Audio Toolbox™

Getting Started Guide

MATLAB&SIMULINK®

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

Contents

1
Introduction

2
Audio IO

3
Audio Plugins

4
Deep Learning

5
Intro to Audio Deep Learning

11

Audio Plugins in MATLAB

12

Real-Time Audio in Simulink

13

Convert MATLAB Code to an Audio Plugin

14

Host External Audio Plugins

Introduction

Audio Toolbox Product Description

Design and analyze speech, acoustic, and audio processing systems

Audio Toolbox provides tools for audio processing, speech analysis, and acoustic measurement. It includes algorithms for processing audio signals such as equalization and time stretching, estimating acoustic signal metrics such as loudness and sharpness, and extracting audio features such as MFCC and pitch. It also provides advanced machine learning models, including i-vectors, and pretrained deep learning networks, including VGGish and CREPE. Toolbox apps support live algorithm testing, impulse response measurement, and signal labeling. The toolbox provides streaming interfaces to ASIO™, CoreAudio, and other sound cards; MIDI devices; and tools for generating and hosting VST and Audio Units plugins.

With Audio Toolbox you can import, label, and augment audio data sets, as well as extract features to train machine learning and deep learning models. The pre-trained models provided can be applied to audio recordings for high-level semantic analysis.

You can prototype audio processing algorithms in real time or run custom acoustic measurements by streaming low-latency audio to and from sound cards. You can validate your algorithm by turning it into an audio plugin to run in external host applications such as Digital Audio Workstations. Plugin hosting lets you use external audio plugins as regular MATLAB® objects.

Acknowledgements

VST is a trademark and software of Steinberg Media Technologies GmbH.

ASIO is a trademark and software of Steinberg Media Technologies GmbH.

Audio IO

Audio Input and Audio Output

This example shows how to read audio from a file and write audio to your speakers.

Read and Write Entire Audio Files

To read an entire audio file into the workspace and then write the entire audio signal to your speakers, use the `audioread` and `soundsc` functions. Call `audioread` with a file name to read the entire audio file and the sample rate of the audio. Call `soundsc` with the audio data and sample rate to play the audio to your default speakers.

```
[audioData,fs] = audioread("SpeechDFT-16-8-mono-5secs.wav");
soundsc(audioData,fs)
```

Read and Write Audio Files Frame-by-Frame

To read audio frame-by-frame into the workspace and then write audio frame-by-frame to your speakers, use the `dsp.AudioFileReader` and `audioDeviceWriter` functions.

Create a `dsp.AudioFileReader` object to read audio from a file frame-by-frame. The audio file reader saves the sample rate of the audio file to the `SampleRate` property.

```
fileReader = dsp.AudioFileReader("Filename","SpeechDFT-16-8-mono-5secs.wav")

fileReader =
  dsp.AudioFileReader with properties:

          Filename: 'B:\matlab\toolbox\audio\samples\SpeechDFT-16-8-mono-5secs.wav'
         PlayCount: 1
         ReadRange: [1 Inf]
    SamplesPerFrame: 1024
        SampleRate: 8000
     OutputDataType: 'double'

  Show all properties
```

Create an `audioDeviceWriter` object to write audio to your speakers. Set the sample rate of the `audioDeviceWriter` object to the sample rate of the audio file.

```
deviceWriter = audioDeviceWriter("SampleRate",fileReader.SampleRate)

deviceWriter =
  audioDeviceWriter with properties:

        Driver: 'DirectSound'
        Device: 'Default'
    SampleRate: 8000

  Show all properties
```

In a loop, read from the file and write to the device. While the loop runs, audio is played to your default audio device.

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

```
    % Read one frame of audio data from the file.
    audioData = fileReader();

    % Write one frame of audio data to your speakers.
    deviceWriter(audioData);

end
```

As a best practice, release the file and audio device when you are done.

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

To learn how to implement other audio I/O configurations, such as reading from a microphone or writing to a speaker, see "Real-Time Audio in MATLAB" on page 10-2.

See Also

audioDeviceReader | audioDeviceWriter | audioPlayerRecorder | dsp.AudioFileWriter | dsp.AudioFileReader | asiosettings | getAudioDevices

More About

- "Process and Analyze Streaming Audio" on page 6-2
- "Real-Time Audio in Simulink" on page 12-2
- "Audio I/O: Buffering, Latency, and Throughput" on page 8-2

Audio Plugins

Design an Audio Plugin

An audio plugin encapsulates an audio processing algorithm and enables you to tune the parameters of the algorithm while streaming audio.

Define an Audio Plugin

To define a plugin that enables users to adjust stereo width:

1 Create a class definition that inherits from `audioPlugin.`
2 Parameterize the stereo width of the processing algorithm by defining the public property `Width`.
3 Enable users to tune the stereo width by defining an `audioPluginInterface` that contains `Width` as an `audioPluginParameter`.
4 Define the audio processing by creating a `process` method. The `process` method takes the audio input, `in`, and adjusts the stereo width by: (a) applying mid-side encoding, (b) adjusting the stereo width based on the user-controlled `Width` parameter, and then (c) applying mid-side decoding.

```matlab
classdef StereoWidth < audioPlugin                          % <== (1) Inherit from audioPlugin
    properties
        Width = 1;                                          % <== (2) Define tunable property.
    end
    properties (Constant)
        PluginInterface = audioPluginInterface( ...         % <== (3) Map tunable property to |
            audioPluginParameter('Width', ...
                'Mapping',{'pow',2,0,4}));
    end
    methods
        function out = process(plugin,in)                   %< == (4) Define audio processing.

            x = [in(:,1) + in(:,2), in(:,1) - in(:,2)];         %  (a) Mid-side encoding.
            y = [x(:,1), x(:,2)*plugin.Width];                  %  (b) Adjust stereo width.
            out = [(y(:,1) + y(:,2))/2, (y(:,1) - y(:,2))/2];   %  (c) Mid-side decoding.

        end
    end
end
```

Prototype the Audio Plugin

Once you have defined an audio plugin, you can prototype it using the Audio Test Bench app. The **Audio Test Bench** app enables you to stream audio through the plugin while you tune parameters, perform listening tests, and visualize the original and processed audio. To open your `StereoWidth` plugin in the **Audio Test Bench** app, at the MATLAB® command prompt, enter:

```matlab
audioTestBench(StereoWidth)
```
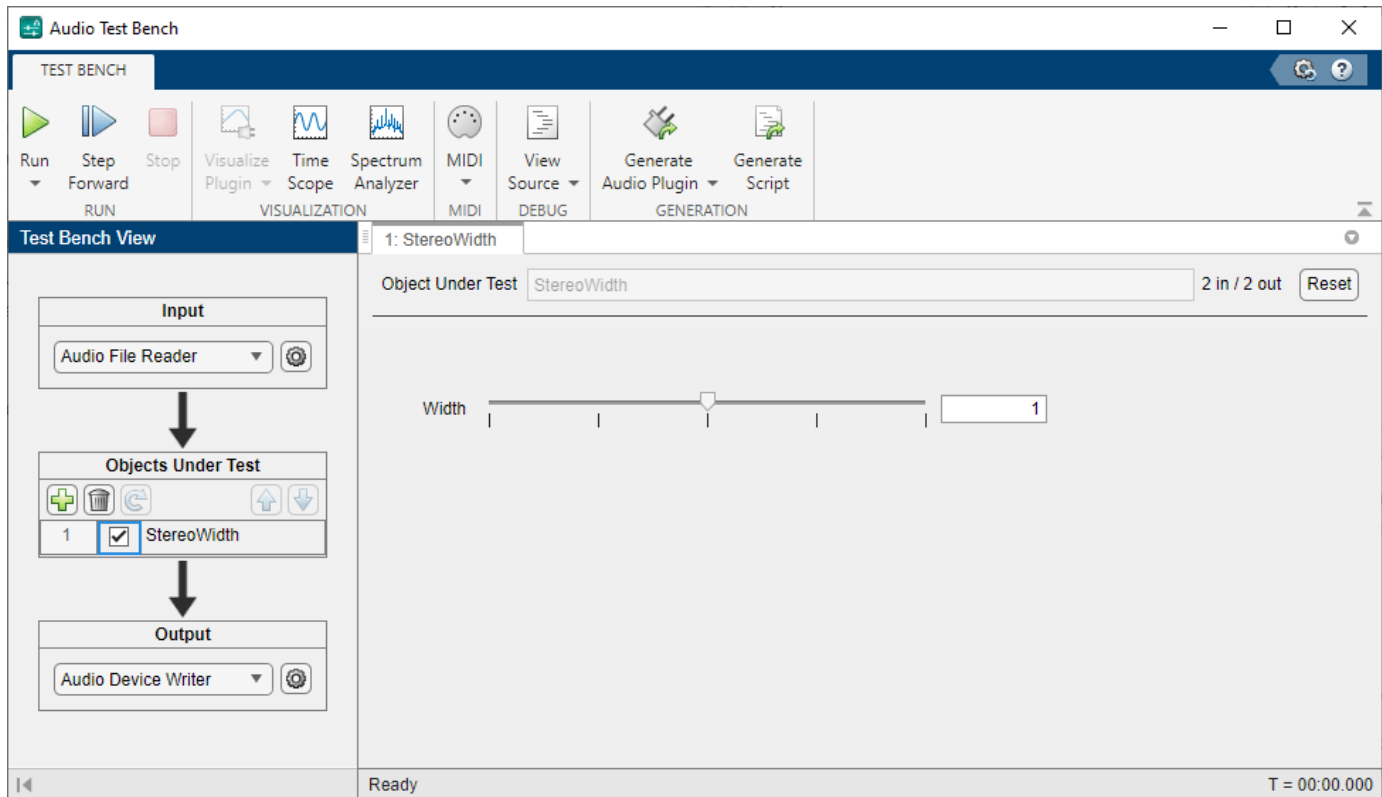
Validate and Generate a VST Plugin

You can validate a MATLAB® audio plugin and generate a VST plugin from the **Audio Test Bench**. You can also validate and generate the plugin from the command line by using the `validateAudioPlugin` and `generateAudioPlugin` functions. Once generated, you can deploy your plugin to a digital audio workstation (DAW).

```
validateAudioPlugin StereoWidth

Checking plugin class 'StereoWidth'...

passed.
Generating testbench file 'testbench_StereoWidth.m'... done.
Running testbench... passed.
Generating mex file 'testbench_StereoWidth_mex.mexw64'... done.
Running mex testbench... passed.
Deleting testbench.
Ready to generate audio plugin.

generateAudioPlugin StereoWidth

.......
```

The VST plugin is saved to your working directory.

See Also

audioPlugin | audioPluginSource | audioPluginGridLayout | audioPluginInterface | audioPluginParameter | generateAudioPlugin | validateAudioPlugin | **Audio Test Bench**

More About

Deep Learning

Classify Sound Using Deep Learning

This example shows how to classify a sound by using deep learning processes.

Create a Data Set

Generate 1000 white noise signals, 1000 brown noise signals, and 1000 pink noise signals. Each signal represents a duration of 0.5 seconds, assuming a 44.1 kHz sample rate.

```
fs = 44.1e3;
duration = 0.5;
N = duration*fs;

wNoise = 2*rand([N,1000]) - 1;
wLabels = repelem(categorical("white"),1000,1);

bNoise = filter(1,[1,-0.999],wNoise);
bNoise = bNoise./max(abs(bNoise),[],'all');
bLabels = repelem(categorical("brown"),1000,1);

pNoise = pinknoise([N,1000]);
pLabels = repelem(categorical("pink"),1000,1);
```

Explore the Data Set

Listen to a white noise signal and visualize it using the `melSpectrogram` function.

```
sound(wNoise(:,1),fs)
melSpectrogram(wNoise(:,1),fs)
title('White Noise')
```
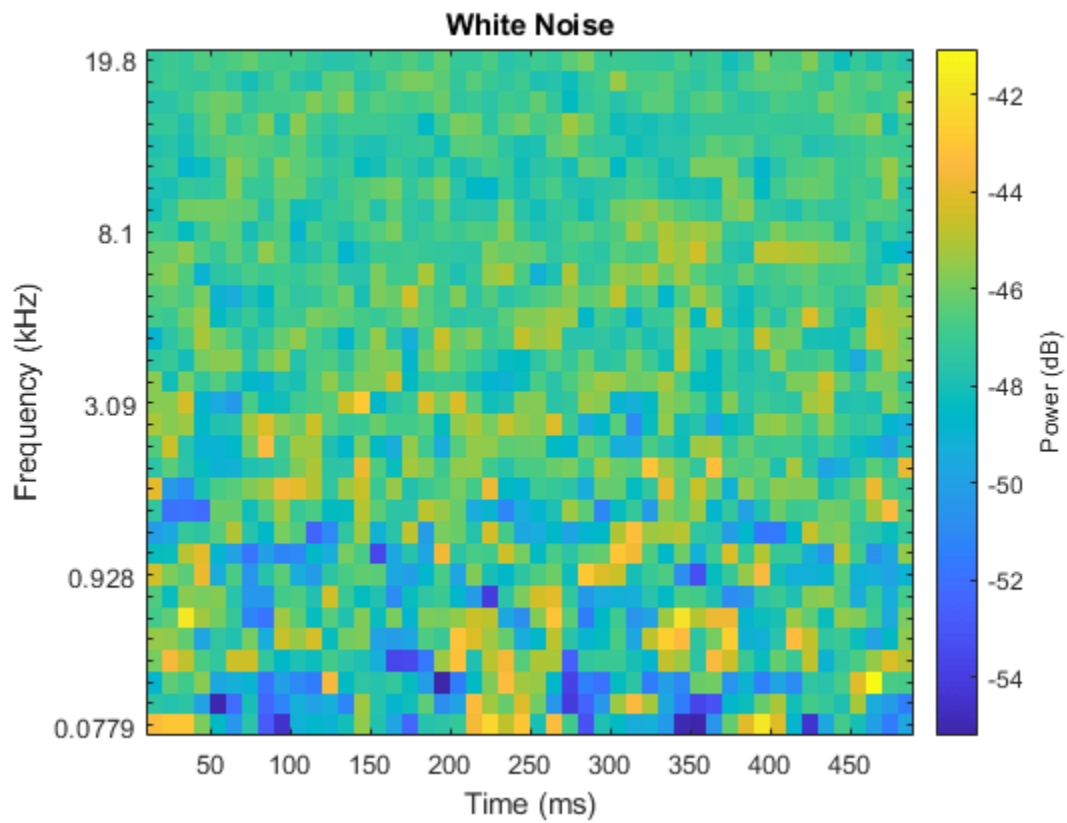
Inspect a brown noise signal.

```
sound(bNoise(:,1),fs)
melSpectrogram(bNoise(:,1),fs)
title('Brown Noise')
```
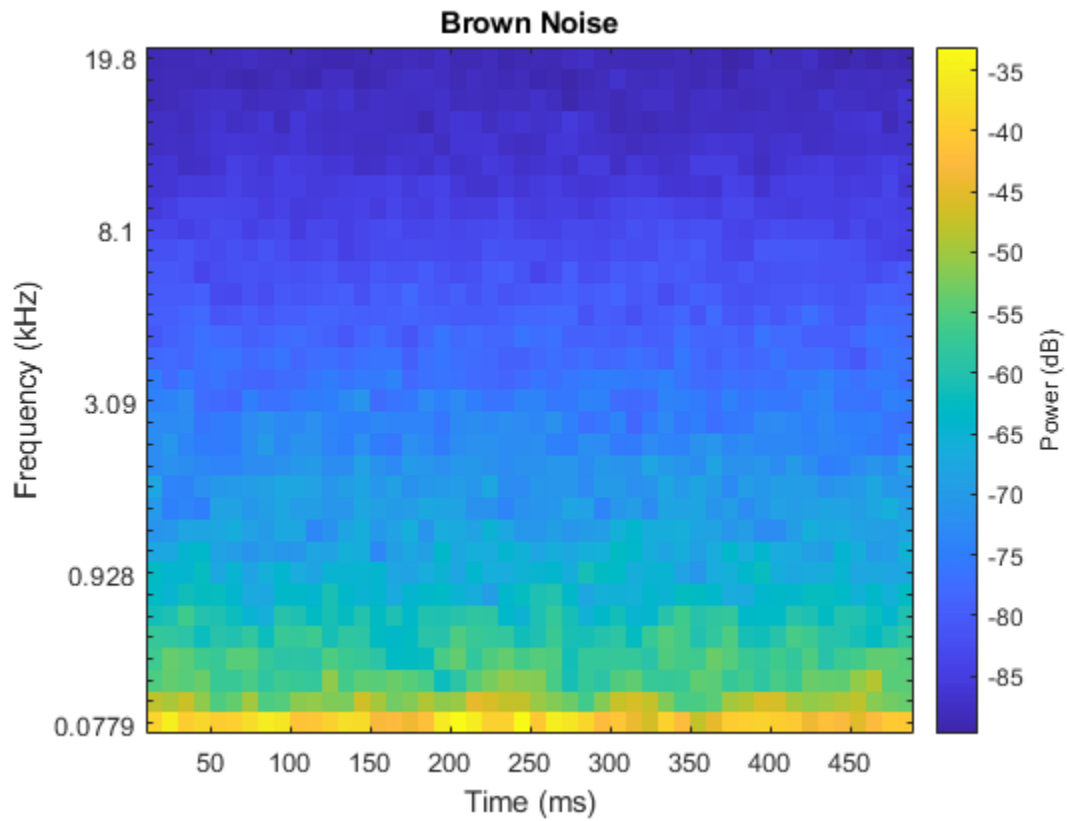
Inspect a pink noise signal.

```
sound(pNoise(:,1),fs)
melSpectrogram(pNoise(:,1),fs)
title('Pink Noise')
```
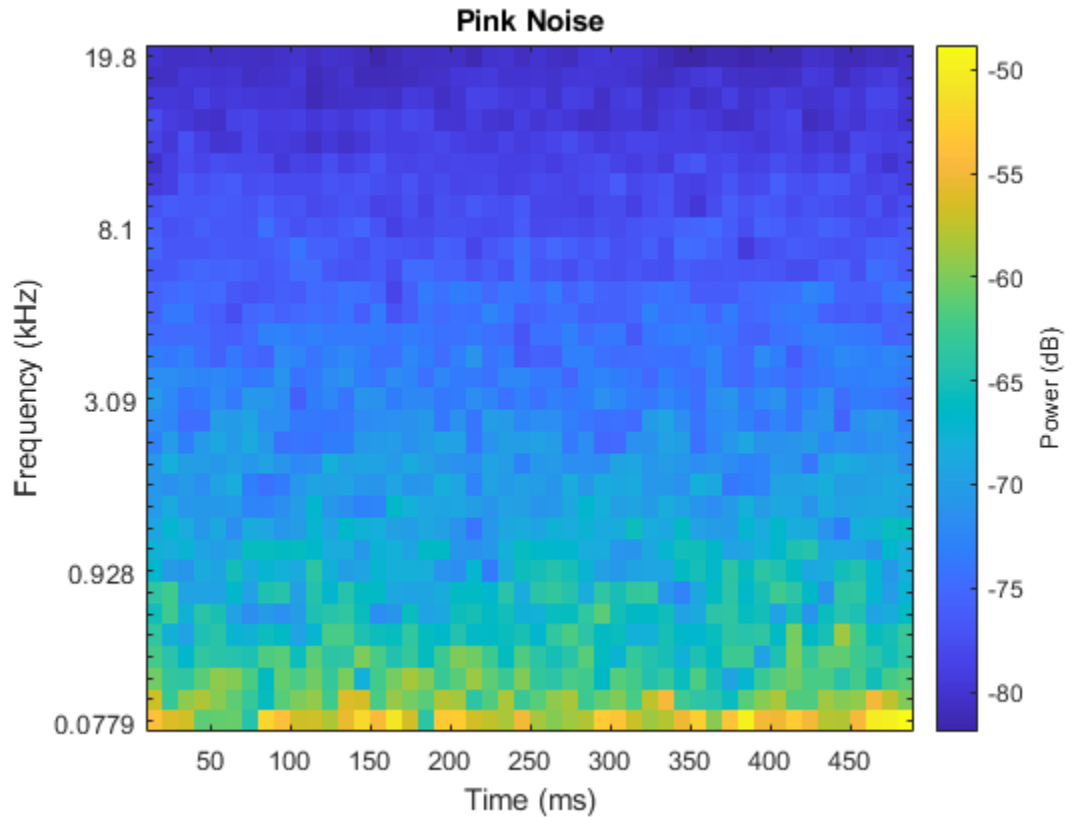
Pink Noise

Separate the Data Set into Train and Validation Sets

Create a training set that consists of 800 of the white noise signals, 800 of the brown noise signals, and 800 of the pink noise signals.

```
audioTrain = [wNoise(:,1:800),bNoise(:,1:800),pNoise(:,1:800)];
labelsTrain = [wLabels(1:800);bLabels(1:800);pLabels(1:800)];
```

Create a validation set using the remaining 200 white noise signals, 200 brown noise signals, and 200 pink noise signals.

```
audioValidation = [wNoise(:,801:end),bNoise(:,801:end),pNoise(:,801:end)];
labelsValidation = [wLabels(801:end);bLabels(801:end);pLabels(801:end)];
```

Extract Features

Audio data is highly dimensional and typically contains redundant information. You can reduce the dimensionality by first extracting features and then training your model using the extracted features. Create an `audioFeatureExtractor` object to extract the centroid and slope of the mel spectrum over time.

```
aFE = audioFeatureExtractor("SampleRate",fs, ...
    "SpectralDescriptorInput","melSpectrum", ...
    "spectralCentroid",true, ...
    "spectralSlope",true);
```

Call extract to extract the features from the audio training data.

```
featuresTrain = extract(aFE,audioTrain);
[numHopsPerSequence,numFeatures,numSignals] = size(featuresTrain)

numHopsPerSequence = 42

numFeatures = 2

numSignals = 2400
```

In the next step, you will treat the extracted features as sequences and use a `sequenceInputLayer` as the first layer of your deep learning model. When you use `sequenceInputLayer` as the first layer in a network, `trainNetwork` expects the training and validation data to be formatted in cell arrays of sequences, where each sequence consists of feature vectors over time. `sequenceInputLayer` requires the time dimension to be along the second dimension.

```
featuresTrain = permute(featuresTrain,[2,1,3]);
featuresTrain = squeeze(num2cell(featuresTrain,[1,2]));

numSignals = numel(featuresTrain)

numSignals = 2400

[numFeatures,numHopsPerSequence] = size(featuresTrain{1})

numFeatures = 2

numHopsPerSequence = 42
```

