*mean(predictedLabels==trueLabels)))
```



Evaluate KNN Backend Performance

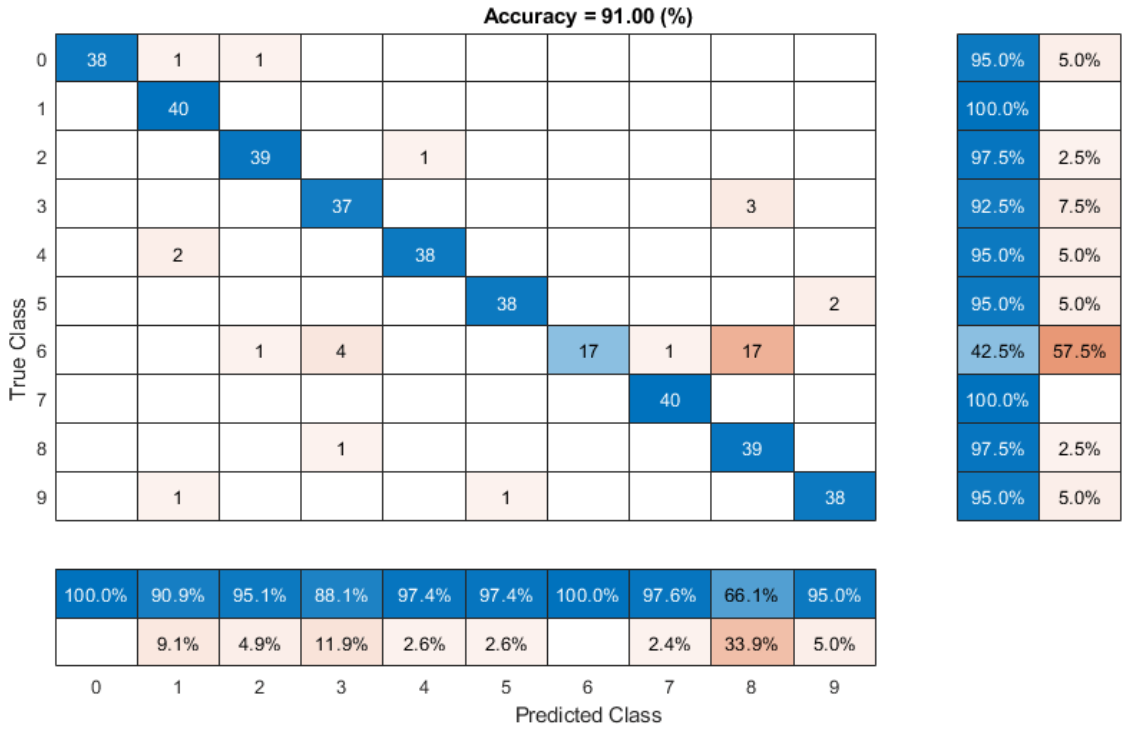
Train and evaluate i-vectors with a *k*-nearest neighbor (KNN) backend.

Use `fitcknn` to train a KNN model.

```
classificationKNN = fitcknn(...
    ivectorsTrain, ...
    adsTrain.Labels, ...
    Distance="Euclidean", ...
    Exponent=[], ...
    NumNeighbors=10, ...
    DistanceWeight="SquaredInverse", ...
    Standardize=true, ...
    ClassNames=unique(adsTrain.Labels));
```

Evaluate the KNN backend.

```
predictedLabels = predict(classificationKNN,ivectorsValidation);
trueLabels = adsValidation.Labels;

figure(Units="normalized",Position=[0.2 0.2 0.5 0.5])
confusionchart(trueLabels,predictedLabels, ...
    ColumnSummary="column-normalized", ...
    RowSummary="row-normalized", ...
    Title=sprintf('Accuracy = %0.2f (%%)',100*mean(predictedLabels==trueLabels)))
```



References

[1] Jakobovski. "Jakobovski/Free-Spoken-Digit-Dataset." GitHub, May 30, 2019. `https://github.com/Jakobovski/free-spoken-digit-dataset`.

Input Arguments

`ivs` — i-vector system

`ivectorSystem` object

i-vector system, specified as an object of type `ivectorSystem`.

`data` — Data to transform

column vector | cell array | `audioDatastore` | `signalDatastore` | `TransformedDatastore`

Data to transform, specified as a cell array or as an `audioDatastore`, `signalDatastore`, or `TransformedDatastore` object.

- If `InputType` is set to `"audio"` when the i-vector system is created, specify `data` as one of these:
  - A column vector with underlying type `single` or `double`.
  - A cell array of single-channel audio signals, each specified as a column vector with underlying type `single` or `double`.
  - An `audioDatastore` object or a `signalDatastore` object that points to a data set of mono audio signals.

  - A `TransformedDatastore` with an underlying `audioDatastore` or `signalDatastore` that points to a data set of mono audio signals. The output from calls to `read` from the transform datastore must be mono audio signals with underlying data type `single` or `double`.
- If `InputType` is set to `"features"` when the i-vector system is created, specify `data` as one of these:
  - A matrix with underlying type `single` or `double`. The matrix must consist of audio features where the number of features (columns) is locked the first time `trainExtractor` is called and the number of hops (rows) is variable-sized. The number of features input in any subsequent calls to any of the object functions must be equal to the number of features used when calling `trainExtractor`.
  - A cell array of matrices with underlying type `single` or `double`. The matrices must consist of audio features where the number of features (columns) is locked the first time `trainExtractor` is called and the number of hops (rows) is variable-sized. The number of features input in any subsequent calls to any of the object functions must be equal to the number of features used when calling `trainExtractor`.
  - A `TransformedDatastore` object with an underlying `audioDatastore` or `signalDatastore` whose `read` function has output as described in the previous bullet.
  - A `signalDatastore` object whose `read` function has output as described in the first bullet.

Data Types: `cell` | `audioDatastore` | `signalDatastore`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example:
`ivector(ivs,data,ApplyProjectionMatrix=false,ExecutionEnvironment="parallel")`

`ApplyProjectionMatrix` — Option to apply projection matrix
`true` | `false`

Option to apply projection matrix, specified as a logical value. This argument specifies whether to apply the linear discriminant analysis (LDA) and within-class covariance normalization (WCCN) projection matrix determined using `trainClassifier`.

- If the projection matrix was trained, then `ApplyProjectionMatrix` defaults to `true`.
- If the projection matrix was not trained, then `ApplyProjectionMatrix` defaults to `false` and cannot be set to `true`.

Data Types: `logical`

`ExecutionEnvironment` — Hardware resource for execution
`"auto"` (default) | `"cpu"` | `"gpu"` | `"multi-gpu"` | `"parallel"`

Hardware resource for execution, specified as one of these:

- `"auto"` — Use the GPU if it is available. Otherwise, use the CPU.
- `"cpu"` — Use the CPU.

- `"gpu"` — Use the GPU. This option requires Parallel Computing Toolbox.
- `"multi-gpu"` — Use multiple GPUs on one machine, using a local parallel pool based on your default cluster profile. If there is no current parallel pool, the software starts a parallel pool with pool size equal to the number of available GPUs. This option requires Parallel Computing Toolbox.
- `"parallel"` — Use a local or remote parallel pool based on your default cluster profile. If there is no current parallel pool, the software starts one using the default cluster profile. If the pool has access to GPUs, then only workers with a unique GPU perform training computation. If the pool does not have GPUs, then the training takes place on all available CPU workers. This option requires Parallel Computing Toolbox.

Data Types: `char` | `string`

`DispatchInBackground` — Option to use prefetch queuing
`false` (default) | `true`

Option to use prefetch queuing when reading from a datastore, specified as a logical value. This argument requires Parallel Computing Toolbox.

Data Types: `logical`

Output Arguments

`w` — i-vectors
column vector | matrix

Extracted i-vectors, returned as a column vector or a matrix. The number of columns of `w` is equal to the number of input signals. The number of rows of `w` is the dimension of the i-vector.

Version History

Introduced in R2021a

See Also

`trainExtractor` | `trainClassifier` | `calibrate` | `enroll` | `unenroll` | `detectionErrorTradeoff` | `verify` | `identify` | `info` | `addInfoHeader` | `release` | `ivectorSystem` | `speakerRecognition`

info

Return training configuration and data info

Syntax

```
ivInfo = info(ivs)

info(ivs)
```

Description

`ivInfo = info(ivs)` returns a structure containing information about the `ivs` object.

`info(ivs)`, with no output arguments, displays information about the `ivs` object.

Examples

Train Speaker Verification System

Use the Pitch Tracking Database from Graz University of Technology (PTDB-TUG) [1] on page 4-419. The data set consists of 20 English native speakers reading 2342 phonetically rich sentences from the TIMIT corpus. Download and extract the data set.

```
downloadFolder = matlab.internal.examples.downloadSupportFile("audio","ptdb-tug.zip");
dataFolder = tempdir;
unzip(downloadFolder,dataFolder)
dataset = fullfile(dataFolder,"ptdb-tug");
```

Create an `audioDatastore` object that points to the data set. The data set was originally intended for use in pitch-tracking training and evaluation and includes laryngograph readings and baseline pitch decisions. Use only the original audio recordings.

```
ads = audioDatastore([fullfile(dataset,"SPEECH DATA","FEMALE","MIC"),fullfile(dataset,"SPEECH DA
                      IncludeSubfolders=true, ...
                      FileExtensions=".wav");
```

The file names contain the speaker IDs. Decode the file names to set the labels in the `audioDatastore` object.

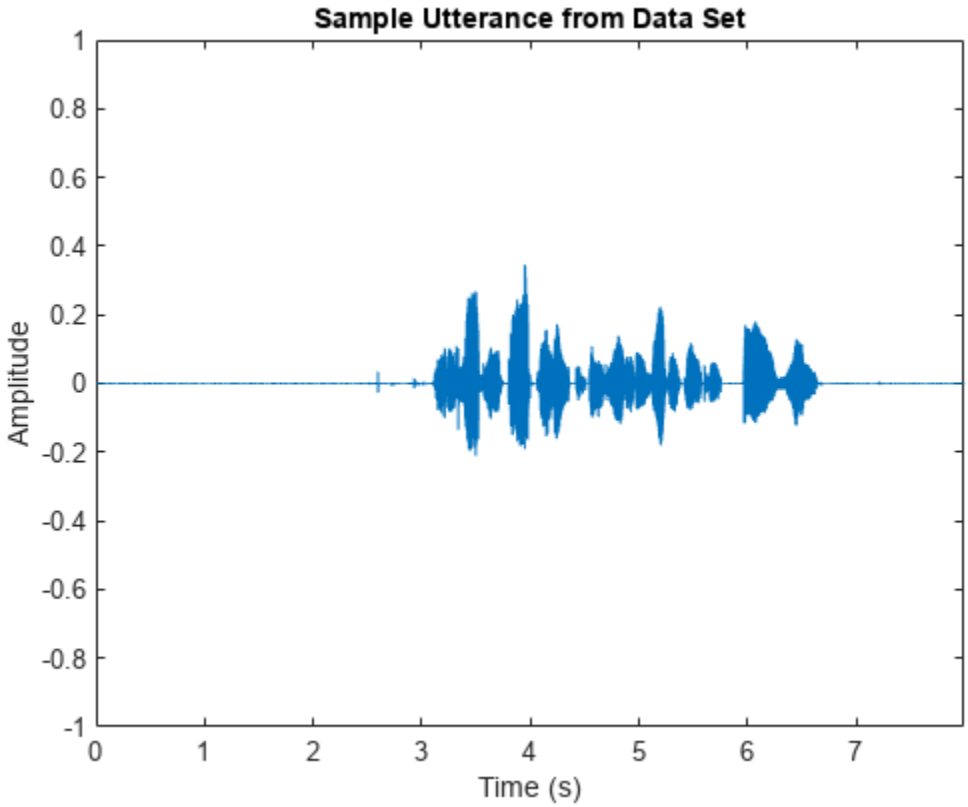
```
ads.Labels = extractBetween(ads.Files,"mic_","_");
countEachLabel(ads)
```

```
ans=20×2 table
    Label    Count
    _____    _____

     F01      236
     F02      236
     F03      236
     F04      236
     F05      236
```

```
F06      236
F07      236
F08      234
F09      236
F10      236
M01      236
M02      236
M03      236
M04      236
M05      236
M06      236
  ⋮
```

Read an audio file from the data set, listen to it, and plot it.

```
[audioIn,audioInfo] = read(ads);
fs = audioInfo.SampleRate;

t = (0:size(audioIn,1)-1)/fs;
sound(audioIn,fs)
plot(t,audioIn)
xlabel("Time (s)")
ylabel("Amplitude")
axis([0 t(end) -1 1])
title("Sample Utterance from Data Set")
```



Separate the `audioDatastore` object into four: one for training, one for enrollment, one to evaluate the detection-error tradeoff, and one for testing. The training set contains 16 speakers. The enrollment, detection-error tradeoff, and test sets contain the other four speakers.

```
speakersToTest = categorical(["M01","M05","F01","F05"]);

adsTrain = subset(ads,~ismember(ads.Labels,speakersToTest));

ads = subset(ads,ismember(ads.Labels,speakersToTest));
[adsEnroll,adsTest,adsDET] = splitEachLabel(ads,3,1);
```

Display the label distributions of the `audioDatastore` objects.

```
countEachLabel(adsTrain)
```

```
ans=16×2 table
    Label    Count
    _____    _____

     F02      236
     F03      236
     F04      236
     F06      236
     F07      236
     F08      234
     F09      236
     F10      236
     M02      236
     M03      236
     M04      236
     M06      236
     M07      236
     M08      236
     M09      236
     M10      236
```

```
countEachLabel(adsEnroll)
```

```
ans=4×2 table
    Label    Count
    _____    _____

     F01       3
     F05       3
     M01       3
     M05       3
```

```
countEachLabel(adsTest)
```

```
ans=4×2 table
    Label    Count
    _____    _____

     F01       1
     F05       1
     M01       1
```

```
     M05       1
```

```
countEachLabel(adsDET)
```

```
ans=4×2 table
    Label    Count
    _____    _____

     F01      232
     F05      232
     M01      232
     M05      232
```

Create an i-vector system. By default, the i-vector system assumes the input to the system is mono audio signals.

```
speakerVerification = ivectorSystem(SampleRate=fs)
```

```
speakerVerification =
  ivectorSystem with properties:

         InputType: 'audio'
        SampleRate: 48000
      DetectSpeech: 1
           Verbose: 1
    EnrolledLabels: [0×2 table]
```

To train the extractor of the i-vector system, call `trainExtractor`. Specify the number of universal background model (UBM) components as 128 and the number of expectation maximization iterations as 5. Specify the total variability space (TVS) rank as 64 and the number of iterations as 3.

```
trainExtractor(speakerVerification,adsTrain, ...
    UBMNumComponents=128,UBMNumIterations=5, ...
    TVSRank=64,TVSNumIterations=3)
```

```
Calculating standardization factors ....done.
Training universal background model ........done.
Training total variability space ......done.
i-vector extractor training complete.
```

To train the classifier of the i-vector system, use `trainClassifier`. To reduce dimensionality of the i-vectors, specify the number of eigenvectors in the projection matrix as 16. Specify the number of dimensions in the probabilistic linear discriminant analysis (PLDA) model as 16, and the number of iterations as 3.

```
trainClassifier(speakerVerification,adsTrain,adsTrain.Labels, ...
    NumEigenvectors=16, ...
    PLDANumDimensions=16,PLDANumIterations=3)
```

```
Extracting i-vectors ...done.
Training projection matrix .....done.
Training PLDA model ......done.
i-vector classifier training complete.
```

To calibrate the system so that scores can be interpreted as a measure of confidence in a positive decision, use `calibrate`.

```
calibrate(speakerVerification,adsTrain,adsTrain.Labels)
```

```
Extracting i-vectors ...done.
Calibrating CSS scorer ...done.
Calibrating PLDA scorer ...done.
Calibration complete.
```

To inspect parameters used previously to train the i-vector system, use `info`.

```
info(speakerVerification)
```

```
i-vector system input
  Input feature vector length: 60
  Input data type: double

trainExtractor
  Train signals: 3774
  UBMNumComponents: 128
  UBMNumIterations: 5
  TVSRank: 64
  TVSNumIterations: 3

trainClassifier
  Train signals: 3774
  Train labels: F02 (236), F03 (236) ... and 14 more
  NumEigenvectors: 16
  PLDANumDimensions: 16
  PLDANumIterations: 3

calibrate
  Calibration signals: 3774
  Calibration labels: F02 (236), F03 (236) ... and 14 more
```

Split the enrollment set.

```
[adsEnrollPart1,adsEnrollPart2] = splitEachLabel(adsEnroll,1,2);
```

To enroll speakers in the i-vector system, call `enroll`.

```
enroll(speakerVerification,adsEnrollPart1,adsEnrollPart1.Labels)
```

```
Extracting i-vectors ...done.
Enrolling i-vectors .......done.
Enrollment complete.
```

When you enroll speakers, the read-only `EnrolledLabels` property is updated with the enrolled labels and corresponding template i-vectors. The table also keeps track of the number of signals used to create the template i-vector. Generally, using more signals results in a better template.

```
speakerVerification.EnrolledLabels
```

```
ans=4×2 table
              ivector       NumSamples
            ____________    __________

    F01    {16×1 double}        1
    F05    {16×1 double}        1
    M01    {16×1 double}        1
```

```
    M05    {16×1 double}        1
```

Enroll the second part of the enrollment set and then view the enrolled labels table again. The i-vector templates and the number of samples are updated.

```
enroll(speakerVerification,adsEnrollPart2,adsEnrollPart2.Labels)
```

```
Extracting i-vectors ...done.
Enrolling i-vectors .......done.
Enrollment complete.
```

```
speakerVerification.EnrolledLabels
```

```
ans=4×2 table
              ivector       NumSamples
           _____________    __________

    F01    {16×1 double}        3
    F05    {16×1 double}        3
    M01    {16×1 double}        3
    M05    {16×1 double}        3
```

To evaluate the i-vector system and determine a decision threshold for speaker verification, call `detectionErrorTradeoff`.

```
[results, eerThreshold] = detectionErrorTradeoff(speakerVerification,adsDET,adsDET.Labels);
```

```
Extracting i-vectors ...done.
Scoring i-vector pairs ...done.
Detection error tradeoff evaluation complete.
```

The first output from `detectionErrorTradeoff` is a structure with two fields: CSS and PLDA. Each field contains a table. Each row of the table contains a possible decision threshold for speaker verification tasks, and the corresponding false alarm rate (FAR) and false rejection rate (FRR). The FAR and FRR are determined using the enrolled speaker labels and the data input to the `detectionErrorTradeoff` function.

```
results
```

```
results = struct with fields:
    PLDA: [1000×3 table]
     CSS: [1000×3 table]
```

```
results.CSS
```

```
ans=1000×3 table
    Threshold       FAR      FRR
    __________    _______    ___

    2.3259e-10          1     0
    2.3965e-10    0.99964     0
    2.4693e-10    0.99928     0
    2.5442e-10    0.99928     0
    2.6215e-10    0.99928     0
     2.701e-10    0.99928     0
     2.783e-10    0.99928     0
```

```
    2.8675e-10    0.99928     0
    2.9545e-10    0.99928     0
    3.0442e-10    0.99928     0
    3.1366e-10    0.99928     0
    3.2318e-10    0.99928     0
    3.3299e-10    0.99928     0
     3.431e-10    0.99928     0
    3.5352e-10    0.99928     0
    3.6425e-10    0.99892     0
      ⋮
```

```
results.PLDA
```

```
ans=1000×3 table
    Threshold       FAR      FRR
    __________    _______    ___

    3.2661e-40          1     0
    3.6177e-40    0.99964     0
    4.0072e-40    0.99964     0
    4.4387e-40    0.99964     0
    4.9166e-40    0.99964     0
    5.4459e-40    0.99964     0
    6.0322e-40    0.99964     0
    6.6817e-40    0.99964     0
    7.4011e-40    0.99964     0
     8.198e-40    0.99964     0
    9.0806e-40    0.99964     0
    1.0058e-39    0.99964     0
    1.1141e-39    0.99964     0
    1.2341e-39    0.99964     0
    1.3669e-39    0.99964     0
    1.5141e-39    0.99964     0
      ⋮
```

The second output from `detectionErrorTradeoff` is a structure with two fields: `CSS` and `PLDA`. The corresponding value is the decision threshold that results in the equal error rate (when FAR and FRR are equal).

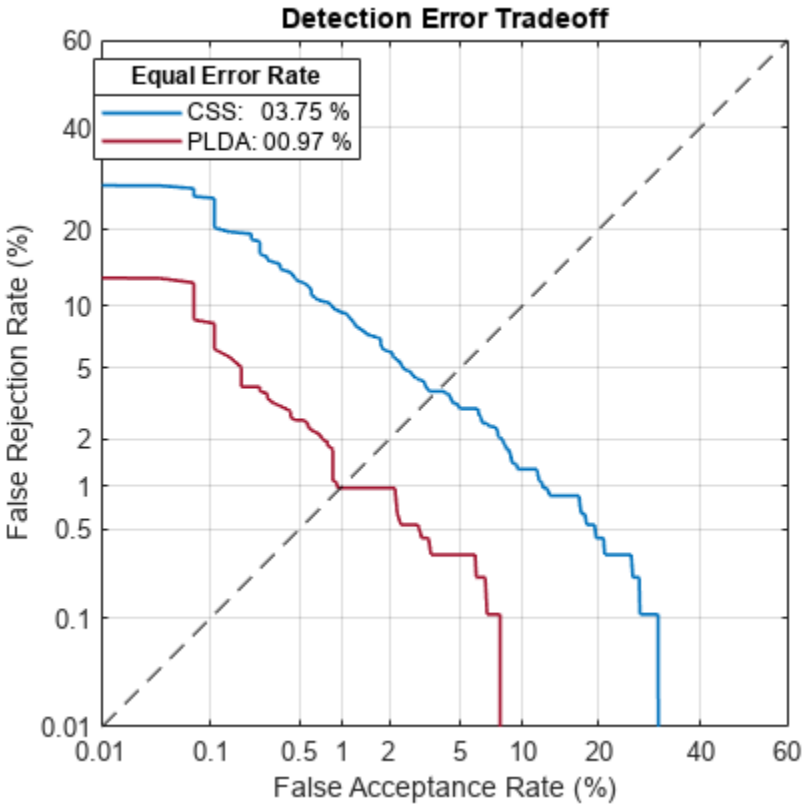
```
eerThreshold
```

```
eerThreshold = struct with fields:
    PLDA: 0.0398
     CSS: 0.9369
```

The first time you call `detectionErrorTradeoff`, you must provide data and corresponding labels to evaluate. Subsequently, you can get the same information, or a different analysis using the same underlying data, by calling `detectionErrorTradeoff` without data and labels.

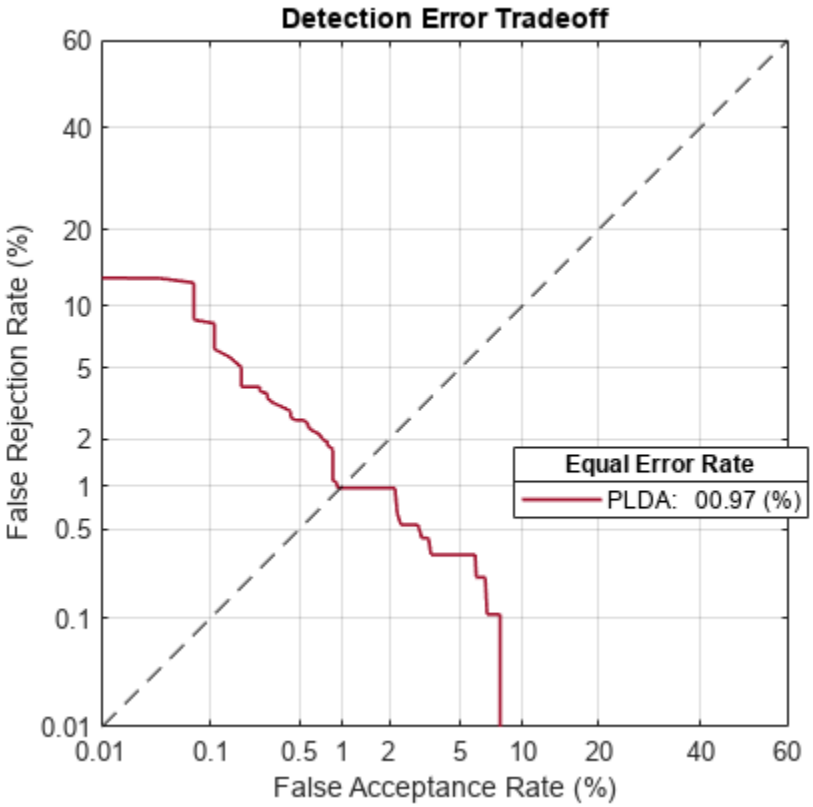
Call `detectionErrorTradeoff` a second time with no data arguments or output arguments to visualize the detection-error tradeoff.

```
detectionErrorTradeoff(speakerVerification)
```



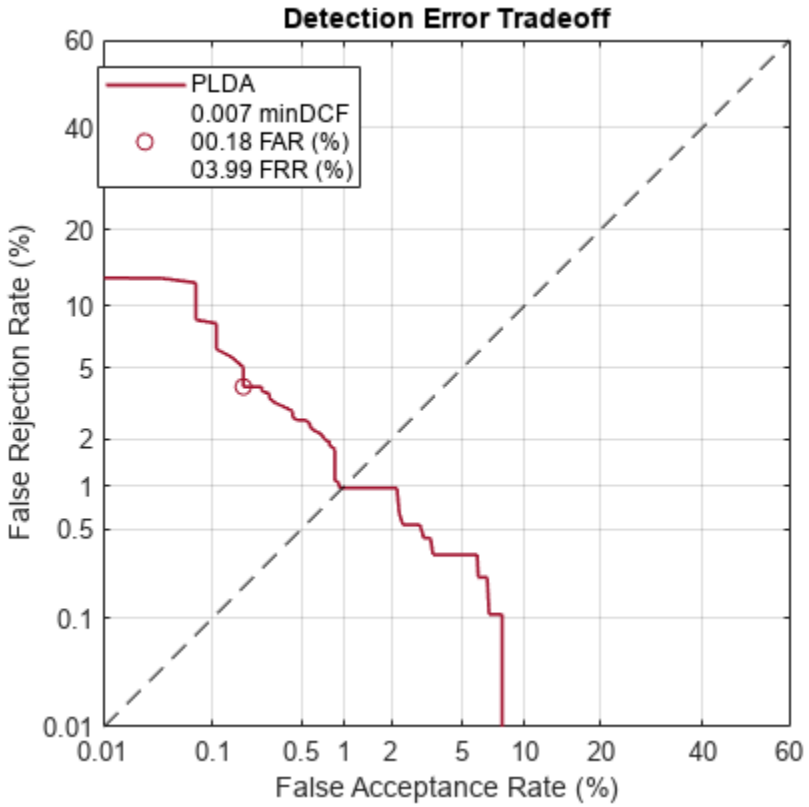
Call `detectionErrorTradeoff` again. This time, visualize only the detection-error tradeoff for the PLDA scorer.

```
detectionErrorTradeoff(speakerVerification,Scorer="plda")
```



Depending on your application, you may want to use a threshold that weights the error cost of a false alarm higher or lower than the error cost of a false rejection. You may also be using data that is not representative of the prior probability of the speaker being present. You can use the `minDCF` parameter to specify custom costs and prior probability. Call `detectionErrorTradeoff` again, this time specify the cost of a false rejection as 1, the cost of a false acceptance as 2, and the prior probability that a speaker is present as 0.1.

```
costFR = 1;
costFA = 2;
priorProb = 0.1;
detectionErrorTradeoff(speakerVerification,Scorer="plda",minDCF=[costFR,costFA,priorProb])
```



Call `detectionErrorTradeoff` again. This time, get the `minDCF` threshold for the PLDA scorer and the parameters of the detection cost function.

```
[~,minDCFThreshold] = detectionErrorTradeoff(speakerVerification,Scorer="plda",minDCF=[costFR,cos
```

```
minDCFThreshold = 0.4709
```

Test Speaker Verification System

Read a signal from the test set.

```
adsTest = shuffle(adsTest);
[audioIn,audioInfo] = read(adsTest);
knownSpeakerID = audioInfo.Label
```

```
knownSpeakerID = 1×1 cell array
    {'F01'}
```

To perform speaker verification, call `verify` with the audio signal and specify the speaker ID, a scorer, and a threshold for the scorer. The `verify` function returns a logical value indicating whether a speaker identity is accepted or rejected, and a score indicating the similarity of the input audio and the template i-vector corresponding to the enrolled label.

```
[tf,score] = verify(speakerVerification,audioIn,knownSpeakerID,"plda",eerThreshold.PLDA);
if tf
    fprintf('Success!\nSpeaker accepted.\nSimilarity score = %0.2f\n\n',score)
else
```

```
    fprinf('Failure!\nSpeaker rejected.\nSimilarity score = %0.2f\n\n',score)
end
```

```
Success!
Speaker accepted.
Similarity score = 1.00
```

Call speaker verification again. This time, specify an incorrect speaker ID.

```
possibleSpeakers = speakerVerification.EnrolledLabels.Properties.RowNames;
imposterIdx = find(~ismember(possibleSpeakers,knownSpeakerID));
imposter = possibleSpeakers(imposterIdx(randperm(numel(imposterIdx),1)))
```

```
imposter = 1×1 cell array
    {'M05'}
```

```
[tf,score] = verify(speakerVerification,audioIn,imposter,"plda",eerThreshold.PLDA);
if tf
    fprintf('Failure!\nSpeaker accepted.\nSimilarity score = %0.2f\n\n',score)
else
    fprintf('Success!\nSpeaker rejected.\nSimilarity score = %0.2f\n\n',score)
end
```

```
Success!
Speaker rejected.
Similarity score = 0.00
```

References

[1] Signal Processing and Speech Communication Laboratory. https://www.spsc.tugraz.at/databases-and-tools/ptdb-tug-pitch-tracking-database-from-graz-university-of-technology.html. Accessed 12 Dec. 2019.

Input Arguments

`ivs` — i-vector system

`ivectorSystem` object

i-vector system, specified as an object of type `ivectorSystem`.

Output Arguments

`ivInfo` — i-vector information

structure

Information about how the i-vector system was trained and evaluated, returned as a structure.

Data Types: `struct`

Version History

Introduced in R2021a

See Also

`trainExtractor` | `trainClassifier` | `calibrate` | `unenroll` | `enroll` | `detectionErrorTradeoff` | `verify` | `identify` | `ivector` | `addInfoHeader` | `release` | `ivectorSystem` | `speakerRecognition`

addInfoHeader

Add custom information about i-vector system

Syntax

```
addInfoHeader(ivs,str)
addInfoHeader(ivs)
```

Description

`addInfoHeader(ivs,str)` adds a field called `Header` to the structure output by `info(ivs)` and populates it with `str`.

`addInfoHeader(ivs)` clears the custom information from the i-vector system `ivs`.

Examples

Add Custom Header Information

Create a default i-vector system.

```
ivs = ivectorSystem;
```

Add custom header information to the object. Use the `info` function to display the information.

```
addInfoHeader(ivs,'Custom Header Information')
```

```
info(ivs)
```

```
Header
  Custom Header Information
```

Input Arguments

`ivs` — i-vector system

`ivectorSystem` object

i-vector system, specified as an object of type `ivectorSystem`.

`str` — Custom information

character vector | string scalar

Custom information about i-vector system, specified as a character vector or string scalar.

Data Types: `char` | `string`

Version History

Introduced in R2021b

See Also

`trainExtractor` | `trainClassifier` | `unenroll` | `enroll` | `detectionErrorTradeoff` | `verify` | `identify` | `ivector` | `info` | `release` | `ivectorSystem` | `speakerRecognition`

release

Allow property values and input characteristics to change

Syntax

```
release(ivs)
```

Description

`release(ivs)` allows property values and input characteristics of the i-vector system `ivs` to change.

Examples

Train Environmental Sound Classification System

Download and unzip the environment sound classification data set. This data set consists of recordings labeled as one of 10 different audio sound classes (ESC-10).

```
loc = matlab.internal.examples.downloadSupportFile("audio","ESC-10.zip");
unzip(loc,pwd)
```

Create an `audioDatastore` object to manage the data and split it into training and validation sets. Call `countEachLabel` to display the distribution of sound classes and the number of unique labels.

```
ads = audioDatastore(pwd,IncludeSubfolders=true,LabelSource="foldernames");
countEachLabel(ads)
```

```
ans=10×2 table
        Label         Count
    ______________    _____

    chainsaw           40
    clock_tick         40
    crackling_fire     40
    crying_baby        40
    dog                40
    helicopter         40
    rain               40
    rooster            38
    sea_waves          40
    sneezing           40
```

Listen to one of the files.

```
[audioIn,audioInfo] = read(ads);
fs = audioInfo.SampleRate;
sound(audioIn,fs)
audioInfo.Label
```

```
ans = categorical
     chainsaw
```

Split the datastore into training and test sets.

```
[adsTrain,adsTest] = splitEachLabel(ads,0.8);
```

Create an `audioFeatureExtractor` to extract all possible features from the audio.

```
afe = audioFeatureExtractor(SampleRate=fs, ...
    Window=hamming(round(0.03*fs),"periodic"), ...
    OverlapLength=round(0.02*fs));
params = info(afe,"all");
params = structfun(@(x)true,params,UniformOutput=false);
set(afe,params);
afe
```

```
afe =
  audioFeatureExtractor with properties:

   Properties
                     Window: [1323×1 double]
              OverlapLength: 882
                 SampleRate: 44100
                  FFTLength: []
    SpectralDescriptorInput: 'linearSpectrum'
        FeatureVectorLength: 862

   Enabled Features
     linearSpectrum, melSpectrum, barkSpectrum, erbSpectrum, mfcc, mfccDelta
     mfccDeltaDelta, gtcc, gtccDelta, gtccDeltaDelta, spectralCentroid, spectralCrest
     spectralDecrease, spectralEntropy, spectralFlatness, spectralFlux, spectralKurtosis, spectra
     spectralSkewness, spectralSlope, spectralSpread, pitch, harmonicRatio, zerocrossrate
     shortTimeEnergy

   Disabled Features
     none


   To extract a feature, set the corresponding property to true.
   For example, obj.mfcc = true, adds mfcc to the list of enabled features.
```

Create two directories in your current folder: train and test. Extract features from the training and the test data sets and write the features as MAT files to the respective directories. Pre-extracting features can save time when you want to evaluate different feature combinations or training configurations.

```
if ~isdir("train")
    mkdir("train")
    mkdir("test")

    outputType = ".mat";
    writeall(adsTrain,"train",WriteFcn=@(x,y,z)writeFeatures(x,y,z,afe))
    writeall(adsTest,"test",WriteFcn=@(x,y,z)writeFeatures(x,y,z,afe))
end
```

Create signal datastores to point to the audio features.

```
sdsTrain = signalDatastore("train",IncludeSubfolders=true);
sdsTest = signalDatastore("test",IncludeSubfolders=true);
```

Create label arrays that are in the same order as the `signalDatastore` files.

```
labelsTrain = categorical(extractBetween(sdsTrain.Files,"ESC-10"+filesep,filesep));
labelsTest = categorical(extractBetween(sdsTest.Files,"ESC-10"+filesep,filesep));
```

Create a transform datastore from the signal datastores to isolate and use only the desired features. You can use the output from `info` on the `audioFeatureExtractor` to map your chosen features to the index in the features matrix. You can experiment with the example by choosing different features.

```
featureIndices = info(afe)
```

```
featureIndices = struct with fields:
          linearSpectrum: [1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 2
             melSpectrum: [663 664 665 666 667 668 669 670 671 672 673 674 675 676 677 678 679 68
            barkSpectrum: [695 696 697 698 699 700 701 702 703 704 705 706 707 708 709 710 711 71
             erbSpectrum: [727 728 729 730 731 732 733 734 735 736 737 738 739 740 741 742 743 74
                    mfcc: [770 771 772 773 774 775 776 777 778 779 780 781 782]
              mfccDelta: [783 784 785 786 787 788 789 790 791 792 793 794 795]
          mfccDeltaDelta: [796 797 798 799 800 801 802 803 804 805 806 807 808]
                    gtcc: [809 810 811 812 813 814 815 816 817 818 819 820 821]
               gtccDelta: [822 823 824 825 826 827 828 829 830 831 832 833 834]
          gtccDeltaDelta: [835 836 837 838 839 840 841 842 843 844 845 846 847]
        spectralCentroid: 848
           spectralCrest: 849
        spectralDecrease: 850
         spectralEntropy: 851
        spectralFlatness: 852
            spectralFlux: 853
        spectralKurtosis: 854
    spectralRolloffPoint: 855
        spectralSkewness: 856
           spectralSlope: 857
          spectralSpread: 858
                   pitch: 859
           harmonicRatio: 860
           zerocrossrate: 861
         shortTimeEnergy: 862
```

```
idxToUse = [...
    featureIndices.harmonicRatio ...
    ,featureIndices.spectralRolloffPoint ...
    ,featureIndices.spectralFlux ...
    ,featureIndices.spectralSlope ...
    ];
tdsTrain = transform(sdsTrain,@(x)x(:,idxToUse));
tdsTest = transform(sdsTest,@(x)x(:,idxToUse));
```

Create an i-vector system that accepts feature input.

```
soundClassifier = ivectorSystem(InputType="features");
```

Train the extractor and classifier using the training set.

```
trainExtractor(soundClassifier,tdsTrain,UBMNumComponents=128,TVSRank=64);
```

```
Calculating standardization factors ....done.
Training universal background model .....done.
Training total variability space ......done.
i-vector extractor training complete.
```

```
trainClassifier(soundClassifier,tdsTrain,labelsTrain,NumEigenvectors=32,PLDANumIterations=0)
```

```
Extracting i-vectors ...done.
Training projection matrix .....done.
i-vector classifier training complete.
```

Enroll the labels from the training set to create i-vector templates for each of the environmental sounds.

```
enroll(soundClassifier,tdsTrain,labelsTrain)
```

```
Extracting i-vectors ...done.
Enrolling i-vectors .............done.
Enrollment complete.
```

Calibrate the i-vector system.

```
calibrate(soundClassifier,tdsTrain,labelsTrain)
```

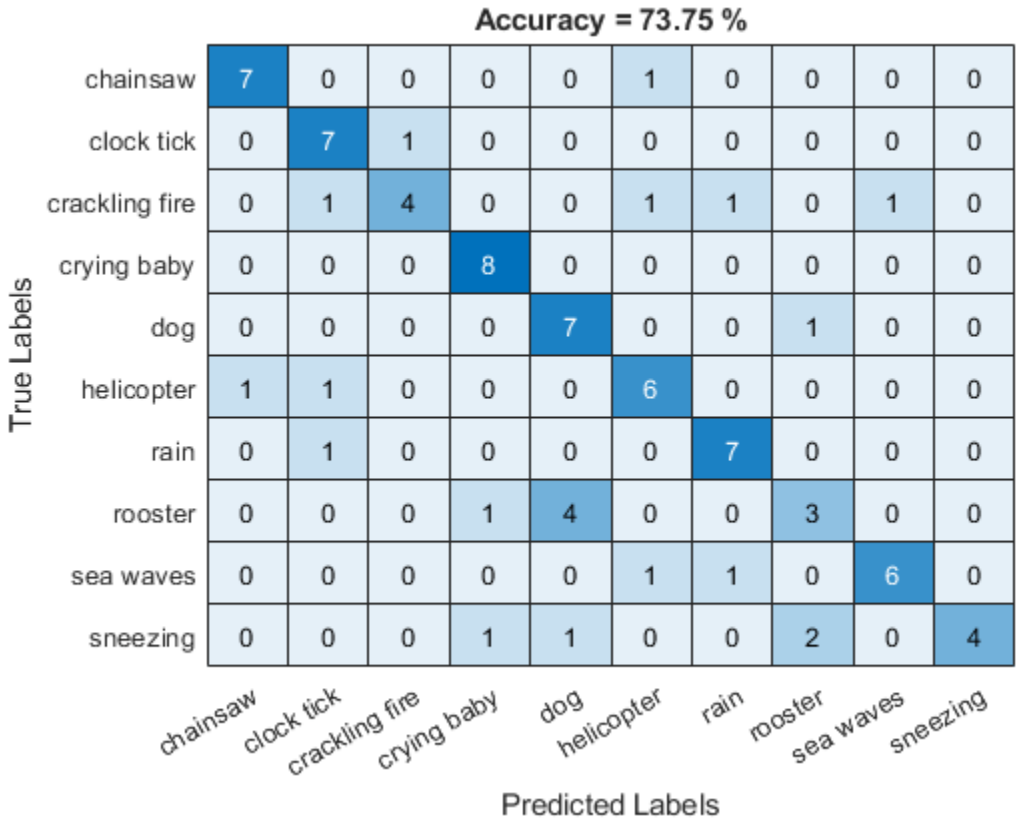
```
Extracting i-vectors ...done.
Calibrating CSS scorer ...done.
Calibration complete.
```

Use the `identify` function on the test set to return the system's inferred label.

```
inferredLabels = labelsTest;
inferredLabels(:) = inferredLabels(1);
for ii = 1:numel(labelsTest)
    features = read(tdsTest);
    tableOut = identify(soundClassifier,features,"css",NumCandidates=1);
    inferredLabels(ii) = tableOut.Label(1);
end
```

Create a confusion matrix to visualize performance on the test set.

```
uniqueLabels = unique(labelsTest);
cm = zeros(numel(uniqueLabels),numel(uniqueLabels));
for ii = 1:numel(uniqueLabels)
    for jj = 1:numel(uniqueLabels)
        cm(ii,jj) = sum((labelsTest==uniqueLabels(ii)) & (inferredLabels==uniqueLabels(jj)));
    end
end
labelStrings = replace(string(uniqueLabels),"_"," ");
heatmap(labelStrings,labelStrings,cm)
colorbar off
ylabel("True Labels")
xlabel("Predicted Labels")
accuracy = mean(inferredLabels==labelsTest);
title(sprintf("Accuracy = %0.2f %%",accuracy*100))
```



Release the i-vector system.