Extract the validation features.

```
featuresValidation = extract(aFE,audioValidation);
featuresValidation = permute(featuresValidation,[2,1,3]);
featuresValidation = squeeze(num2cell(featuresValidation,[1,2]));
```

Define and Train the Network

Define the network architecture. See "List of Deep Learning Layers" (Deep Learning Toolbox) for more information.

```
layers = [ ...
    sequenceInputLayer(numFeatures)
    lstmLayer(50,"OutputMode","last")
    fullyConnectedLayer(numel(unique(labelsTrain)))
    softmaxLayer
    classificationLayer];
```

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

```
options = trainingOptions("adam", ...
    "Shuffle","every-epoch", ...
    "ValidationData",{featuresValidation,labelsValidation}, ...
    "Plots","training-progress", ...
    "Verbose",false);
```

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

```
net = trainNetwork(featuresTrain,labelsTrain,layers,options);
```
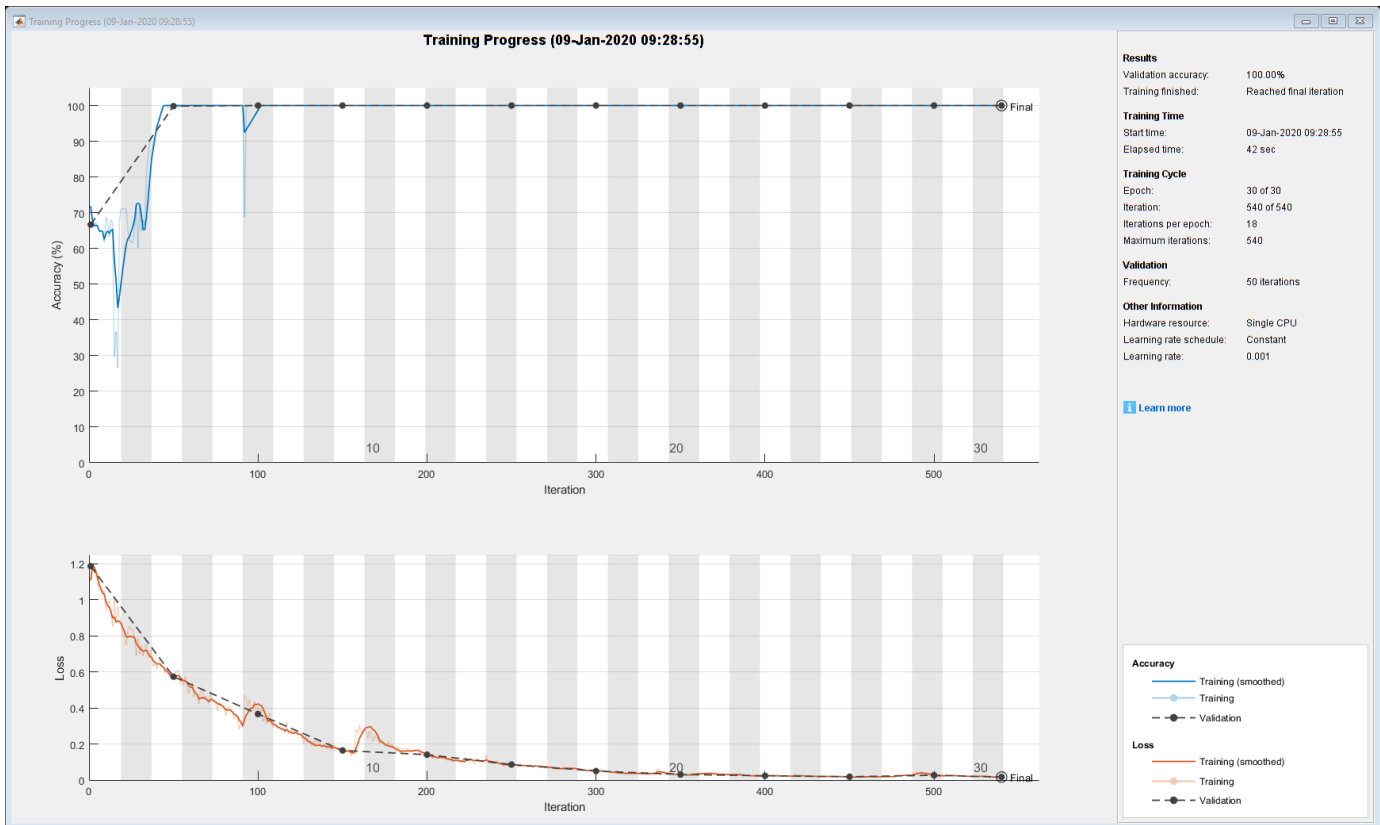
Test the Network

Use the trained network to classify new white noise, brown noise, and pink noise signals.

```
wNoiseTest = 2*rand([N,1]) - 1;
classify(net,extract(aFE,wNoiseTest)')
```

```
ans = categorical
      white
```

```
bNoiseTest = filter(1,[1,-0.999],wNoiseTest);
bNoiseTest= bNoiseTest./max(abs(bNoiseTest),[],'all');
classify(net,extract(aFE,bNoiseTest)')
```

```
ans = categorical
      brown
```

```
pNoiseTest = pinknoise(N);
classify(net,extract(aFE,pNoiseTest)')
```

```
ans = categorical
      pink
```

See Also

`audioFeatureExtractor` | `audioDataAugmenter` | `audioDatastore` | **Signal Labeler**

Related Examples

- "Keyword Spotting in Noise Using MFCC and LSTM Networks"
- "Acoustic Scene Recognition Using Late Fusion"
- "Spoken Digit Recognition with Wavelet Scattering and Deep Learning"
- "Voice Activity Detection in Noise Using Deep Learning"
- "Train Speech Command Recognition Model Using Deep Learning"
- "Denoise Speech Using Deep Learning Networks"
- "Speech Emotion Recognition"

Intro to Audio Deep Learning

Deep Learning for Audio Applications

Developing audio applications with deep learning typically includes creating and accessing data sets, preprocessing and exploring data, developing predictive models, and deploying and sharing applications. MATLAB provides toolboxes to support each stage of the development.
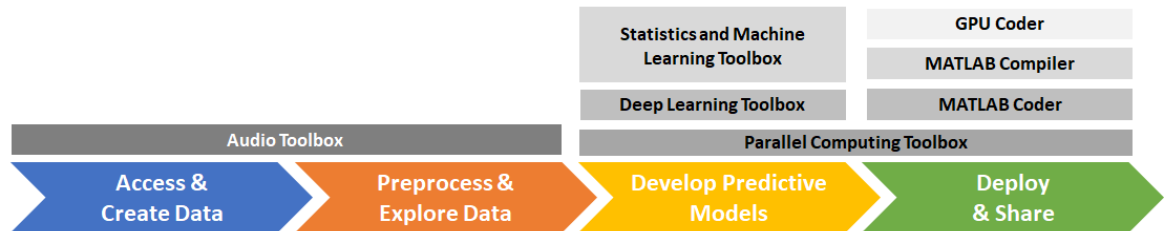
While Audio Toolbox supports each stage of the deep learning workflow, its principal contributions are to "Access and Create Data" on page 5-2 and "Preprocess and Explore Data" on page 5-3.
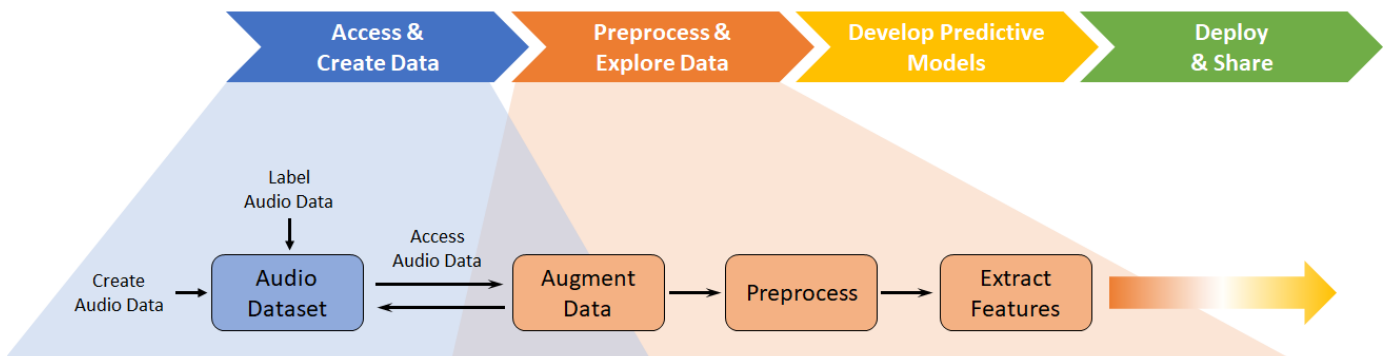
Access and Create Data

Deep learning networks perform best when you have access to large training data sets. However, the diversity of audio, speech, and acoustic signals, and a lack of large well-labeled data sets, makes accessing large training sets difficult. When using deep learning methods on audio files, you may need to develop new data sets or expand on existing ones. You can use the **Signal Labeler** app to help you enlarge or create new labeled data sets.

Once you have an initial data set, you can enlarge it by applying augmentation techniques such as pitch shifting, time shifting, volume control, and noise addition. The type of augmentation you want to apply depends on the relevant characteristics for your audio, speech, or acoustic application. For example, pitch shifting (or *vocal tract perturbation*) and time stretching are typical augmentation techniques for automatic speech recognition (ASR). For far-field ASR, augmenting the training data by using artificial reverberation is common. Audio Toolbox provides `audioDataAugmenter` to help you apply augmentations deterministically or probabilistically.

The training data used in deep learning workflows is typically too large to fit in memory. Accessing data efficiently and performing common deep learning tasks (such as splitting a data set into train, validation, and test sets) can quickly become unmanageable. Audio Toolbox provides `audioDatastore` to help you manage and load large data sets.

Preprocess and Explore Data

Preprocessing audio data includes tasks like resampling audio files to a consistent sample rate, removing regions of silence, and trimming audio to a consistent duration. You can accomplish these tasks by using MATLAB, Signal Processing Toolbox™, and DSP System Toolbox™. Audio Toolbox provides additional audio-specific tools to help you perform preprocessing, such as `detectSpeech` and `voiceActivityDetector`.

Audio is highly dimensional and contains redundant and often unnecessary information. Historically, mel-frequency cepstral coefficients (`mfcc`) and low-level features, such as the zero-crossing rate and spectral shape descriptors, have been the dominant features derived from audio signals for use in machine learning systems. Machine learning systems trained on these features are computationally efficient and typically require less training data. Audio Toolbox provides `audioFeatureExtractor` so that you can efficiently extract audio features.

Advances in deep learning architectures, increased access to computing power, and large and well-labeled data sets have decreased the reliance on hand-designed features. State-of-the-art results are often achieved using mel spectrograms (`melSpectrogram`), linear spectrograms, or raw audio waveforms. Audio Toolbox provides `audioFeatureExtractor` so that you can extract multiple auditory spectrograms, such as the mel spectrogram, gammatone spectrogram, or Bark spectrogram, and pair them with low-level descriptors. Using `audioFeatureExtractor` enables you to systematically determine audio features for your deep learning model. Alternatively, you can use the `melSpectrogram` function to quickly extract just the mel spectrogram. Audio Toolbox also provides the modified discrete cosine transform (`mdct`), which returns a compact spectral representation without any loss of information.

Example Applications and Workflows

Choosing features, deciding what kind of augmentations and preprocessing to apply, and designing a deep learning model all depend on the nature of the training data and the problem you want to solve. Audio Toolbox provides examples that illustrate deep learning workflows adapted to different data sets and audio applications. The table lists audio deep learning examples by network type (convolutional neural network, fully connected neural network, or recurrent neural network) and problem category (classification, regression, or sequence-to-sequence).

	CNN or FC	LSTM, BiLSTM, or GRU

Classification	Examples	Examples	Preprocessing and Augmentation	Feature Extraction and Time-Frequency Transformations
	"Train Speech Command Recognition Model Using Deep Learning"	"Speech Emotion Recognition"	`audioDataAugmenter`	The `audioFeatureExtractor` object is used to extract the `gtcc`, `mfcc`, and mel `spectralCrest`.
	"Acoustic Scene Recognition Using Late Fusion"	"Sequential Feature Selection for Audio Features"	`detectSpeech`	The `audioFeatureExtractor` object is used to sweep through combinations of extracted features.
	"Accelerate Audio Deep Learning Using GPU-Based Feature Extraction"			
		"Acoustics-Based Machine Fault Recognition"	Mix-up	The `audioFeatureExtractor` object is used to extract spectral descriptors.

Regression or Sequence-to-Sequence	Examples	Examples	Preprocessing and Augmentation	Feature Extraction and Time-Frequency Transformations
	"Denoise Speech Using Deep Learning Networks" "Cocktail Party Source Separation Using Deep Learning Networks" "Train Generative Adversarial Network (GAN) for Sound Synthesis" "End-to-End Deep Speech Separation"	"Voice Activity Detection in Noise Using Deep Learning"	`detectSpeech`	The `audioFeatureExtractor` object is used to extract the `spectralCentroid`, `spectralCrest`, `spectralEntropy`, `spectralFlux`, `spectralKurtosis`, `spectralSkewness`, `spectralRolloffPoint`, `spectralSlope`, and `harmonicRatio`.
		"Keyword Spotting in Noise Using MFCC and LSTM Networks"	`detectSpeech`, `audioDataAugmenter`	`mfcc`

References

[1] Purwins, H., B. Li, T. Virtanen, J. Schülter, S. Y. Chang, and T. Sainath. "Deep Learning for Audio Signal Processing." *Journal of Selected Topics of Signal Processing*. Vol. 13, Issue 2, 2019, pp. 206–219.

See Also

audioFeatureExtractor | audioDataAugmenter | audioDatastore | **Signal Labeler**

Related Examples

- "Classify Sound Using Deep Learning" on page 4-2
- "Create Simple Sequence Classification Network Using Deep Network Designer" (Deep Learning Toolbox)

- "Get Started with Deep Network Designer" (Deep Learning Toolbox)

Stream Audio

Process and Analyze Streaming Audio

This example shows how to create an audio test bench and apply real-time processing.

Open the Audio Test Bench

The Audio Test Bench app enables you to graphically set up your audio input and output, audio processing, and open common analysis tools like `timescope` and `spectrumAnalyzer`. Click ▶ to read from a file and write to your speaker.

`audioTestBench`
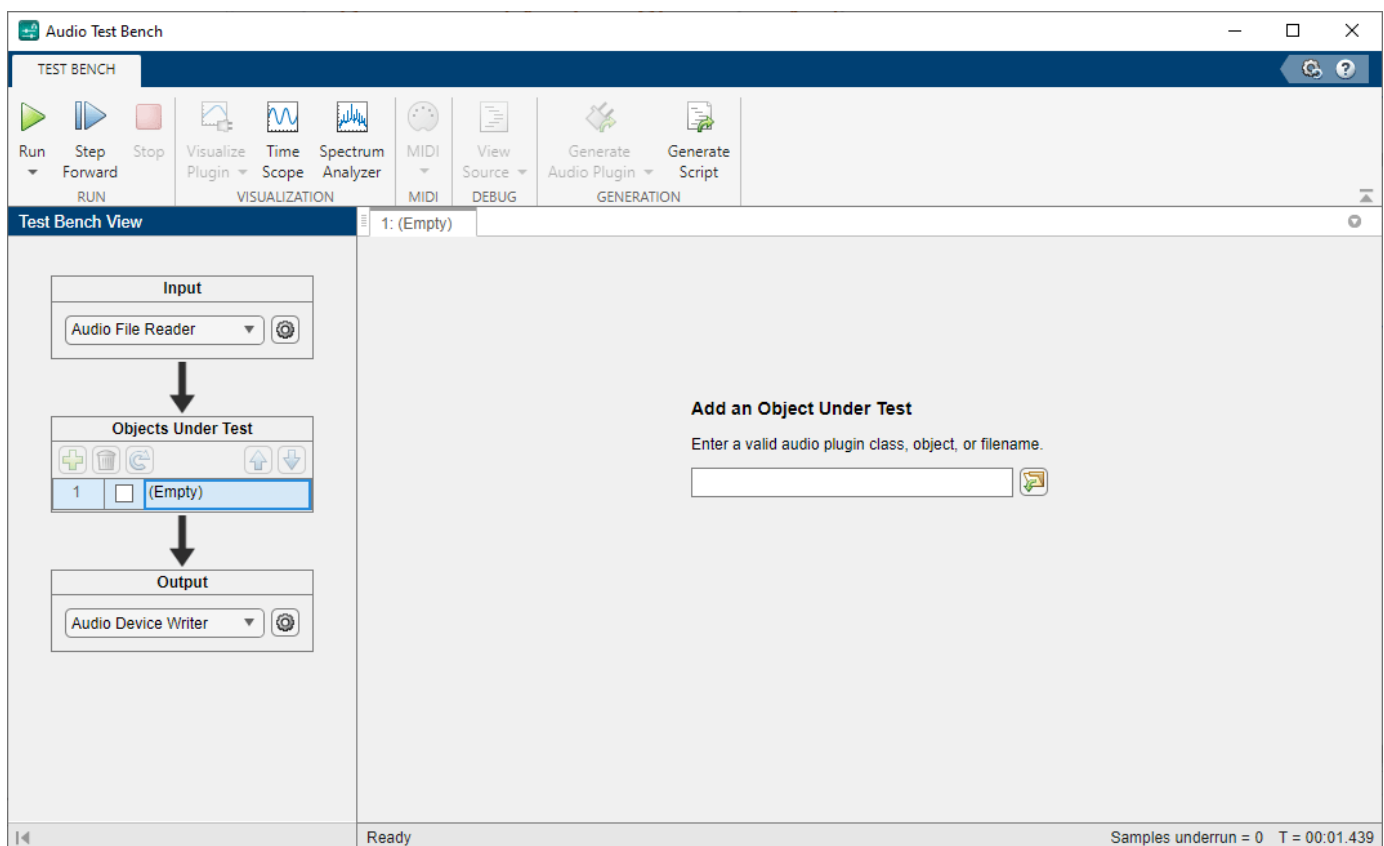
View the Audio Signal in the Time and Frequency Domains

Click and to analyze the audio signal in the time and frequency domains.
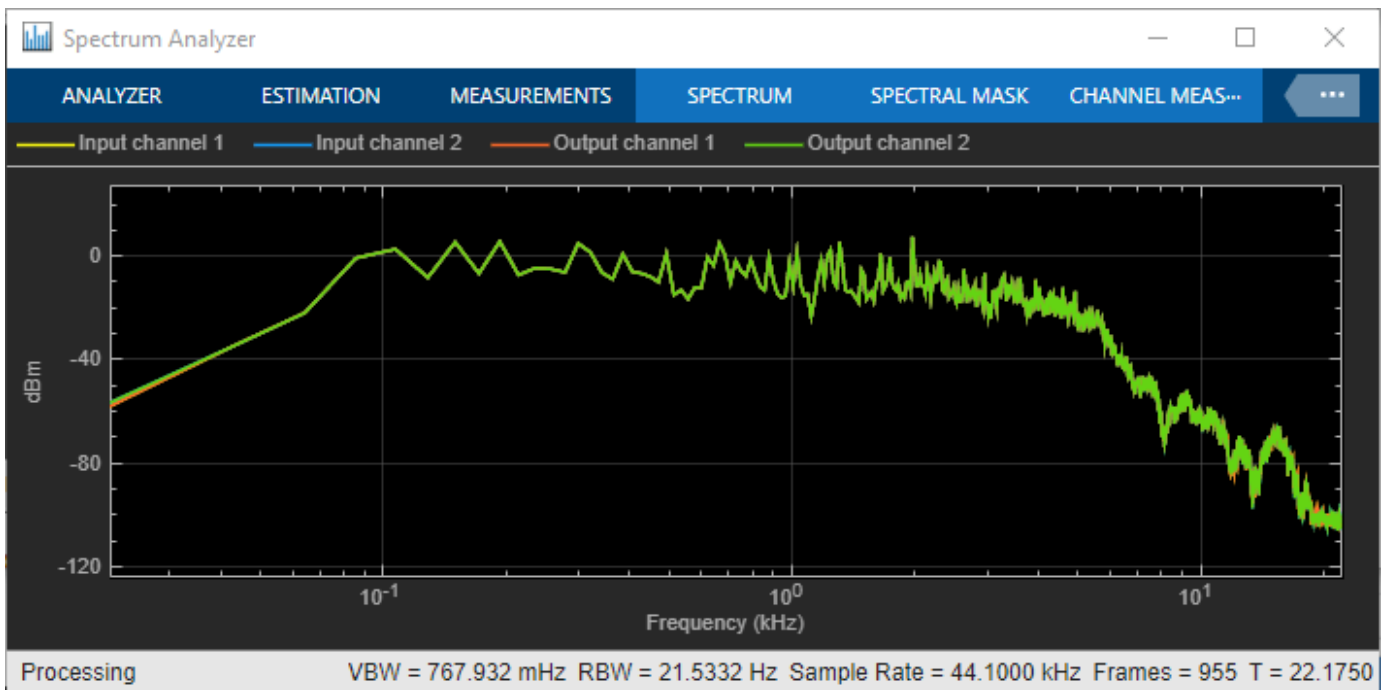
Apply Dynamic Range Compression

To apply dynamic range compression to the audio, first click 🔴 to stop the audio I/O, then enter `compressor` in the **Add an Object Under Test** edit box. The tunable properties of the `compressor` object are exposed. You can tune these properties while the test bench runs.

Generate a Test Bench Script

To generate a test bench script, first click [red stop button] to stop the audio I/O, then click [script icon]. The **Audio Test Bench** generates code in a new untitled script. The code generated by the test bench in this example is shown below.

```
% Test bench script for 'compressor'.
% Generated by Audio Test Bench on 06-Jul-2022 11:47:07 UTC-04:00.

% Create test bench input and output
fileReader = dsp.AudioFileReader('Filename','RockGuitar-16-44p1-stereo-72secs.wav');
deviceWriter = audioDeviceWriter('SampleRate',fileReader.SampleRate);

% Create scopes
timeScope = timescope(SampleRate=fileReader.SampleRate, ...
    TimeSpanSource="property", ...
    TimeSpan=1, ...
    AxesScaling="manual", ...
    BufferLength=176400, ...
    ChannelNames=["Input channel 1","Input channel 2","Output channel 1","Output channel 2"], ..
    Position=[538 508 800 500], ...
    YLimits=[-1 1], ...
    ShowLegend=true);

specScope = spectrumAnalyzer(SampleRate=fileReader.SampleRate, ...
    PlotAsTwoSidedSpectrum=false, ...
    FrequencyScale="log", ...
    AxesScaling="manual", ...
    AveragingMethod="exponential", ...
    ForgettingFactor=0.8, ...
    YLimits=[-132.83298112 25.39711835], ...
    ChannelNames=["Input channel 1","Input channel 2","Output channel 1","Output channel 2"], ..
```

6-5

```matlab
        ShowLegend=true);


    % Set up the system under test
    sut = compressor;
    sut.SampleRate = fileReader.SampleRate;

    % Uncomment to open visualizer:
    % visualize(sut);

    % Open parameterTuner for interactive tuning during simulation
    tuner = parameterTuner(sut);
    drawnow

    % Stream processing loop
    nUnderruns = 0;
    while ~isDone(fileReader)
        % Read from input, process, and write to output
        in = fileReader();
        out = sut(in);
        nUnderruns = nUnderruns + deviceWriter(out);

        % Visualize input and output data in scopes
        timeScope(in,out);
        specScope(in,out);

        % Process parameterTuner callbacks
        drawnow limitrate
    end

    % Clean up
    release(sut)
    release(fileReader)
    release(deviceWriter)
    release(timeScope)
    release(specScope)
```

You can add additional processing steps, scopes, and analysis tools to the script. If you run the generated script, the `parameterTuner` opens and enables you to tune parameters while stream processing.
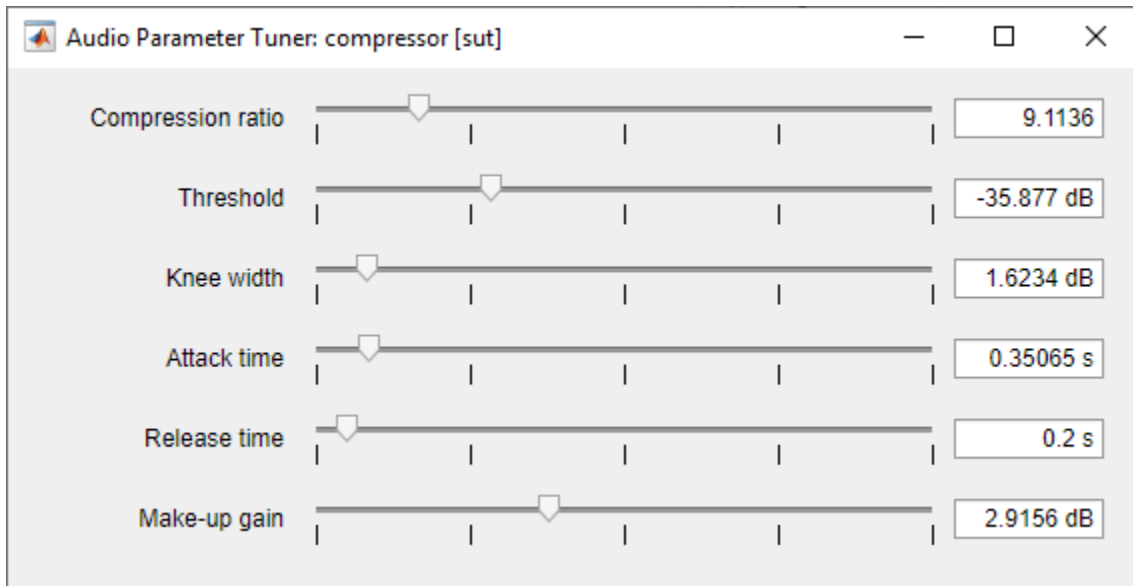
See Also

Audio Test Bench | parameterTuner | audioPlayerRecorder | dsp.AudioFileWriter | dsp.AudioFileReader | audioDeviceReader | audioDeviceWriter

More About

- "Audio Input and Audio Output" on page 2-2
- "Real-Time Audio in Simulink" on page 12-2
- "Audio I/O: Buffering, Latency, and Throughput" on page 8-2

Export a MATLAB Plugin to a DAW

Export a MATLAB Plugin to a DAW

In this section...
"Plugin Development Workflow" on page 7-2
"Considerations When Generating Audio Plugins" on page 7-2
"How Audio Plugins Interact with the DAW Environment" on page 7-2

Audio Toolbox enables generation of VST plugins from MATLAB source code by using the `generateAudioPlugin` function. The generated plugin is compatible with 32-bit and 64-bit Windows, and 64-bit Mac host applications. After you generate a VST plugin, you can use your generated audio plugin in a digital audio workstation (DAW).

Plugin Development Workflow

1   Design an audio plugin. For a tutorial on audio plugin architecture and design in the MATLAB environment, See "Audio Plugins in MATLAB" on page 11-2.

2   Validate your audio plugin using the `validateAudioPlugin` function.

   `validateAudioPlugin myAudioPlugin`

3   Test your audio plugin using **Audio Test Bench**.

   `audioTestBench myAudioPlugin`

4   Generate your audio plugin using the `generateAudioPlugin` function.

   `generateAudioPlugin myAudioPlugin`

5   Use your generated audio plugin in a DAW.

Considerations When Generating Audio Plugins

* Your plugin must be compatible with MATLAB code generation. See "MATLAB Programming for Code Generation" (MATLAB Coder) for more information.
* Your generated plugin must be compatible with DAW environments. The DAW environment:

  * Determines the sample rate and frame size at which a plugin is run, both of which are variable.
  * Calls the reset function of your plugin at the beginning of each use and if the sample rate changes.
  * Requires a consistent input and output frame size for the plugin processing function.
  * Must be synchronized with plugin parameters. Therefore, a plugin must not modify properties associated with parameters.
  * Requires that plugin properties associated with parameters are scalar values.

Use the `validateAudioPlugin`, `Audio Test Bench`, and `generateAudioPlugin` tools to guide your audio plugin into a valid form capable of generation.

How Audio Plugins Interact with the DAW Environment

After you generate your plugin, plug it into a DAW environment. See documentation on your specific DAW for details on adding plugins.

The audio plugin in the DAW environment interacts primarily through the processing function, reset function, and interface properties of your plugin.

Initialization and Reset

- The DAW environment calls the reset function of the plugin the first time the plugin is used, or any time the sample rate of the DAW environment is modified. You can use the `getSampleRate` function to query the sample rate of the environment.

Processing

- The DAW environment passes a frame of an audio signal to the plugin. The DAW determines the frame size. If the audio plugin is a source audio plugin, the DAW does not pass an input audio signal.
- The processing function of your plugin performs the frame-based audio processing algorithm you specified, and updates internal plugin properties as needed. Plugins must not write to properties associated with parameters.
- The processing function of your plugin passes the processed audio signal out to the DAW environment. The frame size of the output signal must match the frame size of the input signal. If the audio plugin is a source audio plugin, you must use `getSamplesPerFrame` to determine the output frame size. Because the environment frame rate is variable, you must call `getSamplesPerFrame` for each output frame.
- Processing is performed iteratively frame by frame on an audio signal.

Tunability

- If you modify a parameter through the plugin dialog box, the synchronized public property updates at that time. You can use the `set` method of MATLAB classes to modify private properties.
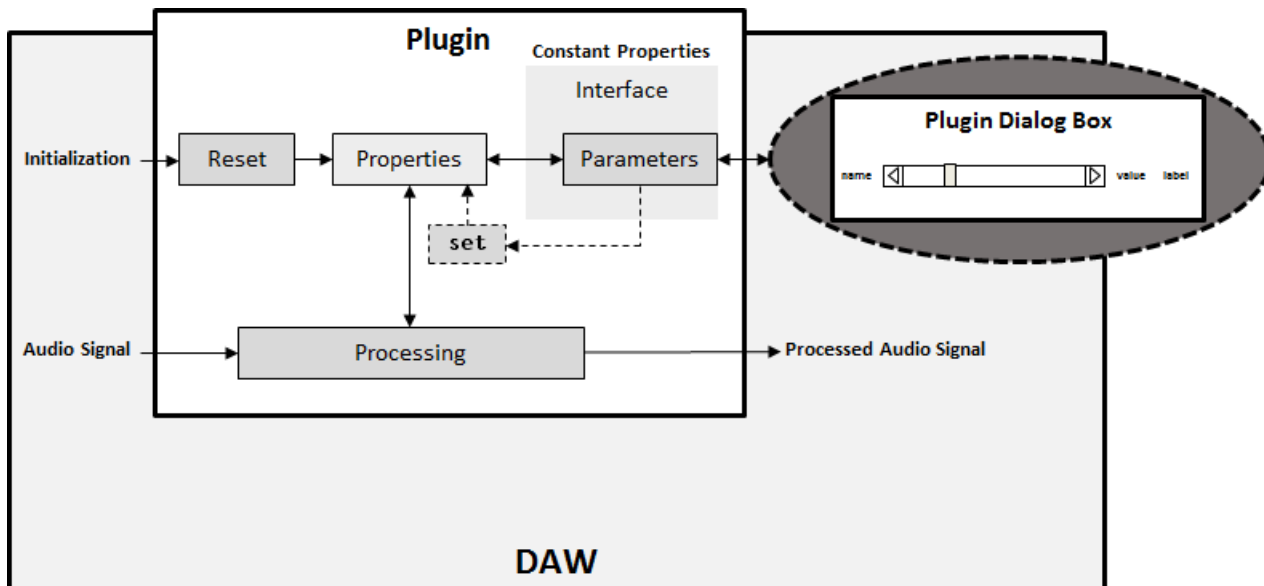
See Also

More About
- "What Are DAWs, Audio Plugins, and MIDI Controllers?" on page 9-2
- "Audio Plugins in MATLAB" on page 11-2
- "Convert MATLAB Code to an Audio Plugin" on page 13-2

Audio I/O: Buffering, Latency, and Throughput

Audio I/O: Buffering, Latency, and Throughput

Audio Toolbox is optimized for real-time stream processing. Its input and output System objects are efficient, low-latency, and they control all necessary parameters so that you can trade off between throughput and latency.

This tutorial describes how MATLAB software implements real-time stream processing. The tutorial presents key terminology and basic techniques for optimizing your stream-processing algorithm. For more detailed technical descriptions and concepts, see the documentation for the audio I/O System objects used in this tutorial.

The concepts presented in this tutorial are described in terms of System objects in the MATLAB environment. The same concepts can be applied to corresponding blocks in the Simulink® environment.

Input Audio Stream

To acquire an audio stream from a file, use the `dsp.AudioFileReader` System object™. To acquire an audio stream from a device, use the `audioDeviceReader` System object.

This diagram and the description that follows indicate the data flow when acquiring a monochannel signal with the `audioDeviceReader` System object.
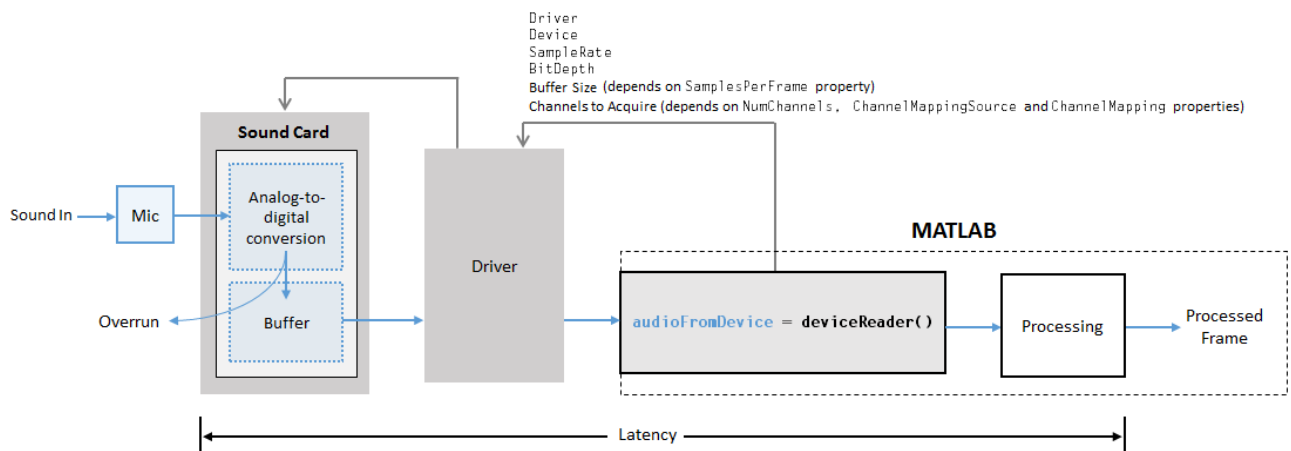
Configuration

- Properties of your `audioDeviceReader` specify the driver, device (sound card), sample rate, bit depth, buffer size, and channel mapping between your device's input channels and columns output from your `audioDeviceReader` object. Your object communicates these specifications to the driver once at `setup`.

Real-Time Processing Loop

1   The microphone picks up the sound and sends a continuous electrical signal to your sound card.
2   The sound card performs analog-to-digital conversion at a sample rate, buffer size, and bit depth specified during configuration.

3   The analog-to-digital converter writes audio samples into the sound card buffer. If the buffer is full, the new samples are dropped. These samples are referred to as overruns.

4   The `audioDeviceReader` uses the driver to pull the oldest frame from the sound card buffer iteratively.

Output Audio Stream

To send an audio stream to a file, use the `dsp.AudioFileWriter` System object. To send an audio stream to a device, use the `audioDeviceWriter` System object.

This diagram and the description that follows indicate the data flow when playing a monochannel signal with the `audioDeviceWriter` System object.
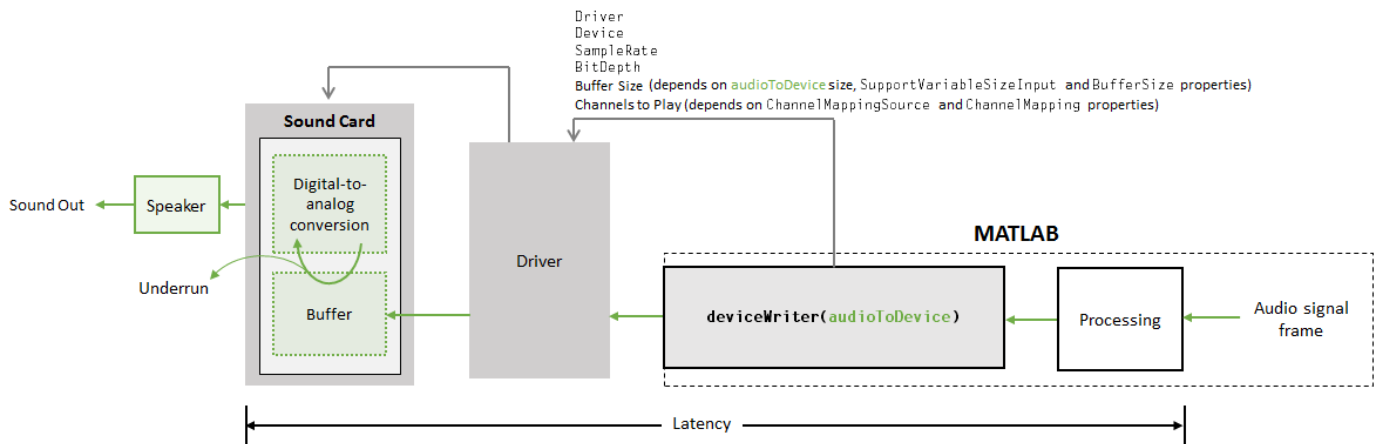
Configuration

- Properties of your `audioDeviceWriter` specify the driver, device (sound card), sample rate, bit depth, buffer size, and channel mapping between your device's output channels and columns input to your `audioDeviceWriter` object. Your object communicates these specifications to the driver once at `setup`.

Real-Time Processing Loop

1   The processing stage passes a frame of variable length to the `audioDeviceWriter` System object.

2   `audioDeviceWriter` sends the frame to the sound card's buffer.

3   The sound card pulls the oldest frame from the buffer and performs digital-to-analog conversion. The sound card sends the analog chunk to the speaker. If the buffer is empty when the sound card tries to pull from it, the sound card outputs a region of silence. This is referred to as underrun.

Synchronize Audio to and from Device

To simultaneously read from and write to a single audio device, use the `audioPlayerRecorder` System object.

This diagram and the description that follows indicate the data flow when playing and recording monochannel signals with the `audioPlayerRecorder` System object.
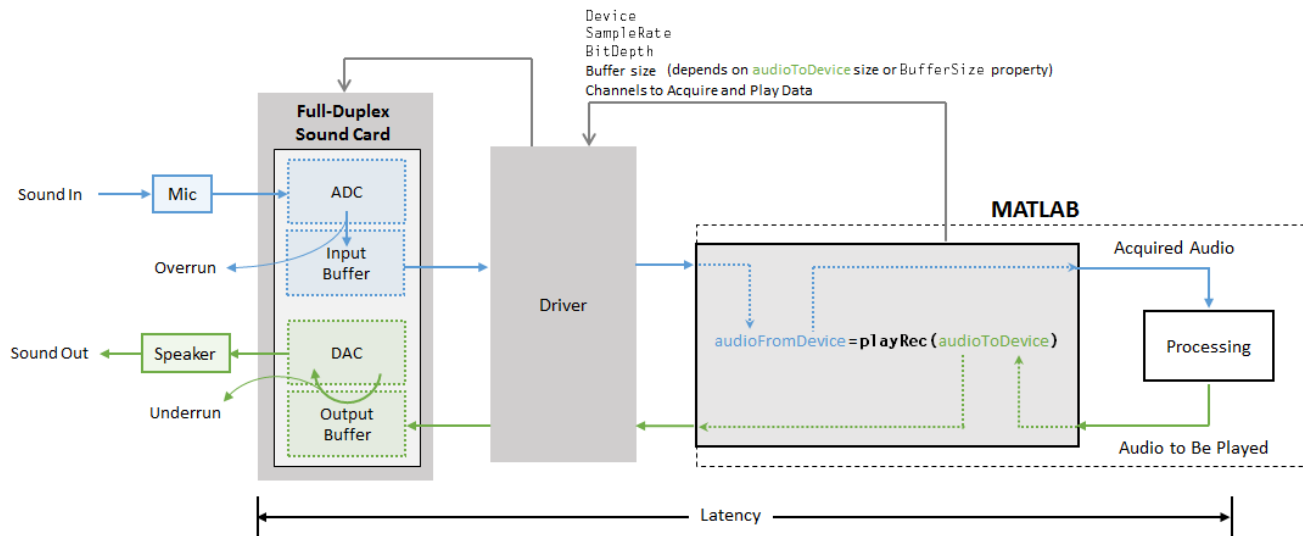
Configuration

- Properties of your `audioPlayerRecorder` specify the device (sound card), sample rate, bit depth, buffer size, and channel mapping between your device and object. Your object communicates these specifications to the driver once at `setup`.

Real-Time Processing Loop

1. The microphone picks up the sound and sends a continuous electrical signal to your sound card. Simultaneously, the speaker plays an analog chunk received from the sound card.

2. The sound card performs analog-to-digital conversion of the acquired audio signal and writes the digital chunk to the input buffer. If the input buffer is full, the new samples are dropped. Simultaneously, the sound card pulls the oldest frame from the output buffer and performs digital-to-analog conversion of the next audio chunk to be played. If the output buffer is empty when the sound card tries to retrieve the data, the sound card outputs a region of silence.

3. The `audioPlayerRecorder` object returns the acquired audio signal to the MATLAB environment for processing. Simultaneously, the audio to be played is specified as an argument of the `audioPlayerRecorder` for playback in the next I/O cycle.

Terminology and Techniques to Optimize Performance

Signal Drops

- *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. The number of samples underrun is returned when you call your `audioPlayerRecorder` or `audioDeviceWriter` object.

- *Overrun* refers to input signal drops. Input signal drops occur when the processing stage does not keep pace with the acquisition of samples. The number of samples overrun is returned when you call your `audioPlayerRecorder` or `audioDeviceReader` object.

If you encounter overrun or underrun, try improving your I/O system in one or more of the following ways:

1   Identify when the overrun or underrun occurs. If it occurs in the first few iterations, consider calling `setupImpl` on your System objects before the loop where real-time processing is required. You can also run the I/O system with dummy data for a few frames before starting the real processing. For more information, see "Measure Performance of Streaming Real-Time Audio Algorithms".

2   If you are using a DirectSound driver on a Windows® platform, consider switching to a WASAPI or ASIO driver. ASIO drivers have the least overhead. If you are using an ASIO driver, make sure to match the frame size in MATLAB to the ASIO buffer size. You can use `asiosettings` to open the ASIO preferences UI from MATLAB.

3   If you can afford to add more latency to your application, consider increasing the buffer size of your object. By default, the buffer size is the frame size of the data processed by the audio object.

4   If you can afford to decrease signal resolution, consider decreasing the sample rate.

5   Close all nonessential processes on your machine, such as mail checkers and file sync utilities. These processes can asynchronously ask for CPU time through interrupts and disturb the audio-processing loop.

6   To maximize performance, remove all plotting and visualization from your real-time loop. If you require a visualization to update in your processing loop, use a DSP System Toolbox scope such as `timescope`, `spectrumAnalyzer`, or `dsp.ArrayPlot`. Follow the recommendations listed in point 1 to setup and pre-run your scopes. If you require custom graphics or are processing callbacks in the loop, use the `drawnow` command and specify a limited update rate to optimize your event queue.

7   If the processing loop is algorithm heavy, try profiling your loop to locate the bottlenecks, and then apply appropriate measures:

   - Replace handwritten code with MATLAB features that have been optimized for speed.

   - Follow best practices for performance: "Techniques to Improve Performance".

   - Generating MATLAB executables (MEX files) using MATLAB Coder™ may result in faster execution. See "Remove Interfering Tone From Audio Stream" for an example.

     You can also generate standalone executables (EXE files). See "Generate Standalone Executable for Parametric Audio Equalizer" for an example.

   - If you are considering turning your algorithm into a VST plugin, then try running it as a VST plugin within MATLAB. VST plugin generation uses C code generation technology under the hood, and running the generated VST plugin within MATLAB may result in faster execution than with your original MATLAB code. See "Audio Plugins in MATLAB" on page 11-2 and "Host External Audio Plugins" on page 14-2 to learn how to design, generate, and then host a VST plugin.

Latency

- *Output latency* is measured as the time delay between the time of generation of an audio frame in MATLAB and the time that audio is heard through the speaker.

- *Input latency* is measured as the time delay between the time that audio enters the sound card and the time that the frame is output by the processing stage.

If properties and frame size remain consistent, the ratio of *input latency* to *output latency* is consistent between calls to an `audioPlayerRecorder` object.