```
release(soundClassifier)
```

Supporting Functions

```
function writeFeatures(audioIn,info,~,afe)
    % Convet to single-precision
    audioIn = single(audioIn);

    % Extract features
    features = extract(afe,audioIn);

    % Replace the file extension of the suggested output name with MAT.
    filename = strrep(info.SuggestedOutputName,".wav",".mat");

    % Save the MFCC coefficients to the MAT file.
    save(filename,"features")
end
```

Input Arguments

`ivs` — i-vector system

`ivectorSystem` object

i-vector system, specified as an object of type `ivectorSystem`.

Version History

Introduced in R2021a

See Also

`trainExtractor` | `trainClassifier` | `calibrate` | `enroll` | `unenroll` | `detectionErrorTradeoff` | `verify` | `identify` | `info` | `addInfoHeader` | `ivector` | `ivectorSystem` | `speakerRecognition`

Blocks

Voice Activity Detector

Detect presence of speech in audio signal

Libraries:
Audio Toolbox / Measurements

Description

The Voice Activity Detector block detects the presence of speech in an audio signal. You can also use the Voice Activity Detector block to output an estimate of the noise variance per frequency bin.

Ports

Input

x — Input signal
matrix | 1-D vector

- Matrix input –– Each column of the input is treated as an independent channel.
- 1-D vector input –– The input is treated as a single channel.

This port is unnamed unless you specify additional input ports.

Data Types: `single` | `double`

SilenceToSpeech — Threshold (dB)
scalar in the range [0, 1]

Dependencies

To enable this port, select **Specify silence-to-speech probability from input port** for the "Probability of transition from a silence frame to a speech frame" on page 5-0 parameter.

Data Types: `single` | `double`

SpeechToSilence — Threshold (dB)
scalar in the range [0, 1]

Dependencies

To enable this port, select **Specify speech-to-silence probability from input port** for the "Probability of transition from a speech frame to a silence frame" on page 5-0 parameter.

Data Types: `single` | `double`

Output

P — Probability that speech is present
scalar | row vector

The block outputs a scalar or row vector with the same number of columns as the input signal.

This port is unnamed until you select the **Output noise variance** parameter.

Data Types: `single` | `double`

N — Estimate of noise variance per frequency bin
column vector | matrix

The block outputs a column vector or a matrix with the same number of columns as the input signal.

Dependencies

To enable this port, select the **Output noise variance** parameter.

Data Types: `single` | `double`

Parameters

If a parameter is listed as tunable, then you can change its value during simulation.

Domain of the input — Domain of the input
`Time` (default) | `Frequency`

Window — Windowing function applied before FFT
`Hann` (default) | `Chebyshev` | `Flat Top` | `Hamming` | `Kaiser` | `Rectangular`

The window function is designed using the algorithms of the following functions:

- `Hann` -- `hann`
- `Chebyshev` -- `chebwin`
- `Flat Top` -- `flattopwin`
- `Hamming` -- `hamming`
- `Kaiser` -- `kaiser`

Tunable: No

Dependencies

To enable this parameter, set **Domain of the input** to `Time`.

Sidelobe attenuation of the window (dB) — Sidelobe attenuation of the window (dB)
`60` (default) | positive finite scalar

Dependencies

To enable this parameter, set **Domain of the input** to `Time` and **Window** to `Chebyshev` or `Kaiser`.

Data Types: `single` | `double`

Inherit FFT length from input dimensions — Set FFT length to number of input samples
on (default) | off

Tunable: No

Dependencies

To enable this parameter, set **Domain of the input** to `Time`.

FFT length — Number of bins in frequency domain
`1024` (default) | positive integer

Tunable: No

Dependencies

To enable this parameter, set **Domain of the input** to `Time` and clear the **Inherit FFT length from input dimensions** parameter.

Data Types: `single` | `double`

Probability of transition from a silence frame to a speech frame — Probability that a speech frame follows a silence frame
`0.2` (default) | scalar in the range [0,1]

To specify **Probability of transition from a silence frame to a speech frame** from an input port, select **Specify silence-to-speech probability from input port**.

Tunable: Yes

Data Types: `single` | `double`

Probability of transition from a speech frame to a silence frame — Probability that a silence frame follows a speech frame
`0.1` (default) | scalar in the range [0,1]

To specify **Probability of transition from a speech frame to a silence frame** from an input port, select **Specify speech-to-silence probability from input port**.

Tunable: Yes

Data Types: `single` | `double`

Output noise variance — Output estimate of noise variance per frequency bin
`off` (default) | `on`

When you select this parameter, an additional output port, **N**, is added to the block.

Simulate using — Specify type of simulation to run
`Code generation` (default) | `Interpreted execution`

- `Code generation` – Simulate the model using generated C code. The first time you run a simulation, Simulink generates C code for the block. The C code is reused for subsequent simulations, as long as the model does not change. This option requires additional startup time, but the speed of the subsequent simulations is comparable to `Interpreted execution`.

- `Interpreted execution` – Simulate the model using the MATLAB interpreter. This option reduces startup time, but has a slower simulation speed than `Code generation`. In this mode, you can debug the source code of the block.

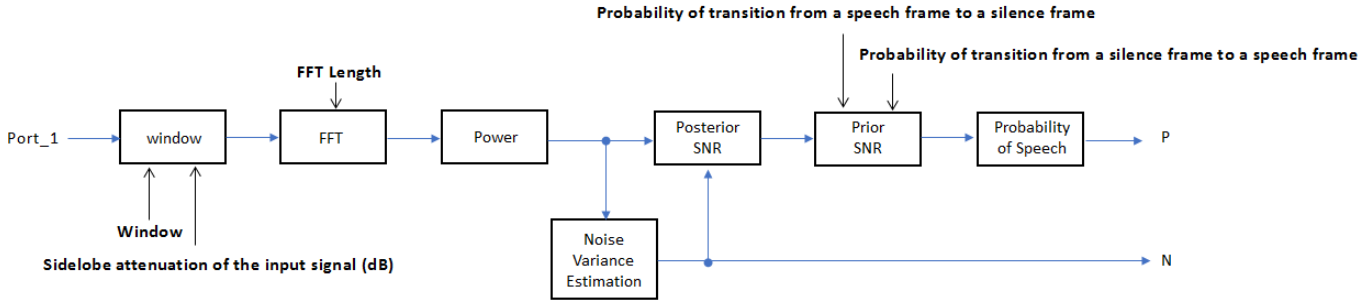
Tunable: No

Block Characteristics

Data Types	`double` | `single`
Direct Feedthrough	no
Multidimensional Signals	no
Variable-Size Signals	no
Zero-Crossing Detection	no

Algorithms

The Voice Activity Detector implements the algorithm described in [1].



If **Domain of the input** is specified as `Time`, the input signal is windowed and then converted to the frequency domain according to the **Window**, **Sidelobe attenuation of the window (dB)**, and **FFT length** parameters. If **Domain of the input** is specified as `Frequency`, the input is assumed to be a windowed discrete time Fourier transform (DTFT) of an audio signal. The signal is then converted to the power domain. Noise variance is estimated according to [2]. The posterior and prior SNR are estimated according to the Minimum Mean-Square Error (MMSE) formula described in [3]. A log likelihood ratio test with a Hidden Markov Model (HMM)-based hang-over scheme is used, according to [1].

Version History

Introduced in R2018a

References

[1] Sohn, Jongseo., Nam Soo Kim, and Wonyong Sung. "A Statistical Model-Based Voice Activity Detection." *Signal Processing Letters IEEE*. Vol. 6, No. 1, 1999.

[2] Martin, R. "Noise Power Spectral Density Estimation Based on Optimal Smoothing and Minimum Statistics." *IEEE Transactions on Speech and Audio Processing*. Vol. 9, No. 5, 2001, pp. 504–512.

[3] Ephraim, Y., and D. Malah. "Speech Enhancement Using a Minimum Mean-Square Error Short-Time Spectral Amplitude Estimator." *IEEE Transactions on Acoustics, Speech, and Signal Processing*. Vol. 32, No. 6, 1984, pp. 1109–1121.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

`voiceActivityDetector`

Cepstral Feature Extractor

(To be removed) Extract cepstral features from audio segment

Note The Cepstral Feature Extractor block will be removed in a future release. For more information, see “Version History”.

Libraries:
Audio Toolbox / Measurements

Description

The Cepstral Feature Extractor block extracts cepstral features from an audio segment. Cepstral features are commonly used to characterize speech and music signals.

Ports

Input

Port_1 — Audio input to cepstral feature extractor
column vector | matrix

Audio input to the cepstral feature extractor, specified as a column vector or a matrix. If specified as a matrix, the columns are treated as independent audio channels.

Data Types: `single` | `double`

Output

coeffs — Cepstral coefficients
column vector | matrix

Cepstral coefficients, returned as a column vector or a matrix. If the coefficients matrix is an *N*-by-*M* matrix, *N* is determined by the values you specify in the **Number of coefficients to return** and **Log energy usage** parameters. *M* equals the number of input audio channels.

When the **Log energy usage** parameter is set to:

- `Append` –– The block prepends the log energy value to the coefficients vector. The length of the coefficients vector is 1 + *NumCoeffs*, where *NumCoeffs* is the value specified in the **Number of coefficients to return** parameter.
- `Replace` –– The block replaces the first coefficient with the log energy of the signal. The length of the coefficients vector is *NumCoeffs*.
- `Ignore` –– The block does not calculate or return the log energy.

This port is unnamed until you select **Output delta** parameter, the **Output delta-delta** parameter, or both.

Data Types: `single` | `double`

delta — Change in coefficients
column vector | matrix

Change in coefficients over consecutive calls to the algorithm, returned as a column vector or a matrix. The **delta** array is of the same size and data type as the **coeffs** array.

Dependencies

To enable this port, select the **Output delta** parameter.

Data Types: `single` | `double`

deltaDelta — Change in delta values
column vector | matrix

Change in **delta** values over consecutive calls to the algorithm, returned as a column vector or a matrix. The **deltaDelta** array is the same size and data type as the **coeffs** and **delta** arrays.

Dependencies

To enable this port, select the **Output delta-delta** parameter.

Data Types: `single` | `double`

Parameters

If a parameter is listed as tunable, then you can change its value during simulation.

Filter bank type — Type of filter bank
`Mel` (default) | `Gammatone`

Type of filter bank, specified as either `Mel` or `Gammatone`:

- `Mel` –– The block computes the mel frequency cepstral coefficients (MFCC).
- `Gammatone` –– The block computes the gammatone cepstral coefficients (GTCC).

Tunable: No

Domain of the input signal — Input signal domain
`Time` (default) | `Frequency`

Input signal domain, specified as either `Time` or `Frequency`.

Tunable: No

Number of coefficients to return — Number of coefficients to return
13 (default) | positive integer

Number of coefficients to return, specified as an integer in the range [2, *v*], where *v* is the number of valid passbands. The number of valid passbands depends on the type of filter bank:

- `Mel` –– The number of valid passbands is defined as `sum(κ <= floor(fs/2))-2`, where `κ` is the number of band edges in the mel filter bank and `fs` is the sample rate.
- `Gammatone` –– The number of valid passbands is defined as `ceil(hz2erb(R(2))-hz2erb(R(1)))`, where `R` is the frequency range of the gammatone filter bank.

Tunable: No

Data Types: `single` | `double`

Nonlinear rectification — Type of nonlinear rectification
`Log` (default) | `Cubic-Root`

Type of nonlinear rectification applied prior to the discrete cosine transform.

Tunable: No

Inherit FFT length from input dimensions — Inherit FFT length from input
`on` (default) | `off`

When you select this parameter, the FFT length is equal to the number of rows in the input signal.

Tunable: No

Dependencies

To enable this parameter, set **Domain of the input signal** to `Time`.

FFTLength — FFT length
`[]` (default) | positive integer

FFT length, specified as a positive integer. The default, `[]`, means that the FFT length is equal to the number of rows in the input signal.

Tunable: No

Dependencies

To enable this parameter, set **Domain of the input signal** to `Time` and select the **Inherit FFT length from input dimensions** parameter.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Log energy usage — Specify how the log energy is shown
`Append` (default) | `Replace` | `Ignore`

Specify how the log energy is shown in the coefficients vector output, specified as:

- `Append` –– The block prepends the log energy to the coefficients vector. The length of the coefficients vector is 1 + *NumCoeffs*, where *NumCoeffs* is the value specified in the **Number of coefficients to return** parameter.
- `Replace` –– The block replaces the first coefficient with the log energy of the signal. The length of the coefficients vector is *NumCoeffs*.

- `Ignore` –– The block does not calculate or return the log energy.

Tunable: No

Output delta — Output delta values
`off` (default) | `on`

When you select this parameter, an additional output port, **delta**, is added to the block. This port outputs the change in coefficients over consecutive calls to the algorithm.

Tunable: No

Output delta-delta — Output delta-delta values
`off` (default) | `on`

When you select this parameter, an additional output port, **deltaDelta**, is added to the block. This port outputs the change in delta values over consecutive calls to the algorithm.

Tunable: No

Inherit sample rate from input — Specify source of input sample rate
`off` (default) | `on`

When you select this parameter, the block inherits its sample rate from the input signal. When you clear this parameter, you specify the sample rate in **Input sample rate (Hz)** parameter.

Tunable: No

Input sample rate (Hz) — Sample rate of input
`16000` (default) | positive scalar

Input sample rate in Hz, specified as a real positive scalar.

Dependencies

To enable this parameter, clear the **Inherit sample rate from input** parameter.

Simulate using — Specify type of simulation to run
`Code generation` (default) | `Interpreted execution`

- `Code generation` –– Simulate model using generated C code. The first time you run a simulation, Simulink generates C code for the block. The C code is reused for subsequent simulations, as long as the model does not change. This option requires additional startup time, but the speed of the subsequent simulations is comparable to `Interpreted execution`.
- `Interpreted execution` –– Simulate model using the MATLAB interpreter. This option shortens startup time but has a slower simulation speed than `Code generation`. In this mode, you can debug the source code of the block.

Tunable: No

Advanced Tab

Gammatone frequency range (Hz) — Frequency range of gammatone filter bank (Hz)
`[50 8000]` (default) | two-element row vector

Frequency range of the gammatone filter bank in Hz, specified as a positive, monotonically increasing two-element row vector. The maximum frequency range can be any finite number. The center frequencies of the filter bank are equally spaced across the frequency range on the ERB scale.

Tunable: No

Dependencies

To enable this parameter, set **Filter bank type** to `Gammatone`.

Band edges of Mel filter bank (Hz) — Band edges of mel filter bank
row vector

Band edges of the filter bank in Hz, specified as a nonnegative monotonically increasing row vector in the range [0, ∞). The maximum bandedge frequency can be any finite number. The number of bandedges must be in the range [4, 80].

The default band edges are spaced linearly for the first ten and then logarithmically thereafter. The default band edges are set as recommended by [1].

Tunable: No

Dependencies

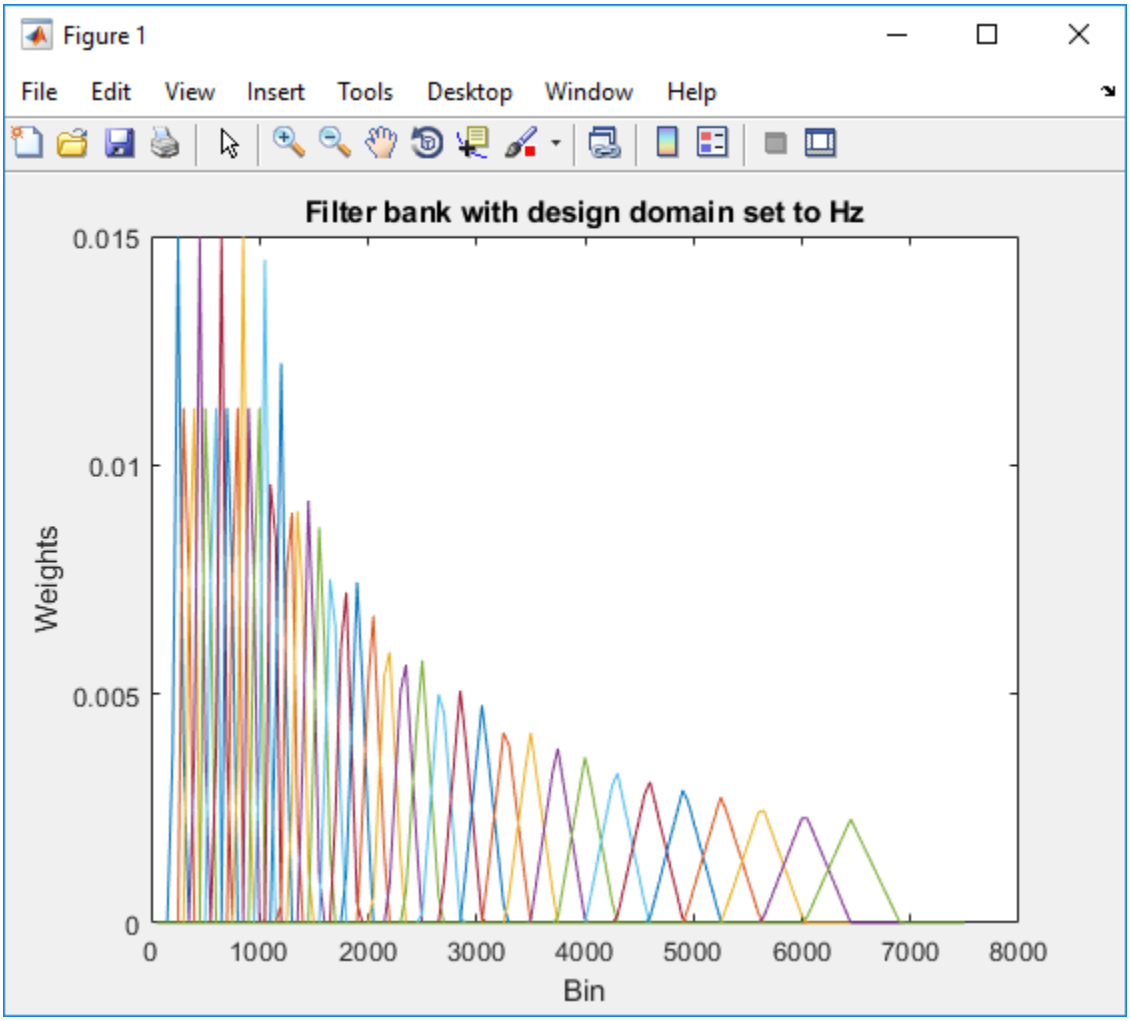
To enable this parameter, set **Filter bank type** to `Mel`.

Domain for Mel filter bank design — Mel filter bank design domain
`Hz` (default) | `Bin`

Mel filter bank design domain, specified as either `Hz` or `Bin`. The filter bank is designed as overlapped triangles with band edges specified by the **Band edges of filter bank (Hz)** parameter.

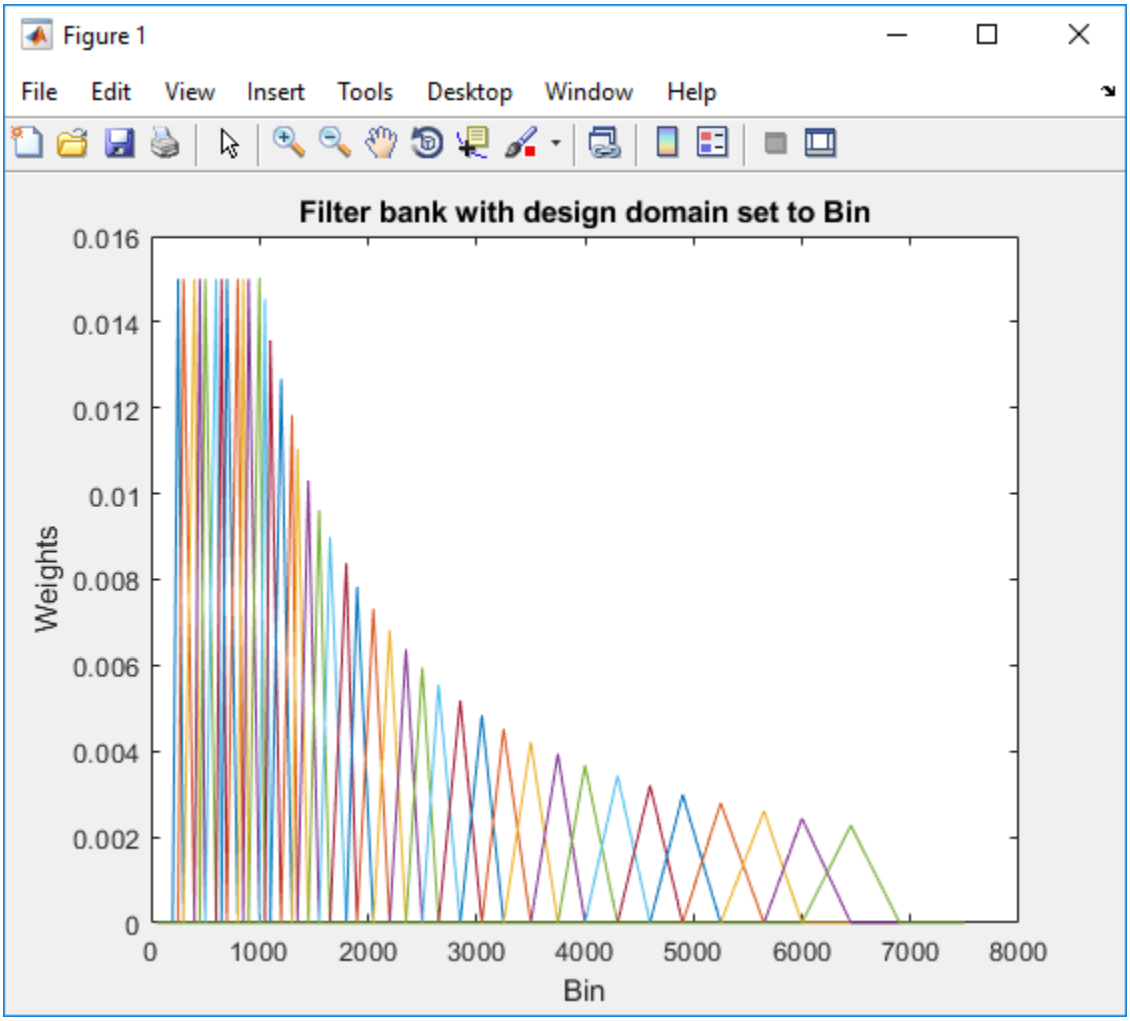
The band edges are specified in Hz. When you set the design domain to:

- `Hz` –– Filter bank triangles are drawn in Hz and are mapped onto bins.



For details, see [1].

- `Bin` –– The band edge frequencies in `Hz` are converted to bins. The filter bank triangles are drawn symmetrically in bins.



For details, see [2].

Tunable: No

Dependencies

To enable this parameter, set **Filter bank type** to `Mel`.

Filter bank normalization — Normalize filter bank
`Bandwidth` (default) | `Area` | `None`

Normalization technique used to normalize the weights of the filter bank, specified as:

- `Bandwidth` –– The weights of each bandpass filter are normalized by the corresponding bandwidth of the filter.
- `Area` –– The weights of each bandpass filter are normalized by the corresponding area of the bandpass filter.
- `None` –– The weights of the filter are not normalized.

Tunable: No

Block Characteristics

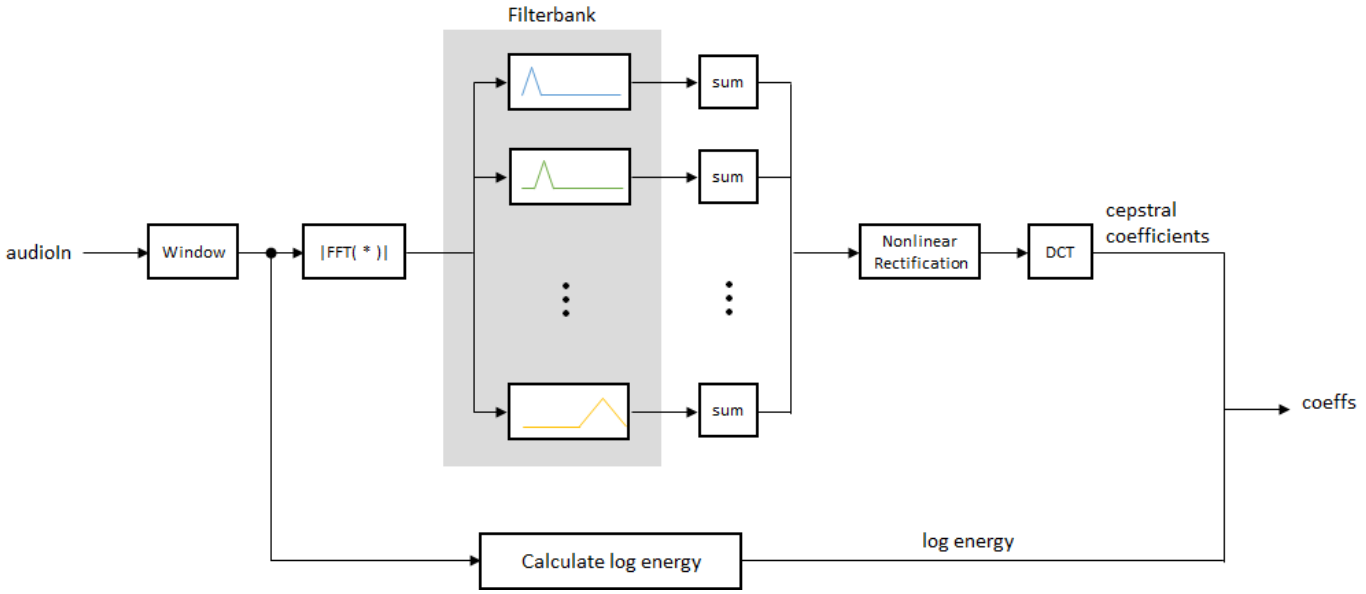
Data Types	`double` | `single`
Direct Feedthrough	no
Multidimensional Signals	no
Variable-Size Signals	no
Zero-Crossing Detection	no

Algorithms

Auditory Cepstrum Coefficients

Auditory cepstrum coefficients are popular features extracted from speech signals for use in recognition tasks. In the source-filter model of speech, cepstral coefficients are understood to represent the filter (vocal tract). The vocal tract frequency response is relatively smooth, whereas the source of voiced speech can be modeled as an impulse train. As a result, the vocal tract can be estimated by the spectral envelope of a speech segment.

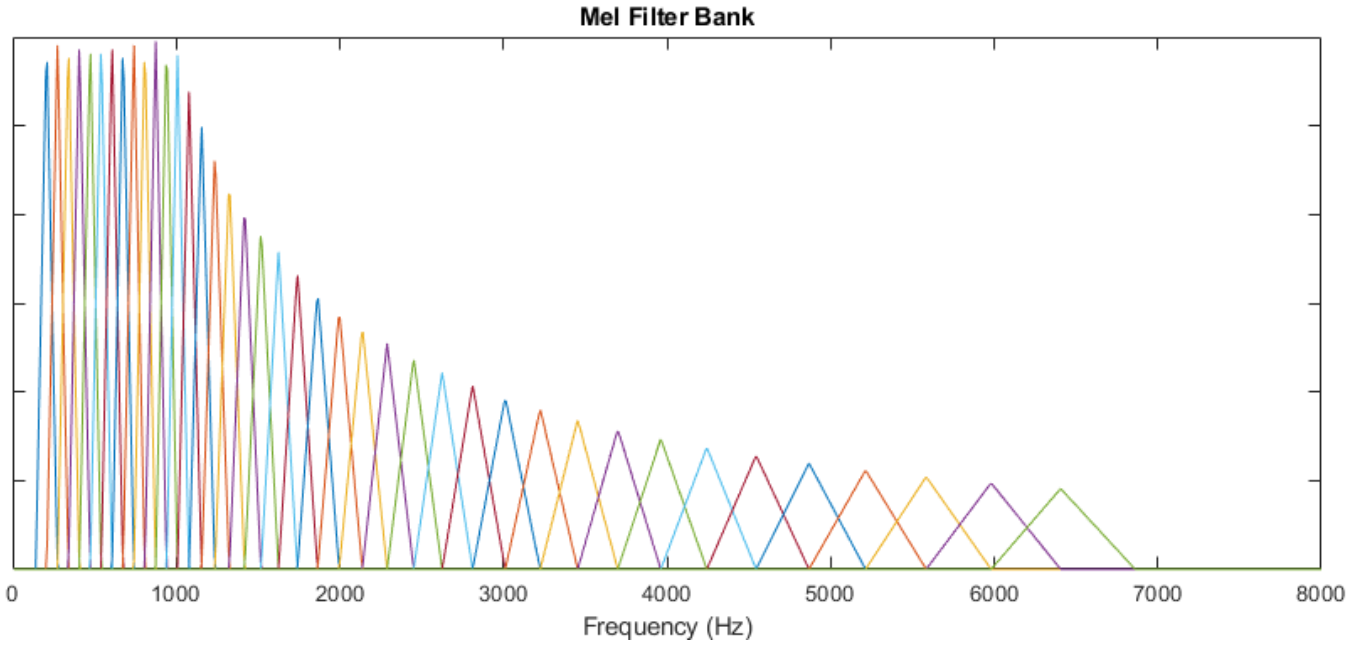
The motivating idea of cepstral coefficients is to compress information about the vocal tract (smoothed spectrum) into a small number of coefficients based on an understanding of the cochlea. Although there is no hard standard for calculating the coefficients, the basic steps are outlined by the diagram.



Two popular implementations of the filter bank are the mel filter bank and the gammatone filter bank.

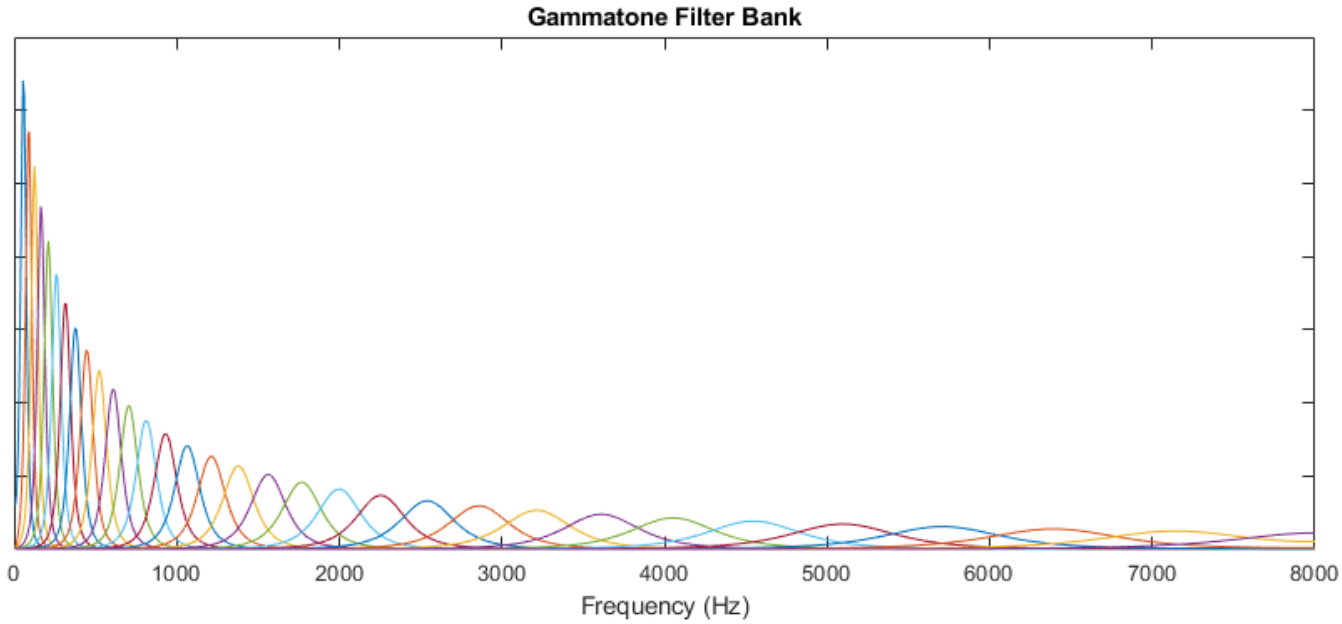
Mel Filter Bank

The default mel filter bank linearly spaces the first 10 triangular filters and logarithmically spaces the remaining filters.



Gammatone Filter Bank

The default gammatone filter bank is composed of gammatone filters spaced linearly on the ERB scale between 50 and 8000 Hz. The filter bank is designed by `gammatoneFilterBank`.



Log Energy

If the input (x) is a time-domain signal, the log energy is computed using the following equation:

$$\log E = \log(\text{sum}(x^2))$$

If the input (x) is a frequency-domain signal, the log energy is computed using the following equation:

$$\log E = \log\left(\text{sum}\left(|x|^2\right)/FFTLength\right)$$

Version History

Introduced in R2018a

R2022b: To be removed

Not recommended starting in R2022b

The Cepstral Feature Extractor block will be removed in a future release. Use the MFCC block or a combination of the Auditory Spectrogram, Cepstral Coefficients, and Audio Delta blocks instead.

Cepstral Feature Extractor Configuration	Recommended Replacement
Filter bank type parameter set to `Mel`	Use the MFCC block.
Filter bank type parameter set to `Gammatone`	Use the Auditory Spectrogram block combined with the Cepstral Coefficients block. See “Extract GTCC from Audio in Simulink” for an example.
Output delta or **Output delta-delta** parameters selected	If using the MFCC block, select the **Append delta** or **Append delta-delta** parameters. If using the Cepstral Coefficients block instead, use the Audio Delta block to extract delta features.
Log energy usage parameter set to `Append` or `Replace`	No replacement
Band edges of Mel filter bank (Hz) parameter specified	No replacement
Domain for Mel filter bank design parameter set to `Bin`	No replacement

References

[1] Auditory Toolbox. https://engineering.purdue.edu/~malcolm/interval/1998-010/AuditoryToolboxTechReport.pdf

[2] ETSI ES 201 108 V1.1.3 (2003-09). https://www.etsi.org/deliver/etsi_es/201100_201199/201108/01.01.03_60/es_201108v010103p.pdf

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

MFCC | Cepstral Coefficients | `mfcc` | `gtcc` | `cepstralCoefficients` | `audioFeatureExtractor`

Audio Delta

Compute delta features

Libraries:
Audio Toolbox / Features

Description

The Audio Delta block computes the delta of the input audio features. The delta is an approximation of the first derivative of the audio features with respect to time.

Ports

Input

Port_1 — Audio features
scalar | vector | matrix | 3-D array

Audio features, specified as a scalar, vector, matrix, or 3-D array. The delta computation operates along the first dimension. All other dimensions are treated as independent channels.

Data Types: `single` | `double`

Output

Port_1 — Delta of audio features
scalar | vector | matrix | array

Delta of audio features, returned as an array that is the same size and data type as the input.

Data Types: `single` | `double`

Parameters

Delta window length — Window length over which to calculate delta

9 (default) | odd integer greater than 2

Window length over which to calculate delta, specified as an odd integer greater than 2.

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 but has a slower simulation speed than `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`.

Block Characteristics

Data Types	`double` | `single`
Direct Feedthrough	`no`
Multidimensional Signals	`no`
Variable-Size Signals	`yes`
Zero-Crossing Detection	`no`

Algorithms

The delta of an audio feature x is a least-squares approximation of the local slope of a region centered on sample $x(k)$, which includes M samples before the current sample and M samples after the current sample.

$$delta = \frac{\sum_{k=-M}^{M} k\,x(k)}{\sum_{k=-M}^{M} k^2}$$

The delta window length defines the length of the region from $-M$ to M.

For more information, see [1].

Version History

Introduced in R2022b

References

[1] Rabiner, Lawrence R., and Ronald W. Schafer. *Theory and Applications of Digital Speech Processing*. Upper Saddle River, NJ: Pearson, 2010.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

Blocks

MFCC | Cepstral Coefficients

Functions
`mfcc` | `audioDelta` | `cepstralCoefficients`

Objects
`audioFeatureExtractor`

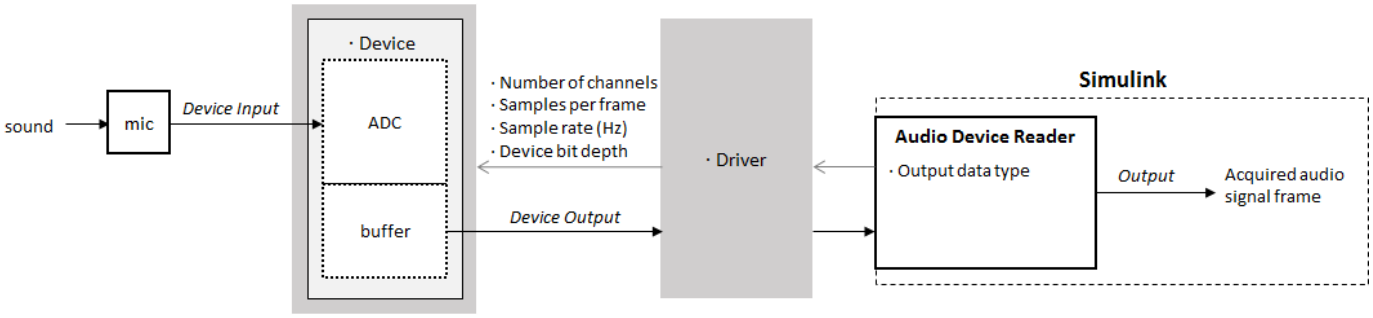
Audio Device Reader

Record from sound card

Libraries:
Audio 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`

O — Number of samples overrun
scalar

This port outputs the number of samples overrun while acquiring a frame of data (one output matrix).

Dependencies

To enable this port, select the **Output number of samples overrun** parameter.

Data Types: `uint32`

Parameters

Main Tab

Driver — Driver used to access your audio device
`DirectSound` (default) | `ASIO` | `WASAPI`

- ASIO drivers do not come pre-installed on Windows machines. To use the `ASIO` driver option, install an ASIO driver outside of MATLAB.

  Note If **Driver** is set to `ASIO`, open the ASIO UI outside of MATLAB to set the sound card buffer size to the value specified by the **Samples per frame** parameter. See the documentation of your ASIO driver for more information.

- WASAPI drivers are supported for exclusive-mode only.

ASIO and WASAPI drivers do not provide sample rate conversion. For ASIO and WASAPI drivers, set **Sample rate (Hz)** to a sample rate supported by your audio device.

This parameter applies only on Windows machines. Linux machines always use the ALSA driver. Mac machines always use the CoreAudio driver.

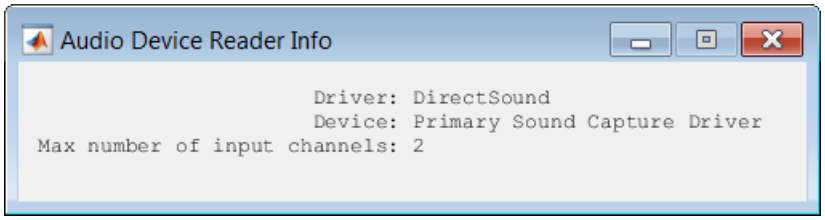
Device — Device used to acquire audio samples

default audio device (default)

The device list is populated with devices available on your computer.

Info — View information about your audio input configuration
button

This button opens a dialog box that lists your selected audio driver, the full name of your audio device, and the maximum input channels for your configuration. For example:



Sample rate (Hz) — Sample rate your device uses to acquire audio data
`44100` (default) | integer

The possible range of **Sample rate (Hz)** depends on your audio hardware.

Number of channels — Number of channels acquired by your audio device
`1` (default) | integer

The number of input channels is also the number of channels (matrix columns) output by the Audio Device Reader block.

Dependencies

To specify which input channels your audio device acquires, on the **Advanced** tab, select the **Use default channel mapping** 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`

Data type used by device to acquire audio data, specified as a character vector or string.

Use default channel mapping — 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`

Data type of the output.

Note If this parameter is specified as `double` or `single`, the block outputs data in the range [–1, 1]. For other data types, the range is [min, max] of the specified data type.

Block Characteristics

Data Types	`double` | `integer`[a] | `single`
Direct Feedthrough	no
Multidimensional Signals	no
Variable-Size Signals	no
Zero-Crossing Detection	no

a Supports 16- and 32-bit signed and 8-bit unsigned integers.

Version History

Introduced in R2016a

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

The executable generated from this block relies on prebuilt dynamic library files (.dll files) included with MATLAB. Use the `packNGo` function to package the code generated from this object and all the relevant files in a compressed zip file. Using this zip file, you can relocate, unpack, and rebuild your project in another development environment where MATLAB is not installed. For more details, see "Run Audio I/O Features Outside MATLAB and Simulink".

See Also

`audioDeviceReader` | `audioDeviceWriter` | Audio Device Writer

Topics

"Run Audio I/O Features Outside MATLAB and Simulink"
"Audio I/O: Buffering, Latency, and Throughput"

Audio Device Writer

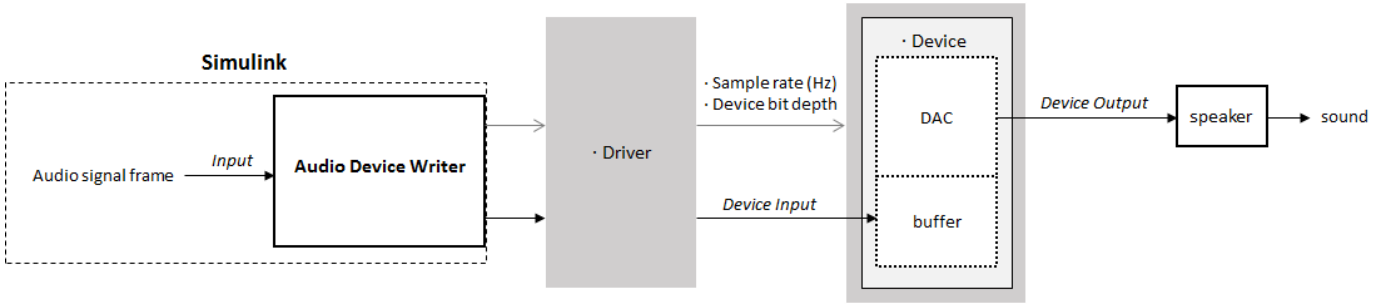
Play to sound card

Libraries:
Audio Toolbox / Sinks
DSP System Toolbox / Sinks

Description

The Audio Device Writer block writes audio samples to an audio output device.

Parameters of the Audio Device Writer block specify the driver, the device, and device attributes such as sample rate and bit depth.



Ports

Input

Port_1 — Input signal
scalar | vector | matrix

If input to the Audio Device Writer block is of data type `double` or `single`, the block clips values outside the range [–1, 1]. For other data types, the allowed input range is [min, max] of the specified data type.

Data Types: `single` | `double` | `int16` | `int32` | `uint8`

Output

Port_1 — Number of samples underrun
scalar

This port outputs the number of samples underrun while writing a frame of data (one input matrix).

Dependencies

To enable this port, select the **Output number of samples underrun** parameter.

Data Types: `uint32`

Parameters

Main Tab

Driver — Driver used to access your audio device
`DirectSound` (default) | `ASIO` | `WASAPI`

- ASIO drivers do not come pre-installed on Windows machines. To use the `ASIO` driver option, install an ASIO driver outside of MATLAB.

  Note If **Driver** is set to `ASIO`, open the ASIO UI outside of MATLAB to set the sound card buffer size to the frame size (number of rows) input to the Audio Device Writer block. See the documentation of your ASIO driver for more information.

- WASAPI drivers are supported for exclusive-mode only.

ASIO and WASAPI drivers do not provide sample rate conversion. For ASIO and WASAPI drivers, supply an audio stream with a sample rate supported by your audio device.

This parameter applies only on Windows machines. Linux machines always use the ALSA driver. Mac machines always use the CoreAudio driver.

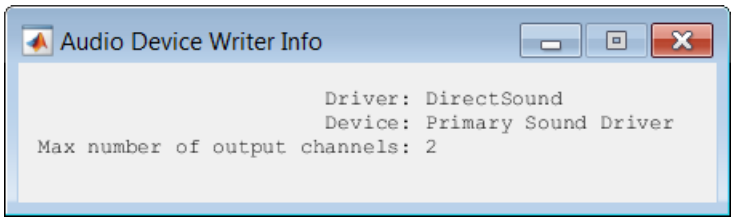
To specify nondefault **Driver** values, you must install Audio Toolbox. If the toolbox is not installed, specifying nondefault **Driver** values returns an error.

Device — Device used to play audio samples
default audio device (default)

The device list is populated with devices available on your computer.

Info — View information about your audio output configuration
button

This button opens a dialog box that lists your selected audio driver, the full name of your audio device, and the maximum output channels for your configuration. For example:



Inherit sample rate from input — Specify source of input sample rate
on (default) | off

When you select this parameter, the block inherits its sample rate from the input signal. When you clear this parameter, you specify the sample rate in **Sample rate (Hz)**.

Sample rate (Hz) — Sample rate used by device to play audio data
`44100` (default) | positive scalar

The possible range of **Sample rate (Hz)** depends on your audio hardware.

Dependencies

To enable this parameter, clear the **Inherit sample rate from input** parameter.

Advanced Tab

Device bit depth — Data type used by device to perform digital-to-analog conversion
`16-bit integer` (default) | `8-bit integer` | `24-bit integer` | `32-bit float`

Before performing digital-to-analog conversion, the input data is cast to a data type specified by this parameter.

Note To specify a nondefault **Device bit depth**, you must install Audio Toolbox. If the toolbox is not installed, specifying a nondefault **Device bit depth** returns an error.

Use default channel mapping — 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 Toolbox. If the toolbox is not installed, specifying nondefault values for **Device output channels** returns an error.

Dependencies

To enable this parameter, clear the **Use default mapping between columns of input of this block and sound card's output channels** parameter.

Output number of samples underrun — Specify output port for number of samples underrun
off (default) | on

When you select this parameter, an output port is added to the block. The port outputs the number of samples underrun while writing a frame of data (one input matrix).

Block Characteristics

Data Types	`double` | `integer`[a] | `single`
Direct Feedthrough	`no`
Multidimensional Signals	`no`
Variable-Size Signals	`yes`
Zero-Crossing Detection	`no`

a Supports 16- and 32-bit signed and 8-bit unsigned integers.

Version History

Introduced in R2016a

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

The following code generation limitations apply:

- Host computer only. Excludes Simulink Desktop Real-Time™ code generation.
- 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 "Run Audio I/O Features Outside MATLAB and Simulink".

See Also

Audio Device Reader | Binary File Reader | `audioDeviceWriter` | `audioDeviceReader`

Topics

"Run Audio I/O Features Outside MATLAB and Simulink"
"Audio I/O: Buffering, Latency, and Throughput"

Auditory Spectrogram

Extract mel, Bark, or ERB spectrogram from audio

Libraries:
Audio Toolbox / Features

Description

The Auditory Spectrogram block extracts a spectrogram from the audio input signal. A spectrogram contains an estimate of the short-term, time-localized frequency content of the input signal.

Ports

Input

Port_1 — Audio input
column vector | matrix

Audio input signal, specified as a column vector or a matrix. When you specify a matrix, the block treats columns as independent audio channels.

Data Types: `single` | `double`

Output

spec — Spectrogram
matrix | 3-D array

Spectrogram, returned as a matrix or 3-D array. The dimensions of **spec** are *L*-by-*M*-by-*N*, where:

- *L* is the number of spectra, which is determined by the **Number of spectra** parameter.
- *M* is the number of bands, which is determined by the **Auto-determine number of bands** and **Number of bands** parameters.
- *N* is the number of channels in the input audio signal.

Trailing singleton dimensions are removed from the output.

This port is unnamed until you select the **Output center frequencies** parameter.

Data Types: `single` | `double`

fvec — Center frequencies
row vector

Center frequencies of the bandpass filters in Hz, returned as a row vector with number of elements equal to the number of bands.

Dependencies

To enable this port, select the **Output center frequencies** parameter.

Data Types: `single` | `double`

Parameters

Filter Bank Parameters

Frequency scale — Frequency scale of filter bank

`mel` (default) | `bark` | `erb`

Frequency scale used to design the auditory filter bank, specified as `mel`, `bark`, or `erb`.

- `mel` –– Design the filter bank as half-overlapped triangles equally spaced on the mel scale.
- `bark` –– Design the filter bank as half-overlapped triangles equally spaced on the Bark scale.
- `erb` –– Design the filter bank as gammatone filters whose center frequencies are equally spaced on the ERB scale.

Auto-determine number of bands — Automatically determine number of bandpass filters

`on` (default) | `off`

When you select this parameter, the block automatically determines the number of bandpass filters based on the **Frequency scale** parameter.

- If you set **Frequency scale** to `mel` or `bark`, then the number of bands is `32`.
- If you set **Frequency scale** to `erb`, then the number of bands is equal to `ceil(hz2erb(fr(2))-hz2erb(fr(1)))`, where `fr` is specified using **Frequency range (Hz)**.

Number of bands — Number of bandpass filters

`32` (default) | positive integer

Number of bandpass filters, specified as a positive integer.

Dependencies

To enable this parameter, clear the **Auto-determine number of bands** parameter.

Auto-determine frequency range — Automatically determine frequency range

`on` (default) | `off`

When you select this parameter, the block sets the **Frequency range** to `[0,fs/2]`, where `fs` is the sample rate. The sample rate is determined by the **Inherit sample rate from input** and **Input sample rate (Hz)** parameters.

Frequency range (Hz) — Frequency range over which to design auditory filter bank

`[0,22050]` (default) | two-element row vector

Frequency range in Hz over which to design the auditory filter bank, specified as a two-element row vector.

Dependencies

To enable this parameter, clear the **Auto-determine frequency range** parameter.

Filter bank design domain — Domain to design filter bank

`linear` (default) | `warped`

Domain in which the block designs the filter bank, specified as `linear` or `warped`. Set the filter bank design domain to `linear` to design the bandpass filters in the linear (Hz) domain. Set the filter bank design domain to `warped` to design the bandpass filters in the warped (mel or Bark) domain.

Dependencies

To enable this parameter, set **Frequency scale** to `mel` or `bark`.

Filter bank normalization — Normalization technique for filter bank

`bandwidth` (default) | `area` | `none`

Normalization technique used for the filter bank weights, specified as `bandwidth`, `area`, or `none`.

- `bandwidth` –– Normalize the weights of each bandpass filter by the corresponding bandwidth of the filter.
- `area` –– Normalize the weights of each bandpass filter by the corresponding area of the bandpass filter.
- `none` –– The block does not normalize the weights of the filters.

Output center frequencies — Specify additional output port for center frequencies

`off` (default) | `on`

When you select this parameter, the block displays an additional output port, **fvec**. This port outputs the center frequencies of the bandpass filters.

Visualize filter bank — Open plot to visualize filter bank

button

Open plot to visualize the filters in the frequency domain.

Spectrogram Parameters

Window — Analysis window

`hamming(1024,'periodic')` (default) | real vector

Analysis window applied in the time domain, specified as a real vector.

Normalize window — Normalize analysis window

`on` (default) | `off`

When you select this parameter, the block applies window normalization.

Overlap length — Overlap length of adjacent analysis windows

`512` (default) | integer in the range [0, `windowLength`)

Overlap length of adjacent analysis windows, specified as an integer in the range [0, `windowLength`), where `windowLength` is the length of the analysis window, which is specified by **Window**.

Auto-determine FFT length — Automatically determine FFT length

`on` (default) | `off`

When you select this parameter, the block automatically sets the FFT length to the window length `numel(Window)`.

FFT length — Number of DFT points

`1024` (default) | positive integer

Number of points used to calculate the DFT, specified as a positive integer.

Dependencies

To enable this parameter, clear the **Auto-determine FFT length** parameter.

Spectrum type — Type of spectrum

`magnitude` (default) | `power`

Type of spectrum, specified as `magnitude` or `power`.

Number of spectra — Number of spectra

`1` (default) | positive integer

Number of spectra in the spectrogram, specified as a positive integer.

Number of spectra overlap — Number of overlapped spectra

`0` (default) | integer in the range [0, **Number of spectra**)

Number of spectra overlapped across consecutive spectrograms, specified as an integer in the range [0, **Number of spectra**).

Simulation Parameters

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 the **Input sample rate (Hz)** parameter.

Input sample rate (Hz) — Sample rate of input

`44.1e3` (default) | positive scalar

Input sample rate in Hz, specified as a real positive scalar.

Dependencies

To enable this parameter, clear the **Inherit sample rate from input** parameter.

Block Characteristics

Data Types	`double` | `single`
Direct Feedthrough	`no`
Multidimensional Signals	`no`
Variable-Size Signals	`no`
Zero-Crossing Detection	`no`

Version History

Introduced in R2022a

R2023a: Generate optimized C/C++ code for computing auditory spectrogram

The Auditory Spectrogram block supports optimized C/C++ code generation using single instruction, multiple data (SIMD) instructions.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

The Auditory Spectrogram block supports optimized code generation using single instruction, multiple data (SIMD) instructions. For more information about SIMD code generation, see “Generate SIMD Code from Simulink Blocks” (Simulink Coder).

See Also

Blocks
Design Auditory Filter Bank | Design Mel Filter Bank | Mel Spectrogram | Cepstral Coefficients

Functions
`designAuditoryFilterBank` | `melSpectrogram`

Objects
`audioFeatureExtractor`

Cepstral Coefficients

Extract cepstral coefficients from spectrogram

Libraries:
Audio Toolbox / Features

Description

The Cepstral Coefficients block extracts the cepstral coefficients from a real-valued spectrogram or auditory spectrogram. Cepstral coefficients are commonly used as compact representations of audio signals.

Ports

Input

Port_1 — Spectrogram or auditory spectrogram
matrix | 3-D array

Spectrogram or auditory spectrogram, specified as an *L*-by-*M* matrix or *L*-by-*M*-by-*N* array, where:

- *L* is the number of frequency bands.
- *M* is the number of frames.
- *N* is the number of channels.

Data Types: `single` | `double`

Output

Port_1 — Cepstral Coefficients
matrix | 3-D array

Cepstral coefficients, returned as an *M*-by-*B* matrix or *M*-by-*B*-by-*N* array, where:

- *M* is the number of frames in the input spectrogram.
- *B* is the number of coefficients returned per frame, which is specified by the **Number of cepstral coefficients** parameter.
- *N* is the number of channels in the input spectrogram.

Data Types: `single` | `double`

Parameters

Number of cepstral coefficients — Number of cepstral coefficients returned

13 (default) | positive integer greater than 1

Number of cepstral coefficients, specified as a positive integer greater than 1.

Rectification — Type of nonlinear rectification

`Logarithm` (default) | `Cubic root` | `None`

Type of nonlinear rectification applied to the spectrum prior to the discrete cosine transform, specified as `Logarithm`, `Cubic root`, or `None`.

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 but has a slower simulation speed than `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`.

Block Characteristics

Data Types	`double` | `single`
Direct Feedthrough	`no`
Multidimensional Signals	`no`
Variable-Size Signals	`no`
Zero-Crossing Detection	`no`

Algorithms

Given an auditory spectrogram, the algorithm to extract *N* cepstral coefficients from each individual spectrum comprises the following steps.

1. Rectify the spectrum by applying a logarithm, cubic root, or optionally perform no rectification.
2. Apply the discrete cosine transform (DCT-II) to the rectified spectrum.
3. Return the first *N* coefficients from the cepstral representation.

For more information, see [1].

Version History

Introduced in R2022b

References

[1] Rabiner, Lawrence R., and Ronald W. Schafer. *Theory and Applications of Digital Speech Processing*. Upper Saddle River, NJ: Pearson, 2010.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

Blocks

MFCC | Audio Delta | Auditory Spectrogram | Design Mel Filter Bank

Functions

`mfcc` | `audioDelta` | `cepstralCoefficients` | `designAuditoryFilterBank` | `melSpectrogram`

Objects

`audioFeatureExtractor`

Compressor

Dynamic range compressor

Libraries:
Audio Toolbox / Dynamic Range Control

Description

The Compressor block performs dynamic range compression independently across each input channel. Dynamic range compression attenuates the volume of loud sounds that cross a given threshold. The block uses specified attack and release times to achieve a smooth applied gain curve.

Ports

Input

x — Input signal
matrix | 1-D vector

- Matrix input –– Each column of the input is treated as an independent channel.
- 1-D vector input –– The input is treated as a single channel.

This port is unnamed unless you specify additional input ports.

Data Types: `single` | `double`

T — Threshold (dB)
scalar

Dependencies

To enable this port, select **Specify from input port** for the “Threshold (dB)” on page 5-0 parameter.

Data Types: `single` | `double`

R — Ratio
scalar

Dependencies

To enable this port, select **Specify from input port** for the “Ratio” on page 5-0 parameter.

Data Types: `single` | `double`

K — Knee width (dB)
scalar

Dependencies

To enable this port, select **Specify from input port** for the “Knee width (dB)” on page 5-0 parameter.

Data Types: `single` | `double`

AT — Attack time (s)
scalar

Dependencies

To enable this port, select **Specify from input port** for the "Attack time (s)" on page 5-0 parameter.

Data Types: `single` | `double`

RT — Release time (s)
scalar

Dependencies

To enable this port, select **Specify from input port** for the "Release time (s)" on page 5-0 parameter.

Data Types: `single` | `double`

Output

Y — Output signal
matrix

The Compressor block outputs a signal with the same data type as the input signal. The size of the output depends on the size of the input:

- Matrix input –– The block outputs a matrix the same size and data type as the input signal.
- 1-D vector input –– The block outputs an *N*-by-1 matrix (column vector), where *N* is the number of elements in the 1-D vector.

Data Types: `single` | `double`

G — Gain applied to each input sample
matrix

Dependencies

To enable this port, select the **Output gain (dB)** parameter.

Data Types: `single` | `double`

Parameters

If a parameter is listed as tunable, then you can change its value during simulation.

Main Tab

Threshold (dB) — Operation threshold
`–10` (default) | scalar in the range –50 to 0 inclusive

Operation threshold is the level above which gain is applied to the input signal.

To specify **Threshold (dB)** from an input port, select **Specify from input port** for the parameter.

Tunable: Yes

Ratio — Compression ratio
5 (default) | scalar in the range 1 to 50 inclusive

Compression ratio is the input/output ratio for signals that overshoot the operation threshold.

Assuming a hard knee characteristic and a steady-state input such that $x[n]$ dB > **Threshold (dB)**, the compression ratio is defined as $R = \frac{(x[n] - T)}{(y[n] - T)}$, where

- R is the compression ratio.
- $x[n]$ is the input signal in dB.
- $y[n]$ is the output signal in dB.
- T is the threshold in dB.

To specify **Ratio** from an input port, select **Specify from input port** for the parameter.

Tunable: Yes

Knee width (dB) — Transition area in compression characteristic
`0` (default) | scalar in the range 0 to 20 inclusive

For soft knee characteristics, the transition area is defined by the relation

$$y = x + \frac{\left(\frac{1}{R} - 1\right) \times \left(x - T + \frac{W}{2}\right)^2}{(2 \times W)}$$

for the range $(2 \times |x - T|) \leq W$, where

- y is the output level in dB.
- x is the input level in dB.
- R is the compression ratio.
- T is the threshold in dB.
- W is the knee width in dB.

To specify **Knee width (dB)** from an input port, select **Specify from input port** for the parameter.

Tunable: Yes

View static characteristic — Open static characteristic plot of dynamic range compressor
button

The plot is updated automatically when parameters of the Compressor block change.

Tunable: Yes

Attack time (s) — Time for applied gain to ramp up
`0.05` (default) | scalar in the range 0 to 4 inclusive

Attack time is the time the compressor gain takes to rise from 10% to 90% of its final value when the input goes above the threshold. The **Attack time (s)** parameter smooths the applied gain curve.

To specify **Attack time (s)** from an input port, select **Specify from input port** for the parameter.

Tunable: Yes

Release time (s) — Time for applied gain to ramp down
`0.2` (default) | scalar in the range 0 to 4 inclusive

Release time is the time the compressor gain takes to drop from 90% to 10% of its final value when the input goes below the threshold. The **Release time (s)** parameter smooths the applied gain curve.

To specify **Release time (s)** from an input port, select **Specify from input port** for the parameter.

Tunable: Yes

Make-up gain mode — Make-up gain mode
`Property` (default) | `Auto`

- `Property` –– Make-up gain is set to the value specified by the **Make-up gain (dB)** parameter.
- `Auto` –– Make-up gain is applied at the output of the Compressor block such that a steady-state 0 dB input has a 0 dB output.

Tunable: No

Make-up gain (dB) — Applied make-up gain
`0` (default) | scalar in the range –10 to 24 inclusive

Make-up gain compensates for gain lost during compression. It is applied at the output of the Compressor block.

Tunable: Yes

Dependencies

To enable this parameter, set the **Make-up gain mode** parameter to `Property`.

Inherit sample rate from input — Specify source of input sample rate
`on` (default) | `off`

When you select this parameter, the block inherits its sample rate from the input signal. When you clear this parameter, specify the sample rate in the **Input sample rate (Hz)** parameter.

Tunable: No

Input sample rate (Hz) — Sample rate of input
`44100` (default) | positive scalar

Tunable: Yes

Dependencies

To enable this parameter, clear the **Inherit sample rate from input** parameter.

Advanced Tab

Output gain (dB) — Gain applied on each input sample
`off` (default) | `on`

When you select this parameter, an additional output port, **G**, is added to the block. The **G** port outputs the gain applied on each input channel in dB.

Tunable: No

Sidechain — Enable sidechain input
`off` (default) | `on`

When you select this parameter, an additional input port **SC** is added to the block. The **SC** port enables dynamic range compression of the input signal **x** using a separate sidechain signal.

The datatype and (frame) length input to the **SC** port must be the same as the input to the **x** port.

The number of channels of the sidechain input must be equal to the number of channels of `x` or be equal to one.

- `Sidechain channel count is equal to one` –– The computed gain, `G`, based on this channel is applied to all channels of `x`.
- `Sidechain channel count is equal to channel count of x` –– The computed gain, `G`, for each sidechain channel is applied to the corresponding channel of `x`.

Simulate using — Specify type of simulation to run
`Interpreted execution` (default) | `Code generation`

- `Interpreted execution` –– Simulate model using the MATLAB interpreter. This option shortens startup time and has a simulation speed comparable to `Code generation`. In this mode, you can debug the source code of the block.
- `Code generation` –– Simulate model using generated C code. The first time you run a simulation, Simulink generates C code for the block. The C code is reused for subsequent simulations, as long as the model does not change. This option requires additional startup time, but the speed of the subsequent simulations is comparable to `Interpreted execution`.

Tunable: No

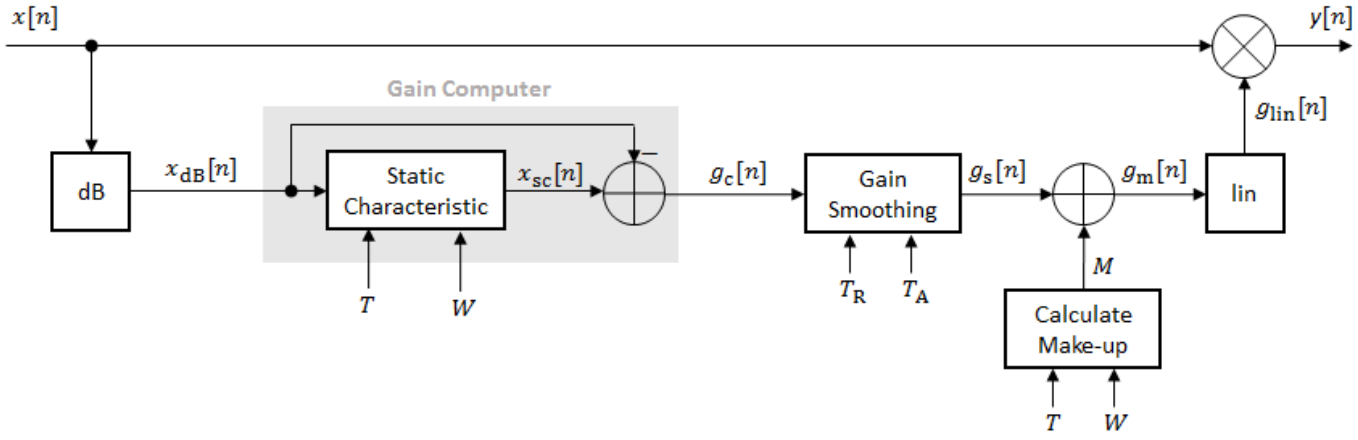
Block Characteristics

Data Types	`double` | `single`
Direct Feedthrough	`no`
Multidimensional Signals	`no`

Variable-Size Signals	yes
Zero-Crossing Detection	no

Algorithms

The Compressor block processes a signal frame by frame and element by element.



1 The N-point signal, $x[n]$, is converted to decibels:

$$x_{\mathrm{dB}}[n] = 20 \times \log_{10}|x[n]|$$

2 $x_{\mathrm{dB}}[n]$ 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:

$$x_{\mathrm{sc}}(x_{\mathrm{dB}}) = \begin{cases} x_{\mathrm{dB}} & x_{\mathrm{dB}} < \left(T - \frac{W}{2}\right) \\ x_{\mathrm{dB}} + \frac{\left(\frac{1}{R} - 1\right)\left(x_{\mathrm{dB}} - T + \frac{W}{2}\right)^2}{2W} & \left(T - \frac{W}{2}\right) \le x_{\mathrm{dB}} \le \left(T + \frac{W}{2}\right) \\ T + \frac{(x_{\mathrm{dB}} - T)}{R} & x_{\mathrm{dB}} > \left(T + \frac{W}{2}\right) \end{cases},$$

where T is the threshold, R is the compression ratio, and W is the knee width.

If you specified a hard knee, the gain computer has the following static characteristic:

$$x_{\mathrm{sc}}(x_{\mathrm{dB}}) = \begin{cases} x_{\mathrm{dB}} & x_{\mathrm{dB}} < T \\ T + \frac{(x_{\mathrm{dB}} - T)}{R} & x_{\mathrm{dB}} \ge T \end{cases}$$

3 The computed gain, $g_c[n]$, is calculated as

$$g_c[n] = x_{\mathrm{sc}}[n] - x_{\mathrm{dB}}[n]\,.$$

4 $g_c[n]$ is smoothed using specified attack and release time parameters:

$$g_s[n] = \begin{cases} \alpha_A g_s[n-1] + (1-\alpha_A) g_c[n], & g_c[n] \le g_s[n-1] \\ \alpha_R g_s[n-1] + (1-\alpha_R) g_c[n], & g_c[n] > g_s[n-1] \end{cases}$$

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

$$y[n] = x[n] \times g_{lin}[n].$$

Version History

Introduced in R2016a

References

[1] Giannoulis, Dimitrios, Michael Massberg, and Joshua D. Reiss. "Digital Dynamic Range Compressor Design –– A Tutorial And Analysis." *Journal of Audio Engineering Society*. Vol. 60, Issue 6, 2012, pp. 399–408.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

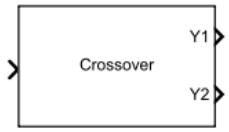
`compressor` | Limiter | Expander | Noise Gate

Topics

“Dynamic Range Control”

Crossover Filter

Audio crossover filter



Libraries:
Audio 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

x — Input signal
matrix | 1-D vector

- Matrix input –– Each column of the input is treated as an independent channel.
- 1-D vector input –– The input is treated as a single channel.

This port is unnamed unless you specify additional input ports.

Data Types: `single` | `double`

F1 — Crossover frequency (Hz)
real scalar in the range `20` to `20000`

Dependencies

To enable this port, select **Specify from input port** for the “Crossover frequency (Hz)” on page 5-0 parameter.

Data Types: `single` | `double`

O1 — Crossover order
integer in the range `0` to `8`

Dependencies

To enable this port, select **Specify from input port** for the “Crossover order” on page 5-0 parameter.

Data Types: `single` | `double`

F2 — Crossover frequency (Hz)
real scalar in the range `20` to `20000`

Dependencies

To enable this port, you need to both:

- Select **Specify from input port** for the “Crossover frequency (Hz)” on page 5-0 parameter.
- Set “Number of crossovers” on page 5-0 to 2, 3 or 4.

Data Types: `single` | `double`

O2 — Crossover order
integer in the range `0` to `8`

Dependencies

To enable this port, you need to both:

- Select **Specify from input port** for the “Crossover order” on page 5-0 parameter.
- Set “Number of crossovers” on page 5-0 to 2, 3 or 4.

Data Types: `single` | `double`

F3 — Crossover frequency (Hz)
real scalar in the range `20` to `20000`

Dependencies

To enable this port, you need to both:

- Select **Specify from input port** for the “Crossover frequency (Hz)” on page 5-0 parameter.
- Set “Number of crossovers” on page 5-0 to 3 or 4.

Data Types: `single` | `double`

O3 — Crossover order
integer in the range `0` to `8`

Dependencies

To enable this port, you need to both:

- Select **Specify from input port** for the “Crossover order” on page 5-0 parameter.
- Set “Number of crossovers” on page 5-0 to 3 or 4.

Data Types: `single` | `double`

F4 — Crossover frequency (Hz)
real scalar in the range `20` to `20000`

Dependencies

To enable this port, you need to both:

- Select **Specify from input port** for the “Crossover frequency (Hz)” on page 5-0 parameter.
- Set “Number of crossovers” on page 5-0 to 4.

Data Types: `single` | `double`

O4 — Crossover order
integer in the range `0` to `8`

Dependencies

To enable this port, you need to both:

- Select **Specify from input port** for the "Crossover order" on page 5-0 parameter.
- Set "Number of crossovers" on page 5-0 to `4`.

Data Types: `single` | `double`

Output

Y1 — Output signal
matrix

Port **Y1** always corresponds to a lowpass filter.

Dependencies

Available if **Number of crossovers** is set to `1`, `2`, `3`, or `4`.

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.

Dependencies

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.

Dependencies

Available if **Number of crossovers** is set to `2`, `3`, or `4`.

Data Types: `single` | `double`

Y4 — Output signal
matrix

Dependencies

Available if **Number of crossovers** is set to `3` or `4`.

Data Types: `single` | `double`

Y5 — Output signal
matrix

Dependencies

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 in Output
1	Two
2	Three
3	Four
4	Five

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) | integer in the range [0, 8]

The crossover filter order relates to the crossover filter slope in dB/octave: $slope = N \times 6$, where N is the crossover order.

Tunable: Yes

View filter response — Open plot of magnitude response of each filter band
button

The plot is updated automatically when parameters of the Crossover Filter block change.

Tunable: Yes

Variable name — Variable name of exported filter
`myFilt` (default) | valid variable name

Name of the variable in the base workspace to contain the filter when it is exported. The name must be a valid MATLAB variable name.

Overwrite variable if it already exists — Overwrite variable if it already exists
`on` (default) | `off`

When you select this parameter, exporting the filter overwrites the variable specified by the **Variable name** parameter if it already exists in the base workspace. If you do not select this parameter and the specified variable already exists in the workspace, exporting the filter creates a new variable with an underscore and a number appended to the variable name. For example, if the variable name is `var` and it already exists, the exported variable will be named `var_1`.

Export filter to workspace — Export filter to workspace
button

Export the filter to the base workspace in the variable specified by the **Variable name** parameter.

Tips

- You cannot export the filter if you have enabled the **Inherit sample rate from input** parameter and the model is not running.
- You cannot export the filter if you are specifying filter characteristics from input ports.

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

Tunable: Yes

Dependencies

To enable this parameter, clear the **Inherit sample rate from input** parameter.

Simulate using — Specify type of simulation to run
`Interpreted execution` (default) | `Code generation`

- `Interpreted execution` – Simulate the model using the MATLAB interpreter. This option reduces startup time and the simulation speed is comparable to `Code generation`. In this mode, you can debug the source code of the block.
- `Code generation` – Simulate the model using generated C code. The first time you run a simulation, Simulink generates C code for the block. The C code is reused for subsequent simulations, as long as the model does not change. This option requires additional startup time, but the speed of the subsequent simulations is comparable to `Interpreted execution`.

Tunable: No

Block Characteristics

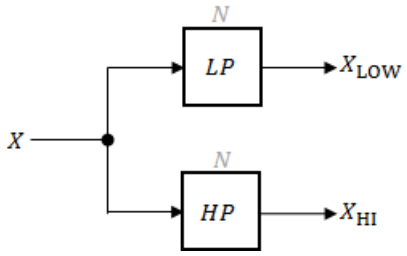
Data Types	`double` | `single`
Direct Feedthrough	`no`
Multidimensional Signals	`no`
Variable-Size Signals	`yes`
Zero-Crossing Detection	`no`

Algorithms

The Crossover Filter block is implemented as a binary tree of crossover pairs with additional phase-compensating sections [1]. Odd-order crossovers are implemented with Butterworth filters, while even-order crossovers are implemented with cascaded Butterworth filters (Linkwitz-Riley filters).

Odd-Order Crossover Pair

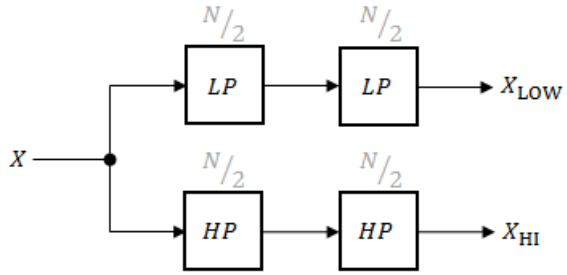
Odd-order two-band (one crossover) filters are implemented as parallel complementary highpass and lowpass filters.



LP and *HP* are Butterworth filters of order *N*, implemented as direct-form II transposed second-order sections. The shared cutoff frequency used in their design corresponds to the crossover of the resulting bands.

Even-Order Crossover Pair

Even-order two-band (one crossover) filters are implemented as parallel complementary highpass and lowpass filters.

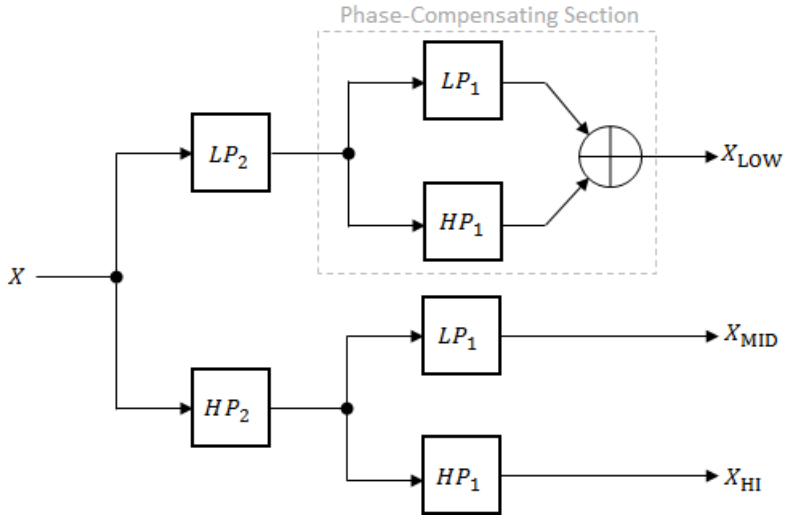


LP and *HP* are Butterworth filters of order *N/2*, where *N* is the order of the overall filter. The filters are implemented as direct-form II transposed second-order sections.

For overall filters of orders 2 and 6, X_{HI} is multiplied by –1 internally so that the branches of your crossover pair are in-phase.

Even-Order Three-Band Filter

Even-order three-band (two crossovers) filters are implemented as parallel complementary highpass and lowpass filters organized in a tree structure.



The phase-compensating section is equivalent to an allpass filter.

The design of four-band and five-band filters (three and four crossovers) are extensions of the pattern developed for even-order and odd-order crossovers and the tree structure specified for three-band (two crossover) filters.

Version History

Introduced in R2016a

References

[1] D'Appolito, Joseph A. "Active Realization of Multiway All-Pass Crossover Systems." *Journal of Audio Engineering Society*. Vol. 35, Issue 4, 1987, pp. 239–245.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

`crossoverFilter`

Topics

"Multiband Dynamic Range Compression"

Design Auditory Filter Bank

Design frequency-domain auditory filter bank

Libraries:
Audio Toolbox / Features

Description

The Design Auditory Filter Bank block outputs a frequency-domain auditory filter bank. You can use an auditory filter bank to decompose an audio signal into separate frequency bands for feature extraction.

Ports

Output

fb — Filter bank
column vector | matrix

Auditory filter bank, returned as an *M*-by-*N* matrix, where:

- *M* is the number of bands, which is determined by the **Auto-determine number of bands** and **Number of bands** parameters.
- *N* is the number of points in the spectrum. If you select **Design one-sided filter bank**, then *N* is equal to `ceil(NFFT/2)`, where `NFFT` is the **FFT length**. If you do not select **Design one-sided filter bank**, then *N* is equal to the **FFT length**.

This port is unnamed until you select the **Output center frequencies** parameter.

Data Types: `single` | `double`

fvec — Center frequencies
row vector

Center frequencies of the bandpass filters in Hz, returned as a row vector with number of elements equal to the number of bands.

Dependencies

To enable this port, select the **Output center frequencies** parameter.

Data Types: `single` | `double`

Parameters

Frequency scale — Frequency scale of filter bank

`mel` (default) | `bark` | `erb`

Frequency scale used to design the auditory filter bank, specified as `mel`, `bark`, or `erb`.

- `mel` –– Design the filter bank as half-overlapped triangles equally spaced on the mel scale.
- `bark` –– Design the filter bank as half-overlapped triangles equally spaced on the Bark scale.
- `erb` –– Design the filter bank as gammatone filters whose center frequencies are equally spaced on the ERB scale.

FFT length — Number of DFT points

`1024` (default) | positive integer

Number of points used to calculate the DFT, specified as a positive integer.

Design one-sided filter bank — Design one-sided or two-sided filter bank

`on` (default) | `off`

When you select this parameter, the block designs a one-sided filter bank. Otherwise, the filter bank is two sided.

Auto-determine number of bands — Automatically determine number of bandpass filters

`on` (default) | `off`

When you select this parameter, the block automatically determines the number of bandpass filters based on the **Frequency scale** parameter.

- If you set **Frequency scale** to `mel` or `bark`, then the number of bands is `32`.
- If you set **Frequency scale** to `erb`, then the number of bands is equal to `ceil(hz2erb(fr(2))-hz2erb(fr(1)))`, where `fr` is specified using **Frequency range (Hz)**.

Number of bands — Number of bandpass filters

`32` (default) | positive integer

Number of bandpass filters, specified as a positive integer.

Dependencies

To enable this parameter, clear the **Auto-determine number of bands** parameter.

Auto-determine frequency range — Automatically determine frequency range

`on` (default) | `off`

When you select this parameter, the block sets the **Frequency range** to `[0,fs/2]`, where `fs` is specified using **Sample rate (Hz)**.

Frequency range (Hz) — Frequency range over which to design auditory filter bank

`[0,22050]` (default) | two-element row vector

Frequency range in Hz over which to design the auditory filter bank, specified as a two-element row vector.

Dependencies

To enable this parameter, clear the **Auto-determine frequency range** parameter.

Filter bank design domain — Domain to design filter bank

`linear` (default) | `warped`

Domain in which the block designs the filter bank, specified as `linear` or `warped`. Set the filter bank design domain to `linear` to design the bandpass filters in the linear (Hz) domain. Set the filter bank design domain to `warped` to design the bandpass filters in the warped (mel or Bark) domain.

Dependencies

To enable this parameter, set **Frequency scale** to `mel` or `bark`.

Filter bank normalization — Normalization technique for filter bank

`bandwidth` (default) | `area` | `none`

Normalization technique used for the filter bank weights, specified as `bandwidth`, `area`, or `none`.

- `bandwidth` –– Normalize the weights of each bandpass filter by the corresponding bandwidth of the filter.
- `area` –– Normalize the weights of each bandpass filter by the corresponding area of the bandpass filter.
- `none` –– The block does not normalize the weights of the filters.

Output data type — Data type of output

`double` (default) | `single`

Data type of output, specified as `double` or `single`.

Sample rate (Hz) — Sample rate

`44.1e3` (default) | positive scalar

Sample rate in Hz of the filter design, specified as a positive scalar.

Visualize filter bank — Open plot to visualize filter bank

button

Open plot to visualize the filters in the frequency domain.

Output center frequencies — Specify additional output port for center frequencies

`off` (default) | `on`

When you select this parameter, the block displays an additional output port, **fvec**. This port outputs the center frequencies of the bandpass filters.

Block Characteristics

Data Types	`double` | `single`
Direct Feedthrough	`no`
Multidimensional Signals	`no`
Variable-Size Signals	`no`
Zero-Crossing Detection	`no`

Version History

Introduced in R2022a

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

Blocks

Auditory Spectrogram | Design Mel Filter Bank | Mel Spectrogram

Functions

`designAuditoryFilterBank` | `melSpectrogram`

Objects

`audioFeatureExtractor`

Design Mel Filter Bank

Design frequency-domain mel filter bank

Libraries:
Audio Toolbox / Features

Description

The Design Mel Filter Bank block outputs a frequency-domain auditory filter bank using the mel frequency scale. You can use a mel filter bank to decompose an audio signal into separate frequency bands in the mel frequency scale, which mimics the nonlinear human perception of sound.

Ports

Output

fb — Filter bank
column vector | matrix

Mel filter bank, returned as an *M*-by-*N* matrix, where:

- *M* is the number of bands, which is determined by the **Auto-determine number of bands** and **Number of bands** parameters.
- *N* is the number of points in the spectrum. If you select **Design one-sided filter bank**, then *N* is equal to `ceil(NFFT/2)`, where `NFFT` is the **FFT length**. If you do not select **Design one-sided filter bank**, then *N* is equal to the **FFT length**.

This port is unnamed until you select the **Output center frequencies** parameter.

Data Types: `single` | `double`

fvec — Center frequencies
row vector

Center frequencies of the bandpass filters in Hz, returned as a row vector with number of elements equal to the number of bands.

Dependencies

To enable this port, select the **Output center frequencies** parameter.

Data Types: `single` | `double`

Parameters

FFT length — Number of DFT points

`1024` (default) | positive integer

Number of points used to calculate the DFT, specified as a positive integer.

Design one-sided filter bank — Design one-sided or two-sided filter bank

`on` (default) | `off`

When you select this parameter, the block designs a one-sided filter bank. Otherwise, the filter bank is two sided.

Number of bands — Number of bandpass filters

`32` (default) | positive integer

Number of bandpass filters, specified as a positive integer.

Auto-determine frequency range — Automatically determine frequency range

`on` (default) | `off`

When you select this parameter, the block sets the **Frequency range** to `[0,fs/2]`, where `fs` is specified using **Sample rate (Hz)**.

Frequency range (Hz) — Frequency range over which to design filter bank

`[0,22050]` (default) | two-element row vector

Frequency range in Hz over which to design the filter bank, specified as a two-element row vector.

Dependencies

To enable this parameter, clear the **Auto-determine frequency range** parameter.

Filter bank design domain — Domain to design filter bank

`linear` (default) | `warped`

Domain in which the block designs the filter bank, specified as `linear` or `warped`. Set the filter bank design domain to `linear` to design the bandpass filters in the linear (Hz) domain. Set the filter bank design domain to `warped` to design the bandpass filters in the warped (mel) domain.

Filter bank normalization — Normalization technique for filter bank

`bandwidth` (default) | `area` | `none`

Normalization technique used for the filter bank weights, specified as `bandwidth`, `area`, or `none`.

- `bandwidth` –– Normalize the weights of each bandpass filter by the corresponding bandwidth of the filter.
- `area` –– Normalize the weights of each bandpass filter by the corresponding area of the bandpass filter.
- `none` –– The block does not normalize the weights of the filters.

Output data type — Data type of output

`double` (default) | `single`

Data type of output, specified as `double` or `single`.

Sample rate (Hz) — Sample rate

`44.1e3` (default) | positive scalar

Sample rate in Hz of the filter design, specified as a positive scalar.

Visualize filter bank — Open plot to visualize filter bank

button

Open plot to visualize the filters in the frequency domain.

Output center frequencies — Specify additional output port for center frequencies

`off` (default) | `on`

When you select this parameter, the block displays an additional output port, **fvec**. This port outputs the center frequencies of the bandpass filters.

Block Characteristics

Data Types	`double` | `single`
Direct Feedthrough	no
Multidimensional Signals	no
Variable-Size Signals	no
Zero-Crossing Detection	no

Version History

Introduced in R2022a

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

Blocks

Auditory Spectrogram | Design Auditory Filter Bank | Mel Spectrogram

Functions

`designAuditoryFilterBank` | `melSpectrogram`

Objects

`audioFeatureExtractor`

Expander

Dynamic range expander

Libraries:
Audio Toolbox / Dynamic Range Control

Description

The Expander block performs dynamic range expansion independently across each input channel. Dynamic range expansion attenuates the volume of quiet sounds below a given threshold. The block uses specified attack, release, and hold times to achieve a smooth applied gain curve.

Ports

Input

x — Input signal
matrix | 1-D vector

- Matrix input –– Each column of the input is treated as an independent channel.
- 1-D vector input –– The input is treated as a single channel.

This port is unnamed unless you specify additional input ports.

Data Types: `single` | `double`

R — Ratio
scalar

Dependencies

To enable this port, select **Specify from input port** for the “Ratio” on page 5-0 parameter.

Data Types: `single` | `double`

T — Threshold (dB)
scalar

Dependencies

To enable this port, select **Specify from input port** for the “Threshold (dB)” on page 5-0 parameter.

Data Types: `single` | `double`

K — Knee width (dB)
scalar

Dependencies

To enable this port, select **Specify from input port** for the “Knee width (dB)” on page 5-0 parameter.

Data Types: `single` | `double`

AT — Attack time (s)
scalar

Dependencies

To enable this port, select **Specify from input port** for the “Attack time (s)” on page 5-0 parameter.

Data Types: `single` | `double`

RT — Release time (s)
scalar

Dependencies

To enable this port, select **Specify from input port** for the “Release time (s)” on page 5-0 parameter.

Data Types: `single` | `double`

HT — Hold time (s)
scalar

Dependencies

To enable this port, select **Specify from input port** for the “Hold time (s)” on page 5-0 parameter.

Data Types: `single` | `double`

Output

Y — Output signal
matrix

The Expander block outputs a signal with the same data type as the input signal. The size of the output depends on the size of the input:

- Matrix input –– The block outputs a matrix the same size and data type as the input signal.
- 1-D vector input –– The block outputs an *N*-by-1 matrix (column vector), where *N* is the number of elements in the 1-D vector.

This port is unnamed until you select the **Output gain (dB)** parameter.

Data Types: `single` | `double`

G — Gain applied to each input sample
matrix

Dependencies

To enable this port, select the **Output gain (dB)** parameter.

Data Types: `single` | `double`

Parameters

If a parameter is listed as tunable, then you can change its value during simulation.

Main Tab

Ratio — Expansion ratio
5 (default) | scalar in the range 1 to 50 inclusive

Expansion ratio is the input/output ratio for signals that undershoot the operation threshold.

Assuming a hard knee characteristic and a steady-state input such that $x[n]$ dB < **Threshold (dB)**, the expansion ratio is defined as $R = \frac{(y[n] - T)}{(x[n] - T)}$, where

- R is the expansion ratio.
- $y[n]$ is the output signal in dB.
- $x[n]$ is the input signal in dB.
- T is the threshold in dB.

To specify **Ratio** from an input port, select **Specify from input port** for the parameter.

Tunable: Yes

Threshold (dB) — Operation threshold
`−10` (default) | scalar in the range –140 to 0 inclusive

Operation threshold is the level below which gain is applied to the input signal.

To specify **Threshold (dB)** from an input port, select **Specify from input port** for the parameter.

Tunable: Yes

Knee width (dB) — Transition area in the compression characteristic
`0` (default) | scalar in the range 0 to 20

For soft knee characteristics, the transition area is defined by the relation

$$y = x + \frac{(1 - R) \times \left(x - T - \frac{W}{2}\right)^2}{(2 \times W)}$$

for the range $(2 \times |x - T|) \leq W$, where

- y is the output level in dB.
- x is the input level in dB.
- R is the expansion ratio.
- T is the threshold in dB.
- W is the knee width in dB.

To specify **Knee width (dB)** from an input port, select **Specify from input port** for the parameter.

Tunable: Yes

View static characteristic — Open static characteristic plot of dynamic range expander
button

The plot is updated automatically when parameters of the Expander block change.

Tunable: Yes

Attack time (s) — Time for applied gain to ramp up
`0.05` (default) | scalar in the range 0 to 4 inclusive

Attack time is the time the expander gain takes to rise from 10% to 90% of its final value when the input goes below the threshold. The **Attack time (s)** parameter smooths the applied gain curve.

To specify **Attack time (s)** from an input port, select **Specify from input port** for the parameter.

Tunable: Yes

Release time (s) — Time for applied gain to ramp down
`0.2` (default) | scalar in the range 0 to 4 inclusive

Release time is the time the expander gain takes to drop from 90% to 10% of its final value when the input goes above the threshold. The **Release time (s)** parameter smooths the applied gain curve.

To specify **Release time (s)** from an input port, select **Specify from input port** for the parameter.

Tunable: Yes

Hold time (s) — Time during which applied gain holds steady
`0.05` (default) | scalar in the range 0 to 4 inclusive

Hold time is the period for which the (negative) gain is held before starting to decrease towards its steady state value when the input level drops below the threshold.

To specify **Hold time (s)** from an input port, select **Specify from input port** for the parameter.

Tunable: Yes

Inherit sample rate from input — Specify source of input sample rate
`on` (default) | `off`

When you select this parameter, the block inherits its sample rate from the input signal. When you clear this parameter, specify the sample rate in the **Input sample rate (Hz)** parameter.

Tunable: No

Input sample rate (Hz) — Sample rate of input
`44100` (default) | positive scalar

Tunable: Yes

Dependencies

To enable this parameter, clear the **Inherit sample rate from input** parameter.

Advanced Tab

Output gain (dB) — Gain applied on each input sample
`off` (default) | `on`

When you select this parameter, an additional output port, **G**, is added to the block. The **G** port outputs the gain applied on each input channel in dB.

Tunable: No

Sidechain — Enable sidechain input
`off` (default) | `on`

When you select this parameter, an additional input port **SC** is added to the block. The **SC** port enables dynamic range expansion of the input signal **x** using a separate sidechain signal.

The datatype and (frame) length input to the **SC** port must be the same as the input to the **x** port.

The number of channels of the sidechain input must be equal to the number of channels of `x` or be equal to one.

- `Sidechain channel count is equal to one` –– The computed gain, `G`, based on this channel is applied to all channels of `x`.
- `Sidechain channel count is equal to channel count of x` –– The computed gain, `G`, for each sidechain channel is applied to the corresponding channel of `x`.

Tunable: No

Simulate using — Specify type of simulation to run
`Interpreted execution` (default) | `Code generation`

- `Interpreted execution` –– Simulate model using the MATLAB interpreter. This option shortens startup time and has a simulation speed comparable to `Code generation`. In this mode, you can debug the source code of the block.
- `Code generation` –– Simulate model using generated C code. The first time you run a simulation, Simulink generates C code for the block. The C code is reused for subsequent simulations, as long as the model does not change. This option requires additional startup time, but the speed of the subsequent simulations is comparable to `Interpreted execution`.

Tunable: No

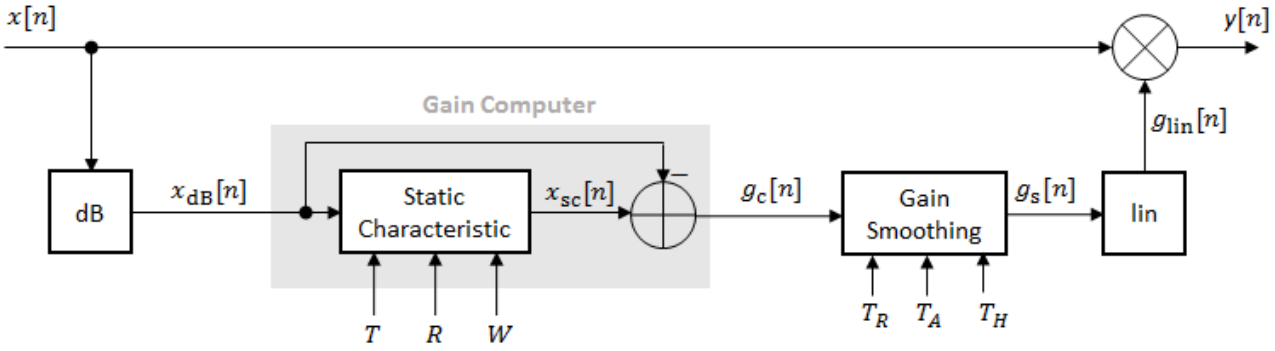
Block Characteristics

Data Types	`double` | `single`
Direct Feedthrough	`no`

Multidimensional Signals	no
Variable-Size Signals	yes
Zero-Crossing Detection	no

Algorithms

The Expander block processes a signal frame by frame and element by element.



1 The N-point signal, $x[n]$, is converted to decibels:

$$x_{\mathrm{dB}}[n] = 20 \times \log_{10}|x[n]|$$

2 $x_{\mathrm{dB}}[n]$ 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:

$$x_{\mathrm{sc}}(x_{\mathrm{dB}}) = \begin{cases} T + (x_{\mathrm{dB}} - T) \times R & x_{\mathrm{dB}} < \left(T - \frac{W}{2}\right) \\ x_{\mathrm{dB}} + \frac{(1-R)\left(x_{\mathrm{dB}} - T - \frac{W}{2}\right)^2}{2W} & \left(T - \frac{W}{2}\right) \le x_{\mathrm{dB}} \le \left(T + \frac{W}{2}\right) \\ x_{\mathrm{dB}} & x_{\mathrm{dB}} > \left(T + \frac{W}{2}\right) \end{cases},$$

where T is the threshold, R is the expansion ratio, and W is the knee width.

If you specified a hard knee, the gain computer has the following static characteristic:

$$x_{\mathrm{sc}}(x_{\mathrm{dB}}) = \begin{cases} T + (x_{\mathrm{dB}} - T) \times R & x_{\mathrm{dB}} < T \\ x_{\mathrm{dB}} & x_{\mathrm{dB}} \ge T \end{cases}$$

3 The computed gain, $g_{\mathrm{c}}[n]$, is calculated as

$$g_{\mathrm{c}}[n] = x_{\mathrm{sc}}[n] - x_{\mathrm{dB}}[n]\,.$$

4 $g_{\mathrm{c}}[n]$ is smoothed using specified attack, release, and hold time parameters:

$$g_s[n] = \begin{cases} \alpha_A g_s[n-1] + (1-\alpha_A) g_c[n] & (C_A > T_H) \ \& \ (g_c[n] \le g_s[n-1]) \\ g_s[n-1] & C_A \le T_H \\ \alpha_R g_s[n-1] + (1-\alpha_R) g_c[n] & g_c[n] > g_s[n-1] \end{cases}$$

C_A is the hold counter for attack. The limit, T_H, is determined by the **Hold time (s)** parameter.

The attack time coefficient, α_A, is calculated as

$$\alpha_A = \exp\left(\frac{-\log(9)}{Fs \times T_A}\right).$$

The release time coefficient, α_R, is calculated as

$$\alpha_R = \exp\left(\frac{-\log(9)}{Fs \times T_R}\right).$$

T_A is the attack time period, specified by the **Attack time (s)** parameter. T_R is the release time period, specified by the **Release time (s)** parameter. Fs is the input sampling rate, specified by the **Inherit sample rate from input** or **Input sample rate (Hz)** parameter.

5 The smoothed gain in dB, $g_s[n]$, is translated to a linear domain:

$$g_{lin}[n] = 10^{\left(\frac{g_s[n]}{20}\right)}.$$

6 The output of the dynamic range expander is given as

$$y[n] = x[n] \times g_{lin}[n].$$

Version History

Introduced in R2016a

References

[1] Giannoulis, Dimitrios, Michael Massberg, and Joshua D. Reiss. "Digital Dynamic Range Compressor Design –– A Tutorial And Analysis." *Journal of Audio Engineering Society*. Vol. 60, Issue 6, 2012, pp. 399–408.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

`expander` | Limiter | Compressor | Noise Gate

Topics

“Dynamic Range Control”

Graphic EQ

Standards-based graphic equalizer

Libraries:
Audio Toolbox / Filters

Description

The Graphic EQ block implements a graphic equalizer that can tune the gain on individual octave or fractional octave bands. The block filters the data independently across each input channel over time using the filter specifications. Center frequencies for bands in the graphic equalizer are based on the ANSI S1.11-2004 standard.

Ports

Input

Port_1 — Input signal
matrix | 1-D vector

- Matrix input –– Each column of the input is treated as an independent channel.
- 1-D vector input –– The input is treated as a signal channel.

Data Types: `single` | `double`

Output

Port_1 — Output signal
matrix

The Graphic EQ block outputs a signal with the same data type as the input signal. The size of the output depends on the size of the input:

- Matrix input –– The block outputs a matrix the same size and data type as the input signal.
- 1-D vector input –– The block outputs an *N*-by-1 matrix (column vector), where *N* is the number of elements in the 1-D vector input.

Data Types: `single` | `double`

Parameters

If a parameter is listed as tunable, then you can change its value during simulation.

EQ Order — Order of individual equalizer bands
2 (default) | positive even integer

Specify the order of individual equalizer bands as a positive even integer. All equalizer bands have the same order.

Tunable: Yes

Bandwidth — Filter bandwidth (octaves)
`1 octave` (default) | `2/3 octave` | `1/3 octave`

Specify the filter bandwidth as `1 octave`, `2/3 octave`, or `1/3 octave`.

The ANSI S1.11-2004 standard defines the center and edge frequencies of your equalizer. The ISO 266:1997(E) standard specifies corresponding preferred frequencies for labeling purposes.

1-Octave Bandwidth

Center frequencies	`32 63 126 251 501 1000 1995 3981 7943 15849`
Edge frequencies	`22 45 89 178 355 708 1413 2818 5623 1122 22387`
Preferred frequencies	`31.5 63 125 250 500 1000 2000 4000 8000 16000`

2/3-Octave Bandwidth

Center frequencies	`25 40 63 100 158 251 398 631 1000 1585 2512 3981 6310 10000 15849`
Edge frequencies	`20 32 50 79 126 200 316 501 794 1259 1995 3162 5012 7943 12589 19953`
Preferred frequencies	`25 40 63 100 160 250 400 630 1000 1600 2500 4000 6300 10000 16000`

1/3-Octave Bandwidth

Center frequencies	`25 32 40 50 63 79 100 126 158 200 251 316 398 501 631 794 1000 1259 1585 1995 2512 3162 3981 5012 6310 7943 10000 12589 15849 19953`
Edge frequencies	`22 28 35 45 56 71 89 112 141 178 224 282 355 447 562 708 891 1122 1413 1778 2239 2818 3548 4467 5623 7079 8913 11220 14125 17783 22387`
Preferred frequencies	`25 31.5 40 50 63 80 100 125 160 200 250 315 400 500 630 800 1000 1250 1600 2000 2500 3150 4000 5000 6300 8000 10000 12500 16000 20000`

Tunable: Yes

Structure — Type of implementation
`Cascade` (default) | `Parallel`

Specify the type of implementation as `Cascade` or `Parallel`. See “Algorithms” on page 5-68 and “Graphic Equalization” for information about these implementation structures.

Tunable: No

Gains — Gain of each octave or fractional octave band (dB)
`0` | scalar

Specify the gain of each octave or fractional octave band in dB. The number and position of filters in the graphic equalizer depends on the **Bandwidth** on page 5-0 parameter.

Tunable: Yes

Variable name — Variable name of exported filter
`myFilt` (default) | valid variable name

Name of the variable in the base workspace to contain the filter when it is exported. The name must be a valid MATLAB variable name.

Overwrite variable if it already exists — Overwrite variable if it already exists
`on` (default) | `off`

When you select this parameter, exporting the filter overwrites the variable specified by the **Variable name** parameter if it already exists in the base workspace. If you do not select this parameter and the specified variable already exists in the workspace, exporting the filter creates a new variable with an underscore and a number appended to the variable name. For example, if the variable name is `var` and it already exists, the exported variable will be named `var_1`.

Export filter to workspace — Export filter to workspace
button

Export the filter to the base workspace in the variable specified by the **Variable name** parameter.

Tips

You cannot export the filter if you have enabled the **Inherit sample rate from input** parameter and the model is not running.

Inherit sample rate from input — Specify source of input sample rate
off (default) | on

When you select this parameter, the block inherits its sample rate from the input signal. When you clear this parameter, specify the sample rate in **Input sample rate (Hz)** on page 5-0 .

Tunable: No

Input sample rate (Hz) — Sample rate of input
`44100` (default) | scalar

Tunable: Yes

Dependencies

To enable this parameter, clear the **Inherit sample rate from input** on page 5-0 parameter.

Simulate using — Specify type of simulation to run
`Code generation` (default) | `Interpreted execution`

- `Interpreted execution` –- Simulate model using the MATLAB interpreter. This option shortens startup time and has simulation speed comparable to `Code generation`. In this mode, you can debug the source code of the block.
- `Code generation` –- Simulate model using generated C code. The first time you run a simulation, Simulink generates C code for the block. The C code is reused for subsequent simulations, as long as the model does not change. This option requires additional startup time but the speed of the subsequent simulations is faster than `Interpreted execution`.

Tunable: No

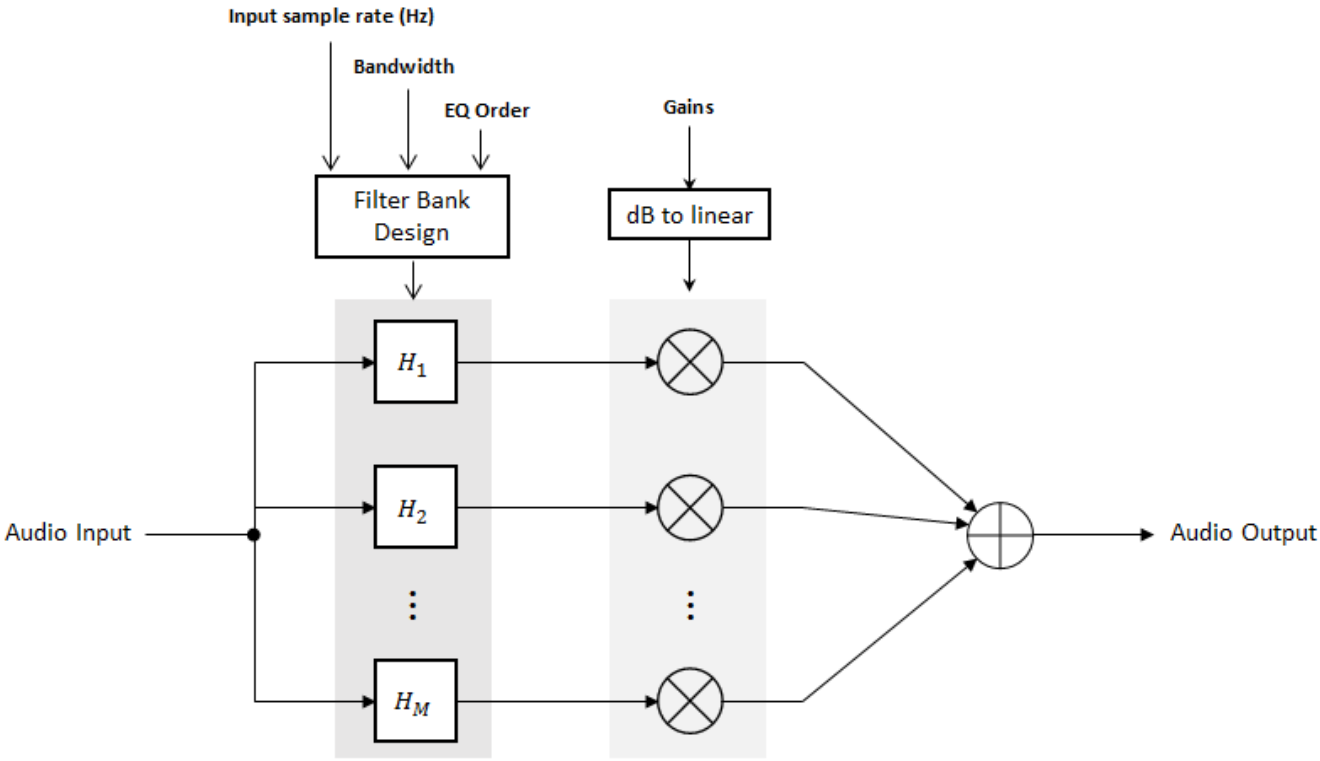
Block Characteristics

Data Types	`double` | `single`
Direct Feedthrough	`no`
Multidimensional Signals	`no`
Variable-Size Signals	`yes`
Zero-Crossing Detection	`no`

Algorithms

The implementation of your graphic equalizer depends on the **Structure** on page 5-0 parameter. See “Graphic Equalization” for a discussion of the pros and cons of the parallel and cascade implementations. Refer to the following sections to understand how these algorithms are implemented in Audio Toolbox.

Parallel Structure



Filter Bank Design

The parallel implementation designs the individual equalizers using the `octaveFilter design method` and spaces them on the spectrum according to the ANSI S1.11-2004 standard.

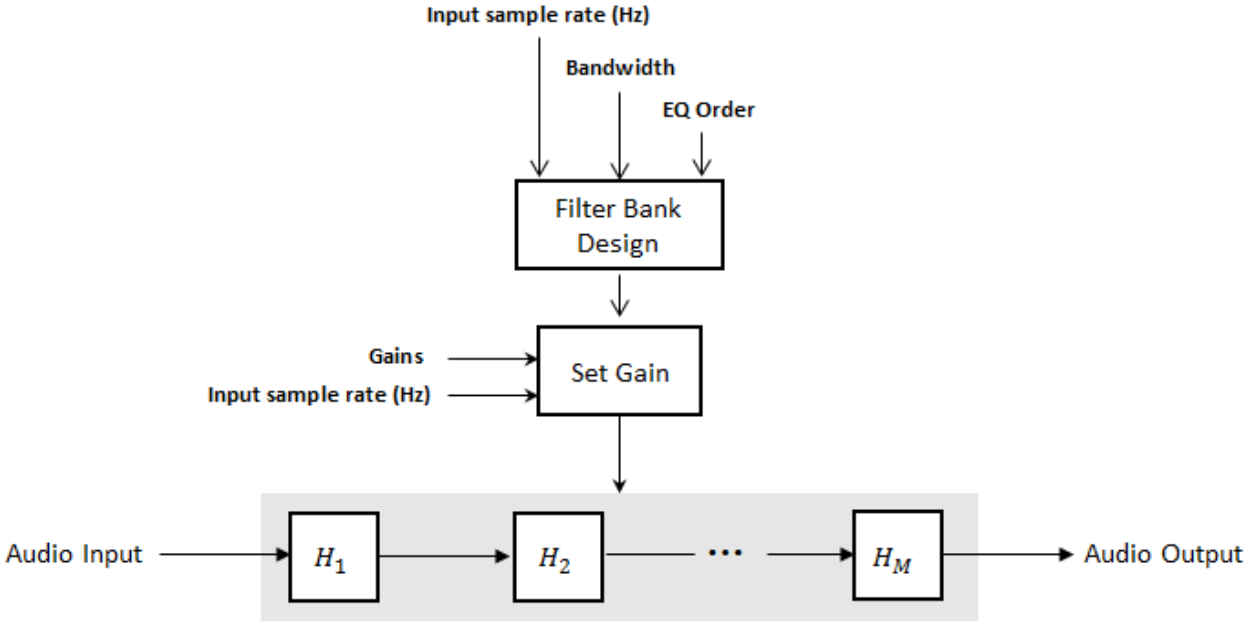
If you set the **Input sample rate (Hz)** parameter so that the Nyquist frequency (**Input sample rate (Hz)**/2) is less than the final bandpass edge defined by the ANSI S1.11-2004 standard, then:

- The final bandpass filter is the one whose upper bandpass edge is less than the Nyquist frequency.
- The final filter is implemented as a highpass filter designed by the `designParamEQ` function.

Real-Time Computation

1. The input signal is fed into a filterbank of *M* filters, where *M* depends on the specified **Bandwidth** and **Input sample rate (Hz)** parameters.
2. Each branch of the filterbank is multiplied by the linear form of the corresponding element of the **Gains** parameter.
3. The branches are summed and the output signal is returned.

Cascade Structure



Filter Bank Design

The cascade implementation designs the graphic equalizer filter bank using the `multibandParametricEQ` System object.

Gain Setting

If the **EQ Order** on page 5-0 parameter is set to 2, then a gain correction is calculated according to [1]. The gain correction is independent of the requested gains. The gain correction is recomputed during the real-time processing only if the **Input sample rate (Hz)** parameter is modified.

If the **EQ Order** parameter is not set to 2, no gain correction is applied and the requested gains are passed on to the `multibandParametricEQ` object.

Real-Time Computation

The input signal is fed into a cascade of *M* biquad filters, where *M* depends on the specified **Bandwidth** and **Input sample rate (Hz)** parameters.

Version History

Introduced in R2017b

References

[1] Oliver, Richard J., and Jean-Marc Jot. "Efficient Multi-Band Digital Audio Graphic Equalizer with Accurate Frequency Response Control." Presented at the 139th Convention of the AES, New York, October 2015.

[2] Acoustical Society of America. *American National Standard Specification for Octave-Band and Fractional-Octave-Band Analog and Digital Filters*. ANSI S1.11-2004. Melville, NY: Acoustical Society of America, 2009.

[3] International Organization for Standardization. *Acoustics –– Preferred frequencies*. ISO 266:1997(E). Second Edition. 1997.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

Single-Band Parametric EQ | Multiband Parametric EQ | `graphicEQ` | `multibandParametricEQ` | `designVarSlopeFilter` | `designParamEQ` | `designShelvingEQ`

Topics

“Graphic Equalization”
“Equalization”

Limiter

Dynamic range limiter

Libraries:
Audio Toolbox / Dynamic Range Control

Description

The Limiter block performs dynamic range limiting independently across each input channel. Dynamic range limiting suppresses the volume of loud sounds that cross a given threshold. The block uses specified attack and release times to achieve a smooth applied gain curve.

Ports

Input

x — Input signal
1-D vector | matrix

- Matrix input –– Each column of the input is treated as an independent channel.
- 1-D vector input –– The input is treated as a single channel.

This port is unnamed unless you specify additional input ports.

Data Types: `single` | `double`

T — Threshold (dB)
scalar

Dependencies

To enable this port, select **Specify from input port** for the “Threshold (dB)” on page 5-0 parameter.

Data Types: `single` | `double`

K — Knee width (dB)
scalar

Dependencies

To enable this port, select **Specify from input port** for the “Knee width (dB)” on page 5-0 parameter.

Data Types: `single` | `double`

AT — Attack time (s)
scalar

Dependencies

To enable this port, select **Specify from input port** for the “Attack time (s)” on page 5-0 parameter.

Data Types: `single` | `double`

RT — Release time (s)
scalar

Dependencies

To enable this port, select **Specify from input port** for the “Release time (s)” on page 5-0 parameter.

Data Types: `single` | `double`

Output

Y — Output signal
matrix

The Limiter block outputs a signal with the same data type as the input signal. The size of the output depends on the size of the input:

- Matrix input –– The block outputs a matrix the same size and data type as the input signal.
- 1-D vector input –– The block outputs an *N*-by-1 matrix (column vector), where *N* is the number of elements in the 1-D vector.

This port is unnamed until you select the **Output gain (dB)** parameter.

Data Types: `single` | `double`

G — Gain applied to each input sample
matrix

Dependencies

To enable this port, select the **Output gain (dB)** parameter.

Data Types: `single` | `double`

Parameters

If a parameter is listed as tunable, then you can change its value during simulation.

Main Tab

Threshold (dB) — Operation threshold
`–10` (default) | scalar in the range –50 to 0 inclusive

Operation threshold is the level above which gain is applied to the input signal.

To specify **Threshold (dB)** from an input port, select **Specify from input port** for the parameter.

Tunable: Yes

Knee width (dB) — Transition area in the limiter characteristic
`0` (default) | scalar in the range 0 to 20 inclusive

For soft knee characteristics, the transition area is defined by the relation

$$y = x - \frac{\left(x - T + \frac{W}{2}\right)^2}{(2 \times W)}$$

for the range $(2 \times |x - T|) \le W$, where

- y is the output level in dB.
- x is the input level in dB.
- T is the threshold in dB.
- W is the knee width in dB.

To specify **Knee width (dB)** from an input port, select **Specify from input port** for the parameter.

Tunable: Yes

View static characteristic — Open static characteristic plot of dynamic range limiter
button

The plot is updated automatically when parameters of the Limiter block change.

Tunable: Yes

Attack time (s) — Time for applied gain to ramp up
`0` (default) | scalar in the range 0 to 4 inclusive

Attack time is the time the limiter gain takes to rise from 10% to 90% of its final value when the input goes above the threshold. The **Attack time (s)** parameter smooths the applied gain curve.

To specify **Attack time (s)** from an input port, select **Specify from input port** for the parameter.

Tunable: Yes

Release time (s) — Time for applied gain to ramp down
`0.2` (default) | scalar in the range 0 to 4 inclusive

Release time is the time the limiter gain takes to drop from 90% to 10% of its final value when the input goes below the threshold. The **Release time (s)** parameter smooths the applied gain curve.

To specify **Release time (s)** from an input port, select **Specify from input port** for the parameter.

Tunable: Yes

Make-up gain mode — Make-up gain mode
`Property` (default) | `Auto`

- `Property` –– Make-up gain is set to the value specified by the **Make-up gain (dB)** parameter.
- `Auto` –– Make-up gain is applied at the output of the Limiter block such that a steady-state 0 dB input has a 0 dB output.

Tunable: No

Make-up gain (dB) — Applied make-up gain
`0` (default) | scalar in the range –10 to 24 inclusive

Make-up gain compensates for gain lost during limiting. It is applied at the output of the Limiter block.

Tunable: Yes

Dependencies

To enable this parameter, set the **Make-up gain mode** parameter to `Property`.

Inherit sample rate from input — Specify source of input sample rate
`on` (default) | `off`

When you select this parameter, the block inherits its sample rate from the input signal. When you clear this parameter, specify the sample rate in the **Input sample rate (Hz)** parameter.

Tunable: No

Input sample rate (Hz) — Specify input sample rate
`44100` (default) | positive scalar

Tunable: Yes

Dependencies

To enable this parameter, clear the **Inherit sample rate from input** parameter.

Advanced Tab

Output gain (dB) — Gain applied on each input sample
`off` (default) | `on`

When you select this parameter, an additional output port, **G**, is added to the block. The **G** port outputs the gain applied on each input channel in dB.

Tunable: No

Sidechain — Enable sidechain input
`off` (default) | `on`

When you select this parameter, an additional input port **SC** is added to the block. The **SC** port enables dynamic range limiting of the input signal **x** using a separate sidechain signal.

The datatype and (frame) length input to the **SC** port must be the same as the input to the **x** port.

The number of channels of the sidechain input must be equal to the number of channels of `x` or be equal to one.

- `Sidechain channel count is equal to one` –– The computed gain, `G`, based on this channel is applied to all channels of `x`.
- `Sidechain channel count is equal to channel count of x` –– The computed gain, `G`, for each sidechain channel is applied to the corresponding channel of `x`.

Tunable: No

Simulate using — Specify type of simulation to run
`Interpreted execution` (default) | `Code generation`

- `Interpreted execution` –– Simulate model using the MATLAB interpreter. This option shortens startup time and has a simulation speed comparable to `Code generation`. In this mode, you can debug the source code of the block.
- `Code generation` –– Simulate model using generated C code. The first time you run a simulation, Simulink generates C code for the block. The C code is reused for subsequent simulations, as long as the model does not change. This option requires additional startup time, but the speed of the subsequent simulations is comparable to `Interpreted execution`.

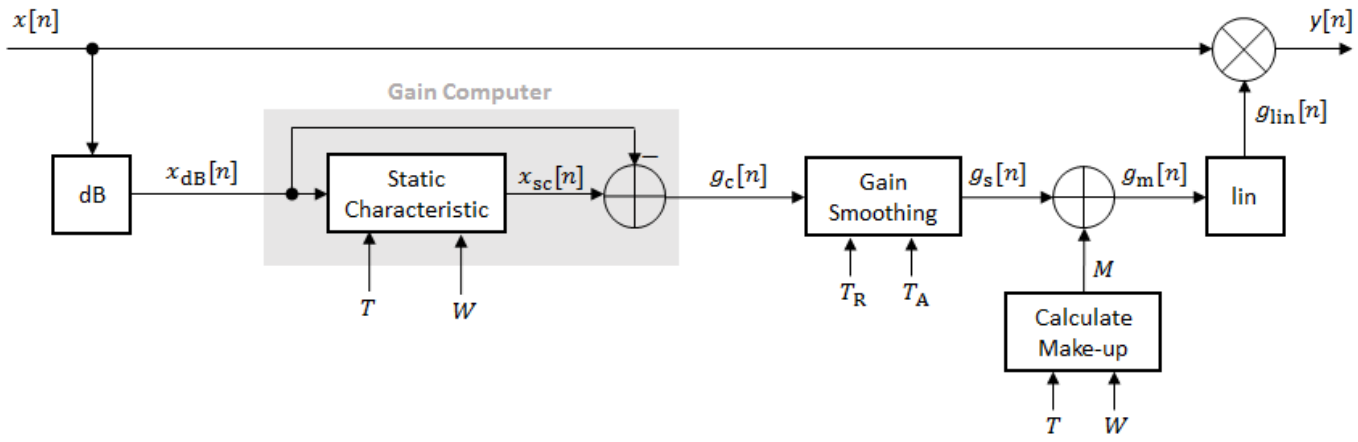
Tunable: No

Block Characteristics

Data Types	`double` | `single`
Direct Feedthrough	`no`
Multidimensional Signals	`no`
Variable-Size Signals	`yes`
Zero-Crossing Detection	`no`

Algorithms

The Limiter block processes a signal frame by frame and element by element.



1 The N-point signal, $x[n]$, is converted to decibels:

$$x_{\mathrm{dB}}[n] = 20 \times \log_{10}|x[n]|$$

2 $x_{\mathrm{dB}}[n]$ 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:

$$x_{\mathrm{sc}}(x_{\mathrm{dB}}) = \begin{cases} x_{\mathrm{dB}} & x_{\mathrm{dB}} < \left(T - \frac{W}{2}\right) \\ x_{\mathrm{dB}} - \frac{\left(x_{\mathrm{dB}} - T + \frac{W}{2}\right)^2}{2W} & \left(T - \frac{W}{2}\right) \le x_{\mathrm{dB}} \le \left(T + \frac{W}{2}\right) \\ T & x_{\mathrm{dB}} > \left(T + \frac{W}{2}\right) \end{cases},$$

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:

$$x_{\mathrm{sc}}(x_{\mathrm{dB}}) = \begin{cases} x_{\mathrm{dB}} & x_{\mathrm{dB}} < T \\ T & x_{\mathrm{dB}} \ge T \end{cases}$$

3 The computed gain, $g_{\mathrm{c}}[n]$, is calculated as

$$g_{\mathrm{c}}[n] = x_{\mathrm{sc}}[n] - x_{\mathrm{dB}}[n].$$

4 $g_{\mathrm{c}}[n]$ is smoothed using specified attack and release time parameters:

$$g_{\mathrm{s}}[n] = \begin{cases} \alpha_{\mathrm{A}} g_{\mathrm{s}}[n-1] + (1-\alpha_{\mathrm{A}}) g_{\mathrm{c}}[n], & g_{\mathrm{c}}[n] \le g_{\mathrm{s}}[n-1] \\ \alpha_{\mathrm{R}} g_{\mathrm{s}}[n-1] + (1-\alpha_{\mathrm{R}}) g_{\mathrm{c}}[n], & g_{\mathrm{c}}[n] > g_{\mathrm{s}}[n-1] \end{cases}$$

The attack time coefficient, α_{A}, is calculated as

$$\alpha_{\mathrm{A}} = \exp\left(\frac{-\log(9)}{Fs \times T_{\mathrm{A}}}\right).$$

The release time coefficient, α_{R}, is calculated as

$$\alpha_{\mathrm{R}} = \exp\left(\frac{-\log(9)}{Fs \times T_{\mathrm{R}}}\right).$$

T_A is the attack time period, specified by the **Attack time (s)** parameter. T_{R} is the release time period, specified by the **Release time (s)** parameter. *Fs* is the input sampling rate, specified by the **Inherit sample rate from input** or **Input sample rate (Hz)** parameter.

5 If the **Make-up gain (dB)** parameter is set to `Auto`, the make-up gain is calculated as the negative of the computed gain for a 0 dB input:

$$M = -x_{\mathrm{sc}}(x_{\mathrm{dB}} = 0)$$

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, $g_{\mathrm{s}}[n]$:

$$g_{\mathrm{m}}[n] = g_{\mathrm{s}}[n] + M$$

7 The calculated gain in dB, $g_{\mathrm{m}}[n]$, is translated to a linear domain:

$$g_{\mathrm{lin}}[n] = 10^{\left(\frac{g_{\mathrm{m}}[n]}{20}\right)}$$

8 The output of the dynamic range limiter is given as

$$y[n] = x[n] \times g_{\mathrm{lin}}[n].$$

Version History

Introduced in R2016a

References

[1] Giannoulis, Dimitrios, Michael Massberg, and Joshua D. Reiss. "Digital Dynamic Range Compressor Design –– A Tutorial And Analysis." *Journal of Audio Engineering Society*. Vol. 60, Issue 6, 2012, pp. 399–408.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

Compressor | Expander | Noise Gate | `limiter`

Topics

“Dynamic Range Control”

Loudness Meter

Standard-compliant loudness measurements

Libraries:
Audio Toolbox / Measurements

Description

The Loudness Meter block measures the loudness and true-peak of an audio signal based on EBU R 128 and ITU-R BS.1770-4 standards.

Ports

Input

Port_1 — Input signal
matrix | 1-D vector

- Matrix input –– Each column of the input is treated as an independent channel. If you use the default **Channel weights**, specify the input channels in order: [Left, Right, Center, Left surround, Right surround].
- 1-D vector input –– The input is treated as a single channel.

Data Types: `single` | `double`

Output

M — Momentary loudness measurement
column vector

The block outputs a column vector with the same data type and number of rows as the input signal.

Data Types: `single` | `double`

S — Short-term loudness measurement
column vector

The block outputs a column vector with the same data type and number of rows as the input signal.

Data Types: `single` | `double`

TP — True-peak value
real scalar

The block outputs a real scalar with the same data type as the input signal.

Dependencies

To enable this port, select the **Output true-peak value** parameter.

Data Types: `single` | `double`

Parameters

If a parameter is listed as tunable, then you can change its value during simulation.

Channel weights — Linear weighting applied to each input channel
`[1, 1, 1, 1.41, 1.41]` (default) | nonnegative row vector

The number of elements of the row vector must be equal to or greater than the number of input channels. Excess values in the vector are ignored.

The default channel weights follow the ITU-R BS.1170-4 standard. To use the default channel weights, specify the input to the Loudness Meter block as a matrix whose columns correspond to channels in this order: [Left, Right, Center, Left surround, Right surround].

It is a best practice to specify the channel weights in order: [Left, Right, Center, Left surround, Right surround].

Tunable: Yes

Use relative scale for loudness measurements — Specify block to output loudness measurements relative to target level
off (default) | on

- On — The loudness measurements are relative to the value specified by **Target loudness level (LUFS)**. The output of the block is returned in loudness units (LU).
- Off — The loudness measurements are absolute, and returned in loudness units full scale (LUFS).

Tunable: No

Target loudness level (LUFS) — Reference level for relative loudness measurements
–23 (default) | real scalar

For example, if the **Target loudness level (LUFS)** is –23, then a loudness value of –24 LUFS is reported as –1 LU.

Tunable: Yes

Dependencies

To enable this parameter, select the **Use relative scale for loudness measurements** parameter.

Output true-peak value — Add output port for true-peak value
off (default) | on

When you select this parameter, an additional output port, **TP**, is added to the block. The **TP** port outputs the true-peak value of the input frame.

Tunable: No

Inherit sample rate from input — Specify source of input sample rate
on (default) | off

When you select this parameter, the block inherits its sample rate from the input signal. When you clear this parameter, you specify the sample rate in **Input sample rate (Hz)**.

Tunable: No

Input sample rate (Hz) — Sample rate of input
`44100` (default) | scalar

Tunable: Yes

Dependencies

To enable this parameter, clear the **Inherit sample rate from input** parameter.

Simulate using — Specify type of simulation to run
`Code generation` (default) | `Interpreted execution`

- `Code generation` –– Simulate model using generated C code. The first time you run a simulation, Simulink generates C code for the block. The C code is reused for subsequent simulations, as long as the model does not change. This option requires additional startup time but the speed of the subsequent simulations is comparable to `Interpreted execution`.
- `Interpreted execution` –– Simulate model using the MATLAB interpreter. This option shortens startup time but has a slower simulation speed than `Code generation`. In this mode, you can debug the source code of the block.

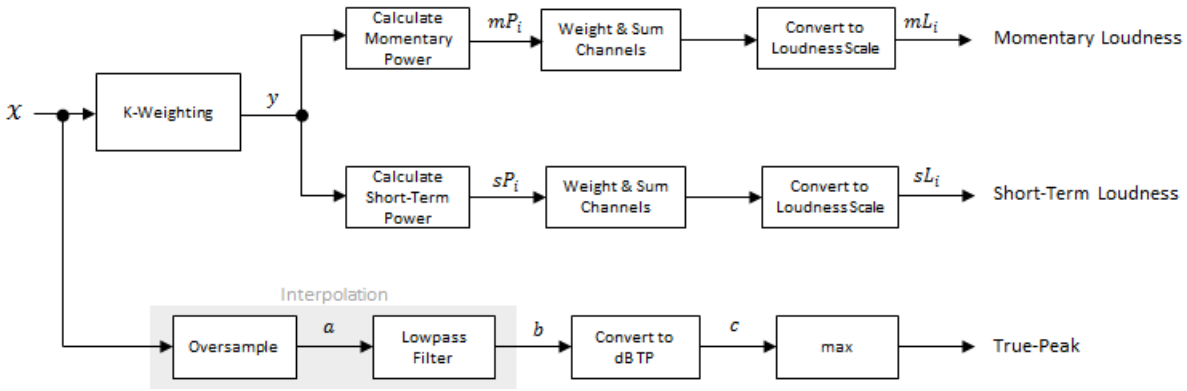
Tunable: No

Block Characteristics

Data Types	`double` | `single`
Direct Feedthrough	`no`
Multidimensional Signals	`no`
Variable-Size Signals	`yes`
Zero-Crossing Detection	`no`

Algorithms

The Loudness Meter block calculates the momentary loudness, short-term loudness, and true-peak value of an audio signal. You can specify any number of channels and nondefault channel weights used for loudness measurements. The block algorithm is described for the general case of *n* channels and default channel weights.



Loudness Measurements

The input channels, *x*, pass through a K-weighted filter implemented using the algorithm of the Weighting Filter block. The K-weighted filter shapes the frequency spectrum to reflect perceived loudness.

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 ith segment.
   - w is the segment length in samples.

2. The momentary loudness, *mL*, is computed for each segment:

$$mL_i = -0.691 + 10\log_{10}\left(\sum_{c=1}^{n} G_c \times mP_{(i,c)}\right) \quad LUFS$$

   - G_c is the weighting for channel c.

   mL is the momentary loudness returned by your Loudness Meter block.

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:

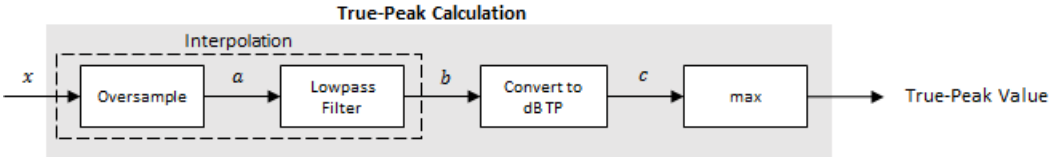
$$sL_i = -0.691 + 10\log_{10}\left(\sum_{c=1}^{n} G_c \times sP_{(i,c)}\right) \quad LUFS$$

- 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. The algorithm does not consider an input sample rate below 750 Hz.

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:

$$c = 20 \times \log_{10}(|b|)$$

4 The true-peak is determined as the maximum of the converted signal *c*.

Version History

Introduced in R2016b

References

[1] International Telecommunication Union; Radiocommunication Sector. *Algorithms to Measure Audio Programme Loudness and True-Peak Audio Level.* ITU-R BS.1770-4. 2015.

[2] European Broadcasting Union. *Loudness Normalisation and Permitted Maximum Level of Audio Signals.* EBU R 128. 2014.

[3] European Broadcasting Union. *Loudness Metering: 'EBU Mode' Metering to Supplement EBU R 128 Loudness Normalization.* EBU R 128 Tech 3341. 2014.

Extended Capabilities

C/C++ Code Generation

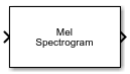
Generate C and C++ code using Simulink® Coder™.

See Also

`integratedLoudness` | `loudnessMeter`

Mel Spectrogram

Extract mel spectrogram from audio



Libraries:
Audio Toolbox / Features

Description

The Mel Spectrogram block extracts the mel spectrogram from the audio input signal. A mel spectrogram contains an estimate of the short-term, time-localized frequency content of the input signal in the mel frequency scale.

Ports

Input

Port_1 — Audio input
column vector | matrix

Audio input signal, specified as a column vector or a matrix. When you specify a matrix, the block treats columns as independent audio channels.

Data Types: `single` | `double`

Output

spec — Mel spectrogram
matrix | 3-D array

Mel spectrogram, returned as a matrix or 3-D array. The dimensions of **spec** are *L*-by-*M*-by-*N*, where:

- *L* is the number of spectra, which is determined by the **Number of spectra** parameter.
- *M* is the number of bands, which is determined by the **Number of bands** parameter.
- *N* is the number of channels in the input audio signal.

Trailing singleton dimensions are removed from the output.

This port is unnamed until you select the **Output center frequencies** parameter.

Data Types: `single` | `double`

fvec — Center frequencies
row vector

Center frequencies of the bandpass filters in Hz, returned as a row vector with number of elements equal to the number of bands.

Dependencies

To enable this port, select the **Output frequency vector** parameter.

Data Types: `single` | `double`

Parameters

Filter Bank Parameters

Number of bands — Number of bandpass filters

`32` (default) | positive integer

Number of bandpass filters, specified as a positive integer.

Auto-determine frequency range — Automatically determine frequency range

`on` (default) | `off`

When you select this parameter, the block sets the **Frequency range** to `[0,fs/2]`, where `fs` is the sample rate. The sample rate is determined by the **Inherit sample rate from input** and **Input sample rate (Hz)** parameters.

Frequency range (Hz) — Frequency range over which to design auditory filter bank

`[0,22050]` (default) | two-element row vector

Frequency range in Hz over which to design the auditory filter bank, specified as a two-element row vector.

Dependencies

To enable this parameter, clear the **Auto-determine frequency range** parameter.

Filter bank design domain — Domain to design filter bank

`linear` (default) | `warped`

Domain in which the block designs the filter bank, specified as `linear` or `warped`. Set the filter bank design domain to `linear` to design the bandpass filters in the linear (Hz) domain. Set the filter bank design domain to `warped` to design the bandpass filters in the warped (mel) domain.

Filter bank normalization — Normalization technique for filter bank

`bandwidth` (default) | `area` | `none`

Normalization technique used for the filter bank weights, specified as `bandwidth`, `area`, or `none`.

- `bandwidth` –– Normalize the weights of each bandpass filter by the corresponding bandwidth of the filter.
- `area` –– Normalize the weights of each bandpass filter by the corresponding area of the bandpass filter.
- `none` –– The block does not normalize the weights of the filters.

Visualize filter bank — Open plot to visualize filter bank

button

Open plot to visualize the filters in the frequency domain.

Output frequency vector — Specify additional output port for center frequencies

`off` (default) | `on`

When you select this parameter, the block displays an additional output port, **fvec**. This port outputs the center frequencies of the bandpass filters.

Spectrogram Parameters

Window — Analysis window

`hamming(1024,'periodic')` (default) | real vector

Analysis window applied in the time domain, specified as a real vector.

Normalize window — Normalize analysis window

`on` (default) | `off`

When you select this parameter, the block applies window normalization.

Overlap length — Overlap length of adjacent analysis windows

`512` (default) | integer in the range [0, `windowLength`)

Overlap length of adjacent analysis windows, specified as an integer in the range [0, `windowLength`), where `windowLength` is the length of the analysis window, which is specified by **Window**.

Auto-determine FFT length — Automatically determine FFT length

`on` (default) | `off`

When you select this parameter, the block automatically sets the FFT length to the window length, `numel(Window)`.

FFT length — Number of DFT points

`1024` (default) | positive integer

Number of points used to calculate the DFT, specified as a positive integer.

Dependencies

To enable this parameter, clear the **Auto-determine FFT length** parameter.

Spectrum type — Type of spectrum

`magnitude` (default) | `power`

Type of spectrum, specified as `magnitude` or `power`.

Number of spectra — Number of spectra

`1` (default) | positive integer

Number of spectra in the spectrogram, specified as a positive integer.

Number of spectra overlap — Number of overlapped spectra

`0` (default) | integer in the range [0, **Number of spectra**).

Number of spectra overlapped across consecutive spectrograms, specified as an integer in the range [0, **Number of spectra**)

Simulation Parameters

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 the **Input sample rate (Hz)** parameter.

Input sample rate (Hz) — Sample rate of input

`44.1e3` (default) | positive scalar

Input sample rate in Hz, specified as a real positive scalar.

Dependencies

To enable this parameter, clear the **Inherit sample rate from input** parameter.

Block Characteristics

Data Types	`double` | `single`
Direct Feedthrough	no
Multidimensional Signals	no
Variable-Size Signals	no
Zero-Crossing Detection	no

Version History

Introduced in R2022a

R2023a: Generate optimized C/C++ code for computing mel spectrogram

The Mel Spectrogram block supports optimized C/C++ code generation using single instruction, multiple data (SIMD) instructions.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

The Mel Spectrogram block supports optimized code generation using single instruction, multiple data (SIMD) instructions. For more information about SIMD code generation, see “Generate SIMD Code from Simulink Blocks” (Simulink Coder).

See Also

Blocks
Auditory Spectrogram | Design Auditory Filter Bank | Design Mel Filter Bank | MFCC

Functions
`designAuditoryFilterBank` | `melSpectrogram`

Objects
`audioFeatureExtractor`

MFCC

Extract mel-frequency cepstral coefficients from audio

Libraries:
Audio Toolbox / Features

Description

The MFCC block extracts feature vectors containing the mel-frequency cepstral coefficients (MFCCs), as well as their delta and delta-delta features, from the audio input signal. MFCCs are popular features extracted from speech signals for use in classification tasks.

Ports

Input

Port_1 — Audio input
column vector | matrix

Audio input signal, specified as a column vector or a matrix. When you specify a matrix, the block treats columns as independent audio channels.

Data Types: `single` | `double`

Output

Port_1 — MFCC features
matrix | 3-D array

MFCC features returned as a matrix or 3-D array. The features include the MFCCs themselves and optionally include the delta and delta-delta features of the MFCCs. The dimensions of the output are *L*-by-*M*-by-*N*, where:

- *L* is the number of feature vectors, which is specified by the **Number of feature vectors** parameter.
- *M* is the number of features in each feature vector, which is determined by the **Number of cepstral coefficients**, **Append delta**, and **Append delta-delta** parameters.
- *N* is the number of channels in the input audio signal.

Trailing dimensions of size 1 are removed from the output.

Data Types: `single` | `double`

Parameters

Mel-Frequency Cepstral Coefficients

Window — Analysis window

`hamming(1024,'periodic')` (default) | real vector

Analysis window applied to the input signal in the time domain, specified as a real vector.

Overlap length — Number of overlapping samples between adjacent windows

`512` (default) | integer in the range [0, `windowLength`)

Number of overlapping samples between adjacent windows, specified as an integer in the range [0, `windowLength`), where `windowLength` is the length of the analysis window and is specified by the **Window** parameter.

Number of cepstral coefficients — Number of cepstral coefficients in each feature vector

`13` (default) | positive integer greater than 1

Number of cepstral coefficients in each feature vector, specified as a positive integer greater than 1.

Rectification — Type of nonlinear rectification

`Logarithm` (default) | `Cubic root`

Type of nonlinear rectification applied to the spectrum prior to the discrete cosine transform, specified as `Logarithm` or `Cubic root`.

Append delta — Append delta of MFCCs to feature vectors

`on` (default) | `off`

When you select this parameter, the block appends the delta of the MFCCs to the coefficients in each feature vector. The delta is an approximation of the first derivative of the MFCCs with respect to time. The number of delta features is equal to the number of MFCCs, which is specified by **Number of cepstral coefficients**.

Append delta-delta — Append delta-delta of MFCCs to feature vectors

`on` (default) | `off`

When you select this parameter, the block appends the delta-delta of the MFCCs to each output feature vector. The delta-delta is an approximation of the second derivative of the MFCCs with respect to time. The number of delta-delta features is equal to the number of MFCCs, which is specified by **Number of cepstral coefficients**.

The block appends the delta-delta after the delta in the feature vectors if you also select the **Append delta** parameter.

Delta window length — Number of coefficients for calculating delta and delta-delta

`9` (default) | odd integer greater than 2

Number of coefficients for calculating delta and delta-delta, specified as an odd integer greater than 2.

Output Buffering

Number of feature vectors — Number of MFCC feature vectors in output

`1` (default) | positive integer

Number of MFCC feature vectors in output, specified as a positive integer. The block buffers the output to return the specified number of feature vectors.

Number of overlapped feature vectors — Number of feature vectors overlapped in output

`0` (default) | nonnegative integer

Number of feature vectors the block overlaps in the output, specified as a nonnegative integer less than **Number of feature vectors**.

Simulation Parameters

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 the **Input sample rate (Hz)** parameter.

Input sample rate (Hz) — Sample rate of input

`44.1e3` (default) | positive scalar

Input sample rate in Hz, specified as a positive scalar.

Dependencies

To enable this parameter, clear the **Inherit sample rate from input** parameter.

Mel Filter Bank Design

Number of bands — Number of bands in mel filter bank

`32` (default) | positive integer

Number of bands in mel filter bank, specified as a positive integer.

Auto-determine frequency range — Automatically determine frequency range

`on` (default) | `off`

When you select this parameter, the block sets the **Frequency range** to `[0,fs/2]`, where `fs` is the sample rate. The block determines the sample rate using the **Inherit sample rate from input** and **Input sample rate (Hz)** parameters.

Frequency range (Hz) — Frequency range of mel filter bank

`[0,22050]` (default) | two-element row vector

Frequency range in Hz of mel filter bank, specified as a two-element row vector.