To minimize latency, you can:

1   Optimize the processing stage. If your processing stage has reached a peak algorithmically, compiling your MATLAB code into C code using MATLAB Coder may result in faster execution.
2   Increase the sample rate.
3   Decrease the frame size.

For a tutorial on measuring the round-trip latency of your system, see "Measure Audio Latency".

See Also
asiosettings | getAudioDevices | dsp.AudioFileReader | dsp.AudioFileWriter | audioDeviceReader | audioDeviceWriter | audioPlayerRecorder | Audio Device Reader | Audio Device Writer | From Multimedia File | To Multimedia File

More About
*   "Real-Time Audio in MATLAB" on page 10-2
*   "Real-Time Audio in Simulink" on page 12-2

What Are DAWs, Audio Plugins, and MIDI Controllers?

What Are DAWs, Audio Plugins, and MIDI Controllers?

In this section...
"Digital Audio Workstation (DAW)" on page 9-2
"Audio Plugins" on page 9-2
"Musical Instrument Digital Interface (MIDI)" on page 9-2

Digital Audio Workstation (DAW)

A digital audio workstation (DAW) is an electronic device or software application used to record, edit, and produce sound files. DAWs are controlled with a user interface. Most DAWs allow MIDI controls to tune parameters during live editing.

In the music industry, DAWs are typically used to acquire and save multiple tracks of audio recordings, and to mix, equalize, and add audio effects. DAWs generally have access to libraries of sounds and are used to create electronic music from scratch. Commercial DAWs, such as those found in recording studios, can be hardware integrated into computers.

DAWs are also used in the production of radio, television, film, podcasts, games, and anywhere complex manipulation of audio signals is needed.

DAWs generally support plugins, which are smaller pieces of software with unique functionality, therefore expanding the abilities of the DAW user.

Audio Plugins

Plugins are self-contained pieces of code that can be "plugged in" to DAWs to enhance their functionality. Generally, plugins fall into the categories of audio signal processing, analysis, or sound synthesis. Plugins usually specify a user-interface containing UI widgets, but the DAW interface might mask it. Typical plugins include equalization, dynamic range control, reverberation, delay, and virtual instruments.

To process streaming audio data, the DAW calls the plugin, passes in a frame of input audio data, and receives back a frame of processed output audio data. When a plugin parameter changes (for example, when you move a control on the plugin's UI), the DAW notifies the plugin of the new parameter value. Plugins usually have their own custom UI, but DAWs also provide a generic UI for all plugins.

Audio Toolbox supports code generation to the most common plugin format, Steinberg's VST (Virtual Studio Technology). Audio Toolbox also enables you to run and test externally authored VST and VST3 plugins directly in MATLAB.

For a discussion of plugin terminology and usage in the MATLAB environment, see "Audio Plugins in MATLAB" on page 11-2.

Musical Instrument Digital Interface (MIDI)

Musical Instrument Digital Interface (MIDI) is a technical standard for communication between electronic instruments, computers, and related devices. MIDI carries event messages specific to

audio signals, such as pitch and velocity, as well as control signals for parameters such as volume, vibrato, panning, cues, and clock signals to synchronize tempo.

MIDI controllers are devices that send MIDI messages. Common devices include electronic keyboards or surfaces with sliders, knobs, and buttons. For DAWs, MIDI controllers can be physical instantiations of functionality present in the DAW. The DAW user can interact using a keyboard and mouse and MIDI controllers.

See Also

More About

- "Audio Plugins in MATLAB" on page 11-2
- "Host External Audio Plugins" on page 14-2
- "Audio Plugin Example Gallery"
- "MIDI Control Surface Interface"
- "MIDI Control for Audio Plugins"

External Websites

- MIDI Manufacturers Association

Real-Time Audio in MATLAB

Real-Time Audio in MATLAB

In this section...
"Create a Development Test Bench" on page 10-2
"Add Tunability" on page 10-6
"Quick Start Example" on page 10-7

Audio Toolbox is optimized for real-time audio processing. `audioDeviceReader`, `audioDeviceWriter`, `audioPlayerRecorder`, `dsp.AudioFileReader`, and `dsp.AudioFileWriter` are designed for streaming multichannel audio, and they provide necessary parameters so that you can trade off between throughput and latency.

For information on real-time processing and tips on how to optimize your algorithm, see "Audio I/O: Buffering, Latency, and Throughput" on page 8-2.

This tutorial describes how you can implement audio stream processing in MATLAB. It outlines the workflow for creating a development test bench and provides examples for each stage of the workflow.

Create a Development Test Bench

This tutorial creates a development test bench in four steps:

1 Build objects to input and output audio from your test bench.
2 Create an audio stream loop that processes your audio frame-by-frame.
3 Add a scope to visualize both the input and output of your audio stream loop.
4 Add a processing algorithm for your audio stream loop.

This tutorial also discusses tools for visualizing and tuning your processing algorithm in real time.
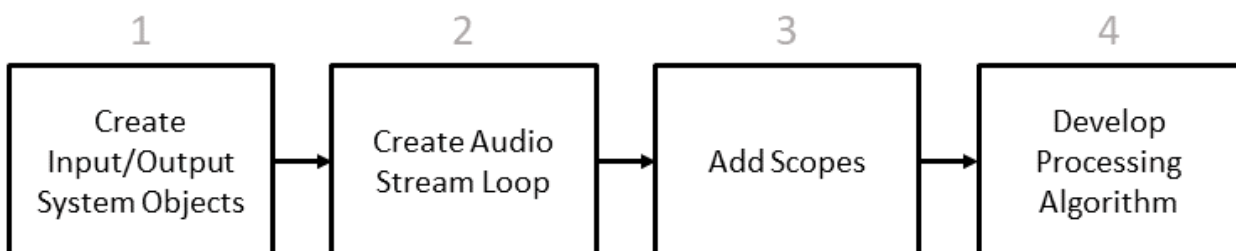
For an overview of the processing loop, consider the completed test bench below. You can recreate this test bench by walking step-by-step through this tutorial.

Anatomy of a Test Bench

Create objects for audio input/output

```
frameLength = 1024;

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

Create object for visualization

```
scope = dsp.TimeScope( ...
    'SampleRate',fileReader.SampleRate, ...
    'TimeSpan',2, ...
    'BufferLength',fileReader.SampleRate*2*2, ...
    'YLimits',[-1,1], ...
    'TimeSpanOverrunAction',"Scroll");
```

Create object for audio processing

```
reverb = reverberator( ...
    'SampleRate',fileReader.SampleRate, ...
    'PreDelay',0.5, ...
    'WetDryMix',0.4);
```

Audio stream loop

```
while ~isDone(fileReader)
    signal = fileReader();
    reverbSignal = reverb(signal);
    deviceWriter(reverbSignal);
    scope([signal,mean(reverbSignal,2)])
end
```

Input signal frame into audio stream loop

Process signal frame in audio stream loop

Output processed signal frame from audio stream loop

Visualize processed and unprocessed signal frame

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

Completed Test Bench Code

Click here to open the file.

```
frameLength = 1024;
fileReader = dsp.AudioFileReader( ...
    'Counting-16-44p1-mono-15secs.wav', ...
    'SamplesPerFrame',frameLength);
deviceWriter = audioDeviceWriter( ...
    'SampleRate',fileReader.SampleRate);

scope = timescope( ...
    'SampleRate',fileReader.SampleRate, ...
    'TimeSpan',2, ...
    'BufferLength',fileReader.SampleRate*2*2, ...
    'YLimits',[-1,1], ...
    'TimeSpanOverrunAction',"Scroll");

reverb = reverberator( ...
```

```
    'SampleRate',fileReader.SampleRate, ...
    'PreDelay',0.5, ...
    'WetDryMix',0.4);

while ~isDone(fileReader)
    signal = fileReader();
    reverbSignal = reverb(signal);
    deviceWriter(reverbSignal);
    scope([signal,mean(reverbSignal,2)])
end

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

1. Create Input/Output System objects

Your audio stream loop can read from a device or a file, and it can write to a device or a file. In this example, you build an audio stream loop that reads audio frame-by-frame from a file and writes audio frame-by-frame to a device. See "Quick Start Example" on page 10-7 for an alternative input/output configuration.

Create a `dsp.AudioFileReader` System object and specify a file. To reduce latency, set the `SamplesPerFrame` property of the `dsp.AudioFileReader` System object to a small frame size.

Next, create an `audioDeviceWriter` System object and specify its sample rate as the sample rate of the file reader.

For more information on how to use System objects, see "What Are System Objects?"

View Example Code

```
frameLength = 1024;
fileReader = dsp.AudioFileReader( ...
    'Counting-16-44p1-mono-15secs.wav', ...
    'SamplesPerFrame',frameLength);
deviceWriter = audioDeviceWriter( ...
    'SampleRate',fileReader.SampleRate);
```

2. Create Audio Stream Loop

An audio stream loop processes audio iteratively. It does so by:

- Reading a frame of an audio signal
- Processing that frame of audio signal
- Writing that frame of audio signal to a device or file
- Moving to the next frame

In this tutorial, the input to the audio stream loop is read from a file. The output is written to a device.

To read an audio file frame-by-frame, call your `dsp.AudioFileReader` within your audio stream loop, and provide no arguments. To write an audio signal frame-by-frame, call your `audioDeviceWriter` within your audio stream loop with an audio signal as an argument.

View Example Code

```matlab
frameLength = 1024;

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

while ~isDone(fileReader)                    %<--- new lines of code
    signal = fileReader();                   %<---
    deviceWriter(signal);                    %<---
end                                          %<---

release(fileReader)                          %<---
release(deviceWriter)                        %<---
```

All System objects have a `release` function. As a best practice, release your System objects after use, especially if those System objects are communicating with hardware devices such as sound cards.

3. Add Scopes

There are several scopes available. Two common scopes are the `timescope` and the `spectrumAnalyzer`. This tutorial uses `timescope` to visualize the audio signal.

The `timescope` System object displays an audio signal in the time domain. Create the System object. To aid visualization, specify values for the `TimeSpan`, `BufferLength`, and `YLimits` properties. To visualize an audio signal frame-by-frame, call the `timescope` System object within your audio stream loop with an audio signal as an argument.

View Example Code

```matlab
frameLength = 1024;

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

scope = timescope( ...                              %<--- new lines of code
    'SampleRate',fileReader.SampleRate, ...         %<---
    'TimeSpan',2, ...                               %<---
    'BufferLength',fileReader.SampleRate*2*2, ...   %<---
    'YLimits',[-1,1], ...                           %<---
    'TimeSpanOverrunAction',"Scroll");              %<---

while ~isDone(fileReader)
    signal = fileReader();
    deviceWriter(signal);
    scope(signal)                                   %<---
end

release(fileReader)
release(deviceWriter)
release(scope)                                      %<---
```

4. Develop Processing Algorithm

In most applications, you want to process your audio signal within your audio stream loop. The processing stage can be:

- A block of MATLAB code within your audio stream loop
- A separate function called within your audio stream loop
- A System object called within your audio stream loop

In this tutorial, you call the `reverberator` to process the signal within your audio stream loop.

Create a `reverberator` System object, and specify the `SampleRate` property as the sample rate of your file reader. To adjust the reverberation effect, specify values for the `PreDelay` and `WetDryMix` properties. To apply the reverberation effect to an audio signal frame-by-frame, call the `reverberator` within your audio stream loop with an audio signal as an argument.

View Example Code

```matlab
frameLength = 256;
fileReader = dsp.AudioFileReader( ...
    'Counting-16-44p1-mono-15secs.wav', ...
    'SamplesPerFrame',frameLength);
deviceWriter = audioDeviceWriter( ...
    'SampleRate',fileReader.SampleRate);

scope = timescope( ...
    'SampleRate',fileReader.SampleRate, ...
    'TimeSpan',2, ...
    'BufferLength',fileReader.SampleRate*2*2, ...
    'YLimits',[-1,1], ...
    'TimeSpanOverrunAction',"Scroll");

reverb = reverberator( ...                   %<--- new lines of code
    'SampleRate',fileReader.SampleRate, ...  %<---
    'PreDelay',0.5, ...                      %<---
    'WetDryMix',0.4);                        %<---

while ~isDone(fileReader)
    signal = fileReader();
    reverbSignal = reverb(signal);           %<---
    deviceWriter(reverbSignal);              %<---
    scope([signal,mean(reverbSignal,2)])     %<---
end

release(fileReader)
release(deviceWriter)
release(reverb)                              %<---
release(scope)
```

Add Tunability

The Audio Toolbox user has several options to add real-time tunability to a processing algorithm. To add tunability to your audio stream loop, you can use:

- The **Audio Test Bench** – UI-based exercises for `audioPlugin` classes and most Audio Toolbox System objects.

- Built-in functions – Functions in Audio Toolbox for visualizing key aspects of your processing algorithms.
- A custom-built user interface – See "Real-Time Parameter Tuning" for a tutorial.
- A MIDI Controller – Many Audio Toolbox System objects include functions that support MIDI controls. You can use the `configureMIDI` function in the `reverberator` System object to synchronize your System object properties to MIDI controls. To use MIDI controls with System objects that do not have a `configureMIDI` function, see "MIDI Control Surface Interface".
- The User Datagram Protocol (UDP) – You can use UDP within MATLAB for connectionless transmission. You can also use UDP to receive or transmit datagrams between environments. Possible applications include using MATLAB tools to tune your audio processing algorithm while playing and visualizing your audio in a third-party environment. For an example application of UDP communication, see "Communicate Between a DAW and MATLAB Using UDP".

Quick Start Example

Audio Stream from Device to File

This example shows how to acquire an audio signal using your microphone using `audioDeviceReader`, perform basic signal processing, and write your signal to a file using `dsp.AudioFileWriter`.

Construct input and output objects. Use the sample rate of your input as the sample rate of your output.

```
deviceReader = audioDeviceReader;
fileWriter = dsp.AudioFileWriter(SampleRate=deviceReader.SampleRate);
```

Specify an audio processing algorithm. For simplicity, only add gain.

```
process = @(x) x.*5;
```

Place the following steps in a while loop for continuous stream processing:

1 Call your audio device reader like a function with no arguments to acquire one input frame.
2 Perform your signal processing operation on the input frame.
3 Call your audio file writer like a function with the processed frame as an argument.

The file is named `output.wav` and written to current folder by default.

```
disp("Begin Signal Input...")
```

```
Begin Signal Input...
```

```
tic
while toc<5
    mySignal = deviceReader();
    myProcessedSignal = process(mySignal);
    fileWriter(myProcessedSignal)
end
disp("End Signal Input")
```

```
End Signal Input
```

```
release(deviceReader)
release(fileWriter)
```

See Also

More About

- "Real-Time Audio in Simulink" on page 12-2
- "Audio I/O: Buffering, Latency, and Throughput" on page 8-2
- "MIDI Control Surface Interface"
- "Develop, Analyze, and Debug Plugins In Audio Test Bench"

Audio Plugins in MATLAB

Audio Plugins in MATLAB

Role of Audio Plugins in Audio Toolbox

The audio plugin is the suggested paradigm for developing your audio processing algorithm in Audio Toolbox. Once designed, the audio plugin can be validated, generated, and deployed to a third-party digital audio workstation (DAW).

Additional benefits of developing your audio processing as an audio plugin include:

- Rapid prototyping using the **Audio Test Bench**
- Integration with MIDI devices
- Code reuse

Some understanding of object-oriented programming (OOP) in the MATLAB environment is required to optimize your use of the audio plugin paradigm. If you are unfamiliar with these concepts, see "Why Use Object-Oriented Design".

For a review of audio plugins as defined outside the MATLAB environment, see "What Are DAWs, Audio Plugins, and MIDI Controllers?" on page 9-2

Defining Audio Plugins in the MATLAB Environment

In the MATLAB environment, an audio plugin refers to a class derived from the `audioPlugin` base class or the `audioPluginSource` base class.
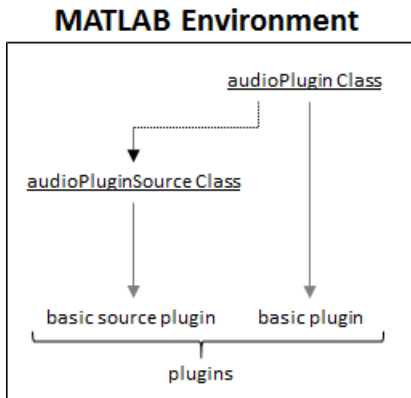
Audio Toolbox documentation uses the following terminology:

- A plugin is any audio plugin that derives from the `audioPlugin` class or the `audioPluginSource` class.
- A basic plugin is an audio plugin that derives from the `audioPlugin` class.
- A basic source plugin is an audio plugin that derives from the `audioPluginSource` class.

Audio plugins can also inherit from `matlab.System`. Any object that derives from `matlab.System` is referred to as a System object. Deriving from `matlab.System` allows for additional functionality, including Simulink integration. However, manipulating System objects requires a more advanced understanding of OOP in the MATLAB environment.

See "Audio Toolbox Extended Terminology" on page 11-14 for a detailed visualization of inheritance and terminology.

Design a Basic Plugin

In this example, you create a simple plugin, and then gradually increase complexity. Your final plugin uses a circular buffer to add an echo effect to an input audio signal. For additional considerations for generating a plugin, see "Export a MATLAB Plugin to a DAW" on page 7-2.

1 **Define a Basic Plugin Class.**   Begin with a simple plugin that copies input to output without modification.

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

`myEchoPlugin` illustrates the two minimum requirements for audio plugin classes. They must:

- Inherit from `audioPlugin` class
- Have a `process` method

The `process` method contains the primary frame-based audio processing algorithm. It is called in an audio stream loop to process an audio signal over time.

By default, both the input to and output from the `process` method have two channels (columns). The number of input rows (frame size) passed to `process` is determined by the environment in which it is run. The output must have the same number of rows as the input.

2 **Add a Plugin Property.**   A property can store information in an object. Add a property, `Gain`, to your class definition. Modify your `process` method to multiply the input by the value specified by the `Gain` property.

View Code

```
classdef myEchoPlugin < audioPlugin
    properties                              %<---
        Gain = 1.5;                         %<---
    end                                     %<---
    methods
        function out = process(plugin, in)  %<---
            out = in*plugin.Gain;           %<---
        end
    end
end
```

The first argument of the `process` method has changed from ~ to `plugin`. The first argument of `process` is reserved for the audio plugin object.

3 **Add a Plugin Parameter.**   Plugin parameters are the interface between plugin properties and the plugin user. The definition of this interface is handled by `audioPluginInterface`, which holds `audioPluginParameter` objects. To associate a plugin property to a parameter, specify the first argument of `audioPluginParameter` as a character vector entered exactly as the property you want to associate. The remaining arguments of `audioPluginParameter` specify optional additional parameter attributes.

In this example, you specify a mapping between the value of the parameter and its associated property, as well as the parameter display name as it appears on a plugin dialog box. By specifying `'Mapping'` as `{'lin',0,3}`, you set a linear mapping between the `Gain` property and the associated user-facing parameter, with an allowable range for the property between 0 and 3.

View Code

```
classdef myEchoPlugin < audioPlugin
    properties
        Gain = 1.5;
    end
    properties (Constant)                              %<---
        PluginInterface = audioPluginInterface( ...    %<---
            audioPluginParameter('Gain', ...           %<---
            'DisplayName','Echo Gain', ...             %<---
            'Mapping',{'lin',0,3}))                    %<---
    end                                                %<---
    methods
        function out = process(plugin, in)
            out = in*plugin.Gain;
        end
    end
end
```

4 **Add Private Properties.**   Add properties to store a circular buffer, a buffer index, and the N-sample delay of your echo. Because the plugin user does not need to see them, make

CircularBuffer, BufferIndex, and NSamples private properties. It is best practice to initialize properties to their type and size.

View Code

```matlab
classdef myEchoPlugin < audioPlugin
    properties
        Gain = 1.5;
    end
    properties (Access = private)              %<---
        CircularBuffer = zeros(192001,2);      %<---
        BufferIndex = 1;                       %<---
        NSamples = 0;                          %<---
    end                                        %<---
    properties (Constant)
        PluginInterface = audioPluginInterface( ...
            audioPluginParameter('Gain', ...
            'DisplayName','Echo Gain', ...
            'Mapping',{'lin',0,3}))
    end
    methods
        function out = process(plugin, in)
            out = in*plugin.Gain;
        end
    end
end
```

5   **Add an Echo.**   In the `process` method, write to and read from your circular buffer to create an output that consists of your input and a gain-adjusted echo. The first line of the `process` method initializes the output to the size of the input. It is best practice to initialize your output to avoid errors when generating plugins.

View Code

```matlab
classdef myEchoPlugin < audioPlugin
    properties
        Gain = 1.5;
    end
    properties (Access = private)
        CircularBuffer = zeros(192001,2);
        BufferIndex = 1;
        NSamples = 0;
    end
    properties (Constant)
        PluginInterface = audioPluginInterface(...
            audioPluginParameter('Gain',...
            'DisplayName','Echo Gain',...
            'Mapping',{'lin',0,3}))
    end
    methods
        function out = process(plugin, in)
            out = zeros(size(in));                     %<---
            writeIndex = plugin.BufferIndex;           %<---
            readIndex = writeIndex - plugin.NSamples;  %<---
            if readIndex <= 0                          %<---
                readIndex = readIndex + 192001;        %<---
            end                                        %<---
                                                       %<---
```

```
                for i = 1:size(in,1)                              %<---
                    plugin.CircularBuffer(writeIndex,:) = in(i,:);  %<---
                                                                  %<---
                    echo = plugin.CircularBuffer(readIndex,:);    %<---
                    out(i,:) = in(i,:) + echo * plugin.Gain;      %<---
                                                                  %<---
                    writeIndex = writeIndex + 1;                  %<---
                    if writeIndex > 192001                        %<---
                        writeIndex = 1;                           %<---
                    end                                           %<---
                                                                  %<---
                    readIndex = readIndex + 1;                    %<---
                    if readIndex > 192001                         %<---
                        readIndex = 1;                            %<---
                    end                                           %<---
                end                                               %<---
                plugin.BufferIndex = writeIndex;                  %<---
            end
        end
    end
```

6   **Make the Echo Delay Tunable.**   To allow the user to modify the NSamples delay of the echo, define a public property, Delay, and associate it with a parameter. Use the default audioPluginParameter mapping to allow the user to set the echo delay between 0 and 1 seconds.

Add a set method that listens for changes to the Delay property. Use the getSampleRate method of the audioPlugin base class to return the environment sample rate. Approximate a delay specified in seconds as a number of samples, NSamples. If the plugin user modifies the Delay property, set.Delay is called and the delay in samples (NSamples) is calculated. If the environment sample rate is above 192,000 Hz, the plugin does not perform as expected.