Dependencies

To enable this parameter, clear the **Auto-determine frequency range** parameter.

Filter bank design domain — Design domain of mel filter bank

`linear` (default) | `warped`

Design domain of mel filter bank, specified as `linear` or `warped`.

Filter bank normalization — Normalization technique for filter bank

`bandwidth` (default) | `area` | `none`

Normalization technique that the block uses for the filter bank weights, specified as `bandwidth`, `area`, or `none`.

- `bandwidth` –– Normalize the weights of each bandpass filter by the corresponding bandwidth of the filter.
- `area` –– Normalize the weights of each bandpass filter by the corresponding area of the bandpass filter.
- `none` –– The block does not normalize the weights of the filters.

Spectrogram

Normalize window — Normalize analysis window

`on` (default) | `off`

When you select this parameter, the block applies window normalization.

Spectrum type — Type of spectrum

`power` (default) | `magnitude`

Type of spectrum, specified as `power` or `magnitude`.

Auto-determine FFT length — Automatically determine FFT length

`on` (default) | `off`

When you select this parameter, the block automatically sets the FFT length to the window length. The window length is determined by the **Window** parameter.

FFT length — Number of DFT points

`1024` (default) | positive integer

Number of points used to calculate the DFT, specified as a positive integer.

Dependencies

To enable this parameter, clear the **Auto-determine FFT length** parameter.

Block Characteristics

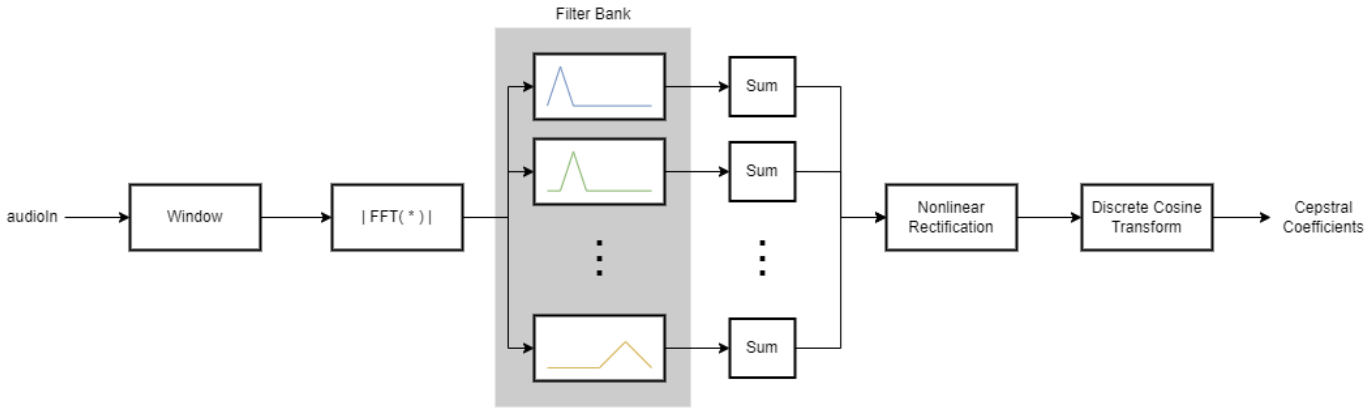
Data Types	`double` | `single`
Direct Feedthrough	`no`
Multidimensional Signals	`no`
Variable-Size Signals	`no`
Zero-Crossing Detection	`no`

Algorithms

MFCC

Mel-frequency cepstrum coefficients are popular features extracted from speech signals for use in recognition tasks. In the source-filter model of speech, cepstral coefficients are understood to represent the filter (vocal tract). The vocal tract frequency response is relatively smooth, whereas the source of voiced speech can be modeled as an impulse train. As a result, the vocal tract can be estimated by the spectral envelope of a speech segment.

The motivating idea of mel-frequency cepstral coefficients is to compress information about the vocal tract (smoothed spectrum) into a small number of coefficients based on an understanding of the cochlea. Although there is no hard standard for calculating the coefficients, the basic steps are outlined by the diagram.



Delta

The delta of an audio feature x is a least-squares approximation of the local slope of a region centered on sample $x(k)$, which includes M samples before the current sample and M samples after the current sample.

$$delta = \frac{\sum_{k=-M}^{M} k\,x(k)}{\sum_{k=-M}^{M} k^2}$$

The delta window length defines the length of the region from –M to M.

Version History

Introduced in R2022b

R2023a: Generate optimized C/C++ code for computing MFCCs

The MFCC block supports optimized C/C++ code generation using single instruction, multiple data (SIMD) instructions.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

The MFCC block supports optimized code generation using single instruction, multiple data (SIMD) instructions. For more information about SIMD code generation, see “Generate SIMD Code from Simulink Blocks” (Simulink Coder).

See Also

Blocks

Cepstral Coefficients | Audio Delta | Design Mel Filter Bank

Functions

`mfcc` | `audioDelta` | `cepstralCoefficients` | `designAuditoryFilterBank` | `melSpectrogram`

Objects

`audioFeatureExtractor`

MIDI Controls

Output values from controls on MIDI control surface

Libraries:
Audio Toolbox / Sources
DSP 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)`

Output mode for MIDI control value, specified as `Normalized (0-1)` or `RAW MIDI (0-127)`.

Block Characteristics

Data Types	`double` | `integer`
Direct Feedthrough	`no`
Multidimensional Signals	`no`
Variable-Size Signals	`no`
Zero-Crossing Detection	`no`

Tips

- The MIDI Controls block is not supported for rapid accelerator mode.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

The executable generated from this block relies on prebuilt dynamic library files (.dll files) included with MATLAB. Use the `packNGo` function to package the code generated from this object and all the relevant files in a compressed zip file. Using this zip file, you can relocate, unpack, and rebuild your project in another development environment where MATLAB is not installed. For more details, see "Run Audio I/O Features Outside MATLAB and Simulink".

See Also

`parameterTuner` | **Audio Test Bench** | `midiid` | `midicontrols` | `midiread` | `midisync`

Topics

"MIDI Control Surface Interface"

Noise Gate

Dynamic range gate

Libraries:
Audio Toolbox / Dynamic Range Control

Description

The Noise Gate block performs dynamic range gating independently across each input channel. Dynamic range gating suppresses signals below a given threshold. The block uses specified attack, release, and hold times to achieve a smooth applied gain curve.

Ports

Input

x — Input signal
matrix | 1-D vector

- Matrix input –– Each column of the input is treated as an independent channel.
- 1-D vector input –– The input is treated as a single channel.

This port is unnamed unless you specify additional input ports.

Data Types: `single` | `double`

T — Threshold (dB)
scalar

Dependencies

To enable this port, select **Specify from input port** for the “Threshold (dB)” on page 5-0 parameter.

Data Types: `single` | `double`

AT — Attack time (s)
scalar

Dependencies

To enable this port, select **Specify from input port** for the “Attack time (s)” on page 5-0 parameter.

Data Types: `single` | `double`

RT — Release time (s)
scalar

Dependencies

To enable this port, select **Specify from input port** for the “Release time (s)” on page 5-0 parameter.

Data Types: `single` | `double`

HT — Hold time (s)
scalar

Dependencies

To enable this port, select **Specify from input port** for the “Hold time (s)” on page 5-0 parameter.

Data Types: `single` | `double`

Output

Y — Output signal
matrix

The Noise Gate block outputs a signal with the same data type as the input signal. The size of the output depends on the size of the input:

- Matrix input –– The block outputs a matrix the same size and data type as the input signal.
- 1-D vector input –– The block outputs an *N*-by-1 matrix (column vector), where *N* is the number of elements in the 1-D vector.

This port is unnamed until you select the **Output gain (dB)** parameter.

Data Types: `single` | `double`

G — Gain applied to each input sample
matrix

Dependencies

To enable this port, select the **Output gain (dB)** parameter.

Data Types: `single` | `double`

Parameters

If a parameter is listed as tunable, then you can change its value during simulation.

Main Tab

Threshold (dB) — Operation threshold
`–10` (default) | scalar in the range –140 to 0 inclusive

Operation threshold is the level below which gain is applied to the input signal.

To specify **Threshold (dB)** from an input port, select **Specify from input port** for the parameter.

Tunable: Yes

View static characteristic — Open static characteristic plot of dynamic range gate
button

The plot is updated automatically when parameters of the Noise Gate block change.

Tunable: Yes

Attack time (s) — Time for applied gain to ramp up
`0.05` (default) | scalar in the range 0 to 4 inclusive

Attack time is the time the applied gain takes to rise from 10% to 90% of its final value when the input goes below the threshold. The **Attack time (s)** parameter smooths the applied gain curve.

To specify **Attack time (s)** from an input port, select **Specify from input port** for the parameter.

Tunable: Yes

Release time (s) — Time for applied gain to ramp down
`0.2` (default) | scalar in the range 0 to 4 inclusive

Release time is the time the applied gain takes to drop from 90% to 10% of its final value when the input goes above the threshold. The **Release time (s)** parameter smooths the applied gain curve.

To specify **Release time (s)** from an input port, select **Specify from input port** for the parameter.

Tunable: Yes

Hold time (s) — Time during which applied gain holds steady
`0.05` (default) | scalar in the range 0 to 4

Hold time is the period for which the (negative) gain is held before starting to decrease towards its steady state value when the input level drops below the threshold.

To specify **Hold time (s)** from an input port, select **Specify from input port** for the parameter.

Tunable: Yes

Inherit sample rate from input — Specify source of input sample rate
`on` (default) | `off`

When you select this parameter, the block inherits its sample rate from the input signal. When you clear this parameter, you specify the sample rate in **Input sample rate (Hz)**.

Tunable: No

Input sample rate (Hz) — Specify input sample rate
`44100` (default) | scalar

Tunable: Yes

Dependencies

To enable this parameter, clear the **Inherit sample rate from input** parameter.

Advanced Tab

Output gain (dB) — Gain applied on each input sample
`off` (default) | `on`

When you select this parameter, an additional output port, **G**, is added to the block. The **G** port outputs the gain applied on each input channel in dB.

Tunable: No

Sidechain — Enable sidechain input
`off` (default) | `on`

When you select this parameter, an additional input port **SC** is added to the block. The **SC** port enables dynamic range gating of the input signal **x** using a separate sidechain signal.

The datatype and (frame) length input to the **SC** port must be the same as the input to the **x** port.

The number of channels of the sidechain input must be equal to the number of channels of `x` or be equal to one.

- `Sidechain channel count is equal to one` –– The computed gain, `G`, based on this channel is applied to all channels of `x`.
- `Sidechain channel count is equal to channel count of x` –– The computed gain, `G`, for each sidechain channel is applied to the corresponding channel of `x`.

Tunable: No

Simulate using — Specify type of simulation to run
`Interpreted execution` (default) | `Code generation`

- `Interpreted execution` –– Simulate model using the MATLAB interpreter. This option shortens startup time and has a simulation speed comparable to `Code generation`. In this mode, you can debug the source code of the block.
- `Code generation` –– Simulate model using generated C code. The first time you run a simulation, Simulink generates C code for the block. The C code is reused for subsequent simulations, as long as the model does not change. This option requires additional startup time, but the speed of the subsequent simulations is comparable to `Interpreted execution`.

Tunable: No

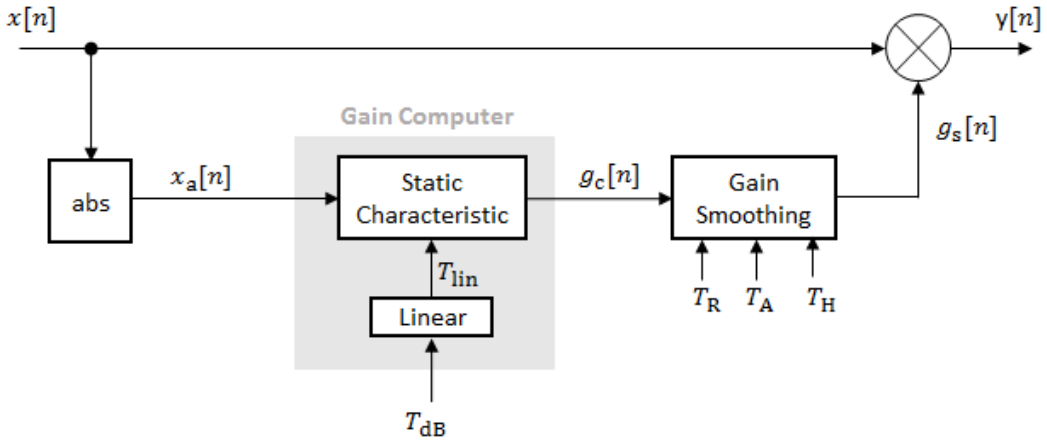
Block Characteristics

Data Types	`double` | `single`
Direct Feedthrough	`no`

Multidimensional Signals	no
Variable-Size Signals	yes
Zero-Crossing Detection	no

Algorithms

The Noise Gate block processes a signal frame by frame and element by element.



1 The N-point signal, $x[n]$, is converted to magnitude:

$$x_a[n] = |x[n]|$$

2 $x_a[n]$ passes through the gain computer. The gain computer uses the static characteristic properties of the dynamic range gate to apply a brickwall gain for signal below the threshold:

$$g_c(x_a) = \begin{cases} 0 & x_a < T_{lin} \\ 1 & x_a \geq T_{lin} \end{cases}$$

T_{lin} is the threshold property converted to a linear domain:

$$T_{lin} = 10^{\left(T_{dB}/20\right)}.$$

3 The computed gain, $g_c[n]$, is smoothed using specified attack, release, and hold time parameters:

$$g_s[n] = \begin{cases} \alpha_A g_s[n-1] + (1-\alpha_A) g_c[n] & (C_A > T_H) \ \& \ (g_c[n] \leq g_s[n-1]) \\ g_s[n-1] & C_A \leq T_H \\ \alpha_R g_s[n-1] + (1-\alpha_R) g_c[n] & g_c[n] > g_s[n-1] \end{cases}$$

C_A is the hold counter for attack. The limit, T_H, is determined by the **Hold time (s)** parameter.

The attack time coefficient, α_A, is calculated as

$$\alpha_A = \exp\left(\frac{-\log(9)}{Fs \times T_A}\right).$$

The release time coefficient, α_R, is calculated as

$$\alpha_R = \exp\left(\frac{-\log(9)}{Fs \times T_R}\right).$$

T_A is the attack time period, specified by the **Attack time (s)** parameter. T_R is the release time period, specified by the **Release time (s)** parameter. *Fs* is the input sampling rate, specified by the **Inherit sample rate from input** or **Input sample rate (Hz)** parameter.

4 The output of the dynamic range gate is given as

$$y[n] = x[n] \times g_s[n].$$

Version History

Introduced in R2016a

References

[1] Giannoulis, Dimitrios, Michael Massberg, and Joshua D. Reiss. "Digital Dynamic Range Compressor Design –– A Tutorial And Analysis." *Journal of Audio Engineering Society*. Vol. 60, Issue 6, 2012, pp. 399–408.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

`noiseGate` | Compressor | Expander | Limiter

Topics

“Dynamic Range Control”

Octave Filter

Octave-band and fractional octave-band filter

Libraries:
Audio 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

x — Input signal
matrix | 1-D vector

- Matrix input –– Each column of the input is treated as an independent channel.
- 1-D vector input –– The input is treated as a single channel.

This port is unnamed unless you specify additional input ports.

Data Types: `single` | `double`

CF — Center frequency (Hz)
scalar in the range `3` to `22,000` inclusive

Dependencies

To enable this port, select **Specify from input port** for the “Center frequency (Hz)” on page 5-0 parameter.

Data Types: `single` | `double`

Output

Port_1 — Output signal
matrix

The Octave Filter block outputs a signal with the same data type as the input signal. The size of the output depends on the size of the input:

- Matrix input –– The block outputs a matrix the same size and data type as the input signal.
- 1-D vector input –– The block outputs an *N*-by-1 matrix (column vector), where *N* is the number of elements in the 1-D vector.

Data Types: `single` | `double`

Parameters

If a parameter is listed as tunable, then you can change its value during simulation.

Filter order — Order of octave filter
6 (default) | even integer

Tunable: No

Center frequency (Hz) — Center frequency of octave filter
`1000` (default) | scalar in the range 3 to `22,000` inclusive

- The maximum center frequency is the value that causes the upper band edge to be equal to the Nyquist frequency, *Fs*/2. Frequencies above this value are saturated.
- The minimum center frequency is the value that causes the lower band edge to be equal to 1 Hz. Frequencies below this value are quantized to 1 Hz.

To specify **Center frequency (Hz)** from an input port, select **Specify from input port** for the parameter.

Tunable: Yes

Bandwidth — Filter bandwidth in octaves
`1 octave` (default) | `2/3 octave` | `1/2 octave` | `1/3 octave` | `1/6 octave` | `1/12 octave` | `1/24 octave` | `1/48 octave`

Tunable: Yes

Oversample the input by 2 for this filter — Oversample toggle
off (default) | on

- off –– The Octave Filter block runs at the input sample rate.
- on –– The Octave Filter block runs at two times the input sample rate. Oversampling minimizes the frequency warping effects introduced by the bilinear transformation. An FIR halfband interpolator implements oversampling before octave filtering. A halfband decimator reduces the sample rate back the input sampling rate after octave filtering.

Tunable: No

Inherit sample rate from input — Specify source of input sample rate
off (default) | on

When you select this parameter, the block inherits its sample rate from the input signal. When you clear this parameter, you specify the sample rate in **Input sample rate (Hz)**.

Tunable: No

Input sample rate (Hz) — Sample rate of input
`44100` (default) | scalar

Tunable: Yes

Dependencies

To enable this parameter, clear the **Inherit sample rate from input** parameter.

Simulate using — Specify type of simulation to run
`Code generation` (default) | `Interpreted execution`

- `Code generation` –– Simulate the model using generated C code. The first time you run a simulation, Simulink generates C code for the block. The C code is reused for subsequent simulations as long as the model does not change. This option requires additional startup time, but the speed of the subsequent simulations is comparable to `Interpreted execution`.
- `Interpreted execution` –– Simulate the model using the MATLAB interpreter. This option shortens startup time but has a slower simulation speed than `Code generation`. In this mode, you can debug the source code of the block.

Tunable: No

Mask for attenuation limits — Create a mask for filter response visualization
`No mask` (default) | `Class 0` | `Class 1` | `Class 2`

The mask attenuation limits are defined in the ANSI S1.11-2004 standard.

- If the mask is green, the design is compliant.
- If the mask is red, the design breaks compliance.

Tunable: Yes

Visualize filter response — Open plot to visualize magnitude response and compliance mask
button

A 2048-point FFT is used to calculate the magnitude response.

Tunable: Yes

Variable name — Variable name of exported filter
`myFilt` (default) | valid variable name

Name of the variable in the base workspace to contain the filter when it is exported. The name must be a valid MATLAB variable name.

Overwrite variable if it already exists — Overwrite variable if it already exists
`on` (default) | `off`

When you select this parameter, exporting the filter overwrites the variable specified by the **Variable name** parameter if it already exists in the base workspace. If you do not select this parameter and

the specified variable already exists in the workspace, exporting the filter creates a new variable with an underscore and a number appended to the variable name. For example, if the variable name is `var` and it already exists, the exported variable will be named `var_1`.

Export filter to workspace — Export filter to workspace button

Export the filter to the base workspace in the variable specified by the **Variable name** parameter.

Tips

- You cannot export the filter if you have enabled the **Inherit sample rate from input** parameter and the model is not running.
- You cannot export the filter if you are specifying filter characteristics from input ports.

Block Characteristics

Data Types	`double` | `single`
Direct Feedthrough	`no`
Multidimensional Signals	`no`
Variable-Size Signals	`yes`
Zero-Crossing Detection	`no`

More About

Band Edge

A band edge frequency refers to the lower or upper edge of the passband of a bandpass filter.

Center Frequency of Octave Filter

The center frequency of an octave filter is the geometric mean of the lower- and upper-band edge frequencies.

Algorithms

Octave Bandwidth to Band Edge Conversion

The Octave Filter block uses the specified center frequency and filter bandwidth in octaves to determine the normalized band edges [2].

First the block computes the upper and lower band edge frequencies:

$$f_{\mathrm{pa}} = f_{\mathrm{c}} \times G^{-1/2b}$$

$$f_{\mathrm{pb}} = f_{\mathrm{c}} \times G^{1/2b}$$

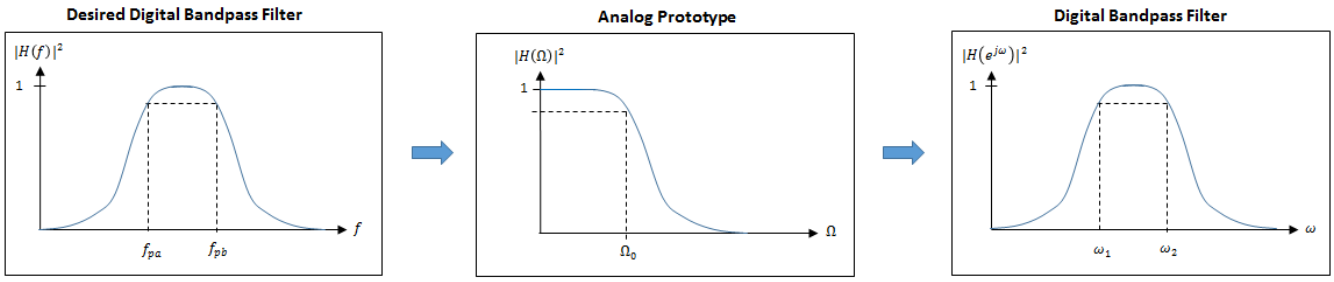
- f_c is the normalized center frequency specified by the **Center frequency (Hz)** parameter.
- b is the octave bandwidth specified by the **Bandwidth** parameter. For example, if **Bandwidth** is specified as `1/3 octave`, the value of b is 3.
- G is a conversion constant:

  $G = 10^{3/10}$.

Digital Filter Design

The Octave Filter block implements a higher-order digital bandpass filter design method as specified in [1].

In this design method, a desired digital bandpass filter maps to a Butterworth lowpass analog prototype, which is then mapped back to a digital bandpass filter:



1 The analog Butterworth filter is expressed as a cascade of second-order sections:

$H(s) = H_1(s)H_2(s)\cdots H_{2N}(s)$, where:

- $H_i(s) = \dfrac{1}{1 - 2\dfrac{s}{\Omega_0}\cos\theta_i + \dfrac{s^2}{\Omega_0^2}}, \quad i = 1, 2, \ldots, 2N$
- $\theta_i = \dfrac{\pi}{2N}(N - 1 + 2i), \quad i = 1, 2, \ldots, N, \ldots, 2N$

N is the filter order specified by the **Filter order** parameter.

2 The analog Butterworth filter is mapped to a digital filter using a bandpass version of the bilinear transformation:

$$s = \frac{1 - cz^{-1} + z^{-2}}{1 - z^{-2}},$$

where

$$c = \frac{\sin(\omega_{\text{pa}} + \omega_{\text{pb}})}{\sin\omega_{\text{pa}} + \sin\omega_{\text{pb}}}.$$

This mapping results in the following substitution:

$$\Omega_0 = \frac{c - \cos\omega_{\text{pb}}}{\sin\omega_{\text{pb}}}$$

3 The analog prototype is evaluated:

$$H_i(z) = \left. \frac{1}{1 - 2\frac{s}{\Omega_0}\cos\theta_i + \frac{s^2}{\Omega_0^2}} \right|_{s = \frac{1 - 2cz^{-1} + z^{-2}}{1 - z^{-2}}}$$

Because s is second-order in z, the bandpass version of the bilinear transformation is fourth-order in z.

Version History

Introduced in R2016b

References

[1] Orfanidis, Sophocles J. *Introduction to Signal Processing*. Englewood Cliffs, NJ: Prentice Hall, 2010.

[2] Acoustical Society of America. *American National Standard Specification for Octave-Band and Fractional-Octave-Band Analog and Digital Filters: ANSI S1.11-2004*. Melville, NY: Acoustical Society of America, 2009.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

`octaveFilter` | `weightingFilter` | Weighting Filter | Octave Filter Bank

Octave Filter Bank

Octave-band and fractional octave-band filter bank

Libraries:
Audio Toolbox / Filters

Description

The Octave Filter Bank block decomposes a signal into octave or fractional-octave subbands. 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.

Ports

Input

Port_1 — Input signal
vector | matrix

- Vector input –– The block treats the input as a single channel.
- Matrix input –– The block treats each column of the input as an independent channel.

Data Types: `single` | `double`

Output

Port_1 — Output signal
matrix | 3-D array

The Octave Filter Bank block outputs a signal with the same data type as the input signal. The shape of the output depends on the shape of the input, the number of filters in the bank, and whether or not you enable the **Bands as separate output ports** parameter.

If *F* is the number of filters in the bank, and the input signal is an *L*-by-*C* matrix, then the block returns an *L*-by-*F*-by-*C* array. If *C* is 1, then the block outputs a matrix.

- Vector input –– When you provide a vector input, the block outputs an *L*-by-*F* matrix, where *L* is the number of elements in the vector and *F* is the number of filters in the bank.
- Matrix input –– When you provide a matrix input, the block outputs a 3-D array with size *L*-by-*F*-by-*C*, where *C* is the number of channels in the matrix input.

Note When you enable the **Bands as separate output ports** parameter, each output is the same size as the input.

Data Types: `single` | `double`

Parameters

Bandwidth (octaves) — Bandwidth of filters specified 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`

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`, `1/48 octave`.

Frequency range (Hz) — Frequency range of filter bank (Hz)
`[22 22050]` (default) | two-element row vector of positive monotonically increasing values

Frequency range of the filter bank in Hz, specified as a two-element row vector of positive monotonically increasing values. The block places filter bank center frequencies according to the **Bandwidth (octaves)**, **Reference frequency (Hz)**, and **Octave ratio base** parameters. Filters that have a center frequency outside of **Frequency range (Hz)** are ignored.

Reference frequency (Hz) — Reference frequency of filter bank (Hz)
`1000` (default) | positive integer scalar

Reference frequency of the filter bank in Hz, specified as a positive integer scalar. The reference frequency defines one of the center frequencies. All other center frequencies are set relative to the reference frequency.

Filter order — Order of octave filters
`12` (default) | positive even integer

Order of the octave filters, specified as a positive even integer. The filter order applies to each individual filter in the bank.

Note The default filter order for the `octaveFilterBank` object is 2.

Octave ratio — Distance between filters
`Base ten (ANSI S1.11 preferred)` (default) | `Base two (musical scale)`

Octave ratio base, specified as `Base ten (ANSI S1.11 preferred)` or `Base two (musical scale)`. The octave ratio base determines the distribution of the center frequencies of the octave filters. The ANSI S1.11 standard recommends base 10. Base 2 is popular for music applications. **Base two (musical scale)** defines an octave as a factor of 2, and **Base ten (ANSI S1.11 preferred)** defines an octave as a factor of $10^{0.3}$.

Inherit sample rate from input — Allow sample rate to be set by input signal
`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 the **Input sample rate (Hz)** parameter.

Input sample rate (Hz) — Sample rate of input
`44100` (default) | positive scalar

When you select this parameter, the block accepts the sample rate from the user.

Dependencies

To enable this parameter, clear the **Inherit sample rate from input** parameter.

Bands as separate output ports — One output port per filter band
`off` (default) | `on`

When you select this parameter, the block provides an output port for each filter in the bank. Each output port is labeled with the center frequency of the filter and has a size identical to the input signal.

Simulate using — Specify type of simulation to run
`Interpreted execution` (default) | `Code generation`

- `Code generation` –– Simulate the model using generated C code. The first time you run a simulation, Simulink generates C code for the block. The C code is reused for subsequent simulations as long as the model does not change. This option requires additional startup time.
- `Interpreted execution` –– Simulate the model using the MATLAB interpreter. In this mode, you can debug the source code of the block.

View filter response — Open plot to visualize magnitude response
button

Octave filters are best understood when viewed on a logarithmic scale, which models how the human ear weights the spectrum. The block uses a 2048-point FFT to calculate the magnitude response. The filter bank's response is displayed on a log-frequency scale with a legend to indicate the center frequency of each filter.

Variable name — Variable name of exported filter bank
`myFilt` (default) | valid variable name

Name of the variable in the base workspace to contain the filter bank when it is exported. The name must be a valid MATLAB variable name.

Overwrite variable if it already exists — Overwrite variable if it already exists
`on` (default) | `off`

When you select this parameter, exporting the filter bank overwrites the variable specified by the **Variable name** parameter if it already exists in the base workspace. If you do not select this parameter and the specified variable already exists in the workspace, exporting the filter bank creates a new variable with an underscore and a number appended to the variable name. For example, if the variable name is `var` and it already exists, the exported variable will be named `var_1`.

Export filter to workspace — Export filter bank to workspace
button

Export the filter bank to the base workspace in the variable specified by the **Variable name** parameter.

Tips

You cannot export the filter bank if you have enabled the **Inherit sample rate from input** parameter and the model is not running.

Block Characteristics

Data Types	`double` | `single`
Direct Feedthrough	`no`
Multidimensional Signals	`no`
Variable-Size Signals	`yes`
Zero-Crossing Detection	`no`

More About

Band Edge

A band edge frequency refers to the lower or upper edge of the passband of a bandpass filter.

Center Frequency of Octave Filter

The center frequency of an octave filter is the geometric mean of the lower- and upper-band edge frequencies.

Algorithms

The Octave Filter Bank block is implemented as a parallel structure of octave filters. Individual octave filters are designed as described by `octaveFilter`. By default, the octave filter bank center frequencies are placed as specified by the `ANSI S1.11-2004` standard. You can modify the filter placements using the **Bandwidth (octaves)**, **Frequency range (Hz)**, **Reference frequency (Hz)**, and **Octave ratio** parameters.

Octave Bandwidth to Band Edge Conversion

The Octave Filter Bank block uses the specified **Frequency range (Hz)** and **Bandwidth (octaves)** to determine the normalized band edges [2].

First the block computes the upper and lower band edge frequencies:

$$f_{\mathrm{pa}} = f_{\mathrm{c}} \times G^{-1/2b}$$

$$f_{\mathrm{pb}} = f_{\mathrm{c}} \times G^{1/2b}$$

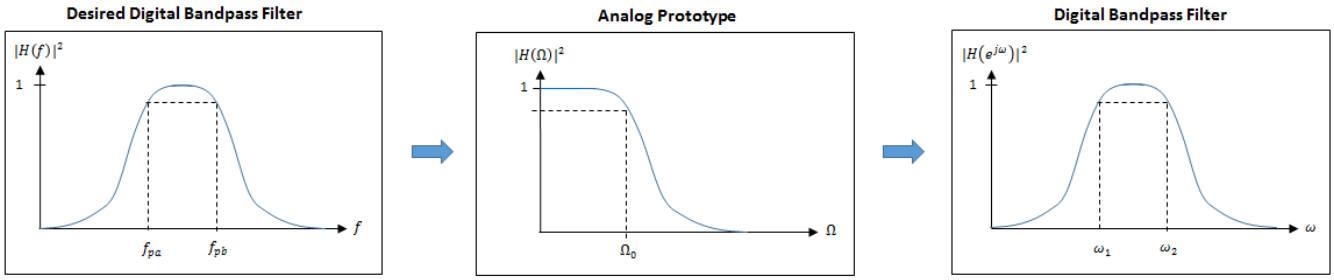
- f_{c} is the normalized center frequency specified by the **Bandwidth (octaves)** and **Frequency range (Hz)** parameters.
- b is the octave bandwidth specified by the **Bandwidth (octaves)** parameter. For example, if **Bandwidth (octaves)** is specified as `1/3 octave`, the value of b is 3.
- G is a conversion constant:

  $G = 10^{3/10}$.

Digital Filter Design

The Octave Filter Bank block implements a higher-order digital bandpass filter design method as specified in [1].

In this design method, a desired digital bandpass filter maps to a Butterworth lowpass analog prototype, which is then mapped back to a digital bandpass filter:



1. The analog Butterworth filter is expressed as a cascade of second-order sections:

   $H(s) = H_1(s)H_2(s)\cdots H_{2N}(s)$, where:

   - $H_i(s) = \dfrac{1}{1 - 2\dfrac{s}{\Omega_0}\cos\theta_i + \dfrac{s^2}{\Omega_0^2}}, \quad i = 1, 2, ..., 2N$
   - $\theta_i = \dfrac{\pi}{2N}(N - 1 + 2i), \quad i = 1, 2, ..., N, ..., 2N$

   N is the filter order specified by the **Filter order** parameter.

2. The analog Butterworth filter is mapped to a digital filter using a bandpass version of the bilinear transformation:

   $$s = \frac{1 - cz^{-1} + z^{-2}}{1 - z^{-2}},$$

   where

   $$c = \frac{\sin(\omega_{\text{pa}} + \omega_{\text{pb}})}{\sin\omega_{\text{pa}} + \sin\omega_{\text{pb}}}.$$

   This mapping results in the following substitution:

   $$\Omega_0 = \frac{c - \cos\omega_{\text{pb}}}{\sin\omega_{\text{pb}}}$$

3. The analog prototype is evaluated:

   $$H_{\text{i}}(z) = \left.\frac{1}{1 - 2\dfrac{s}{\Omega_0}\cos\theta_{\text{i}} + \dfrac{s^2}{\Omega_0^2}}\right|_{s = \frac{1 - 2cz^{-1} + z^{-2}}{1 - z^{-2}}}$$

Because s is second-order in z, the bandpass version of the bilinear transformation is fourth-order in z.

Version History

Introduced in R2021a

References

[1] Orfanidis, Sophocles J. *Introduction to Signal Processing*. Englewood Cliffs, NJ: Prentice Hall, 2010.

[2] Acoustical Society of America. *American National Standard Specification for Octave-Band and Fractional-Octave-Band Analog and Digital Filters: ANSI S1.11-2004*. Melville, NY: Acoustical Society of America, 2009.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

`octaveFilterBank` | `octaveFilter` | `weightingFilter` | Weighting Filter | Octave Filter

OpenL3

OpenL3 embeddings extraction network

Libraries:
Audio Toolbox / Deep Learning

Description

The OpenL3 block leverages a pretrained convolutional neural network that extracts feature embeddings from audio signals. These embeddings are powerful audio representations that can be used for tasks such as classification. This block requires Deep Learning Toolbox.

Ports

Input

Port_1 — Spectrograms
matrix | 4-D array

Spectrograms generated from audio, specified as an *N*-by-*M* matrix or an *N*-by-*M*-by-1-by-*K* array. *K* represents the number of spectrograms, and *N*-by-*M* is the size of the spectrograms and depends on the value of the **Spectrum type** parameter.

- `Mel (128 bands)` –– The network accepts mel spectrograms of size 128-by-199, where 128 is the number of mel bands, and 199 is the number of time hops.
- `Mel (256 bands)` –– The network accepts mel spectrograms of size 256-by-199, where 256 is the number of mel bands, and 199 is the number of time hops.
- `Linear` –– The network accepts positive one-sided spectrograms of size 257-by-197, where 257 is the FFT length and 197 is the number of time hops.

Data Types: `single` | `double`

Output

Port_1 — Embeddings
matrix

Output embeddings, returned as a *K*-by-*L* matrix, where *K* is the number of input spectrograms, and *L* is specified by the **Embedding length** parameter.

Data Types: `single`

Parameters

Spectrum type — Type of spectrum

`Mel (128 bands)` (default) | `Mel (256 bands)` | `Linear`

Type of spectrum generated from audio and used as input to the neural network, specified as `Mel (128 bands)`, `Mel (256 bands)`, or `Linear`. This parameter specifies the size of the network input “Port_1” on page 5-0 .

Content type — Type of audio content

`Environmental sounds` (default) | `Musical sounds`

Type of audio content the neural network was trained on, specified as `Environmental sounds` or `Musical sounds`. Set this parameter to `Environmental sounds` to use a neural network pretrained on environmental audio data, and set it to `Musical sounds` to use a network pretrained on musical data.

Embedding length — Output embedding length

`512` (default) | `6144`

Length of output embedding, specified as `512` or `6144`.

Mini-batch size — Size of mini-batches

`128` (default) | positive integer

Size of mini-batches to use for prediction, specified as a positive integer. Larger mini-batch sizes require more memory but can lead to faster predictions.

Block Characteristics

Data Types	`double` | `single`
Direct Feedthrough	`no`
Multidimensional Signals	`no`
Variable-Size Signals	`no`
Zero-Crossing Detection	`no`

Version History

Introduced in R2022b

References

[1] Cramer, Jason, et al. "Look, Listen, and Learn More: Design Choices for Deep Audio Embeddings." In *ICASSP 2019 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, IEEE, 2019, pp. 3852-56. *DOI.org (Crossref)*, doi:/10.1109/ICASSP.2019.8682475.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

Usage notes and limitations:

- To generate generic C code that does not depend on third-party libraries, in the **Configuration Parameters** > **Code Generation** general category, set the **Language** parameter to `C`.
- To generate C++ code, in the **Configuration Parameters** > **Code Generation** general category, set the **Language** parameter to `C++`. To specify the target library for code generation, in the **Code Generation** > **Interface** category, set the **Target Library** parameter. Setting this parameter to `None` generates generic C++ code that does not depend on third-party libraries.
- For ERT-based targets, the **Support: variable-size signals** parameter in the **Code Generation**> **Interface** pane must be enabled.
- For a list of networks and layers supported for code generation, see “Networks and Layers Supported for Code Generation” (MATLAB Coder).

See Also

Blocks

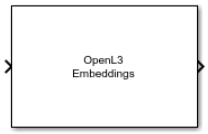
OpenL3 Embeddings | OpenL3 Preprocess | YAMNet | VGGish

Functions

`openl3` | `openl3Embeddings` | `openl3Preprocess` | `yamnet` | `vggish`

OpenL3 Embeddings

Extract OpenL3 embeddings



Libraries:
Audio Toolbox / Deep Learning

Description

The OpenL3 Embeddings block uses OpenL3 to extract feature embeddings from audio signals. The OpenL3 Embeddings block combines necessary audio preprocessing and OpenL3 network inference and returns feature embeddings that are a compact representation of audio data. This block requires Deep Learning Toolbox.

Ports

Input

Port_1 — Sound data
column vector

Sound data, specified as a one-channel signal (column vector). If **Sample rate of input signal (Hz)** is 48e3, there are no restrictions on the input frame length. If **Sample rate of input signal (Hz)** is different from 48e3, then the input frame length must be a multiple of the decimation factor of the resampling operation that the block performs. If the input frame length does not satisfy this condition, the block throws an error message with information on the decimation factor.

Data Types: `single` | `double`

Output

Port_1 — Embedding
row vector

Output embedding, returned as a row vector whose length is specified by the **Embedding length** parameter.

Data Types: `single`

Parameters

Sample rate of input signal (Hz) — Sample rate of input signal in Hz

`48e3` (default) | positive scalar

Sample rate of the input signal in Hz, specified as a positive scalar.

Overlap percentage (%) — Overlap percentage between consecutive spectrograms

`90` (default) | [0 100)

Specify the overlap percentage between consecutive spectrograms as a scalar in the range [0 100).

Spectrum type — Type of spectrum

`Mel (128 bands)` (default) | `Mel (256 bands)` | `Linear`

Type of spectrum generated from audio and used as input to the neural network, specified as `Mel (128 bands)`, `Mel (256 bands)`, or `Linear`.

- `Mel (128 bands)` –– The neural network accepts mel spectrograms generated from the input audio with 128 mel bands.
- `Mel (256 bands)` –– The neural network accepts mel spectrograms generated from the input audio with 256 mel bands.
- `Linear` –– The neural network accepts positive one-sided spectrograms generated from the input audio with an FFT length of 257.

Content type — Type of audio content

`Environmental sounds` (default) | `Musical sounds`

Type of audio content the neural network was trained on, specified as `Environmental sounds` or `Musical sounds`. Set this parameter to `Environmental sounds` to use a neural network pretrained on environmental audio data, and set it to `Musical sounds` to use a network pretrained on musical data.

Embedding length — Output embedding length

`512` (default) | `6144`

Length of output embedding, specified as `512` or `6144`.

Block Characteristics

Data Types	`double` | `single`
Direct Feedthrough	no
Multidimensional Signals	no
Variable-Size Signals	no
Zero-Crossing Detection	no

Version History

Introduced in R2022b

References

[1] Cramer, Jason, et al. "Look, Listen, and Learn More: Design Choices for Deep Audio Embeddings." In *ICASSP 2019 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, IEEE, 2019, pp. 3852-56. *DOI.org (Crossref)*, doi:/10.1109/ICASSP.2019.8682475.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

Usage notes and limitations:

- To generate generic C code that does not depend on third-party libraries, in the **Configuration Parameters > Code Generation** general category, set the **Language** parameter to `C`.
- To generate C++ code, in the **Configuration Parameters > Code Generation** general category, set the **Language** parameter to `C++`. To specify the target library for code generation, in the **Code Generation > Interface** category, set the **Target Library** parameter. Setting this parameter to `None` generates generic C++ code that does not depend on third-party libraries.
- For ERT-based targets, the **Support: variable-size signals** parameter in the **Code Generation> Interface** pane must be enabled.
- For a list of networks and layers supported for code generation, see “Networks and Layers Supported for Code Generation” (MATLAB Coder).

See Also

Blocks

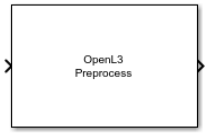
OpenL3 | OpenL3 Preprocess | VGGish Embeddings | Sound Classifier

Functions

`openl3` | `openl3Embeddings` | `openl3Preprocess` | `vggishEmbeddings` | `classifySound`

OpenL3 Preprocess

Preprocess audio for OpenL3 embeddings extraction



Libraries:
Audio Toolbox / Deep Learning

Description

The OpenL3 Preprocess block generates spectrograms from an audio input. You can then feed these spectrograms to an OpenL3 pretrained network or to a network that accepts the same inputs as OpenL3.

Ports

Input

Port_1 — Sound data
column vector

Sound data, specified as a one-channel signal (column vector). If **Sample rate of input signal (Hz)** is 48e3, there are no restrictions on the input frame length. If **Sample rate of input signal (Hz)** is different from 48e3, then the input frame length must be a multiple of the decimation factor of the resampling operation that the block performs. If the input frame length does not satisfy this condition, the block throws an error message with information on the decimation factor.

Data Types: `single` | `double`

Output

Port_1 — Spectrogram
matrix

Spectrogram generated from input audio, returned as a matrix whose size depends on the value of the **Spectrum type** parameter.

- `Mel (128 bands)` –– The block returns a mel spectrogram of size 128-by-199, where 128 is the number of mel bands, and 199 is the number of time hops.
- `Mel (256 bands)` –– The block returns a mel spectrogram of size 256-by-199, where 256 is the number of mel bands, and 199 is the number of time hops.
- `Linear` –– The block returns a positive one-sided spectrogram of size 257-by-197, where 257 is the FFT length and 197 is the number of time hops.

You can use this spectrogram as input to an OpenL3 block that has the same **Spectrum type**.

Data Types: `single`

Parameters

Sample rate of input signal (Hz) — Sample rate of input signal in Hz

`48e3` (default) | positive scalar

Sample rate of the input signal in Hz, specified as a positive scalar.

Overlap percentage (%) — Overlap percentage between consecutive spectrograms

`90` (default) | [0 100)

Specify the overlap percentage between consecutive spectrograms as a scalar in the range [0 100).

Spectrum type — Type of spectrum

`Mel (128 bands)` (default) | `Mel (256 bands)` | `Linear`

Type of spectrum generated from input audio, specified as `Mel (128 bands)`, `Mel (256 bands)`, or `Linear`.

Block Characteristics

Data Types	`double` | `single`
Direct Feedthrough	no
Multidimensional Signals	no
Variable-Size Signals	no
Zero-Crossing Detection	no

Version History

Introduced in R2022b

References

[1] Cramer, Jason, et al. "Look, Listen, and Learn More: Design Choices for Deep Audio Embeddings." In *ICASSP 2019 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, IEEE, 2019, pp. 3852-56. *DOI.org (Crossref)*, doi:/10.1109/ICASSP.2019.8682475.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

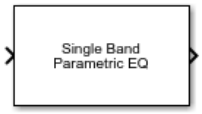
See Also

Blocks
OpenL3 | OpenL3 Embeddings | YAMNet | VGGish

Functions
`openl3` | `openl3Embeddings` | `openl3Preprocess` | `vggish` | `yamnet`

Single-Band Parametric EQ

Second-order parametric equalizer filter



Libraries:
Audio Toolbox / Filters

Description

The Single-Band Parametric EQ 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 is designed using `designParamEQ` and implemented using `dsp.BiquadFilter`.

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

x — Input signal
matrix | 1-D vector

- Matrix input –– Each column of the input is treated as an independent channel.
- 1-D vector input –– The input is treated as a signal channel.

This port is unnamed unless you specify additional input ports.

Data Types: `single` | `double`

Fc — Center frequency (Hz)
scalar

Specify the center frequency as a positive scalar that is less than half the sample rate of the input signal.

Dependencies

To enable this port, select **Specify from input port** for the **Center Frequency (Hz)** parameter.

Data Types: `single` | `double`

BW — Bandwidth (Hz)
scalar

Specify the filter bandwidth as a positive scalar that is less than or equal to half the sample rate of the input signal and 20 kHz.

Dependencies

To enable this port, select `Bandwidth and Center Frequency` for the **Filter specification** and **Specify from input port** for the **Filter Bandwidth (Hz)** parameter.

Data Types: `single` | `double`

GdB — Peak or dip gain (dB)
scalar

Specify the peak or dip gain in dB as a scalar.

Dependencies

To enable this port, select **Specify from input port** for the **Peak Gain (dB)** parameter.

Data Types: `single` | `double`

Q — Quality factor
scalar

Specify the quality factor as a positive scalar.

Dependencies

To enable this port, select `Quality factor and center frequency` for the **Filter Specification** and **Specify from input port** for the **Quality Factor** parameter.

Data Types: `single` | `double`

Output

Port_1 — Output signal
matrix

The Single-Band Parametric EQ block outputs a signal with the same data type as the input signal. The size of the output depends on the size of the input:

- Matrix input –– The block outputs a matrix the same size and data type as the input signal.
- 1-D vector input –– The block outputs an *N*-by-1 matrix (column vector), where *N* is the number of elements in the 1-D vector.

Data Types: `single` | `double`

Parameters

If a parameter is listed as tunable, then you can change its value during simulation.

Filter order — Order of filter
2 (default) | positive even scalar

Tunable: No

Filter specification — Specify parameters used to design filter
`Bandwidth and center frequency` (default) | `Quality factor and center frequency`

- `Bandwidth and center frequency` –– Design the filter using **Filter Bandwidth (Hz)**, **Center Frequency (Hz)**, and **Peak Gain (dB)**.
- `Quality factor and center frequency` –– Design the filter using **Center Frequency (Hz)**, **Peak Gain (dB)**, and **Quality Factor**.

Tunable: No

Center Frequency (Hz) — Center frequency of 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.

To specify **Center Frequency (Hz)** from an input port, select **Specify from input port** for the parameter.

Tunable: Yes

Filter Bandwidth (Hz) — Bandwidth of filter
`2205` (default) | positive scalar in the range [0, max(fs/2, 20,000)]

Specify the filter bandwidth as a positive scalar that is less than or equal to half the sample rate of the input signal or 20 kHz, whichever is larger.

To specify **Filter Bandwidth (Hz)** from an input port, select **Specify from input port** for the parameter.

Tunable: Yes

Dependencies

To enable this parameter, set **Filter specification** to `Bandwidth and center frequency`.

Quality Factor — Quality factor
`5` (default) | scalar in the range [0.1, 20]

Specify the quality factor as a scalar in the range [0.1, 20].

To specify **Quality Factor** from an input port, select **Specify from input port** for the parameter.

Tunable: Yes

Dependencies

To enable this parameter, set **Filter specification** to `Quality factor and center frequency`.

Peak Gain (dB) — Peak or dip gain of filter
`6.0206` (default) | scalar in the range [−30, 30]

Specify the peak gain in dB as a scalar in the range [−30, 30].

Tunable: Yes

Inherit sample rate from input — Specify source of input sample rate
on (default) | off

When you select this parameter, the block inherits its sample rate from the input signal. When you clear this parameter, you specify the sample rate in **Input sample rate (Hz)**.

Tunable: No

Input sample rate (Hz) — Sample rate of input
`44100` (default) | scalar

Tunable: Yes

Dependencies

To enable this parameter, clear the **Inherit sample rate from input** parameter.

Simulate using — Specify type of simulation to run
`Code generation` (default) | `Interpreted execution`

- `Interpreted execution` –– Simulate model using the MATLAB interpreter. This option shortens startup time and has simulation speed comparable to `Code generation`. In this mode, you can debug the source code of the block.
- `Code generation` –– Simulate model using generated C code. The first time you run a simulation, Simulink generates C code for the block. The C code is reused for subsequent simulations, as long as the model does not change. This option requires additional startup time, but the speed of the subsequent simulations is faster compared to `Interpreted execution`.

Tunable: No

Block Characteristics

Data Types	`double` | `single`
Direct Feedthrough	`no`
Multidimensional Signals	`no`
Variable-Size Signals	`yes`
Zero-Crossing Detection	`no`

Version History

Introduced in R2019a

R2022b: Input sample rate (Hz) parameter is tunable

Behavior changed in R2022b

You can change the value of the **Input sample rate (Hz)** parameter during simulation.

References

[1] Orfanidis, Sophocles J. "High-Order Digital Parametric Equalizer Design." *Journal of the Audio Engineering Society*. Vol. 53, November 2005, pp. 1026–1046.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

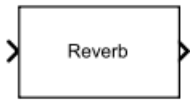
`multibandParametricEQ` | `designParamEQ` | `designVarSlopeFilter` | `designShelvingEQ`

Topics

“Parametric Equalizer Design”
“Equalization”

Reverberator

Add reverberation to audio signal



Libraries:
Audio 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

x — Input signal
matrix | 1-D vector

- Matrix input –– Each column of the input is treated as an independent channel.
- 1-D vector input –– The input is treated as a single channel.

This port is unnamed unless you specify additional input ports.

Data Types: `single` | `double`

Delay — Pre-delay for reverberation (s)
scalar in the range [0,1]

Dependencies

To enable this port, select **Specify from input port** for the “Pre-delay (s)” on page 5-0 parameter.

Data Types: `single` | `double`

HighCut — Lowpass filter cutoff
positive scalar in the range [0, (Sample Rate)/2]

Dependencies

To enable this port, select **Specify from input port** for the “Highcut frequency (Hz)” on page 5-0 parameter.

Data Types: `single` | `double`

Diffusion — Density of reverb tail
scalar in the range [0,1]

Dependencies

To enable this port, select **Specify from input port** for the “Diffusion” on page 5-0 parameter.

Data Types: `single` | `double`

Decay — Decay factor of reverb tail
scalar in the range [0,1]

Dependencies

To enable this port, select **Specify from input port** for the “Decay factor” on page 5-0 parameter.

Data Types: `single` | `double`

Damping — High-frequency damping
scalar in the range [0,1]

Dependencies

To enable this port, select **Specify from input port** for the “High frequency damping” on page 5-0 parameter.

Data Types: `single` | `double`

WetDry — Ratio of wet (reverberated) signal to dry (original) signal
scalar in the range [0,1]

Dependencies

To enable this port, select **Specify from input port** for the “Wet/dry mix” on page 5-0 parameter.

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 of 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, 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.

To specify **Pre-delay (s)** from an input port, select **Specify from input port** for the parameter.

Tunable: Yes

Highcut frequency (Hz) — Lowpass filter cutoff
`20000` (default) | scalar in the range [0, (Sample Rate)/2]

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.

To specify **Highcut frequency (Hz)** from an input port, select **Specify from input port** for the parameter.

Tunable: Yes

Diffusion — Density of reverb tail
`0.50` (default) | scalar in the range [0, 1]

Diffusion is proportional to the rate at which the reverb tail builds in density. Increasing **Diffusion** pushes the reflections closer together, thickening the sound. Reducing **Diffusion** creates more discrete echoes.

To specify **Diffusion** from an input port, select **Specify from input port** for the parameter.

Tunable: Yes

Decay factor — Decay factor of reverb tail
`0.50` (default) | scalar in the range [0, 1]

Decay factor is inversely 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).

To specify **Decay factor** from an input port, select **Specify from input port** for the parameter.

Tunable: Yes

High frequency damping — High-frequency damping
`0.0005` (default) | scalar in the range [0, 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.

To specify **High frequency damping** from an input port, select **Specify from input port** for the parameter.

Tunable: Yes

Wet/dry mix — Ratio of wet (reverberated) signal to dry (original) signal
`0.3` (default) | scalar in the range [0, 1]

Wet/dry mix is the ratio of wet (reverberated) signal to dry (original) signal that your Reverberator block outputs.

To specify **Wet/dry mix** from an input port, select **Specify from input port** for the parameter.

Tunable: Yes

Inherit sample rate from input — Specify source of input sample rate
on (default) | off

When you select this parameter, the block inherits its sample rate from the input signal. When you clear this parameter, you specify the sample rate in **Input sample rate (Hz)**.

Input sample rate (Hz) — Sample rate of input
`44100` (default) | positive scalar

Tunable: Yes

Dependencies

To enable this parameter, clear the **Inherit sample rate from input** parameter.

Simulate using — Specify type of simulation to run
`Interpreted execution` (default) | `Code generation`

- `Interpreted execution` – Simulate the model using the MATLAB interpreter. This option reduces startup time and the simulation speed is comparable to `Code generation`. In this mode, you can debug the source code of the block.
- `Code generation` – Simulate the model using generated C code. The first time you run a simulation, Simulink generates C code for the block. The C code is reused for subsequent simulations, as long as the model does not change. This option requires additional startup time, but the speed of the subsequent simulations is comparable to `Interpreted execution`.

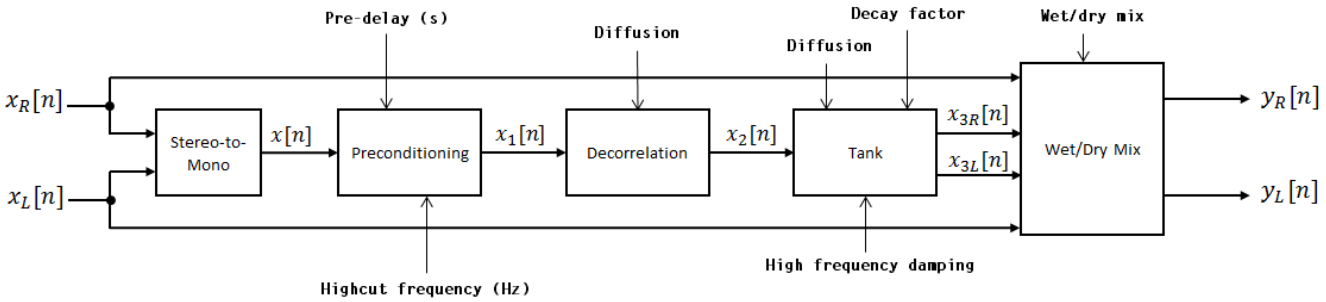
Block Characteristics

Data Types	`double` | `single`
Direct Feedthrough	`no`
Multidimensional Signals	`no`
Variable-Size Signals	`yes`
Zero-Crossing Detection	`no`

Algorithms

The algorithm to add reverberation follows the plate-class reverberation topology described in [1] and is based on a 29,761 Hz sample rate.

The algorithm has five stages.



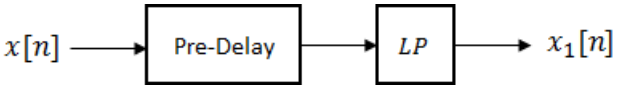
The description for the algorithm that follows is for a stereo input. A mono input is a simplified case.

Stereo-to-Mono

A stereo signal is converted to a mono signal: $x[n] = 0.5 \times (x_R[n] + x_L[n])$.

Preconditioning

A delay followed by a lowpass filter preconditions the mono signal.



- The pre-delay output is determined as $x_p[n] = x[n - k]$, where the **Pre-delay (s)** parameter determines the value of k.
- The signal is fed through a single-pole lowpass filter with transfer function

  $$LP(z) = \frac{1 - \alpha}{1 - \alpha z^{-1}},$$

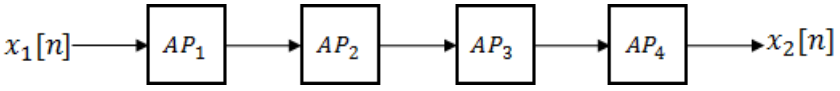
  where

  $$\alpha = \exp\left(-2\pi \times \frac{f_c}{f_s}\right).$$

  - f_c is the cutoff frequency specified by the **Pre-delay (s)** parameter.
  - f_s is the sampling frequency specified by the **Inherit sample rate from input** parameter or the **Input sample rate (Hz)** parameter.

Decorrelation

The signal is decorrelated by passing through a series of four allpass filters.



The allpass filters are of the form

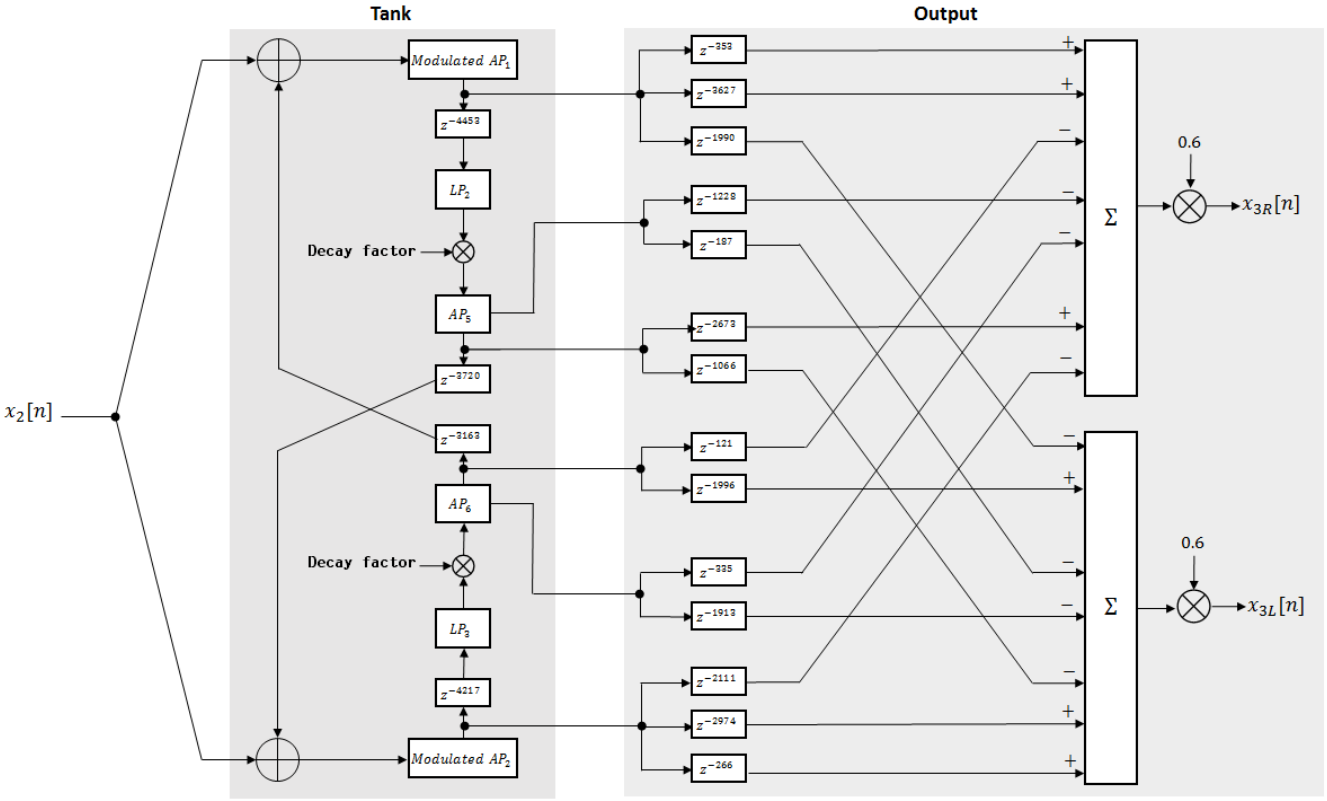
$$AP(z) = \frac{\beta + z^{-k}}{1 + \beta z^{-k}},$$

where β is the coefficient specified by the `Diffusion` property and k is the delay as follows:

- For AP_1, k = `142`.
- For AP_2, k = `107`.
- For AP_3, k = `379`.
- For AP_4, k = `277`.

Tank

The signal is fed into the tank, where it circulates to simulate the decay of a reverberation tail.



The following description tracks the signal as it progresses through the top of the tank. The signal progression through the bottom of the tank follows the same pattern, with different delay specifications.

1. The new signal enters the top of the tank and is added to the circulated signal from the bottom of the tank.
2. The signal passes through a modulated allpass filter:

$$Modulated\ AP_1(z) = \frac{-\beta + z^{-k}}{1 - \beta z^{-k}}$$

   - β is the coefficient specified by the **Diffusion** parameter.

- k is the variable delay specified by a 1 Hz sinusoid with amplitude = (8/29761) × (sample rate). To account for fractional delay resulting from the modulating k, allpass interpolation is used [2].

3 The signal is delayed again, and then passes through a lowpass filter:

$$LP_2(z) = \frac{1-\varphi}{1-\varphi z^{-1}}$$

- φ is the coefficient specified by the **High frequency damping** parameter.

4 The signal is multiplied by a gain specified by the **Decay factor** parameter. The signal then passes through an allpass filter:

$$AP_5(z) = \frac{\beta + z^{-k}}{1+\beta z^{-k}} \ .$$

- β is the coefficient specified by the **Diffusion** parameter.
- k is set to `1800` for the top of the tank and `2656` for the bottom of the tank.

5 The signal is delayed again and then circulated to the bottom half of the tank for the next iteration.

A similar pattern is executed in parallel for the bottom half of the tank. The output of the tank is calculated as the signed sum of delay lines picked off at various points from the tank. The summed output is multiplied by `0.6`.

Wet/Dry Mix

The wet (processed) signal is then added to the dry (original) signal:

$$y_R[n] = (1-\kappa)x_R[n] + \kappa x_{3R}[n]\,,$$

$$y_L[n] = (1-\kappa)x_L[n] + \kappa x_{3L}[n]\,,$$

where the **Wet/dry mix** parameter determines κ.

Version History

Introduced in R2016a

References

[1] Dattorro, Jon. "Effect Design, Part 1: Reverberator and Other Filters." *Journal of the Audio Engineering Society*. Vol. 45, Issue 9, 1997, pp. 660–684.

[2] Dattorro, Jon. "Effect Design, Part 2: Delay-Line Modulation and Chorus." *Journal of the Audio Engineering Society*. Vol. 45, Issue 10, 1997, pp. 764–788.

Extended Capabilities

C/C++ Code Generation

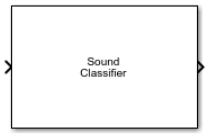
Generate C and C++ code using Simulink® Coder™.

See Also

`reverberator`

Sound Classifier

Classify sounds in audio signal



Libraries:
Audio Toolbox / Deep Learning

Description

The Sound Classifier block uses YAMNet to classify audio segments into sound classes described by the AudioSet ontology. The Sound Classifier block combines necessary audio preprocessing and YAMNet network inference. The block returns predicted sound labels, predicted scores from the sounds, and class labels for predicted scores.

Ports

Input

audioIn — Sound data
column vector

Sound data to classify, specified as a one-channel signal (column vector). If **Sample rate of input signal (Hz)** is 16e3, there are no restrictions on the input frame length. If **Sample rate of input signal (Hz)** is different from 16e3, then the input frame length must be a multiple of the decimation factor of the resampling operation that the block performs. If the input frame length does not satisfy this condition, the block throws an error message with information on the decimation factor.

Data Types: `single` | `double`

Output

sound — Predicted sound label
enumerated scalar

Predicted sound label, returned as an enumerated scalar.

Data Types: `enumerated`

scores — Predicted activations or scores
vector

Predicted activation or score values for each supported sound label, returned as a 1-by-521 vector, where 521 is the number of classes in YAMNet.

Data Types: `single`

labels — Class labels for predicted scores
vector

Class labels for predicted scores, returned as a 1-by-521 vector.

Data Types: `enumerated`

Parameters

Sample rate of input signal (Hz) — Sample rate of input signal in Hz

`16e3` (default) | positive scalar

Specify the sample rate of the input signal as a positive scalar in Hz. If the sample rate is different from 16e3, then the block resamples the signal to 16e3, which is the sample rate that YAMNet supports.

Data Types: `single` | `double`

Overlap percentage (%) — Overlap percentage between consecutive mel spectrograms

`50` (default) | [0 100)

Specify the overlap percentage between consecutive mel spectrograms as a scalar in the range [0 100).

Data Types: `single` | `double`

Classification — Select to output sound classification

`on` (default) | `off`

Enable the output port **sound**, which outputs the classified sound.

Predictions — Output all scores and associated labels

`off` (default) | `on`

Enable the output ports **scores** and **labels**, which output all predicted scores and associated class labels.

Block Characteristics

Data Types	`double` | `single`
Direct Feedthrough	no
Multidimensional Signals	no
Variable-Size Signals	no
Zero-Crossing Detection	no

Algorithms

The Sound Classifier block algorithm consists of two steps:

1 Preprocessing –– YAMNet specific preprocessing. Generates mel spectrograms.

2. Prediction –– Predicting the sounds, scores, and labels of the input signal using the YAMNet sound classification network.

Preprocessing

1. Cast **audioIn** to single and resample to 16 kHz.
2. Compute the one-sided short-time Fourier transform (STFT) using a 25 ms periodic Hann window (400 samples) with a 10 ms hop (160 samples) and a 512-point DFT.
3. Convert the complex spectral values to magnitude and discard phase information.
4. Pass the one-sided magnitude STFTs through a 64-band mel-spaced filter bank. Doing so converts the 257-length STFT vectors to 64-length vectors in the mel scale.
5. Convert the 64-length vectors to a log scale.
6. Buffer the vectors into outputs of size 96-by-64, where 96 is the number of 10 ms frames in each mel spectrogram and 64 is the number of mel bands. The overlap between consecutive 96-by-64 mel spectrograms is determined by the value of the **Overlap percentage (%)** parameter.

Prediction

These 96-by-64 spectrograms are passed to the YAMNet block. The YAMNet block has a maximum of three outputs:

- **sound**: The label of the most likely sound. You get one "sound" for each 96-by-64 spectrogram input.
- **scores**: 1-by-512 vectors, with a score value for each supported sound label.
- **labels**: 1-by-521 vectors containing the sound labels.

Version History

Introduced in R2021b

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

Usage notes and limitations:

- To generate generic C code that does not depend on third-party libraries, in the **Configuration Parameters > Code Generation** general category, set the **Language** parameter to `C`.
- To generate C++ code, in the **Configuration Parameters > Code Generation** general category, set the **Language** parameter to `C++`. To specify the target library for code generation, in the **Code Generation > Interface** category, set the **Target Library** parameter. Setting this parameter to `None` generates generic C++ code that does not depend on third-party libraries.
- For ERT-based targets, the **Support: variable-size signals** parameter in the **Code Generation> Interface** pane must be enabled.
- For a list of networks and layers supported for code generation, see “Networks and Layers Supported for Code Generation” (MATLAB Coder).

See Also

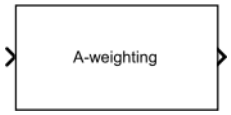
Apps
Signal Labeler

Blocks
YAMNet | YAMNet Preprocess

Functions
`classifySound` | `vggish` | `vggishFeatures` | `vggishPreprocess` | `yamnet` | `yamnetPreprocess` | `yamnetGraph`

Weighting Filter

Weighted frequency response filter



Libraries:
Audio 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 5-145, “C-Weighting” on page 5-146, and “K-Weighting” on page 5-146 for the definition of the weighting curves.

Tunable: No

Inherit sample rate from input — Specify source of input sample rate
`off` (default) | `on`

When you select this parameter, the block inherits its sample rate from the input signal. When you clear this parameter, you specify the sample rate in **Input sample rate (Hz)**.

Tunable: No

Input sample rate (Hz) — Sample rate of input
`44100` (default) | `positive scalar`

Tunable: Yes

Dependencies

To enable this parameter, clear the **Inherit sample rate from input** parameter.

Simulate using — Specify type of simulation to run
`Code generation` (default) | `Interpreted execution`

- `Code generation` –– Simulate model using generated C code. The first time you run a simulation, Simulink generates C code for the block. The C code is reused for subsequent simulations, as long as the model does not change. This option requires additional startup time but the speed of the subsequent simulations is faster than `Interpreted execution`.
- `Interpreted execution` –– Simulate model using the MATLAB interpreter. This option shortens startup time but has a slower simulation speed compared to `Code generation`. In this mode, you can debug the source code of the block.

Tunable: No

Mask for attenuation limits — Creates a mask for filter response visualization
`No mask` (default) | `Class 1` | `Class 2`

The mask attenuation limits are defined in the IEC 61672-1:2002 standard.

- If the mask is green, the design is compliant.
- If the mask is red, the design breaks compliance.

Tunable: Yes

Dependencies

To enable this parameter, set **Weighting method** to `A-weighting` or `C-weighting`.

Visualize filter response — Open plot to visualize magnitude response and compliance mask
button

A 2048-point FFT is used to calculate the magnitude response.

Tunable: Yes

Variable name — Variable name of exported filter
`myFilt` (default) | valid variable name

Name of the variable in the base workspace to contain the filter when it is exported. The name must be a valid MATLAB variable name.

Overwrite variable if it already exists — Overwrite variable if it already exists
`on` (default) | `off`

When you select this parameter, exporting the filter overwrites the variable specified by the **Variable name** parameter if it already exists in the base workspace. If you do not select this parameter and the specified variable already exists in the workspace, exporting the filter creates a new variable with an underscore and a number appended to the variable name. For example, if the variable name is `var` and it already exists, the exported variable will be named `var_1`.

Export filter to workspace — Export filter to workspace
button

Export the filter to the base workspace in the variable specified by the **Variable name** parameter.

Tips

You cannot export the filter if you have enabled the **Inherit sample rate from input** parameter and the model is not running.

Block Characteristics

Data Types	`double` | `single`
Direct Feedthrough	`no`
Multidimensional Signals	`no`
Variable-Size Signals	`yes`
Zero-Crossing Detection	`no`

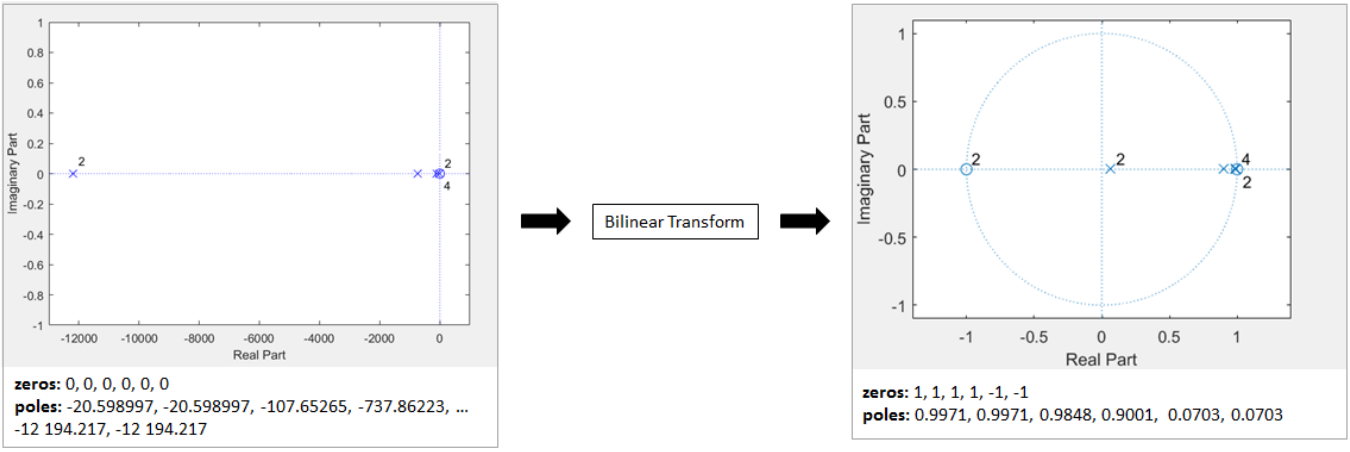
More About

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 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

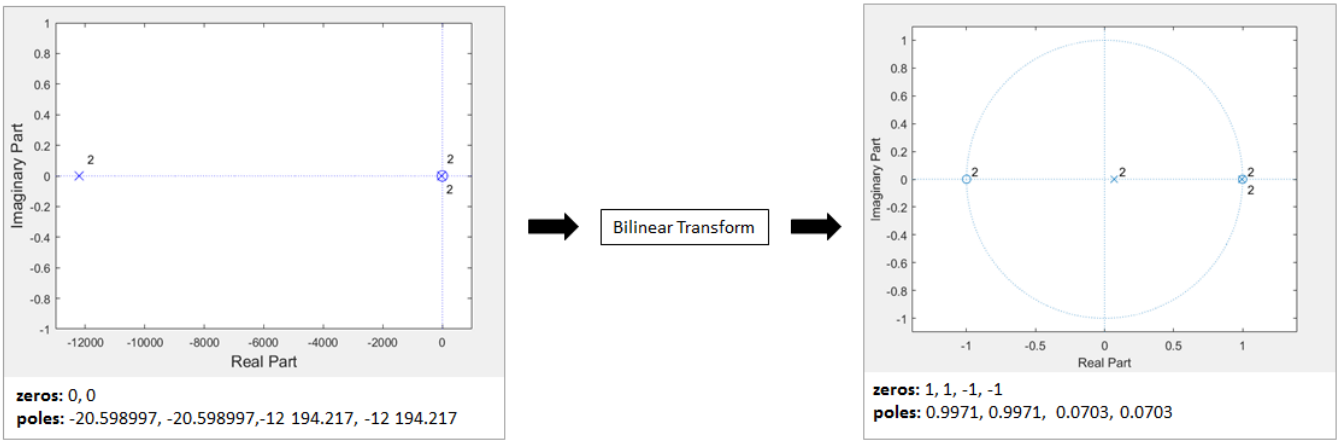
zeros: 1, 1, 1, 1, -1, -1
poles: 0.9971, 0.9971, 0.9848, 0.9001, 0.0703, 0.0703

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 sounds intended for A-weighting filters.

The ANSI S1.42-2001 [1] defines the C-weighting curve. The IEC 61672-1:2002 [2] standard defines the minimum and maximum attenuation limits for C-weighting filters.

ANSI S1.42.2001 defines the weighting curve by specifying analog poles and zeros. Audio 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

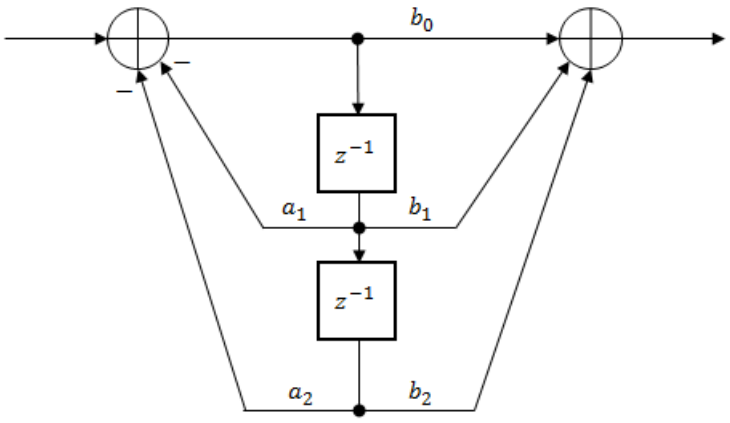
zeros: 1, 1, -1, -1
poles: 0.9971, 0.9971, 0.0703, 0.0703

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].

Version History

Introduced in R2016b

References

[1] Acoustical Society of America. *Design Response of Weighting Networks for Acoustical Measurements*. ANSI S1.42-2001. New York, NY: American National Standards Institute, 2001.

[2] International Electrotechnical Commission. *Electroacoustics Sound Level Meters Part 1: Specifications*. First Edition. IEC 61672-1. 2002-2005.

[3] International Telecommunication Union. *Algorithms to measure audio programme loudness and true-peak audio level*. ITU-R BS.1770-4. 2015.

[4] Mansbridge, Stuart, Saoirse Finn, and Joshua D. Reiss. "Implementation and Evaluation of Autonomous Multi-track Fader Control." Paper presented at the 132nd Audio Engineering Society Convention, Budapest, Hungary, 2012.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

`weightingFilter` | `octaveFilter` | `loudnessMeter` | Octave Filter | Loudness Meter

Shelving Filter

Second-order IIR shelving filter

Libraries:
Audio Toolbox / Filters

Description

The Shelving Filter block applies a shelving filter to the input signal. A shelving filter boosts or cuts the frequency spectrum of the input signal above or below a given cutoff frequency.

Ports

Input

x — Input signal
column vector | matrix

Input signal to be filtered, specified as a column vector or a matrix. If the input is a matrix, each column is treated as an independent channel.

Data Types: `single` | `double`

G — Peak gain (dB)
real scalar

This port specifies the value of the **Gain (dB)** parameter.

Dependencies

To enable this port, select the **Specify gain from input port** parameter.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

S — Slope
positive scalar

This port specifies the value of the **Slope** parameter.

Dependencies

To enable this port, select the **Specify slope from input port** parameter.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

F — Cutoff frequency
nonnegative scalar

This port specifies the value of the **Cutoff frequency (Hz)** parameter.

Dependencies

To enable this port, select the **Specify cutoff frequency from input port** parameter.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Output

Port_1 — Output signal
column vector | matrix

Filtered output signal, returned as a column vector or matrix that is the same size and data type as the input signal.

Data Types: `single` | `double`

Parameters

Specify gain from input port — Use additional input port to specify gain

`off` (default) | `on`

When you select this parameter, an additional input port, **G**, is added to the block. This port specifies the gain of the filter.

Gain (dB) — Peak gain (dB)

`0` (default) | real scalar

Peak gain of the filter in dB, specified as a real scalar. The gain specifies how much the filter will boost (if the gain is positive) or cut (if the gain is negative) the frequency spectrum of the input signal.

Tunable: Yes

Dependencies

To enable this parameter, clear the **Specify gain from input port** parameter.

Specify slope from input port — Use additional input port to specify slope

`off` (default) | `on`

When you select this parameter, an additional input port, **S**, is added to the block. This port specifies the slope of the filter.

Slope — Filter slope

`1.5` (default) | positive scalar

Slope of the filter specified as a positive scalar. The slope controls the width of the transition band in the filter response.

Tunable: Yes

Dependencies

To enable this parameter, clear the **Specify slope from input port** parameter.

Specify cutoff frequency from input port — Use additional input port to specify cutoff frequency

`off` (default) | `on`

When you select this parameter, an additional input port, **F**, is added to the block. This port specifies the cutoff frequency of the filter.

Cutoff frequency (Hz) — Filter cutoff frequency

`200` (default) | nonnegative scalar

Cutoff frequency of the filter in Hz, specified as a nonnegative scalar in the range [0,`Fs`/2], where `Fs` is the sample rate specified by the **Input sample rate (Hz)** and **Inherit sample rate from input** parameters. The cutoff frequency specifies the frequency at half of the peak gain of the filter, `G`/2 dB, where `G` is the peak gain.

Tunable: Yes

Dependencies

To enable this parameter, clear the **Specify cutoff frequency from input port** parameter.

Type — Type of filter

`lowpass` (default) | `highpass`

Type of shelving filter, specified as `lowpass` or `highpass`.

- `lowpass` –– Boost or cut the frequency spectrum below the cutoff frequency.
- `highpass` –– Boost or cut the frequency spectrum above the cutoff frequency.

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

Sample rate of the input, specified as a positive scalar.

Dependencies

To enable this parameter, clear the **Inherit sample rate from input** parameter.

Visualize filter response — Open plot to visualize filter response

button

Open plot to visualize the magnitude response of the filter.

Variable name — Variable name of exported filter

`myFilt` (default) | valid variable name

Name of the variable in the base workspace to contain the filter when it is exported. The name must be a valid MATLAB variable name.

Overwrite variable if it already exists — Overwrite variable if it already exists

`on` (default) | `off`

When you select this parameter, exporting the filter overwrites the variable specified by the **Variable name** parameter if it already exists in the base workspace. If you do not select this parameter and the specified variable already exists in the workspace, exporting the filter creates a new variable with an underscore and a number appended to the variable name. For example, if the variable name is `var` and it already exists, the exported variable will be named `var_1`.

Export filter to workspace — Export filter to workspace

button

Export the filter to the base workspace in the variable specified by the **Variable name** parameter.

Tips

- You cannot export the filter if you have enabled the **Inherit sample rate from input** parameter and the model is not running.
- You cannot export the filter if you are specifying filter characteristics from input ports.

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 but has a slower simulation speed than `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`.

Block Characteristics

Data Types	`double` | `single`
Direct Feedthrough	`no`
Multidimensional Signals	`no`
Variable-Size Signals	`yes`
Zero-Crossing Detection	`no`

Version History

Introduced in R2022a

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

Blocks

Graphic EQ | Multiband Parametric EQ

Functions

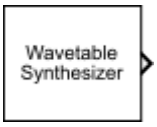
`designParamEQ` | `designShelvingEQ` | `designVarSlopeFilter`

Objects

`shelvingFilter` | `dsp.SOSFilter` | `graphicEQ` | `multibandParametricEQ`

Wavetable Synthesizer

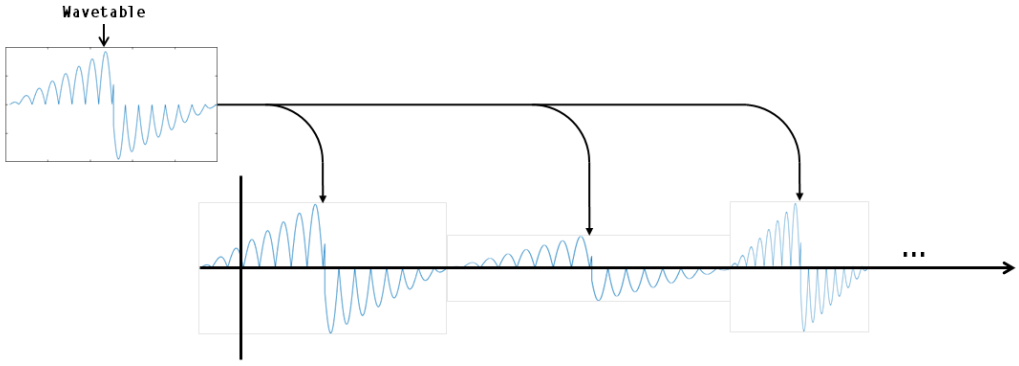
Generate periodic signal from single-cycle waveforms



Libraries:
Audio Toolbox / Sources

Description

The Wavetable Synthesizer block generates a periodic signal with tunable parameters. The periodic signal is defined by a single-cycle waveform cached as the **Single-cycle waveform** parameter of your Wavetable Synthesizer block.



Ports

Input

WT — Single-cycle waveform
vector of real values

Dependencies

To enable this port, select **Specify wavetable from input port** for the “Single-cycle waveform” on page 5-0 parameter.

Data Types: `single` | `double`

F — Output wave frequency (Hz)
nonnegative scalar | vector of nonnegative values

Dependencies

To enable this port, select **Specify frequency from input port** for the “Output wave frequency (Hz)” on page 5-0 parameter.

Data Types: `single` | `double`

A — Output wave amplitude
nonnegative scalar

Dependencies

To enable this port, select **Specify amplitude from input port** for the "Output wave amplitude" on page 5-0 parameter.

Data Types: `single` | `double`

DC — Output wave DC offset
scalar

Dependencies

To enable this port, select **Specify DC offset from input port** for the "Output wave DC offset" on page 5-0 parameter.

Data Types: `single` | `double`

Output

Port_1 — Output signal
vector | matrix

The Wavetable Synthesizer block outputs a periodic signal defined by the parameters of the block.

Data Types: `single` | `double`

Parameters

If a parameter is listed as tunable, then you can change its value during simulation.

Single-cycle waveform — Wavetable
`sin(2*pi*(0:511)/512)` (default) | vector of real values

The Wavetable Synthesizer block indexes into the single-cycle waveform to synthesize a periodic wave.

To specify **Single-cycle waveform** from an input port, select **Specify wavetable from input port** for the parameter.

Tunable: Yes

Output wave frequency (Hz) — Frequency of generated signal
`100` (default) | nonnegative scalar

The number of times the single-cycle waveform is repeated in one second.

To specify **Output wave frequency (Hz)** from an input port, select **Specify frequency from input port** for the parameter.

Tunable: Yes

Output wave amplitude — Amplitude of generated signal
`1` (default) | nonnegative scalar

Amplitude scaling is applied before DC offset.

To specify **Output wave amplitude** from an input port, select **Specify amplitude from input port** for the parameter.

Tunable: Yes

Output wave phase offset — Normalized phase offset of generated signal
`0` (default) | scalar in the range `[0,1]`

The phase offset range, `[0,1]`, corresponds to a normalized 2π radians interval.

Output wave DC offset — Value added to each element of generated signal
`0` (default) | scalar

To specify **Output wave DC offset** from an input port, select **Specify DC offset from input port** for the parameter.

Tunable: Yes

Samples per frame — Number of samples per frame output from block
`512` (default) | positive integer scalar

Number of samples per frame output from block, specified as a positive integer scalar.

Tunable: No

Sample rate (Hz) — Sample rate of generated signal
`44100` (default) | positive scalar

Sample rate of generated signal, specified as a positive scalar.

Tunable: No

Output data type — Data type of generated signal
`double` (default) | `single`