View Code

```
classdef myEchoPlugin < audioPlugin
    properties
        Gain = 1.5;
        Delay = 0.5;            %<---
    end
    properties (Access = private)
        CircularBuffer = zeros(192001,2);
        BufferIndex = 1;
        NSamples = 0;
    end
    properties (Constant)
        PluginInterface = audioPluginInterface(...
            audioPluginParameter('Gain',...
            'DisplayName','Echo Gain',...
            'Mapping',{'lin',0,3}),...          %<---
            audioPluginParameter('Delay',...    %<---
            'DisplayName','Echo Delay',...      %<---
            'Label','seconds'))                 %<---
    end
    methods
        function out = process(plugin, in)
            out = zeros(size(in));
            writeIndex = plugin.BufferIndex;
            readIndex = writeIndex - plugin.NSamples;
```

```
                        if readIndex <= 0
                            readIndex = readIndex + 192001;
                        end

                        for i = 1:size(in,1)
                            plugin.CircularBuffer(writeIndex,:) = in(i,:);

                            echo = plugin.CircularBuffer(readIndex,:);
                            out(i,:) = in(i,:) + echo*plugin.Gain;

                            writeIndex = writeIndex + 1;
                            if writeIndex > 192001
                                writeIndex = 1;
                            end

                            readIndex = readIndex + 1;
                            if readIndex > 192001
                                readIndex = 1;
                            end
                        end
                        plugin.BufferIndex = writeIndex;
                    end
                    function set.Delay(plugin, val)                      %<---
                        plugin.Delay = val;                             %<---
                        plugin.NSamples = floor(getSampleRate(plugin)*val); %<---
                    end                                                 %<---
                end
            end
        end
```

7 **Add a Reset Function.**   The `reset` method of a plugin contains instructions to reset the plugin between uses or when the environment sample rate changes. Because `NSamples` depends on the environment sample rate, update its value in the `reset` method.

View Code

```
classdef myEchoPlugin < audioPlugin
    properties
        Gain = 1.5;
        Delay = 0.5;
    end
    properties (Access = private)
        CircularBuffer = zeros(192001,2);
        BufferIndex = 1;
        NSamples = 0;
    end
    properties (Constant)
        PluginInterface = audioPluginInterface(...
            audioPluginParameter('Gain',...
            'DisplayName','Echo Gain',...
            'Mapping',{'lin',0,3}),...
            audioPluginParameter('Delay',...
            'DisplayName','Echo',...
            'Label','seconds'))
    end
    methods
        function out = process(plugin, in)
            out = zeros(size(in));
            writeIndex = plugin.BufferIndex;
            readIndex = writeIndex - plugin.NSamples;
            if readIndex <= 0
                readIndex = readIndex + 192001;
            end

            for i = 1:size(in,1)
                plugin.CircularBuffer(writeIndex,:) = in(i,:);

                echo = plugin.CircularBuffer(readIndex,:);
                out(i,:) = in(i,:) + echo*plugin.Gain;
```

```
                            writeIndex = writeIndex + 1;
                            if writeIndex > 192001
                                writeIndex = 1;
                            end

                            readIndex = readIndex + 1;
                            if readIndex > 192001
                                readIndex = 1;
                            end
                        end
                        plugin.BufferIndex = writeIndex;
                    end
                    function set.Delay(plugin, val)
                        plugin.Delay = val;
                        plugin.NSamples = floor(getSampleRate(plugin)*val);
                    end
                    function reset(plugin)                                      %<---
                        plugin.CircularBuffer = zeros(192001,2);               %<---
                        plugin.NSamples = floor(getSampleRate(plugin)*plugin.Delay);%<---
                    end                                                        %<---
                end
            end
```

Click here to open the completed plugin in MATLAB.

Design a System Object Plugin

You can map the basic plugin to a System object plugin. Note the differences between the two plugin types:

- A System object plugin inherits from both the `audioPlugin` base class and the `matlab.System` base class, not just `audioPlugin` base class.
- The primary audio processing method of a System object plugin is named `stepImpl`, not `process`.
- The reset method of a System object is named `resetImpl`, not `reset`.
- Both `resetImpl` and `stepImpl` must be defined as protected methods.
- System objects enable alternatives to the `set` method. For more information, see `processTunedPropertiesImpl`.

System Object Plugin

```
classdef myEchoSystemObjectPlugin < audioPlugin & matlab.System
    properties
        Gain = 1.5;
        Delay = 0.5;
    end
    properties (Access = private)
        CircularBuffer = zeros(192001,2);
        BufferIndex = 1;
        NSamples = 0;
    end
    properties (Constant)
        PluginInterface = audioPluginInterface(...
            audioPluginParameter('Gain',...
            'DisplayName','Echo Gain',...
            'Mapping',{'lin',0,3}),...
            audioPluginParameter('Delay',...
            'DisplayName','Echo',...
            'Label','seconds'))
    end
    methods (Access = protected)
        function out = stepImpl(plugin, in)
            out = zeros(size(in));
            writeIndex = plugin.BufferIndex;
            readIndex = writeIndex - plugin.NSamples;
            if readIndex <= 0
                readIndex = readIndex + 192001;
            end
```

```matlab
            for i = 1:size(in,1)
                plugin.CircularBuffer(writeIndex,:) = in(i,:);

                echo = plugin.CircularBuffer(readIndex,:);
                out(i,:) = in(i,:) + echo * plugin.Gain;

                writeIndex = writeIndex + 1;
                if writeIndex > 192001
                    writeIndex = 1;
                end

                readIndex = readIndex + 1;
                if readIndex > 192001
                    readIndex = 1;
                end
            end
            plugin.BufferIndex = writeIndex;
        end
        function resetImpl(plugin)
            plugin.CircularBuffer = zeros(192001,2);
            plugin.NSamples = floor(getSampleRate(plugin) * plugin.Delay);
        end
    end
    methods
        function set.Delay(plugin, val)
            plugin.Delay = val;
            plugin.NSamples = floor(getSampleRate(plugin) * val);
        end
    end
end
```

Click here to open the file.

Quick Start Basic Plugin

Template

```matlab
classdef myBasicPlugin < audioPlugin
    % myBasicPlugin is a template basic plugin. Use this template to create
    % your own basic plugin.

    properties
        % Use this section to initialize properties that the end-user interacts
        % with.
    end
    properties (Access = private)
        % Use this section to initialize properties that the end-user does not
        % interact with directly.
    end
    properties (Constant)
        % This section contains instructions to build your audio plugin
        % interface. The end-user uses the interface to adjust tunable
        % parameters. Use audioPluginParameter to associate a public property
        % with a tunable parameter.
    end
    methods
        function out = process(plugin, in)
            % This section contains instructions to process the input audio
            % signal. Use plugin.MyProperty to access a property of your
            % plugin.
        end
        function reset(plugin)
            % This section contains instructions to reset the plugin between
            % uses or if the environment sample rate changes.
        end
        function set.MyProperty(plugin, val)
            % This section contains instructions to execute if the
            % specified property is modified. Properties associated with
            % parameters are updated automatically. Use the set method to
            % execute more complicated instructions.
        end
    end
end
```

Annotated Example

This basic plugin enables the user to tune a damped applied gain. Click here to open the file.

Define properties to hold data.

Define plugin interface as a **Constant** property. The property must be named **PluginInterface**.

Define methods to update and process data.

```matlab
classdef myBasicPlugin < audioPlugin
    properties
        GaindB = 0;
    end
    properties (Access=private)
        GainMemory = 1;
        GainLinear = 1;
    end
    properties (Constant)
        PluginInterface = audioPluginInterface(...
            audioPluginParameter('GaindB',...
            'DisplayName','Damped Gain',...
            'Label','dB',...
            'Mapping',{'pow',1/3,-60,20}));
    end
    methods
        function out = process(plugin,in)
            gain = 0.5*(plugin.GainLinear + plugin.GainMemory);
            out = in*gain;

            plugin.GainMemory = gain;
        end
        function reset(plugin)
            plugin.GainMemory = 1;
        end
        function set.GaindB(plugin, val)
            plugin.GaindB = val;
            plugin.GainLinear = 10^(val/20);
        end
    end
end
```

Inherit from **audioPlugin** base class.

If the plugin user does not interact with property directly, specify as a **private** property.

Main processing algorithm. The method must be named **process**.

Called at the beginning of each use, or when the environment sample rate changes. The function must be named **reset**.

Called if specified property, *GaindB*, is modified.

Quick Start Basic Source Plugin

Template

```matlab
classdef myBasicSourcePlugin < audioPluginSource
    % myBasicSourcePlugin is a template for a basic source plugin. Use this
    % template to create your own basic source plugin.

    properties
        % Use this section to initialize properties that the end-user
        % interacts with.
    end
    properties (Access = private)
        % Use this section to initialize properties that the end-user does
        % not interact with directly.
    end
    properties (Constant)
        % This section contains instructions to build your audio plugin
        % interface. The end-user uses the interface to adjust tunable
        % parameters. Use audioPluginParameter to associate a public
        % property with a tunable parameter.
    end
    methods
        function out = process(plugin)
            % This section contains instructions to produce the output
            % audio signal. Use plugin.MyProperty to access a property of
            % your plugin. Use getSamplesPerFrame(plugin) to get the frame
            % size used by the environment.
        end
        function reset(plugin)
            % This section contains instructions to reset the plugin
            % between uses, or when the environment sample rate changes.
        end
        function set.MyProperty(plugin, val)
            % This section contains instructions to execute if the
            % specified property is modified. Properties associated with
            % parameters are updated automatically. Use the set method to
            % execute more complicated instructions.
        end
```
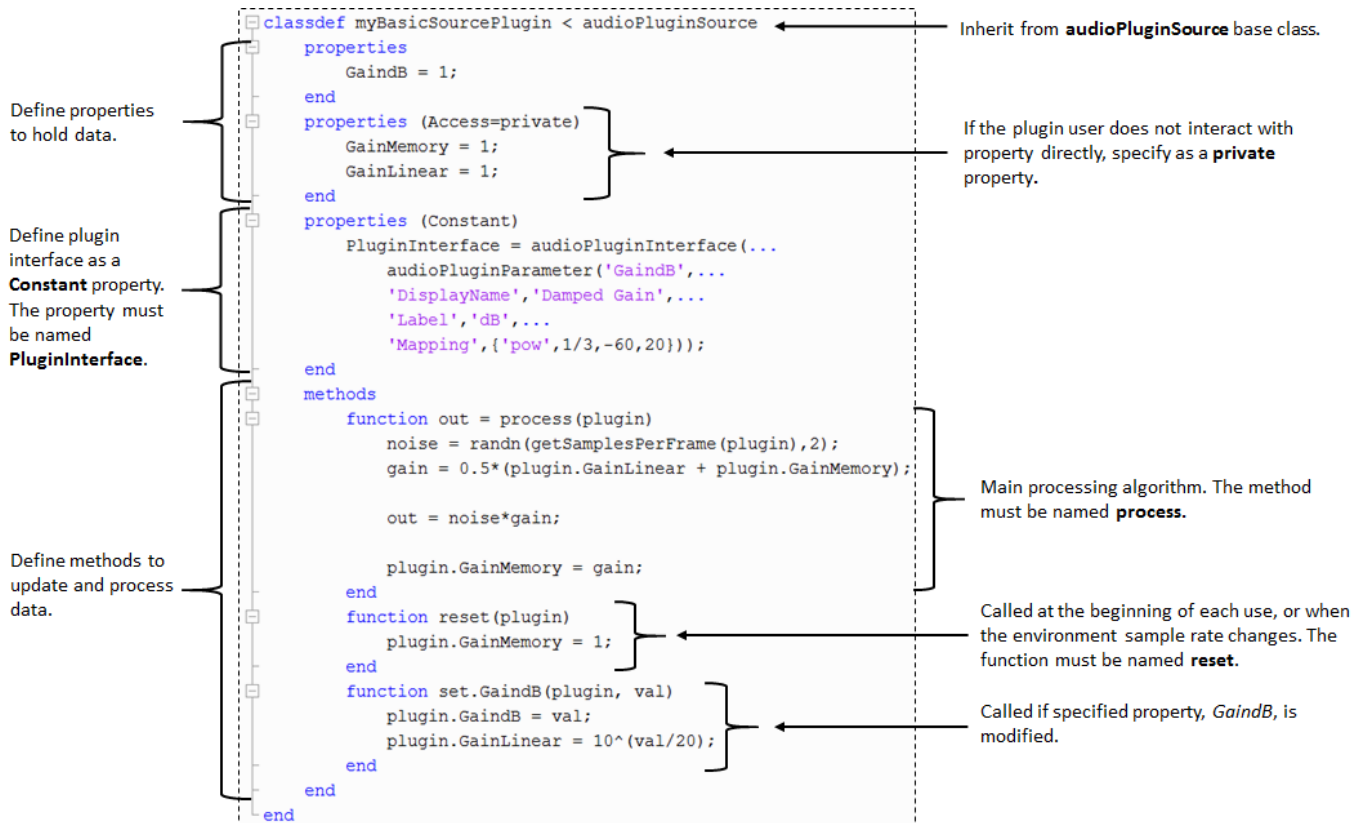
```
        end
end
```

Annotated Example

This basic source plugin enables the user to tune the damped gain of a noise signal. Click here to open the file.

Quick Start System Object Plugin

Template

```
classdef mySystemObjectPlugin < audioPlugin & matlab.System
    % mySystemObjectPlugin is a template for System object plugins.
    % Use this template to create your own System object plugin.

    properties
        % Use this section to initialize properties that the end-user interacts
        % with.
    end
    properties (Access = private)
        % Use this section to initialize properties that the end-user does not
        % interact with directly.
    end
    properties (Constant)
        % This section contains instructions to build your audio plugin
        % interface. The end-user uses the interface to adjust tunable
        % parameters. Use audioPluginParameter to associate a public property
        % with a tunable parameter.
    end
    methods (Access = protected)
        function out = stepImpl(plugin)
            % This section contains instructions to process the input audio
            % signal. Use plugin.MyProperty to access a property of your
            % plugin.
```

```
        end
    function resetImpl(plugin)
        % This section contains instructions to reset the plugin between
        % uses or if the environment sample rate changes.
    end
end
methods
    function set.MyProperty(plugin, val)
        % This section contains instructions to execute if the specified
        % property is modified. Properties associated with parameters are updated
        % automatically. Use the set method to execute more complicated
        % instructions.
    end
end
end
```

Annotated Example

This System object plugin enables the user to tune a damped applied gain. Click here to open the file.
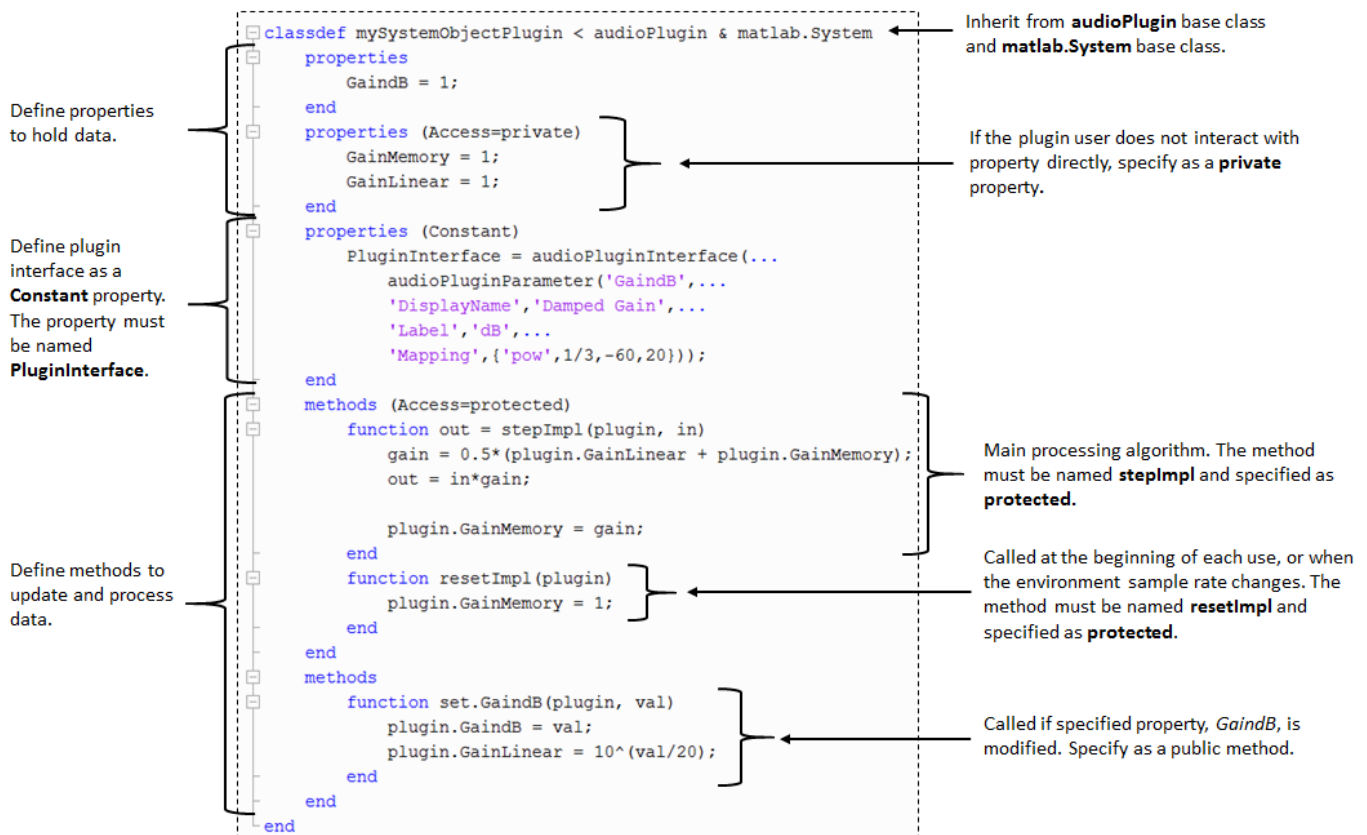
Quick Start System Object Source Plugin

Template

```
classdef mySystemObjectSourcePlugin < audioPluginSource & matlab.System
    % mySystemObjectPlugin is a template for System object source plugins.
    % Use the template to create your own System object source plugin.

    properties
        % Use this section to initialize properties that the end-user
        % interacts with.
    end
    properties (Access = private)
        % Use this section to initialize properties that the end-user does
```
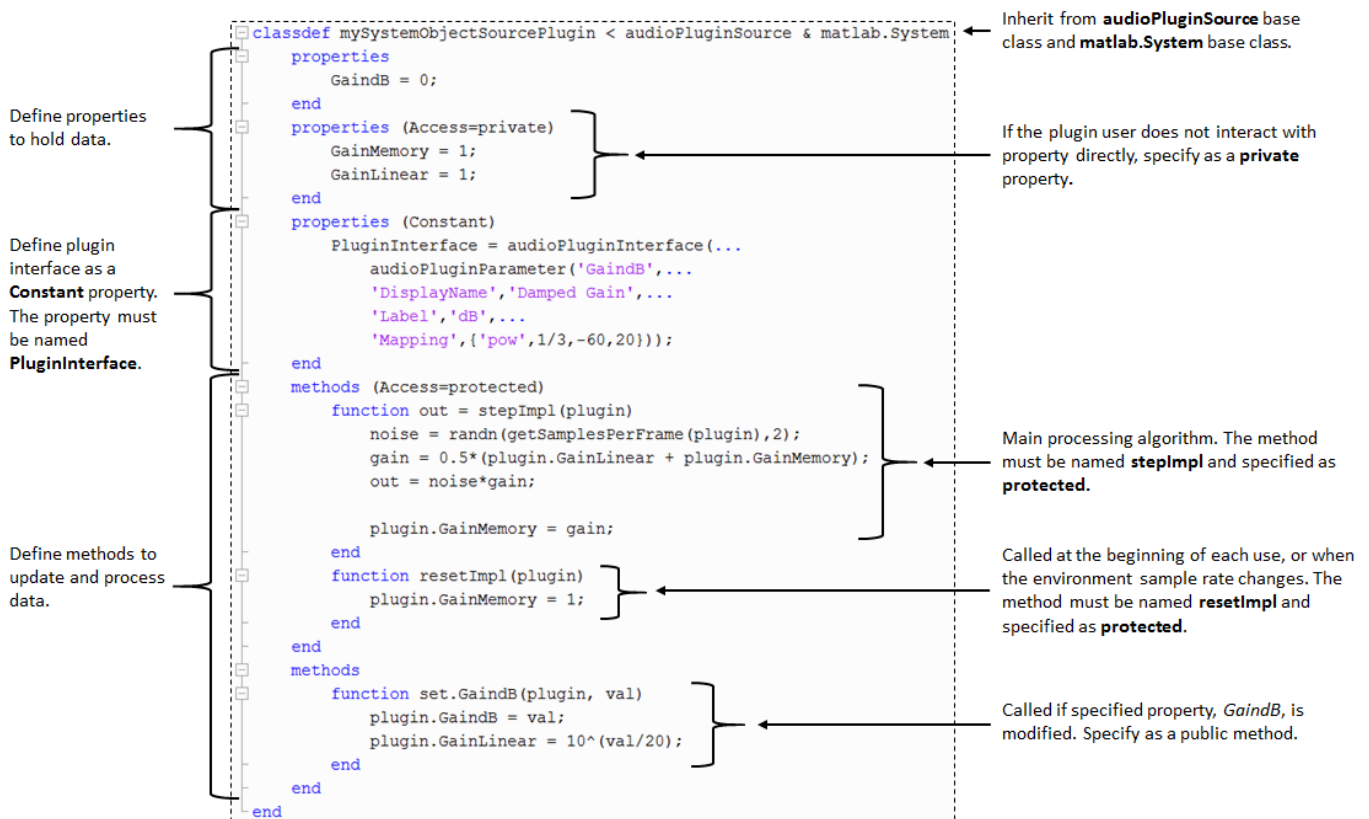
```matlab
        % not interact with directly.
    end
    properties (Constant)
        % This section contains instructions to build your audio plugin
        % interface. The end-user uses the interface to adjust tunable
        % parameters. Use audioPluginParameter to associate a public
        % property with a tunable parameter.
    end
    methods (Access = protected)
        function out = stepImpl(plugin)
            % This section contains instructions to produce the output
            % audio signal. Use plugin.MyProperty to access a property of
            % your plugin. Use getSamplesPerFrame(plugin) to get the frame
            % size used by the environment.
        end
        function resetImpl(plugin)
            % This section contains instructions to reset the plugin
            % between uses or if the environment sample rate changes.
        end
    end
    methods
        function set.MyProperty(plugin, val)
            % This section contains instructions to execute if the
            % specified property is modified. Properties associated with
            % parameters are updated automatically. Use the set method to
            % execute more complicated instructions.
        end
    end
end
```

Annotated Example

This System object source plugin enables the user to tune the damped gain of a noise signal. Click here to open the file.

Audio Toolbox Extended Terminology

In the MATLAB environment, an audio plugin refers to a class derived from the `audioPlugin` base class or the `audioPluginSource` base class. Audio plugins can also inherit from `matlab.System`. Any object that derives from `matlab.System` is referred to as a System object. Deriving from `matlab.System` allows for additional functionality, including Simulink integration. However, manipulating System objects requires a more advanced understanding of OOP in the MATLAB environment.
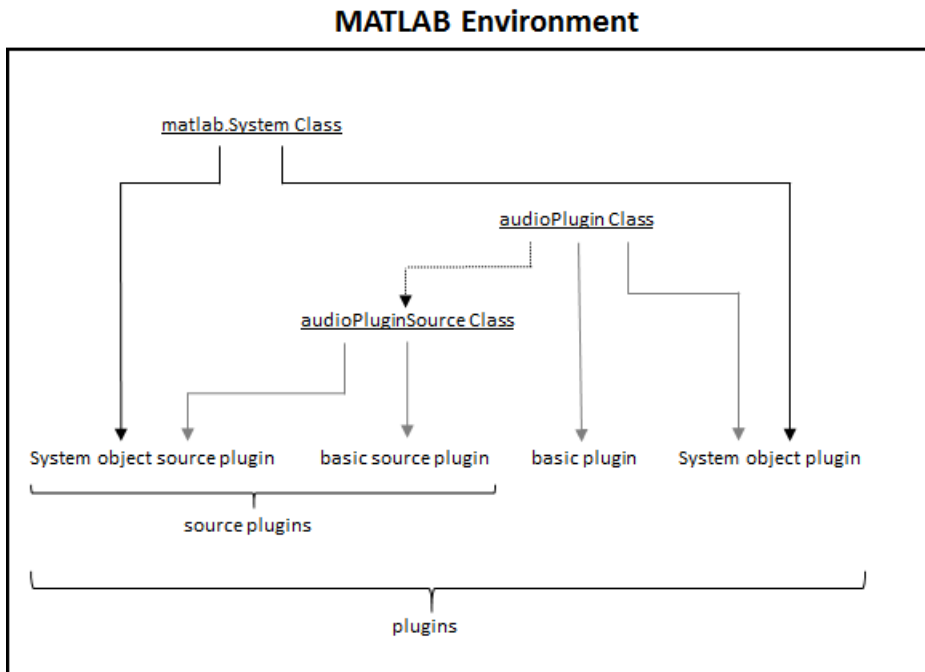
See Also

More About

- "Convert MATLAB Code to an Audio Plugin" on page 13-2
- "Include an Audio Plugin in Simulink"
- "What Are DAWs, Audio Plugins, and MIDI Controllers?" on page 9-2
- "Export a MATLAB Plugin to a DAW" on page 7-2

Real-Time Audio in Simulink

Real-Time Audio in Simulink

| **In this section...** |
| --- |
| "Create Model Using Audio Toolbox Simulink Model Templates" on page 12-2 |
| "Add Audio Toolbox Blocks to Model" on page 12-3 |
| "Block Characteristics" on page 12-4 |

Create Model Using Audio Toolbox Simulink Model Templates

The Audio Toolbox Simulink model templates provide a Simulink environment suitable for audio signal processing.

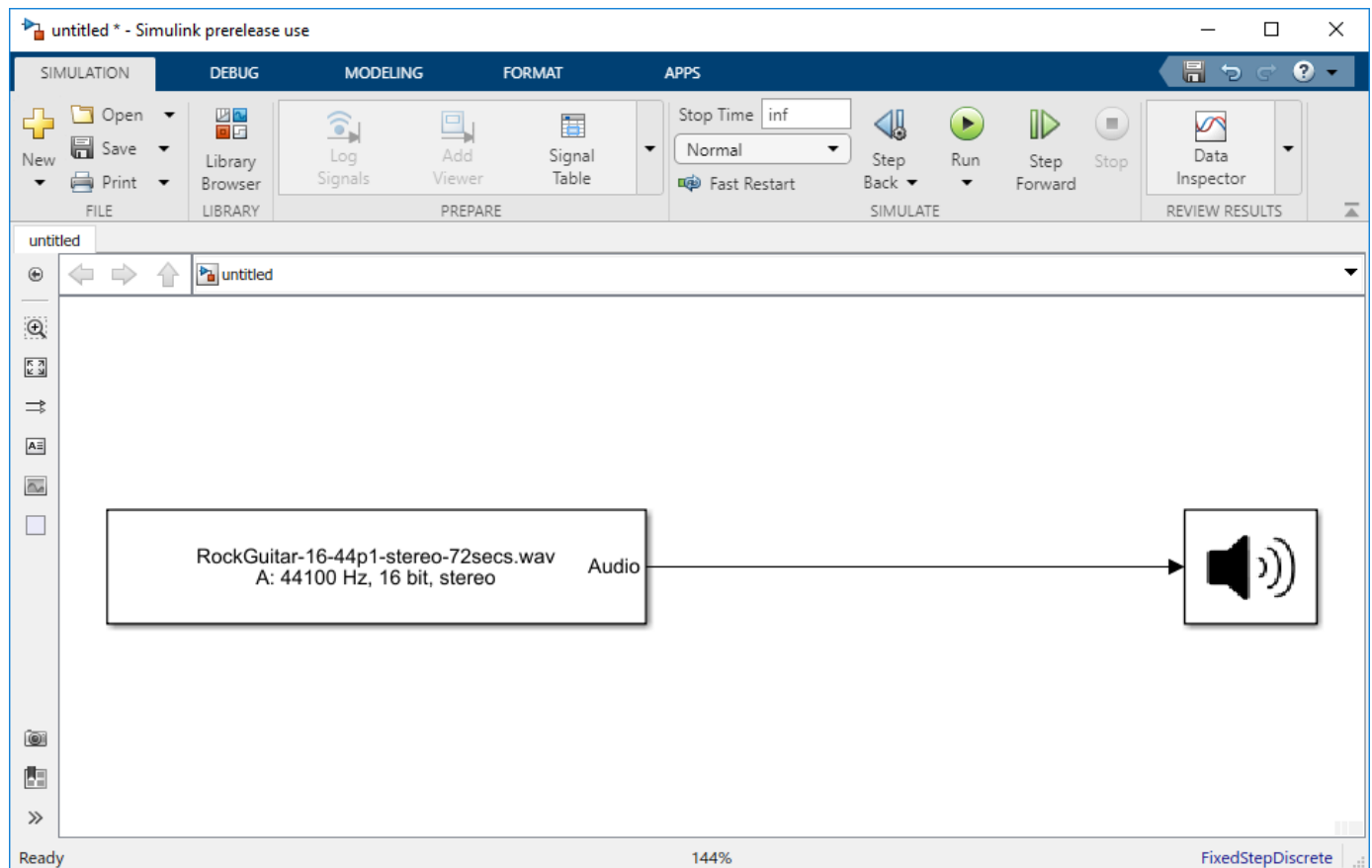To create a model using the Audio Toolbox Simulink model templates:

1   Open the Simulink Start Page by typing `simulink` at the MATLAB command prompt.
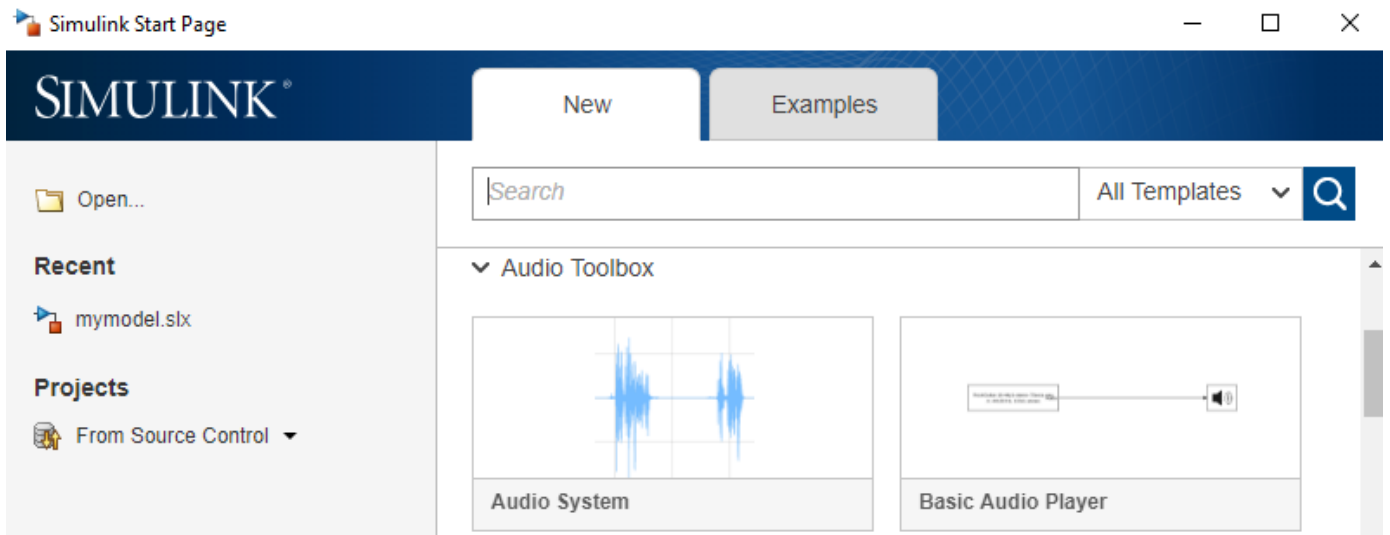
2   Under Audio Toolbox, click the model template you want.

The two Audio Toolbox Simulink model templates are:

- Audio System – Creates a blank model configured with settings recommended for Audio Toolbox.
- Basic Audio Player – Creates an audio model configured with settings recommended for Audio Toolbox. This model uses a From Multimedia File block to read multimedia files, and an Audio Device Writer block to send sound data to the default audio device of your computer. Adjust the model as needed to model your audio system. For example, to process live audio input, replace the From Multimedia File block with an Audio Device Reader block.

Add Audio Toolbox Blocks to Model

1. Create a model using an Audio Toolbox template.

2. Open the Simulink Library Browser and select Audio Toolbox.

3. Select a block from one of the Audio Toolbox Block Library categories and add it to your model.

4. In this example, a Compressor is added to the model by dragging and dropping from the Dynamic Range Control category of the Simulink Library Browser.

5. To run your model, click the ▶ button.

6. Open a block parameter user interface by double-clicking the block. You can modify parameters while the model runs. For example, if you added a Compressor block, you can adjust the **Threshold (dB)** dial to compress the dynamic range of your audio signal.
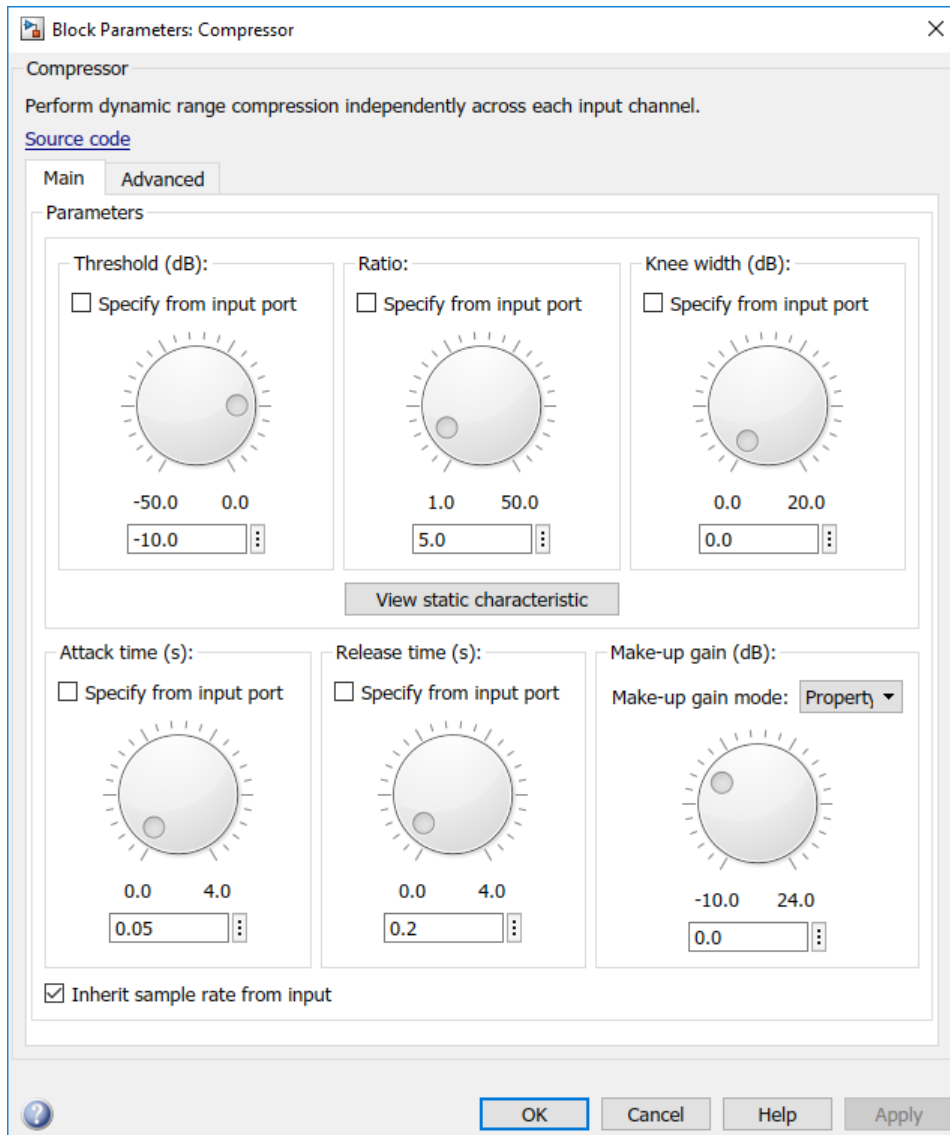
7   Running a model in the Simulink environment does not save the model. Save your model by clicking the 🖫 button.

Block Characteristics

You can type `showaudioblockdatatypetable` at the MATLAB command line to generate a table showing characteristics of Simulink blocks in Audio Toolbox.

See Also

More About

*   "Audio Input and Audio Output" on page 2-2
*   "Include an Audio Plugin in Simulink"

Convert MATLAB Code to an Audio Plugin

Convert MATLAB Code to an Audio Plugin

Audio Toolbox supports several approaches for the development of audio processing algorithms. Two common approaches include procedural programming using MATLAB scripts and object-oriented programming using MATLAB classes. The audio plugin class is the suggested paradigm for developing your audio processing algorithm in Audio Toolbox. See "Audio Plugins in MATLAB" on page 11-2 for a tutorial on the structure, benefits, and uses of audio plugins.

This tutorial presents an existing algorithm developed as a MATLAB script, and then walks through the steps to convert the script to an audio plugin class. Use this tutorial to understand the relationship between procedural programming and object-oriented programming. You can also use this tutorial as a template to convert any audio processing you developed as MATLAB scripts to the audio plugin paradigm.

Inspect Existing MATLAB Script

The MATLAB script has these sections:

A **Variable Initialization.** Variables are initialized with known values, including the number of samples per frame (`frameSize`) for frame-based stream processing.

B **Object Construction.**

- Two `audioOscillator` System objects –– Construct to create time-varying gain control signals.
- `dsp.AudioFileReader` System object –– Construct to read an audio signal from a file.
- `audioDeviceWriter` System object –– Construct to write an audio signal to your default audio device.

C **Audio Stream Loop.** Mixes stereo channels into a mono signal. The mono signal is used to create a new stereo signal. Each channel of the new stereo signal oscillates in applied gain between 0 and 2, with a respective 90-degree phase shift.

View Code

Click here to open this example.

```matlab
%% Section A: Variable Initialization

% Specify frequency of gain oscillation.
Frequency = 1;

% Determine sample rate of audio file (input audio signal).
fileInfo = audioinfo(...
    'RockGuitar-16-44p1-stereo-72secs.wav');
sampleRate = fileInfo.SampleRate;

% Specify size of frame to read in from audio file.
frameSize = 256;

%% Section B: Object Construction

Sine = audioOscillator(...
    'DCOffset',1,...
    'SamplesPerFrame',frameSize,...
```

```
        'Frequency',Frequency,...
        'SampleRate',sampleRate);

    Cosine = audioOscillator(...
        'DCOffset',1,...
        'PhaseOffset',0.5,...
        'Frequency',Frequency,...
        'SamplesPerFrame',frameSize,...
        'SampleRate',sampleRate);

    fileReader = dsp.AudioFileReader(...
        'Filename',fileInfo.Filename,...
        'SamplesPerFrame',frameSize);

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

    %% Section C: Audio Stream Loop

    while ~isDone(fileReader)

        % Read in one frame of audio signal from file.
        in = fileReader();

        % Mix stereo input to mono.
        mono = 0.5*sum(in,2);

        % Get current frame of Sine and Cosine gain functions.
        gainLeft = Sine();
        gainRight = Cosine();

        % Process signal by multiplying by variable gain matrix.
        out = [mono,mono] .* [gainLeft,gainRight];

        % Write one frame of audio signal to device.
        deviceWriter(out);

    end
```
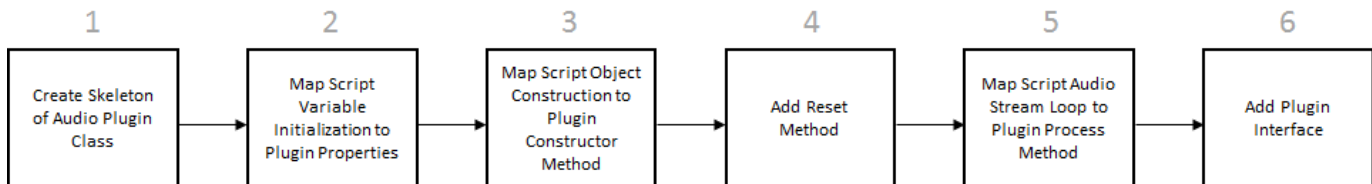
Convert MATLAB Script to Plugin Class

This tutorial converts a MATLAB script to an audio plugin class in six steps. You begin by creating a skeleton of a basic audio plugin class, and then map sections of the MATLAB script to the audio plugin class.

For an overview of how a MATLAB script is converted to a plugin class, inspect the script to plugin visual mapping. To perform this conversion, walk through the example for explanations and step-by-step instructions.

MATLAB Script → **Plugin Class Definition**
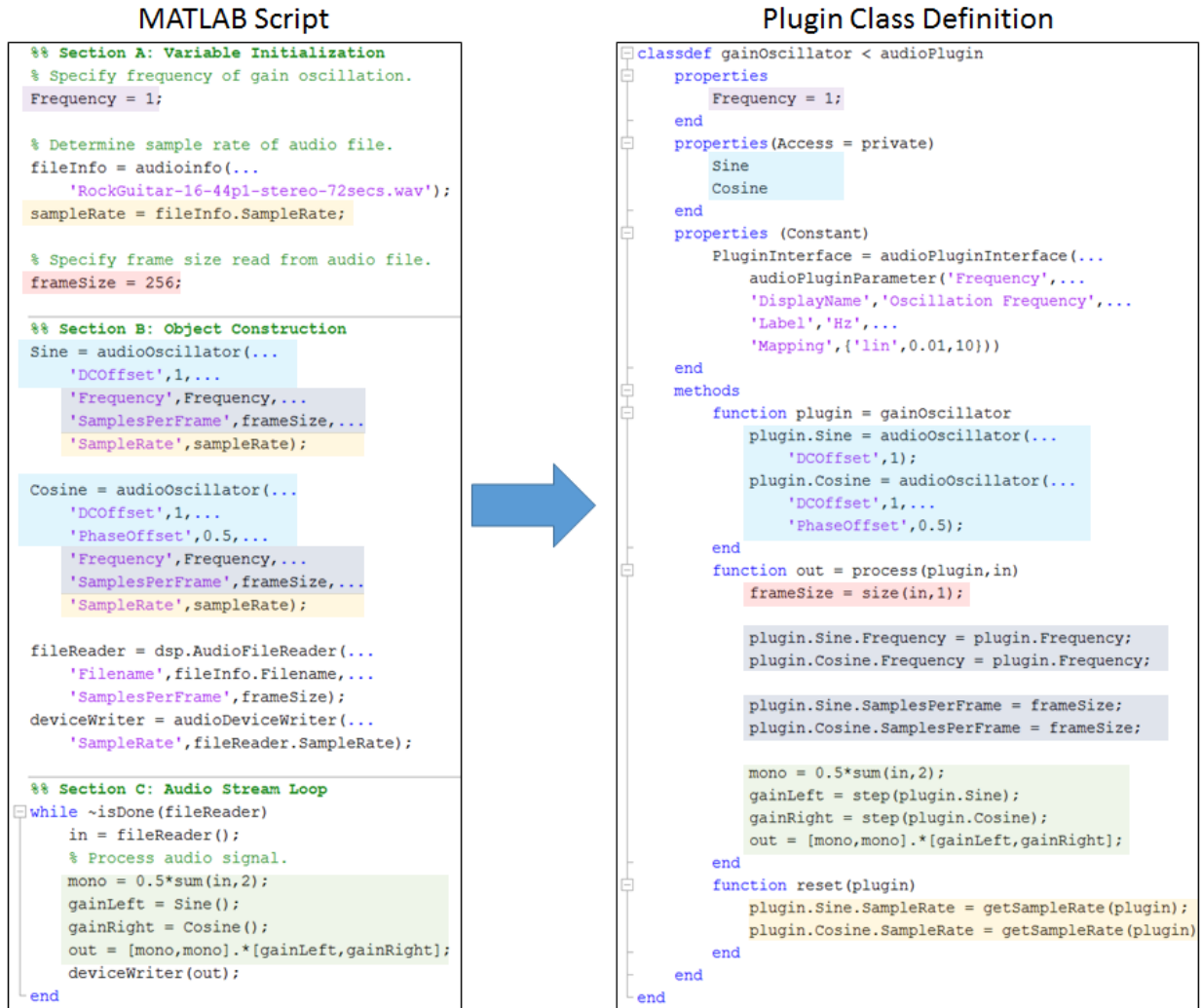
1. Create Skeleton of Audio Plugin Class

Begin with the basic skeleton of an audio plugin class. This skeleton is not the minimum required, but a common minimum to create an interesting audio plugin. See "Audio Plugins in MATLAB" on page 11-2 for the minimum requirements to create a basic audio plugin.