Data type of generated signal, specified as `double` or `single`.

Tunable: No

Simulate using — Specify type of simulation to run
`Code generation` (default) | `Interpreted execution`

- `Code generation` – Simulate the model using generated C code. The first time you run a simulation, Simulink generates C code for the block. The C code is reused for subsequent simulations, as long as the model does not change. This option requires additional startup time, but the speed of the subsequent simulations is comparable to `Interpreted execution`.
- `Interpreted execution` – Simulate the model using the MATLAB interpreter. This option reduces startup time and the simulation speed is comparable to `Code generation`. In this mode, you can debug the source code of the block.

Block Characteristics

Data Types	`double` | `single`
Direct Feedthrough	`no`
Multidimensional Signals	`no`
Variable-Size Signals	`no`
Zero-Crossing Detection	`no`

Version History

Introduced in R2020a

Extended Capabilities

C/C++ Code Generation

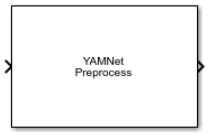
Generate C and C++ code using Simulink® Coder™.

See Also

Audio Device Writer | Audio Oscillator | `wavetableSynthesizer`

YAMNet Preprocess

Preprocess audio for YAMNet classification



Libraries:
Audio Toolbox / Deep Learning

Description

The YAMNet Preprocess block generates mel spectrograms from audio input that can be fed to the YAMNet pretrained network or to a network that accepts the same inputs as YAMNet.

Ports

Input

audioIn — Sound data
column vector

Sound data to classify, specified as a one-channel signal (column vector). If **Sample rate of input signal (Hz)** is 16e3, there are no restrictions on the input frame length. If **Sample rate of input signal (Hz)** is different from 16e3, then the input frame length must be a multiple of the decimation factor of the resampling operation that the block performs. If the input frame length does not satisfy this condition, the block throws an error message with information on the decimation factor.

Data Types: `single` | `double`

Output

features — Mel spectrograms that can be fed to YAMNet pretrained network
96-by-64 matrix

Mel spectrograms generated from **audioIn**, returned as a 96-by-64 matrix, where:

- 96 –– Represents the number of 25 ms frames in each mel spectrogram
- 64 –– Represents the number of mel bands spanning 125 Hz to 7.5 kHz

The overlap between consecutive 96-by-64 mel spectrograms is determined by the value of the **Overlap percentage (%)** parameter.

Each 96-by-64 matrix represents a single mel spectrogram. For more details on how this block generates mel spectrograms, see “Algorithms” on page 5-159.

Data Types: `single`

Parameters

Sample rate of input signal (Hz) — Sample rate of input signal in Hz

`16e3` (default) | positive scalar

Sample rate of the input signal in Hz, specified as a positive scalar.

Data Types: `single` | `double`

Overlap percentage (%) — Overlap percentage between consecutive mel spectrograms

`50` (default) | [0 100)

Specify the overlap percentage between consecutive mel spectrograms as a scalar in the range [0 100).

Data Types: `single` | `double`

Block Characteristics

Data Types	`double` | `single`
Direct Feedthrough	`no`
Multidimensional Signals	`no`
Variable-Size Signals	`no`
Zero-Crossing Detection	`no`

Algorithms

The YAMNet Preprocess block generates mel spectrograms from audio input. These mel spectrograms can be fed to a YAMNet pretrained network or to a network that accepts the same inputs as YAMNet.

Preprocessing steps

1. Cast **audioIn** to single and resample to 16 kHz.
2. Compute one-sided short-time Fourier transform using a 25 ms periodic Hann window (400 samples) with a 10 ms hop (160 samples) and a 512-point DFT.
3. Convert the complex spectral values to magnitude and discard phase information.
4. Pass the one-sided magnitude STFTs through a 64-band mel-spaced filter bank. Doing so converts the 257-length STFT vectors to 64-length vectors in the mel scale.
5. Convert the 64-length vectors to a log scale.
6. Buffer the vectors into outputs of size 96-by-64, where 96 is the number of spectra in the mel spectrogram and 64 is the number of mel bands. The overlap between consecutive 96-by-64 mel spectrograms is determined by the value of the **Overlap percentage (%)** parameter.

Version History

Introduced in R2021b

References

[1] Gemmeke, Jort F., Daniel P. W. Ellis, Dylan Freedman, Aren Jansen, Wade Lawrence, R. Channing Moore, Manoj Plakal, and Marvin Ritter. “Audio Set: An Ontology and Human-Labeled Dataset for Audio Events.” *2017 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, IEEE, 2017, pp. 776–80. *DOI.org (Crossref)*, doi:10.1109/ICASSP.2017.7952261.

[2] Hershey, Shawn, Sourish Chaudhuri, Daniel P. W. Ellis, Jort F. Gemmeke, Aren Jansen, R. Channing Moore, Manoj Plakal, et al. “CNN Architectures for Large-Scale Audio Classification.” *2017 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, IEEE, 2017, pp. 131–35. *DOI.org (Crossref)*, doi:10.1109/ICASSP.2017.7952132.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

Apps

Signal Labeler

Blocks

Sound Classifier | VGGish Embeddings | VGGish Preprocess | VGGish | YAMNet

Functions

`classifySound` | `vggish` | `vggishEmbeddings` | `vggishPreprocess` | `yamnet` | `yamnetPreprocess` | `yamnetGraph`

YAMNet

YAMNet sound classification network

Libraries:
Audio Toolbox / Deep Learning

Description

The YAMNet block leverages a pretrained sound classification network that is trained on the AudioSet dataset to predict audio events from the AudioSet ontology.

Ports

Input

features — Mel spectrograms
96-by-64 matrix | 96-by-64-by-1-by-*N* array

Mel spectrograms, specified as a 96-by-64 matrix or a 96-by-64-by-1-by-*N* array, where:

- 96 –– Represents the number of 25 ms frames in each mel spectrogram
- 64 –– Represents the number of mel bands spanning 125 Hz to 7.5 kHz
- *N* –– Number of channels.

You can use the YAMNet Preprocess block to generate mel spectrograms. The dimensions of these spectrograms are 96-by-64.

Data Types: `single` | `double`

Output

sound — Predicted sound label
enumerated scalar

Predicted sound label, returned as an enumerated scalar.

Data Types: `enumerated`

scores — Predicted activations or scores
vector

Predicted activation or score values for each supported sound label, returned as a 1-by-521 vector, where 521 is the number of classes in YAMNet.

Data Types: `single`

labels — Class labels for predicted scores
vector

Class labels for predicted scores, returned as a 1-by-521 vector.

Data Types: `enumerated`

Parameters

Mini-batch size — Size of mini-batches

`128` (default) | positive integer

Size of mini-batches to use for prediction, specified as a positive integer. Larger mini-batch sizes require more memory, but can lead to faster predictions.

Data Types: `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Classification — Select to output sound classification

`on` (default) | `off`

Enable the output port **sound**, which outputs the classified sound.

Predictions — Output all scores and associated labels

`off` (default) | `on`

Enable the output ports **scores** and **labels**, which output all predicted scores and associated class labels.

Block Characteristics

Data Types	`double` | `single`
Direct Feedthrough	`no`
Multidimensional Signals	`no`
Variable-Size Signals	`no`
Zero-Crossing Detection	`no`

Algorithms

Prediction

The block accepts mel spectrograms of size 96-by-64 or 96-by-64-by-1-by-*N*, and computes a maximum of three outputs using these spectrograms:

- **sound**: The label of the most likely sound. You get one "sound" for each 96-by-64 spectrogram input.
- **scores**: 1-by-512 vectors. Each element in the vector is a score value for each supported sound label.
- **labels**: 1-by-521 vectors. Each element in the vector is a sound label.

Version History

Introduced in R2021b

References

[1] Gemmeke, Jort F., Daniel P. W. Ellis, Dylan Freedman, Aren Jansen, Wade Lawrence, R. Channing Moore, Manoj Plakal, and Marvin Ritter. “Audio Set: An Ontology and Human-Labeled Dataset for Audio Events.” *2017 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, IEEE, 2017, pp. 776–80. *DOI.org (Crossref)*, doi:10.1109/ICASSP.2017.7952261.

[2] Hershey, Shawn, Sourish Chaudhuri, Daniel P. W. Ellis, Jort F. Gemmeke, Aren Jansen, R. Channing Moore, Manoj Plakal, et al. “CNN Architectures for Large-Scale Audio Classification.” *2017 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, IEEE, 2017, pp. 131–35. *DOI.org (Crossref)*, doi:10.1109/ICASSP.2017.7952132.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

Usage notes and limitations:

- To generate generic C code that does not depend on third-party libraries, in the **Configuration Parameters > Code Generation** general category, set the **Language** parameter to `C`.
- To generate C++ code, in the **Configuration Parameters > Code Generation** general category, set the **Language** parameter to `C++`. To specify the target library for code generation, in the **Code Generation > Interface** category, set the **Target Library** parameter. Setting this parameter to `None` generates generic C++ code that does not depend on third-party libraries.
- For ERT-based targets, the **Support: variable-size signals** parameter in the **Code Generation> Interface** pane must be enabled.
- For a list of networks and layers supported for code generation, see “Networks and Layers Supported for Code Generation” (MATLAB Coder).

See Also

Apps

Signal Labeler

Blocks

Sound Classifier | VGGish Embeddings | VGGish Preprocess | VGGish | YAMNet Preprocess

Functions

`classifySound` | `vggish` | `vggishEmbeddings` | `vggishPreprocess` | `yamnet` | `yamnetPreprocess` | `yamnetGraph`

VGGish Embeddings

Extract VGGish embeddings

Libraries:
Audio Toolbox / Deep Learning

Description

The VGGish Embeddings block uses VGGish to extract feature embeddings from audio segments. The VGGish Embeddings block combines necessary audio preprocessing and VGGish network inference and returns feature embeddings that are a compact representation of audio data.

Ports

Input

Port_1 — Sound data
column vector

Sound data, specified as a one-channel signal (column vector). If **Sample rate of input signal (Hz)** is 16e3, there are no restrictions on the input frame length. If **Sample rate of input signal (Hz)** is different from 16e3, then the input frame length must be a multiple of the decimation factor of the resampling operation that the block performs. If the input frame length does not satisfy this condition, the block throws an error message with information on the decimation factor.

Data Types: `single` | `double`

Output

Port_1 — Embeddings
row vector of length 128

VGGish feature embeddings, returned as a row vector of length 128. The feature embeddings are a compact representation of audio data.

Data Types: `single`

Parameters

Sample rate of input signal (Hz) — Sample rate of input signal in Hz
`16e3` (default) | positive scalar

Sample rate of the input signal in Hz, specified as a positive scalar.

Overlap percentage (%) — Overlap percentage between consecutive mel spectrograms
`50` (default) | [0 100)

Specify the overlap percentage between consecutive mel spectrograms as a scalar in the range [0 100).

Block Characteristics

Data Types	`double` | `single`
Direct Feedthrough	`no`
Multidimensional Signals	`no`
Variable-Size Signals	`no`
Zero-Crossing Detection	`no`

Algorithms

Preprocessing Steps

The VGGish Embeddings block preprocesses the audio data using the following steps to be in the format required by the VGGish network.

1. Cast the audio data to single precision and resample to 16 kHz.
2. Compute one-sided short-time Fourier transform using a 25 ms periodic Hann window (400 samples) with a 10 ms hop (160 samples) and a 512-point DFT.
3. Convert the complex spectral values to magnitude and discard phase information.
4. Pass the one-sided magnitude STFTs through a 64-band mel-spaced filter bank. Doing so converts the 257-length STFT vectors to 64-length vectors in the mel scale.
5. Convert the 64-length vectors to a log scale.
6. Buffer the vectors into outputs of size 96-by-64, where 96 is the number of spectra in the mel spectrogram and 64 is the number of mel bands. The overlap between consecutive 96-by-64 mel spectrograms is determined by the value of the **Overlap percentage (%)** parameter.

Version History

Introduced in R2022a

References

[1] Gemmeke, Jort F., Daniel P. W. Ellis, Dylan Freedman, Aren Jansen, Wade Lawrence, R. Channing Moore, Manoj Plakal, and Marvin Ritter. “Audio Set: An Ontology and Human-Labeled Dataset for Audio Events.” In *2017 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 776–80. New Orleans, LA: IEEE, 2017. https://doi.org/10.1109/ICASSP.2017.7952261.

[2] Hershey, Shawn, Sourish Chaudhuri, Daniel P. W. Ellis, Jort F. Gemmeke, Aren Jansen, R. Channing Moore, Manoj Plakal, et al. “CNN Architectures for Large-Scale Audio Classification.” In *2017 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 131–35. New Orleans, LA: IEEE, 2017. https://doi.org/10.1109/ICASSP.2017.7952132.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

Usage notes and limitations:

- To generate generic C code that does not depend on third-party libraries, in the **Configuration Parameters > Code Generation** general category, set the **Language** parameter to `C`.
- To generate C++ code, in the **Configuration Parameters > Code Generation** general category, set the **Language** parameter to `C++`. To specify the target library for code generation, in the **Code Generation > Interface** category, set the **Target Library** parameter. Setting this parameter to `None` generates generic C++ code that does not depend on third-party libraries.
- For ERT-based targets, the **Support: variable-size signals** parameter in the **Code Generation> Interface** pane must be enabled.
- For a list of networks and layers supported for code generation, see “Networks and Layers Supported for Code Generation” (MATLAB Coder).

See Also

Apps

Signal Labeler

Blocks

Sound Classifier | VGGish | VGGish Preprocess | YAMNet | YAMNet Preprocess

Functions

`classifySound` | `vggish` | `vggishFeatures` | `vggishPreprocess` | `yamnet` | `yamnetPreprocess` | `yamnetGraph`

VGGish Preprocess

Preprocess audio for VGGish feature extraction

Libraries:
Audio Toolbox / Deep Learning

Description

The VGGish Preprocess block generates mel spectrograms from an audio input that you can then feed to the VGGish pretrained network or to a network that accepts the same inputs as VGGish.

Ports

Input

Port_1 — Sound data
column vector

Sound data, specified as a one-channel signal (column vector). If **Sample rate of input signal (Hz)** is 16e3, there are no restrictions on the input frame length. If **Sample rate of input signal (Hz)** is different from 16e3, then the input frame length must be a multiple of the decimation factor of the resampling operation that the block performs. If the input frame length does not satisfy this condition, the block throws an error message with information on the decimation factor.

Data Types: `single` | `double`

Output

Port_1 — Mel spectrogram
96-by-64 matrix

Mel spectrogram generated from the input audio signal, returned as a 96-by-64 matrix, where:

- 96 –– Represents the number of 25 ms frames in each mel spectrogram
- 64 –– Represents the number of mel bands spanning 125 Hz to 7.5 kHz

The overlap between consecutive 96-by-64 mel spectrograms is determined by the value of the **Overlap percentage (%)** parameter. You can provide the mel spectrogram as an input to the VGGish pretrained network or to a network that accepts the same inputs as VGGish.

Data Types: `single`

Parameters

Sample rate of input signal (Hz) — Sample rate of input signal in Hz
`16e3` (default) | positive scalar

Sample rate of the input signal in Hz, specified as a positive scalar.

Overlap percentage (%) — Overlap percentage between consecutive mel spectrograms
`50` (default) | [0 100)

Specify the overlap percentage between consecutive mel spectrograms as a scalar in the range [0 100).

Block Characteristics

Data Types	`double` | `single`
Direct Feedthrough	`no`
Multidimensional Signals	`no`
Variable-Size Signals	`no`
Zero-Crossing Detection	`no`

Algorithms

Preprocessing Steps

The VGGish Embeddings block preprocesses the audio data using the following steps to be in the format required by the VGGish network.

1. Cast the audio data to single precision and resample to 16 kHz.
2. Compute one-sided short-time Fourier transform using a 25 ms periodic Hann window (400 samples) with a 10 ms hop (160 samples) and a 512-point DFT.
3. Convert the complex spectral values to magnitude and discard phase information.
4. Pass the one-sided magnitude STFTs through a 64-band mel-spaced filter bank. Doing so converts the 257-length STFT vectors to 64-length vectors in the mel scale.
5. Convert the 64-length vectors to a log scale.
6. Buffer the vectors into outputs of size 96-by-64, where 96 is the number of spectra in the mel spectrogram and 64 is the number of mel bands. The overlap between consecutive 96-by-64 mel spectrograms is determined by the value of the **Overlap percentage (%)** parameter.

Version History

Introduced in R2022a

References

[1] Gemmeke, Jort F., Daniel P. W. Ellis, Dylan Freedman, Aren Jansen, Wade Lawrence, R. Channing Moore, Manoj Plakal, and Marvin Ritter. “Audio Set: An Ontology and Human-Labeled Dataset for Audio Events.” In *2017 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 776–80. New Orleans, LA: IEEE, 2017. https://doi.org/10.1109/ICASSP.2017.7952261.

[2] Hershey, Shawn, Sourish Chaudhuri, Daniel P. W. Ellis, Jort F. Gemmeke, Aren Jansen, R. Channing Moore, Manoj Plakal, et al. “CNN Architectures for Large-Scale Audio Classification.” In *2017 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 131–35. New Orleans, LA: IEEE, 2017. https://doi.org/10.1109/ICASSP.2017.7952132.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

Apps

Signal Labeler

Blocks

Sound Classifier | VGGish | VGGish Embeddings | YAMNet | YAMNet Preprocess

Functions

`classifySound` | `vggish` | `vggishFeatures` | `vggishPreprocess` | `yamnet` | `yamnetPreprocess` | `yamnetGraph`

VGGish

VGGish embeddings extraction network

Libraries:
Audio Toolbox / Deep Learning

Description

The VGGish block leverages a pretrained convolutional neural network that is trained on the AudioSet data set to extract feature embeddings from audio signals.

Ports

Input

Port_1 — Mel spectrograms
96-by-64 matrix | 96-by-64-by-1-by-*N* array

Mel spectrograms, specified as a 96-by-64 matrix or a 96-by-64-by-1-by-*N* array, where:

- 96 –– Represents the number of 25 ms frames in each mel spectrogram
- 64 –– Represents the number of mel bands spanning 125 Hz to 7.5 kHz
- *N* –– Represents the number of mel spectrograms.

You can use the VGGish Preprocess block to generate mel spectrograms. All spectrograms are of the dimension 96-by-64.

Data Types: `single` | `double`

Output

Port_1 — Embeddings
N-by-128 matrix

VGGish feature embeddings, returned as an *N*-by-128 matrix, where *N* is the number of mel spectrograms in the input. The feature embeddings are a compact representation of audio data.

Data Types: `single`

Parameters

Mini-batch size — Size of mini-batches
128 (default) | positive integer

Size of mini-batches to use for prediction specified as a positive integer. Larger mini-batch sizes require more memory but can lead to faster predictions.

Block Characteristics

Data Types	`double` | `single`
Direct Feedthrough	`no`
Multidimensional Signals	`no`
Variable-Size Signals	`no`
Zero-Crossing Detection	`no`

Version History

Introduced in R2022a

References

[1] Gemmeke, Jort F., Daniel P. W. Ellis, Dylan Freedman, Aren Jansen, Wade Lawrence, R. Channing Moore, Manoj Plakal, and Marvin Ritter. “Audio Set: An Ontology and Human-Labeled Dataset for Audio Events.” In *2017 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 776–80. New Orleans, LA: IEEE, 2017. https://doi.org/10.1109/ICASSP.2017.7952261.

[2] Hershey, Shawn, Sourish Chaudhuri, Daniel P. W. Ellis, Jort F. Gemmeke, Aren Jansen, R. Channing Moore, Manoj Plakal, et al. “CNN Architectures for Large-Scale Audio Classification.” In *2017 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 131–35. New Orleans, LA: IEEE, 2017. https://doi.org/10.1109/ICASSP.2017.7952132.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

Usage notes and limitations:

- To generate generic C code that does not depend on third-party libraries, in the **Configuration Parameters > Code Generation** general category, set the **Language** parameter to `C`.
- To generate C++ code, in the **Configuration Parameters > Code Generation** general category, set the **Language** parameter to `C++`. To specify the target library for code generation, in the **Code Generation > Interface** category, set the **Target Library** parameter. Setting this parameter to `None` generates generic C++ code that does not depend on third-party libraries.
- For ERT-based targets, the **Support: variable-size signals** parameter in the **Code Generation> Interface** pane must be enabled.
- For a list of networks and layers supported for code generation, see “Networks and Layers Supported for Code Generation” (MATLAB Coder).

See Also

Apps
Signal Labeler

Blocks
Sound Classifier | VGGish Preprocess | VGGish Embeddings | YAMNet | YAMNet Preprocess

Functions
`classifySound` | `vggish` | `vggishEmbeddings` | `vggishPreprocess` | `yamnet` | `yamnetPreprocess` | `yamnetGraph`

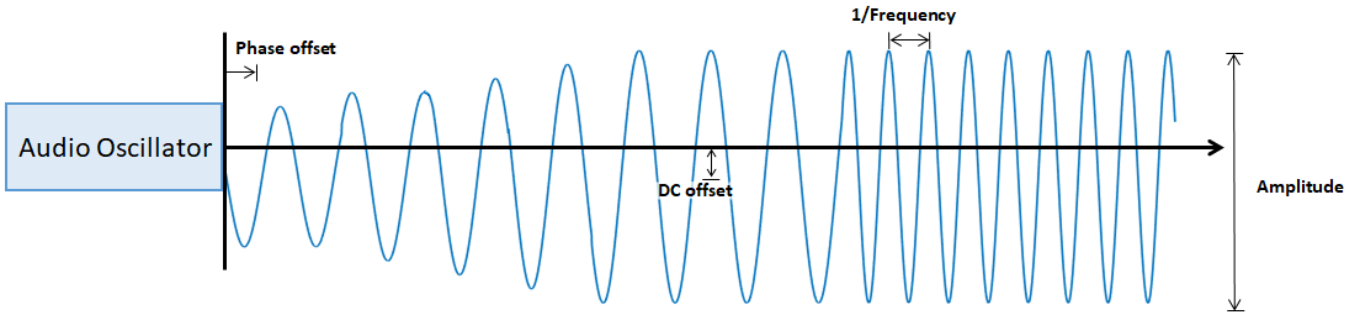
Audio Oscillator

Generate sine, square, and sawtooth waveforms

Libraries:
Audio Toolbox / Sources

Description

The Audio Oscillator block generates tunable waveforms. Typical uses include the generation of test signals for test benches, and the generation of control signals for audio effects. Parameters of the Audio Oscillator block specify the type of waveform generated.



Ports

Input

F — Frequency (Hz)
nonnegative scalar | vector of nonnegative values

Dependencies

To enable this port, select **Specify frequency from input port** for the "Frequency (Hz)" on page 5-0 parameter.

Data Types: `single` | `double`

A — Amplitude
nonnegative scalar | vector of nonnegative values

Dependencies

To enable this port, select **Specify amplitude from input port** for the "Amplitude" on page 5-0 parameter.

Data Types: `single` | `double`

DC — DC offset
scalar | vector

Dependencies

To enable this port, select **Specify DC offset from input port** for the "DC offset" on page 5-0 parameter.

Data Types: `single` | `double`

Output

Port_1 — Output signal
vector

The Audio Oscillator block outputs a periodic signal defined by the parameters of the block.

Data Types: `single` | `double`

Parameters

If a parameter is listed as tunable, then you can change its value during simulation.

Signal type — Type of generated waveform
`sine` (default) | `square` | `sawtooth`

The waveforms are generated using the algorithms specified by the `sin`, `square`, and `sawtooth` functions.

Frequency (Hz) — Frequency of generated waveform
`100` (default) | nonnegative scalar | vector of nonnegative values

- If "Signal type" on page 5-0 is set to `sine`, specify **Frequency (Hz)** as a scalar or as a vector. If **Frequency (Hz)** is set to an *N*-element vector, then the output from the block is the single-channel sum of *N* sinusoids. If **Frequency (Hz)** is set to an *N*-element vector, then "Amplitude" on page 5-0 , "Phase offset" on page 5-0 , and "DC offset" on page 5-0 must be scalars or *N*-element vectors.
- For square waveforms, specify **Frequency (Hz)** as a scalar.
- For sawtooth waveforms, specify **Frequency (Hz)** as a scalar.

To specify **Frequency (Hz)** from an input port, select **Specify frequency from input port**.

Tunable: Yes

Amplitude — Amplitude of generated waveform
`1` (default) | nonnegative scalar | vector of nonnegative values

- If "Signal type" on page 5-0 is set to `sine`, specify **Amplitude** as a scalar or as a vector. If **Amplitude** is set to an *N*-element vector, then the output from the block is the single-channel sum of *N* sinusoids. If **Amplitude** is set to an *N*-element vector, then "Frequency (Hz)" on page 5-0 , "Phase offset" on page 5-0 , and "DC offset" on page 5-0 must be scalars or *N*-element vectors.
- For square waveforms, specify **Amplitude** as a scalar.
- For sawtooth waveforms, specify **Amplitude** as a scalar.

To specify **Amplitude** from an input port, select **Specify amplitude from input port**.

Tunable: Yes

Phase offset — Normalized phase offset of generated waveform
`0` (default) | scalar in the range [0, 1] | vector with values in the range [0, 1]

The phase offset range, `[0,1]`, corresponds to a normalized 2π radians interval.

- If "Signal type" on page 5-0 is set to `sine`, specify **Phase offset** as a scalar or as a vector. If **Phase offset** is set to an *N*-element vector, then the output from the block is the single-channel sum of *N* sinusoids. If **Phase offset** is set to an *N*-element vector, then "Frequency (Hz)" on page 5-0 , "Amplitude" on page 5-0 , and "DC offset" on page 5-0 must be scalars or *N*-element vectors.
- For square waveforms, specify **Amplitude** as a scalar.
- For sawtooth waveforms, specify **Amplitude** as a scalar.

DC offset — Value added to each element of generated waveform
`0` (default) | scalar | vector

- If "Signal type" on page 5-0 is set to `sine`, specify **DC offset** as a scalar or as a vector. If **DC offset** is set to an *N*-element vector, then the output from the block is the single-channel sum of *N* sinusoids. If **DC offset** is set to an *N*-element vector, then "Frequency (Hz)" on page 5-0 , "Amplitude" on page 5-0 , and "Phase offset" on page 5-0 must be scalars or *N*-element vectors.
- For square waveforms, specify **Amplitude** as a scalar.
- For sawtooth waveforms, specify **Amplitude** as a scalar.

To specify **DC offset** from an input port, select **Specify DC offset from input port**.

Tunable: Yes

Duty cycle — Square waveform duty cycle
`0.5` (default) | scalar in the range [0, 1]

Square waveform duty cycle is the percentage of one period in which the waveform is above the median amplitude. A duty cycle value of 1 or 0 is equivalent to a DC signal.

Dependencies

To enable this parameter, set **Signal type** to `square`.

Width — Sawtooth width
`1` (default) | scalar in the range [0, 1]

Sawtooth width determines the point in a sawtooth waveform period at which the maximum occurs.

Dependencies

To enable this property, set **Signal type** to `sawtooth`.

Samples per frame — Number of samples per frame
`512` (default) | positive integer

Number of samples per frame, specified as a positive integer.

Sample rate (Hz) — Sample rate of generated waveform
`44100` (default) | positive scalar

The sample rate must be greater than twice the value specified in “Frequency (Hz)” on page 5-0 .

Output data type — Data type of generated waveform
`double` (default) | `single`

Data type of generated waveform, specified as `double` or `single`.

Tunable: No

Simulate using — Specify type of simulation to run
`Code generation` (default) | `Interpreted execution`

- `Code generation` – Simulate the model using generated C code. The first time you run a simulation, Simulink generates C code for the block. The C code is reused for subsequent simulations, as long as the model does not change. This option requires additional startup time, but the speed of the subsequent simulations is comparable to `Interpreted execution`.
- `Interpreted execution` – Simulate model using the MATLAB interpreter. This option reduces startup time and the simulation has speed comparable to `Code generation`. In this mode, you can debug the source code of the block.

Block Characteristics

Data Types	`double` | `single`
Direct Feedthrough	no
Multidimensional Signals	no
Variable-Size Signals	no
Zero-Crossing Detection	no

Version History

Introduced in R2020a

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

Wavetable Synthesizer | `audioOscillator` | Audio Device Writer

Multiband Parametric EQ

Multiband parametric equalizer

Libraries:
Audio Toolbox / Filters

Description

The Multiband Parametric EQ block performs multiband parametric equalization independently across each channel of input using specified center frequencies, gains, and quality factors. You can configure this block 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.

Ports

Input

Port_1 — Audio input to equalizer
matrix | vector

Audio input to the equalizer, specified as one of the following:

- Matrix input –– The block treats each column of the input as an independent channel.
- Vector input –– The block treats the input as having a single channel.

Data Types: `single` | `double`

Output

Port_1 — Audio output from equalizer
matrix

Audio output from the equalizer, returned as a matrix. If you specify the input as a matrix, the block returns the output with the same size as matrix input. If you specify the input as a vector then the output is N-by-1 matrix.

Data Types: `single` | `double`

Parameters

Main

EQ order — Order of individual equalizer bands
2 (default) | even positive integer

Order of individual equalizer bands, specified as an even positive integer. All equalizer bands have the same order.

Number of bands — Number of equalizer bands
3 (default) | integer in the range `[1,10]`

Number of equalizer bands, specified as an integer in the range `[1,10]`. The number of equalizer bands does not include shelving filters, highpass filters, or lowpass filters.

Specify frequencies from input port — Specify frequencies from input port
`off` (default) | `on`

Select this parameter to specify frequencies from the input port.

Frequencies (Hz) — Center frequencies of equalizer bands
`[100,181,325]` (default) | row vector of length equal to Number of bands

Center frequencies of equalizer bands in Hz, specified as a row vector of length equal to **Number of bands**. The vector consists of real scalars in the range `0` to **Input sample rate (Hz)**/2.

Tunable: Yes

Dependencies

To enable this parameter, set **Specify input frequencies from input port** to `off`.

Specify peak gains from input port — Specify peak gains from input port
`off` (default) | `on`

Select this parameter to specify peak gains from input port.

Peak Gains (dB) — Peak or dip filter gains
`[0,0,0]` (default) | row vector of length equal to Number of bands

Peak or dip filter gains, specified as a row vector of length equal to **Number of bands** in dB. The vector consists of real scalars in the range `[-inf,20]`.

Tunable: Yes

Dependencies

To enable this parameter, set **Specify peak gains from input port** to `off`.

Specify quality factors from input port — Specify quality factors from input port
`off` (default) | `on`

Select this parameter to specify quality factors from the input port.

Quality factors — Quality factors of equalizer bands
`[1.6,1.6,1.6]` (default) | row vector

Quality factors of equalizer bands, specified as a row vector of length equal to **Number of bands**.

Tunable: Yes

Dependencies

To enable this parameter, set **Specify quality factor from input port** to `off`.

Inherit sample rate from input — Specify sample rate from input port
`off` (default) | `on`

Select this parameter to specify sample rate from the input port.

Input sample rate (Hz) — Input sample rate
`44100` (default) | positive scalar

Input sample rate, specified as a positive scalar in Hz.

Tunable: Yes

Dependencies

To enable this parameter, set **Inherit sample rate from input port** to `off`.

Visualize filter response — Visualize magnitude response of equalizer
button

This button plots the filter responses of low-shelf, high-shelf, highpass (low-cut), and lowpass (high-cut) filters in a magnitude (dB) vs. frequencies (Hz) plot.

Note The block does not support filter visualization if any parameters are specified from input ports or the sample rate is inherited.

Variable name — Variable name of exported filter
`myFilt` (default) | valid variable name

Name of the variable in the base workspace to contain the filter when it is exported. The name must be a valid MATLAB variable name.

Overwrite variable if it already exists — Overwrite variable if it already exists
`on` (default) | `off`

When you select this parameter, exporting the filter overwrites the variable specified by the **Variable name** parameter if it already exists in the base workspace. If you do not select this parameter and the specified variable already exists in the workspace, exporting the filter creates a new variable with an underscore and a number appended to the variable name. For example, if the variable name is `var` and it already exists, the exported variable will be named `var_1`.

Export filter to workspace — Export filter to workspace
button

Export the filter to the base workspace in the variable specified by the **Variable name** parameter.

Tips

- You cannot export the filter if you have enabled the **Inherit sample rate from input** parameter and the model is not running.
- You cannot export the filter if you are specifying filter characteristics from input ports.

Advanced

Add low-shelf filter — Add low-shelf filter to equalizer
`off` (default) | `on`

Select this parameter to add a low-shelf filter to your equalizer.

Specify low-shelf cutoff from input port — Specify low-shelf cutoff frequency from input port
`off` (default) | `on`

Select this parameter to specify the cutoff frequency of the low-shelf filter from the input port.

Dependencies

To enable this parameter, set **Add low-shelf filter** to `on`.

Low-shelf cutoff frequency (Hz) — Low-shelf cutoff frequency
`200` (default) | scalar

The cutoff frequency of the low-shelf filter, specified as a scalar greater than or equal to `0` in Hz.

Tunable: Yes

Dependencies

To enable this parameter, set **Add low-shelf filter** to `on` and **Specify low-shelf cutoff from input port** to `off`.

Specify low-shelf slope from input port — Specify low-shelf slope
`off` (default) | `on`

Select this parameter to specify low-shelf slope from the input port.

Dependencies

To enable this parameter, set **Add low-shelf filter** to `on`.

Low-shelf filter slope — Low-shelf slope
`1.5` (default) | positive scalar

The slope of the low-shelf filter, specified as a positive scalar.

Tunable: Yes

Dependencies

To enable this parameter, set **Add low-shelf filter** to `on` and **Specify low-shelf slope from input port** to `off`.

Specify low-shelf gain from input port — Specify low-shelf gain from input port
`off` (default) | `on`

Select this parameter to specify the gain of the low-shelf filter from the input port.

Dependencies

To enable this parameter, set **Add low-shelf filter** to `on`.

Low-shelf filter gain (dB) — Low-shelf gain
`0` (default) | real scalar

The gain of the low-shelf filter, specified as a real scalar.

Tunable: Yes

Dependencies

To enable this parameter, set **Add low-shelf filter** to `on` and **Specify low-shelf gain from input port** to `off`.

Add high-shelf filter — Add high-shelf filter to equalizer
`off` (default) | `on`

Select this parameter to add a high-shelf filter to your equalizer.

Specify high-shelf cutoff from input port — Specify high-shelf cutoff frequency from input port
`off` (default) | `on`

Select this parameter to specify the cutoff frequency of the high-shelf filter from the input port.

Dependencies

To enable this parameter, set **Add high-shelf filter** to `on`.

High-shelf cutoff frequency (Hz) — High-shelf cutoff frequency
`15e3` (default) | scalar

The cutoff frequency of the high-shelf filter, specified as a scalar greater than or equal to `0` in Hz.

Tunable: Yes

Dependencies

To enable this parameter, set **Add high-shelf filter** to `on` and **Specify high-shelf cutoff from input port** to `off`.

Specify high-shelf slope from input port — Specify high-shelf slope from input port
`off` (default) | `on`

Select this parameter to specify the slope of the high-shelf filter from the input port.

Dependencies

To enable this parameter, set **Add high-shelf filter** to `on`.

High-shelf filter slope — High-shelf slope
`1.5` (default) | positive scalar

The slope of the high-shelf filter, specified as a positive scalar.

Tunable: Yes

Dependencies

To enable this parameter, set **Add high-shelf filter** to `on` and **Specify high-shelf slope from input port** to `off`.

Specify high-shelf gain from input port — Specify high-shelf gain from input port
`off` (default) | `on`

Select this parameter to specify the gain of the high-shelf filter from the input port.

Dependencies

To enable this parameter, set **Add high-shelf filter** to `on`.

High-shelf filter gain (dB) — High-shelf gain
`0` (default) | real scalar

The gain of the high-shelf filter, specified as a real scalar.

Tunable: Yes

Dependencies

To enable this parameter, set **Add high-shelf filter** to `on` and **Specify high-shelf gain from input port** to `off`.

Add lowpass filter — Add lowpass filter to equalizer
`off` (default) | `on`

Select this parameter to add a lowpass filter to your equalizer.

Specify lowpass cutoff from input port — Specify lowpass cutoff frequency from input port
`off` (default) | `on`

Select this parameter to specify the cutoff frequency of the lowpass filter from the input port.

Dependencies

To enable this parameter, set **Add lowpass filter** to `on`.

Lowpass cutoff frequency (Hz) — Lowpass cutoff frequency
`18e3` (default) | scalar

The cutoff frequency of the lowpass filter, specified as a scalar greater than or equal to `0` in Hz.

Tunable: Yes

Dependencies

To enable this parameter, set **Add lowpass filter** to `on` and **Specify lowpass cutoff from input port** to `off`.

Specify lowpass slope from input port — Specify lowpass slope from input port
`off` (default) | `on`

Select this parameter to specify the slope of the lowpass filter from the input port.

Dependencies

To enable this parameter, set **Add lowpass filter** to `on`.

Lowpass filter slope — Lowpass slope
`12` (default) | real scalar in the range `[0:6:48]`

The slope of the lowpass filter, specified as a real scalar in the range `[0:6:48]` in dB/octave. Values that are not multiples of 6 are rounded to the nearest multiple of 6.

Tunable: Yes

Dependencies

To enable this parameter, set **Add lowpass filter** to `on` and **Specify lowpass slope from input port** to `off`.

Add highpass filter — Add highpass filter to equalizer
`off` (default) | `on`

Select this parameter to add a highpass filter to your equalizer.

Specify highpass cutoff from input port — Specify highpass cutoff frequency from input port
`off` (default) | `on`

Select this parameter to specify the cutoff frequency of the highpass filter cutoff from the input port.

Dependencies

To enable this parameter, set **Add highpass filter** to `on`.

Highpass cutoff frequency (Hz) — Highpass cutoff frequency
`20` (default) | nonnegative real scalar

The cutoff frequency of the highpass filter, specified as a real scalar greater than or equal to `0` in Hz.

Tunable: Yes

Dependencies

To enable this parameter, set **Add highpass filter** to `on` and **Specify highpass cutoff from input port** to `off`.

Specify highpass slope from input port — Specify highpass slope from input port
`off` (default) | `on`

Select this parameter to specify the slope of the highpass filter from the input port.

Dependencies

To enable this parameter, set **Add highpass filter** to `on`.

Highpass filter slope — Highpass slope
`30` (default) | real scalar in the range `[0:6:48]`

The slope of the highpass filter, specified as a real scalar in the range `[0:6:48]` in dB/octave. Values that are not multiples of 6 are rounded to the nearest multiple of 6.

Tunable: Yes

Dependencies

To enable this parameter, set **Add highpass filter** to `on` and **Specify highpass slope from input port** to `off`.

Oversample — Oversample toggle
`off` (default) | `on`

Oversample toggle, specified as one of the following:

- `off` –– Run the multiband parametric equalizer at the input sample rate.
- `on` –– Run 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.

Simulate using — Specify type of simulation to run
`Code generation` (default) | `Interpreted execution`

Type of simulation to run, specified as one of the following:

- `Interpreted execution` –– Simulate model using the MATLAB interpreter. This option shortens startup time and has simulation speed comparable to `Code generation`. In this mode, you can debug the source code of the block.
- `Code generation` –– Simulate model using generated C code. The first time you run a simulation, Simulink generates C code for the block. The C code is reused for subsequent simulations as long as the model does not change. This option requires additional startup time but the speed of the subsequent simulations is faster than `Interpreted execution`.

Tunable: No

Block Characteristics

Data Types	`double` | `single`
Direct Feedthrough	`no`
Multidimensional Signals	`no`
Variable-Size Signals	`yes`
Zero-Crossing Detection	`no`

Version History

Introduced in R2021b

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

`multibandParametricEQ` | Gammatone Filter Bank | Single-Band Parametric EQ

Gammatone Filter Bank

Gammatone filter bank

Libraries:
Audio Toolbox / Filters

Description

The Gammatone Filter Bank block decomposes a signal by passing it through a bank of gammatone filters equally spaced on the equivalent rectangular bandwidth (ERB) scale. Gammatone filter banks are designed to model the human auditory system.

Ports

Input

Port_1 — Audio input to filter bank
scalar | vector | matrix

Audio input to the filter bank, specified as a scalar, vector, or matrix. If you specify the input as a matrix, the block treats the columns as independent audio channels. If you specify the input as a vector, the block treats the input as containing a single channel.

Data Types: `single` | `double`

Output

Port_1 — Audio output from filter bank
scalar | vector | matrix | 3-D array

Audio output from the filter bank, returned as a scalar, vector, matrix, or 3-D array. The shape of output signal depends on the shape of input signal and **Number of filters**. If input is an *M*-by-*N* matrix, then output is an *M*-by-Number of filters-by-*N* array. If *N* is 1, then output is a matrix.

Data Types: `single` | `double`

Parameters

Frequency range (Hz) — Frequency range of filter bank
`[50 8000]` (default) | two-element row vector of monotonically increasing values

Frequency range of the filter bank, specified as a two-element row vector of monotonically increasing values in Hz.

Tunable: No

Number of filters — Number of filters
32 (default) | positive integer

Number of filters in the filter bank, specified as a positive integer.

Tunable: No

Inherit sample rate from input — Specify sample rate from input port
`off` (default) | `on`

Select this parameter to specify the sample rate from the input port.

Input sample rate (Hz) — Input sample rate
`16000` (default) | positive integer

Input sample rate, specified as a positive integer in Hz.

Tunable: No

Dependencies

To enable this parameter, set **Inherit sample rate from input port** to `off`.

Bands as separate output port — Separate ports for each filter output
`off` (default) | `on`

Select this parameter to separate ports for each filter output.

Tunable: No

View Filter Response — Visualize filter bank responses
button

This button uses the `fvtool` function to visualize gammatone filter bank responses.

Variable name — Variable name of exported filter bank
`myFilt` (default) | valid variable name

Name of the variable in the base workspace to contain the filter bank when it is exported. The name must be a valid MATLAB variable name.

Overwrite variable if it already exists — Overwrite variable if it already exists
`on` (default) | `off`

When you select this parameter, exporting the filter bank overwrites the variable specified by the **Variable name** parameter if it already exists in the base workspace. If you do not select this parameter and the specified variable already exists in the workspace, exporting the filter bank creates a new variable with an underscore and a number appended to the variable name. For example, if the variable name is `var` and it already exists, the exported variable will be named `var_1`.

Export filter to workspace — Export filter bank to workspace
button

Export the filter bank to the base workspace in the variable specified by the **Variable name** parameter.

Tips

You cannot export the filter if you have enabled the **Inherit sample rate from input** parameter and the model is not running.

Simulate using — Specify type of simulation to run
`Interpreted execution` (default) | `Code generation`

Type of simulation to run, specified as one of the following:

- `Interpreted execution` –– Simulate model using the MATLAB interpreter. This option shortens startup time and has simulation speed comparable to `Code generation`. In this mode, you can debug the source code of the block.
- `Code generation` –– Simulate model using generated C code. The first time you run a simulation, Simulink generates C code for the block. The C code is reused for subsequent simulations as long as the model does not change. This option requires additional startup time but the speed of the subsequent simulations is faster than `Interpreted execution`.

Tunable: No

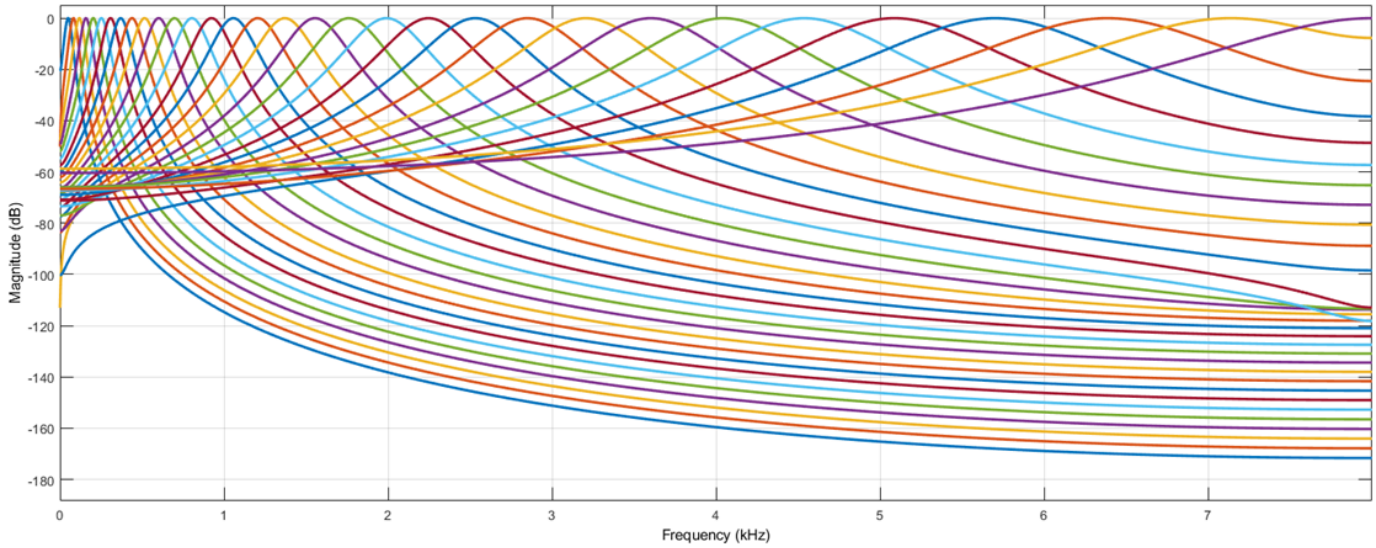
Block Characteristics

Data Types	`double` | `single`
Direct Feedthrough	`no`
Multidimensional Signals	`no`
Variable-Size Signals	`yes`
Zero-Crossing Detection	`no`

Algorithms

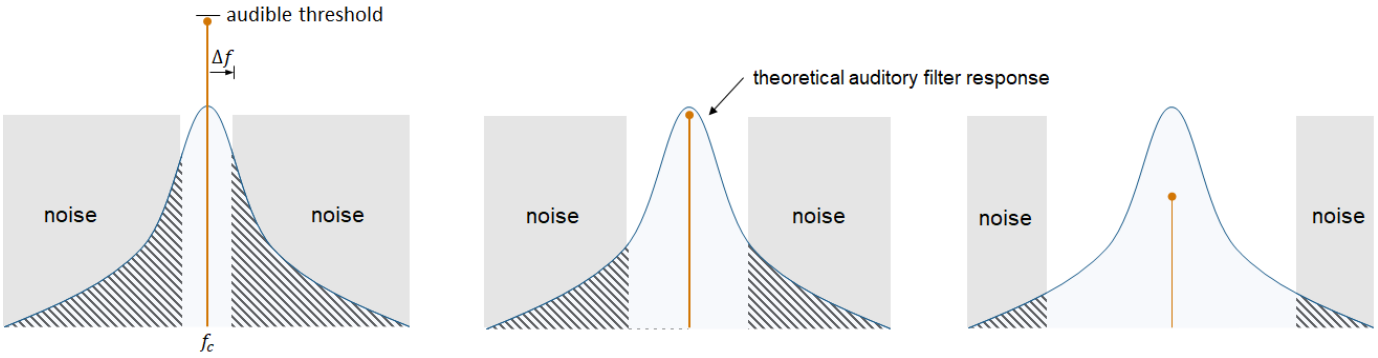
Applications

A gammatone filter bank is often used as the front end of a cochlea simulation. A cochlea simulation transforms complex sounds into a multichannel activity pattern like the one observed in the auditory nerve [2] .The Gammatone Filter Bank block follows the algorithm described in [1]. The algorithm is an implementation of an idea proposed in [2]. The design of the gammatone filter bank can be described in two parts: the filter shape (gammatone) and the frequency scale. The equivalent rectangular bandwidth (ERB) scale defines the relative spacing and bandwidth of the gammatone filters. The derivation of the ERB scale also provides an estimate of the auditory filter response that closely resembles the gammatone filter.

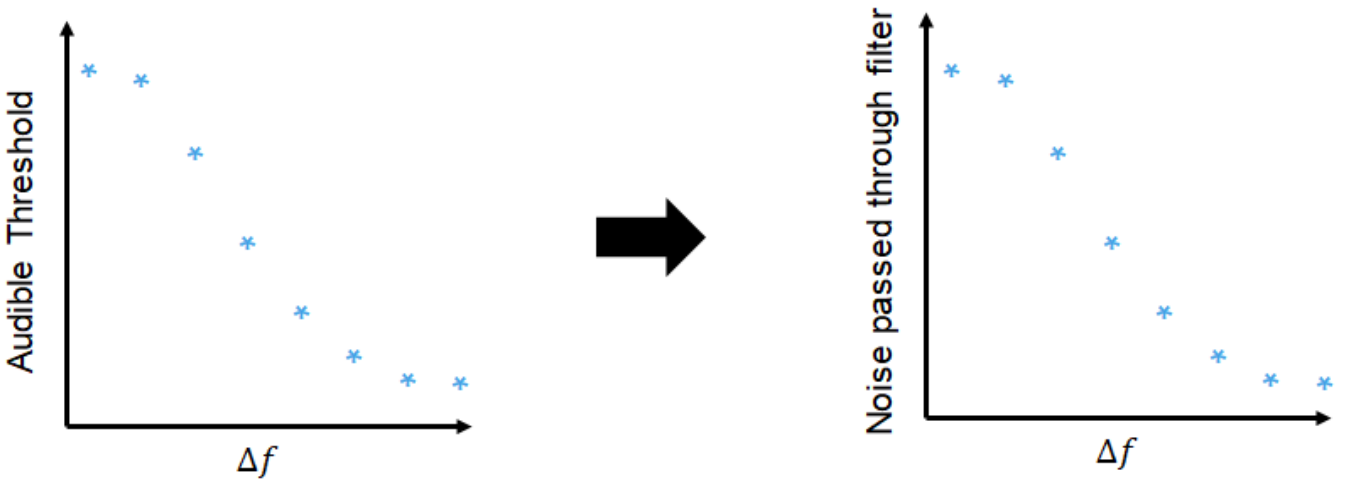


Frequency Scale

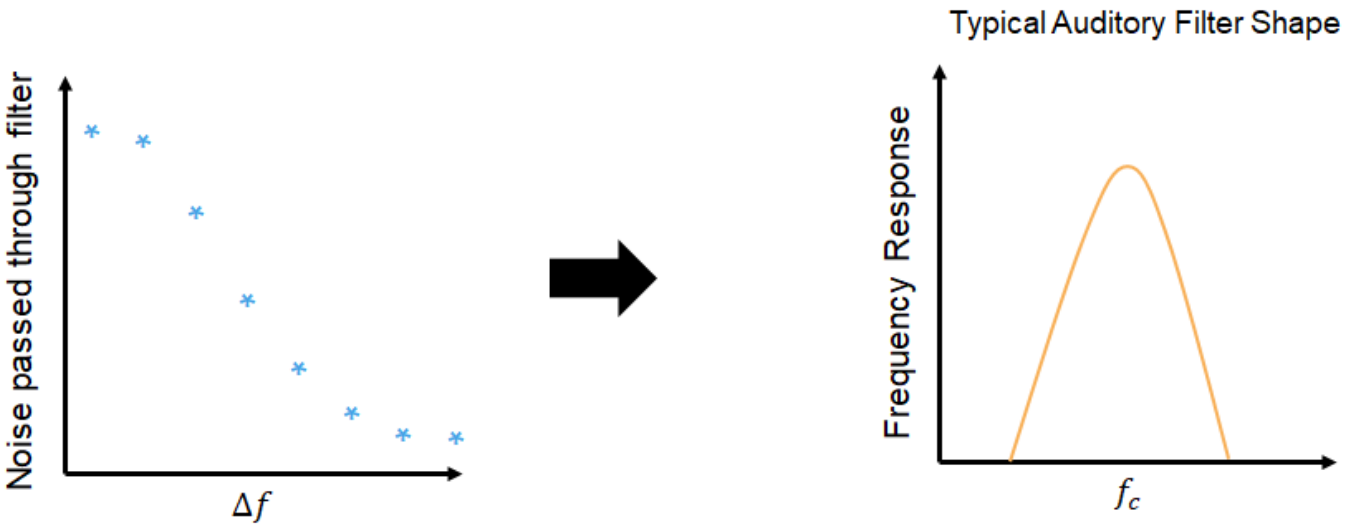
The block determines the ERB scale using the notched-noise masking method. This method involves a listening test wherein notched noise is centered on a tone. The power of the tone is tuned, and the audible threshold (the power required for the tone to be heard) is recorded. The experiment is repeated for different notch widths and center frequencies.



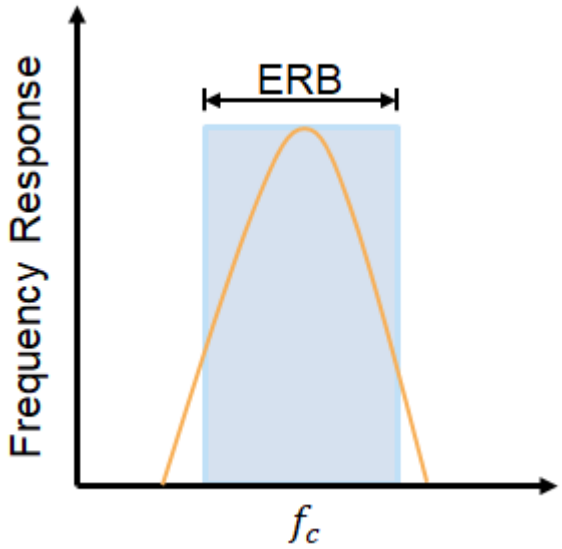
The notched-noise method assumes that the audible threshold corresponds to a constant signal-to-masker ratio at the output of the theoretical auditory filter. That is, the ratio of the power of the f_c tone and the shaded area is constant. Therefore, the relationship between the audible threshold and $2\Delta f$ (the notch bandwidth) is linearly related to the relationship between the noise passed through the filter and $2\Delta f$.



The derivative of the function relating Δf to the noise passed through the filter estimates the shape of the auditory filter. Because Δf has an inverse relationship with the noise power passed through the filter, the derivative of the function must be multiplied by –1. The resulting shape of the auditory filter is usually approximated as a roex filter.



The equivalent rectangular bandwidth of the auditory filter is defined as the width of a rectangular filter required to pass the same noise power as the auditory filter.



[4] defines ERB as a function of center frequency for young listeners with normal hearing and a moderate noise level:

$$\text{ERB} = 24.7(0.00437f_c + 1)$$

The ERB scale (ERBs) is an extension of the relationship between ERB and the center frequency, derived by integrating the reciprocal of the ERB function:

$$\text{ERBs} = 21.4\log_{10}(0.00437f + 1)$$

To design a gammatone filter bank, [2] suggests distributing the center frequencies of the filters in proportion to their bandwidth. To accomplish this, Gammatone Filter Bank block defines the center frequencies as linearly spaced on the ERB scale, covering the specified frequency range with the desired number of filters. You can specify the frequency range and desired number of filters using the **Frequency range (Hz)** and **Number of filters** parameters.

Gammatone Filter

The gammatone filter was introduced in [3]. The continuous impulse response is:

$$g(t) = at^{n-1}e^{-2\pi bt}\cos(2\pi f_c t + \phi)$$

where

- a –- amplitude factor
- t –- time in seconds
- n –- filter order (set to four to model human hearing)
- f_c–- center frequency
- b –- bandwidth, set to `1.019*hz2erb(`f_c`)`.
- ϕ –- phase factor

The gammatone filter is similar to the roex filter derived from the notched-noise experiment. The Gammatone Filter Bank block implements the digital filter as a cascade of four second-order sections, as described in [1].

Version History

Introduced in R2021b

References

[1] Slaney, Malcolm. "An Efficient Implementation of the Patterson-Holdsworth Auditory Filter Bank." Apple Computer Technical Report 35, 1993.

[2] Patterson, R.D., K. Robinson, J. Holdsworth, D. McKeown, C. Zhang, and M. Allerhand. "Complex Sounds and Auditory Images." *Auditory Physiology and Perception*. 1992, pp. 429–446.

[3] Aertsen, A. M. H. J., and P. I. M. Johannesma. "Spectro-Temporal Receptive Fields of Auditory Neurons in the Grassfrog." *Biological Cybernetics*. Vol. 38, Issue 4, 1980, pp. 223–234.

[4] Glasberg, Brian R., and Brian C. J. Moore. "Derivation of Auditory Filter Shapes from Notched-Noise Data." *Hearing Research*. Vol. 47. Issue 1-2, 1990, pp. 103–138.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

`gammatoneFilterBank` | Multiband Parametric EQ | Octave Filter Bank

Audio Plugin

Include audio plugin in model

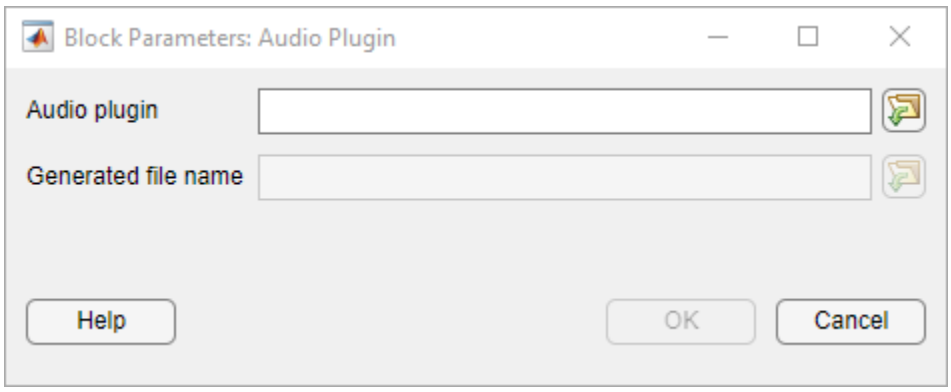
Libraries:
Audio Toolbox / User-Defined Functions

Description

The Audio Plugin block allows you to use an audio plugin as a block in your Simulink model. The Audio Plugin block generates a new block that has the same functionality as the desired plugin, and you can use the generated block to process audio signals in Simulink. For more information about audio plugins in MATLAB, see “Audio Plugins in MATLAB”.

Using the Block

To use the Audio Plugin block, place the block in your model and double-click it to open the dialog box.



Specify the desired audio plugin in the **Audio plugin** field. You can specify the audio plugin as:

- The name or file path of an audio plugin class. The class must derive from `audioPlugin` or `audioPluginSource`.
- An audio plugin binary file that is supported by `loadAudioPlugin`.
- An instance of an audio plugin class. This can either be a plugin authored in MATLAB or an externally authored plugin that is returned by `loadAudioPlugin`.

The Audio Plugin block generates a System object class file from the plugin and uses that System object to create the new block. You can optionally specify the file name and location of the generated System object with the **Generated file name** field. For more information about how the Audio Plugin block works and the System object it generates, see “Algorithms” on page 5-196.

Note The generated System object file must be on the MATLAB path for the plugin block to work. If the plugin is a hosted external plugin, the block generates additional files required by the generated System object. For more information, see “Code Generation” on page 5-197.

After you specify an audio plugin and click **OK**, you have a block with the same functionality as the audio plugin.

For an example with a model that uses the Audio Plugin block, see “Include an Audio Plugin in Simulink”.

Generated Block Parameters

The generated block has the same parameters as the specified audio plugin. These parameters are tunable, unless the parameter is an enumeration class that uses strings or characters as the underlying type. Each tunable parameter can be specified from an input port if you select the associated **Specify *`parameter`* from input port** parameter, where *`parameter`* is the name of the tunable parameter.

Note For audio plugins authored in MATLAB, a parameter must be specified as an `audioPluginParameter` in the plugin's `audioPluginInterface` for it to show in the block.

In addition to the parameters of the original plugin, the generated block has other, nontunable parameters that depend on whether the plugin is a source plugin or not. See “Parameters” on page 5-195 for more information.

Limitations

Some Simulink functionality, such as **Step Back**, requires saving and restoring the simulation state. Blocks that use hosted external plugins do not support simulation save and restore and therefore do not support associated functionality. For tips on using simulation save and restore functionality with blocks that use plugins authored in MATLAB, see “Tips” on page 5-196.

Ports

Input

x — Plugin input
column vector | matrix

The block input is the same as the original audio plugin input and accepts the same data types. If the original plugin takes multiple inputs, the block has multiple input ports. The block has additional input ports if you select any of the **Specify *`parameter`* from input port** parameters, where *`parameter`* is the name of a tunable parameter.

The block does not have an input port if the original plugin is a source plugin. Source plugins derive from `audioPluginSource` or `externalAudioPluginSource`.

Note For audio plugins authored in MATLAB, the number of inputs and the number of channels per input are defined by the `InputChannels` property in the plugin's `audioPluginInterface`.

Output

y — Plugin output
scalar | column vector | matrix

The block output is the same as the original audio plugin and returns the same data types. If the original plugin returns multiple outputs, the block has multiple output ports.

Note For audio plugins authored in MATLAB, the number of outputs and the number of channels per output are defined by the `OutputChannels` property in the plugin's `audioPluginInterface`.

Parameters

Note These parameters are not part of the block until after you specify an audio plugin in the dialog box.

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)**.

The block does not have this parameter if the audio plugin is a source plugin.

Input sample rate (Hz) — Sample rate of input
positive scalar

Sample rate of the input, specified as a positive scalar. The default is what the `getSampleRate` method returns for the original audio plugin.

The block does not have this parameter if the audio plugin is a source plugin.

Dependencies

To enable this parameter, clear the **Inherit sample rate from input** parameter.

Samples per frame — Number of samples per frame
positive integer

Number of samples per frame output by the block, specified as a positive scalar. The default is what the `getSamplesPerFrame` method returns for the original audio plugin.

The block has this parameter only if the audio plugin is a source plugin.

Output data type — Data type of output signal
`double` (default) | `single`

Data type of the output signal, specified as `double` or `single`.

The block has this parameter only if the audio plugin is a source plugin.

Sample rate (Hz) — Sample rate of output signal
positive scalar

Sample rate of the output signal, specified as a positive scalar. The default value of this parameter is the sample rate that the `getSampleRate` method returns for the original audio plugin.

The block has this parameter only if the audio plugin is a source plugin.

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 but has a slower simulation speed than `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`.

Block Characteristics

Data Types	`Boolean`[a] | `bus`[ba] | `double`[a] | `enumerated`[a] | `fixed point`[a] | `half`[a] | `integer`[a] | `single`[a] | `string`[a]
Direct Feedthrough	`no`
Multidimensional Signals	`yes`[a]
Variable-Size Signals	`yes`[ca]
Zero-Crossing Detection	`no`

a Actual data type or capability support depends on block implementation.
b See Nonvirtual Buses and MATLAB System Block for more information.
c See Variable-Size Signals for more information.

Tips

To use Simulink functionality that requires saving and restoring the simulation state, such as **Step Back**, with a block that uses a plugin authored in MATLAB, the original plugin implementation must correctly save and load its state.

- If the original plugin is a System object, it must correctly save and load its state using the `saveObjectImpl` and `loadObjectImpl` methods.
- If the original plugin is an `audioPlugin` and not a System object plugin, it must correctly save and load its state using the `saveobj` and `loadobj` methods.

Note If the original plugin does not maintain any state, no additional considerations are necessary for the save and restore functionality.

Algorithms

The Audio Plugin block generates the code for a System object class from the specified audio plugin using the `generateSimulinkAudioPlugin` function. `generateSimulinkAudioPlugin` designs the System object to be compatible with Simulink through the MATLAB System block. The Audio Plugin block then uses the MATLAB System block to create a new block from the generated System object with the same parameters and functionality as the original audio plugin.

Version History

Introduced in R2022b

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

Generating code from blocks that use external audio plugins has additional requirements. External audio plugins include plugins loaded into MATLAB with `loadAudioPlugin` and plugin binaries such as VST plugins.

- The Audio Plugin block generates files in addition to the System object file to aid in code generation. These files include *`sysObjName`*`PluginLoader.m` and *`sysObjName`*`Interface.m` where *`sysObjName`* is the name of the generated System object. The block also generates *`sysObjName`*`Tables.mat` if the plugin has any parameters. These additional files are required for code generation and running the block in simulation.
- You must select the **Support long long** parameter in the **Hardware Implementation** pane of the **Model Settings**.
- If you are using an ERT target, you must set the **Language** parameter to `C++` under the **Target selection** section of the **Code Generation** pane in the **Model Settings**. You must also select the **Dynamic memory allocation in MATLAB functions** parameter in the **Advanced parameters** section of the **Simulation Target** pane.
- To use a standalone executable generated from a block with an external plugin, you must generate the `jucehost.dll` file on Windows or the `libjucehost.dylib` file on macOS by selecting the **Package code and artifacts** parameter under the **Build process** section of the **Code Generation** pane in the **Model Settings**.
  - On Windows platforms, you must make the `jucehost.dll` file visible to the standalone executable. To do this, add the path to the `jucehost.dll` file to the `PATH` environment variable or copy the `jucehost.dll` file to the same folder as the standalone executable .
  - On macOS platforms, you must make the `libjucehost.dylib` file visible to the standalone executable. To do this, place the `libjucehost.dylib` file in the `/usr/lib` directory.

Hosted AUv3 plugins do not support code generation.

See Also

Functions

`generateSimulinkAudioPlugin` | `loadAudioPlugin`

Classes

`audioPlugin` | `audioPluginSource`

Topics

“Audio Plugins in MATLAB”

CREPE

CREPE deep pitch estimation neural network

Libraries:
Audio Toolbox / Deep Learning

Description

The CREPE block leverages a pretrained convolutional neural model to estimate pitch from an audio signal. This block requires Deep Learning Toolbox.

Examples

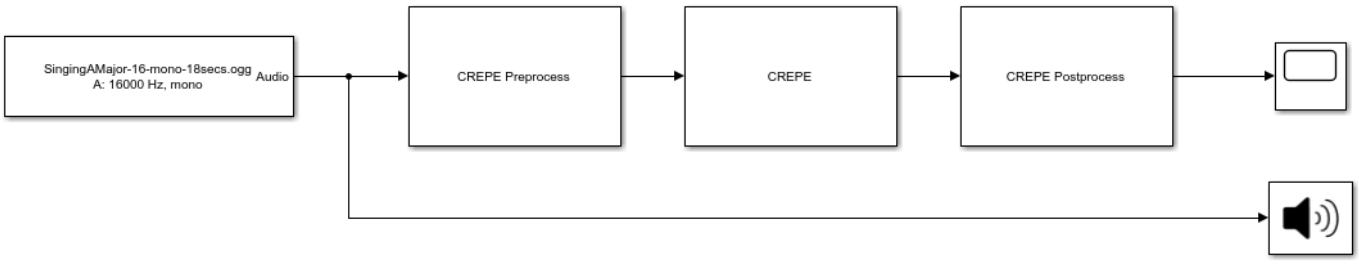
Estimate Pitch Using CREPE Blocks

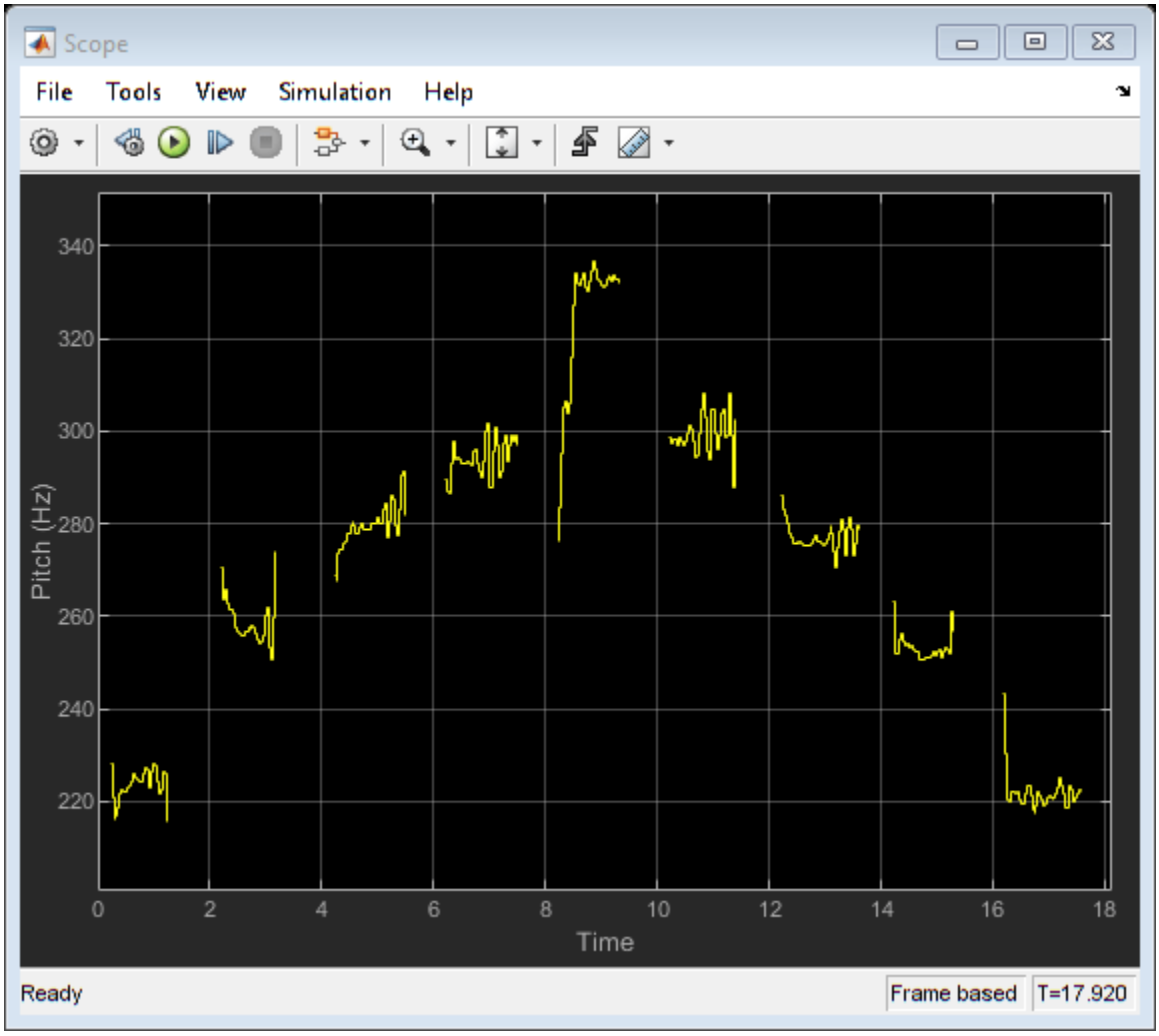
This example shows how to use the CREPE blocks to combine preprocessing, network inference, and postprocessing and obtain pitch estimations from an audio signal. See “Estimate Pitch Using Deep Pitch Estimator Block” on page 5-210 for an example that uses the Deep Pitch Estimator block to perform the same task.

Adjust the parameters of the blocks to speed up computation and see the pitch estimations in real time as the audio plays.

- Set the **Overlap percentage (%)** of the CREPE Preprocess block to `50`. With a lower overlap percentage, the system processes frames less frequently.
- Set the **Number of output frames** of the CREPE Preprocess block to `5`. This causes the CREPE Preprocess block to buffer audio frames and pass them to the CREPE block in batches. Passing batches to the CREPE block improves computational efficiency by allowing it to process multiple frames in parallel. However, it also increases latency because the system outputs pitch estimations in batches instead of one at a time.
- Set the **Model capacity** of the CREPE block to `Large`. This model has fewer parameters than the full-size model, leading to faster computation at the cost of slightly lower accuracy.

Run the model to listen to a singing voice and view the estimated pitch in real time.





Ports

Input

Port_1 — Preprocessed audio input
vector | 4-D array

Preprocessed input to the network, specified as a 1024-by-1-by-1-by-*N* array, where *N* is the number of audio frames. If the input has only one frame, the block accepts a vector.

The CREPE Preprocess block takes in an audio signal and outputs the preprocessed frames.

Data Types: `single` | `double`

Output

Port_1 — Network activations
matrix

Network activations output by the CREPE network, returned as an *N*-by-360 matrix, where *N* is the number of input frames.

The CREPE Postprocess block converts these network activations to pitch estimates in Hz.

Data Types: `single`

Parameters

Model capacity — Size of trained neural network

`Full` (default) | `Large` | `Medium` | `Small` | `Tiny`

Model capacity, specified as `Full`, `Large`, `Medium`, `Small`, or `Tiny`. The smaller sizes correspond to fewer parameters in the model, leading to faster computation but lower accuracy.

Mini-batch size — Size of mini-batches

128 (default) | positive integer

Size of mini-batches to use for prediction, specified as a positive integer. Larger mini-batch sizes require more memory but can lead to faster predictions.

Block Characteristics

Data Types	`double` | `single`
Direct Feedthrough	no
Multidimensional Signals	no
Variable-Size Signals	no
Zero-Crossing Detection	no

Version History

Introduced in R2023a

References

[1] Kim, Jong Wook, Justin Salamon, Peter Li, and Juan Pablo Bello. “Crepe: A Convolutional Representation for Pitch Estimation.” In *2018 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 161–65. Calgary, AB: IEEE, 2018. https://doi.org/10.1109/ICASSP.2018.8461329.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

Usage notes and limitations:

- To generate generic C code that does not depend on third-party libraries, in the **Configuration Parameters > Code Generation** general category, set the **Language** parameter to `C`.
- To generate C++ code, in the **Configuration Parameters > Code Generation** general category, set the **Language** parameter to `C++`. To specify the target library for code generation, in the **Code Generation > Interface** category, set the **Target Library** parameter. Setting this parameter to `None` generates generic C++ code that does not depend on third-party libraries.
- For ERT-based targets, the **Support: variable-size signals** parameter in the **Code Generation> Interface** pane must be enabled.
- For a list of networks and layers supported for code generation, see “Networks and Layers Supported for Code Generation” (MATLAB Coder).

See Also

Blocks

CREPE Preprocess | CREPE Postprocess | Deep Pitch Estimator

Functions

`crepe` | `crepePreprocess` | `crepePostprocess` | `pitchnn` | `pitch`

CREPE Preprocess

Preprocess audio for CREPE deep pitch estimation

Libraries:
Audio Toolbox / Deep Learning

Description

The CREPE Preprocess block generates frames from the input audio signal that you can feed to a CREPE pretrained network or to a network that accepts the same inputs as CREPE.

Examples

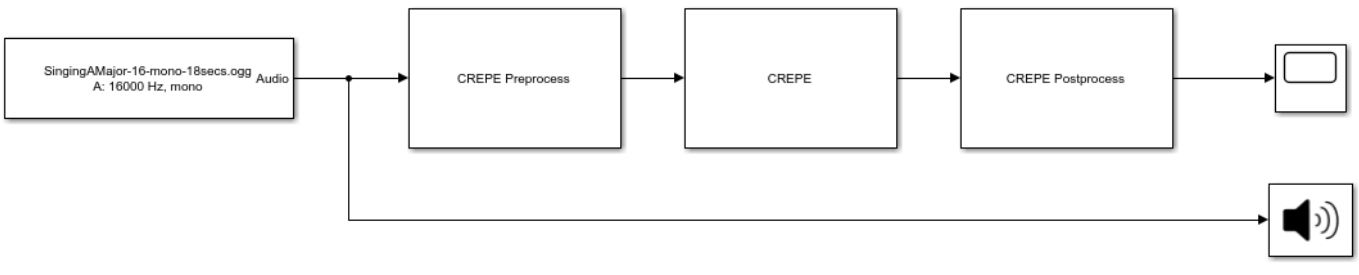
Estimate Pitch Using CREPE Blocks

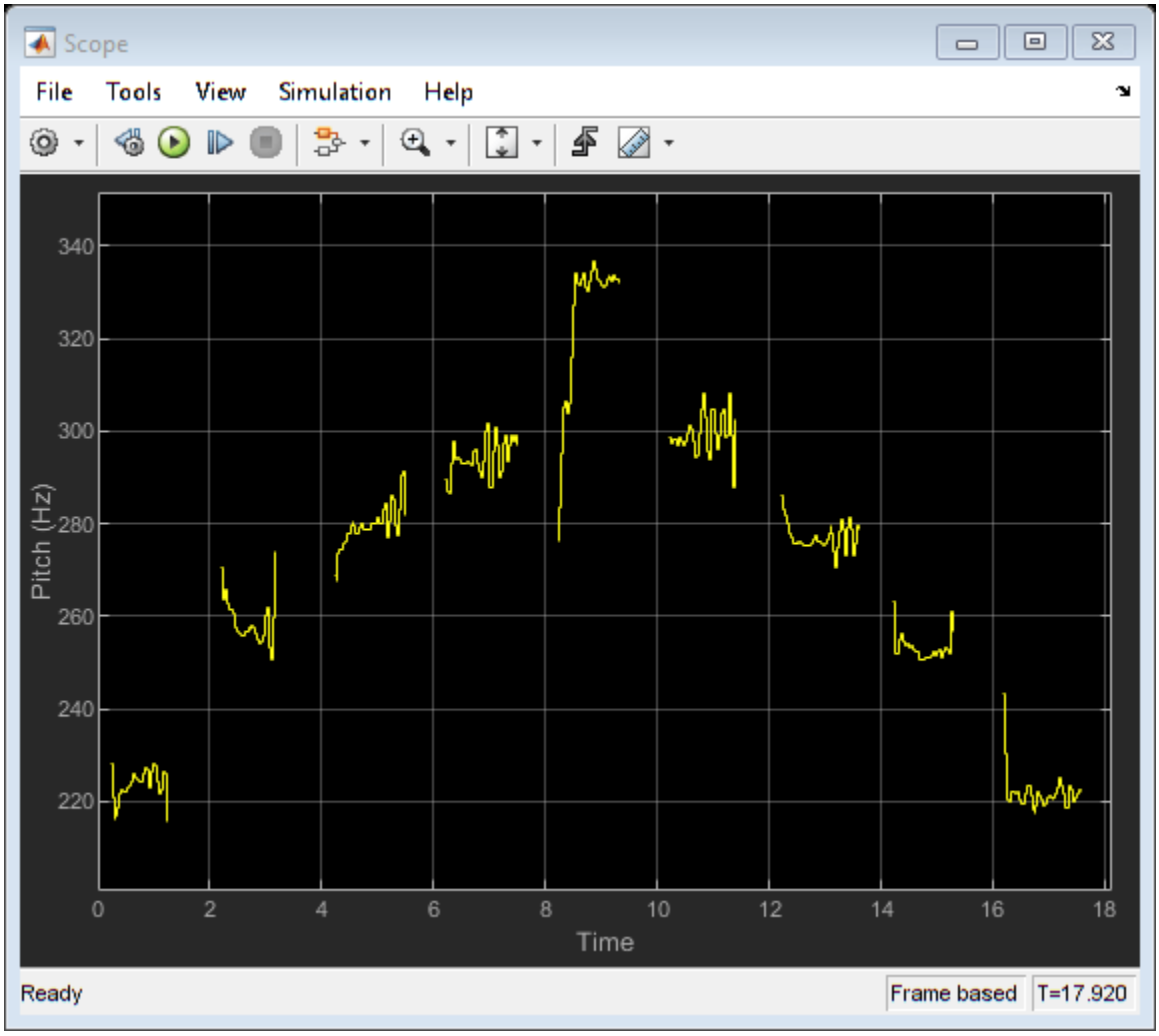
This example shows how to use the CREPE blocks to combine preprocessing, network inference, and postprocessing and obtain pitch estimations from an audio signal. See “Estimate Pitch Using Deep Pitch Estimator Block” on page 5-210 for an example that uses the Deep Pitch Estimator block to perform the same task.

Adjust the parameters of the blocks to speed up computation and see the pitch estimations in real time as the audio plays.

- Set the **Overlap percentage (%)** of the CREPE Preprocess block to `50`. With a lower overlap percentage, the system processes frames less frequently.
- Set the **Number of output frames** of the CREPE Preprocess block to `5`. This causes the CREPE Preprocess block to buffer audio frames and pass them to the CREPE block in batches. Passing batches to the CREPE block improves computational efficiency by allowing it to process multiple frames in parallel. However, it also increases latency because the system outputs pitch estimations in batches instead of one at a time.
- Set the **Model capacity** of the CREPE block to `Large`. This model has fewer parameters than the full-size model, leading to faster computation at the cost of slightly lower accuracy.

Run the model to listen to a singing voice and view the estimated pitch in real time.





Ports

Input

Port_1 — Audio input
vector

Audio input, specified as a one-channel signal (vector). If **Sample rate of input signal (Hz)** is 16e3, there are no restrictions on the input frame length. If **Sample rate of input signal (Hz)** is different from 16e3, then the input frame length must be a multiple of the decimation factor of the resampling operation that the block performs. If the input frame length does not satisfy this condition, the block generates an error message with information on the decimation factor.

Data Types: `single` | `double`

Output

Port_1 — Preprocessed audio frames
4-D array

Preprocessed audio frames for the CREPE neural network, returned as a 1024-by-1-by-1-by-*N* array, where *N* is the number of generated frames specified by **Number of output frames**.

Data Types: `single`

Parameters

Sample rate of input signal (Hz) — Sample rate of input signal in Hz

`16e3` (default) | positive scalar

Sample rate of the input signal in Hz, specified as a positive scalar.

Overlap percentage (%) — Overlap percentage between consecutive frames

85 (default) | [0, 100)

Specify the overlap percentage between consecutive frames as a scalar in the range [0, 100).

Number of output frames — Number of generated frames

1 (default) | positive integer

Number of generated frames in the output, specified as a positive integer.

Block Characteristics

Data Types	`double` | `single`
Direct Feedthrough	no
Multidimensional Signals	no
Variable-Size Signals	no
Zero-Crossing Detection	no

Version History

Introduced in R2023a

References

[1] Kim, Jong Wook, Justin Salamon, Peter Li, and Juan Pablo Bello. “Crepe: A Convolutional Representation for Pitch Estimation.” In *2018 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 161–65. Calgary, AB: IEEE, 2018. https://doi.org/10.1109/ICASSP.2018.8461329.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

Blocks
CREPE | CREPE Postprocess | Deep Pitch Estimator

Functions
`crepePreprocess` | `crepe` | `crepePostprocess` | `pitchnn` | `pitch`

CREPE Postprocess

Postprocess output of CREPE pitch estimation network

Libraries:
Audio Toolbox / Deep Learning

Description

The CREPE Postprocess block converts the output of a CREPE pretrained network to pitch estimates in Hz.

Examples

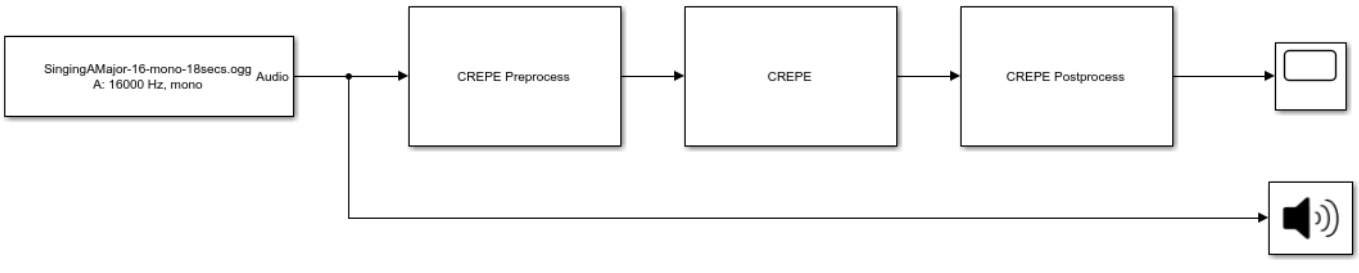
Estimate Pitch Using CREPE Blocks

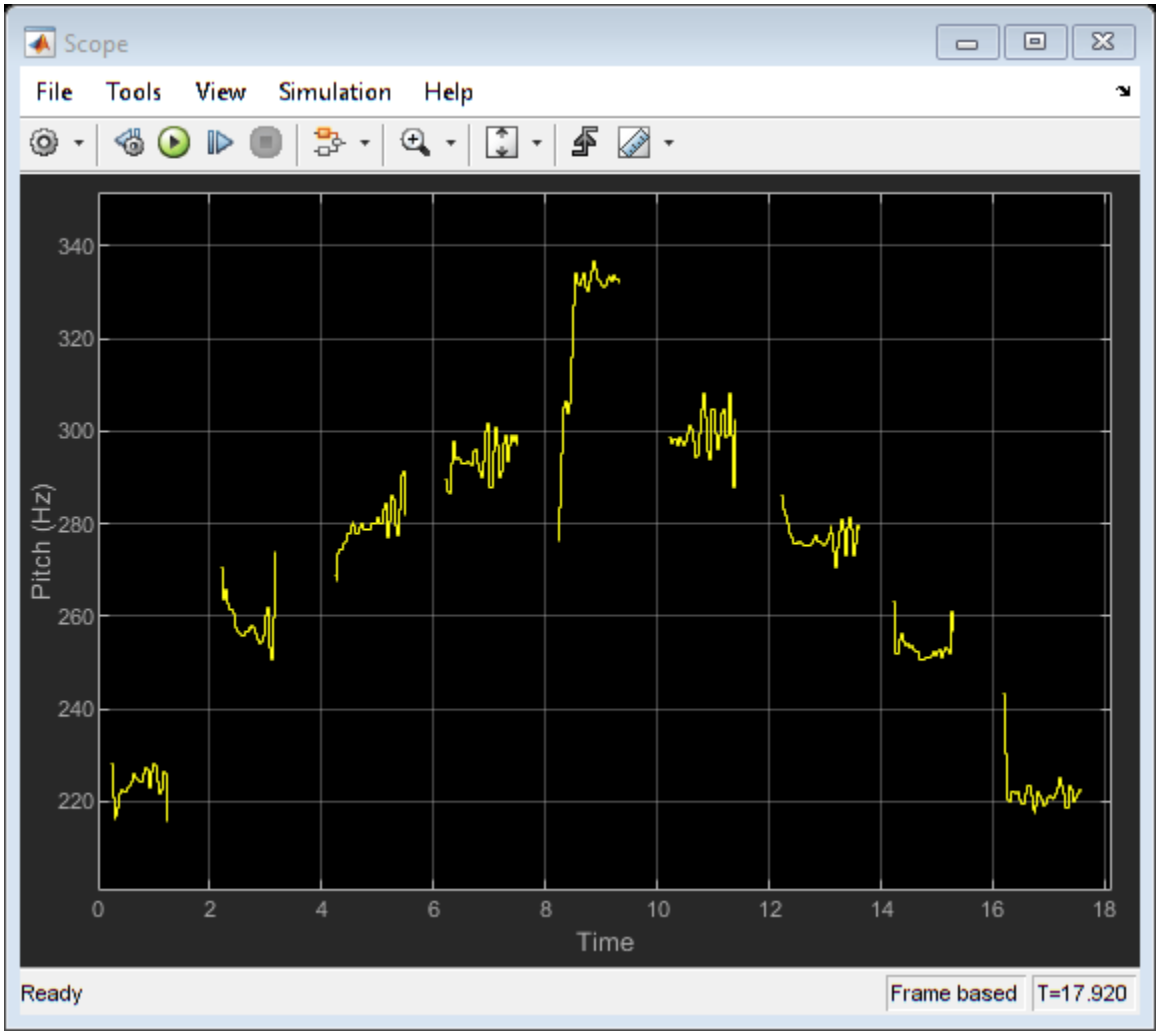
This example shows how to use the CREPE blocks to combine preprocessing, network inference, and postprocessing and obtain pitch estimations from an audio signal. See “Estimate Pitch Using Deep Pitch Estimator Block” on page 5-210 for an example that uses the Deep Pitch Estimator block to perform the same task.

Adjust the parameters of the blocks to speed up computation and see the pitch estimations in real time as the audio plays.

- Set the **Overlap percentage (%)** of the CREPE Preprocess block to `50`. With a lower overlap percentage, the system processes frames less frequently.
- Set the **Number of output frames** of the CREPE Preprocess block to `5`. This causes the CREPE Preprocess block to buffer audio frames and pass them to the CREPE block in batches. Passing batches to the CREPE block improves computational efficiency by allowing it to process multiple frames in parallel. However, it also increases latency because the system outputs pitch estimations in batches instead of one at a time.
- Set the **Model capacity** of the CREPE block to `Large`. This model has fewer parameters than the full-size model, leading to faster computation at the cost of slightly lower accuracy.

Run the model to listen to a singing voice and view the estimated pitch in real time.





Ports

Input

Port_1 — CREPE network activations
matrix

CREPE network activations, specified as an *N*-by-360 matrix, where *N* is the number of frames output by the CREPE neural network.

Data Types: `single` | `double`

Output

Port_1 — Estimated fundamental frequency
column vector

Estimated fundamental frequency in Hz, returned as an *N*-by-1 vector, where *N* is the number of frames in the input.

Data Types: `single` | `double`

Parameters

Confidence threshold — Pitch confidence threshold

`0.5` (default) | scalar in the range [0, 1)

Pitch confidence threshold, specified as a scalar in the range [0, 1). In postprocessing, the block suppresses fundamental frequencies where the network confidence is below the threshold.

Note If the maximum value of the network output is less than the confidence threshold, the block returns `NaN`.

Tunable: Yes

Block Characteristics

Data Types	`double` | `single`
Direct Feedthrough	no
Multidimensional Signals	no
Variable-Size Signals	no
Zero-Crossing Detection	no

Version History

Introduced in R2023a

References

[1] Kim, Jong Wook, Justin Salamon, Peter Li, and Juan Pablo Bello. “Crepe: A Convolutional Representation for Pitch Estimation.” In *2018 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 161–65. Calgary, AB: IEEE, 2018. https://doi.org/10.1109/ICASSP.2018.8461329.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

Blocks

CREPE | CREPE Preprocess | Deep Pitch Estimator

Functions
`crepePostprocess` | `crepe` | `crepePreprocess` | `pitchnn` | `pitch`

Deep Pitch Estimator

Estimate pitch with CREPE deep learning neural network

Libraries:
Audio Toolbox / Deep Learning

Description

The Deep Pitch Estimator block uses a CREPE pretrained neural network to estimate the pitch from audio signals. The block combines necessary audio preprocessing, network inference, and postprocessing of network output to return pitch estimations in Hz. This block requires Deep Learning Toolbox.

Examples

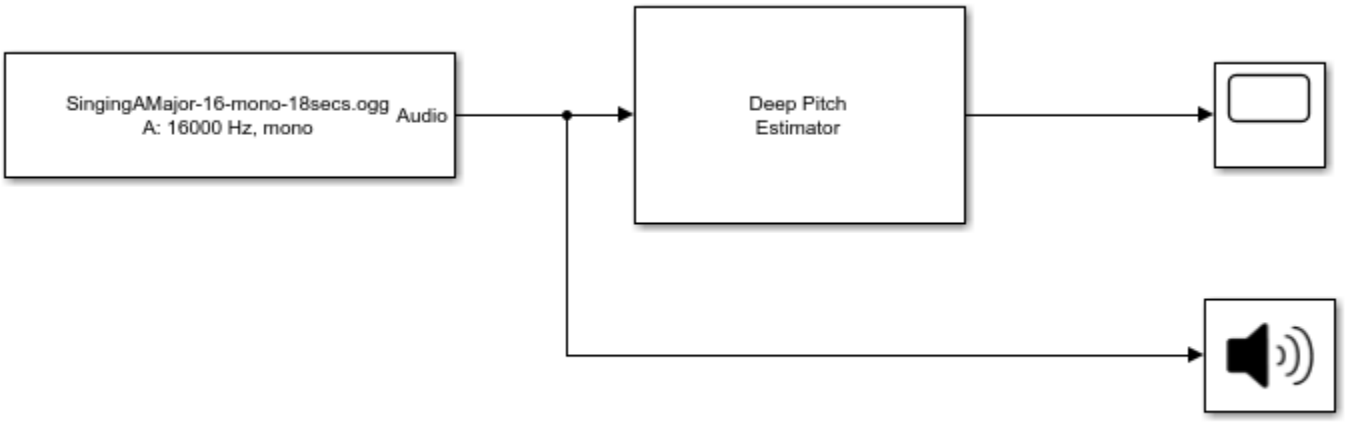
Estimate Pitch Using Deep Pitch Estimator Block

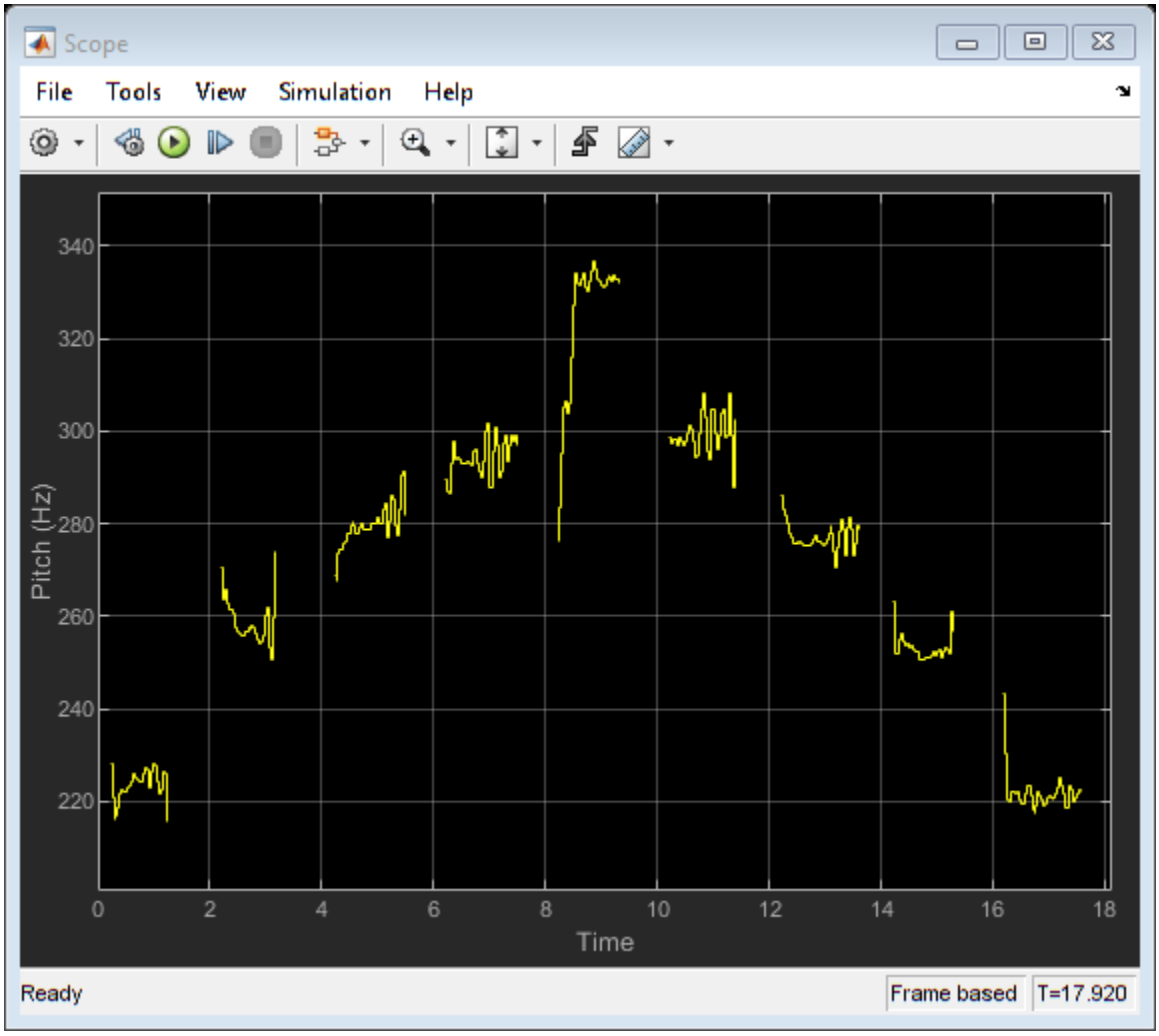
This example shows how to use the Deep Pitch Estimator block to estimate the pitch of an audio signal in Simulink®. See “Estimate Pitch Using CREPE Blocks” on page 5-198 for an example that uses the CREPE Preprocess, CREPE, and CREPE Postprocess blocks to perform the same task.

Adjust the block parameters to speed up computation and see the pitch estimations in real time as the audio plays.

- Set the **Overlap percentage (%)** parameter to `50`. With a lower overlap percentage, the block computes and outputs pitch estimations less frequently.
- Set the **Number of buffered pitch estimations** parameter to `5`. A higher value for this parameter allows the block to improve computational efficiency by operating on multiple audio frames in parallel. However, a higher value also increases latency because the block returns pitch estimations in batches instead of one at a time.
- Set the **Model capacity** parameter to `Large`. This model has fewer parameters than the full-size model, leading to faster computation at the cost of slightly lower accuracy.

Run the model to listen to a singing voice and view the estimated pitch in real time.





Ports

Input

Port_1 — Audio input
vector

Audio input, specified as a one-channel signal (vector). If **Sample rate of input signal (Hz)** is 16e3, there are no restrictions on the input frame length. If **Sample rate of input signal (Hz)** is different from 16e3, then the input frame length must be a multiple of the decimation factor of the resampling operation that the block performs. If the input frame length does not satisfy this condition, the block generates an error message with information on the decimation factor.

Data Types: `single` | `double`

Output

Port_1 — Estimated fundamental frequency
column vector

Estimated fundamental frequency in Hz, returned as an *N*-by-1 vector, where *N* is the number of pitch estimations specified by **Number of buffered pitch estimations**.

Data Types: `single`

Parameters

Sample rate of input signal (Hz) — Sample rate of input signal in Hz

`16e3` (default) | positive scalar

Sample rate of the input signal in Hz, specified as a positive scalar.

Overlap percentage (%) — Overlap percentage between consecutive frames

`85` (default) | [0, 100)

Specify the overlap percentage between consecutive input frames as a scalar in the range [0, 100).

Number of buffered pitch estimations — Number of pitch estimations in output

`1` (default) | positive integer

Number of pitch estimations in output, specified as a positive integer.

A higher value allows the block to improve computational efficiency by operating on multiple audio frames in parallel. However, it also increases latency because the block buffers the specified number of pitch estimations before returning them.

Confidence threshold — Pitch confidence threshold

`0.5` (default) | scalar in the range [0, 1)

Pitch confidence threshold, specified as a scalar in the range [0, 1). In postprocessing, the block suppresses fundamental frequencies where the network confidence is below the threshold.

Note If the maximum value of the network output is less than the confidence threshold, the block returns `NaN`.

Model capacity — Size of trained neural network

`Full` (default) | `Large` | `Medium` | `Small` | `Tiny`

Model capacity, specified as `Full`, `Large`, `Medium`, `Small`, or `Tiny`. The smaller sizes correspond to fewer parameters in the model, leading to faster computation but lower accuracy.

Block Characteristics

Data Types	`double` | `single`
Direct Feedthrough	no

Multidimensional Signals	no
Variable-Size Signals	no
Zero-Crossing Detection	no

Version History

Introduced in R2023a

References

[1] Kim, Jong Wook, Justin Salamon, Peter Li, and Juan Pablo Bello. “Crepe: A Convolutional Representation for Pitch Estimation.” In *2018 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 161–65. Calgary, AB: IEEE, 2018. https://doi.org/10.1109/ICASSP.2018.8461329.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

Usage notes and limitations:

- To generate generic C code that does not depend on third-party libraries, in the **Configuration Parameters > Code Generation** general category, set the **Language** parameter to `C`.
- To generate C++ code, in the **Configuration Parameters > Code Generation** general category, set the **Language** parameter to `C++`. To specify the target library for code generation, in the **Code Generation > Interface** category, set the **Target Library** parameter. Setting this parameter to `None` generates generic C++ code that does not depend on third-party libraries.
- For ERT-based targets, the **Support: variable-size signals** parameter in the **Code Generation> Interface** pane must be enabled.
- For a list of networks and layers supported for code generation, see “Networks and Layers Supported for Code Generation” (MATLAB Coder).

See Also

Blocks

CREPE | CREPE Preprocess | CREPE Postprocess

Functions

`pitchnn` | `crepe` | `crepePreprocess` | `crepePostprocess` | `pitch`