View Code

```
classdef gainOscillator < audioPlugin
    % gainOscillator Phase-shifted stereo gain oscillation.
    %   The process method mixes stereo channels into a mono signal. The
    %   mono signal is used to create a stereo signal, with each channel
    %   oscillating in gain between zero and two, with a respective 90
    %   degree phase shift.

    properties
        % Use this section to initialize properties that the end-user
        % interacts with.
    end
```

```matlab
    properties (Access = private)
        % Use this section to initialize properties that the end-user does
        % not interact with directly.
    end
    properties (Constant)
        % This section contains instructions to build your audio plugin
        % interface. The end-user uses the interface to adjust tunable
        % parameters. Use audioPluginParameter to associate a public
        % property with a tunable parameter.
    end
    methods
        function out = process(plugin, in)
            % This section contains instructions to process the input audio
            % signal. Use plugin.MyProperty to access a property of your
            % plugin.
        end
        function reset(plugin)
            % This section contains instructions to reset the plugin
            % between uses or if the environment sample rate changes.
        end
    end
end
```

2. Map Script Variable Initialization to Plugin Properties

Properties allow a plugin to store information across sections of the plugin class definition. If a property has access set to private, the property is not accessible to the end user of a plugin. Variable initialization in a script maps to plugin properties.

- A valid plugin must allow input to the `process` method to have a variable frame size. Frame size is determined for each input frame in the `process` method of the plugin. Because frame size is used only in the `process` method, you do not declare it in the properties section.
- A valid audio plugin must allow input to the `process` method to have a variable sample rate. The `reset` method of a plugin is called when the environment changes the sample rate. Determine the sample rate in the `reset` method using the `getSampleRate` method inherited from the `audioPlugin` base class.
- The objects used by a plugin must be declared as properties to be used in multiple sections of the plugin. However, the constructor method of a plugin performs object construction.

View Code

```matlab
classdef gainOscillator < audioPlugin
    properties
        Frequency = 1;  %<---
    end
    properties(Access = private)
        Sine            %<---
        Cosine          %<---
    end
    properties (Constant)
    end
    methods
        function out = process(plugin,in)
        end
        function reset(plugin)
        end
    end
end
```

3. Map Script Object Construction to Plugin Constructor Method

Add a constructor method to the methods section of your audio plugin. The constructor method of a plugin has the form:

```matlab
function plugin = myPluginClassName
    % Instructions to construct plugin object.
end
```

If your plugin uses objects, construct them when the plugin is constructed. Set nontunable properties of objects used by your plugin during construction.

In this example, you construct the `Sine` and `Cosine` objects in the constructor method of the plugin.

View Code

```matlab
classdef gainOscillator < audioPlugin
    properties
        Frequency = 1;
    end
    properties(Access = private)
        Sine
        Cosine
    end
    properties (Constant)
    end
    methods
        function plugin = gainOscillator       %<---
            plugin.Sine = audioOscillator(...   %<---
                'DCOffset',1);                  %<---
            plugin.Cosine = audioOscillator(... %<---
                'DCOffset',1,...                %<---
                'PhaseOffset',0.5);             %<---
        end                                     %<---
        function out = process(plugin,in)
        end
        function reset(plugin)
        end
    end
end
```

4. Add Reset Method

The `reset` method of a plugin is called every time a new session is started with the plugin, or when the environment changes sample rate. Use the `reset` method to update the `SampleRate` property of your `Sine` and `Cosine` objects. To query the sample rate, use the `getSampleRate` base class method.

View Code

```matlab
classdef gainOscillator < audioPlugin
    properties
        Frequency = 1;
    end
    properties(Access = private)
        Sine
        Cosine
    end
    properties (Constant)
    end
    methods
        function plugin = gainOscillator
            plugin.Sine = audioOscillator(...
```

```matlab
                'DCOffset',1);
            plugin.Cosine = audioOscillator(...
                'DCOffset',1,...
                'PhaseOffset',0.5);
        end
        function out = process(plugin,in)
        end
        function reset(plugin)
            plugin.Sine.SampleRate = getSampleRate(plugin);     %<---
            plugin.Cosine.SampleRate = getSampleRate(plugin);   %<---
        end
    end
end
```

5. Map Script Audio Stream Loop to Plugin Process Method

The contents of the audio stream loop in a script maps to the `process` method of an audio plugin, with these differences:

- A valid audio plugin must accept a variable frame size, so frame size is calculated for every call to the `process` method. Because frame size is variable, any processing that relies on frame size must update when input frame size changes.
- The environment handles the input and output to the `process` method.

View Code

```matlab
classdef gainOscillator < audioPlugin
    properties
        Frequency = 1;
    end
    properties(Access = private)
        Sine
        Cosine
    end
    properties (Constant)
    end
    methods
        function plugin = gainOscillator
            plugin.Sine = audioOscillator(...
                'DCOffset',1);
            plugin.Cosine = audioOscillator(...
                'DCOffset',1,...
                'PhaseOffset',0.5);
        end
        function out = process(plugin,in)
            frameSize = size(in,1);                      %<---

            plugin.Sine.SamplesPerFrame = frameSize;     %<---
            plugin.Cosine.SamplesPerFrame = frameSize;   %<---

            mono = 0.5*sum(in,2);                        %<---
            gainLeft = step(plugin.sine);                %<---
            gainRight = step(plugin.cosine);             %<---
            out = [mono,mono].*[gainLeft,gainRight];     %<---
        end
        function reset(plugin)
            plugin.Sine.SampleRate = getSampleRate(plugin);
```

```
                    plugin.Cosine.SampleRate = getSampleRate(plugin);
            end
        end
end
```

6. Add Plugin Interface

The plugin interface lets users view the plugin and tune its properties. Specify `PluginInterface` as an `audioPluginInterface` object that contains an `audioPluginParameter` object. The first argument of `audioPluginParameter` is the property you want to synchronize with a tunable parameter. Choose the display name, label the units, and set the parameter range. This example uses 0.1 to 10 as a reasonable range for the `Frequency` property. Write code so that during each call to the `process` method, your `Sine` and `Cosine` objects are updated with the current frequency value.

View Code

```
classdef gainOscillator < audioPlugin
    properties
        Frequency = 1;
    end
    properties(Access = private)
        Sine
        Cosine
    end
    properties (Constant)
        PluginInterface = audioPluginInterface(...          %<---
            audioPluginParameter('Frequency',...            %<---
            'DisplayName','Oscillation Frequency',...       %<---
            'Label','Hz',...                                %<---
            'Mapping',{'lin',0.01,10}))                     %<---
    end
    methods
        function plugin = gainOscillator
            plugin.Sine = audioOscillator(...
                'DCOffset',1);
            plugin.Cosine = audioOscillator(...
                'DCOffset',1,...
                'PhaseOffset',0.5);
        end
        function out = process(plugin,in)
            frameSize = size(in,1);

            plugin.Sine.Frequency = plugin.Frequency;       %<---
            plugin.Cosine.Frequency = plugin.Frequency;     %<---

            plugin.Sine.SamplesPerFrame = frameSize;
            plugin.Cosine.SamplesPerFrame = frameSize;

            mono = 0.5*sum(in,2);
            gainLeft = step(plugin.Sine);
            gainRight = step(plugin.Cosine);
            out = [mono,mono].*[gainLeft,gainRight];
        end
        function reset(plugin)
            plugin.Sine.SampleRate = getSampleRate(plugin);
            plugin.Cosine.SampleRate = getSampleRate(plugin);
        end
```

```
        end
end
```

Once your audio plugin class definition is complete:

1 Save your plugin class definition file.

2 Validate your plugin using `validateAudioPlugin`.

3 Prototype it using **Audio Test Bench**.

4 Generate is using `generateAudioPlugin`.

See Also

More About

- "Real-Time Audio in MATLAB" on page 10-2
- "What Are DAWs, Audio Plugins, and MIDI Controllers?" on page 9-2
- "Audio Plugins in MATLAB" on page 11-2
- "Include an Audio Plugin in Simulink"
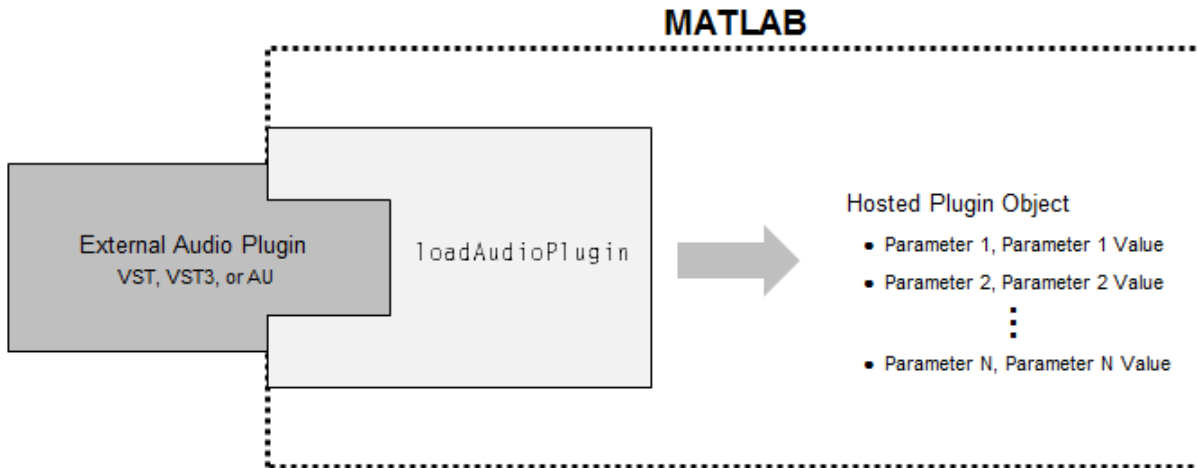- "Export a MATLAB Plugin to a DAW" on page 7-2

Host External Audio Plugins

Host External Audio Plugins

You can host VST, VST3, and AU plugins in MATLAB by using the `loadAudioPlugin` function from Audio Toolbox.

After you load an external audio plugin, you process audio through its main audio-processing algorithm.

Audio Toolbox enables three ways to interact with the hosted audio plugin:

- "Property Display Mode (Default)" on page 14-3
- "Parameter Display Mode" on page 14-7
- "Graphical Interaction" on page 14-13

The following tutorials–one version for property display mode and one version for parameter display mode–walk you through the process of hosting an externally authored VST plugin and interacting with the plugin at the MATLAB command line. You host a plugin from the suite of ReaPlugs VST plugins distributed by Cockos Incorporated. To download the ReaPlugs VST FX Suite for your system, follow the instructions on the REAPER website. A 64-bit Windows platform is used in this tutorial. The `loadAudioPlugin` function cannot load 32-bit plugins.

Property Display Mode (Default)

Setting the display mode to property enables you to interact with the hosted plugin object using standard dot notation. For example:

```
hostedObject.Gain = 5; % dB
```

`Property` is the default display mode of hosted plugins.
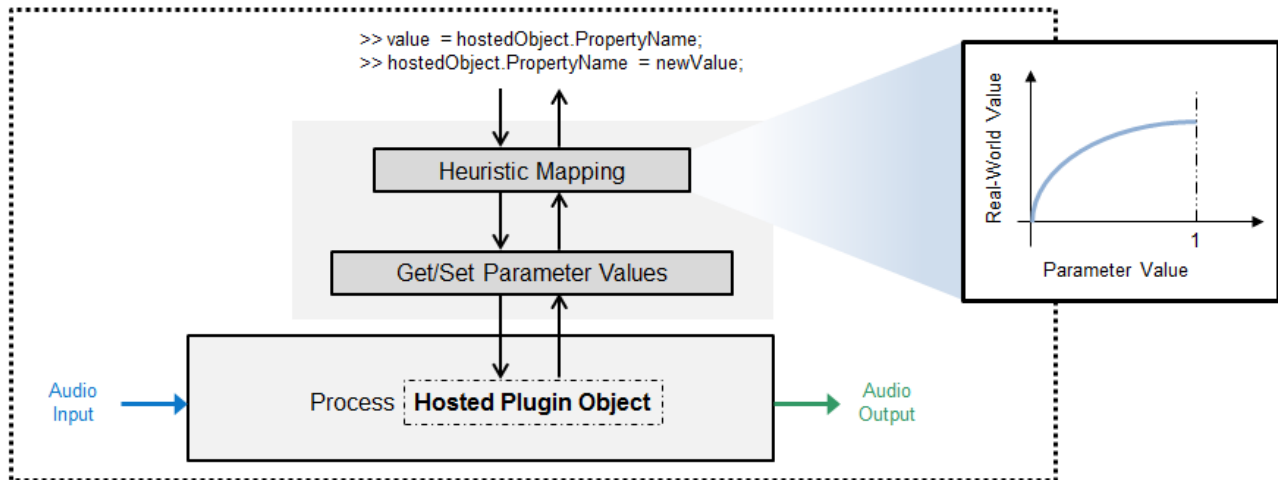
Numeric parameters are mapped through a heuristic interpretation of the normalized parameter values and the corresponding display values. The property display mode is simple and intuitive. However, due to the "Heuristic Mapping" on page 14-13 of normalized parameter values to real-world property values, the property display mode may break down for some plugins. In this case, you should use the parameter display mode.

Host External Audio Plugin Tutorial (Property Display Mode)

The following tutorial walks through the steps of loading and configuring an external audio plugin in property display mode.

1. Load External Audio Plugin

Use the `loadAudioPlugin` function to host the `ReaDelay` VST plugin. If the plugin is in your current folder, you can specify just the file name. Otherwise, you must specify the full path. In this example, the plugin is in the current folder. By default, the display mode is set to property.

```
hostedPlugin = loadAudioPlugin('readelay-standalone.dll')

hostedPlugin =
  VST plugin 'ReaDelay (ReaPlugs Edition)'  2 in, 2 out

            Wet: 0 dB
            Dry: 0 dB
      x1_Enabled: 'ON'
     x1_Length_4: 0 ms
     x1_Length_5: 4 8N
```

```
        x1_Feedback: -Inf dB
         x1_Lowpass: 20000 Hz
          x1_Hipass: 0 Hz
      x1_Resolution: 24 bits
     x1_StereoWidth: 1
          x1_Volume: 0 dB
             x1_Pan: 0 %
```

The first line displays the plugin type, plugin display name, and the number of input and output channels for the main audio-processing algorithm of the plugin. If you are hosting a source plugin, the number of output channels and the default samples per frame are displayed.

By default, all properties are displayed.

2. Tune Hosted Plugin Property Values

You can interact with the properties of the hosted plugin using dot notation. If you go above or below the allowed range of the property, an error message will state the valid boundaries.

```
hostedPlugin.x1_Hipass = 120;
highPassSetting = hostedPlugin.x1_Hipass
```
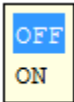
```
highPassSetting = 120
```

You can use tab-completion to get a list of possible values for enumerated properties.

3. Use Hosted Plugin to Process Audio

To process an audio signal with the hosted plugin, use `process`.

```
audioIn = [1,1];
audioOut = process(hostedPlugin,audioIn);
```

Audio plugins are designed for variable-frame-based processing, meaning that you can call `process` with successive audio input frames of different lengths. The hosted plugin saves the internal states required for continuous processing. To process an audio signal read from a file and then written to your audio output device, place your hosted plugin in an audio stream loop. Use `dsp.AudioFileWriter` and `audioDeviceWriter` objects as the input and output to your audio stream loop, respectively. Set the sample rate of the hosted plugin to the sample rate of the audio file by using `setSampleRate`.

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

deviceWriter = audioDeviceWriter('SampleRate',sampleRate);
setSampleRate(hostedPlugin,sampleRate);

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

    % The hosted plugin requires a stereo input.
```

```
    stereoAudioIn = [audioIn,audioIn];

    x = process(hostedPlugin,stereoAudioIn);

    deviceWriter(x);
end

release(fileReader)
release(deviceWriter)
```

You can modify properties in the audio stream loop. To control the `Wet` property of your plugin in an audio stream loop, create an `audioOscillator` System object™. Use the `fileReader`, `deviceWriter`, and `hostedPlugin` objects you created previously to process the audio.

```
osc = audioOscillator('sine', ...
    'Frequency',10, ...
    'Amplitude',20, ...
    'DCOffset',-20, ...
    'SamplesPerFrame',fileReader.SamplesPerFrame, ...
    'SampleRate',sampleRate);

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

    controlSignal = osc();
    hostedPlugin.Wet = controlSignal(1);

    stereoAudioIn = [audioIn,audioIn];
    x = process(hostedPlugin,stereoAudioIn);
    deviceWriter(x);
end

release(fileReader)
release(deviceWriter)
```

4. Analyze Hosted Plugin

You can use the Audio Toolbox measurement and visualization tools to display behavior information about your hosted plugin. To display the input and output of your hosted audio plugin, create a time scope. Create a `loudnessMeter` object and use the `'EBU Mode'` visualization to monitor loudness output by the hosted plugin. Use the `fileReader`, `deviceWriter`, `osc`, and `hostedPlugin` objects you created previously to process the audio.

```
scope = timescope('SampleRate',sampleRate, ...
    'TimeSpanSource','property', ...
    'TimeSpanOverrunAction','scroll', ...
    'TimeSpan',5, ...
    'BufferLength',5*2*sampleRate, ...
    'YLimits',[-1 1]);

loudMtr = loudnessMeter('SampleRate',sampleRate);
visualize(loudMtr)

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

    controlSignal = osc();
    hostedPlugin.Wet = controlSignal(1);
```
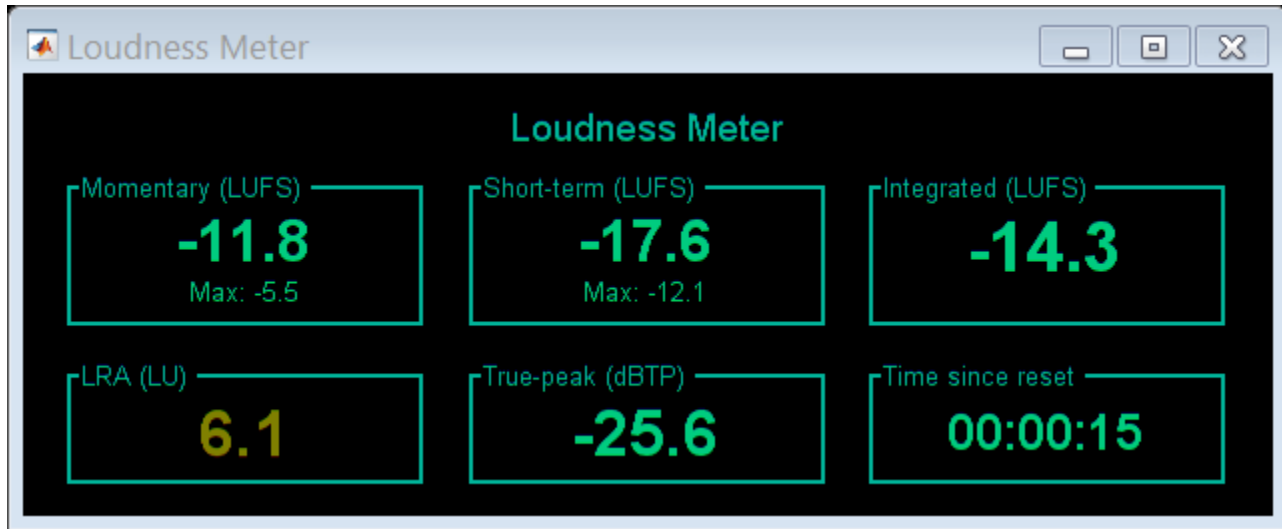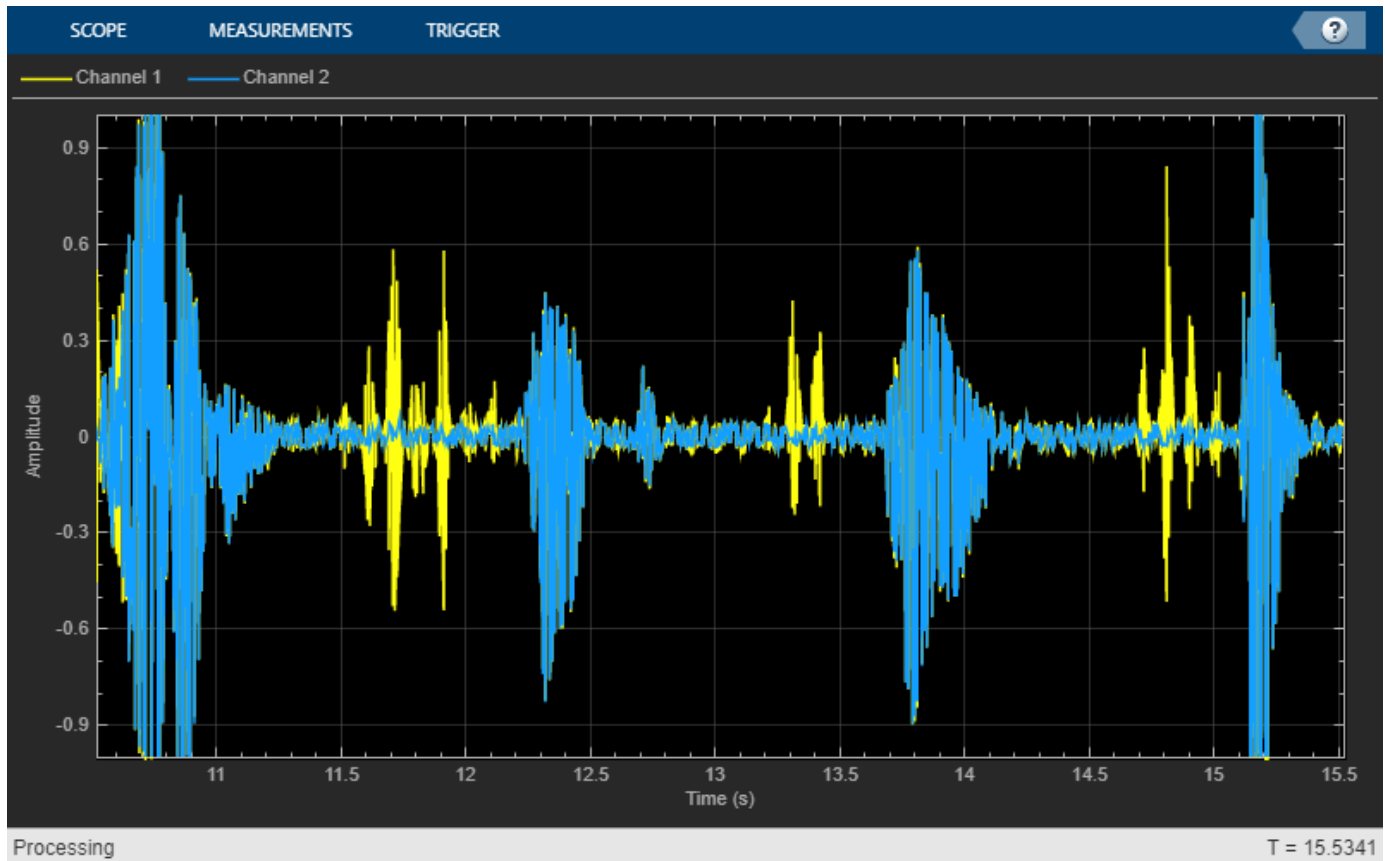
```
        stereoAudioIn = [audioIn,audioIn];
        x = process(hostedPlugin,stereoAudioIn);

        loudMtr(x);
        scope([x(:,1),audioIn(:,1)])

        deviceWriter(x);
    end
```

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

Parameter Display Mode

Setting the display mode to parameter enables you to interact with the hosted plugin in the most basic way possible: by setting and getting normalized parameter values. You can use the information optionally returned by getParameter to interpret normalized values as real-world values, such as decibels and Hertz.
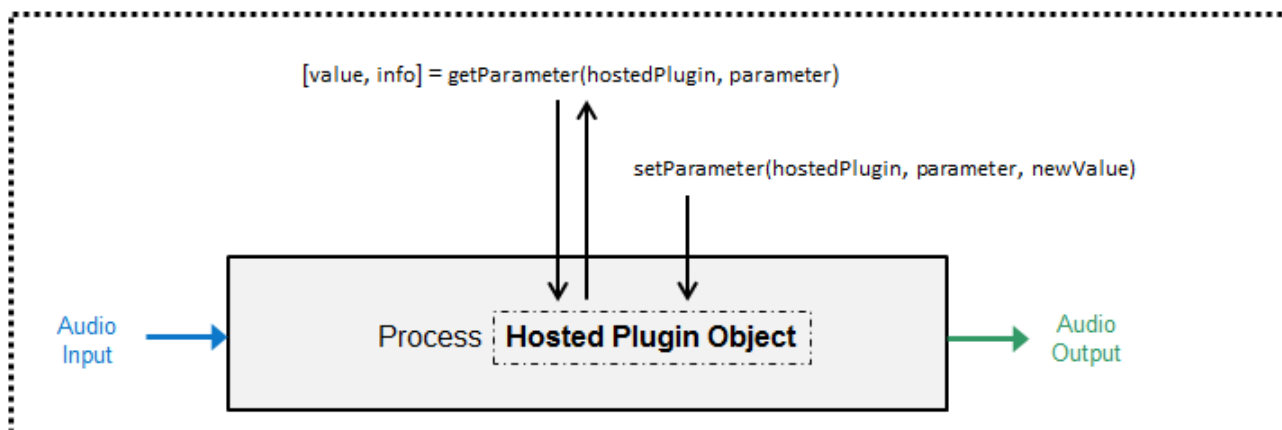
Host External Audio Plugin Tutorial (Parameter Mode)

The following tutorial walks through the steps of loading and configuring an external audio plugin in parameter display mode.

1. Load External Audio Plugin

Use the `loadAudioPlugin` function to host the `ReaDelay` VST plugin. If the plugin is in your current folder, you can specify just the file name. Otherwise, you must specify the full path. In this example, the plugin is in the current folder.

```
hostedPlugin = loadAudioPlugin('readelay-standalone.dll');
```

By default, the display mode is set to property. Set the `DisplayMode` property to `Parameters` for low-level interaction with the hosted plugin.

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

```
hostedPlugin =
  VST plugin 'ReaDelay (ReaPlugs Edition)'  2 in, 2 out

          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
    7 parameters not displayed. See all 12 params.
```

The first line displays the plugin type, plugin display name, and the number of input and output channels for the main audio processing algorithm of the plugin. If you are hosting a source plugin, the number of output channels and the default samples per frame are displayed.

By default, only the first five parameters are displayed. To display all parameters of the hosted plugin, click `See all 12 params`.

The table provides the parameter index, parameter name, normalized parameter value, displayed parameter value, and the displayed parameter value label.

```
        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 %
```

Parameter Index

Parameter Name

Normalized Parameter Value

Displayed Parameter Value

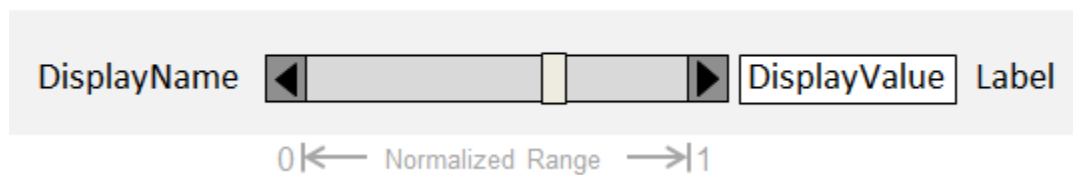Displayed Parameter Value Label

The *normalized parameter value* is always in the range [0,1] and generally corresponds to the position of a user interface (UI) widget in a DAW or the position of a MIDI control on a MIDI control surface. The *parameter display value* is related to the normalized parameter value by an unknown mapping internal to the plugin and typically reflects the value used internally by the plugin for processing.
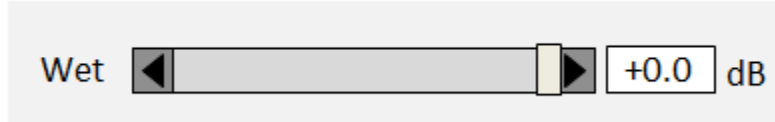
2. Set and Get Hosted Plugin Parameter Values

You can use `getParameter` and `setParameter` to interact with the parameters of the hosted plugin. Using `getParameter` and `setParameter` is the programmatic equivalent of moving widgets in a UI or controls on a MIDI control surface. A typical DAW UI provides the parameter name, a visual representation of the normalized parameter value, the displayed parameter value, and the displayed parameter value label.

Typical DAW User Interface

DisplayName ◀ ☐ ▶ DisplayValue Label

0 |← Normalized Range →|1

For example, the `Wet` parameter of `readelay-standalone.dll` has a normalized parameter value of `1` and a display parameter value of `+0.0`. The `Wet` parameter might be displayed in a DAW as follows:

With Audio Toolbox, you can use `getParameter` to return the normalized parameter value and additional information about a single hosted plugin parameter. You can specify which parameter to get by the parameter index.

```
parameterIndex = 1;
[normParamValue,paramInfo] = getParameter(hostedPlugin,parameterIndex)

normParamValue = 1

paramInfo = struct with fields:
     DisplayName: 'Wet'
    DisplayValue: '+0.0'
           Label: 'dB'
```
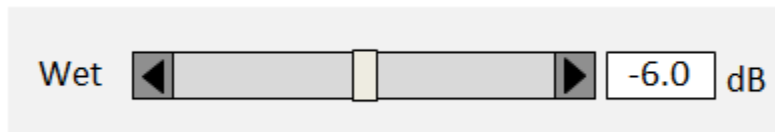
You can use `setParameter` to set a normalized parameter value of your hosted plugin. You can specify which parameter to set by its parameter index.

```
normParamValue = 0.5;
setParameter(hostedPlugin,parameterIndex,normParamValue)
```

Setting the normalized parameter value to 0.5 is equivalent to setting the indicator to the center of a slider in a DAW.

To verify the new normalized parameter value for `Wet`, use `getParameter`.

```
parameterIndex = 1;
[normParamValue,paramInfo] = getParameter(hostedPlugin,parameterIndex);
```

The `DisplayValue` for the `Wet` parameter updates from `+0.0` to `-6.0` because you set the corresponding normalized parameter value. The relationship between the displayed value and the normalized value is determined by an unknown mapping that is internal to the hosted plugin.

3. Use Hosted Plugin to Process Audio

To process an audio signal with the hosted plugin, use `process`.

```
audioIn = [1,1];
audioOut = process(hostedPlugin,audioIn);
```

Audio plugins are designed for variable-frame-based processing, meaning that you can call `process` with successive audio input frames of different lengths. The hosted plugin saves the internal states

required for continuous processing. To process an audio signal read from a file and then written to your audio output device, place your hosted plugin in an audio stream loop. Use `dsp.AudioFileReader` and `audioDeviceWriter` objects as the input and output to your audio stream loop, respectively. Set the sample rate of the hosted plugin to the sample rate of the audio file by using `setSampleRate`.

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

deviceWriter = audioDeviceWriter('SampleRate',sampleRate);
setSampleRate(hostedPlugin,sampleRate);

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

    % The hosted plugin requires a stereo input.
    stereoAudioIn = [audioIn,audioIn];

    x = process(hostedPlugin,stereoAudioIn);

    deviceWriter(x);
end

release(fileReader)
release(deviceWriter)
```

You can modify parameters in the audio stream loop. To control the `Wet` parameter of your plugin in an audio stream loop, create an `audioOscillator` System object™. Use the `fileReader`, `deviceWriter`, and `hostedPlugin` objects you created previously to process the audio.

```
osc = audioOscillator('sine', ...
    'Frequency',10, ...
    'Amplitude',0.5, ...
    'DCOffset',0.5, ...
    'SamplesPerFrame',fileReader.SamplesPerFrame, ...
    'SampleRate',sampleRate);

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

    controlSignal = osc();
    setParameter(hostedPlugin,1,controlSignal(1));

    stereoAudioIn = [audioIn,audioIn];
    x = process(hostedPlugin,stereoAudioIn);
    deviceWriter(x);
end

release(fileReader)
release(deviceWriter)
```

4. Analyze Hosted Plugin

You can use the Audio Toolbox measurement and visualization tools to display behavior information about your hosted plugin. To display the input and output of your hosted audio plugin, create a time scope. Create a `loudnessMeter` object and use the `'EBU Mode'` visualization to monitor loudness

output by the hosted plugin. Use the `fileReader`, `deviceWriter`, `osc`, and `hostedPlugin` objects you created previously to process the audio.

```
scope = timescope('SampleRate',sampleRate, ...
    'TimeSpanSource','property', ...
    'TimeSpanOverrunAction','scroll', ...
    'TimeSpan',5, ...
    'BufferLength',5*2*sampleRate, ...
    'YLimits',[-1 1]);

loudMtr = loudnessMeter('SampleRate',sampleRate);
visualize(loudMtr)

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

    controlSignal = osc();
    setParameter(hostedPlugin,1,controlSignal(1));

    stereoAudioIn = [audioIn,audioIn];
    x = process(hostedPlugin,stereoAudioIn);

    loudMtr(x);
    scope([x(:,1),audioIn(:,1)])

    deviceWriter(x);
end
```
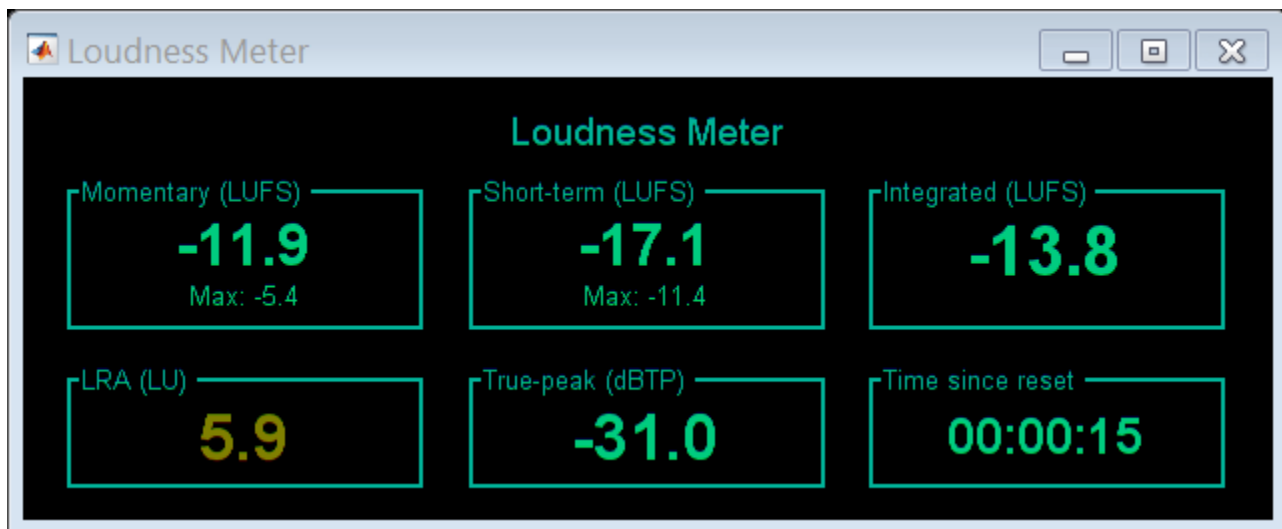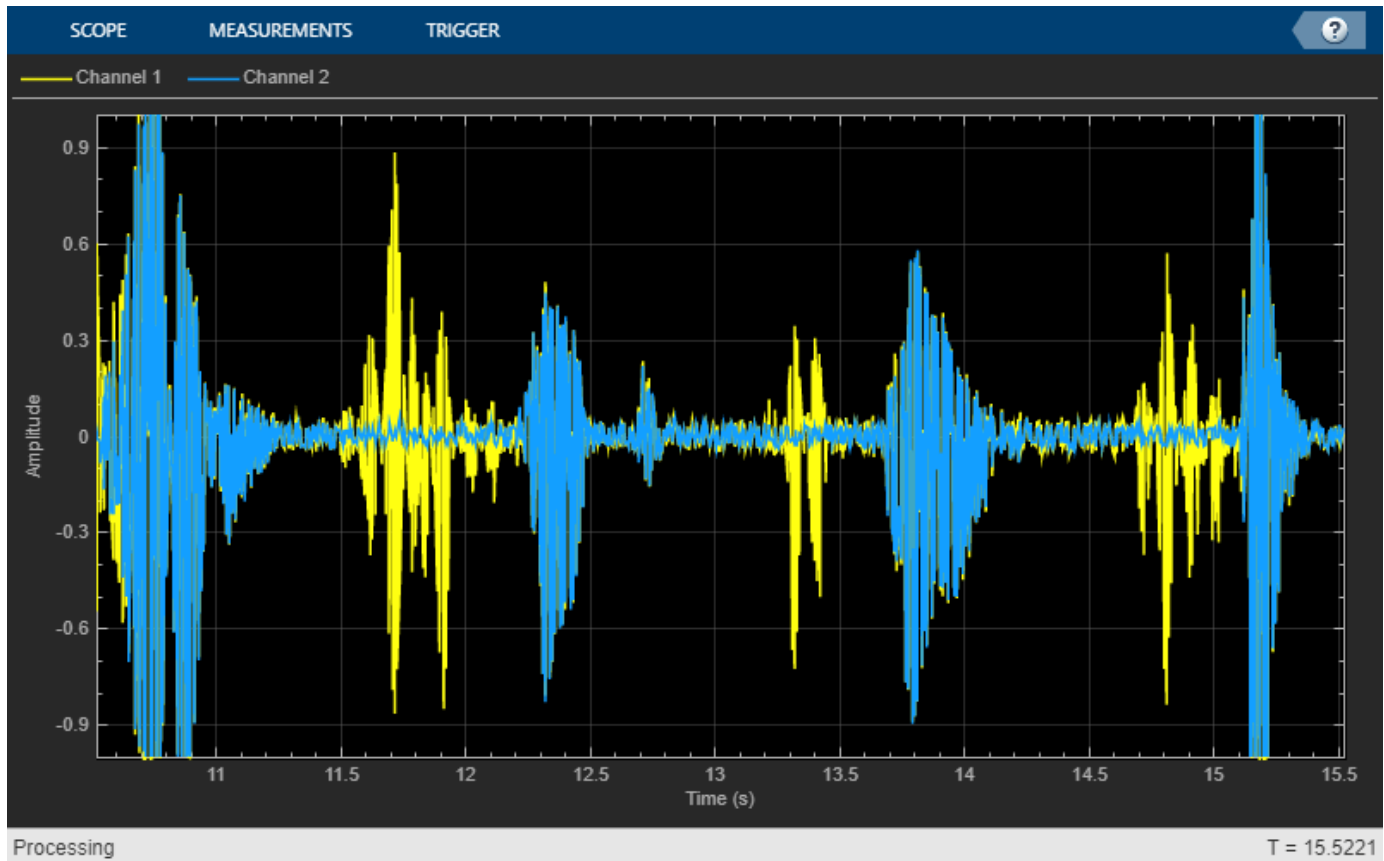
```
release(fileReader)
release(deviceWriter)
```

Graphical Interaction

You can also interact with an externally authored audio plugin graphically using the **Audio Test Bench**. The **Audio Test Bench** mimics the default graphical user interface common to most digital audio workstations.

Heuristic Mapping

Investigate Parameter/Property Mapping

Parameter display values are related to normalized parameter values by unknown mapping rules internal to the plugin. You can investigate the relationship between the normalized parameter values and the displayed values by creating a sweeping function. You can use the sweeping function to map parameter values to their displayed output.

The properties display mode of hosted plugins uses a similar approach to enable you to interact directly with the real-world (displayed) values, instead of the normalized parameter values.

Save the `displayParameterMapping` function in your current folder. This function performs a simplified version of the parameter sweeping used to create the property display mode for hosted plugins.

```matlab
function displayParameterMapping(hPlugin,prmIndx)
x = 0:0.001:1; % Normalized parameter range

[~,prmInfo] = getParameter(hPlugin,prmIndx);
if isnan(str2double(prmInfo.DisplayValue))
    % Non-Numeric Displays - prints normalized parameter range associated
    % with string
    setParameter(hPlugin,prmIndx,0);
    [~,prmInfo] = getParameter(hPlugin,prmIndx);
    txtOld = prmInfo.DisplayValue;
    oldIndx = 1;

    for i = 2:numel(x)
        setParameter(hPlugin,prmIndx,x(i))
        [~,prmInfo] = getParameter(hPlugin,prmIndx);
        txtNew = prmInfo.DisplayValue;
        if ~strcmp(txtNew,txtOld)
            fprintf('%s: %g - %g\n',txtOld, x(oldIndx),x(i-1));
            oldIndx = i;
            txtOld = txtNew;
        end
    end
    fprintf('%s: %g - %g\n',txtOld, x(oldIndx),x(i));
else
    % Numeric Displays - plots normalized parameter value against displayed
    % parameter value
    y = zeros(numel(x),1);
    for i = 1:numel(x)
        setParameter(hPlugin,prmIndx,x(i))
        [~,prmInfo] = getParameter(hPlugin,prmIndx);
        y(i) = str2double(prmInfo.DisplayValue);
    end
    if any(isnan(y))
        warning('NaN detected in numeric display.')
    end
    plot(x,y)
    xlabel('Normalized Parameter Value')
    ylabel(['Displayed Parameter Value (',prmInfo.Label,')'])
    title(prmInfo.DisplayName)
end

end
```
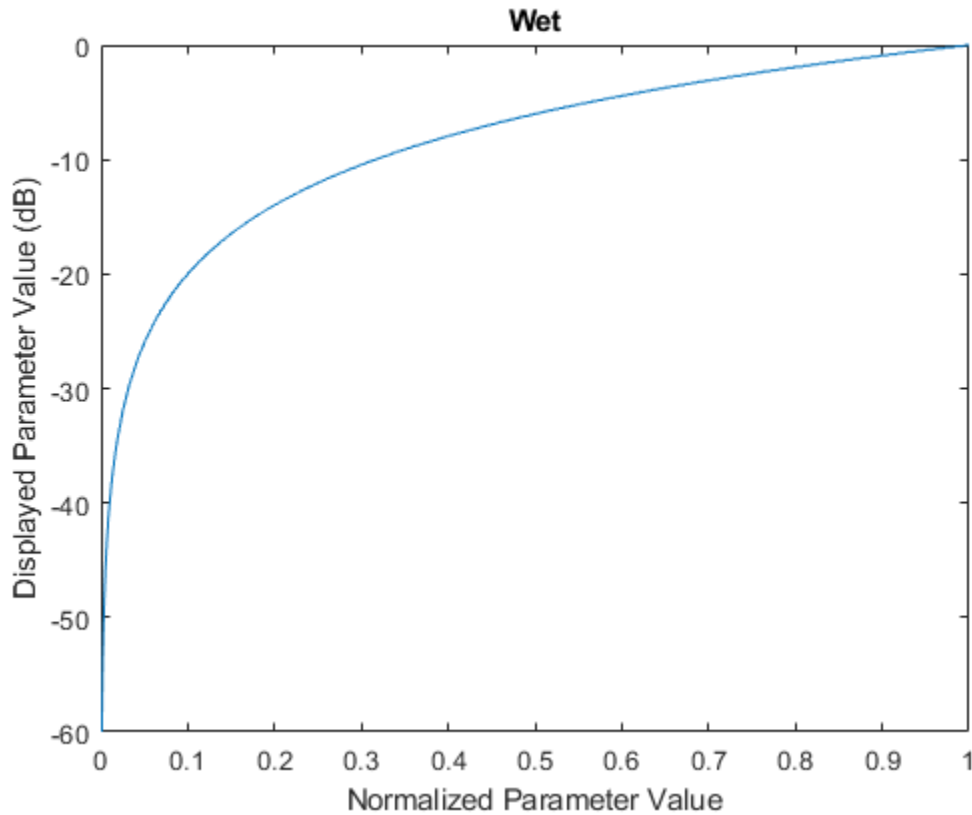
Load the `readelay-standalone.dll` plugin into MATLAB®. Call the `displayParameterMapping` function with the hosted plugin and a parameter index.

```matlab
hostedPlugin = loadAudioPlugin('readelay-standalone.dll');
displayParameterMapping(hostedPlugin,1);
```

If you use the `displayParameterMapping` function with a nonnumeric parameter, the relationship displays in the Command Window:

```
displayParameterMapping(hostedPlugin,3)
```

```
OFF: 0 - 0.499
ON: 0.5 - 1
```

See Also

Functions
loadAudioPlugin

Classes
externalAudioPlugin | externalAudioPluginSource

More About

• "What Are DAWs, Audio Plugins, and MIDI Controllers?" on page 9-2
• "Audio Plugins in MATLAB" on page 11-2